Detection and Operational Cryptanalysis of Weakly Implemented, Weak or Trapped Encryption Systems A Step-by-Step Tutorial - Part II

Eric Filiol, filiol@esiea.fr

ESIEA - Laval
Operational Cryptology and Virology Lab $(C+V)^O$
http://www.esiea-recherche.eu/

H2HC 2010 - Sao Paulo & Cancun

Outline

Introduction : what the next step ?

- We have seen (part I of the tutorial) how
  - weakly implemented
  - weak (at the mathematical design level)
  - or trapped encryption systems
- can be detected and cryptanalyzed !
- What about practical cases ?
- Is it possible to detect this only during limited period of time ?
  - The cryptographic design looks secure on the paper only !
  - Concept of Dynamic Encryption Trapdoor
- What can be the impact of the overall computer security on the cryptographic security ?
- Presentation of the Open source cryptanalysis library Mediggo - Practice.

Introduction

- Without loss of generality, the examples and real cases presented here have been simplified for sake of clarity and to fit to the limited duration of the tutorial
  - Realistic cases involve more mathematics that you are ready to accept (and you really need).
  - Cases coming from satellite communications, encrypted malware, encryption software...
- Everything presented here is
  - Either inspired by real cases during the last 60 years
  - Or are the results of current research in our lab.

Aim of Part II

- Learn on practical cryptanalysis
- Be able to detect any weak encrypted trafficor files.
- Be able to break it without effort.
- Present the Megiddo cryptanalysis library
- Practice !

Summary of the talk - Part II

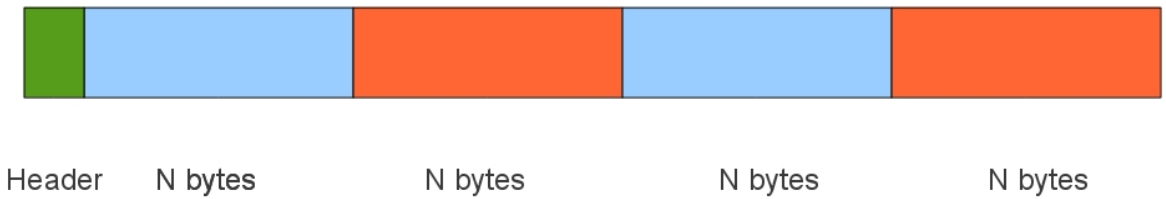
Outline

Introduction

- Despite many encryption systems are public, still many products (hardware and/or software) embed
  - Public algorithms but weakly (intendly or not) implemented
  - Proprietary algorithms which are either weak in their design and/or have implementation trapdoors.
- The issue is : how to detect this situation without performing time-consuming, illegal reverse-engineering ?
- Without loss of generality we will focus on stream ciphers
  - Still widely used (satellite communications, telecommunications, governmental use, encryption of binaries...).
  - Illustrating with block ciphers would require more mathematics !

Example I : Malware Encryption

- Drawn from a real case.
- The malware author uses a N-byte truly random sequence repeatedly combined to the binary to protect it (Vernam cipher) where is itself random in the range $[64, 256]$ (in bytes)

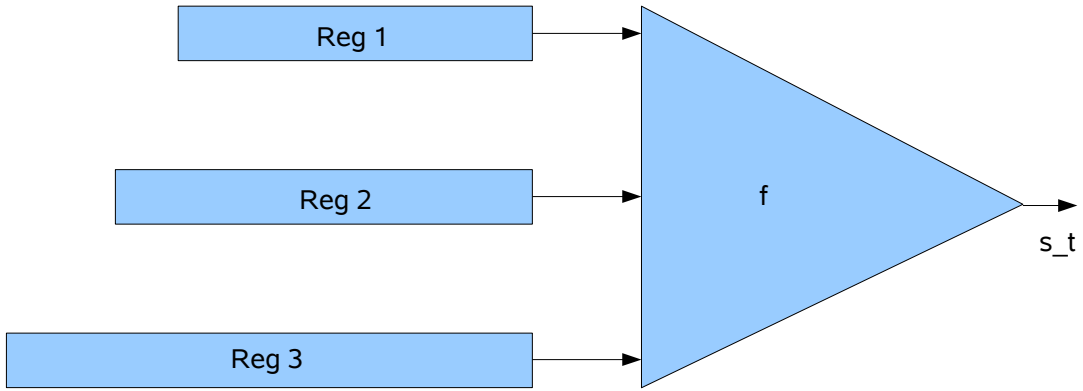


Example I : Malware Encryption (2)

- Trying all the N-subsequence is impossible (about 2^{256} possibilities).
- The solution is :
  1. Try all possible lengths N of code chunks (linear complexity in the size of the binary code).
  2. For each value of N, split the code into N-byte chunks.
  3. Compute the coincidence indices between chunks of code.
- For the correct value of N, we have a statistical pick and the different chunks of code behave like parallel encrypted texts.
- Decryption is then easy.
- See detection in the pratice part of this tutorial.

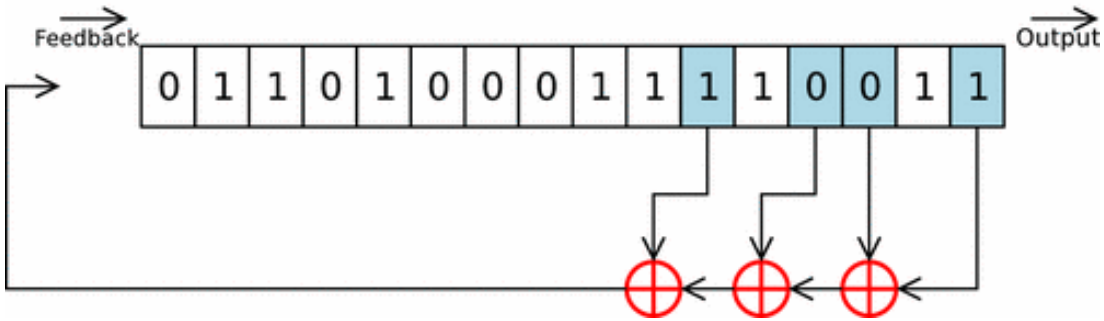
Stream Cipher Generic Scheme

Let us consider a generic stream cipher scheme (most of the existing stream ciphers can be more or less transformed in this generic scheme).



- Two critical components (in which to hide trapdoors) :
  - The Linear Feedback Shift Registers (LFSR).
  - The Boolean combining function.

LFSR Trapdoor



- Used to produce statistically good random sequence of bits.
- Use of a characterictic polynomial $P(x)$ to compute the feedback.
  - $P(x) = x^{16} \oplus x^5 \oplus x^3 \oplus x^2 \oplus 1$ in the example.
- The degree of the polynomial is also the length of the LFSR.
- A LFSR generates periodic sequence by naturei (must start with the non-zero state).
- In a cryptographic context, the key is the initial state.

LFSR Trapdoor (2)

- If the characterictic polynomial $P(x)$ is primitive then the sequence produced is ultimately periodic and has length $2^n - 1$.
  - $P(x) = x^3 \oplus x \oplus 1$ is primitive.
- Otherwise the period length is (far) less than $2^n - 1$.
  - $P(x) = x^3 \oplus x^2 \oplus x \oplus 1$ is not primitive.
- Another constraint : all LFSR lengths must be co-prime (relatively prime).
- Whenever those two properties are not fullfilled (primitive AND co-prime polynomials) then the system can have shorter output sequences.
  - The system comes back to the initial state quicker than expected.

LFSR Trapdoor (3)

- First trap : use non primitive polynomial.
  - The LFSR will produce short periodic cycles.
- Second trap : use non co-prime LFSR length.
- Combining the two traps is better.
- Just calibrate things in such a way that there exist short cycles for every LFSR (since polynomials are not primitive) whose respective length is not co-prime.
  - The system will produce short periodic cycles.
- The system will produce parallel encrypted texts with a frequency you can play on.

Boolean Function Trapdoor

- Here we use the fact that any encryption system use
  - A base key (changed every day, week, month...) K.
  - A message key (changed for every encrypted text) K_m.
- Example drawn from a real case during the 80-90s.
- A Boolean function f is defined on $\mathbb{F}_2^n$ and hence has 2^n variables.
  - $f(x) = \sum_{u \in \mathbb{F}_2^n} a_u x^u$ where $a_u \in \mathbb{F}_2$ and $x \in \mathbb{F}_2^n$.
- How to trap the Boolean function ?
  1. Use a message key K_m of size 2^{n-1}
  2. Xor it by half to the Boolean function truth table $([0, 2^{n-1} - 1], [2^{n-1}, 2^n])$.
- Produce partial parallel encrypted texts (according to a K_m-dependent decimation of the ciphertext).

Outline

Introduction

- Here we consider a strong cryptosystem (AES, TrueCrypt, GPG/PGP...).
- However the security at the operating level is not perfect.
- What is it possible to do with a simple malware ?
  - Of course it can eavesdrop/wiretap the key and send it outside.
- What about computers with no network connection or whenever key wiretapping is no longer possible ?
- The solution is
  - Modify the cryptographic environment on-the-fly.
  - Modify the crypto-system on-the-fly in memory only.
- The "static (mathematical) security" remains unquestioned !
- Just create dynamically periods of time during which the encryption system is weak.

Program Interaction Control

- Here we exploit the fact that very often, the message key K_m is built from data provided by external programs.
  - Message counter, message key, session key...
  - Initialization vectors for block ciphers.
  - Integer nonces.
- Most of the time the resources involved are in the Windows API.
  - They provide random data required by the encryption application to generate message keys and IVs
- You then just have to hook the API function involved.
- Same approach for other equivalent resources (key infrastructure, network-based key management...).

Hooking the CryptGenRandom function

- Drawn from a real case.
- A malicious DLL is injected in some (suitable) processes. This DLL hooks the CryptGenRandom function (included in Microsoft's Cryptographic Application Programming Interface).

CryptGenRandom function

```
BOOL WINAPI CryptGenRandom(
__in HCRYPTPROV hProv,
__in DWORD dwLen,
__inout BYTE *pbBuffer
);
```

- A timing function checks whether we are in the time window given as parameter $sTime(12,00,14,00)[...]$. will hook the CryptGenRandom function between noon and 2pm only.

Hooking the `CryptGenRandom` function (2)

- The integer (random data) returned by `CryptGenRandom` is modified by the function `HookedCryptGenRandom`.
  - They provide random data required by the encryption application to generate message keys and IVs
- You then just have to hook the API function involved.
- Same approach for other equivalent resources (key infrastructure, network-based key management...).

Hooking the CryptGenRandom function (3)

Generate fixed message key $0x1212121212121212$

HookedCryptGenRandom function

```
BOOL WINAPI HookedCryptGenRandom(HCRYPTPROV hProv, DWORD
dwLen, BYTE *pbBuffer)

static BOOL send12 = 0; BOOL isOK; DWORD i;
send12 = 1;
isOK = HookFreeCryptGenRandom(hProv, dwLen, pbBuffer);
if((send12) && (isOK))
for(i = 0; i < dwLen; i++) pbBuffer[i] = 0x12;
return isOK;
```

Let us have a look into the code...

Memory Attack Only

- The idea here consists in scanning for active encryption system in memory and modifying their mathematical design on-the-fly only.
- Volatile modification which does not affect the application on the disk.
- Our Implementation to attack AES
  - `scanKernelModules` function to look for AES' sboxes signature.
  - `patchModule` function to modify (weaken) those Sboxes.
- Let us have a look into the code...
- You can do many other things
  - Switch mode of operation (e.g. CBC to OFB).
  - Modify internal message key or IV generation
  - ... no limit but your imagination !

Outline

Introduction : The Megiddo Library

Megiddo

- Open source cryptanalysis library in C
- At the present time
  - Detection and cryptanalysis of weakly implemented or trapped systems
- To come
  - Automatic detection of statistical biases in cryptographic algorithms.
  - Specific cryptanalysis tools.
- More to come later...
- Source code and samples available on `http://code.google.com/p/mediggo/`

Detection step

- What the issue ?
  - Among thousands of encrypted texts, how to detect the weak subsets (parallel ciphertexts) ?
  - As for a single encrypted file how to detect the existence of parallel parts ?
- As a general principle, compute the coincidence indices
- For the first problem, use file `detect.c`
  - *./detect <ciphertext_dirname> <outputfile>*
  - Apply the equivalence relationship to find subsets.

Detection step (2)

Solution for Ciphertexts2 directory

```
cry35.txt - cry34.txt - Coincidence Index = 0.6760
cry35.txt - cry33.txt - Coincidence Index = 0.6667
cry35.txt - cry32.txt - Coincidence Index = 0.6711
cry35.txt - cry31.txt - Coincidence Index = 0.6755
cry34.txt - cry33.txt - Coincidence Index = 0.6762
cry34.txt - cry32.txt - Coincidence Index = 0.6700
cry34.txt - cry31.txt - Coincidence Index = 0.6738
cry33.txt - cry32.txt - Coincidence Index = 0.6780
cry33.txt - cry31.txt - Coincidence Index = 0.6811
cry32.txt - cry31.txt - Coincidence Index = 0.6713
```

Here ciphertexts `cry31.txt`, `cry32.txt`, `cry33.txt`, `cry34.txt`, `cry35.txt` define a parallel subset.

Detection step : single encrypted file

- To solve the second problem, use file `detect_singlefile.c`
  - *./detect_singlefile <ciphertext_file>*
- If the size of the chunks is $N = n$ (refer to slide *Example I : Malware Encryption*) then you also get a statistical peak for values $n, 2n, 3n....$

Solution for cryptfile1

```
n = 134 - Coincidence Index = 0.5236
n = 268 - Coincidence Index = 0.5229
n = 402 - Coincidence Index = 0.5221
n = 536 - Coincidence Index = 0.5197
....
n = 2010 - Coincidence Index = 0.5152
n = 2144 - Coincidence Index = 0.5154
n = 2680 - Coincidence Index = 0.5152
```

- Here the solution is $n = 134$ (bytes).

Build a corpus

- The aim is to have a statistical model of the plaintext language (at the Chomsky's sense).
- Hence the approach is the same both natural languages (class 1) and programming languages (class 2).
- Extendable to any other class of grammar/language.
- Use file `create_corpus.c`
  - *./create_corpus <ref_text_dirname> <corpus output file>*
- Optimal values : 4-grams over a 96-character alphabet
- Sample corpus provided in the library covers most of the Western languages.

Cryptanalysis step

- On each weak encrypted texts subset, we launch the cryptanalysis
- Use file `decrypt_para.c`
  - *./decrypt_para <corpus> <sequence_file> <crypto1> <crypto 2>...*
- You obtain the pseudo-running sequence. You must use it now to decipher each ciphertext :
  - Use file `decipher.c`
  - *decipher ciphertext_file pseudo-random sequence_file plaintext_file*

More stuff

- Utility `texte_extract.c`
  - Extract encrypted data in MS Word and MS Excel document (up to Office 2003)
- Then you can proceed as previously
- You will find the technical paper and a few other slides (including the present ones) in the archive.

Outline

Conclusion

- Cryptographic strength and security cannot be defined in a static way way only
- The implementation and the way of use are critical parts of that security.
- Environmental security can reduce the cryptographic security dramatically.
- Dynamic, time-limited (or time-dependant) are likely to be the future of cryptographic attacks...
- ... if it is not already the case.
- Enjoy cryptanalysis and stay tuned to further developments in Megiddo

Questions

- Many thanks for your attention.
- Questions ...
- ... and Answers.