Windows Shellcode Mastery (reloaded)

iAWACS 2009

Benjamin CAILLAT

ESIEA - SI&S lab

caillat[at]esiea[dot]fr

bcaillat[at]security-labs[dot]org

Plan

Plan

The PE format (1)

- Under Windows, executables are in the PE format (Portable Executable)
- Executables compounded of a header, a section table and several sections (code, data, resources...)

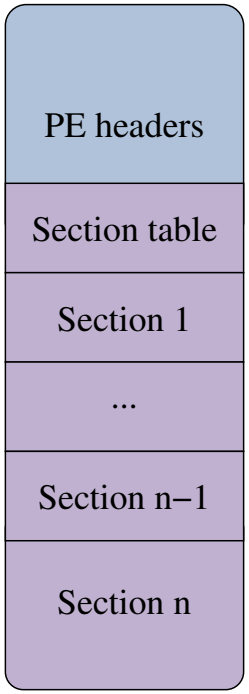


Figure: General structure of an executable

The PE format (2)

Headers

contain metadata used by Windows to load executable : prefered load address, address of entry point, ...

Table of sections

array of structures, each representing one section (name, mapping address, characteristics, ...)
characteristics : rights (RWX), initialised or not, shared, ...

Sections

- two types : code (RX) and data (R or RW)
- data section :
  - data of the program (strings, global variables, ...)
  - metadata : importation and exportation tables

Imported function resolution in Windows

- an executable generally uses/"imports" functions "exported" by a shared library
- importation: by name or by ordinal (index in exportation table)
- two mechanisms to resolve imported functions:
  - when process is created
  - during execution ("dynamic address resolution")

Resolution when process is created (1)

- resolution is done by Windows loader
- PE file contains an "import table": names of every imported dll/function
- Windows loader reads table and fills another table: the IAT (Import Address Table)
- calls to imported functions are done through the IAT

Resolution when process is created (2)

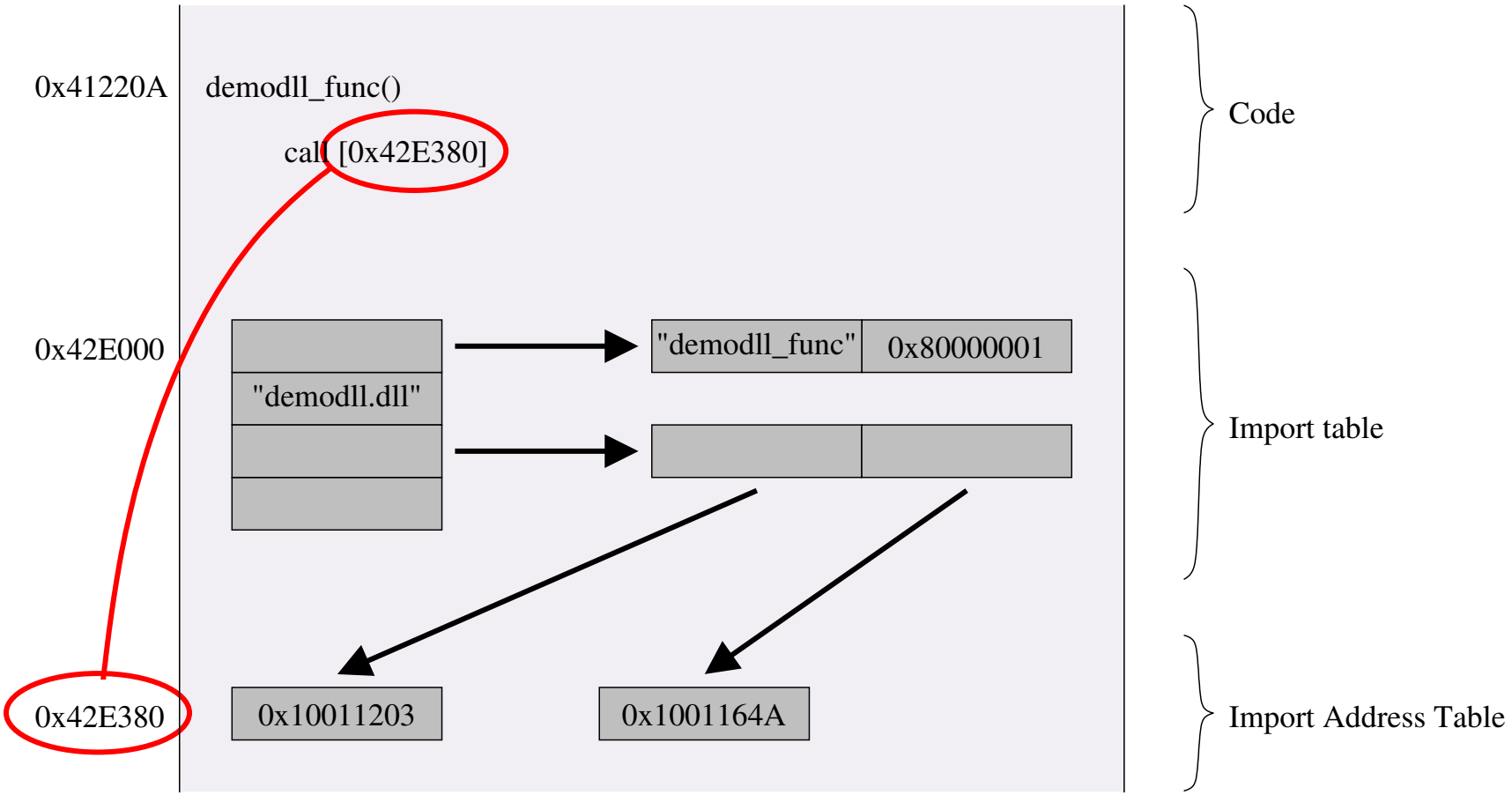


Figure: Calling an imported function

Resolution during execution

- Resolution is done by the code by using two functions:
  - "LoadLibrary": load a library
  - "GetProcAddress": find an exported function by its name/ordinal
- Result of "GetProcAddress" stored in a function pointer

The PE format

Demo : Example of “DemoPE”

Notion of shellcode

Definition (general)

Set of binary data that has the following properties:

- executes some specific operations if execution is transfered to its firts bytes (in general)
- can run in any process at any address:
  - must not contains any hardcoded address
  - must be autonomous and not use external references

Plan

Plan

The use of shellcodes in virology

—

A few techniques used by malicious code . . .

Context definition

- Generally, malicious codes try to do several things:
  - stay undetected by antiviruses
  - propagate to other hosts or executables
  - execute their malicious actions (e.g. capture some private user data, open a backdoor on the system ...)
- Use special techniques, not always easy to implement
- Let us illustrate this with a few specific techniques and try to see how they can be implemented in an executable

Encryption of malicious code - Principle

Description

Malicious code is made up of two parts:

- the real malicious payload which is encrypted
- a decryption part

Encryption of malicious code - Principle

Description

Malicious code is made up of two parts:

- the real malicious payload which is encrypted
- a decryption part

Objective

- Protect malicious payload against an analysis
- Could be an automatic analysis (antivirus) or a manual analysis (disassembling code)

Encryption - protection against automatic analysis

- Malicious code is scanned by a tool that works with signature identification
- Each copy of malicious code must be different:
  - decryption part is transformed through metamorphism
  - encryption key is changed in each copy ⇒ malicious payload is different (polymorphism)

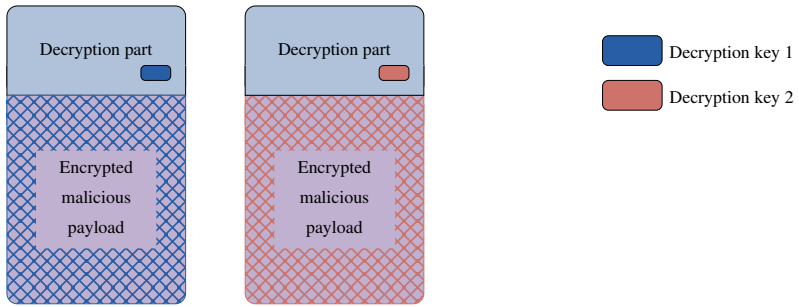


Figure: Two copies of the same virus that implements polymorphism

- Notes:
  - Decryption key may be stored in decryption part
  - Simple encryption algorithm like a XOR with 32-bits key may be used

Encryption - protection against manual analysis

- Aim: if malicious payload is intercepted during introduction on targeted system, it cannot be disassembled and analysed manually
- Little differences with previous encryption:
  - strong encryption algorithm like AES must be used
  - decryption key must not be stored in decryption part

Principle of execution of encrypted malware

Figure: Principle of execution of an encrypted malware

Principle of execution of encrypted malware

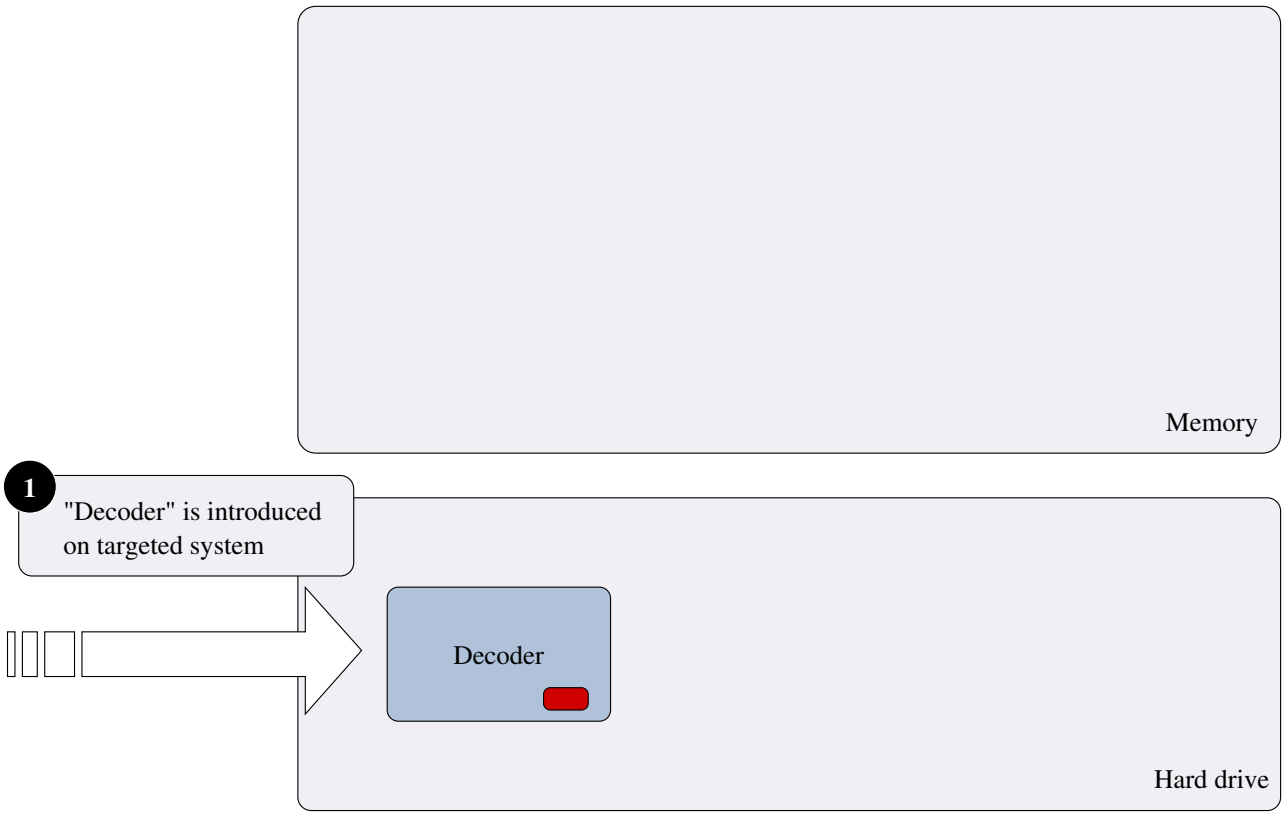


Figure: Principle of execution of an encrypted malware

Principle of execution of encrypted malware

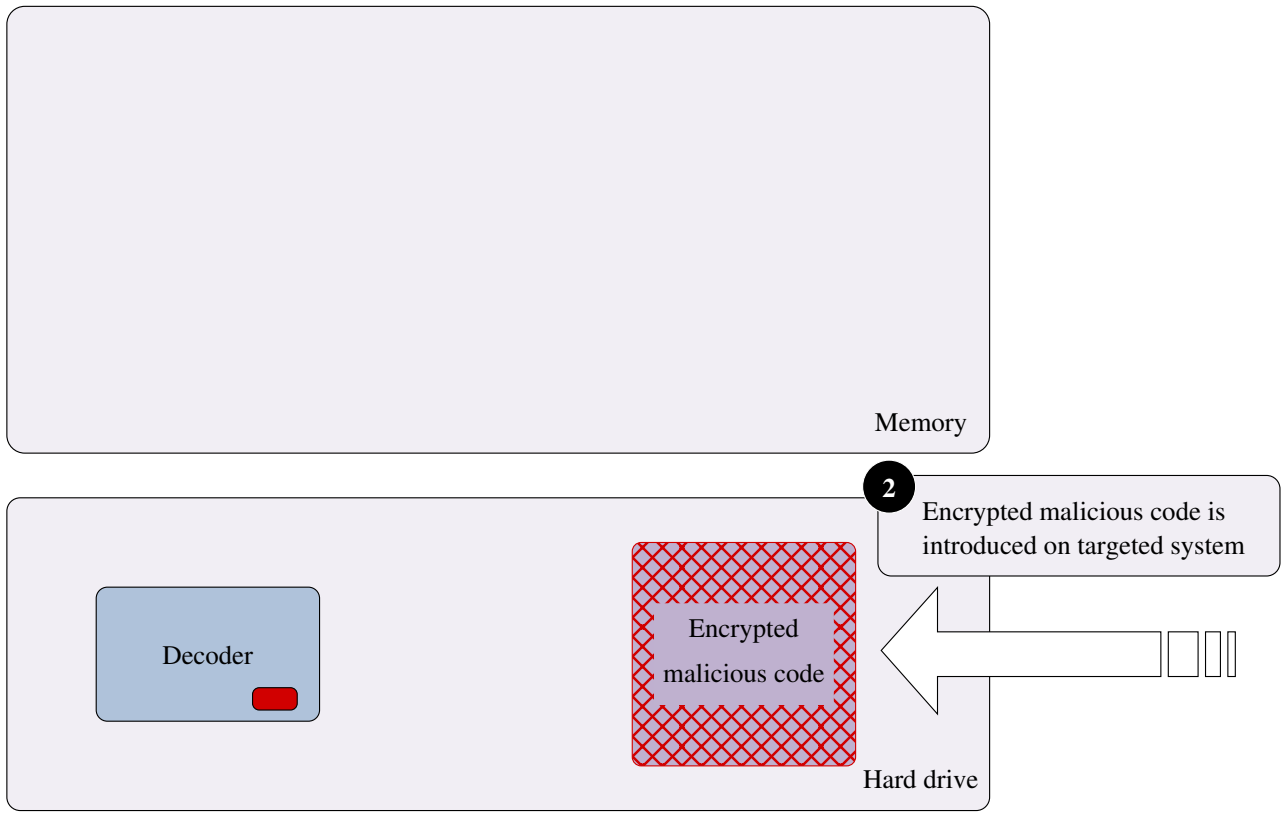


Figure: Principle of execution of an encrypted malware

Principle of execution of encrypted malware

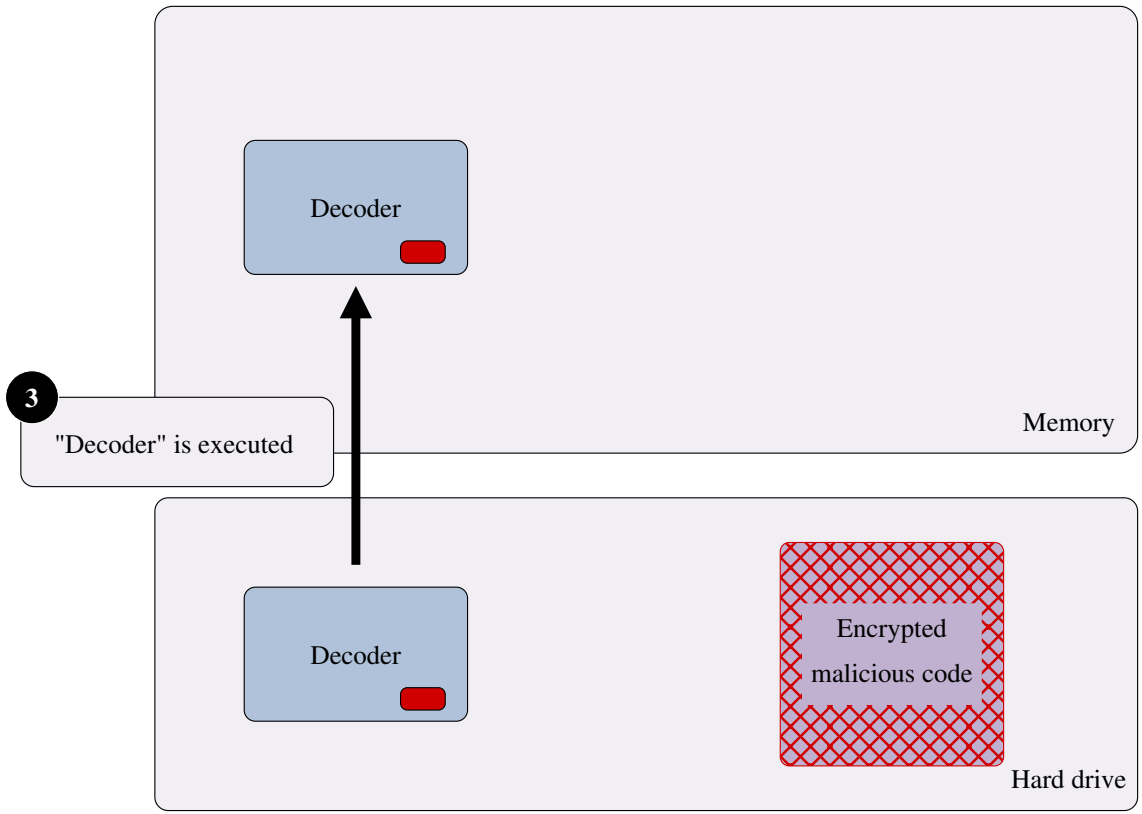


Figure: Principle of execution of an encrypted malware

Principle of execution of encrypted malware

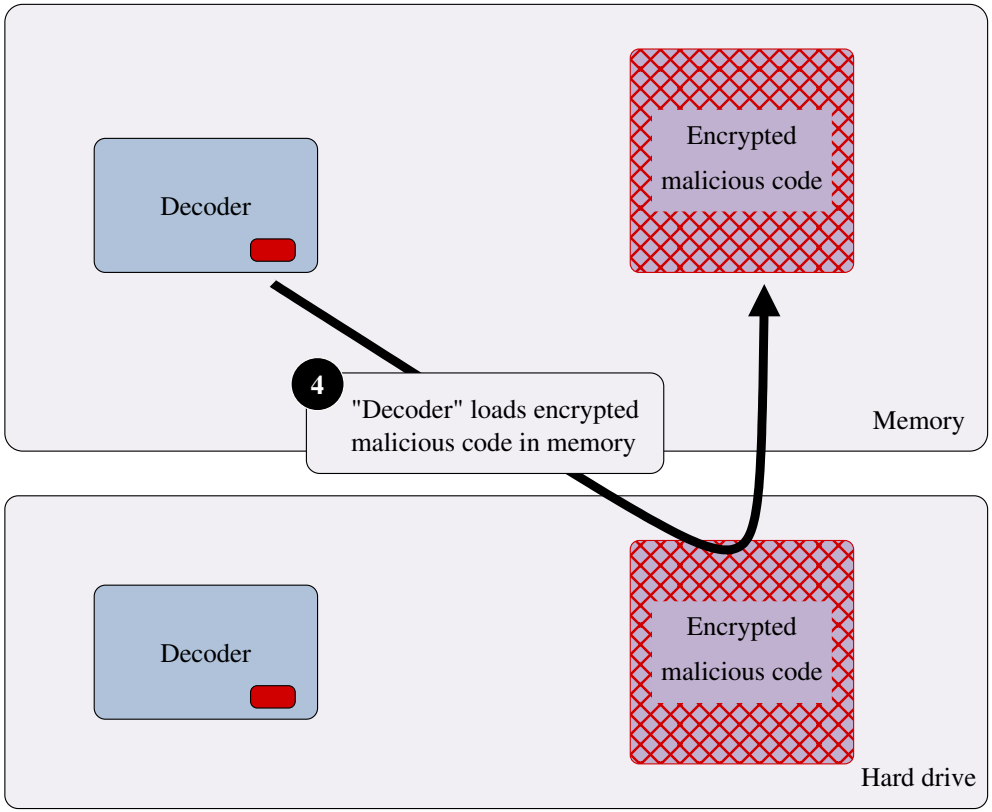


Figure: Principle of execution of an encrypted malware

Principle of execution of encrypted malware

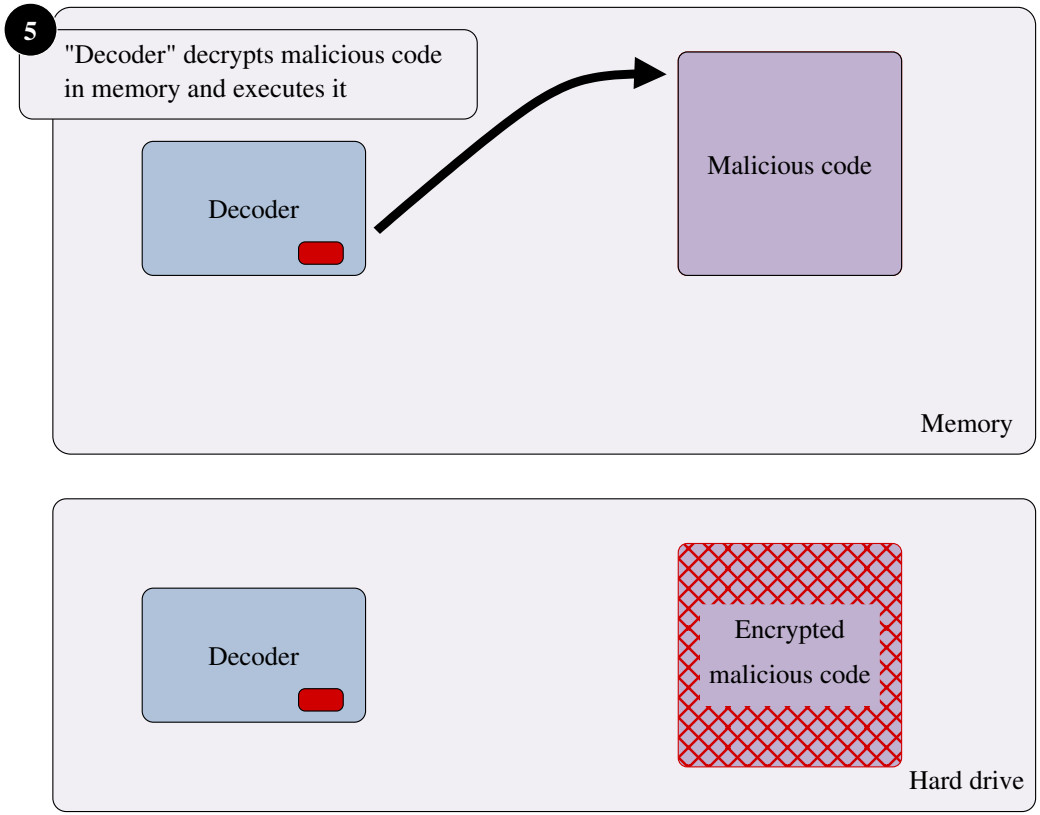


Figure: Principle of execution of an encrypted malware

Encryption - protection against manual analysis

- Of course, several ways to get malicious payload on infected computer (dump the memory, extract encryption key and decrypt malicious payload)
- But malicious payload is protected during introduction onto targeted computer:
  - two parts are introduced in different ways at different times
  - if **one** introduction fails, we will intercept:
    - decryption part: totally generic
    - malicious payload: encrypted

  $\Rightarrow$cannot get any information on the attack

Encryption of malicious code - Implementation

- Encryption of each part of malicious payload in executable not a good solution:
  - complicated: all binary data characteristics of the malicious payload must be encrypted (functions, initialised data and strings)
  - not efficient: PE metadata cannot be encrypted
- Better solution: encrypt the whole executable $\sim$ a packer
  But developing such a tool required some work

Execute only in memory - Principle

Description

Malicious code is able to execute without being copied on hard drive

Execute only in memory - Principle

Description

Malicious code is able to execute without being copied on hard drive

Objective

- Cannot be detected by local antivirus
- Leaves few traces on targeted system
  $\Rightarrow$ complicates an eventual forensic analysis

Principle of execution of malware only in memory

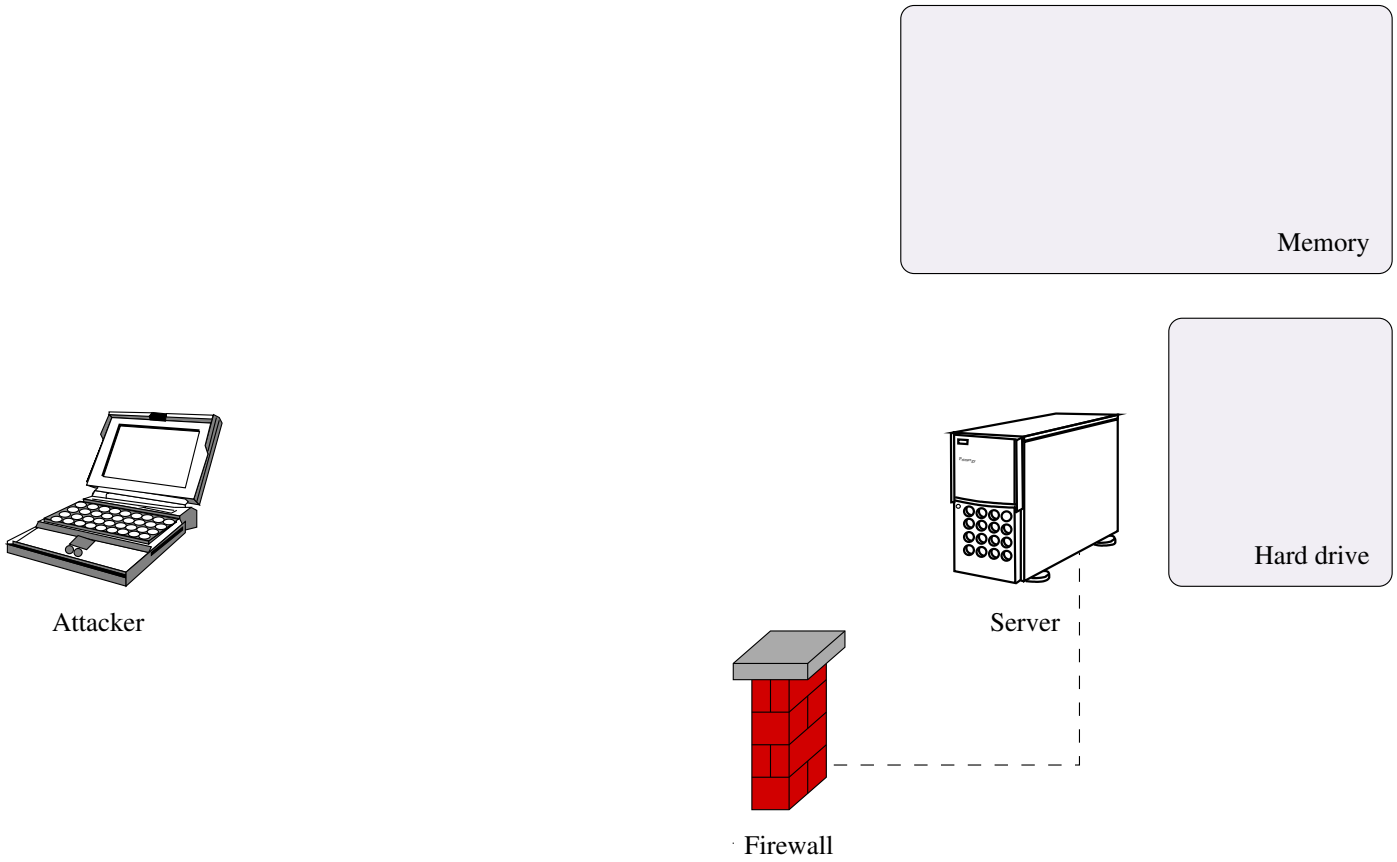


Figure: Principle of execution of malware only in memory

Principle of execution of malware only in memory

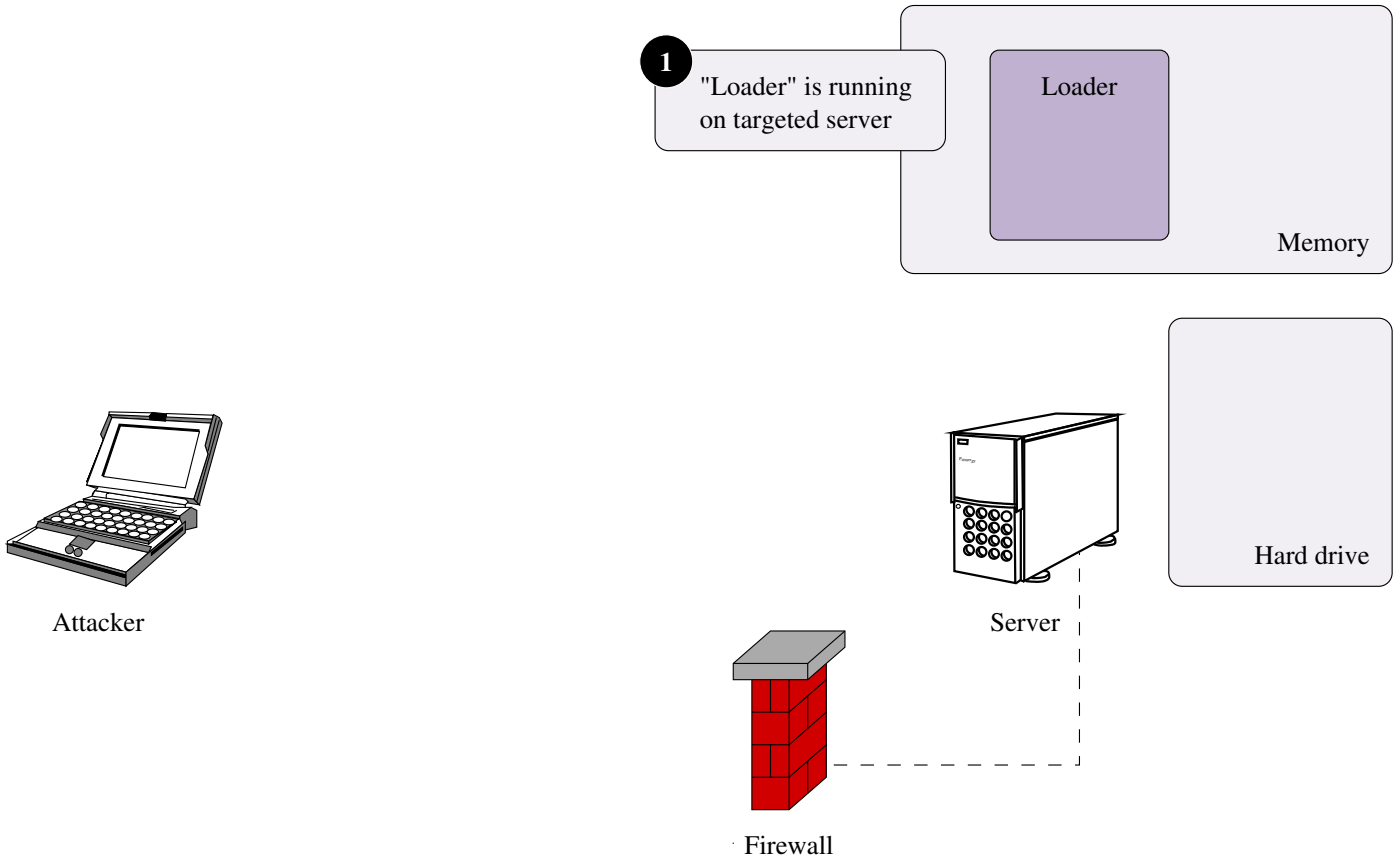


Figure: Principle of execution of malware only in memory

Principle of execution of malware only in memory

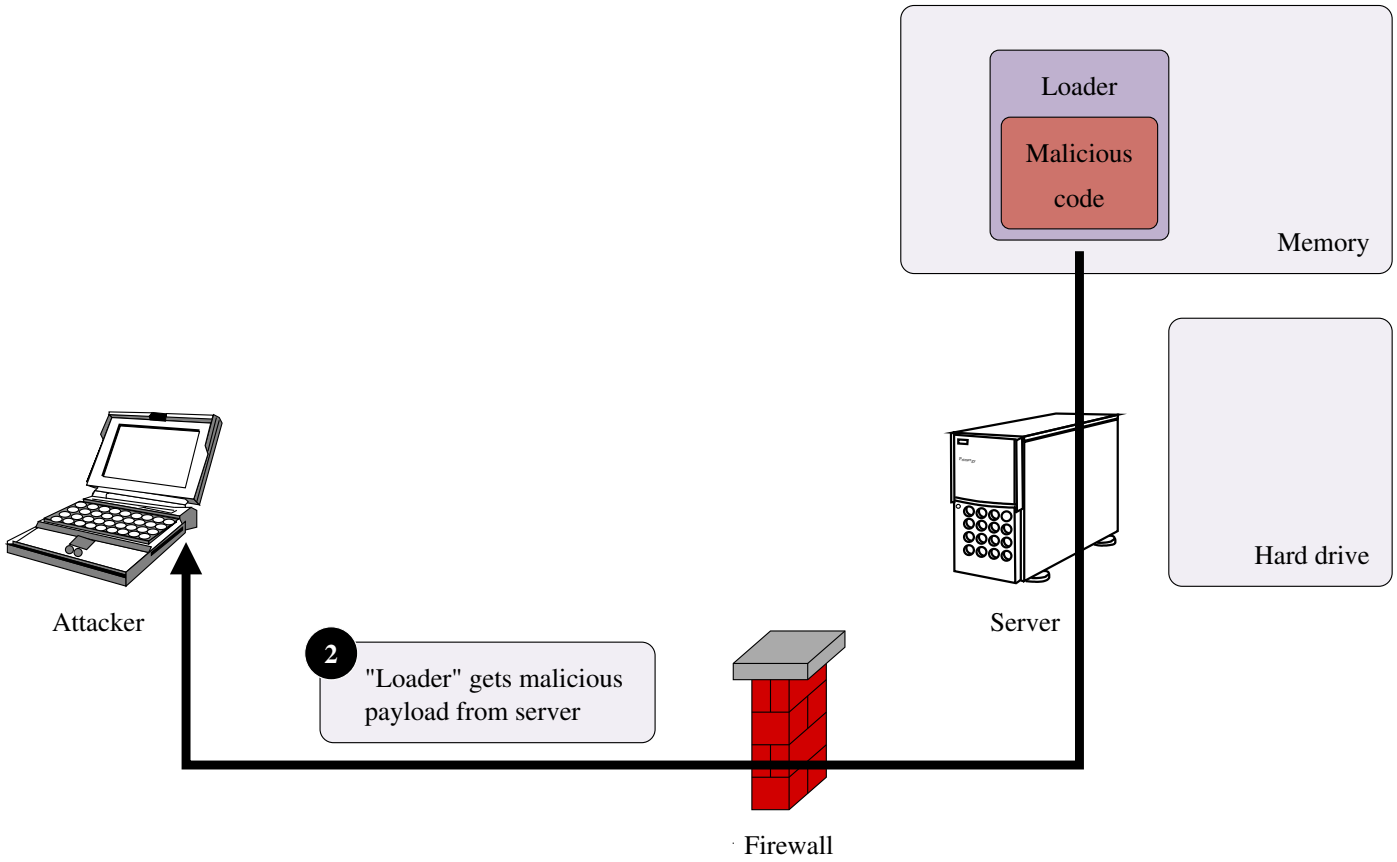


Figure: Principle of execution of malware only in memory

Principle of execution of malware only in memory

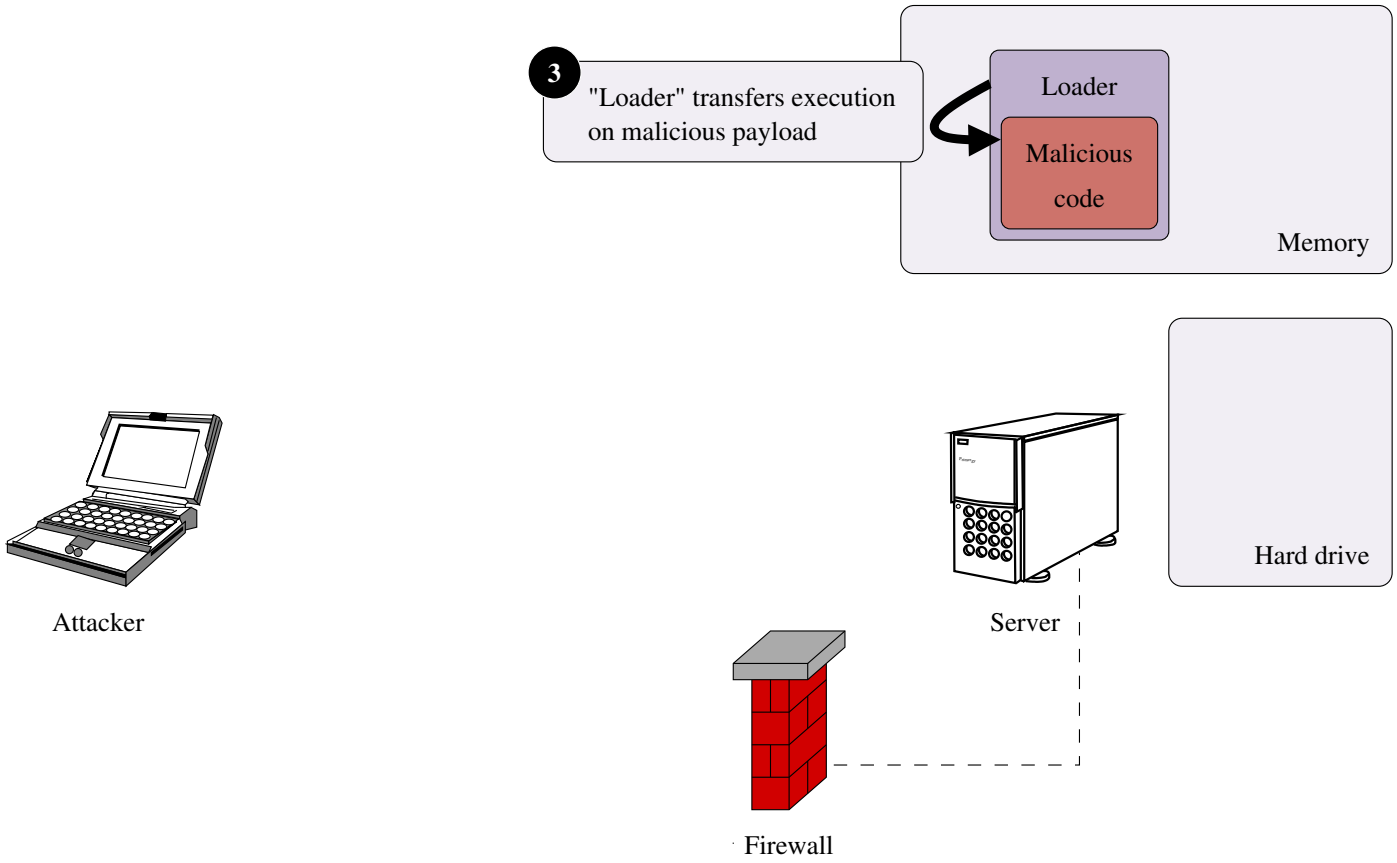


Figure: Principle of execution of malware only in memory

Execute only in memory - Implementation (1)

- Copying executable in memory and jumping on entry point does not work:
  - sections must be mapped at the right address
  - imported functions must be resolved
- A few tricks can be used:
  - use "pragma" directives to group all functions/data in one section
  - play with "preferred load address" so that section is mapped in a memory space "normally" free in process
  - use dynamic address resolution

$\Rightarrow$ Possible. . . but rather tedious

Execute only in memory - Implementation (2)

Demo : Example of “DemoPragma”

Infect an executable - Principle

Description

- Malicious payload is added into another executable
- Execution flow of infected executable is modified to execute malicious payload

Infect an executable - Principle

Description

- Malicious payload is added into another executable
- Execution flow of infected executable is modified to execute malicious payload

Objective

Create a Trojan horse; behaviour of the program must not be disrupted

Infect an executable - Implementation

- Malicious payload added at the end of the executable, after last section
- Several ways to redirect execution flow:
  - patch the executable entry point
  - patch some instructions that will probably be executed
    Example: call to the function "save" in a text editor

Infect an executable - Implementation

- Malicious payload added at the end of the executable, after last section
- Several ways to redirect execution flow:
  - patch the executable entry point
  - patch some instructions that will probably be executed
    Example: call to the function "save" in a text editor
- Each solution has pros and cons:
  - Patching instruction requires manual analysis to find a suitable instruction to patch
  - But execution of malicious code requires action of the user
    $\Rightarrow$ neither executed, nor analysed by an antivirus, even with code emulation

Infect an executable - Implementation

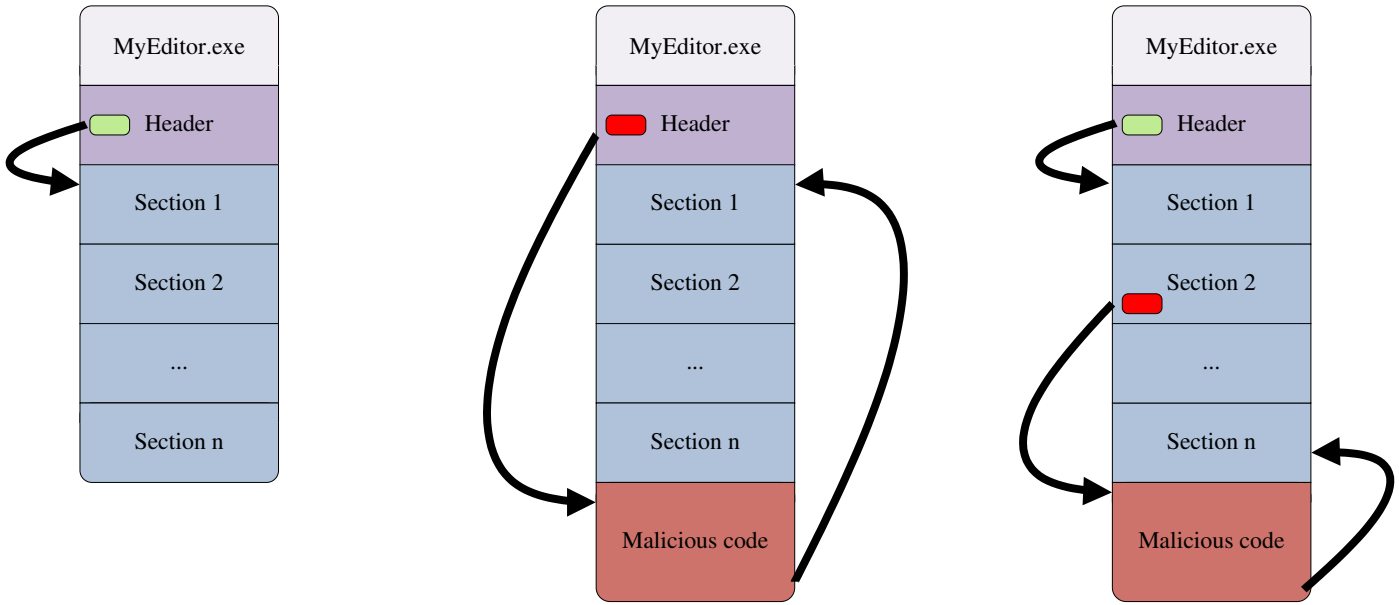


Figure: Principle of infection of an executable

Infect an executable - Implementation

Not so easy to implement:

- several sections might have to be added at the end of the executable
- sections must be mapped at the right address
- code must use dynamic address resolution

Inject code into another process - Principle

Description

- Malicious code injects some code into another process
- Malicious code forces the execution of this injected code in the context of the other process

Inject code into another process - Principle

Description

- Malicious code injects some code into another process
- Malicious code forces the execution of this injected code in the context of the other process

Objectives

- Survive to termination of original process: malicious code injects itself in "explorer.exe" and runs in this process
- Intercept private data of user using infected computer:
  - malicious code injects itself in a specific application
  - injected code uses API hooking to intercept calls of imported functions
  - analyses parameters passed to functions and looks for interesting data
- Bypass bad implemented personal firewalls: malicious code injects itself in a hidden instance of a browser and access to Internet

Inject code into another process - Principle

Code injection may be done in several ways:

- dll injection:
  - code is included in a dll
  - dll is then loaded and "executed" in targeted process
- direct code injection:
  - code injected directly into targeted process
  - relies on standard functions of Win32 API: OpenProcess, VirtualAllocEx, WriteProcessMemory and CreateRemoteThread

Each technique has pro and cons

Dll injection

Principle of dll injection

- Many solution to inject the dll
- One example: inject dll name and create thead on “LoadLibraryA” with injected string as argument

Demo : Example of “DemoDllInjection”

Direct code injection

Figure: Principle of direct code injection

Direct code injection

Figure: Principle of direct code injection

Direct code injection

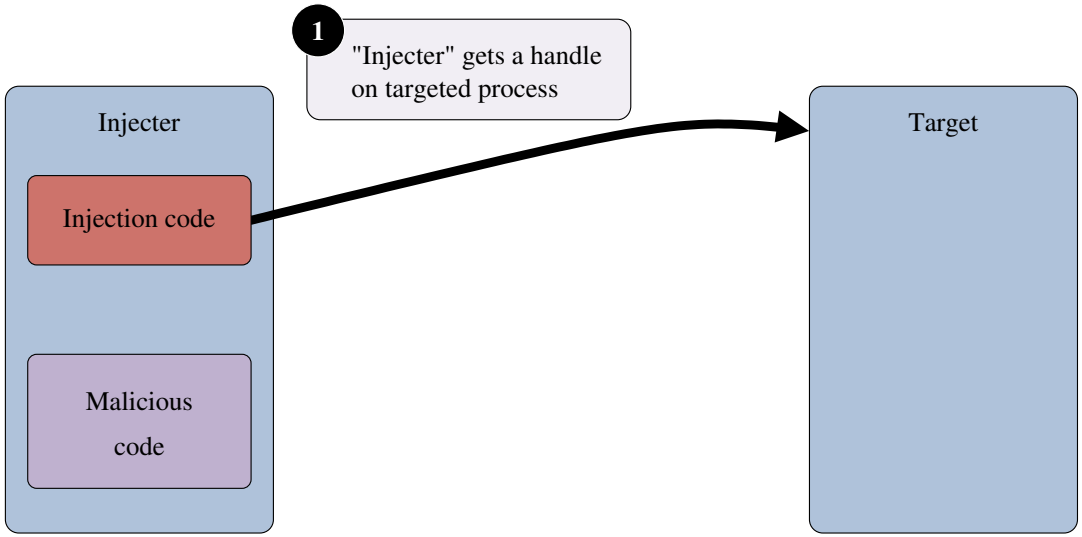


Figure: Principle of direct code injection

Direct code injection

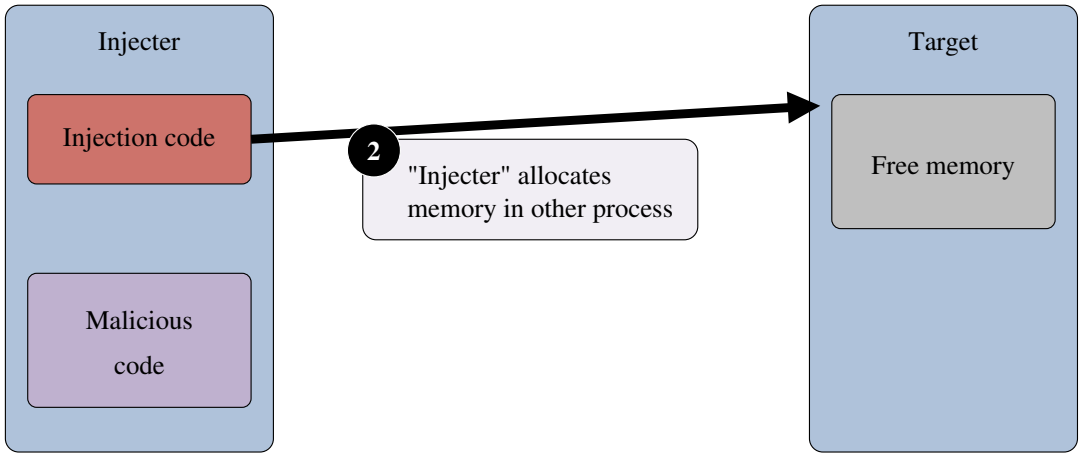


Figure: Principle of direct code injection

Direct code injection

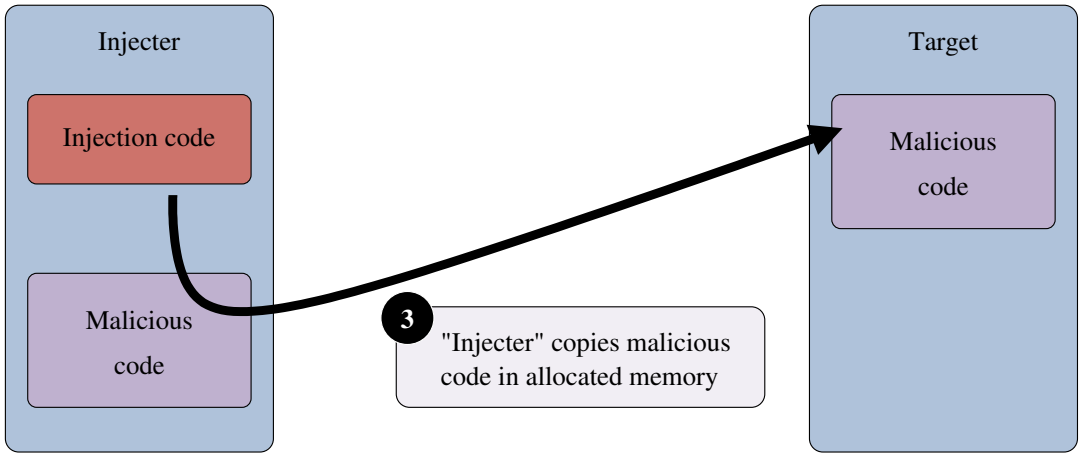


Figure: Principle of direct code injection

Direct code injection

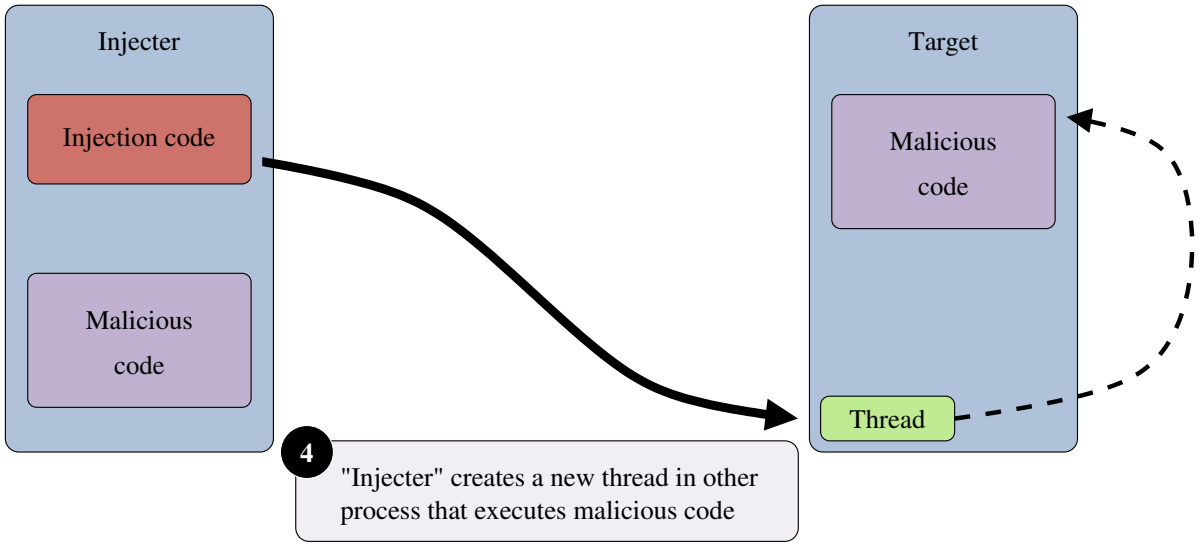


Figure: Principle of direct code injection

Direct code injection

- Encounter same problems as execution only in memory:
  - sections must be mapped at the right address
  - imported functions must be resolved

  $\Rightarrow$ Can use the same tricks
- Note that if memory where code must be mapped is already allocated, injection will fail!

Summary

- Implementation of those techniques in an executable is always possible, but requires lots of work
- Difficulties come from several properties of the executable:
  - code and data are spread in the executable;
  - process requires some of initialisation normally done by Windows loader
  - code contains hardcoded addresses $\Rightarrow$ sections must be mapped at the right addresses

Summary

- Implementation of those techniques in an executable is always possible, but requires lots of work
- Difficulties come from several properties of the executable:
  - code and data are spread in the executable;
  - process requires some of initialisation normally done by Windows loader
  - code contains hardcoded addresses $\Rightarrow$ sections must be mapped at the right addresses
- Those techniques could be implemented more easily if the code:
  - was constituted of only one block
  - was able to initialise the address space
  - contained no hardcoded address

$\Rightarrow$ if the malicious code was a shellcode

Plan

The use of shellcodes in virology

—

Implementation of the techniques from a shellcode

Principle

Consider now that our malicious code is a shellcode:

- constituted of only one block
- can run at any address in any process
- executes exactly the same operations as the normal executable if execution transferred to its first byte

Implementation of the techniques

Encryption of malicious code

Decryption part becomes a simple loop that executes decryption on shellcode $\sim$ array of bytes

Execution only in memory and code injection

Easy to implement since by definition shellcode is able to execute in any process at any address

Executable infection

- shellcode added in last section
- few modifications done on PE header
- entry point or instruction patched to jump on shellcode
- jump to original instruction added at end of shellcode

Summary

- Implementation of presented techniques is greatly simplified if the malicious code is a shellcode rather than an executable
- Next problem is how to get a shellcode?

Plan

Properties of a shellcode

Shellcode are generally used in an exploit. Must follow several constraints :

- must be relocalisable
- must be autonomous
- must be small
- must avoid some special values (null bytes for example)

Lots of constrains $\Rightarrow$ generally written in assembly

Writing a shellcode: tips

Example of structure to avoid hardcoded addresses

```
    jmp short getaddr
function:
    pop     esi     ; Get address of string in esi
    push    esi     ; Put address of string on stack

    ...

getaddr:
    call    function
shell_string:
    db '/bin/sh'
```

Writing a shellcode: tips

Avoid null bytes

```
mov     eax,0       ; b8 00 00 00 00
xor     eax,eax     ; 33 c0
```

How to call functions exported by shared libraries?

- Shared library may be not loaded in process address space
- Even if it is loaded, how to know load address of shared library?

Solutions depend on the operating system

Calling functions exported by shared libraries: Linux case

- System services call through int 0x80 (or sysenter); index of services set in eax
- Number of system services are fixed and stable across kernel versions
- Example : starting a new process : eax=0xb

$\Rightarrow$ Shellcode just has to use system services instead of functions exported by libraries

Calling functions exported by shared libraries: Linux case

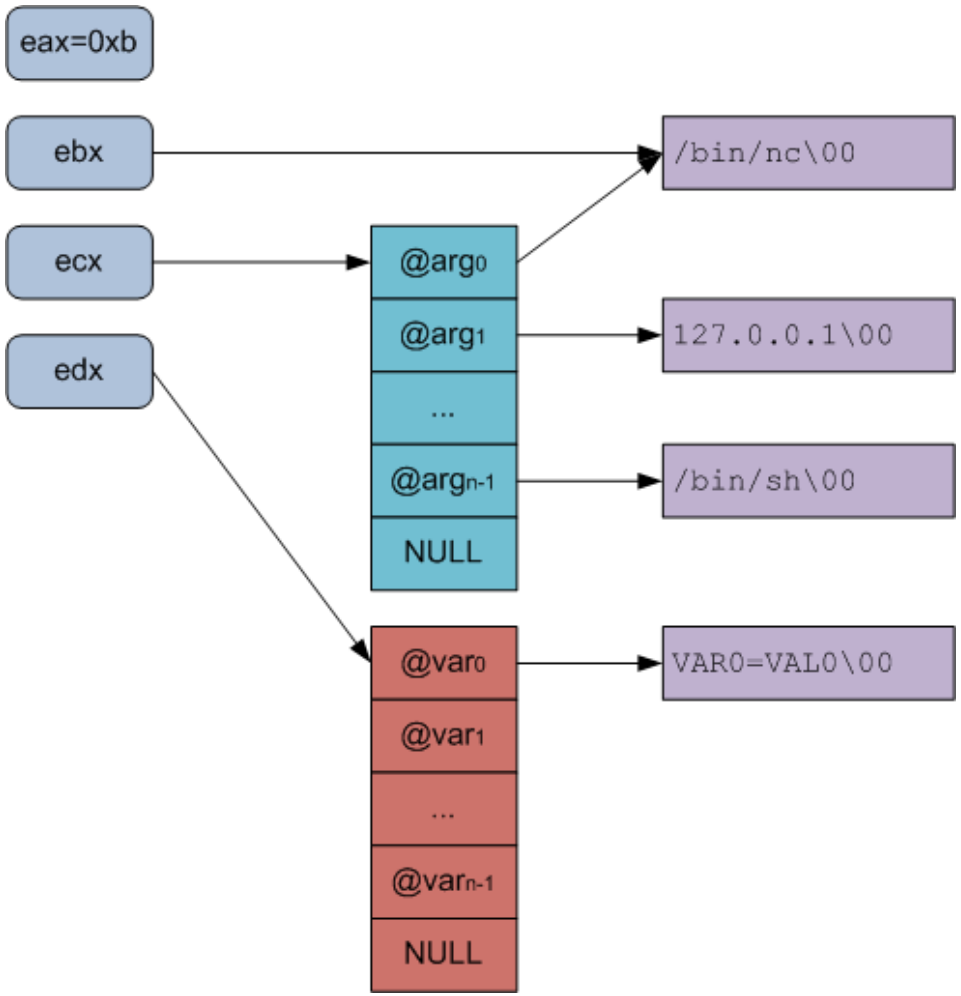


Figure: Values of x86 registers before calling service 0xb of int 0x80

Calling functions exported by shared libraries: Windows case

Problem

Number of system services change from a version of Windows to another
$\Rightarrow$ cannot use system services directly; must use functions exported by shared library

Two solutions

- Assume that shared library are loaded at a known address and hardcode function addresses
  $\Rightarrow$ small size, but not portable
- Dynamically find addresses of required imported functions
  $\Rightarrow$ portable, but bigger

Calling functions exported by shared libraries: Windows case

- Resolving an imported function implies:
  - loading the library that exposes the function
  - finding the function address in this library
- Can use the functions LoadLibrary/GetProcAddress exported by kernel32.dll
- Paradox: how to find the addresses of those functions?

Calling functions exported by shared libraries: Windows case

Common method:

- gets address of kernel32.dll by analysing memory
- walks through kernel32.dll exports table to find addresses of LoadLibrary/GetProcAddress

Finding the load address of kernel32.dll

- Several technics
  - Through PEB
  - Through SEH (UEF generally points in kernel32.dll)
- Use the first, more reliable

Finding the load address of kernel32.dll

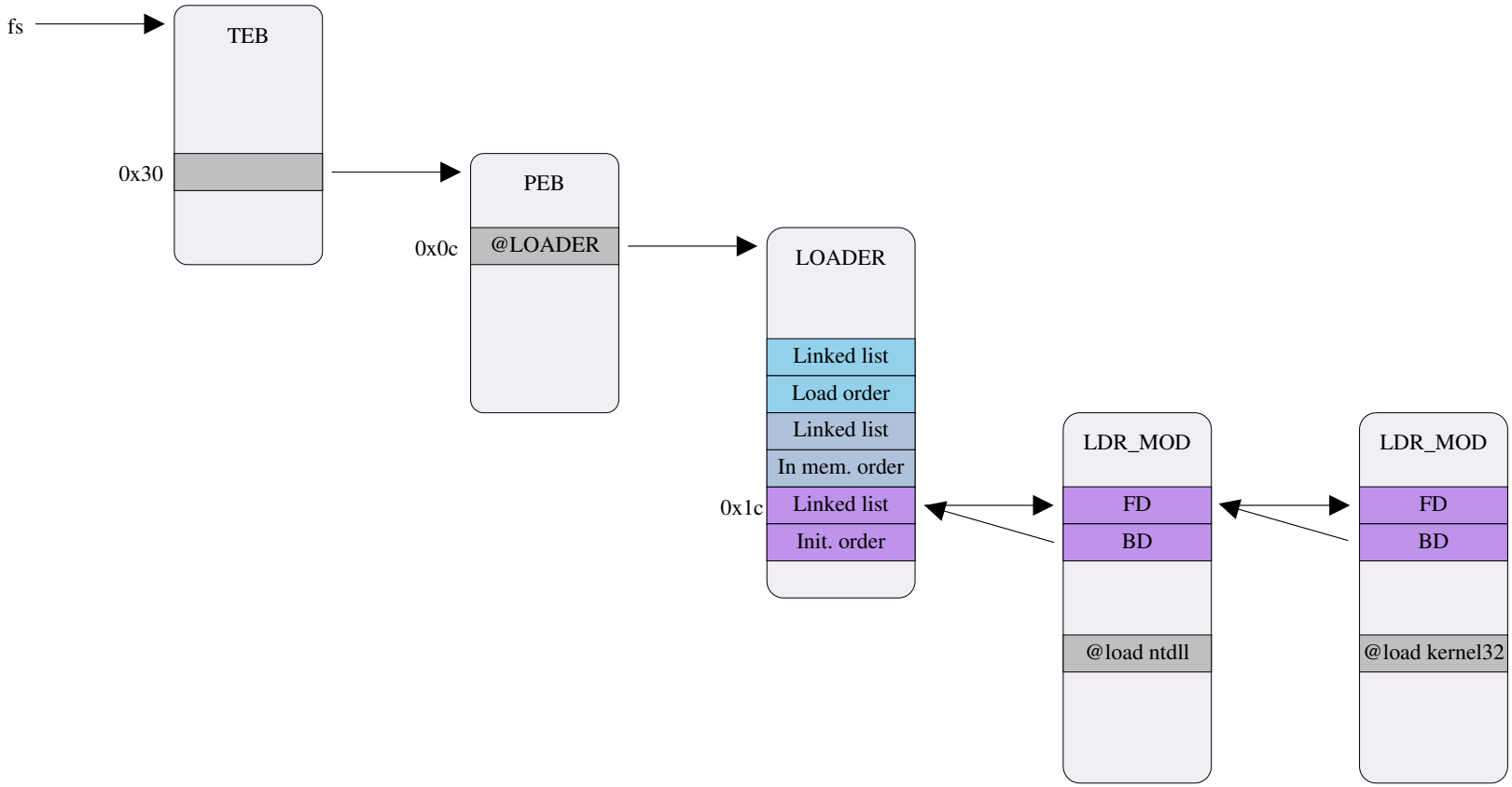


Figure: Finding load address of kernel32.dll through the PEB

Finding address of LoadLibrary/GetProcAddress

- load address of kernel32.dll has been found
- addresses of LoadLibrary/GetProcAddress can be found by parsing exports table
- only need to know the PE format

Finding the addresses of other functions

- Once addresses of LoadLibrary/GetProcAddress have been find, all functions may be resolved
- Problem: names of the functions must be included in the shellcode, and name of Win32 API are *very* long
- A better solution: write a function “getprocaddressbycksum” that resolves a function from a 32-bits checksum computed from its name
- Checksum algorithm must be well chosen to avoid collision as much as possible
- “getprocaddressbycksum” is not really a new function, since we already need such a function to find the addresses of “LoadLibrary” and “GetProcAddress”
- Finally, we don’t really need the function “GetProcAddress”

Writing shellcode for windows

Demo : Using functions getk32 and getprocaddress of the Metasploit project

Plan

Objective of this part

- Present an easy way to write the malicious code as a shellcode
- In this context, shellcode are a little different: can be big (or huge), no forbidden values
- Writing shellcode directly in assembly quickly becomes tedious
  $\Rightarrow$ solution dismissed
- Better solution would be:
  - write code in C language
  - use compiler to generate executable
  - extract some part from this executable
  - form shellcode by assembling them
- First, let us have a look on the binary code generated by a normal compilation on “simpletest”

Presentation of simpletest

- Very simple program that prints messages and displays the content of a file "test.txt"
- Contains:
  - definition of a new type "USER"
  - two global variables;
    - **"g_User"**: type "USER"
    - **"g_szMessage"**: string
  - five internal functions:
    - **"DisplayMessage"**: displays "g_szMessage"
    - **"DisplayFile"**: opens a file "test.txt" and displays its content
    - **"DisplayData"**: function that really executes all operations
    - **"main"**: program entry point that only calls "DisplayData"
    - **"PrintMsg"**: displays log messages
  - several strings
  - several calls to imported functions: CreateFile, HeapAlloc...

$\Rightarrow$ not really useful but contains most elements of C program

Analysis of generated assembly

Demo : Analysis of assembly generated by the build of simpletest original code

Analysis of generated assembly

Binary code cannot be directly used to create a shellcode:

- contains lots of hardcoded addresses (reference to a string or a global variable)
- internal functions calls are relative but distance is hardcoded
- imported function calls rely on IAT

To obtain binary code that may be used in a shellcode, we have to:

- force the compiler to produce code without hardcoded addresses
- find a solution to resolve imported function dynamically

First approach: patching assembly

- Assembly is generated with the compiler, patched with a transformation tool and then assembled to generate binary data
- Several problems:
  - lots of modification to do on assembly
  - transformation tool has to work on assembly, which is not really a natural language
  - transformation tool will be linked to a specific assembly and then to a specific hardware platform

$\Rightarrow$ Solution dismissed

Second approach: using the stack

- C code is written in a specific way so that everything is handled in the stack

Example 1: use of the stack to store a string

```
        CHAR szStrUsername[] = {'U', 's', 'e', 'r', 'n', 'a', 'm', 'e', ':', ' ', '%', 's'};
004130DA   C645 F4 55       MOV BYTE PTR SS:[EBP-C],55
004130DE   C645 F5 73       MOV BYTE PTR SS:[EBP-B],73
004130E2   C645 F6 65       MOV BYTE PTR SS:[EBP-A],65
004130E6   C645 F7 72       MOV BYTE PTR SS:[EBP-9],72
...
```

Example 2: use of the stack to store pointer on an imported function

```
hLib = LoadLibrary(szStrLibraryName);
pFunc = (FunctionTypeDef) GetProcAddress(hLib, szStrFunctionName);
pFunc(...);
```

- Problems: code is far from normal C code: tedious to write, existing code must be adapted $\Rightarrow$ Solution dismissed

Third approach: using global data

- Use one structure that stores all global data and that is transmitted in every internal function call
- Structure, called later "GLOBAL_DATA", will contain:
  - pointers on internal functions
  - pointers on imported functions
  - global variables
  - strings
- C code is modified so that every reference to a previously listed element will be done through GLOBAL_DATA

Third approach: using global data

Original function DisplayFile

```
BOOL DisplayFile(IN CHAR * szFilePath)
{
        ...
        CreateFile(szFilePath, ...)
        pData = (UCHAR *) HeapAlloc(GetProcessHeap(), HEAP_ZERO_MEMORY, dwFileSize+1)
        ReadFile(hFile, pData, ...)
        PrintMsg(LOG_LEVEL_TRACE, "File successfully read: %s", pData);
        ...
}
```

Patched function DisplayFile (modifications are colorized in red)

```
BOOL DisplayFile(IN PGLOBAL_DATA pGlobalData, IN CHAR * szFilePath)
{
        ...
        pGlobalData->CreateFile(szFilePath, ...)
        pData = (UCHAR *) pGlobalData->HeapAlloc(pGlobalData->GetProcessHeap(), \\
                HEAP_ZERO_MEMORY, dwFileSize+1)
        pGlobalData->ReadFile(hFile, pData, ...)
        pGlobalData->PrintMsg(pGlobalData, LOG_LEVEL_TRACE, pGlobalData->szString_00000001, \\
                pData);
        ...
}
```

Third approach: using global data

The GLOBAL_DATA definition looks like the following:

Overview of structure GLOBAL_DATA

```
typedef struct _GLOBAL_DATA
{
        /* Internal functions */
        PrintMsgTypeDef fp_PrintMsg;

        /* Imported functions */
        CreateFileTypeDef fp_CreateFile;
        HeapAllocTypeDef fp_HeapAlloc;
        GetProcessHeapTypeDef fp_GetProcessHeap;
        ReadFileTypeDef fp_ReadFile;

        /* Data strings */
        CHAR szString_00000001[27];

} GLOBAL_DATA, * PGLOBAL_DATA;
```

Third approach: using global data

Demo : Analysis of assembly generated by the build of simpletest patched code

Third approach: using global data

- Generated binary does not contain any hardcoded addresses
  $\Rightarrow$ binary code can be directly extracted and used to form shellcode
- Shellcode may be created simply by concatenating the extracted functions and adding the GLOBAL_DATA structure at the end

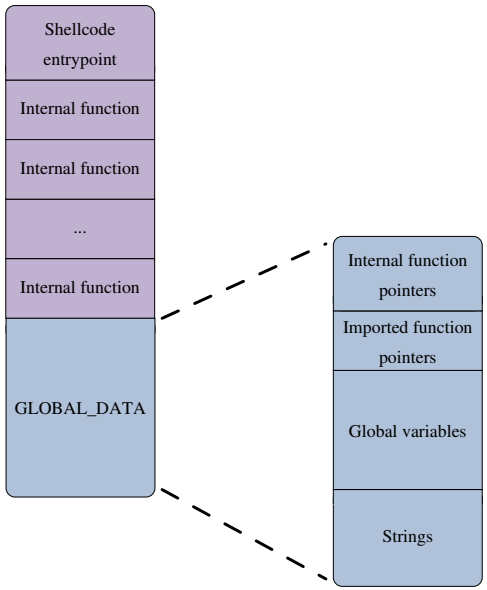


Figure: Overview of the structure of the shellcode

Summary

- This solution allows a shellcode to be created with little modifications of source code
- However, still a few problems to solve:
  - writing the definition of the GLOBAL_DATA structure and the definition of macros is long
  - the GLOBAL_DATA structure must be initialised
  - source code must be modified
  - binary data must be extracted from generated executable and assembled to create final shellcode

$\Rightarrow$ A tool that executes all those operations automatically has been developed: WiShMaster

Plan

Plan

WiShMaster in a nutshell

—

Versions of WiShMaster

Presentation

- WiShMaster is a tool that automatically generates shellcodes, by using the previously described principle
- Takes a set of C source files written "normally" in input and generates a shellcode in output
- Shellcode accomplishes same operations as executable produced by compilation of original source
- Transformation in shellcode called later "shellcodisation"

Development progress - WiShMaster version 1

- WiShMaster v1 has been available on my web site for two years
- Graphical application developed in C#
- Works but has several limitations
  Most important: C code parsed with regular expressions $\Rightarrow$ must conform to a few syntax rules to be successfully analysed

Development progress - WiShMaster version 2

- WiShMaster v2 is beta release
- Corrects many problems of the v1:
  - WiShMaster is now a console application written in Python:
    - shellcodisation process can be scripted
    - user can intercede at any step of the shellcodisation process, view results and correct eventual mistakes
  - parsing of source code with regular expressions has been considerably reduced $\Rightarrow$ most of the constrains on C syntax have been removed

Plan

WiShMaster in a nutshell

—

Format of source code in input

Format of source code in input

During development of WiShMaster v2, have to do a choice on the type of input code:

- either develop WiShMaster so it can operate on normal C code
- either require that C code is written in a specific way

Each solution has pro and cons

Solution 1 : operate on normal C code - Principle

- Write a C analyser or use an existing to analyse C code and recognize specific objects
- Create a patched copy of source code
- Build the patched copy

Solution 1 : operate on normal C code - Analyse source code

- gcc: option -fdump-tree-original-raw generate AST in text file
  Problems:
  - a parser for the AST must be written
  - to debug with Microsoft Visual Studio, code must be analysed by gcc and build by cl.exe
- pycparser: doesn't manage to parse windows.h
- write our own C analyser: very complicated to be able to parse windows.h
- cl.exe: use browse or debug file
  Problem: doesn't give all informations (for example list of strings)
  $\Rightarrow$ still need to analyse C code with regular expressions

Solution 1 : operate on normal C code - Create patched copy

In some special cases, WiShMaster may generate invalid C code

Original code

```
        printf("test", test
(10));
```

Patched code (invalid)

```
        printf("pGlobalData->test", test
(10));
```

Solution 1 : operate on normal C code - Summary

To sum up, this solution works, but :

- requires some work to develop a code analyser
- is not really a proper solution since all files are duplicated
- WiShMaster needs to analyse source code with regular expressions
- may have some problem if code is not formated “normally”:
  - WiShMaster may failed to analyse source code
  - patched code generated by WiShMaster may be invalid

$\Rightarrow$ Solution dismissed

Solution 2 : work on specific code - Objective

Objective

Force C code to be formatted in a special way, so that:

- the same source code may generate a normal executable or relative binary code
- it may be analysed easily by WiShMaster with regular expression to extract some informations

Solution 2 : work on specific code - Objective

Objective

Force C code to be formatted in a special way, so that:

- the same source code may generate a normal executable or relative binary code
- it may be analysed easily by WiShMaster with regular expression to extract some informations

Solution

- Use C macros !
- WiShMaster defines some macros that must be used when declaring/using internal/imported functions, global variables and strings

Solution 2 : work on specific code - Internal functions

- Internal functions are declared through a macro "INTERNAL_FUNCTION"
- Calls are normal

Presentation of the macro "INTERNAL_FUNCTION"

```
INTERNAL_FUNCTION(ReturnType,CallConvention,Special,FunctionName,...)
  ReturnType     = return type of the function
  CallConvention = type of convention
  Special        = special keyword to add after type of convention
  FunctionName   = name of internal function
```

Solution 2 : work on specific code - Internal functions

Declaration of an internal function "PrintMyMessage"

```
#pragma push_macro("PrintMyMessage")
#undef PrintMyMessage
INTERNAL_FUNCTION(UINT,,,PrintMyMessage, IN UINT i)
#pragma pop_macro("PrintMyMessage")
{
    printf("Hello world : %.8x!", i);
    return 0;
}
```

Call of the internal function "PrintMyMessage"

```
...
PrintMyMessage(0xaabbccdd)
...
```

Value of macro "INTERNAL_FUNCTION" is modified according to the type of binary we want to produce

Solution 2 : work on specific code - Internal functions

Definition of macros to produce normal executable

```
#define INTERNAL_FUNCTION(ReturnType,CallConvention,Special,FunctionName,...) \
extern "C" ReturnType CallConvention Special FunctionName(__VA_ARGS__)
```

Declaration of “PrintMyMessage” with those macros

```
UINT PrintMyMessage(IN UINT i)
{
    printf("Hello world : %.8x!", i);
    return 0;
}
```

Call of “PrintMyMessage” with those macros

```
    ...
    PrintMyMessage(0xaabbccdd)
    ...
```

Solution 2 : work on specific code - Internal functions

Definition of macros to produce relative binary code

```
#define INTERNAL_FUNCTION(ReturnType,CallConvention,Special,FunctionName,...) \
extern "C" ReturnType CallConvention Special FunctionName(IN PGLOBAL_DATA pGlobalData,  \
  __VA_ARGS__)

#undef PrintMyMessage
#define PrintMyMessage(...)    pGlobalData->fp_PrintMyMessage(pGlobalData, __VA_ARGS__)
```

Declaration of “PrintMyMessage” with those macros

```
UINT PrintMyMessage(IN PGLOBAL_DATA pGlobalData, IN UINT i)
{
    printf("Hello world : %.8x!", i);
    return 0;
}
```

Call of “PrintMyMessage” with those macros

```
    ...
    pGlobalData->fp_PrintMyMessage(pGlobalData, 0xaabbccdd)
    ...
```

Solution 2 : work on specific code - Imported functions

Definition of macros to produce relative binary code

```
#undef printf
#define printf           pGlobalData->fp_printf
#undef _vsnprintf
#define _vsnprintf       pGlobalData->fp__vsnprintf
#undef CreateProcess
#define CreateProcess    pGlobalData->fp_CreateProcess
```

Call of “printf” with those macros

```
...
pGlobalData->fp_printf("Hello world : %.8x!", i);
...
```

Solution 2 : work on specific code - Global variables

- Global variable declared through a macro
- Use is normal

Presentation of the macro "GLOBAL_VAR"

```
GLOBAL_VAR(type,name,value)
  type   = type of the global variable
  name   = name of the global variable
  value  = initialisation value of the global variable
```

Solution 2 : work on specific code - Global variables

Declaration of a global variable "g_uiValue"

```
#pragma push_macro("g_uiValue")
#undef g_uiValue
GLOBAL_VAR(UINT,g_uiValue,0xaabbccdd);
#pragma pop_macro("g_uiValue")
```

Use of global variable "g_uiValue"

```
...
g_uiValue = 0xabcdabcd;
...
```

Value of macro "GLOBAL_VAR" is modified according to the type of binary we want to produce

Solution 2 : work on specific code - Global variables

Definition of macros to produce normal executable

```
#define GLOBAL_VAR(type,name,value)    type name = value;
```

Declaration of "g_uiValue" with those macros

```
UINT g_uiValue = 0xaabbccdd;
```

Use of "g_uiValue" with those macros

```
    ...
    g_uiValue = 0xabcdabcd;
    ...
```

Solution 2 : work on specific code - Global variables

Definition of macros to produce relative binary code

```
#define GLOBAL_VAR(type,name,value)     type name = value;
#define g_uiValue                       (*((UINT *)&(pGlobalData->_g_uiValue)))
```

Declaration of "g_uiValue" with those macros

```
UINT g_uiValue = 0xaabbccdd;
```

Use of "g_uiValue" with those macros

```
...
(*((UINT *)&(pGlobalData->_g_uiValue))) = 0xabcdabcd;
...
```

Solution 2 : work on specific code - Strings

Strings must be included in a macro “STR”

Presentation of the macro “STR”

```
#define STR(s)
  s = string
```

Solution 2 : work on specific code - Strings

Use of a string

```
...
printf(STR("Hello world : %.8x!"), i);
...
```

Value of macro “STR” is modified according to the type of binary we want to produce

Solution 2 : work on specific code - Strings

Definition of macros to produce normal executable

```
#define STR(s) s
```

Use of the string with those macros

```
...
printf("Hello world : %.8x!", i);
...
```

Solution 2 : work on specific code - Strings

Definition of macros to produce relative binary code

```
#define STRID(file_id,string_id,string) pGlobalData->szString_##file_id##_##string_id
#define STR_TEMP(file_id,string_id,string) STRID(file_id,string_id,string)
#define STR(string) STR_TEMP(FILEID,__COUNTER__,string)
```

Use of the string with those macros

```
...
printf(pGlobalData->szString_7_5, i);
...
```

Plan

WiShMaster in a nutshell

—

The shellcodisation process

The shellcodisation process

Shellcodisation accomplished by WiShMaster is divided into 4 steps:

- **Analysis**: identifies code elements
- **Environment creation**: creates sources file like global_data.h (GLOBAL_DATA structure and macros)
- **Generation**: builds sources, extracts binary data, generates the shellcode and customize it
- **Integration**: (optionnal) builds an external project that may include the generated shellcode

The customization step - 1

Principle

- Step compounded of a chain of functions that will execute some modifications on the shellcode and transmit the modified shellcode to the next function
- Content of the chain is defined by the user
- Customization functions implemented in Python module $\Rightarrow$ user can easily write their own customization module

The customization step - 2

Example 1: encryption

- Customization step may be used to encrypt the shellcode
- WiShMaster comes with two "customization" modules that can encrypt a shellcode:
  - XOR encryption with a 32-bits key (polymorphism)
  - AES-CBC encryption with a 256-bits key

The customization step - 2

Example 1: encryption

- Customization step may be used to encrypt the shellcode
- WiShMaster comes with two "customization" modules that can encrypt a shellcode:
  - XOR encryption with a 32-bits key (polymorphism)
  - AES-CBC encryption with a 256-bits key

Example 2: setting specific values

- Example: shellcode that connects to a server
- Source code contains two variables: IP address and port of the server
- If we put real values directly in those variables:
  - shellcode must be regenerated to connect to another server
  - shellcode cannot be distributed in its binary form

The customization step - 3

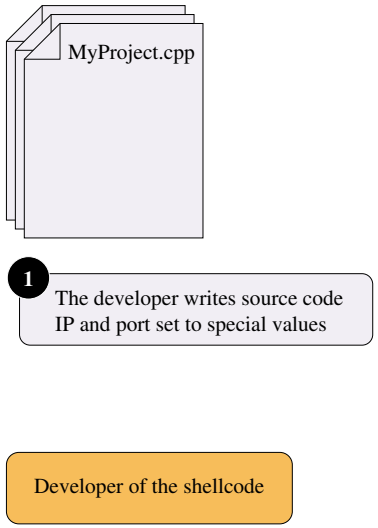


Figure: Principle of the separation between developer / user of a shellcode

The customization step - 3

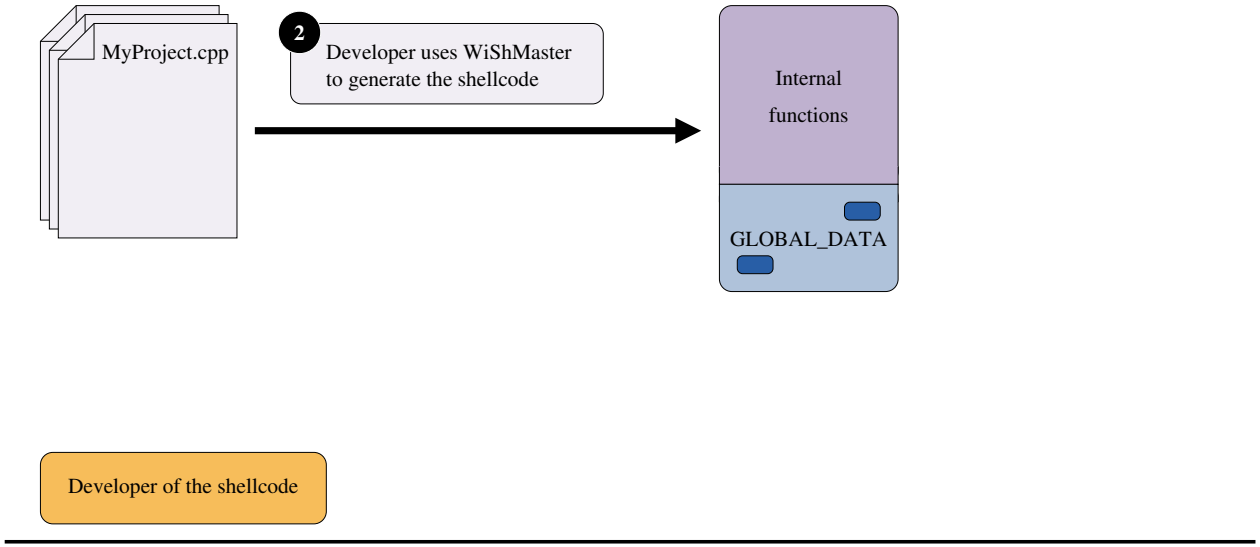


Figure: Principle of the separation between developer / user of a shellcode

The customization step - 3

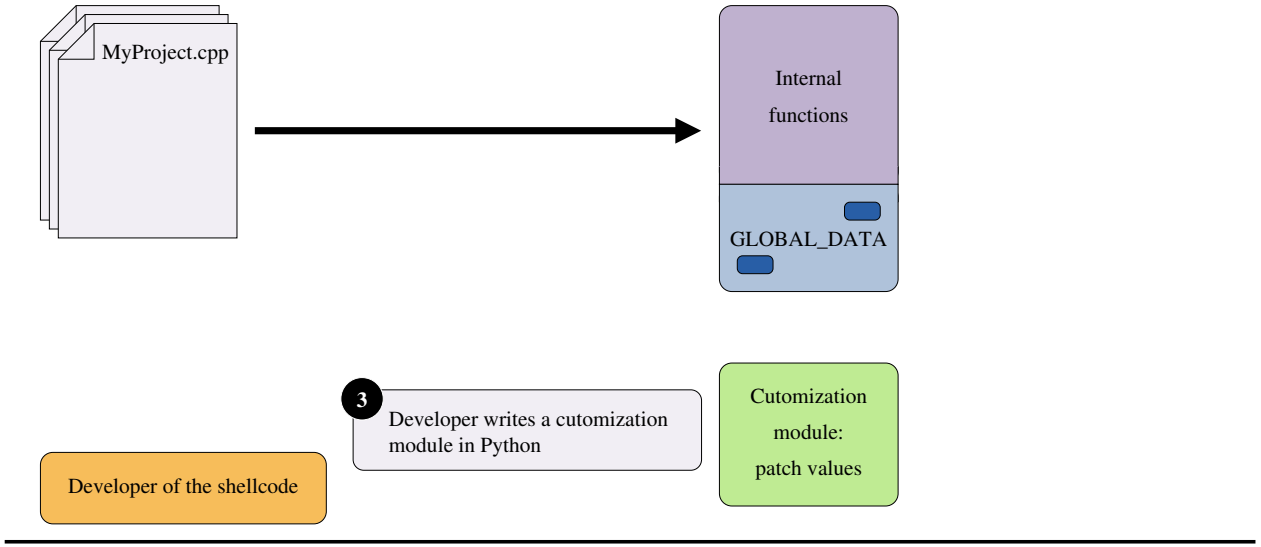


Figure: Principle of the separation between developer / user of a shellcode

The customization step - 3

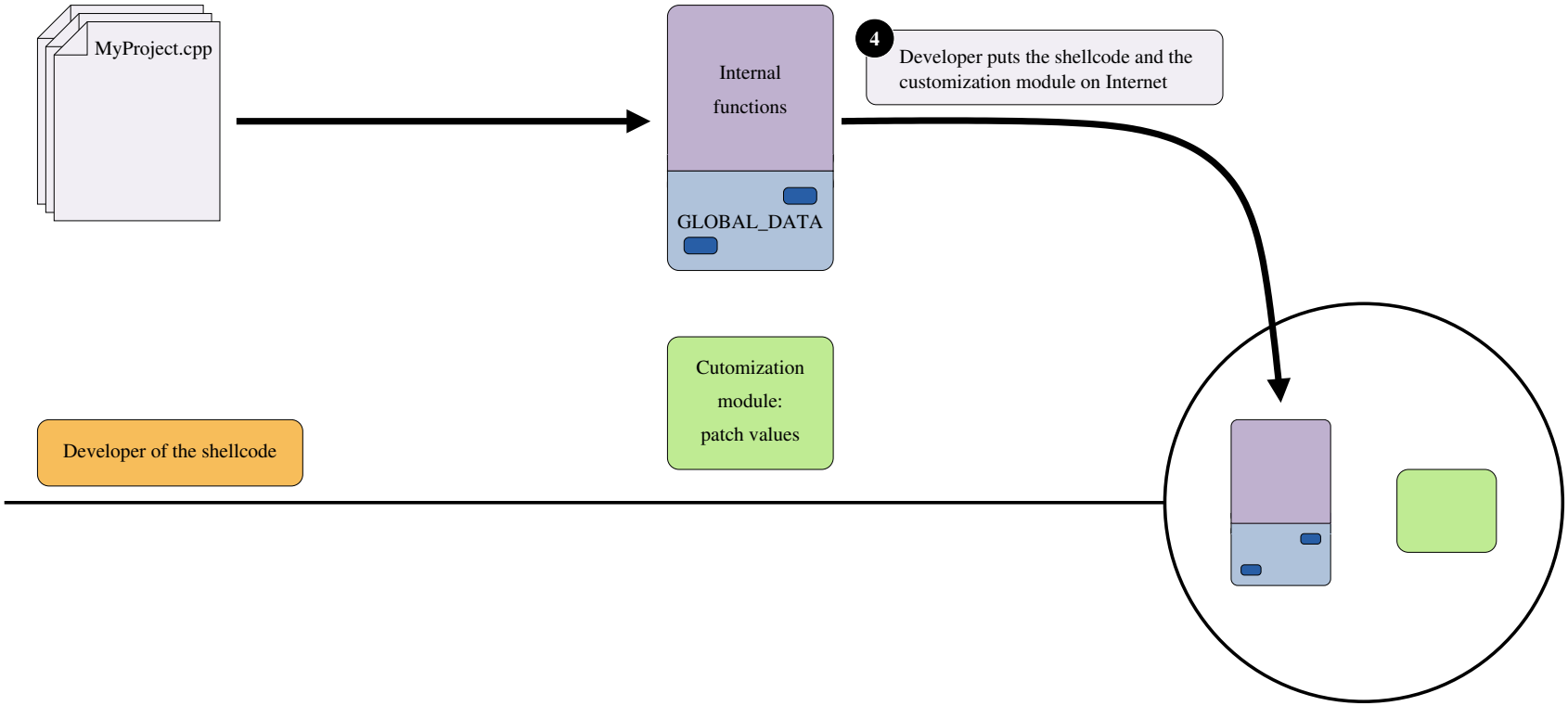


Figure: Principle of the separation between developer / user of a shellcode

The customization step - 3

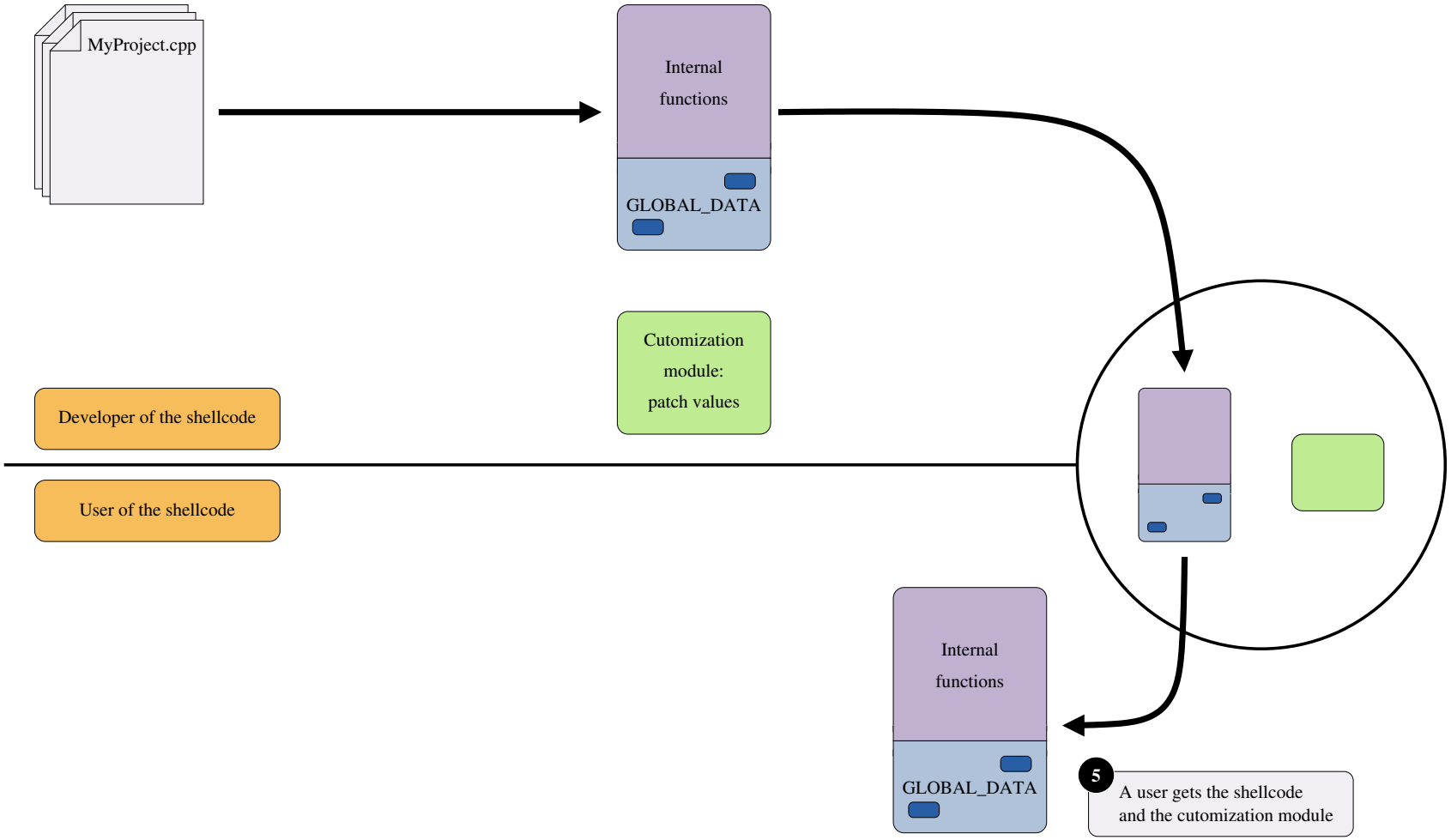


Figure: Principle of the separation between developer / user of a shellcode

The customization step - 3

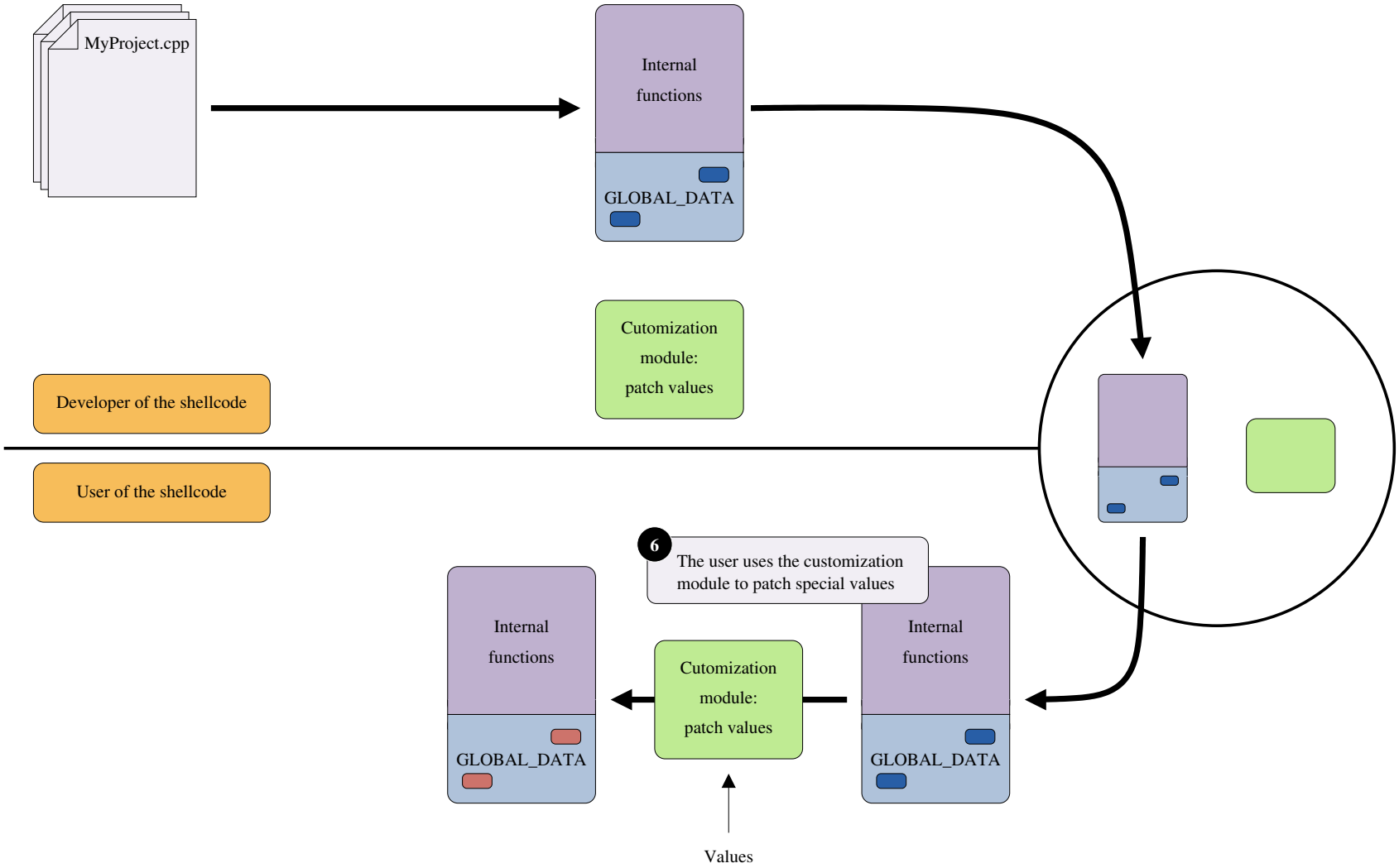


Figure: Principle of the separation between developer / user of a shellcode

The customization step - 3

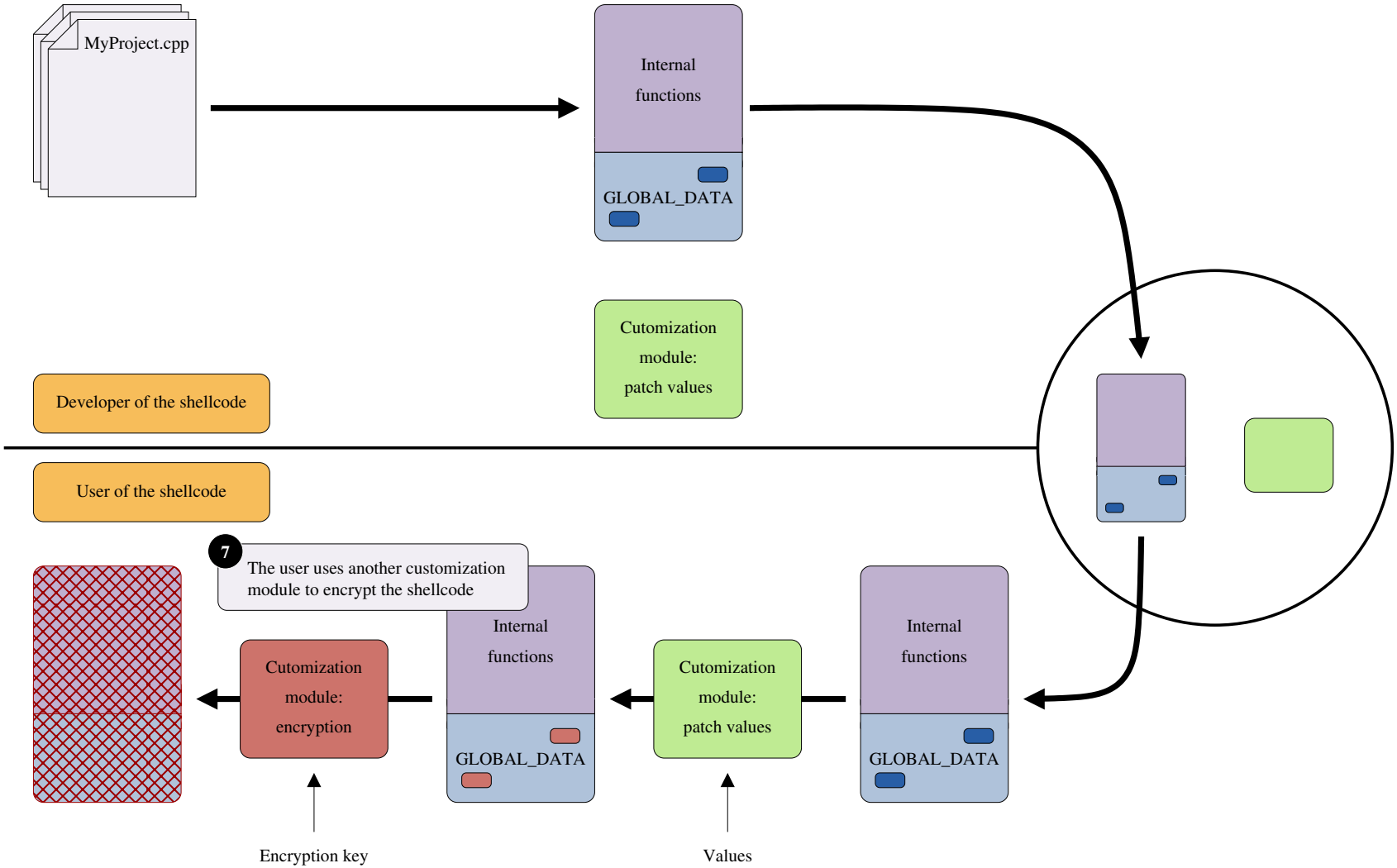


Figure: Principle of the separation between developer / user of a shellcode

Implementation of the shellcodisation in WiShMaster v2 - 1

Internally:

- Every element discovered in the source code $\sim$ an object (internal/imported functions, strings...)
- Every step of the shellcodisation divided into several small sub-steps
- Every sub-step implemented by one function

Implementation of the shellcodisation in WiShMaster v2 - 2

WiShMaster can be launched in three modes:

- **automatic**: executes the shellcodisation process automatically
- **script**: executes an external script that can call step/sub-step functions exported by WiShMaster and manipulate objects
- **interactive**: starts a Python shell (same principle as in Scapy)
  User can then:
  - call step/sub-step functions
  - execute a shellcodisation step by step by calling some functions step(), stepi(), run()... (like in a debugger)
  - display objects, change their properties to correct eventual mistakes

Plan

WiShMaster in a nutshell

—

Initialising the shellcode

Initialising the shellcode: objective

- Shellcodisation process described previously creates a binary code that may run at any address
- However, shellcode must initialise the GLOBAL_DATA structure
- Operation executed by a function added by WiShMaster, placed at the beginning of the shellcode:
  - find address of GLOBAL_DATA structure
  - find addresses of internal functions and fill pointers in GLOBAL_DATA
  - resolve imported functions and fill pointers in GLOBAL_DATA

Finding address of GLOBAL_DATA structure

Code used by the shellcode to find its load address

```
UCHAR * pShellcode = NULL;

/* Use a call/pop to get load address */
__asm
{
    push    eax
    call    GetLoadAddress
GetLoadAddress:
    pop         eax
    mov         pShellcode, eax
    pop         eax
}

/* Find "push ebp"/"mov ebp, esp" instructions to get real load address */
while((*(UINT *)(pShellcode-i) != 0x83EC8B55) && (i < 512))
    i++;
if(i == 512)
    return FALSE;
pShellcode -= i;

/* Get address of GLOBAL_DATA */
uiGlobalDataSize =  *(UINT *)&pShellcode[uiShellcodeSize-8];
pGlobalData = (PGLOBAL_DATA) &pShellcode[uiShellcodeSize-uiGlobalDataSize];
```

Finding addresses of internal functions

- During shellcodisation, WiShMaster includes size of each internal function
- Addresses of internal functions calculated step by step from the shellcode load address

Code used by the shellcode to rebuild pointers on internal functions

```
/* Rebuild pointers on internal functions */
for(i=0;i<pGlobalDataHeader->uiNbOfInternalFunctions;i++)
{
        pGlobalDataHeader->pInternalFunctionsTable[i].pFunctionPointer = p;
        p += pGlobalDataHeader->pInternalFunctionsTable[i].uiFunctionSize;
}
```

Finding addresses of imported functions

WiShMaster uses technics previously presented:

- gets address of kernel32.dll by analysing memory through the PEB (function "GetKernel32Address")
- resolves imported functions from a 32-bits checksum computed from function names ("GetProcAddressByCksumInDll")
  - checksum is computed with the Metaploit algorithm
  - must support dll forwarding (example: HeapAlloc in kernel32.dll)

Initialising the shellcode: summary

The shellcode initialisation relies on three functions:

- **"InitialiseShellcode"**: entry point of the shellcode, which initialises GLOBAL_DATA structure
- **"GetKernel32Address"**: returns the load address of "kernel32.dll"
- **"GetProcAddressByCksumInDll"**: finds an exported function from the checksum of its name (supports dll forwarding)

Plan

Demonstrations

- generation of “simpletest” as an executable
- generation of “simpletest” as a shellcode

Plan

Objectives of WiShMaster

- Version 1 of WiShMaster: creation of monolithic shellcodes
- With version 2, objectives have been considerably extended:
  - development of modular applications
  - user chooses output format: an executable, a dll or a shellcode
  - allows code reusability
  - development in the very powerful IDE Visual Studio
  - projects can be distributed either in source or in binary format

Overview of the application structure - 1

- A WiShMaster application is compounded of one or several “modules”
- A module can be in one of the following 4 forms:
  - an executable
  - a dll
  - a shellcode
  - inlined into another module
- Each module can export some of its functions so that they can be called by other modules
  $\Rightarrow$ each module contains an “export” table and an “import” table

Overview of the application structure - 2

Figure: Structure of an application developed with WiShMaster v2

Overview of the application structure - 2

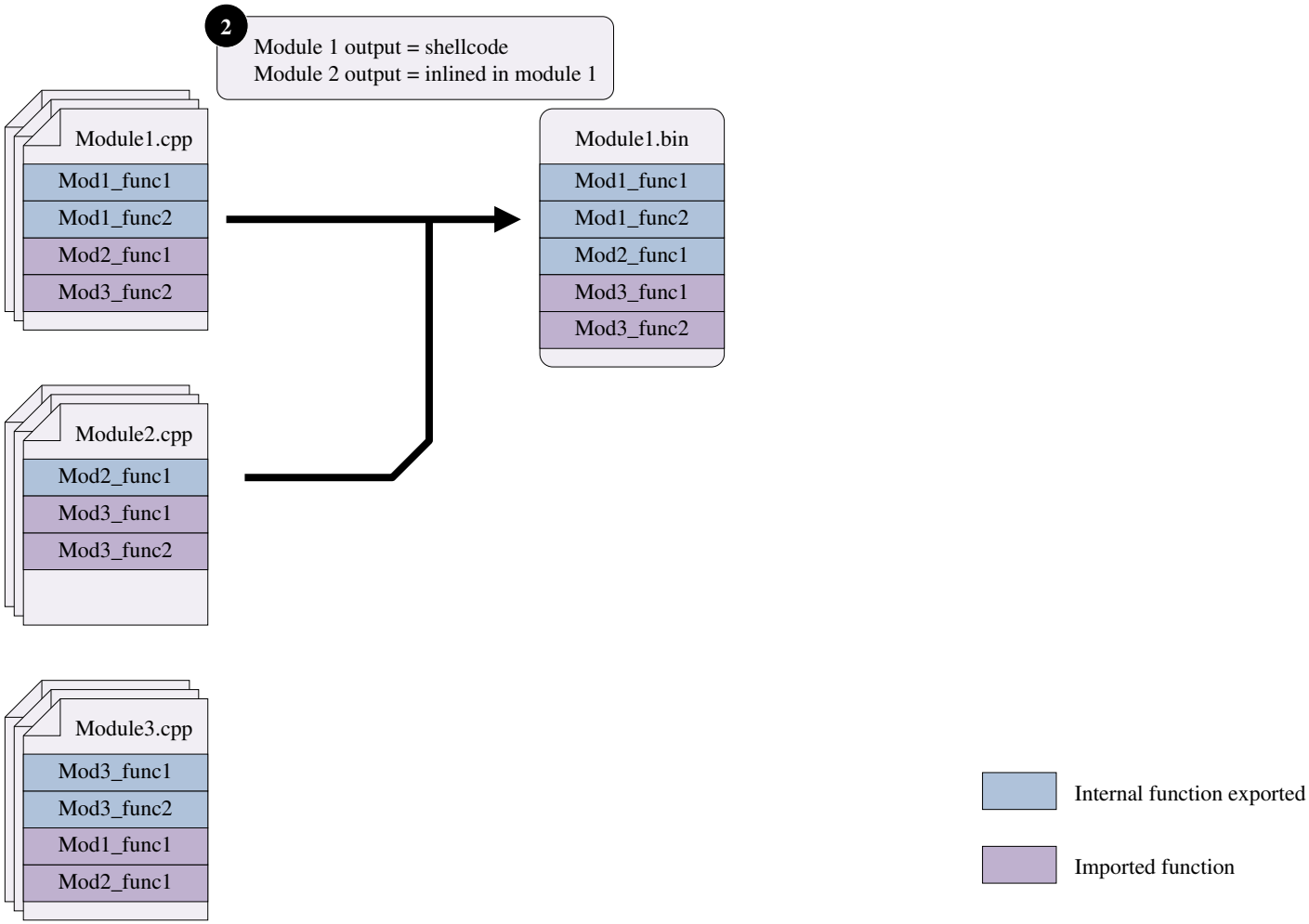


Figure: Structure of an application developed with WiShMaster v2

Overview of the application structure - 2

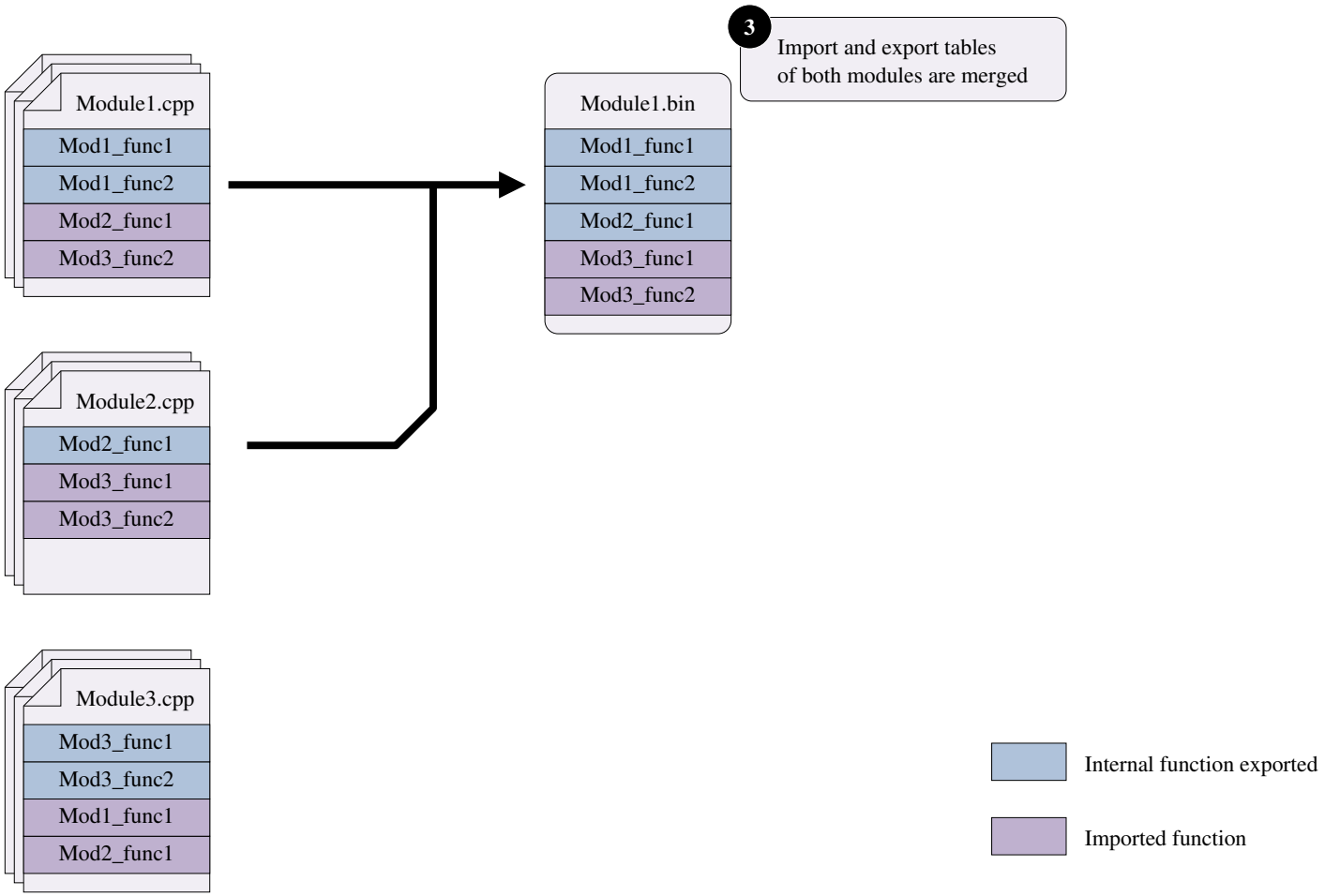


Figure: Structure of an application developed with WiShMaster v2

Overview of the application structure - 2

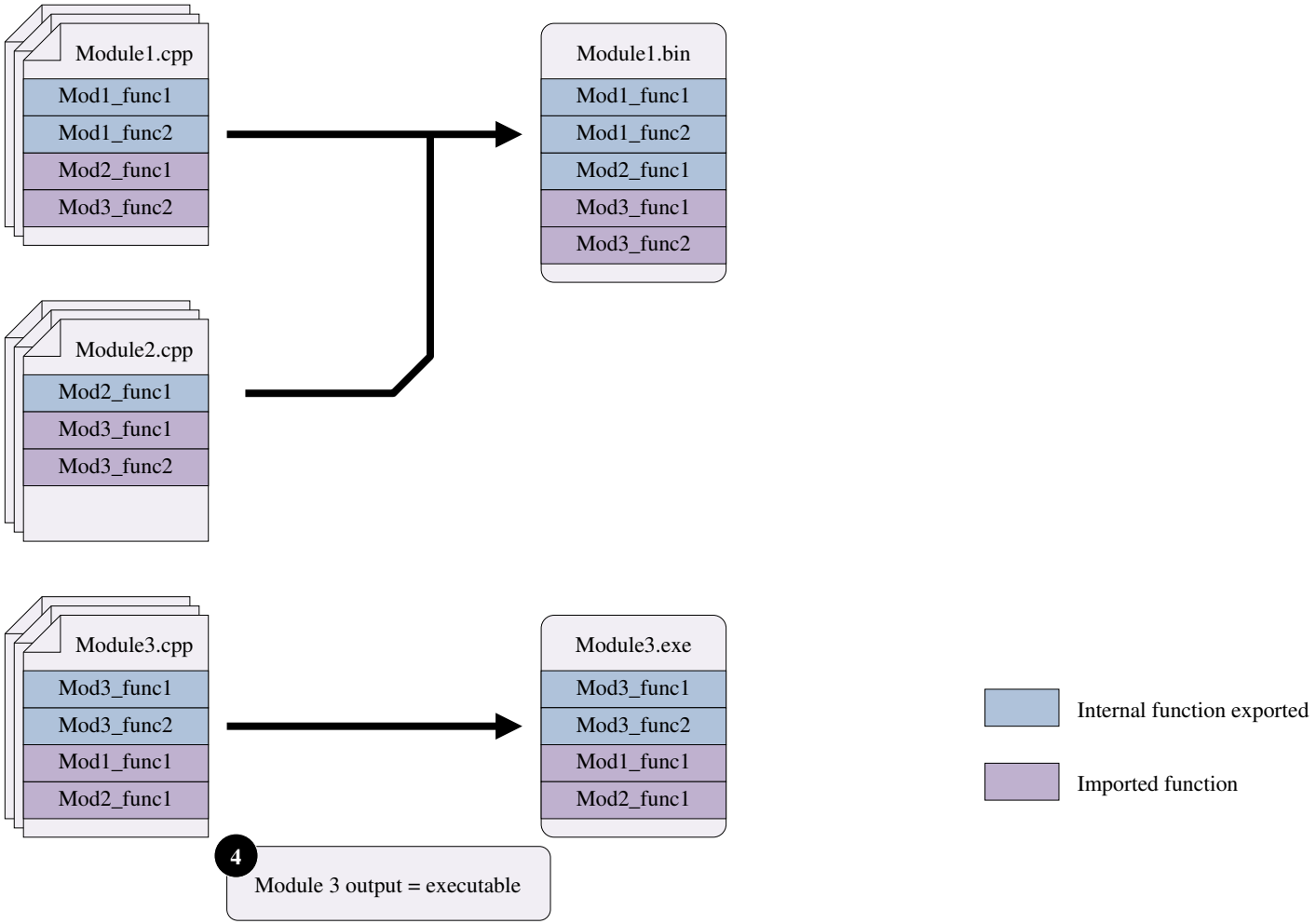


Figure: Structure of an application developed with WiShMaster v2

Overview of the application structure - 2

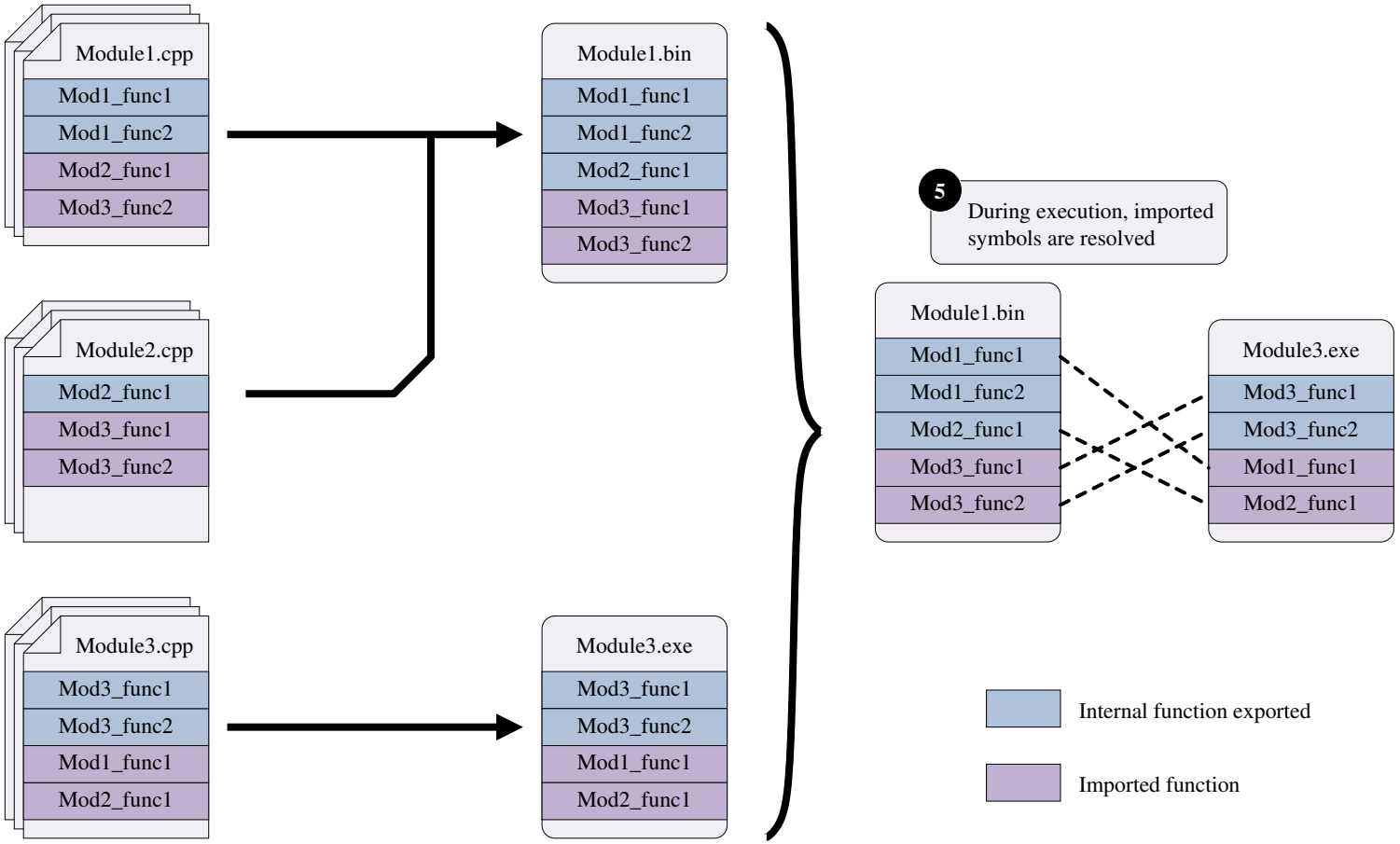


Figure: Structure of an application developed with WiShMaster v2

Binary format of a WiShMaster module - 1

Module must be able to:

- load without generating an error even if a required module is missing
- call function exported by a module independently of the format of this module (exe, dll, shellcode)

$\Rightarrow$ PE format cannot be used: WiShMaster defines its own binary format

Binary format of a WiShMaster module - 2

Structure of GLOBAL_DATA is normalized and contains:

- an export table: contains the checksum of the name of each exported function
- an import table: contains the names of each imported module and the checksum of the names of each imported function
  There is no difference between a function imported from a standard dll and one imported from a module (executable, dll or shellcode)
- an optional entry point: pointer on an internal function that must be called after module initialisation

Standard modules - 1

Module	Description	Functions
log	print of formatted messages	PrintMsg
initsh	initialise a shellcode	InitialiseShellcode GetProcAddressByCksumInDll
modman	manages a set of modules	AddLoadedModule
baseobj	expose functions to manipulate basic objects (linked list, memory, ...)	AddObjectAtHeadOfLinkedList RemoveObject BufferAllocate BufferFree ...
advobj	expose functions to manipulate advanced objects (hash table, managed buffer, ...)	ManagedBufferInitialise ManagedBufferFree ManagedBufferAddData ...
kernel	event dispatcher	SendKernelMessage RegisterNewKernelMessageHandler

Standard modules - 2

Module	Description	Functions
cryptoman	manage cryptographic keys and call cryptographic modules to try to decrypt encrypted files	RegisterCryptographicHandler RegisterNewKernelMessageHandler
filedisp	dispatch file to other modules, according to the type of the file	RegisterFileHandler RegisterDefaultFileHandler
cryptoxor	executes XOR-decryption with a 32-bits key	XorDecryption
cryptoaes	executes AES-decryption with a 256-bits key	AesDecryption
fileloader	load a file from hard drive in memory	LoadFile

Standard modules - 3

- Those modules can be used to create a "loader" = code that is able to load and manage a set of modules
- "loader" can be an executable or a shellcode
- some special capabilities may be added through new modules (find modules on USB key, download them from Internet, ...)

Example: Creation of an encrypted module and load from a simple loader
Demo : Creation of a new skeleton "test"

Module encryption

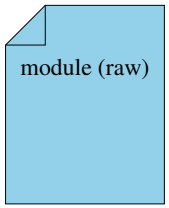


Figure: Module encryption

Module encryption

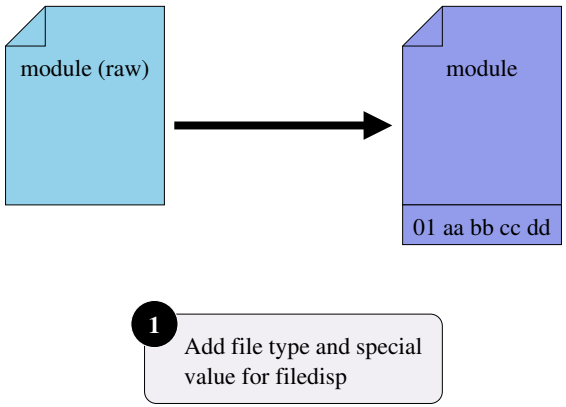


Figure: Module encryption

Module encryption

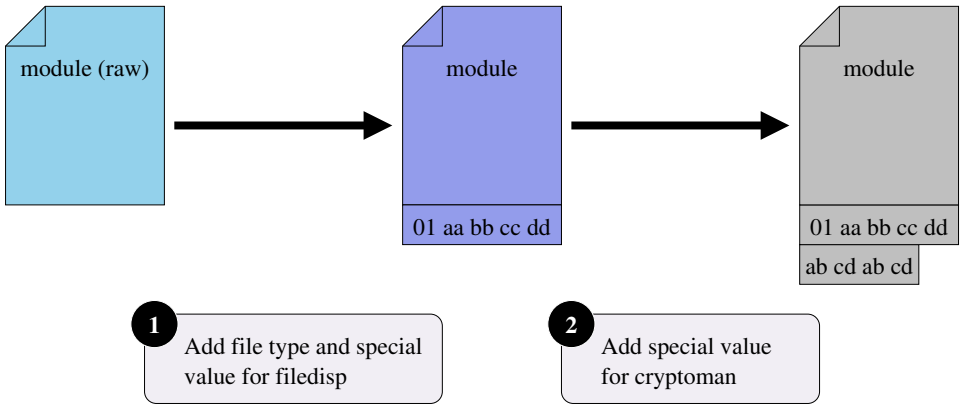


Figure: Module encryption

Module encryption

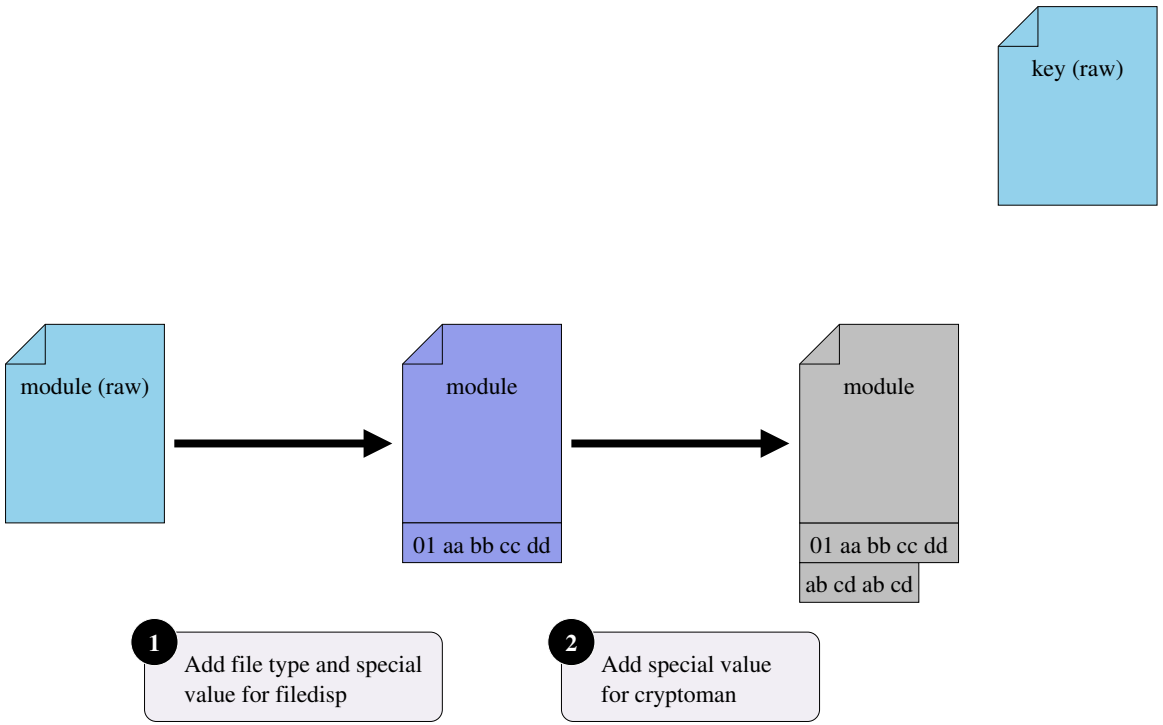


Figure: Module encryption

Module encryption

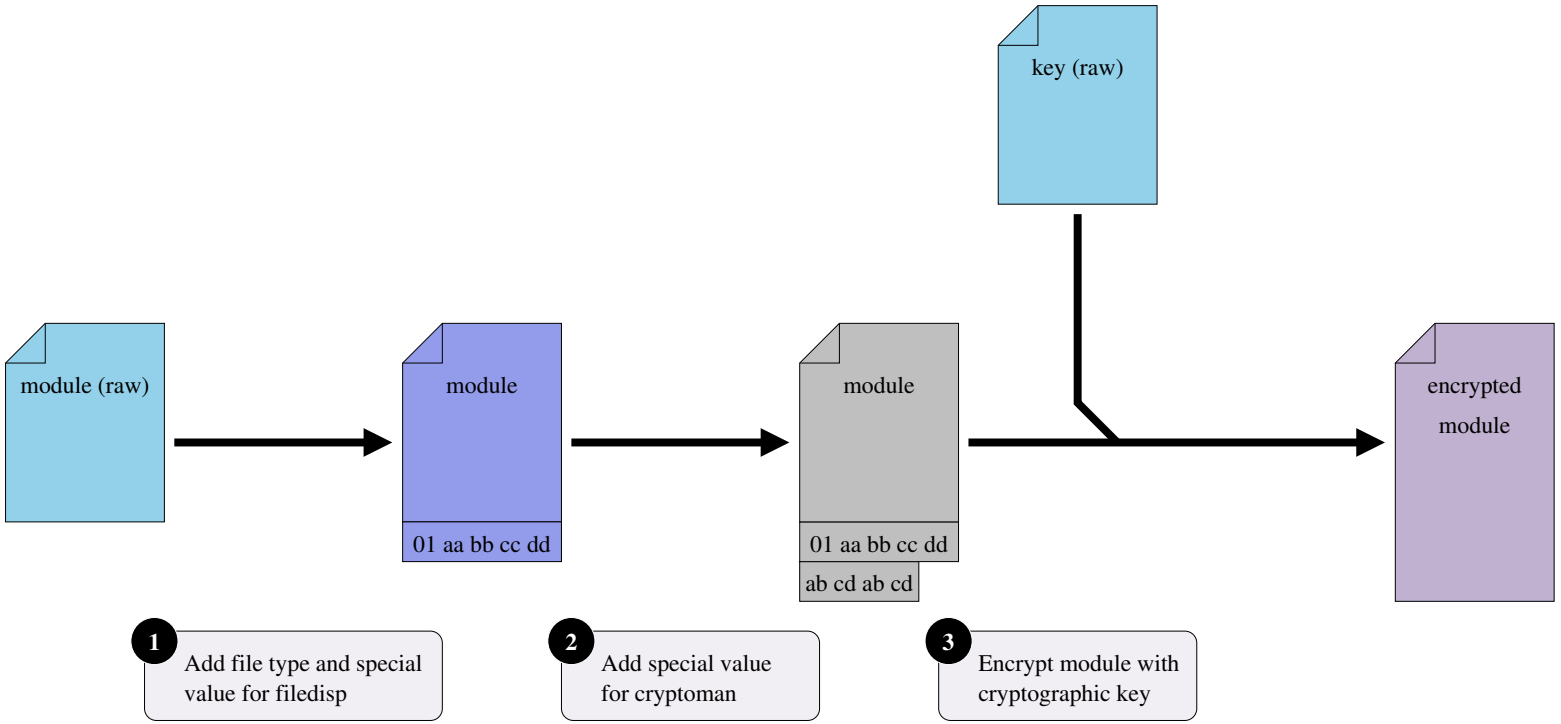


Figure: Module encryption

Module encryption

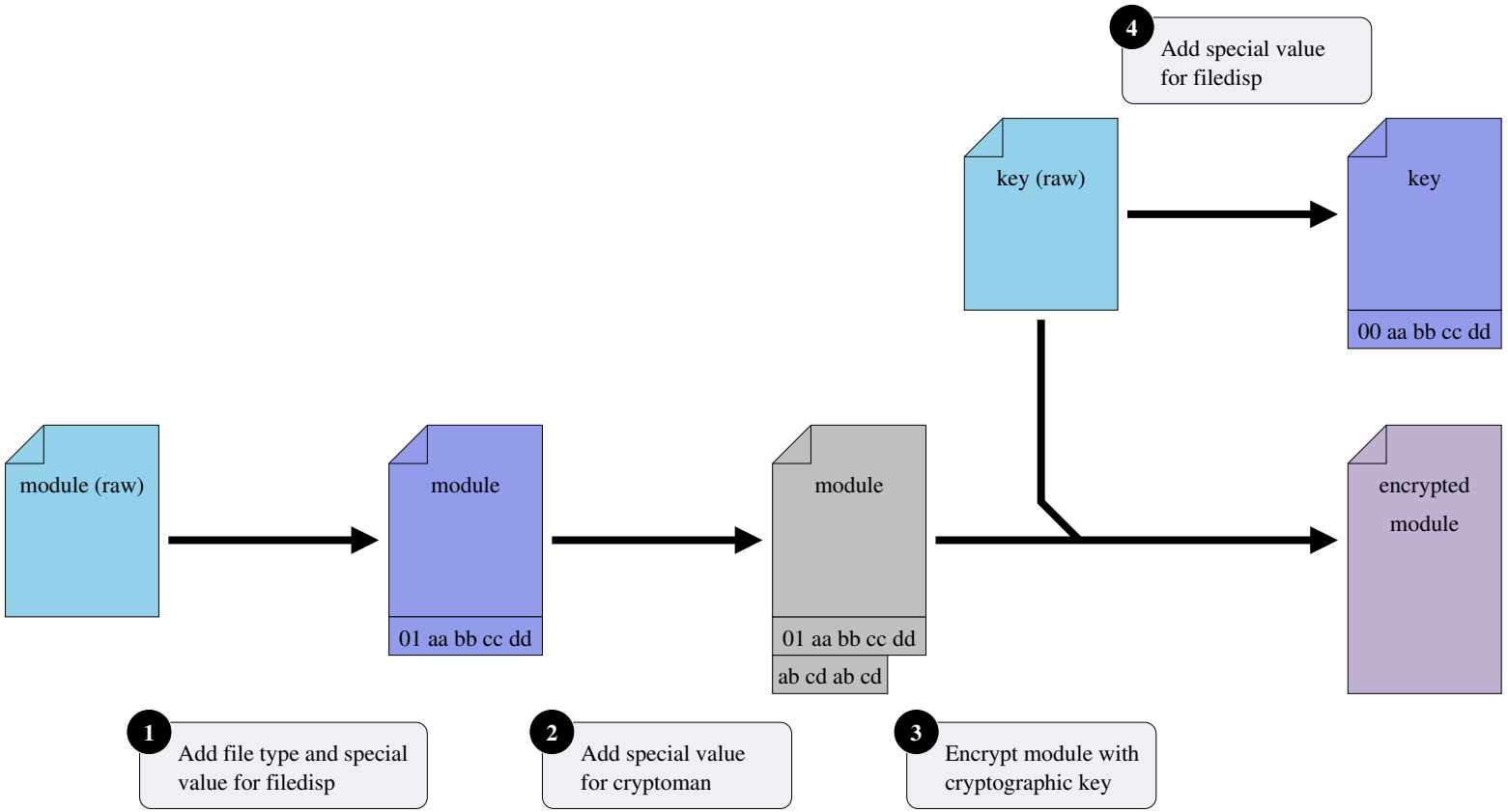


Figure: Module encryption

Module encryption

Demo : Module encryption

Loading encrypted module

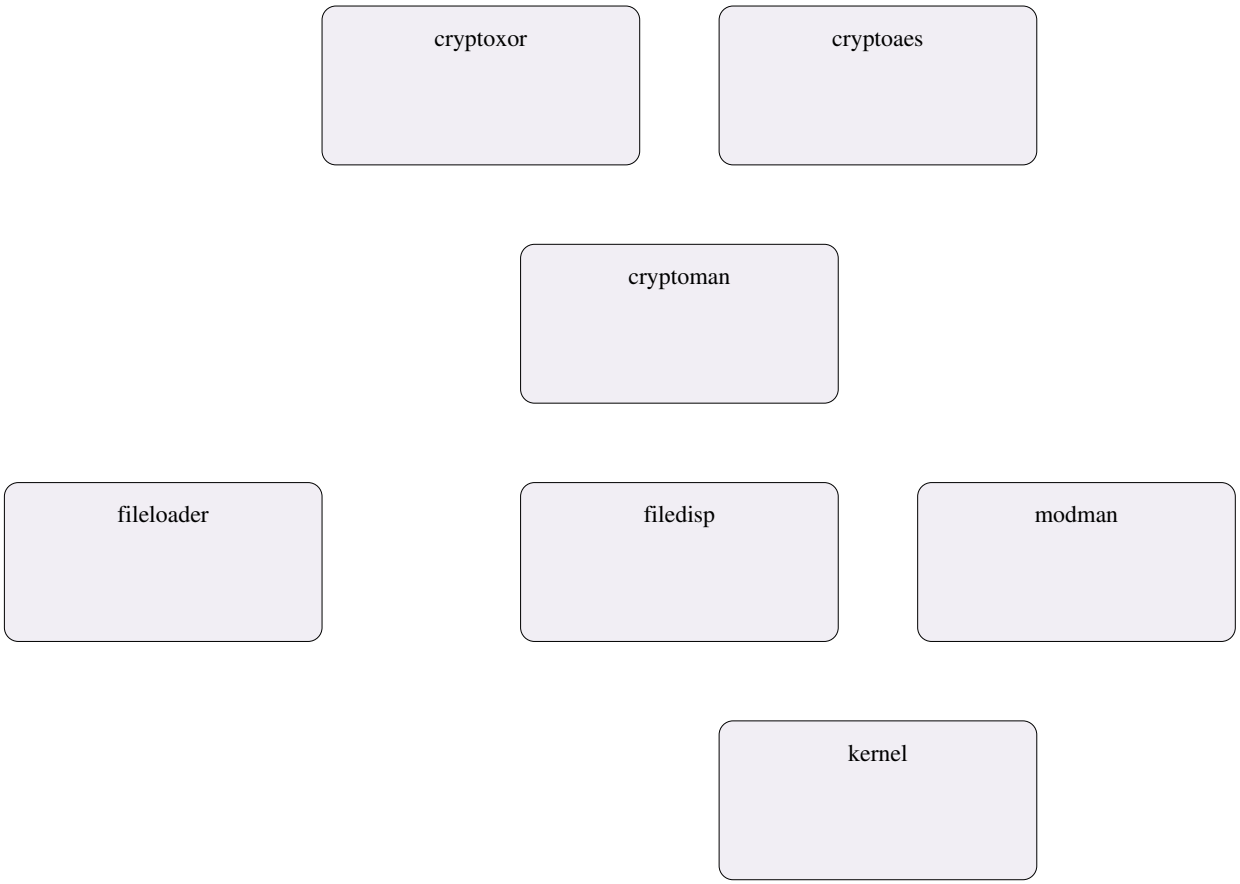


Figure: Loading an encrypted module

Loading encrypted module

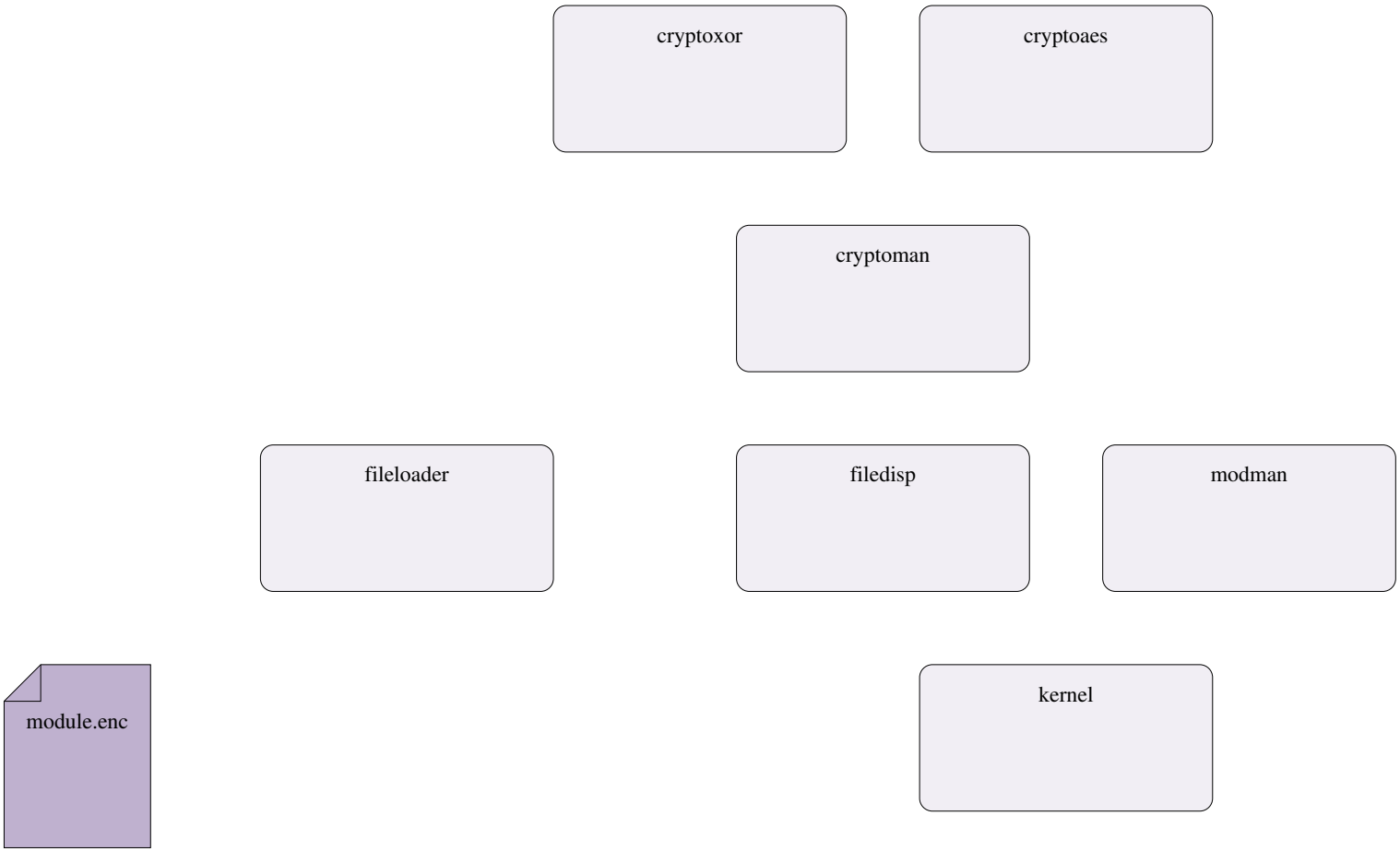


Figure: Loading an encrypted module

Loading encrypted module

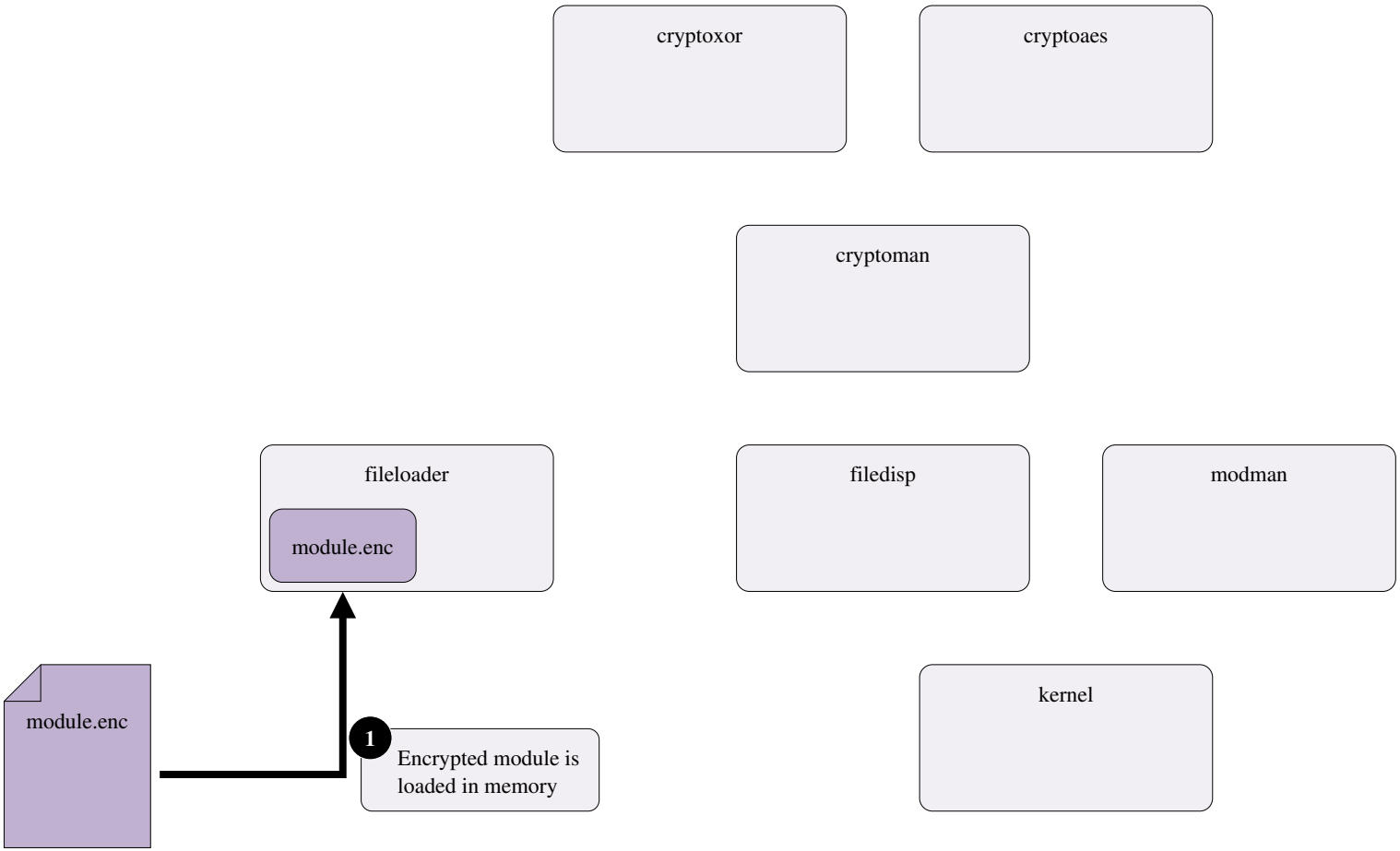


Figure: Loading an encrypted module

Loading encrypted module

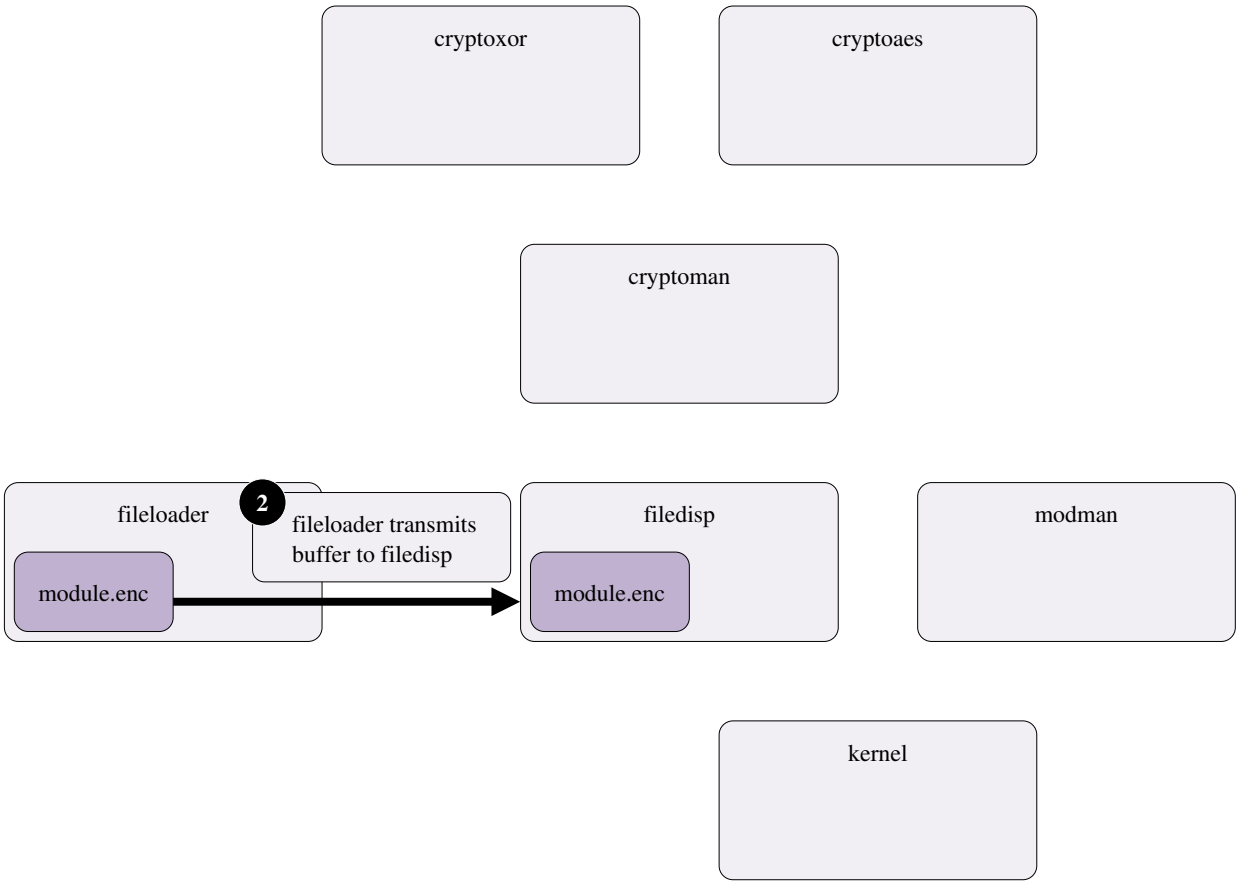


Figure: Loading an encrypted module

Loading encrypted module

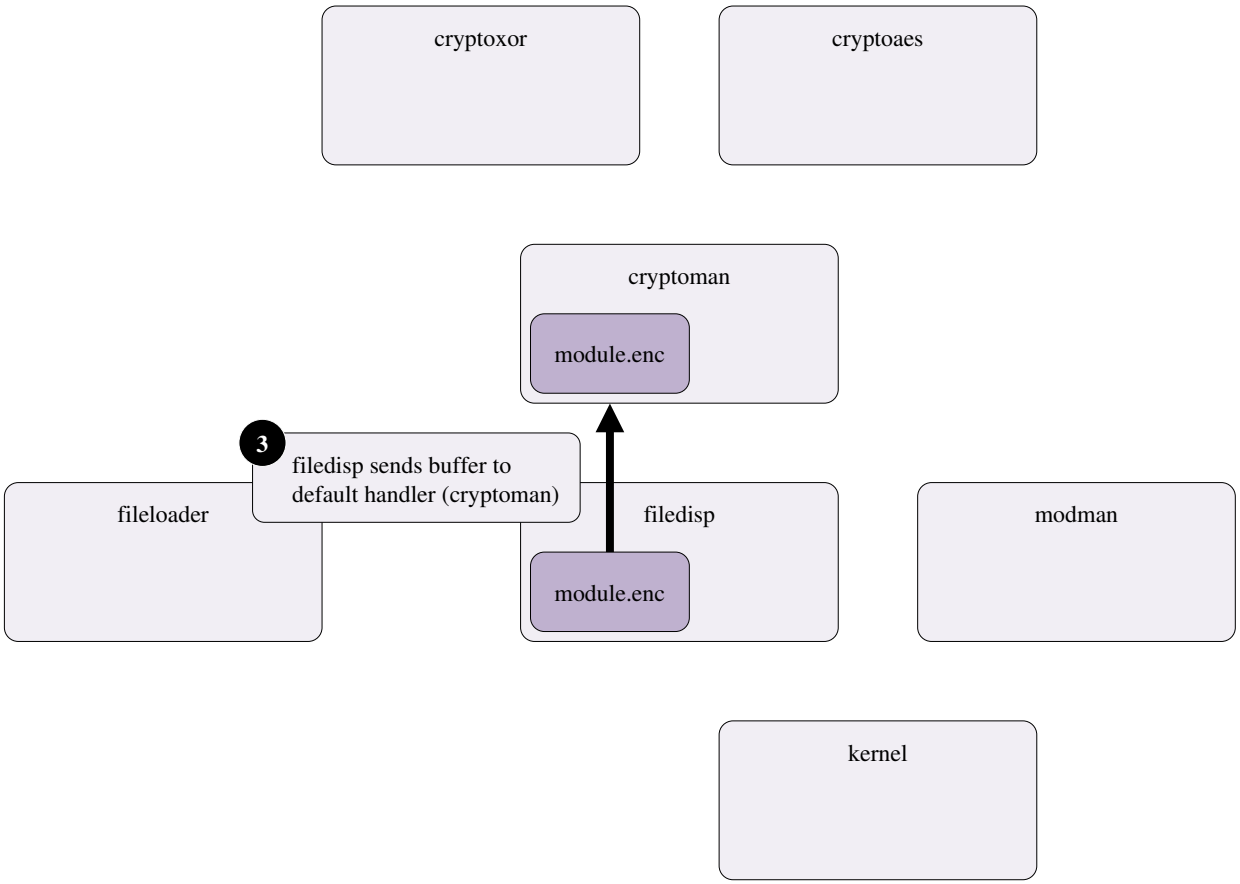


Figure: Loading an encrypted module

Loading encrypted module

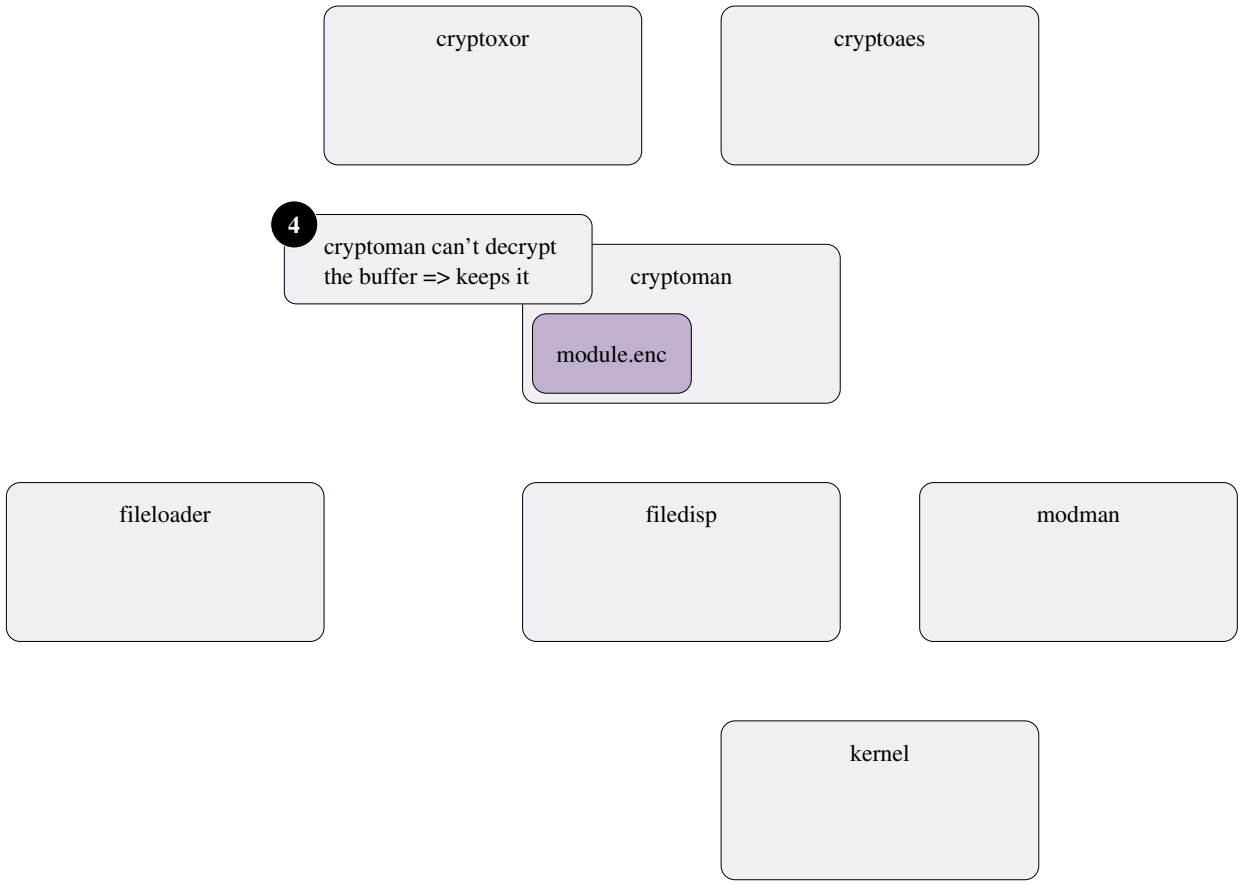


Figure: Loading an encrypted module

Loading encrypted module

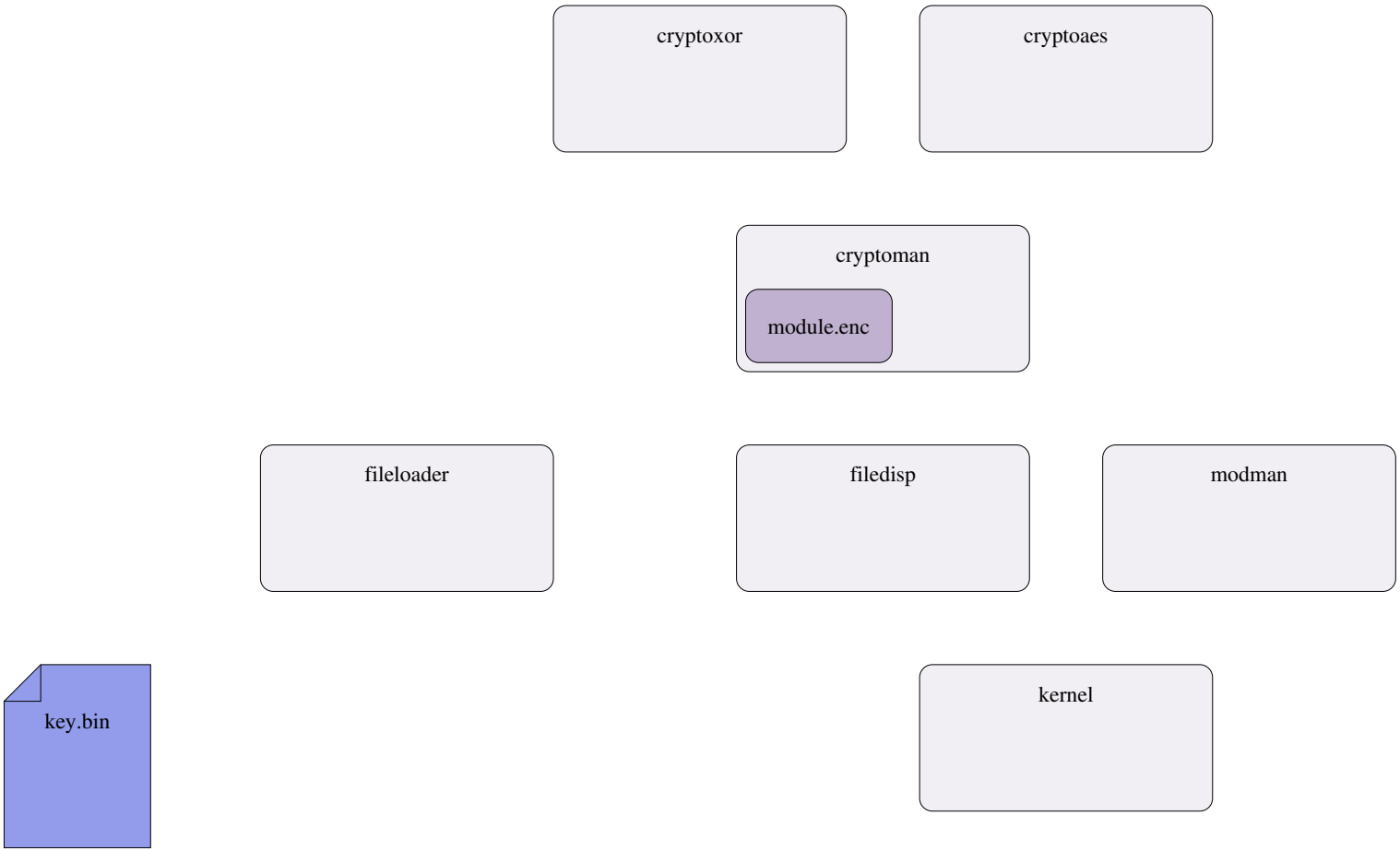


Figure: Loading an encrypted module

Loading encrypted module

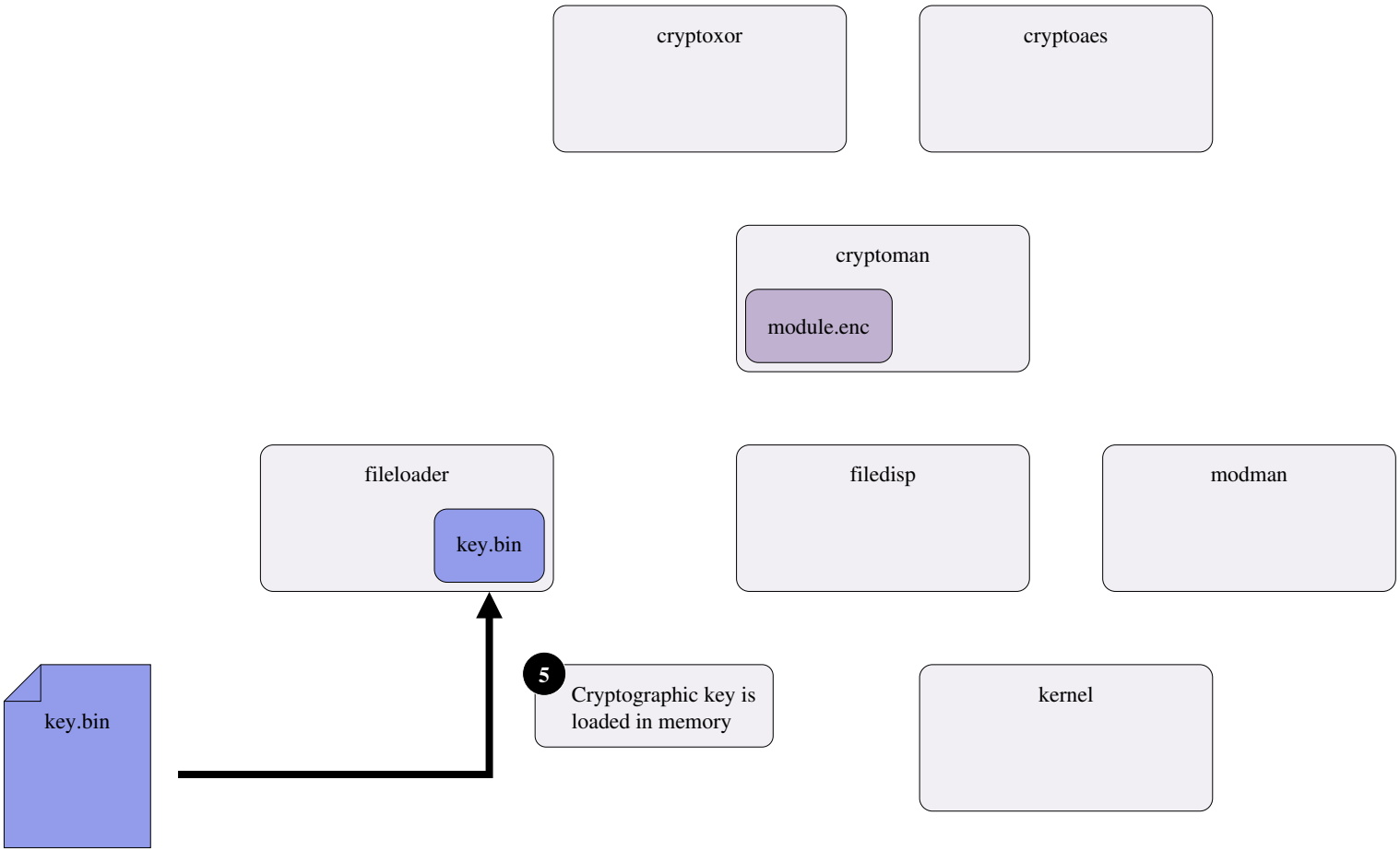


Figure: Loading an encrypted module

Loading encrypted module

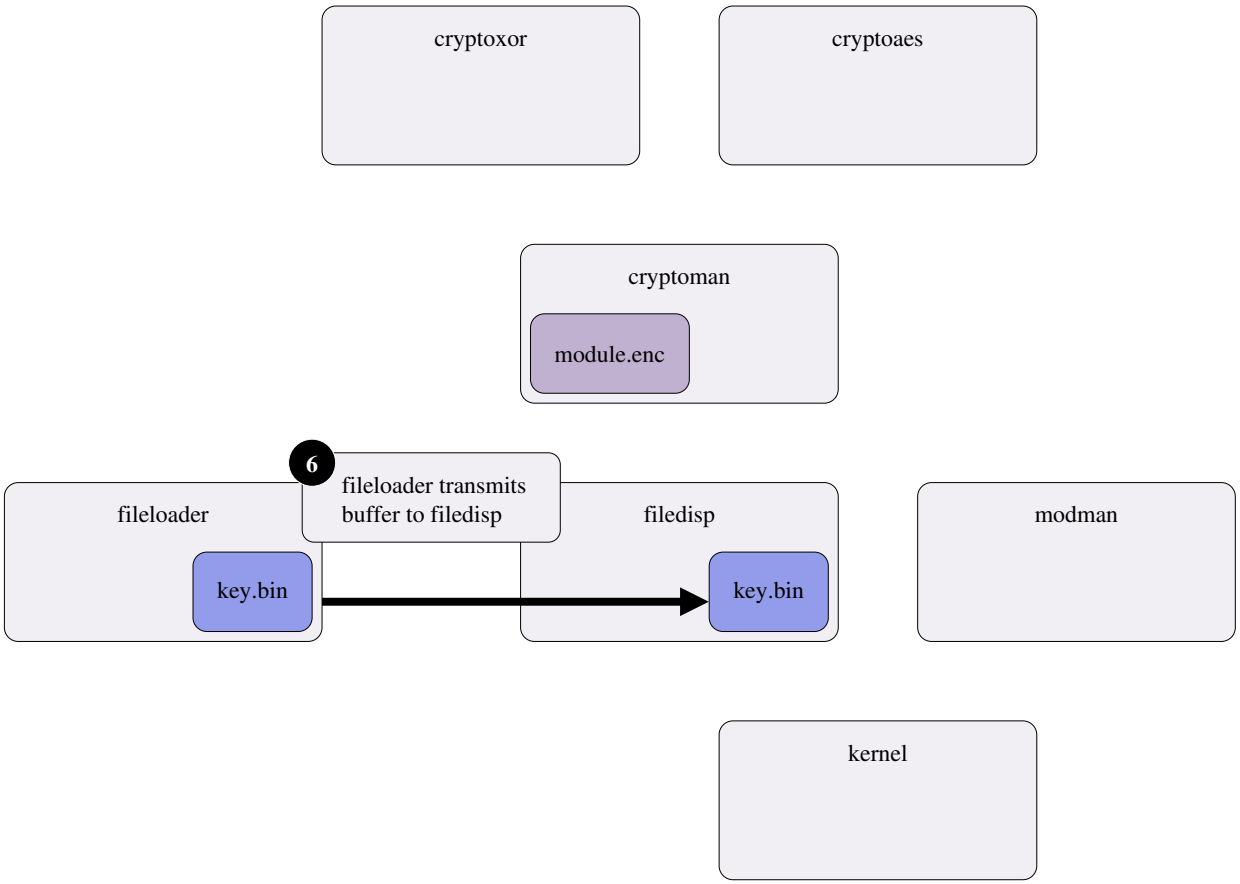


Figure: Loading an encrypted module

Loading encrypted module

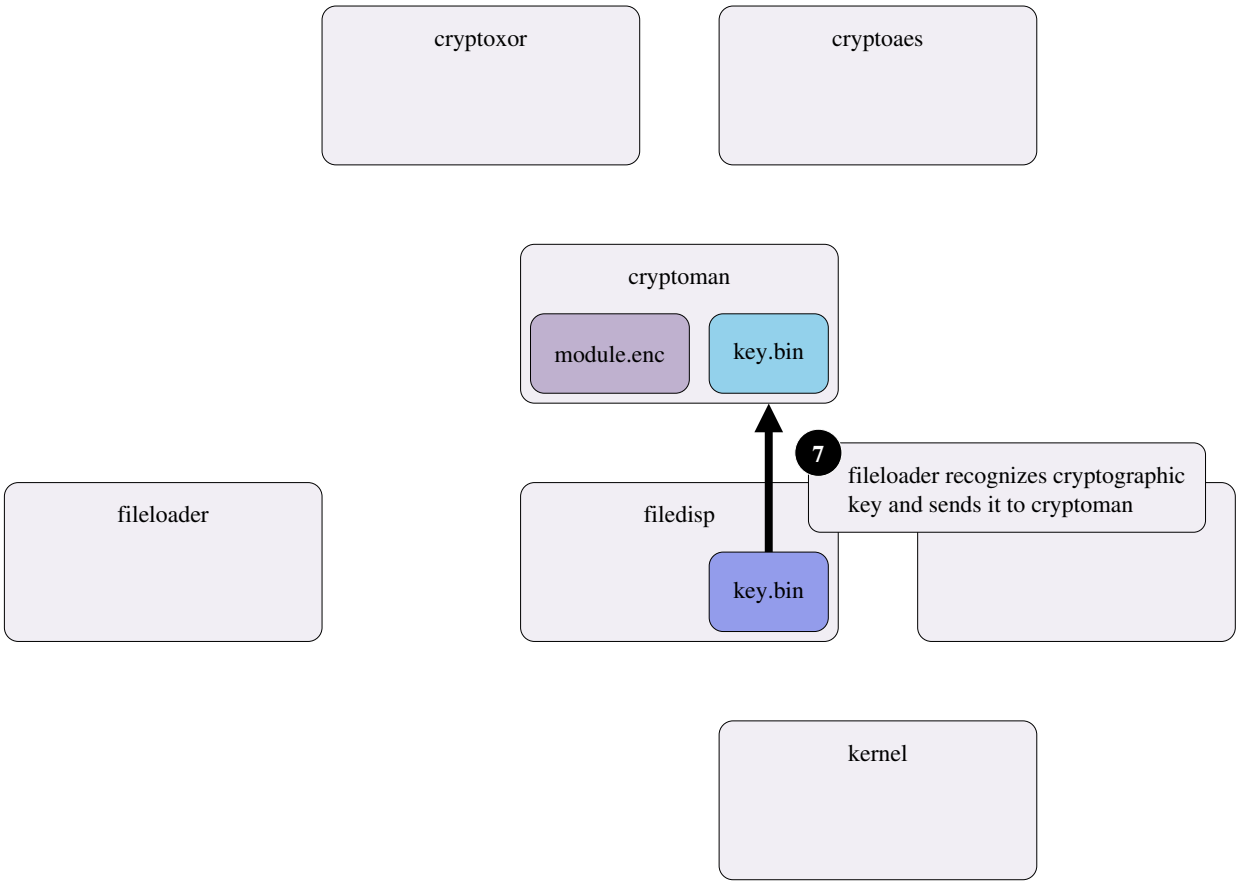


Figure: Loading an encrypted module

Loading encrypted module

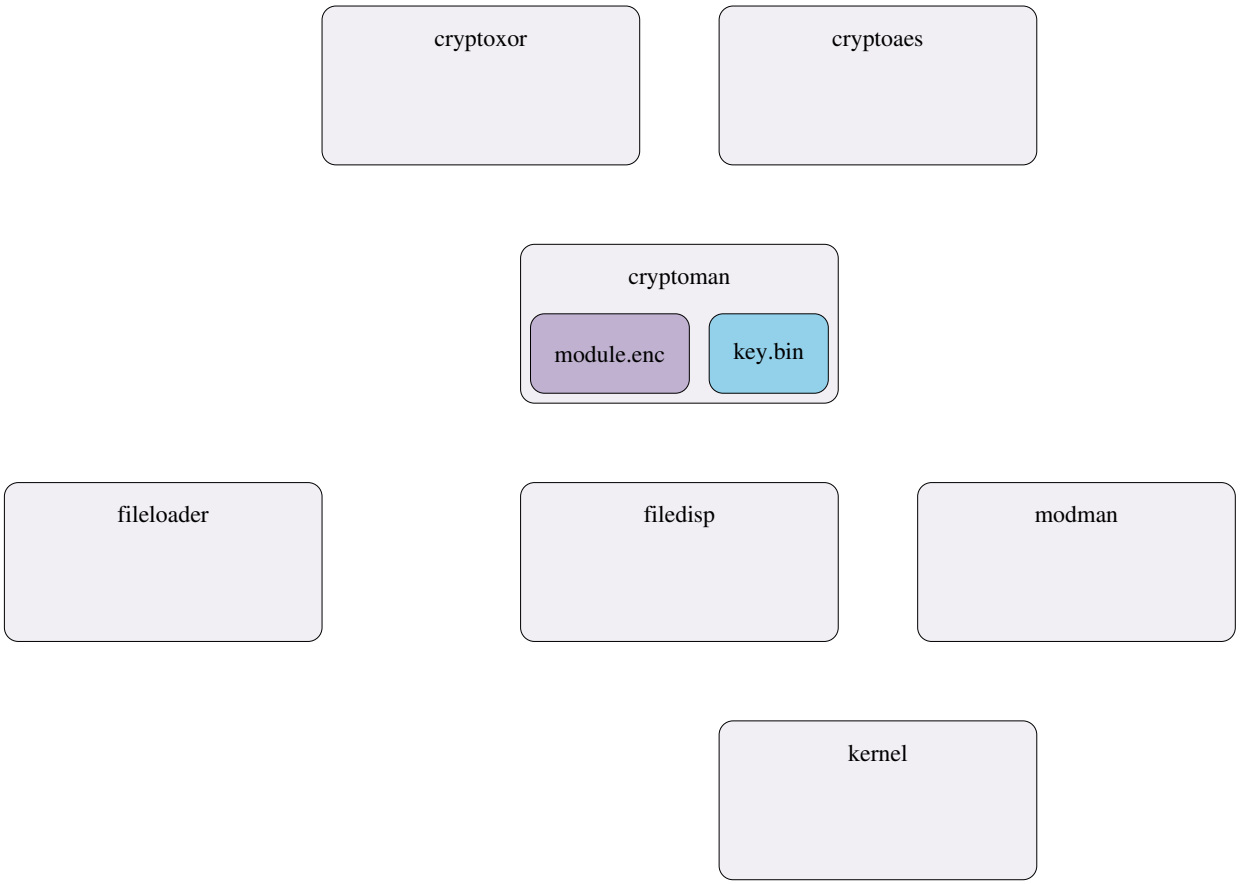


Figure: Loading an encrypted module

Loading encrypted module

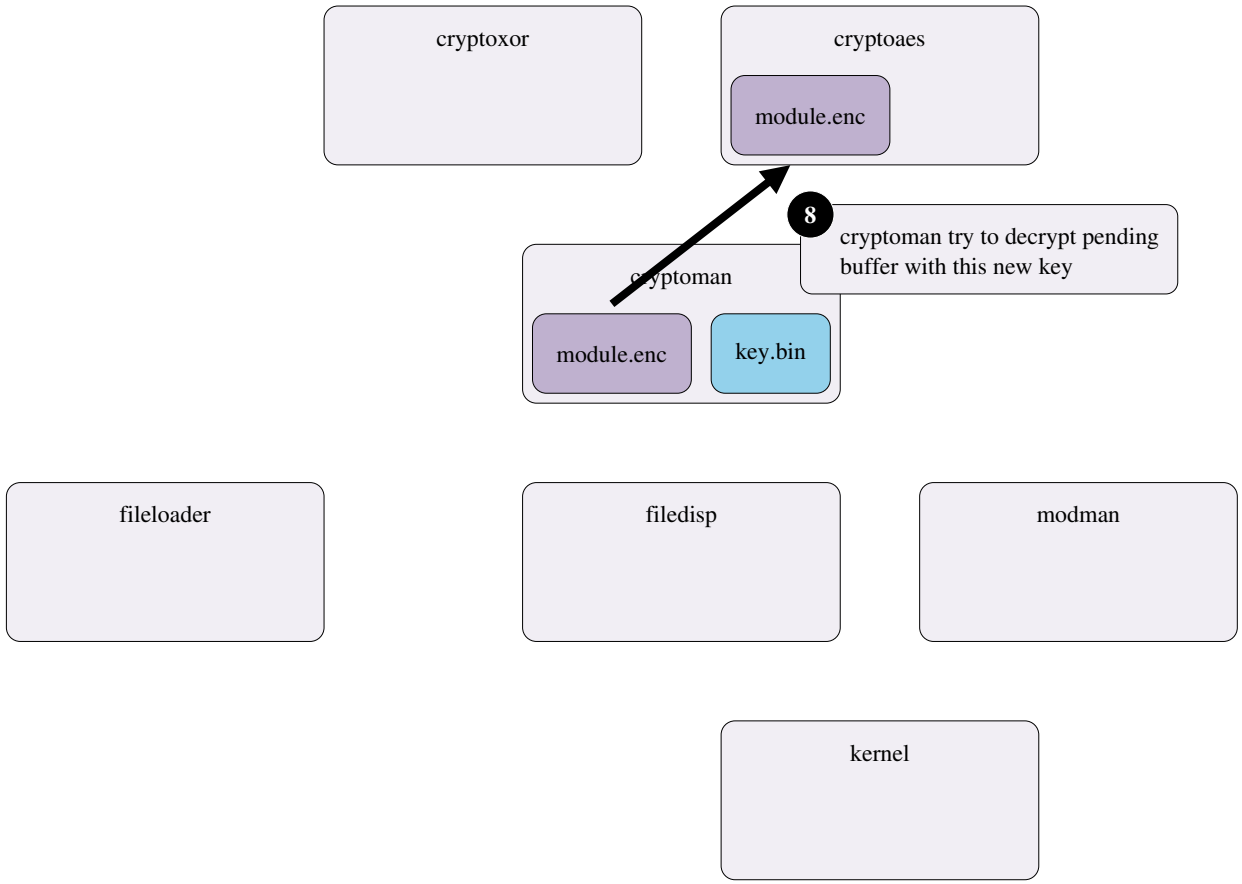


Figure: Loading an encrypted module

Loading encrypted module

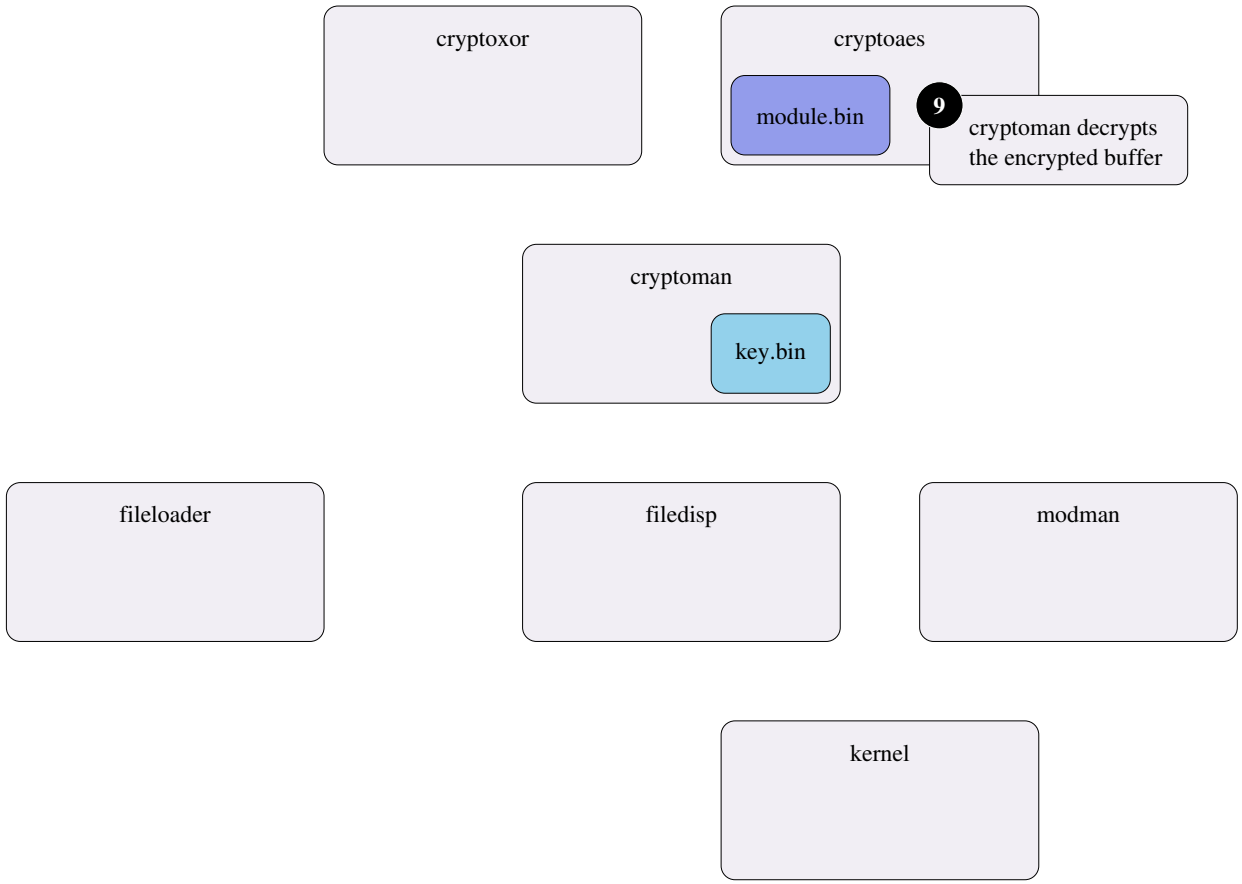


Figure: Loading an encrypted module

Loading encrypted module

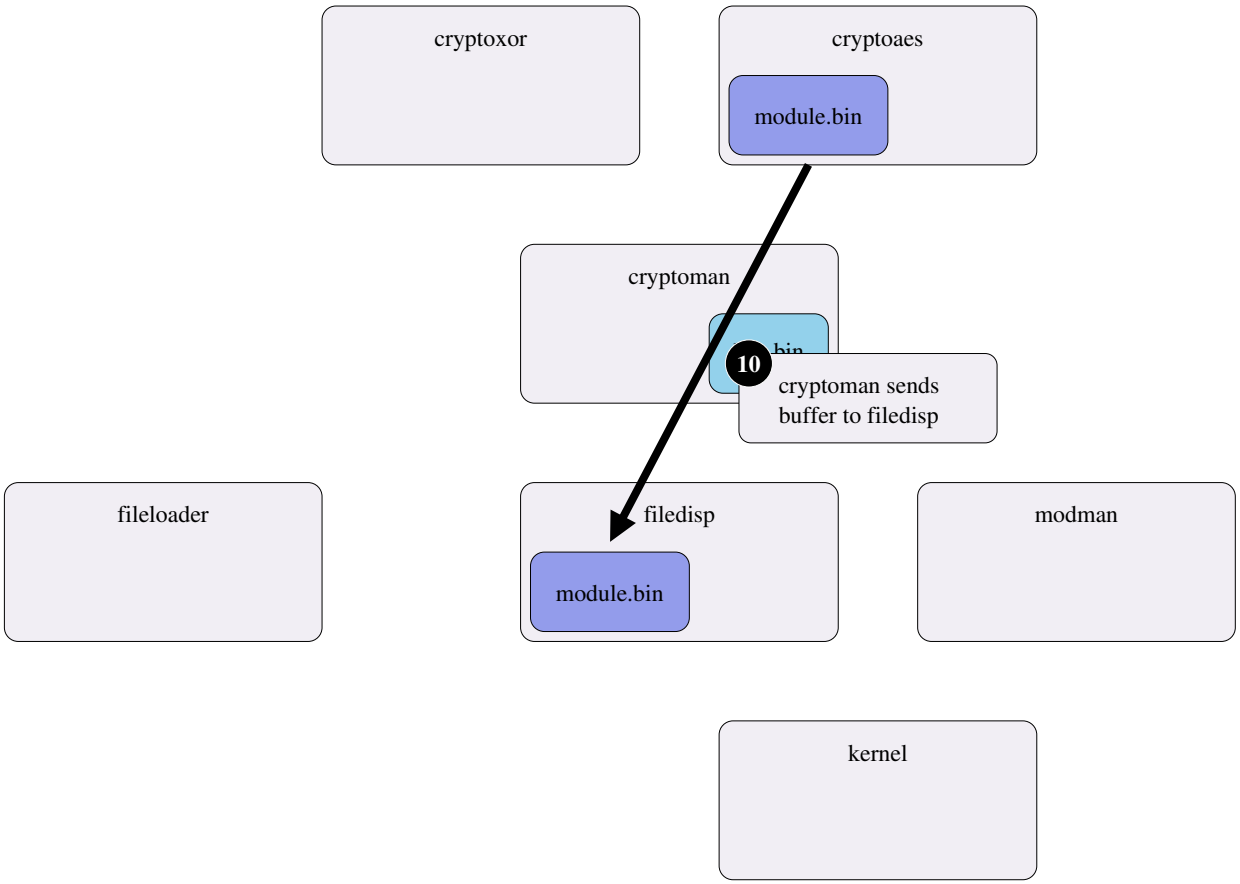


Figure: Loading an encrypted module

Loading encrypted module

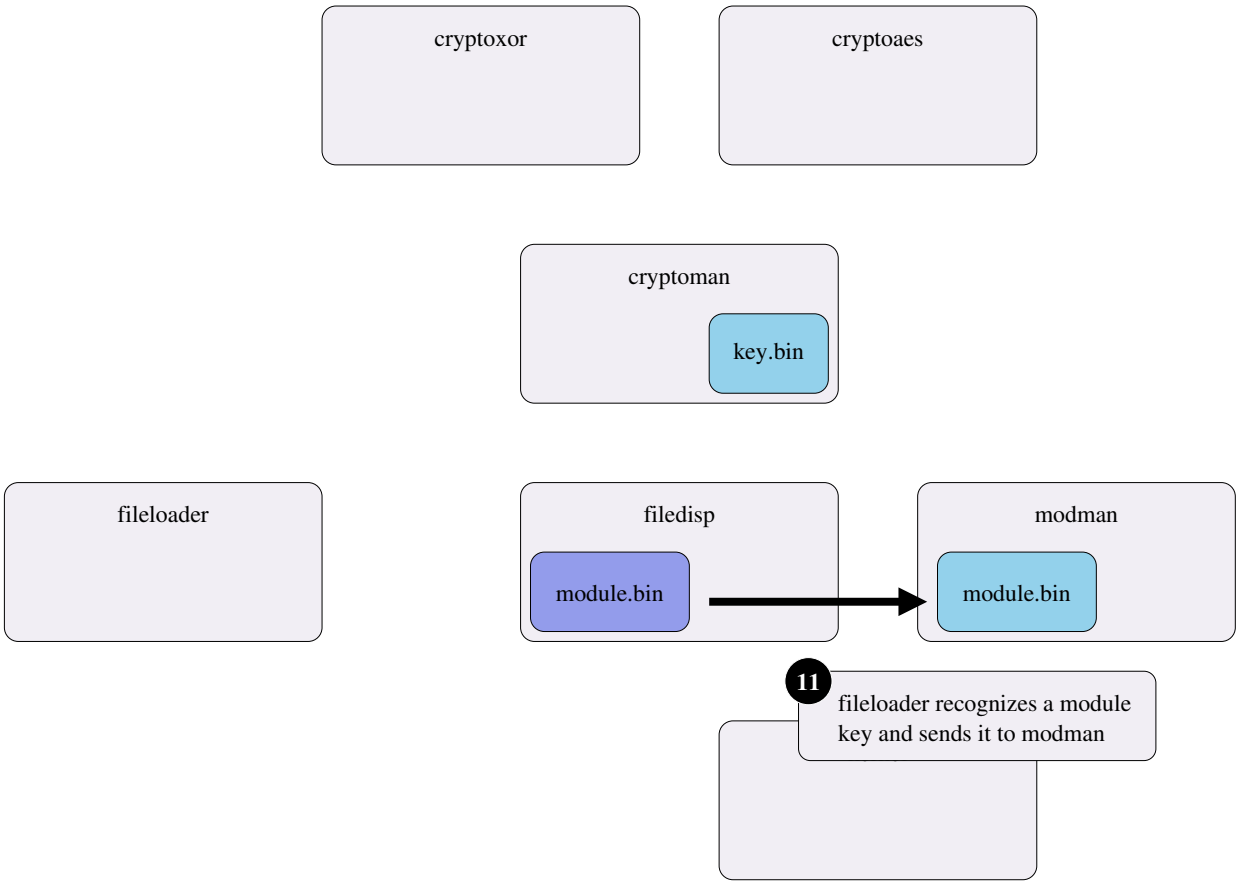


Figure: Loading an encrypted module

Loading encrypted module

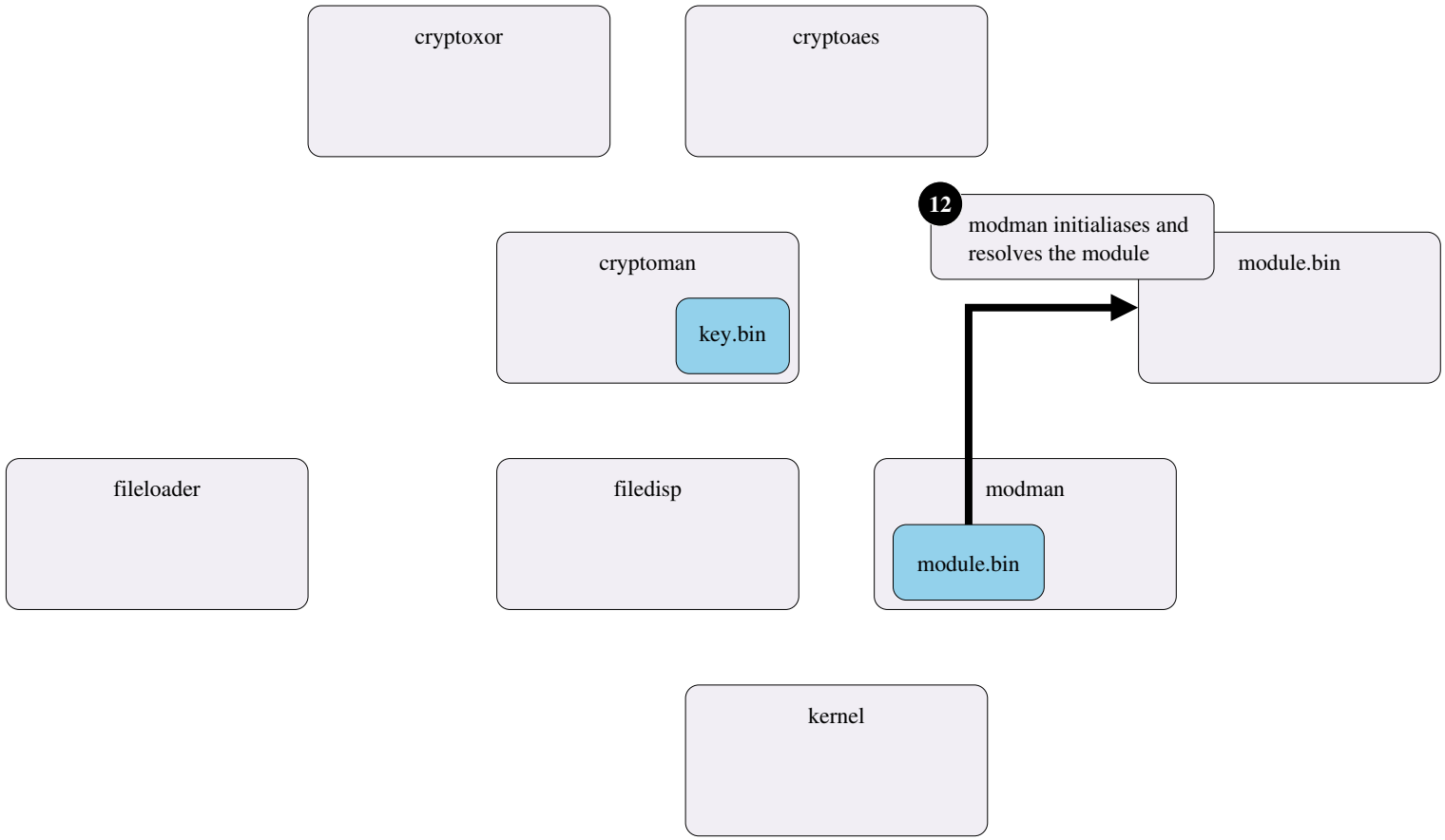


Figure: Loading an encrypted module

Loading encrypted module

Demo : Loading encrypted module

Plan

Plan

Demonstration 3: rvshell

—

Presentation of rvshell

Presentation of rvshell - 1

- "rvshell" is a simple reverse shell: backdoor that establishes a connection between a "cmd" process and a remote server
- backdoor compound of two layers:
  - the network layer that establishes the communication with the server
  - the application layer that creates the "cmd" process and uses the services exposed by the network layer

Presentation of RvShell - 2

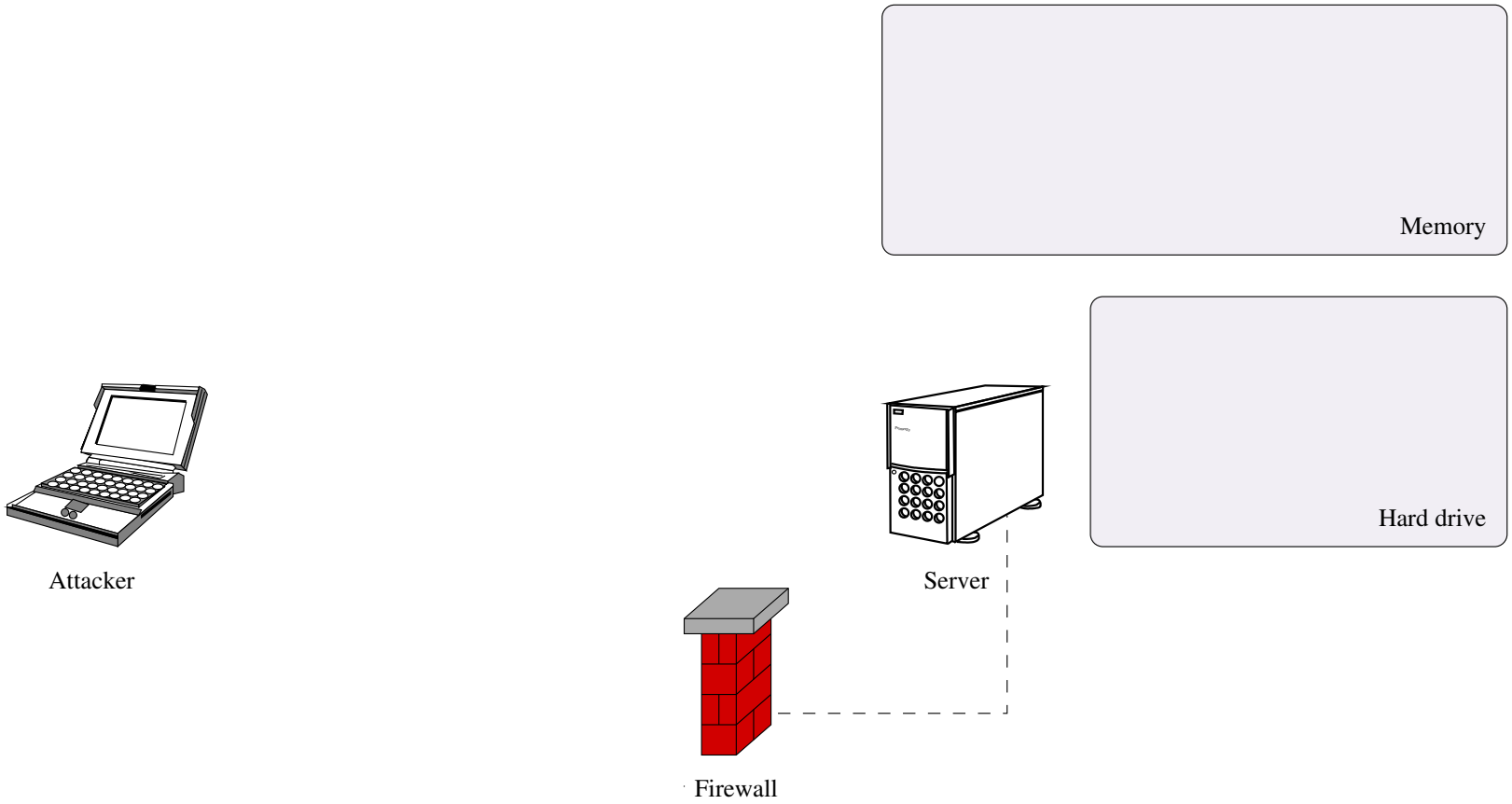


Figure: Working principle of RvShell

Presentation of RvShell - 2

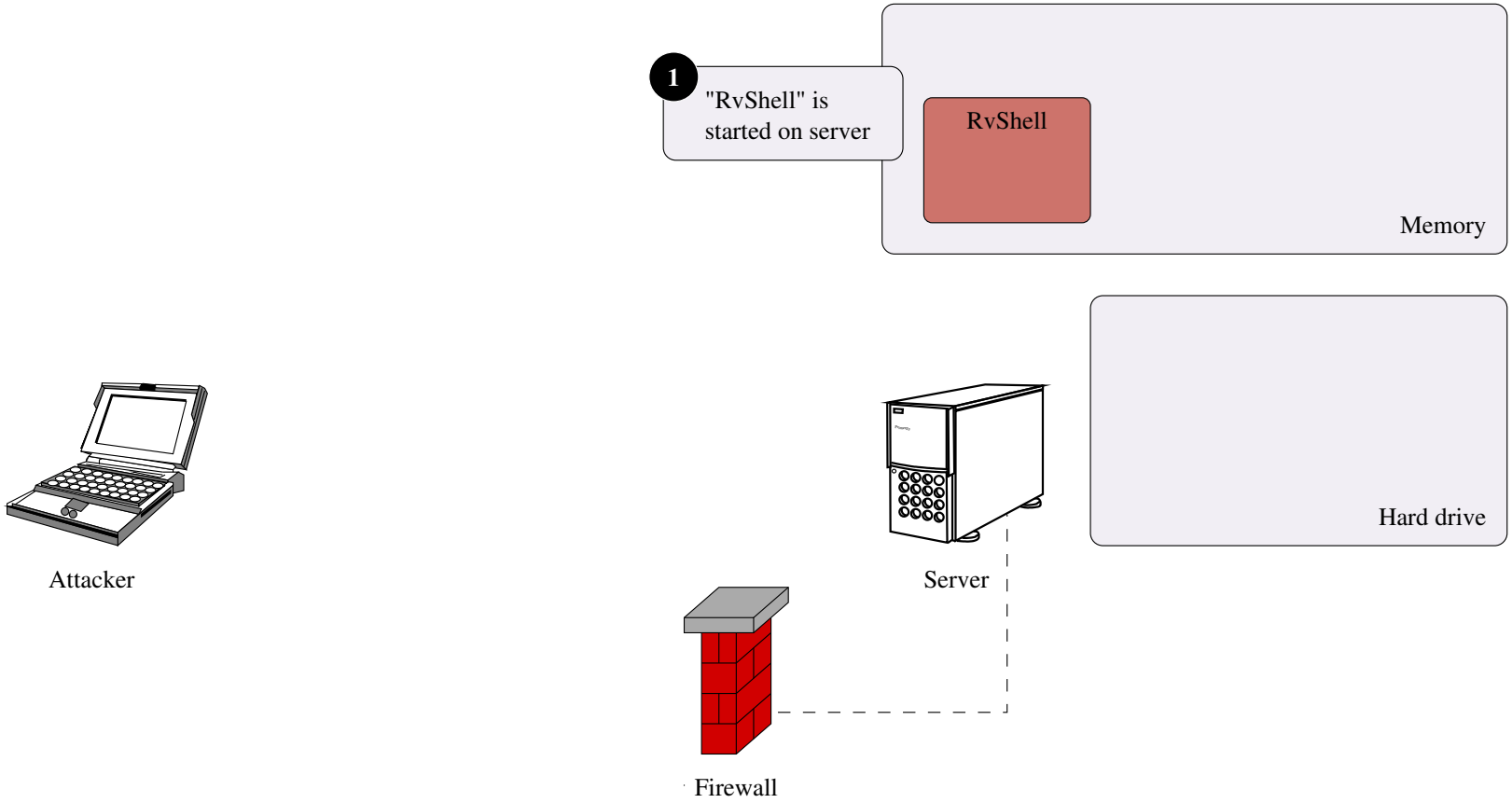


Figure: Working principle of RvShell

Presentation of RvShell - 2

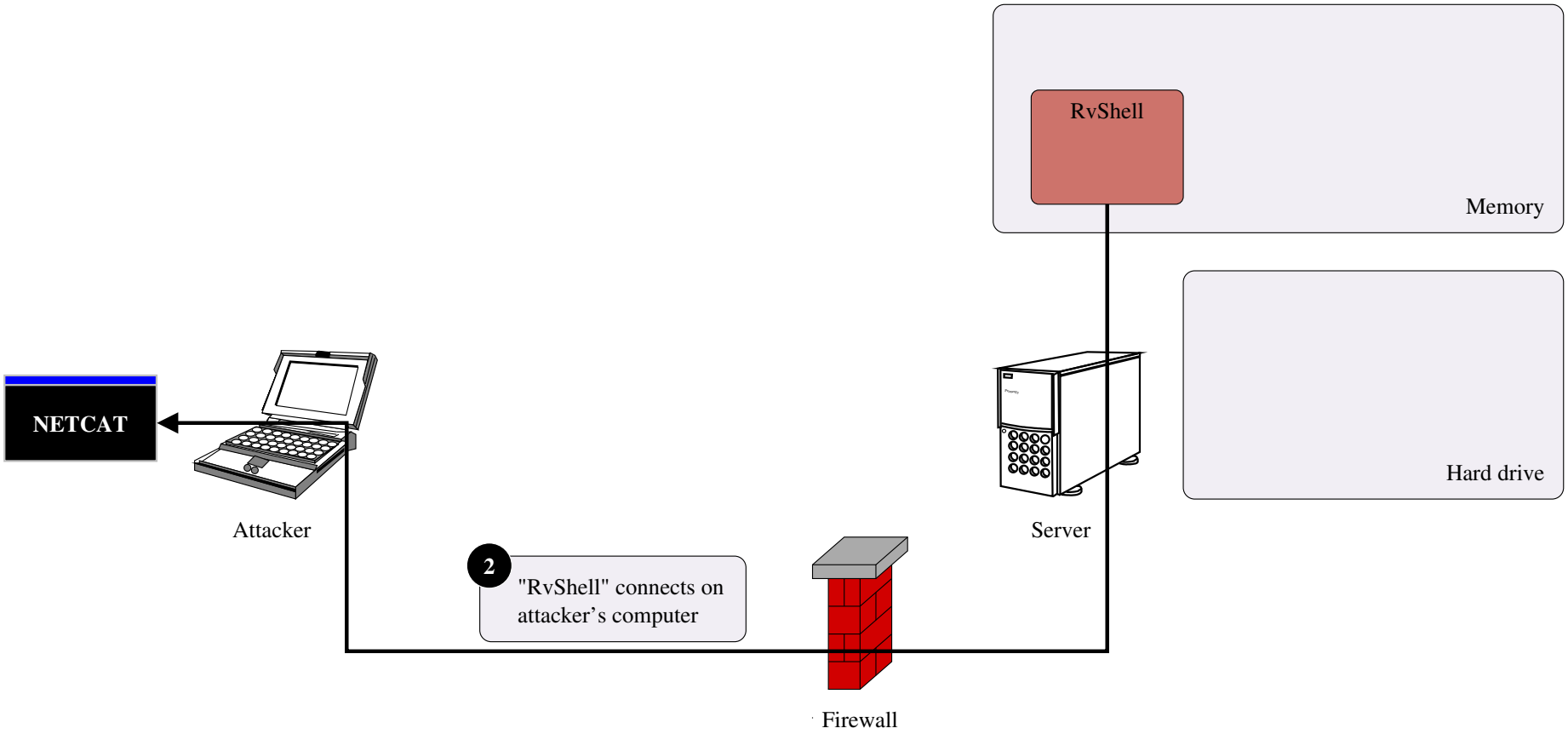


Figure: Working principle of RvShell

Presentation of RvShell - 2

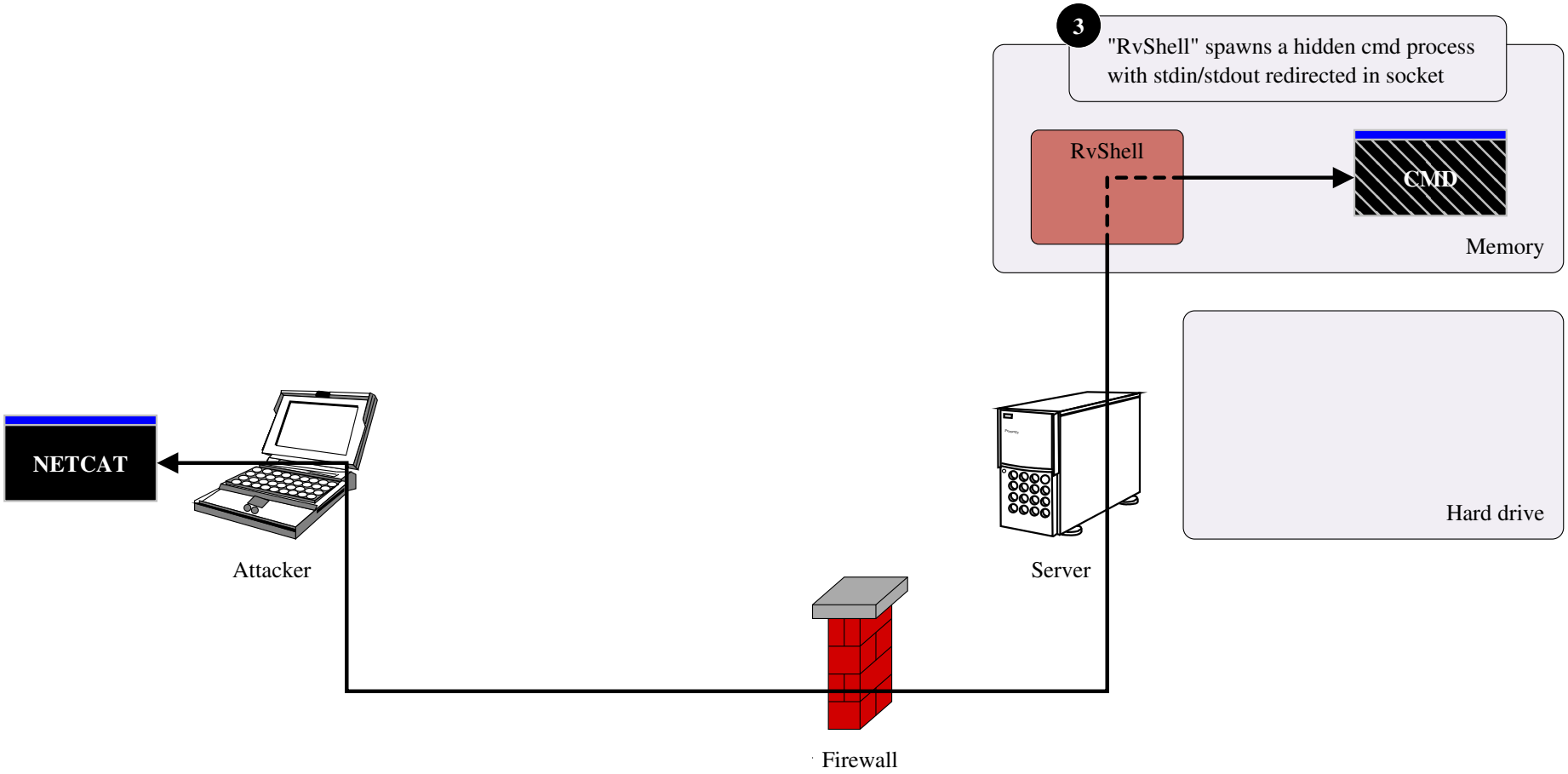


Figure: Working principle of RvShell

Implementation of rvshell

Two modules have been developed:

- "NtStackSmpl" implements the network layer and exports two functions:

```
BOOL OpenConnection(IN UINT uiServerAddressNt, IN USHORT usServerPortNt, OUT SOCKET * pSock);
BOOL CloseConnection(IN SOCKET sock);
```

- "RvShell" implements the application layer:
  - does not export any function
  - has an entry point, the function "ExecuteShell":
    - uses "OpenConnection" to open a TCP connection on the server
    - creates the "cmd" process

Generating RvShell as an executable

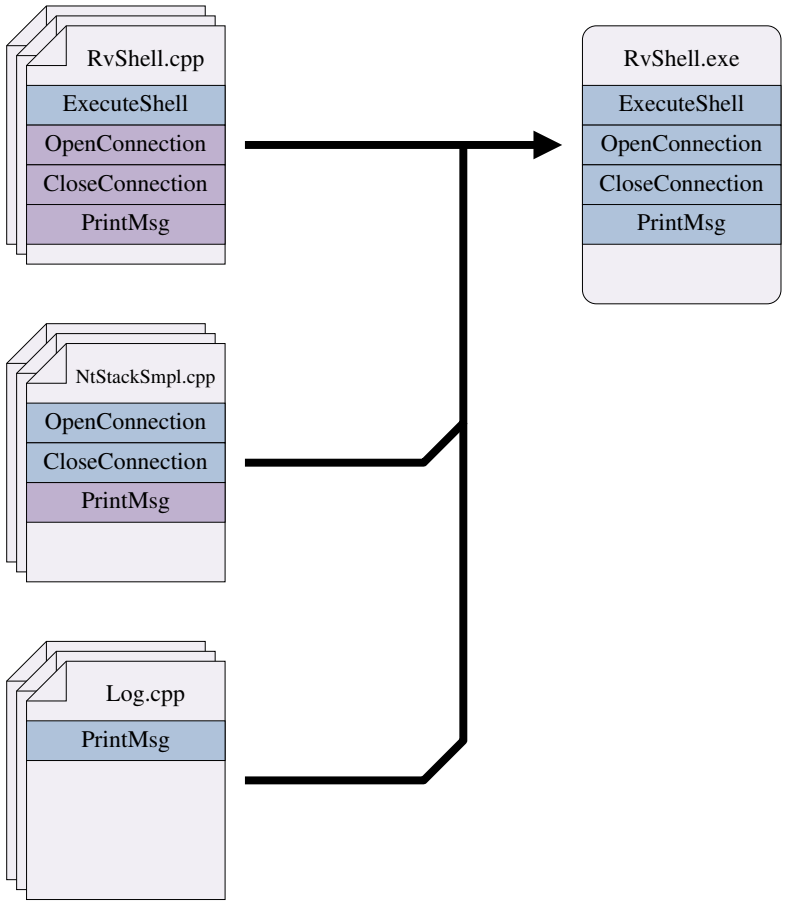
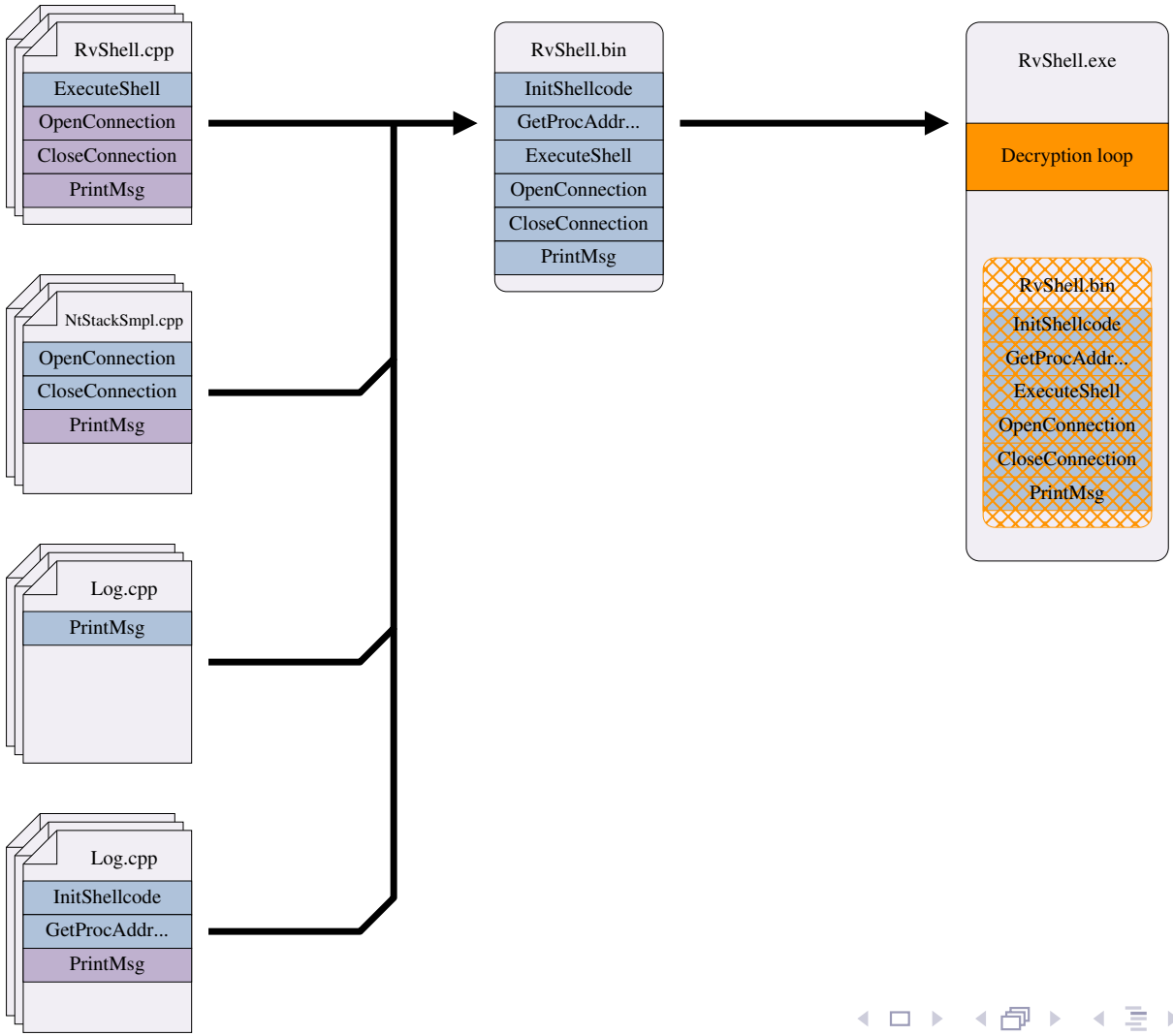


Figure: Result of the creation of the reverse shell as an executable

Generating a polymorphic rvshell - 2

"rvshell" is generated as a shellcode and then included in an executable that decrypts rvshell and jumps on it



Plan

Demonstration 3: rvshell

—

Simulation of an attack with RvShell

Context

Objective

Take control of a targeted computer with a backdoor (reverse shell)

Context

Objective

Take control of a targeted computer with a backdoor (reverse shell)

Context of the attack

Malicious payload must be protected against forensic analysis:

- malicious payload is transferred after encryption on targeted computer
- malicious payload is decrypted only in memory
- decryption code is introduced by another way

Principle of the attack

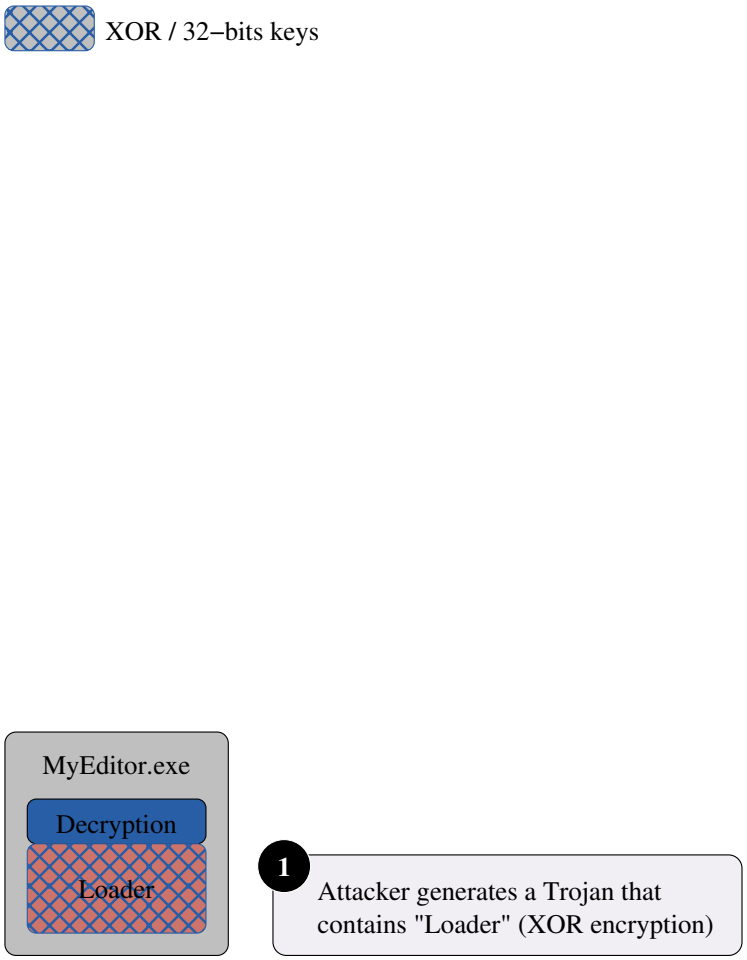


Figure: Principle of the attack with RvShell

Principle of the attack

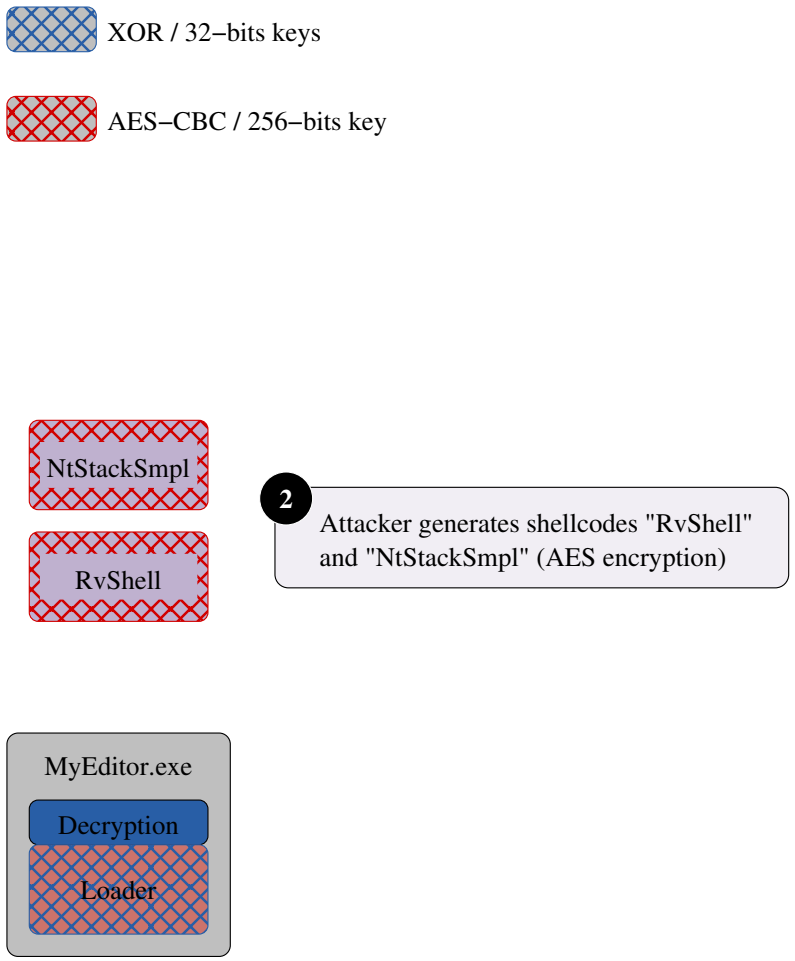


Figure: Principle of the attack with RvShell

Principle of the attack

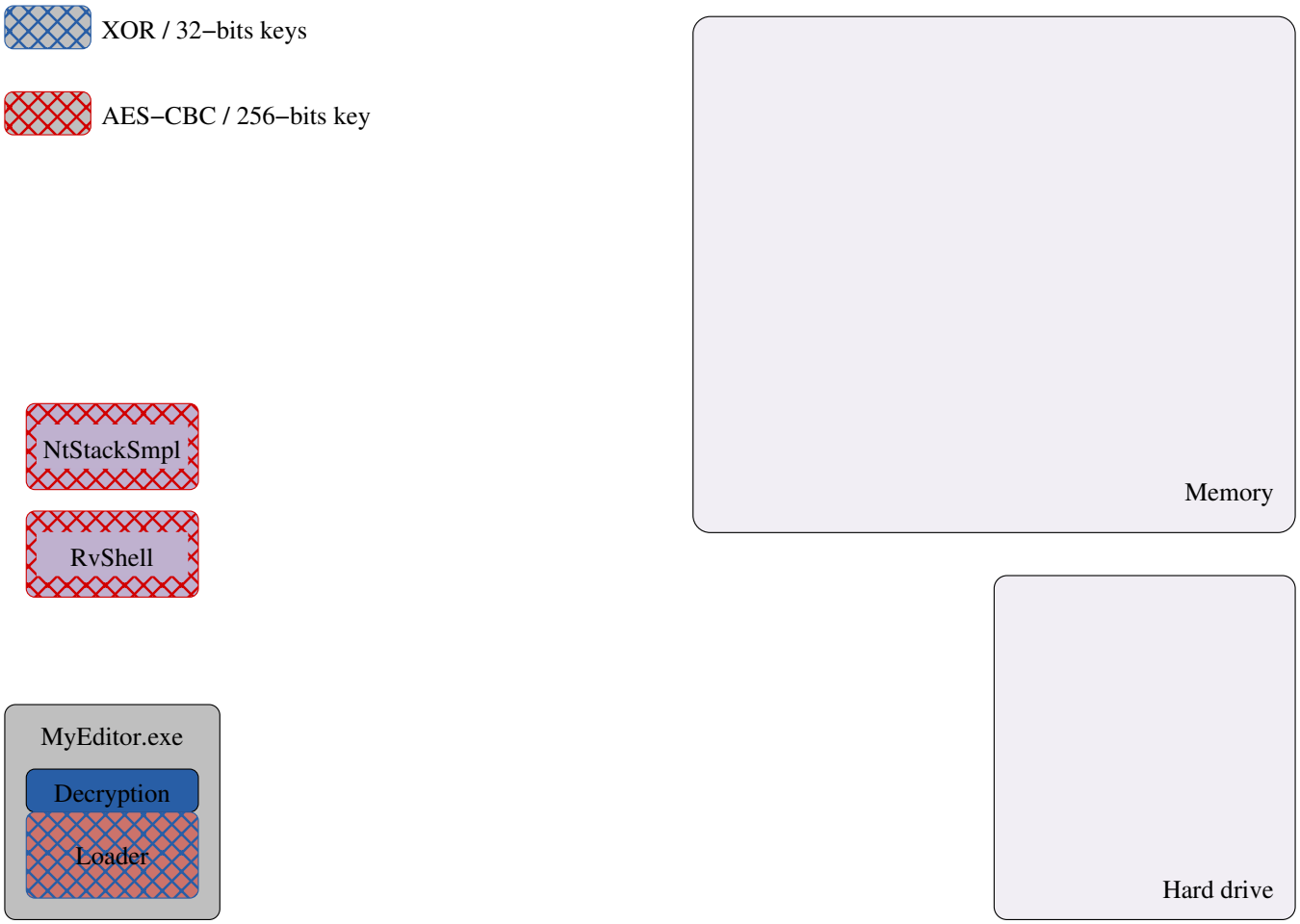


Figure: Principle of the attack with RvShell

Principle of the attack

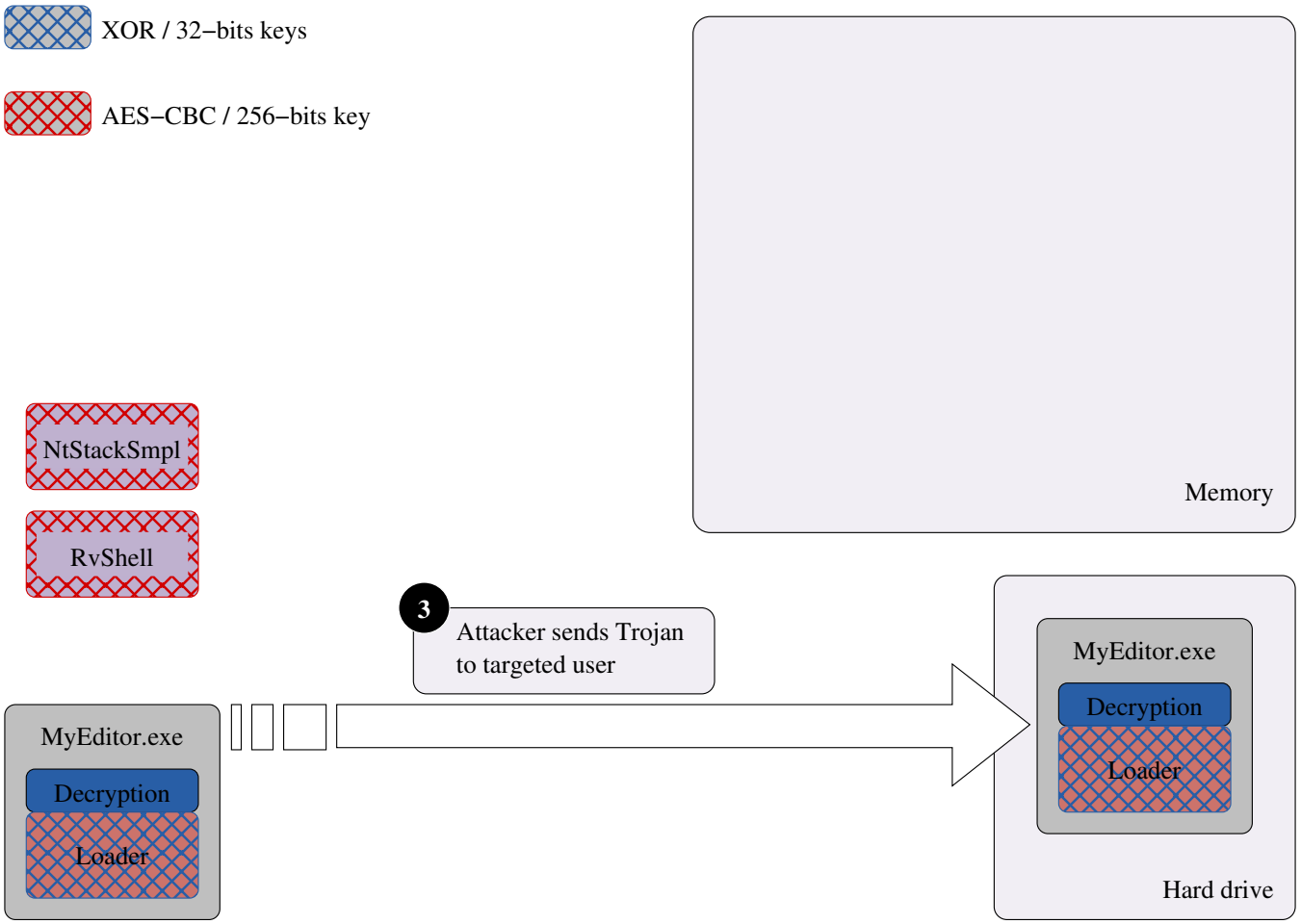


Figure: Principle of the attack with RvShell

Principle of the attack

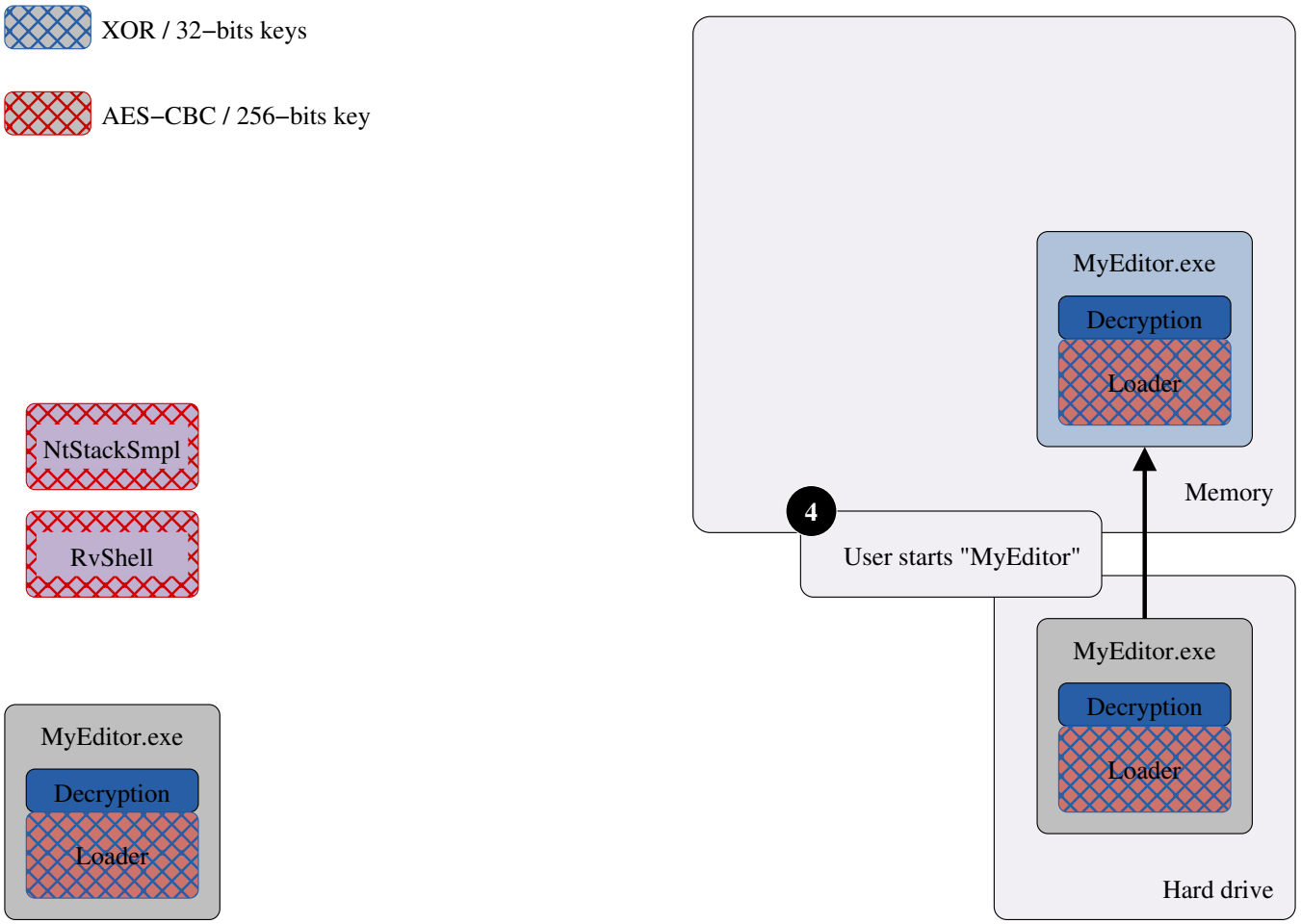


Figure: Principle of the attack with RvShell

Principle of the attack

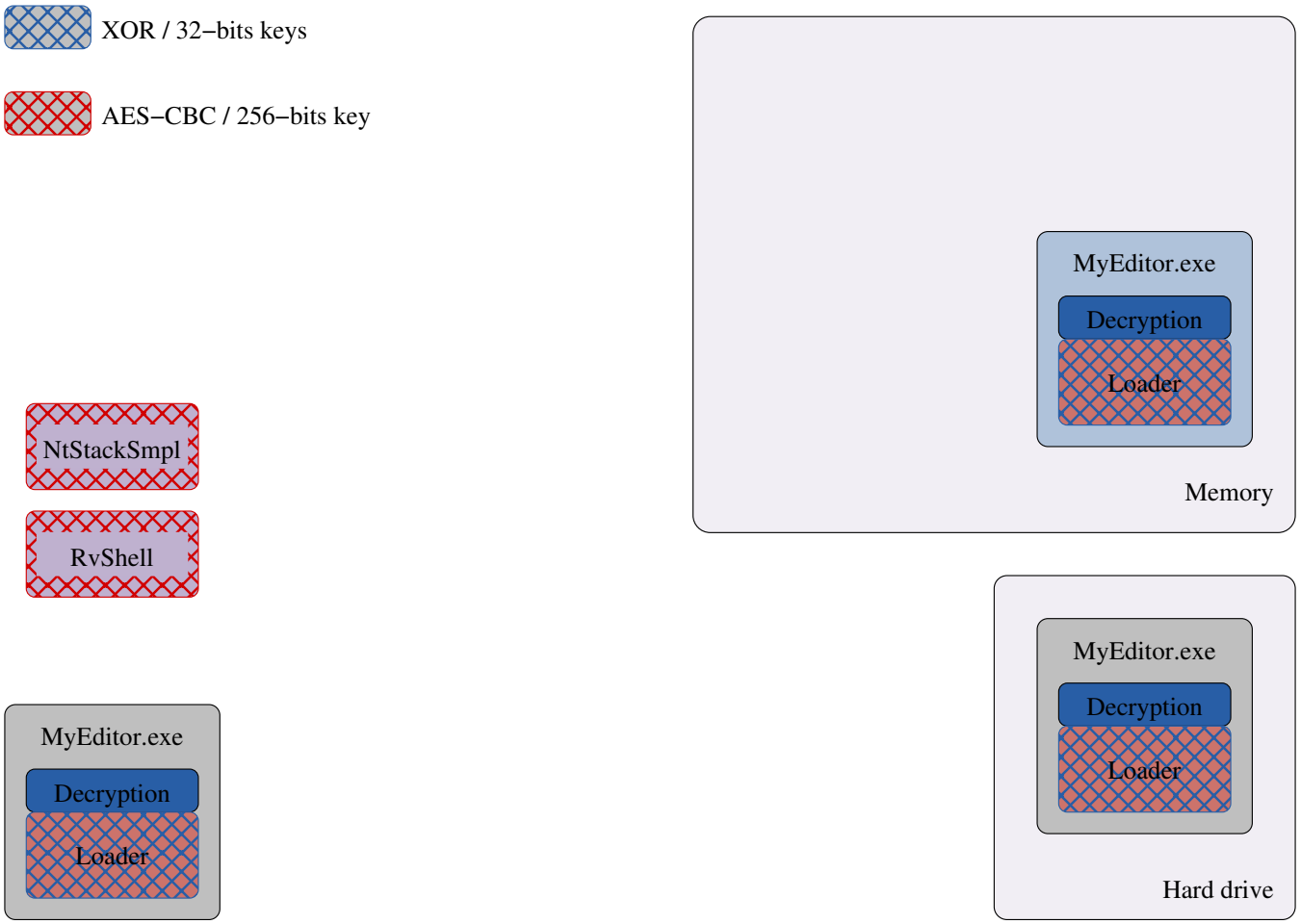


Figure: Principle of the attack with RvShell

Principle of the attack

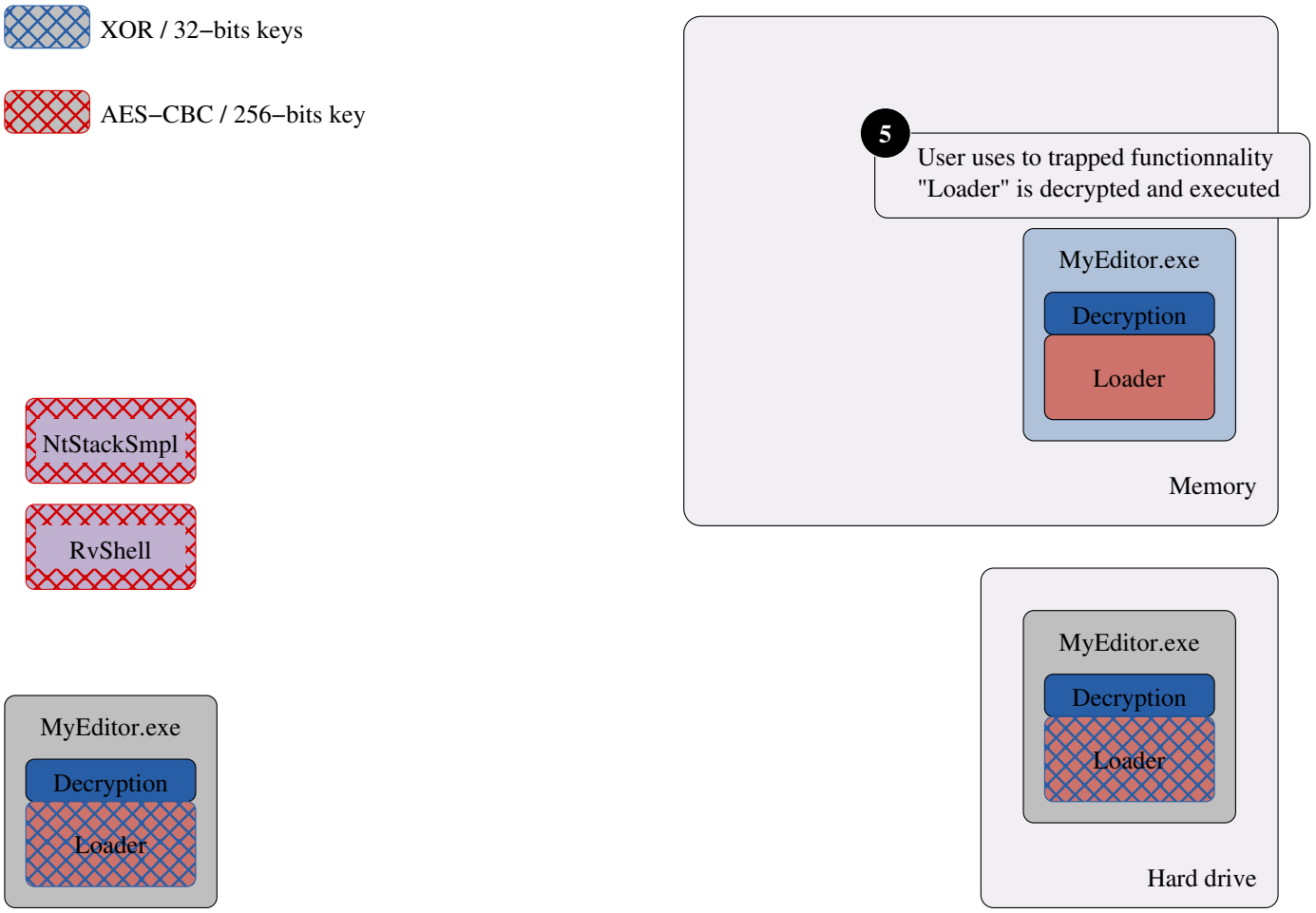


Figure: Principle of the attack with RvShell

Principle of the attack

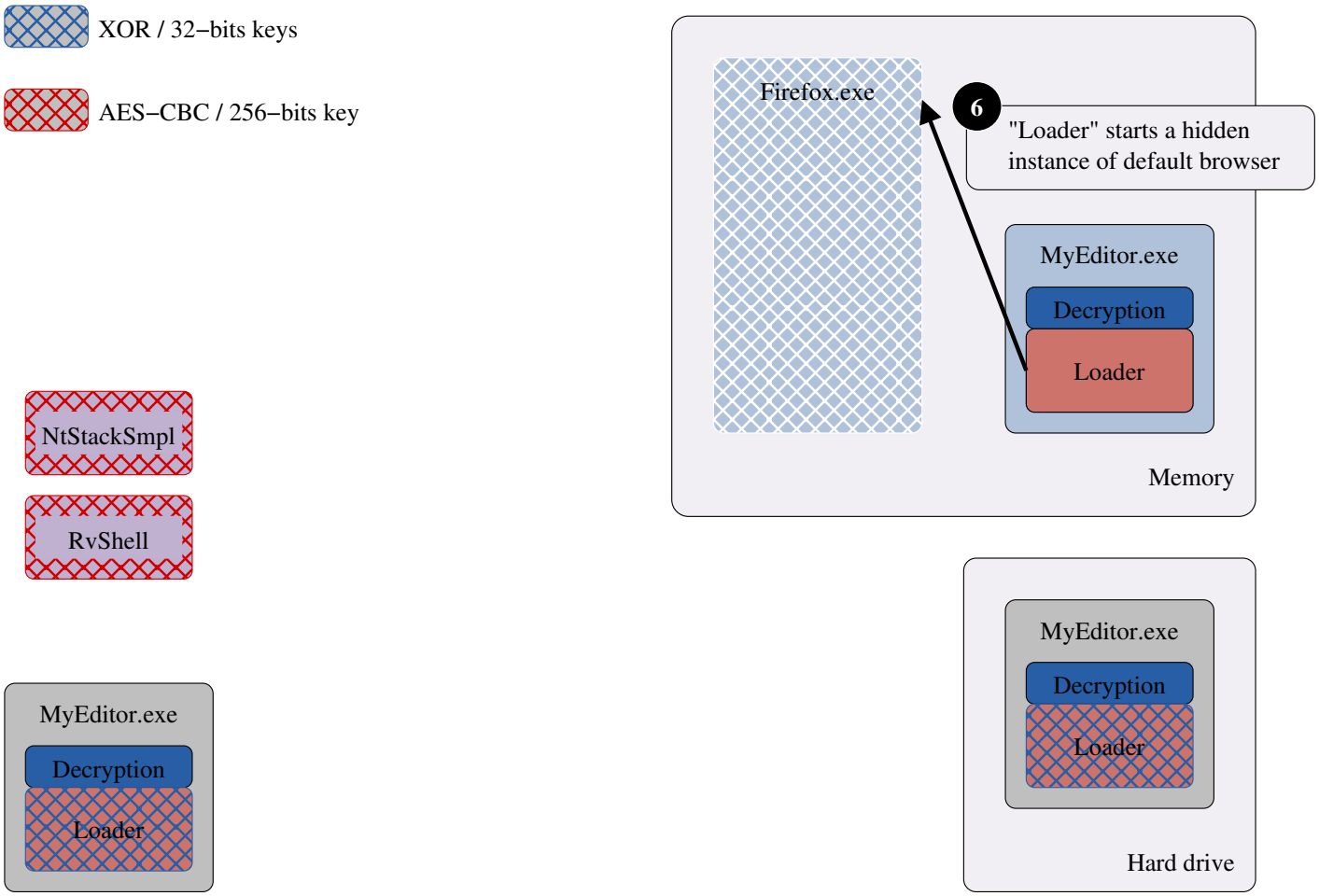


Figure: Principle of the attack with RvShell

Principle of the attack

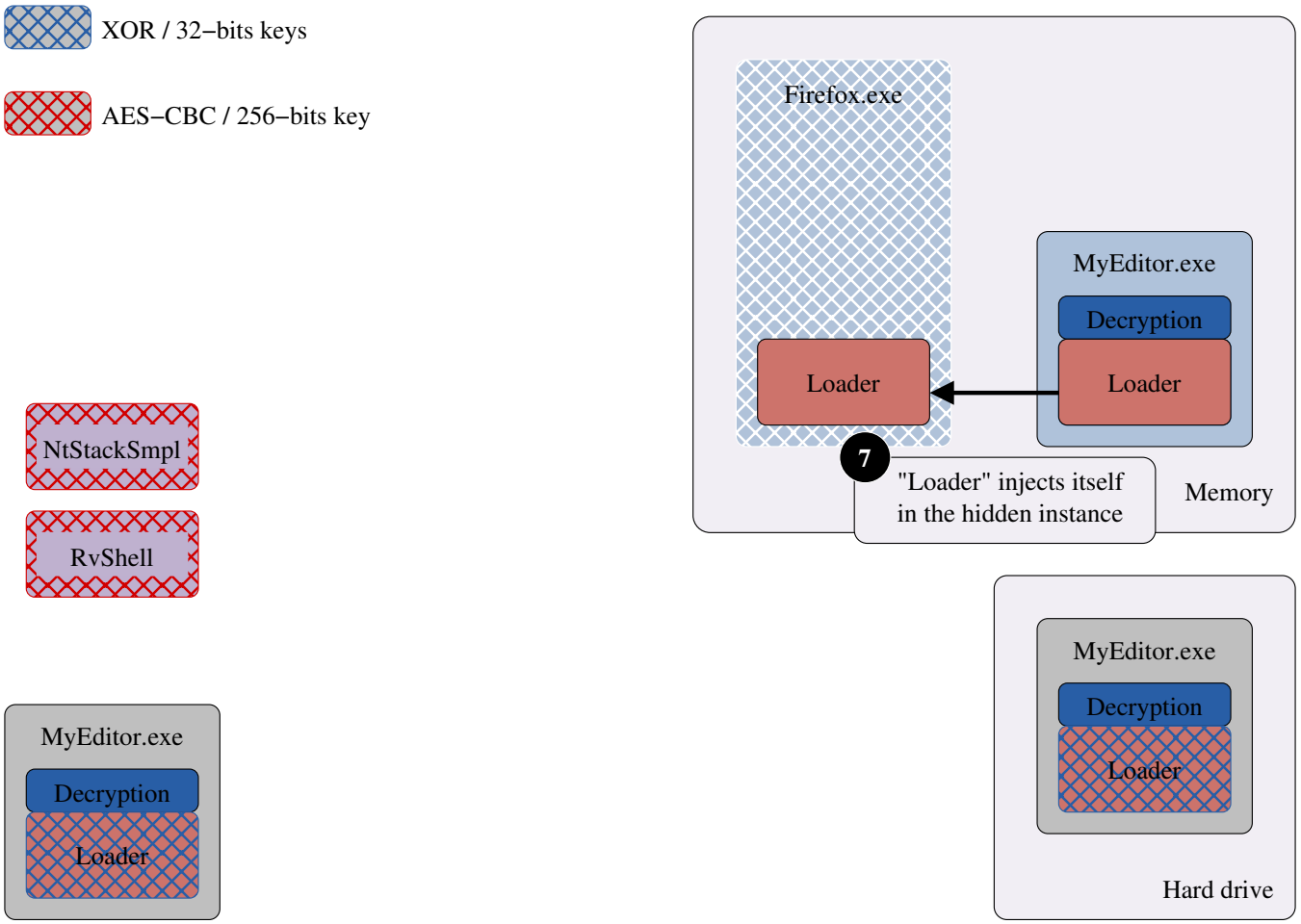


Figure: Principle of the attack with RvShell

Principle of the attack

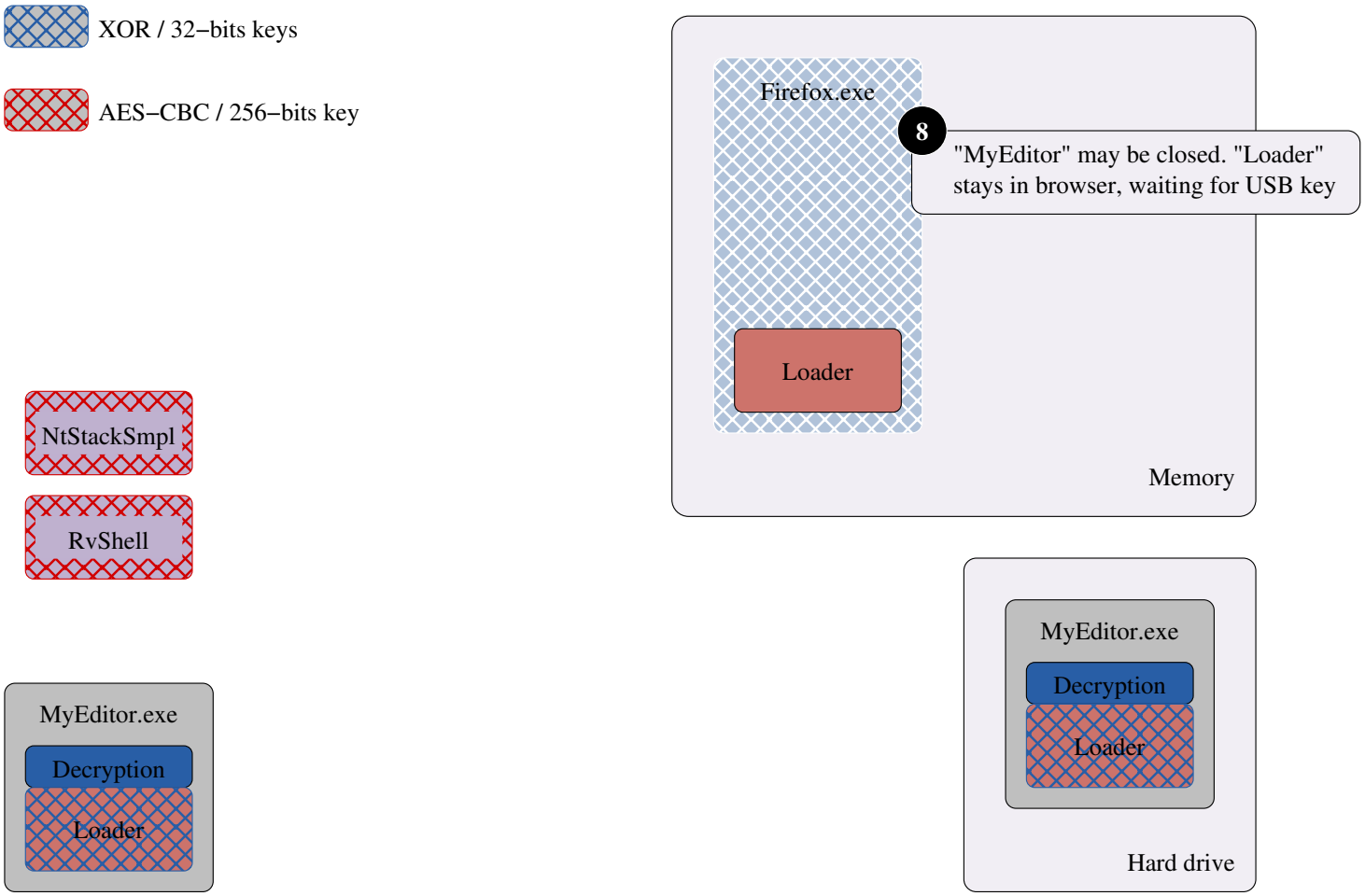


Figure: Principle of the attack with RvShell

Principle of the attack

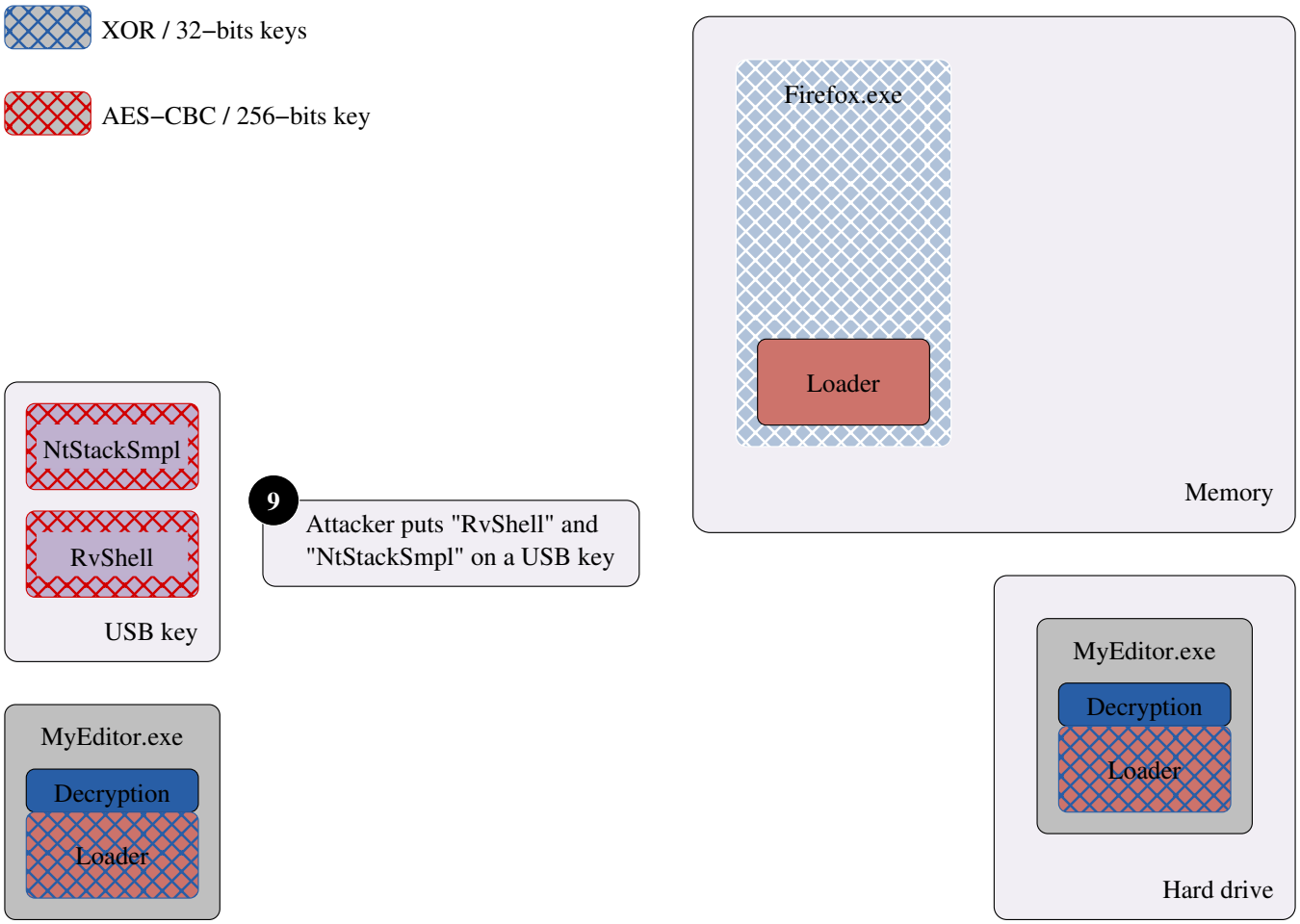


Figure: Principle of the attack with RvShell

Principle of the attack

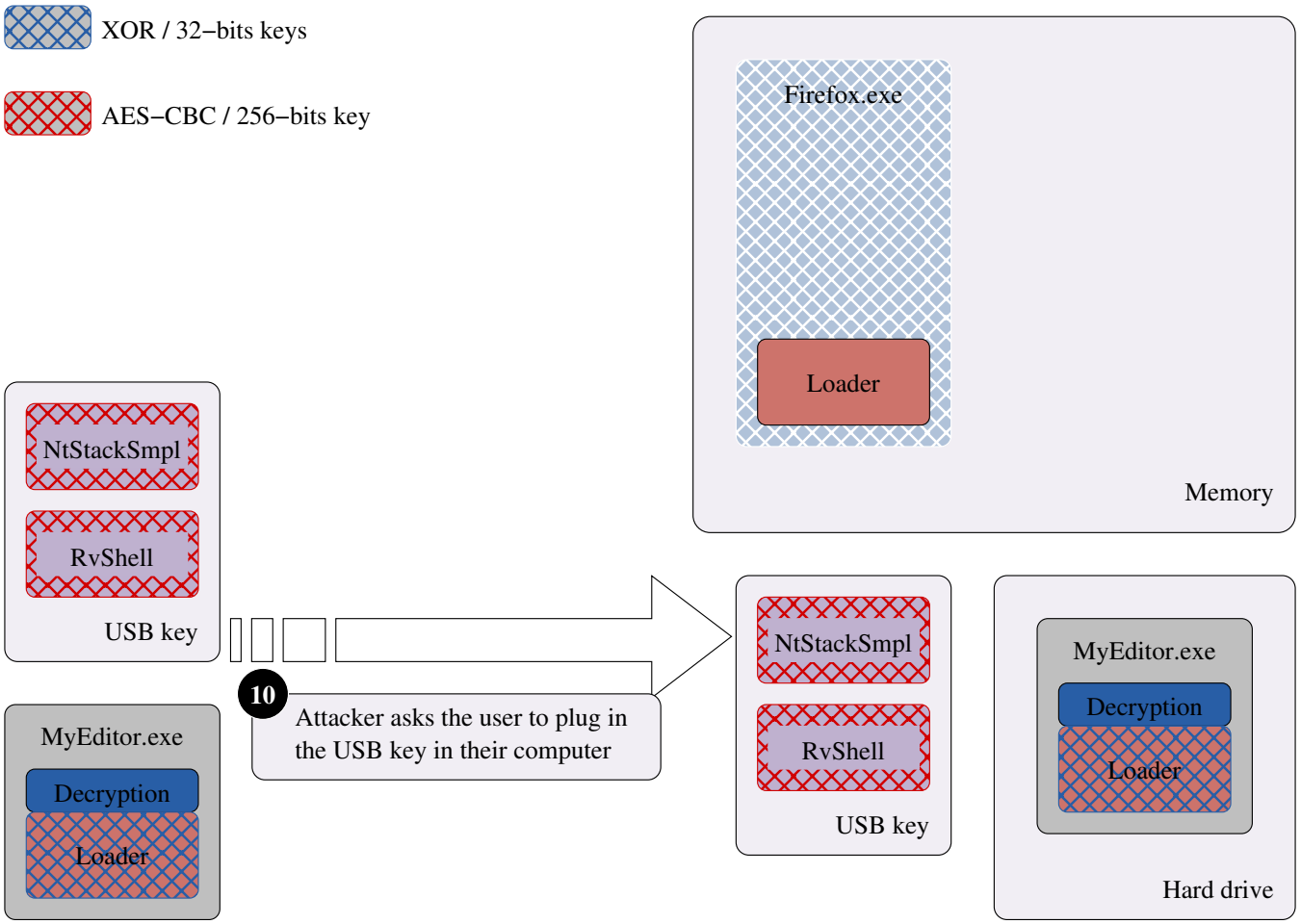


Figure: Principle of the attack with RvShell

Principle of the attack

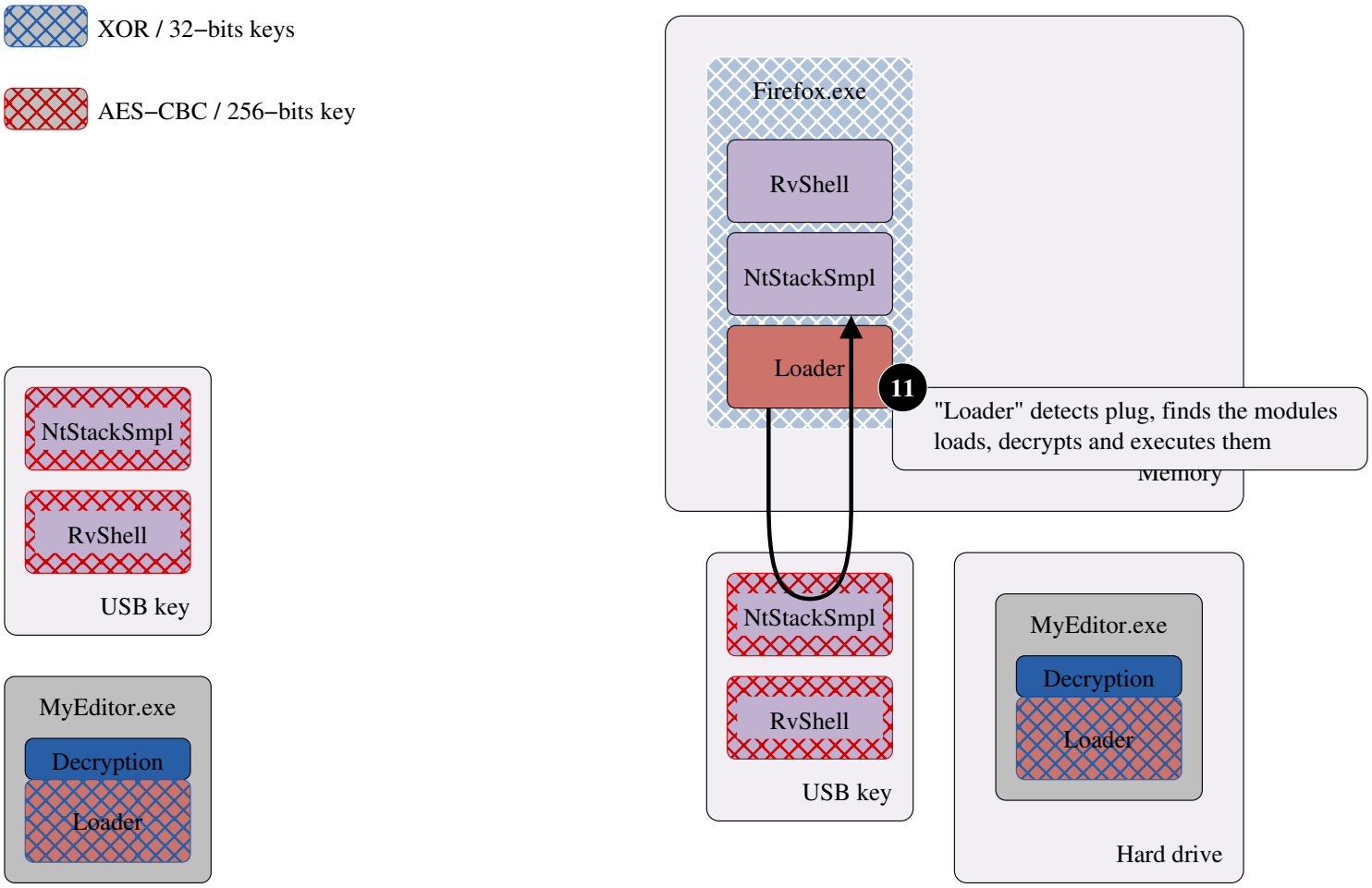


Figure: Principle of the attack with RvShell

Principle of the attack

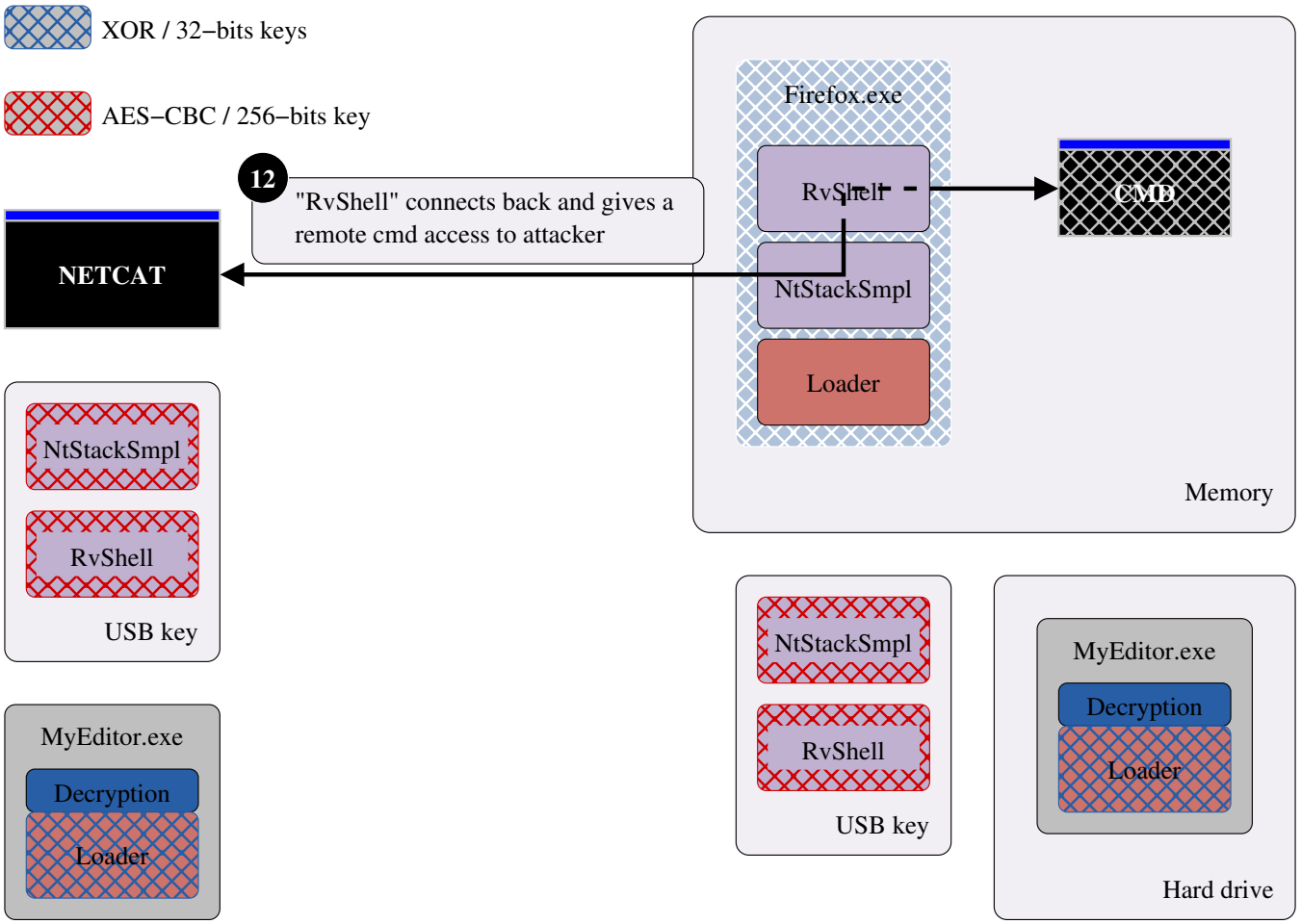


Figure: Principle of the attack with RvShell

Preparing attack - Generation of secret keys

- UsbLoader contains a master secret key
- RvShell and NtStackSmpl are each encrypted with a different secret key
- the secret key of each module is encrypted by the master key
- the two encrypted modules and the two encrypted secret keys are put on the USB key
  $\Rightarrow$ The master key and the module secret key are both required to decrypt a module

Preparing attack - Generation of Loader

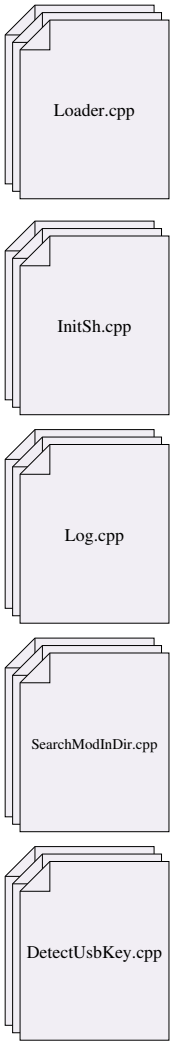


Figure: Generation of Loader

Preparing attack - Generation of Loader

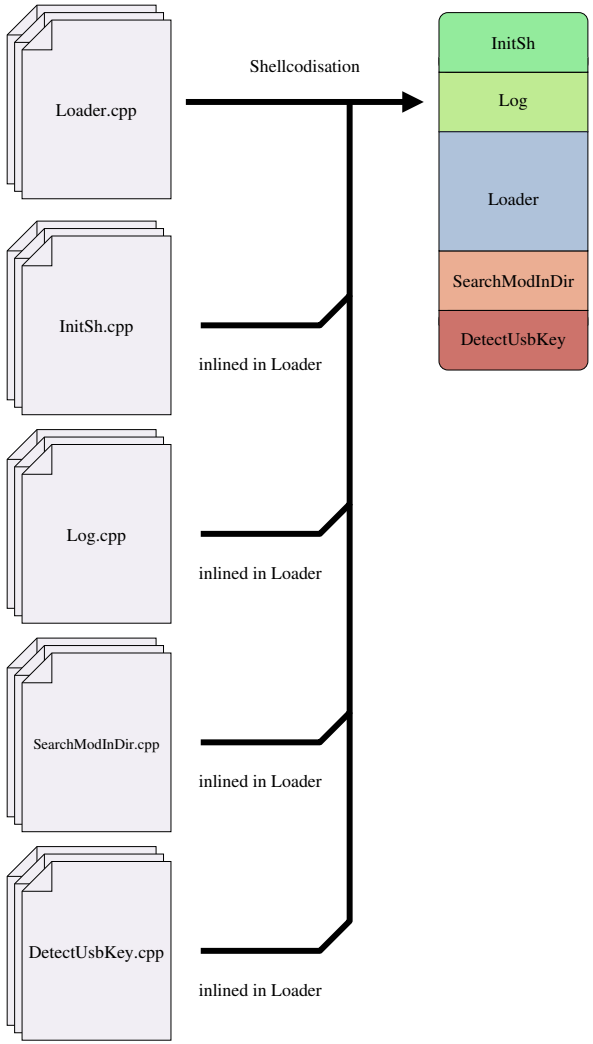


Figure: Generation of Loader

Preparing attack - Generation of Loader

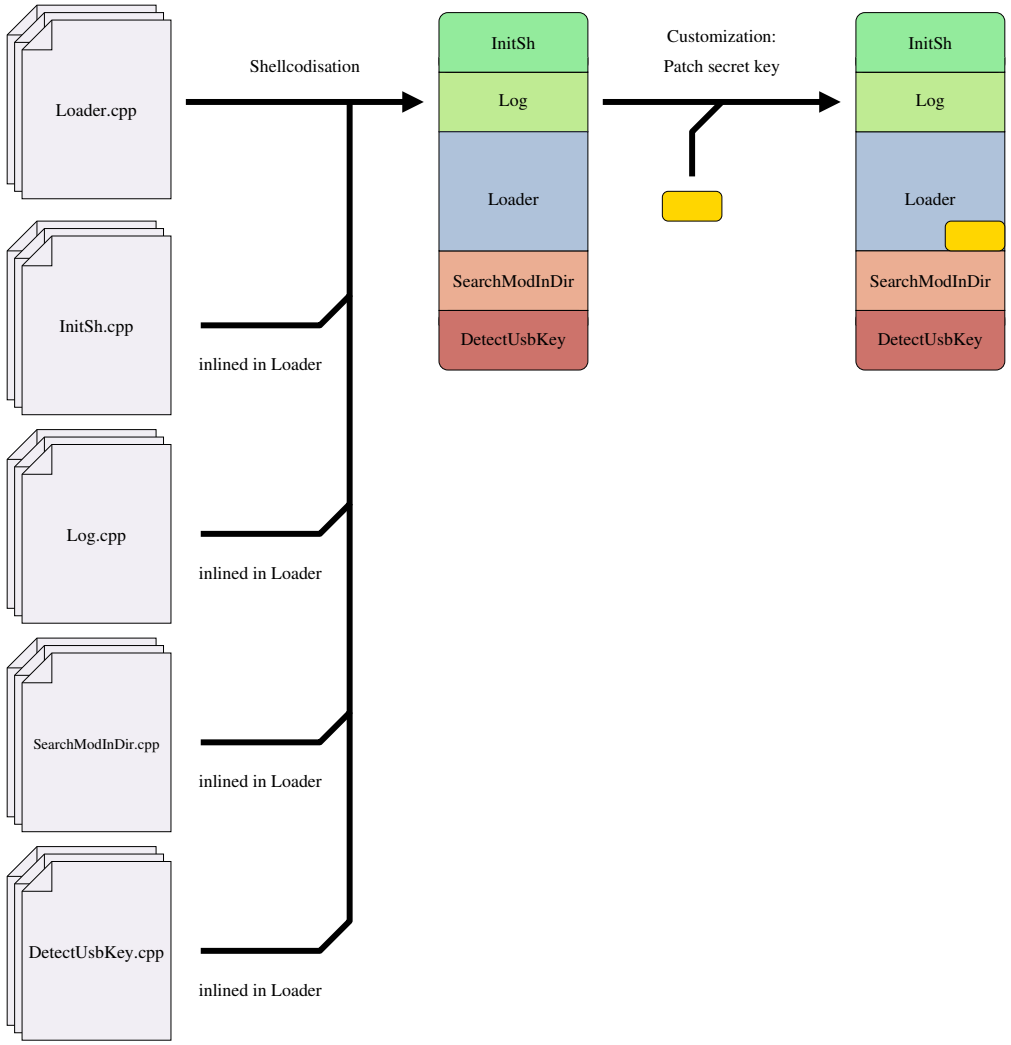


Figure: Generation of Loader

Preparing attack - Generation of RvShell and NtStackSmpl

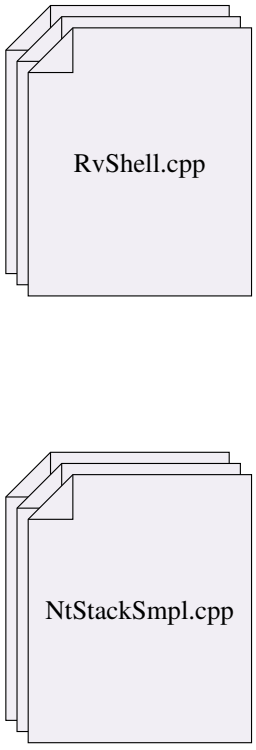


Figure: Generation of RvShell and NtStackSmpl

Preparing attack - Generation of RvShell and NtStackSmpl

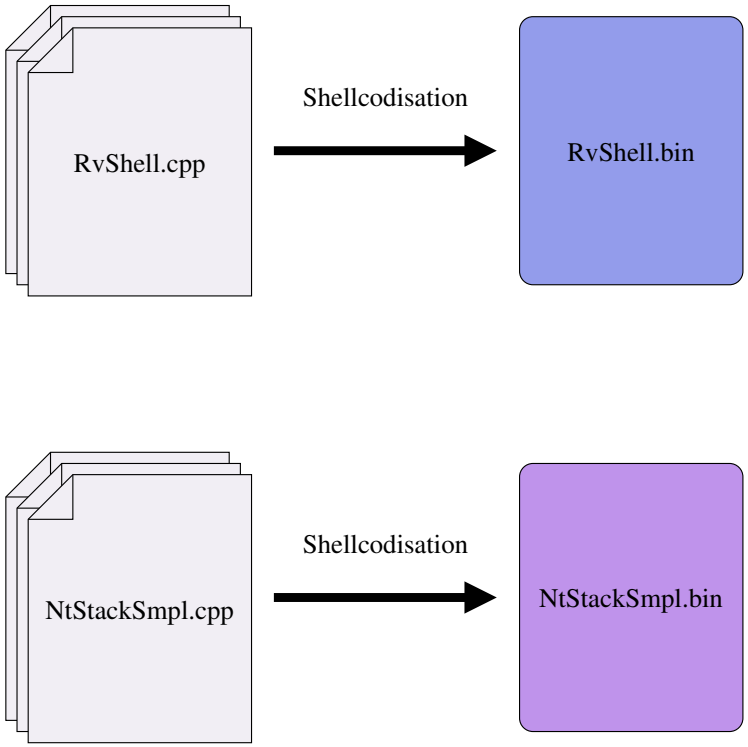


Figure: Generation of RvShell and NtStackSmpl

Preparing attack - Generation of RvShell and NtStackSmpl

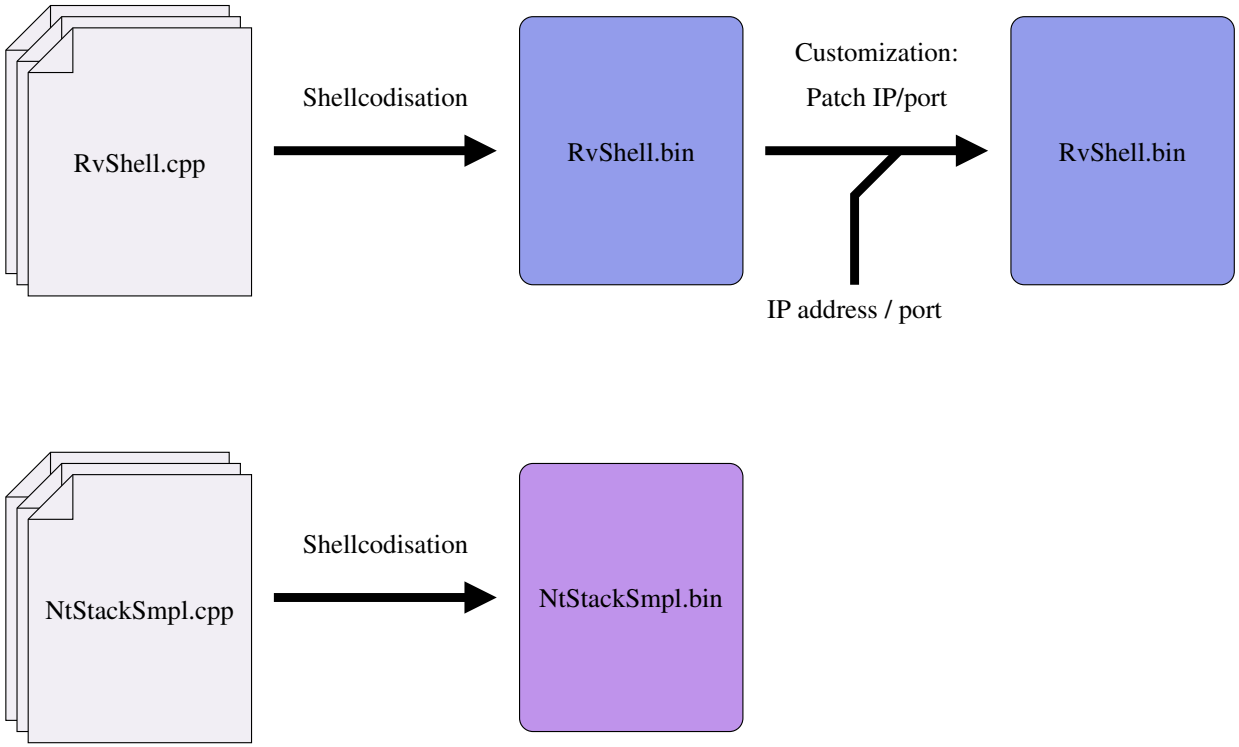


Figure: Generation of RvShell and NtStackSmpl

Preparing attack - Generation of RvShell and NtStackSmpl

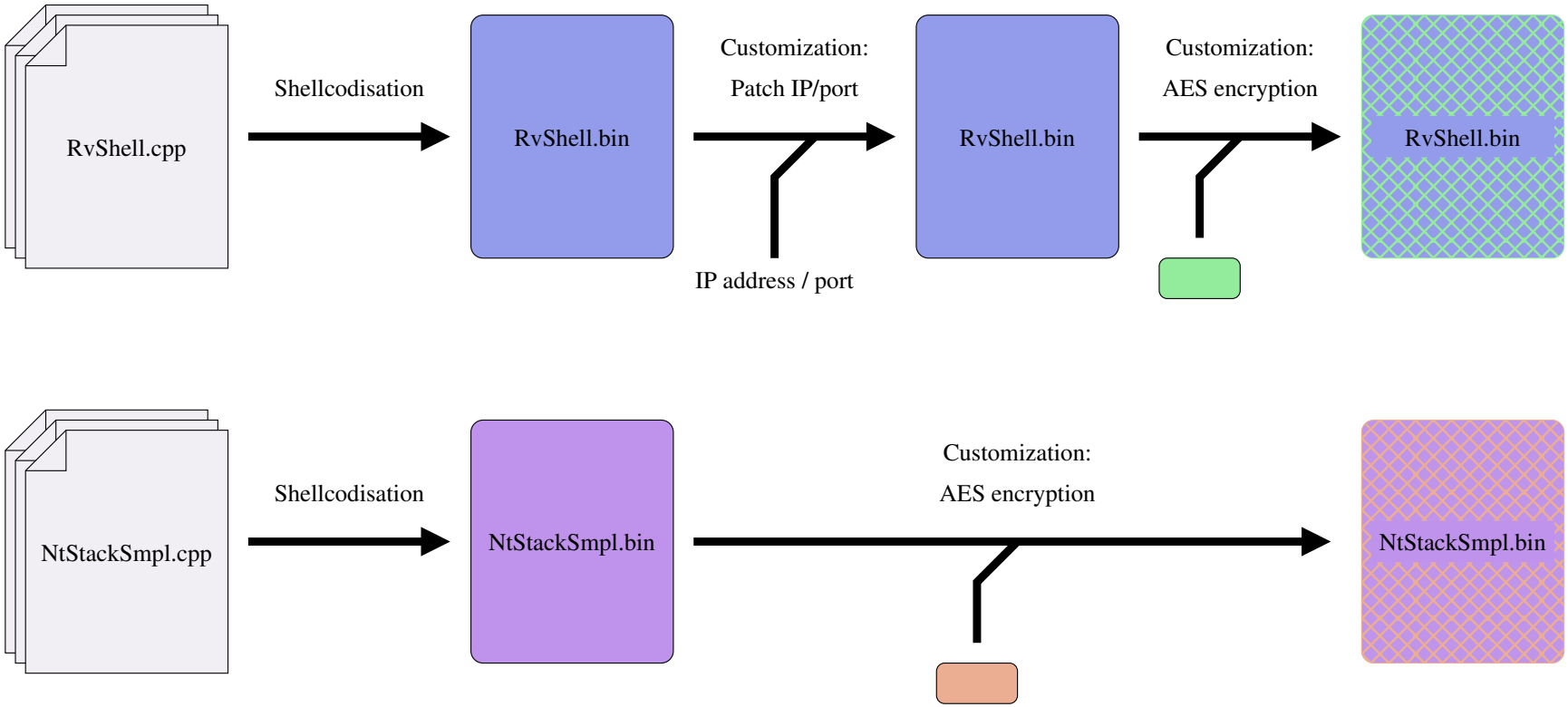


Figure: Generation of RvShell and NtStackSmpl

The module "Injecter"

- "Injecter is a module that injects a shellcode in another process (OpenProcess/WriteProcessMemory/CreateRemoteThread)
- Injection can be:
  - in a new hidden instance of default browser
  - in a new hidden instance of a specified program
  - in first process which name matches specified name
  - in all processes whose names match specified name

Preparing attack - Generation of Injecter

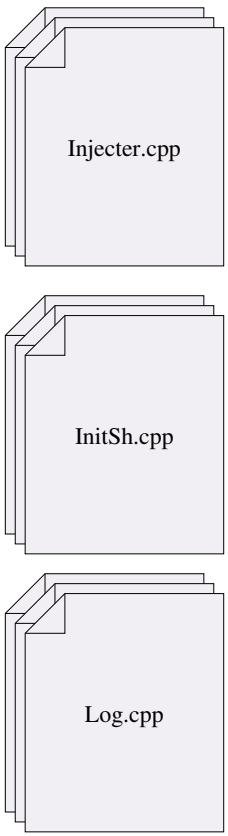


Figure: Generation of Injecter

Preparing attack - Generation of Injecter

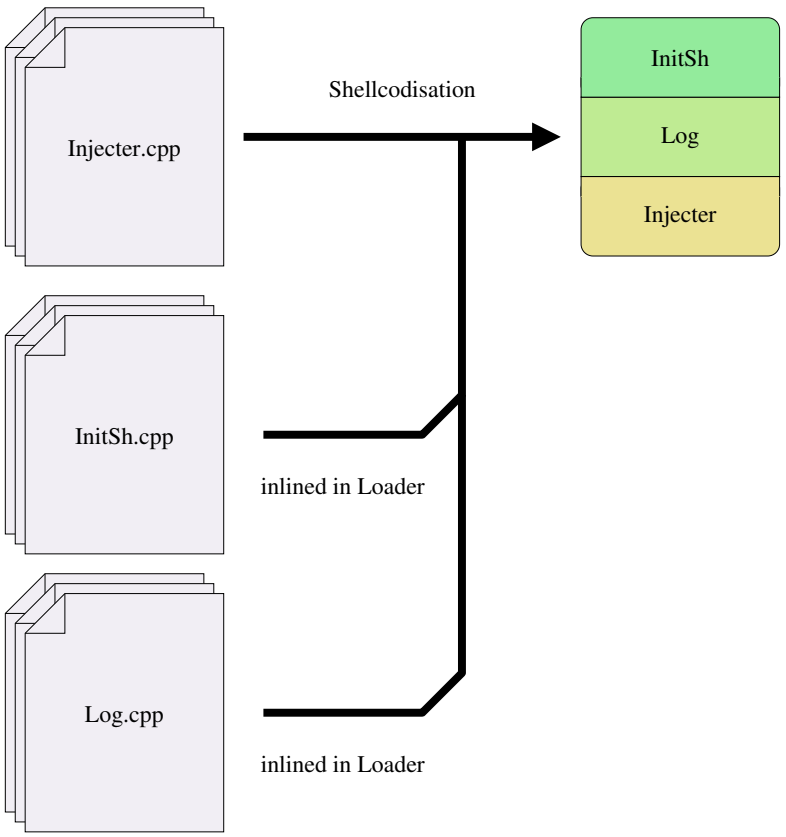


Figure: Generation of Injecter

Preparing attack - Generation of Injecter

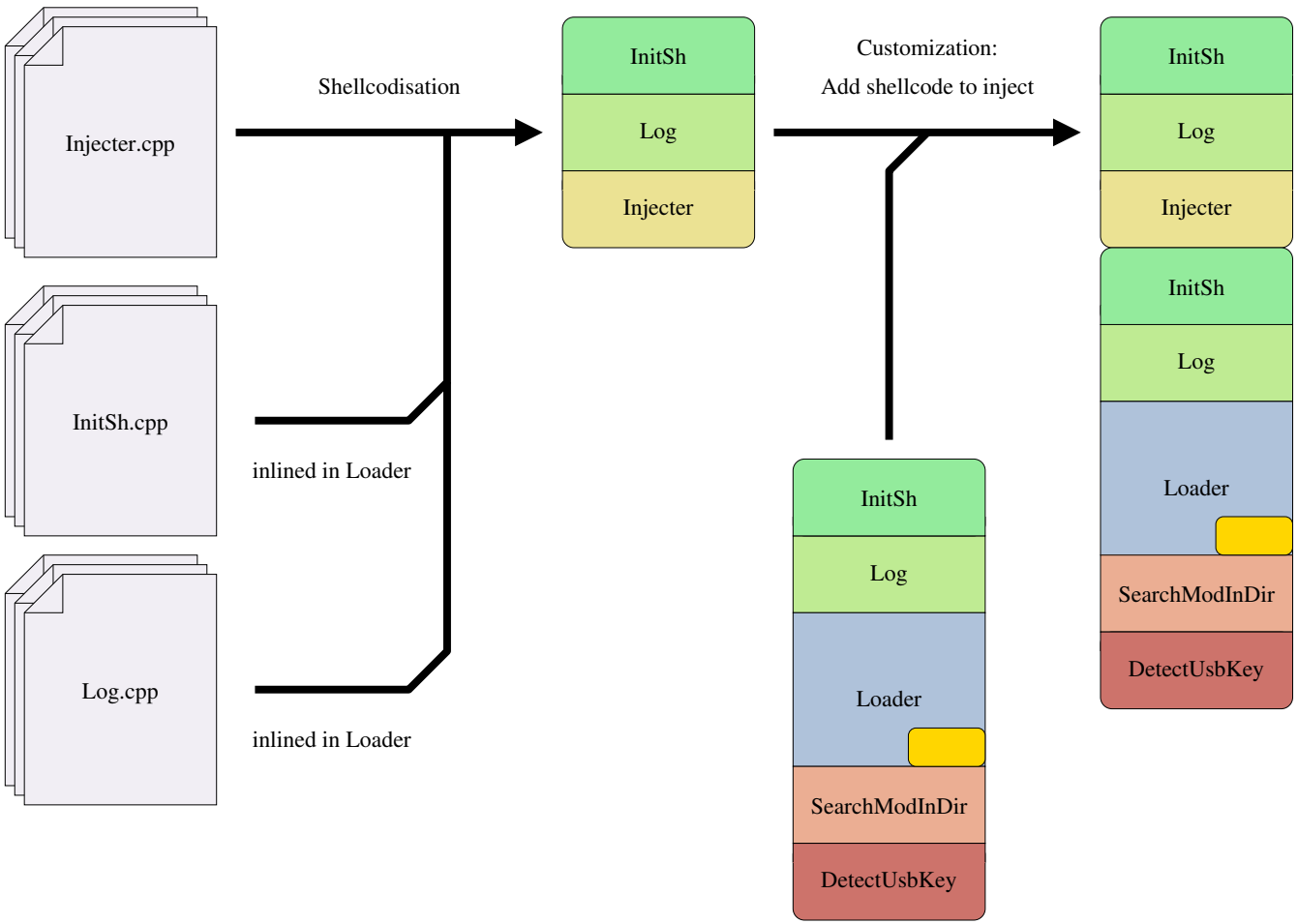


Figure: Generation of Injecter

Preparing attack - Generation of Injecter

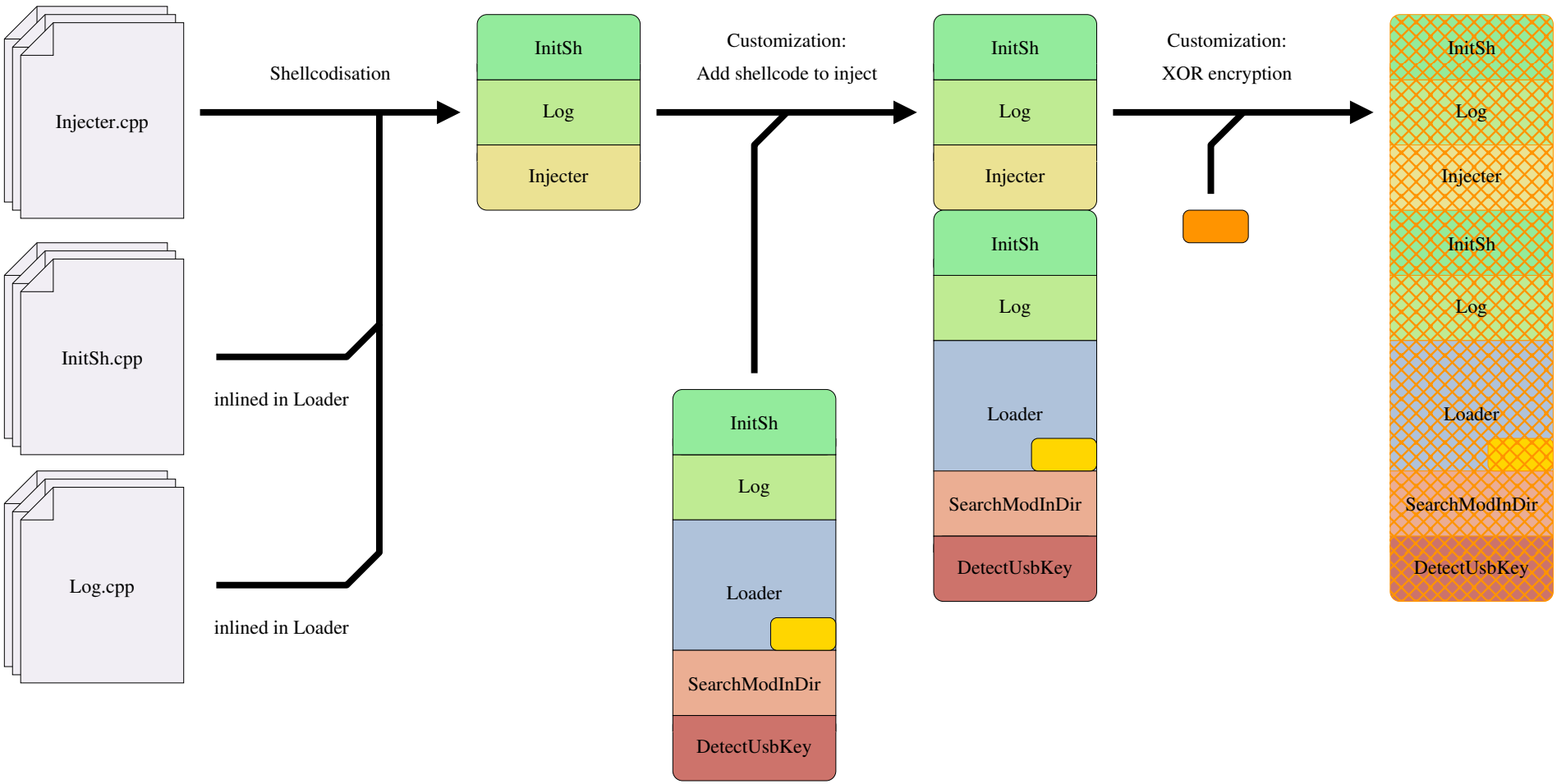


Figure: Generation of Injecter

Preparing attack - Generation of the Trojan

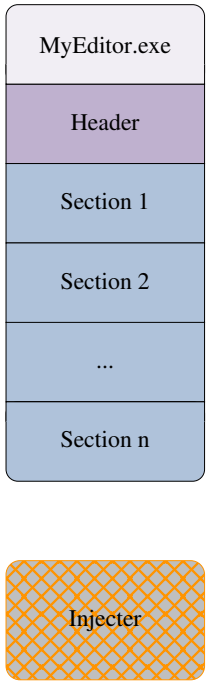


Figure: Generation of the Trojan

Preparing attack - Generation of the Trojan

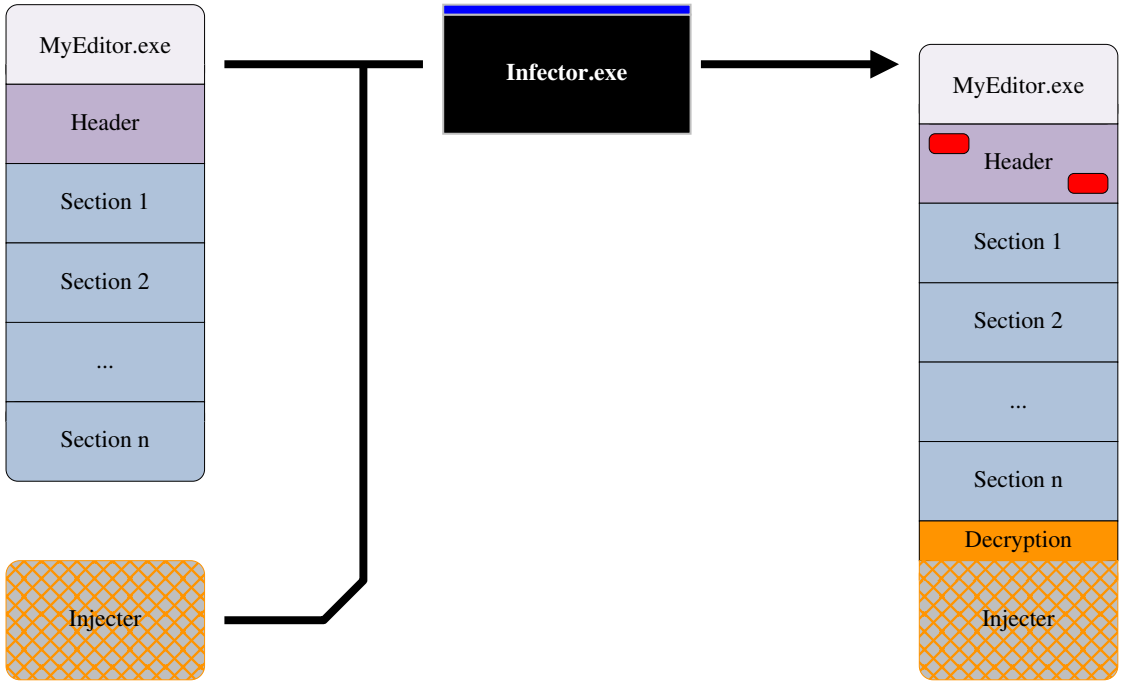


Figure: Generation of the Trojan

Preparing attack - Generation of the Trojan

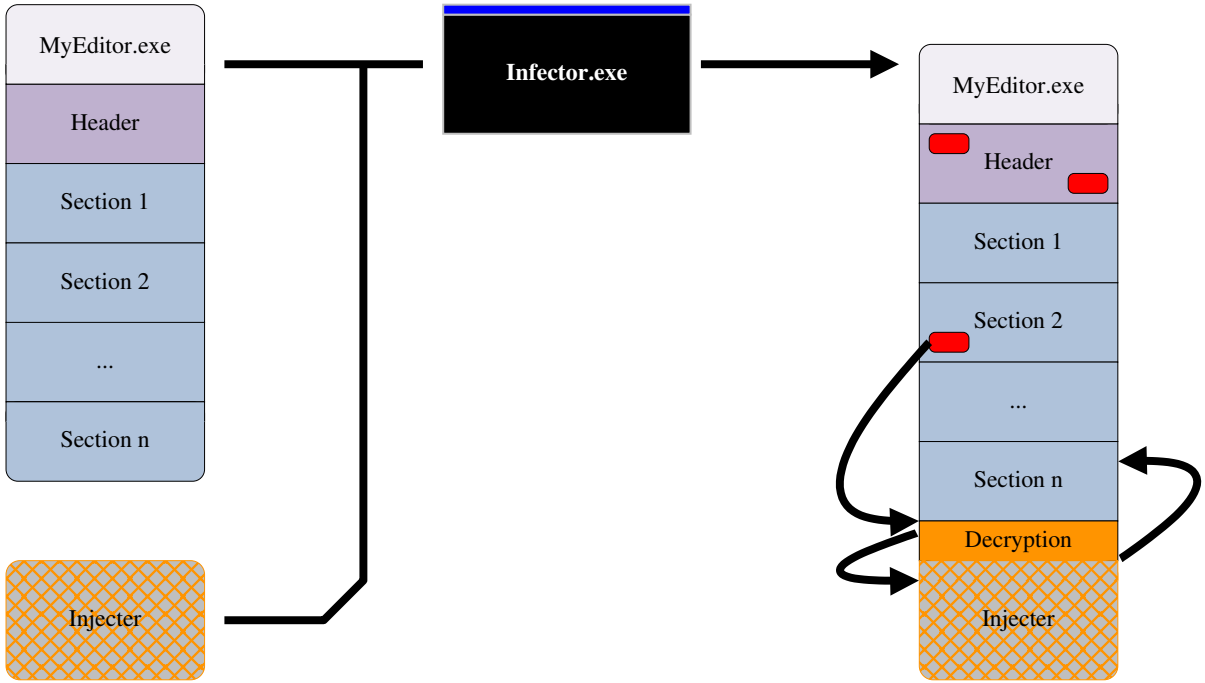


Figure: Generation of the Trojan

RvShell attack in practice

Demo : Execution of the attack

Attack - summary

Techniques used during this attack:

- Encryption of malicious payload:
  - "Injecter" in "MyEditor": polymorphism
  - "NtStackSmpl" and "RvShell": strong encryption (decrypted in memory)
- Code injection: "Loader" executed in a hidden process
- Executable infection: trojan created from "MyEditor"

Plan

Context

Objective

Take control of a web server; steal username/password of web site users

Context

Objective

Take control of a web server; steal username/password of web site users

Description of the target

- Windows
- Two services:
  - Apache/PHP/MySQL with a phpbb (target)
  - FTP server used to update web site
- Server protected by a firewall (allows only incoming HTTP/FTP)

Context

Objective

Take control of a web server; steal username/password of web site users

Description of the target

- Windows
- Two services:
  - Apache/PHP/MySQL with a phpbb (target)
  - FTP server used to update web site
- Server protected by a firewall (allows only incoming HTTP/FTP)

Context of the attack

- Attacker found a valid user/pass for FTP server
- File system regularly checked
  $\Rightarrow$ impossible to leave a backdoor on system
  $\Rightarrow$ Attacker decides to use a personal tool: "WebDoor"

Presentation of WebDoor

Webdoor executes the following actions:

- Finds a targeted process that represents a web server
- Injects a shellcode in this process that will install a hook on function "WSARecv"
- Hook analyses every web request and extracts parameters:
  - parameter "shell" $\Rightarrow$ interpretes command in a mini-shell
    Example: "shell=cmd" gives access to a remote cmd on server
  - otherwise compares every name of parameter with list of keywords to detect username/password
- Web server work not disrupted

Principle of web server attack

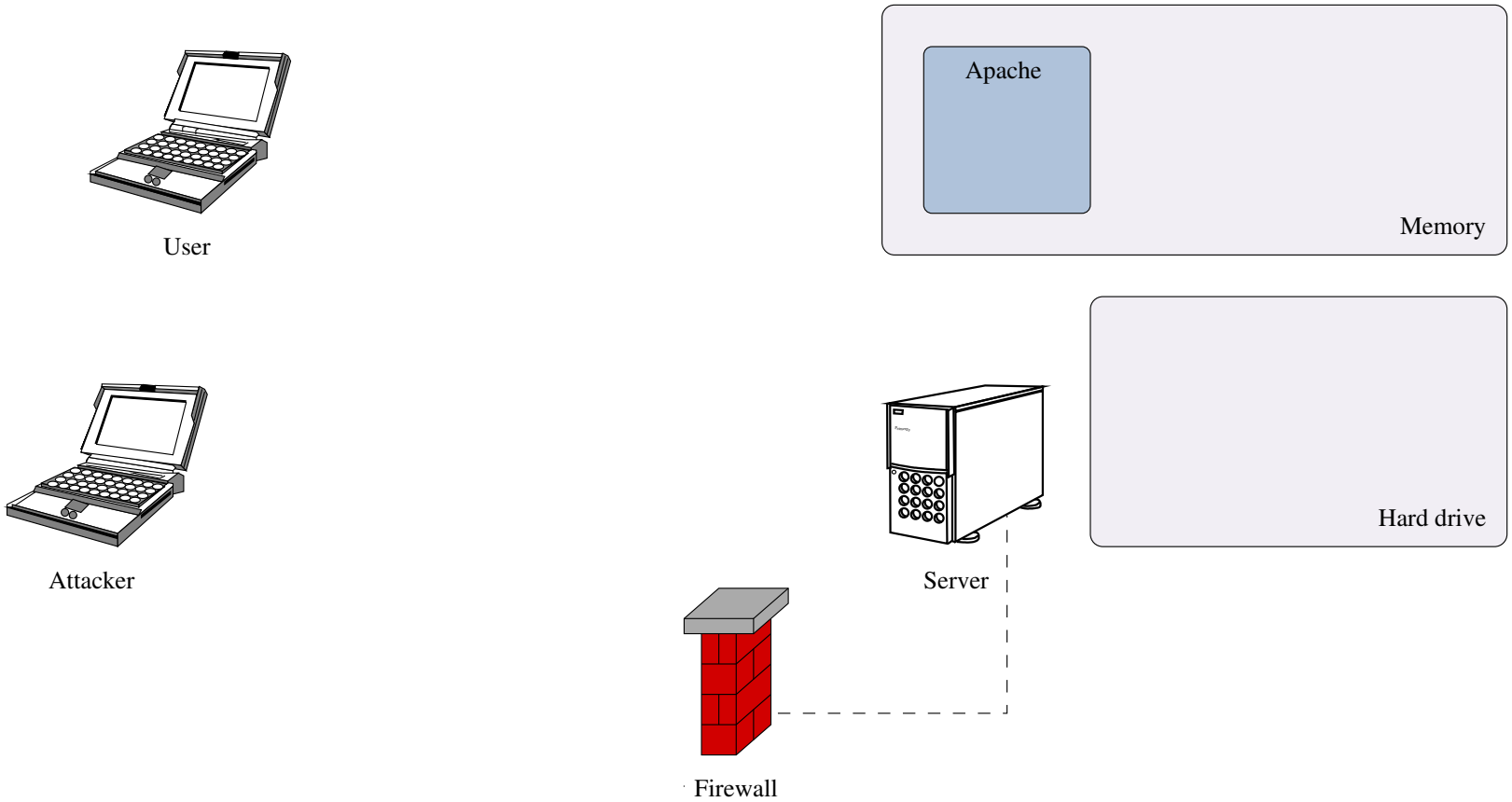


Figure: Principle of web server attack with WebDoor

Principle of web server attack

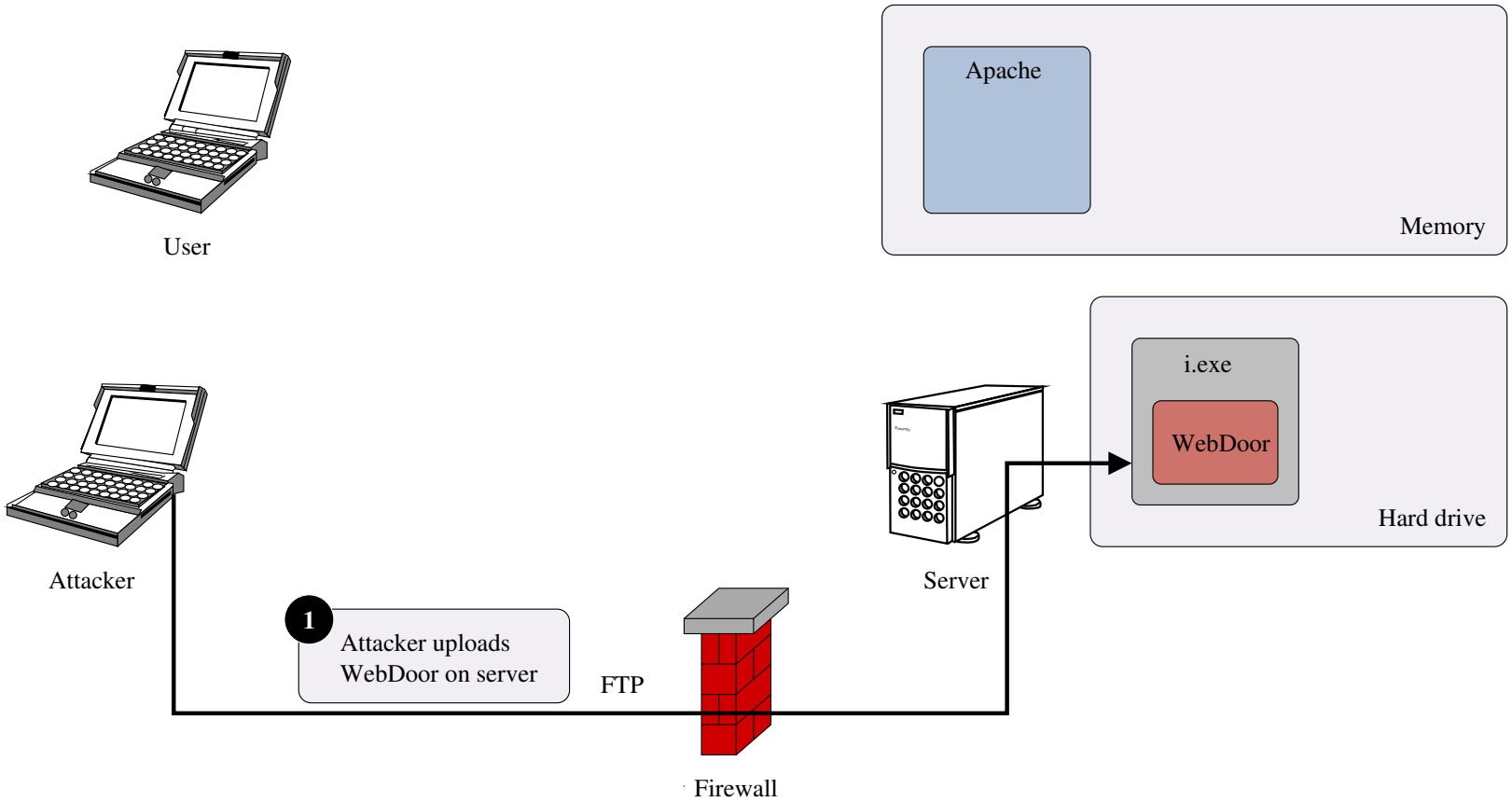


Figure: Principle of web server attack with WebDoor

Principle of web server attack

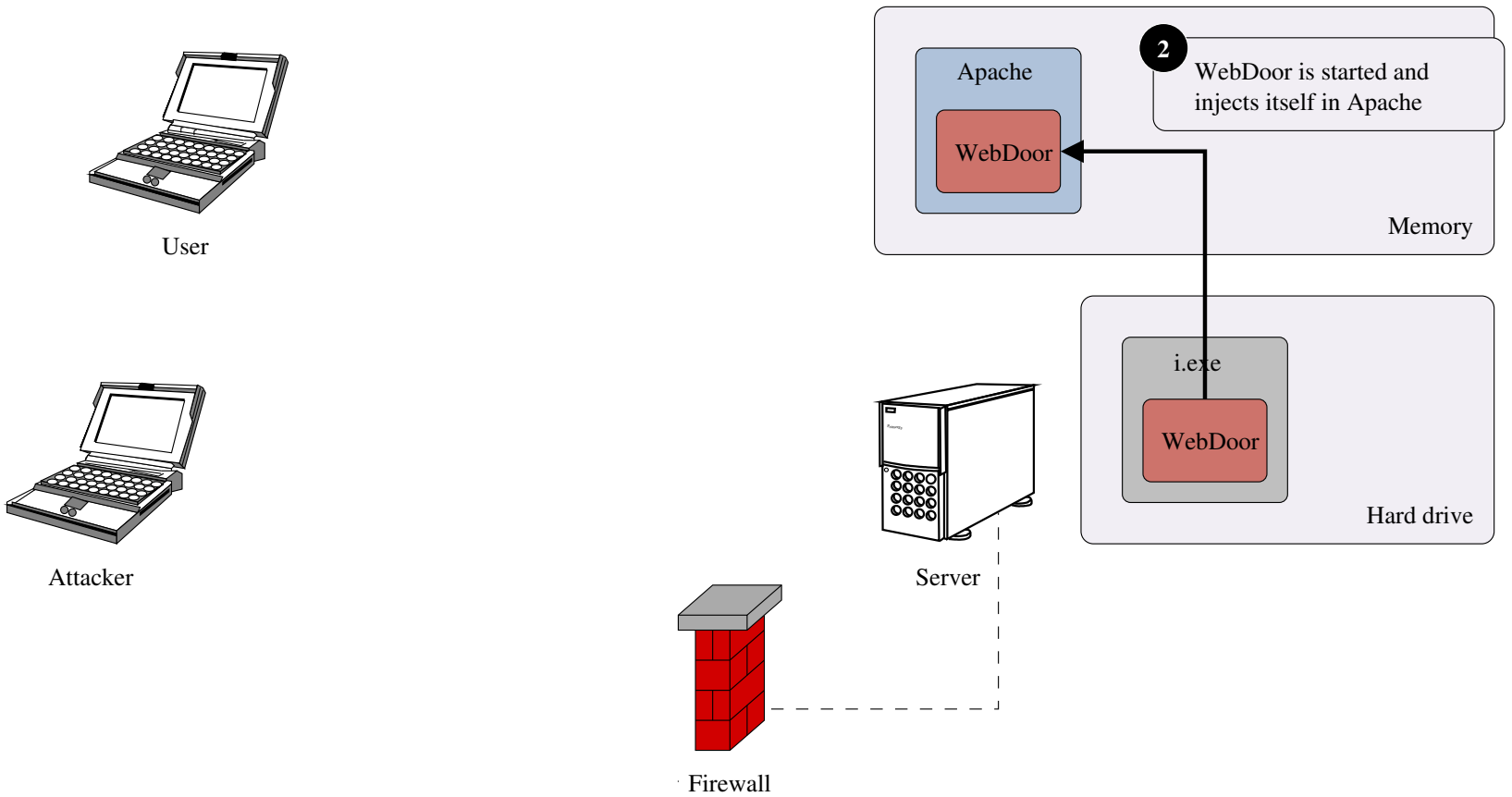


Figure: Principle of web server attack with WebDoor

Principle of web server attack

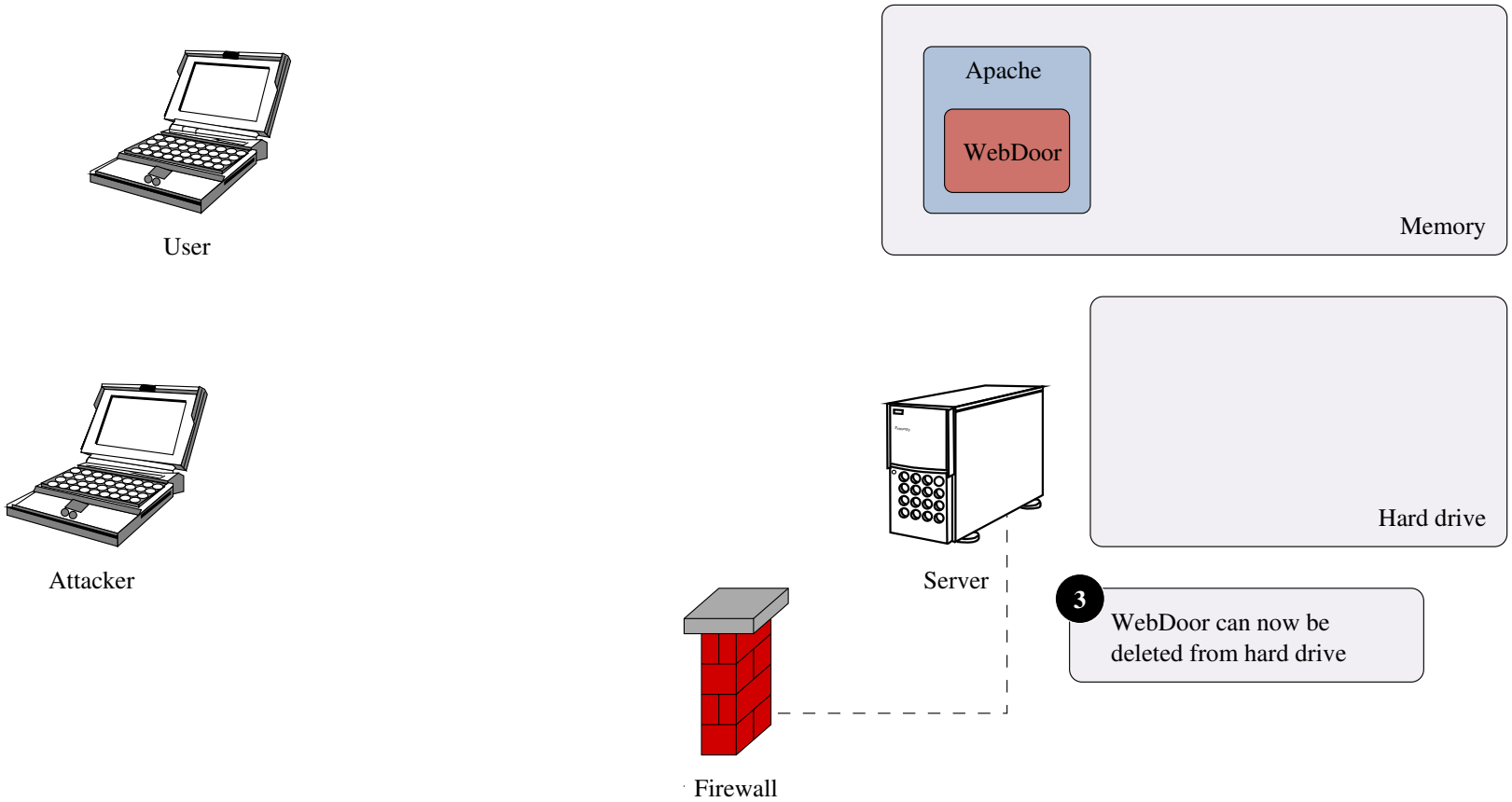


Figure: Principle of web server attack with WebDoor

Principle of web server attack

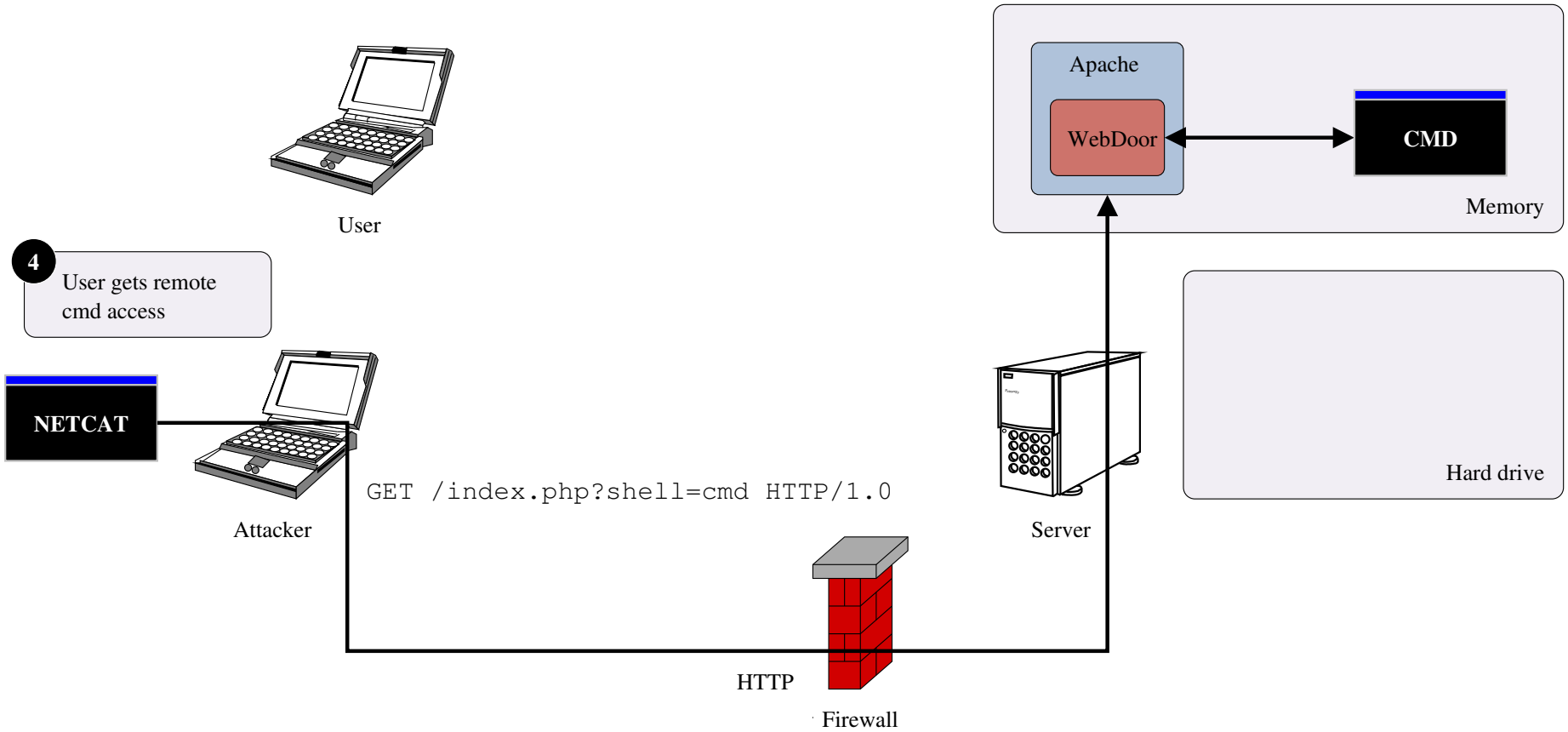


Figure: Principle of web server attack with WebDoor

Principle of web server attack

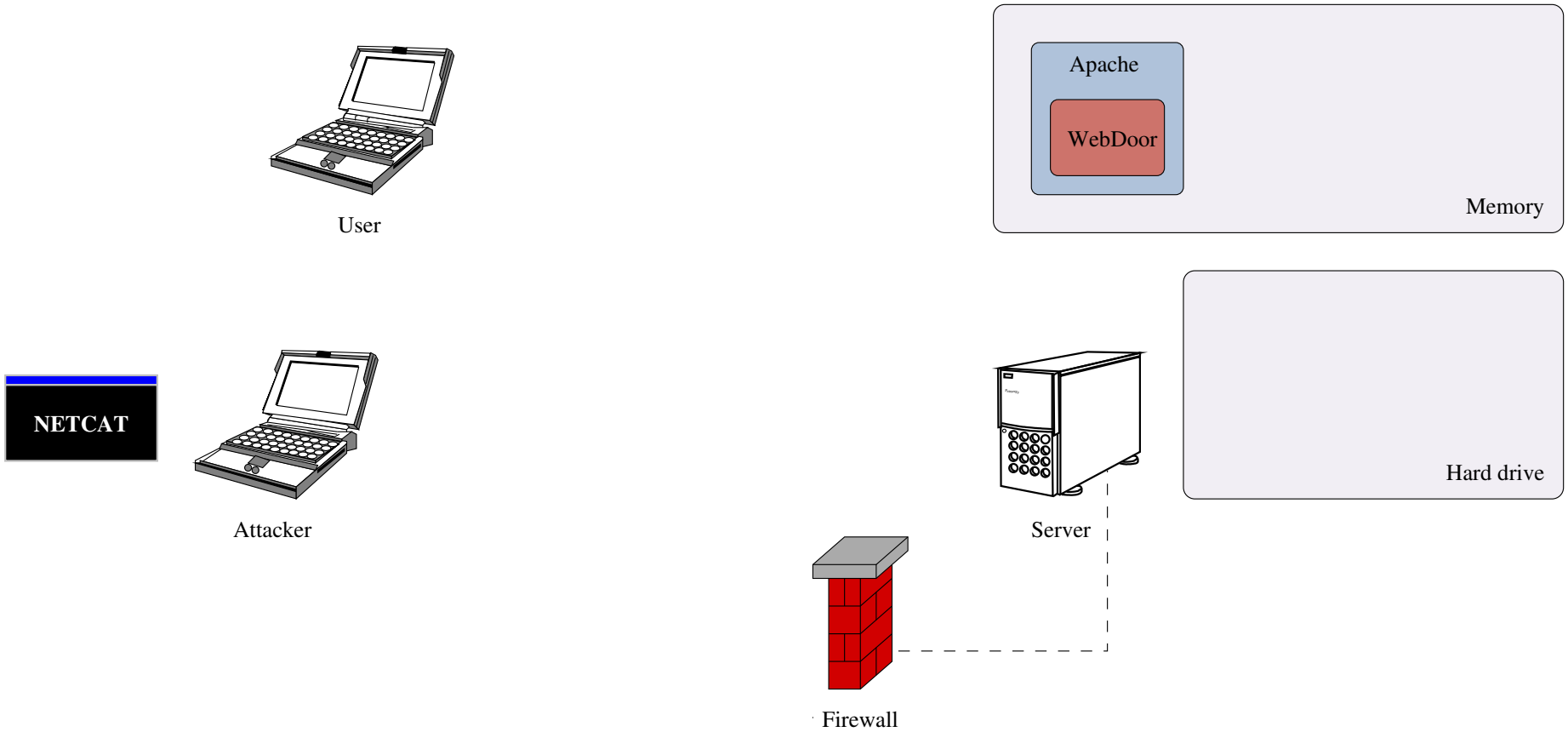


Figure: Principle of web server attack with WebDoor

Principle of web server attack

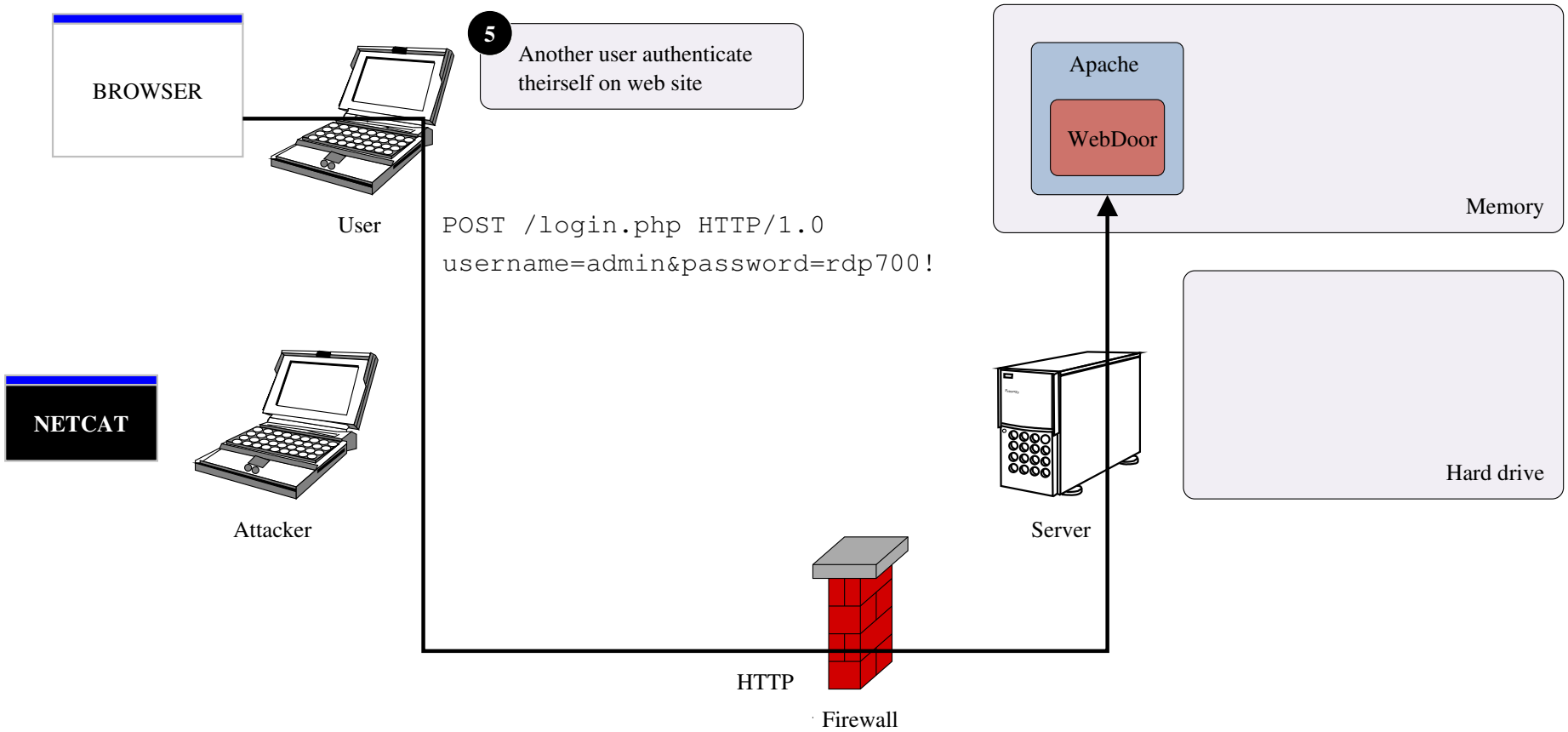


Figure: Principle of web server attack with WebDoor

Principle of web server attack

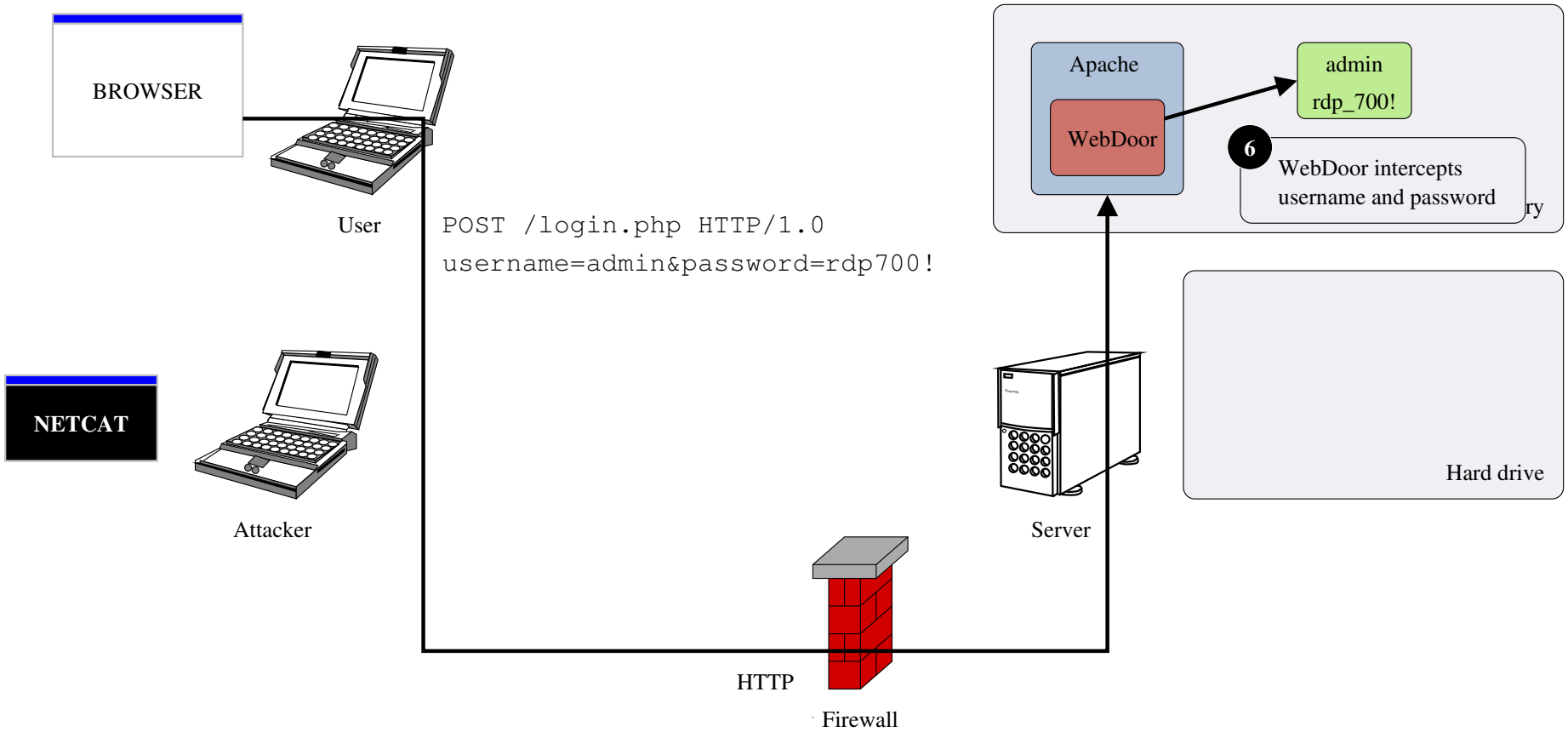


Figure: Principle of web server attack with WebDoor

Principle of web server attack

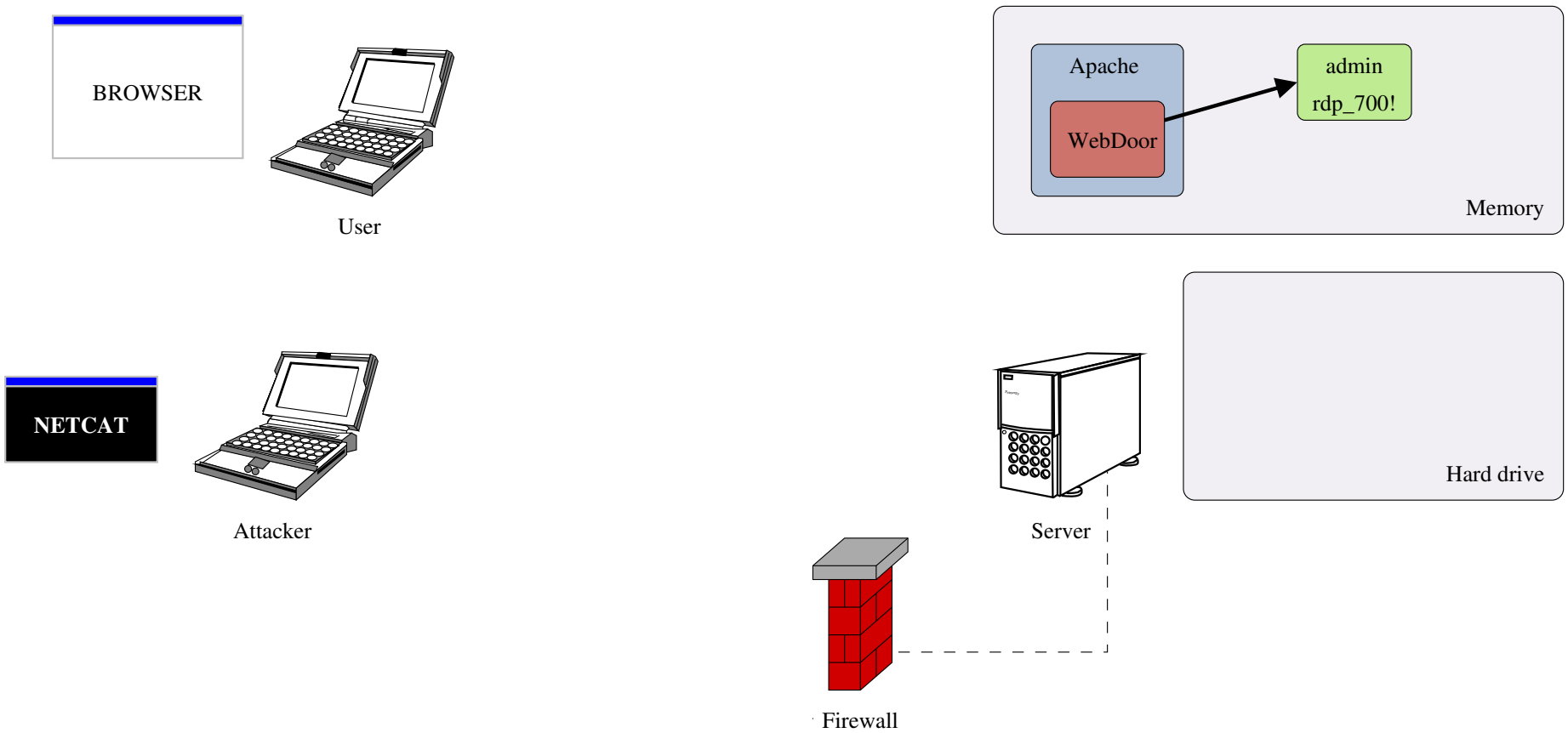


Figure: Principle of web server attack with WebDoor

Principle of web server attack

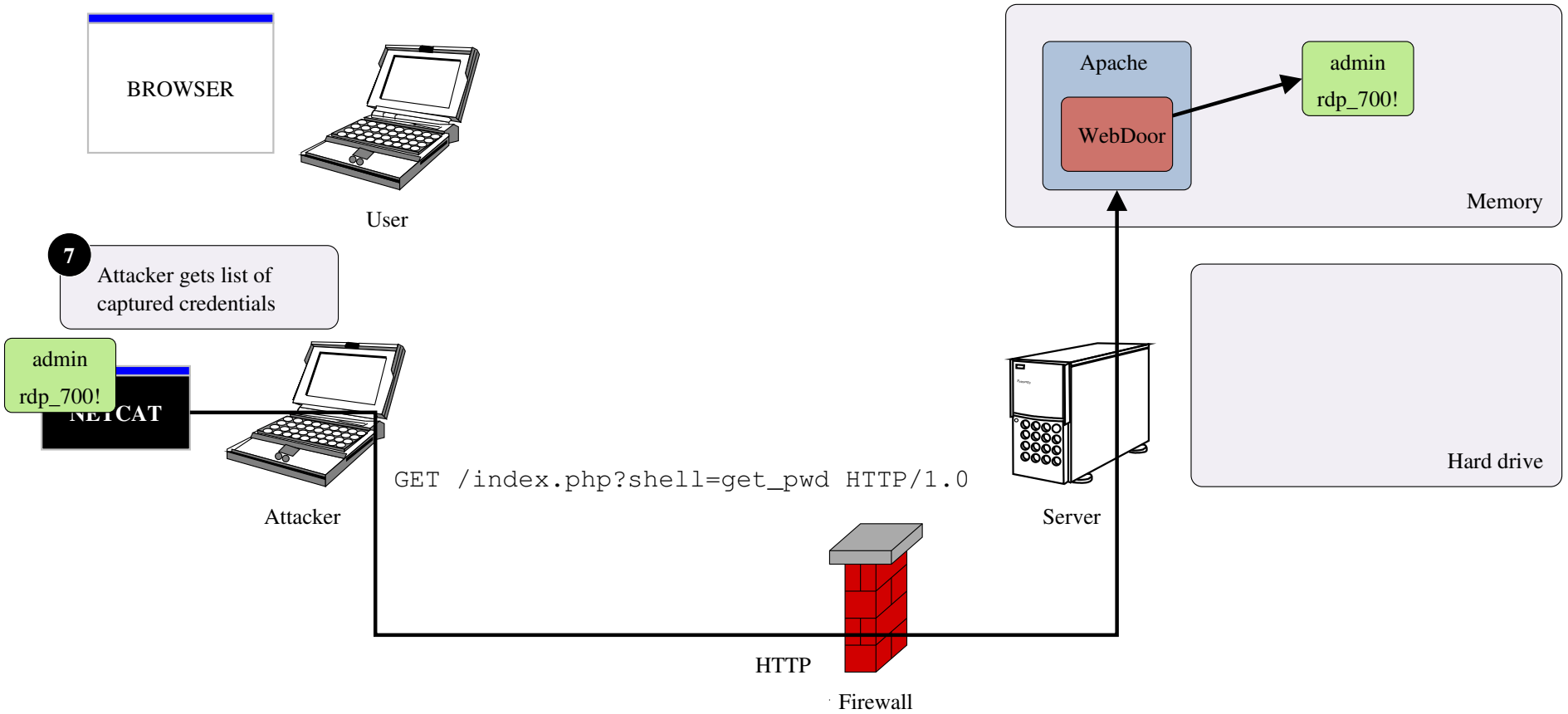


Figure: Principle of web server attack with WebDoor

Principle of API hooking

Several ways to do API hooking:

- Patch the Import Address Table
  - Replace entries in IAT of functions to hook with addresses of hook functions
  - Easy to implement but does not intercept calls to functions resolved dynamically

Principle of API hooking

Several ways to do API hooking:

- Patch the Import Address Table
  - Replace entries in IAT of functions to hook with addresses of hook functions
  - Easy to implement but does not intercept calls to functions resolved dynamically
- Patch header of function
  - Patch first bytes of function to hook with a jmp to hook function
  - All calls are intercepted, independently of the resolution mechanism
  - But solution not so easy to implement:
    - memory rights of the section must be changed
    - instruction alignment must be computed to save the overwritten instructions
    - stack must be rebuilt before calling real function

Principle of API hooking

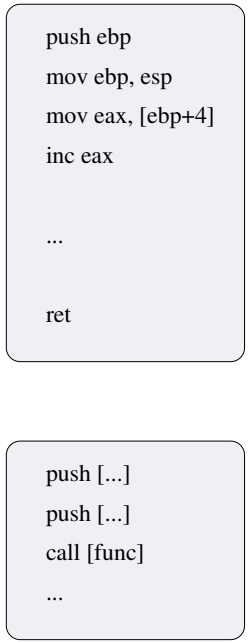


Figure: API hooking by header patching

Principle of API hooking

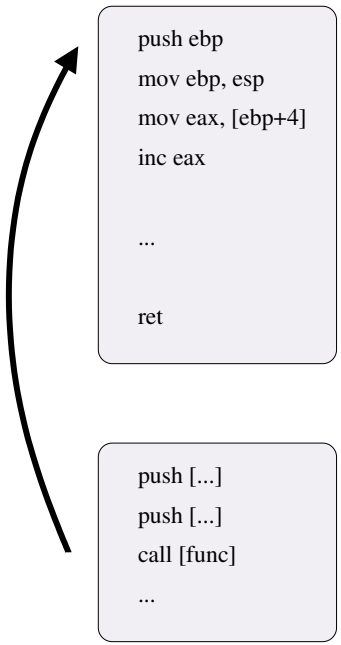


Figure: API hooking by header patching

Principle of API hooking

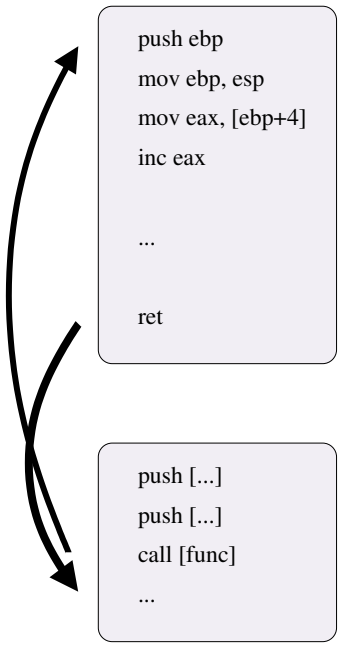


Figure: API hooking by header patching

Principle of API hooking

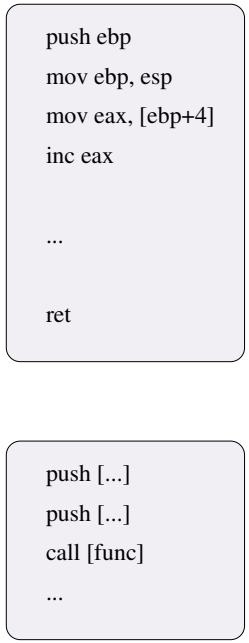


Figure: API hooking by header patching

Principle of API hooking

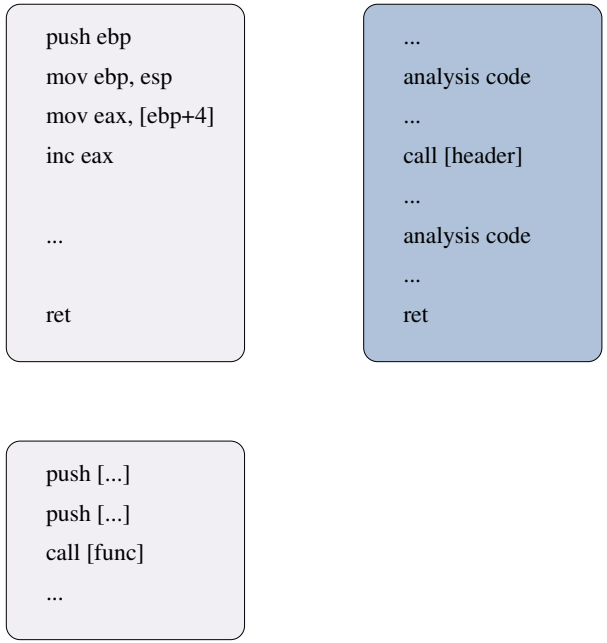


Figure: API hooking by header patching

Principle of API hooking

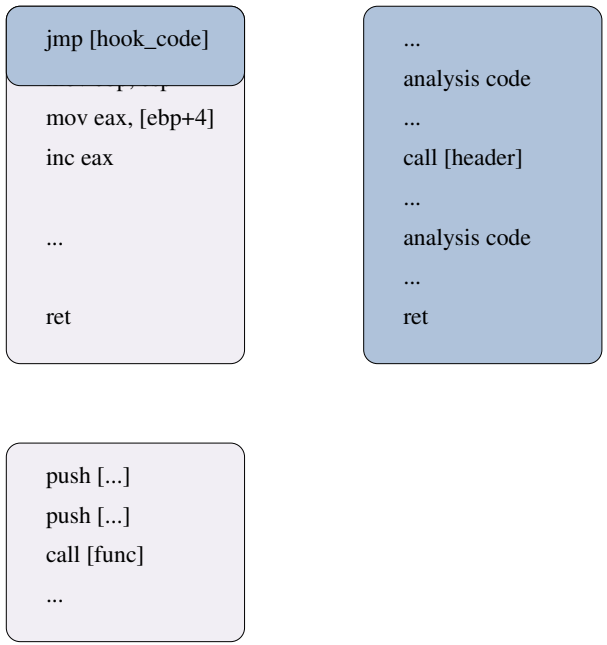


Figure: API hooking by header patching

Principle of API hooking

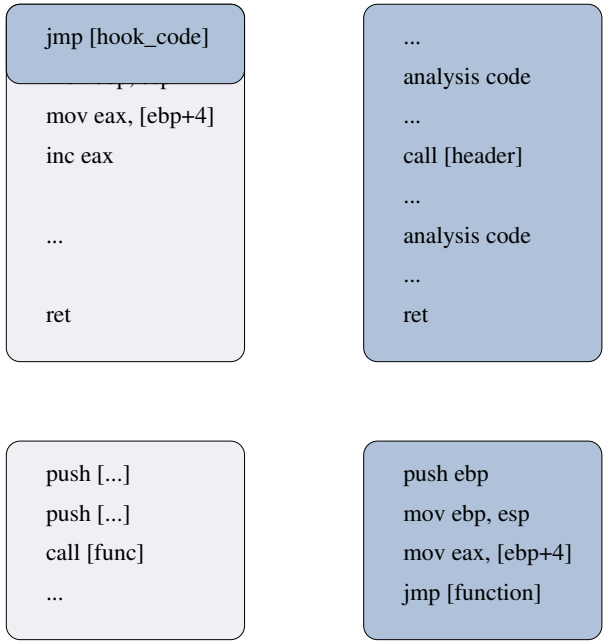


Figure: API hooking by header patching

Principle of API hooking

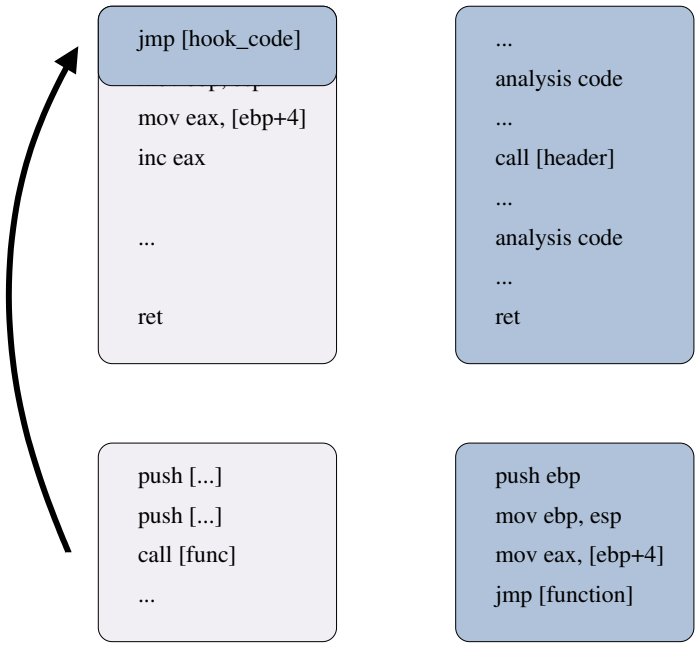


Figure: API hooking by header patching

Principle of API hooking

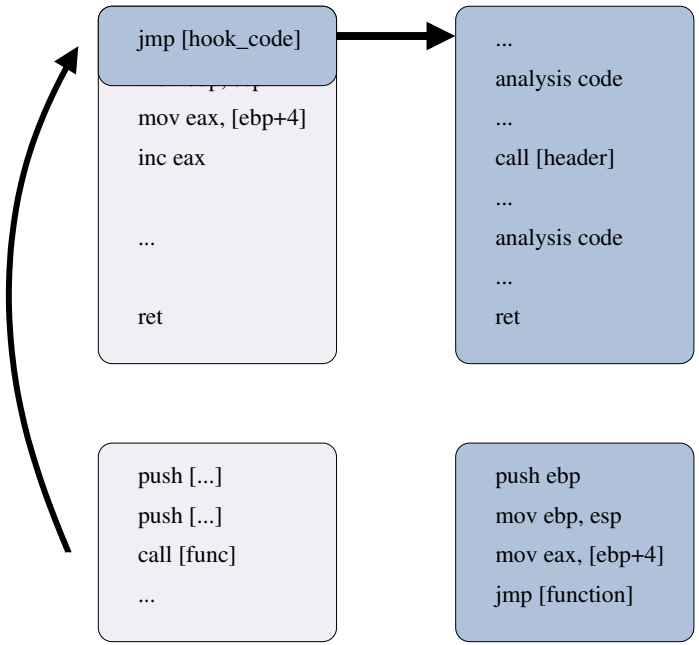


Figure: API hooking by header patching

Principle of API hooking

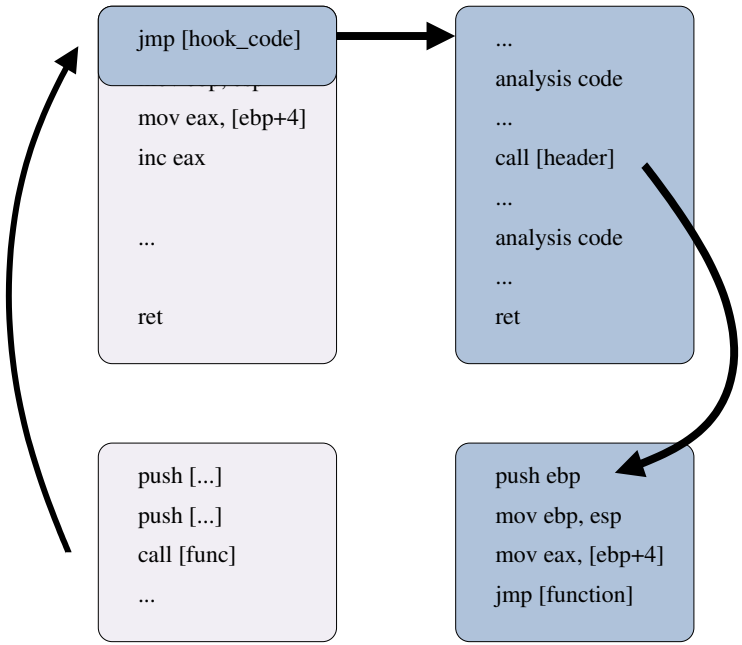


Figure: API hooking by header patching

Principle of API hooking

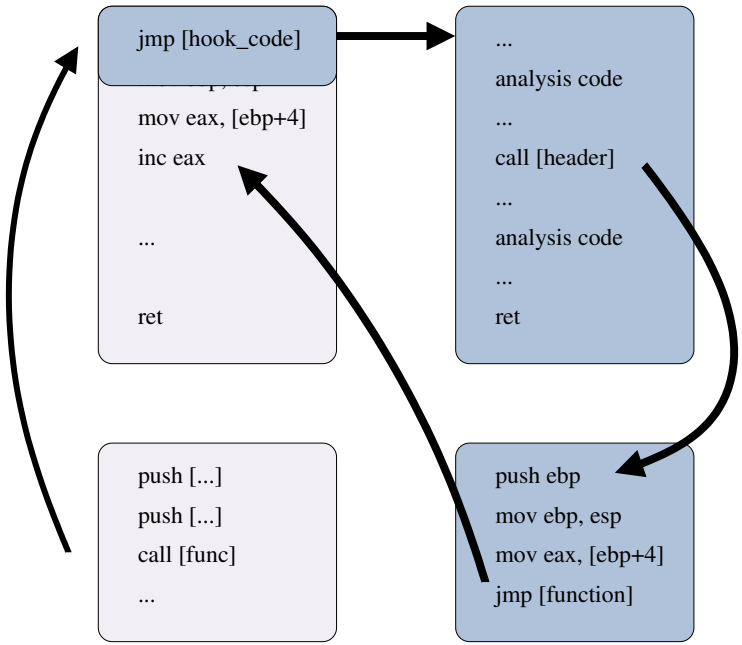


Figure: API hooking by header patching

Principle of API hooking

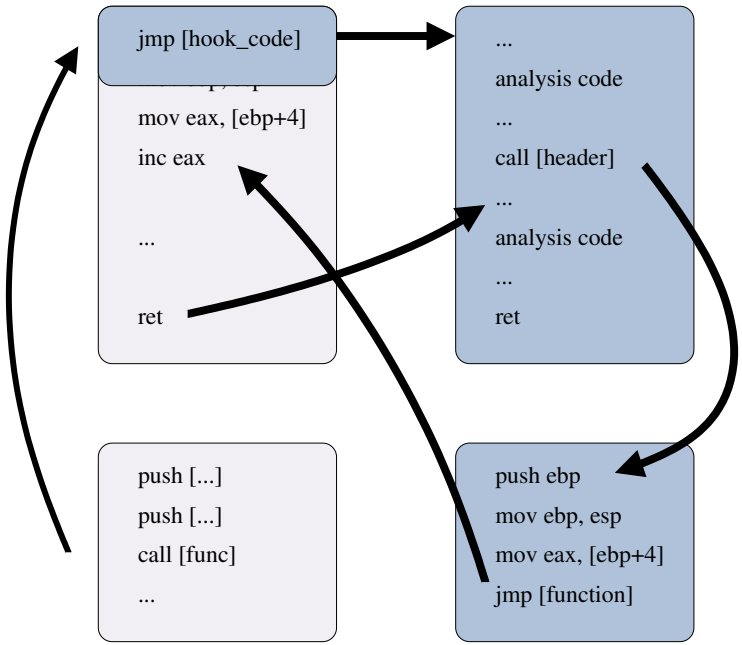


Figure: API hooking by header patching

Principle of API hooking

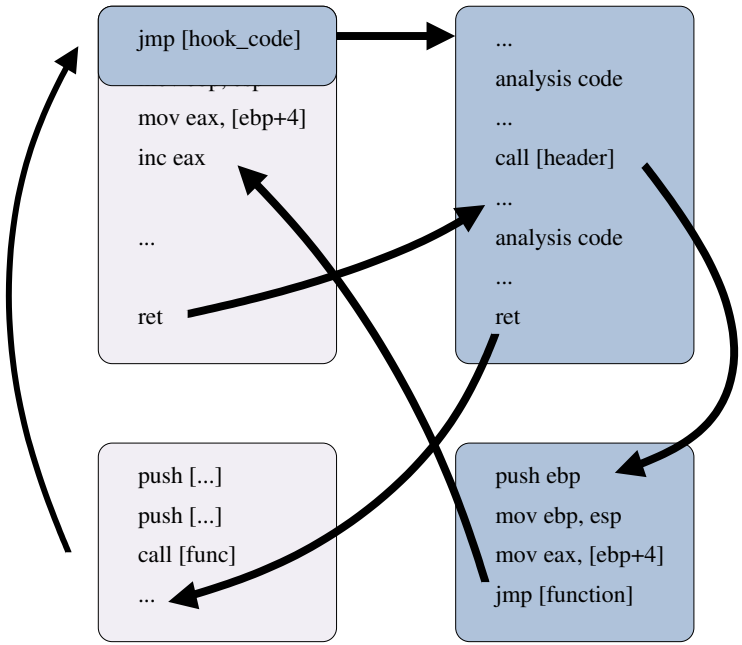


Figure: API hooking by header patching

The module "Hooker"

- "Hooker" is a module that executes API hooking by patch of function header
- calculation of instruction alignement based on z0mbie's LDE32 engine
- exports one function "HookFunctions" that allows to hook a set of functions

Web server attack in practice

Demo : Web server attack

Plan

Conclusion

- Techniques implemented in tools used in two attacks are well-known
- Interesting point : developed very quickly
  Example: integration of the AES of PolarSSL in "Loader" $\sim$ 2 hours

Future work

- Finalise the development of this version of WiShMaster (correct a few bugs)
- Try to shellcodise well-known application like netcat $\Rightarrow$ polymorphic netcat
- Develop more funny applications with WiShMaster

Thank you for your attention...

Any questions?

Shellcodisation is painless. No C code was harmed during this presentation