Windows Shellcode Mastery

BlackHat Europe 2009

Benjamin CAILLAT

ESIEA - SI&S lab

caillat[at]esiea[dot]fr

bcaillat[at]security-labs[dot]org

Plan

Plan

Plan

The use of shellcodes in virology

—

A quick reminder. . .

Reminder: PE format and creation of a process

- Under Windows, executables are in PE format (Portable Executable)
- Executables compounded of a header and several sections (code, data, resources...)
- During creation of a process, Windows loader:
  - maps sections at the right address (may contain hardcoded addresses)
  - initialises memory
  - resolves imported functions

Reminder: imported function resolution in Windows

Two mechanisms to resolve imported functions

Reminder: imported function resolution in Windows

Two mechanisms to resolve imported functions

When process is created

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

Reminder: imported function resolution in Windows

Two mechanisms to resolve imported functions

When process is created

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

During execution: "dynamic address resolution"

Executable uses two functions to resolve an imported function:

- "LoadLibrary": load a library
- "GetProcAddress": find an exported function by its name

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

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 (polymorphism)

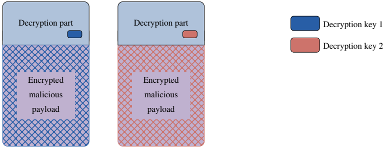

Figure: Two copies of the same virus that implements polymorphism

- 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 encrypted malicious code

Principle of execution of encrypted malware

Figure: Principle of execution of an encrypted malware

Principle of execution of encrypted malware

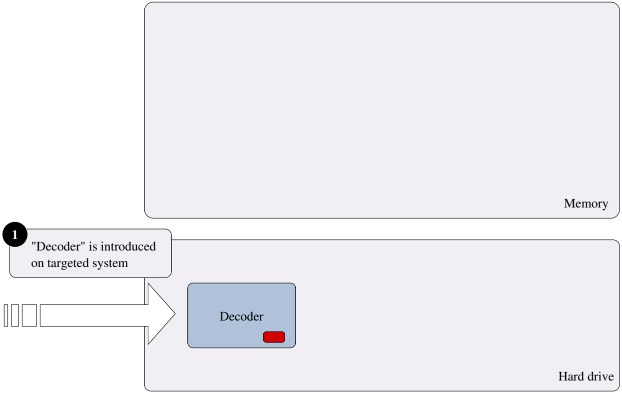

Figure: Principle of execution of an encrypted malware

Principle of execution of encrypted malware

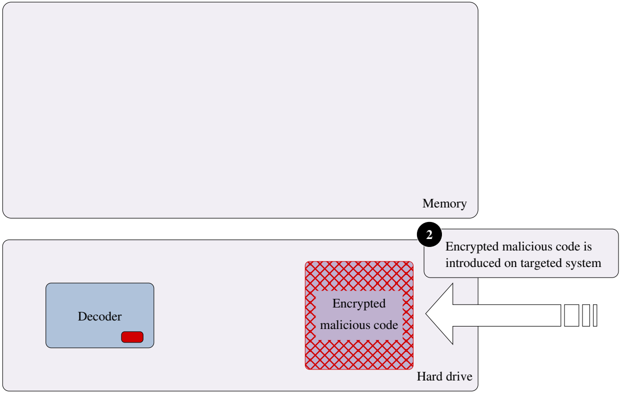

Figure: Principle of execution of an encrypted malware

Principle of execution of encrypted malware

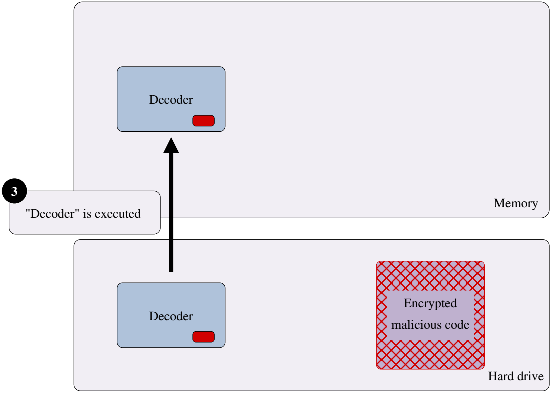

Figure: Principle of execution of an encrypted malware

Principle of execution of encrypted malware

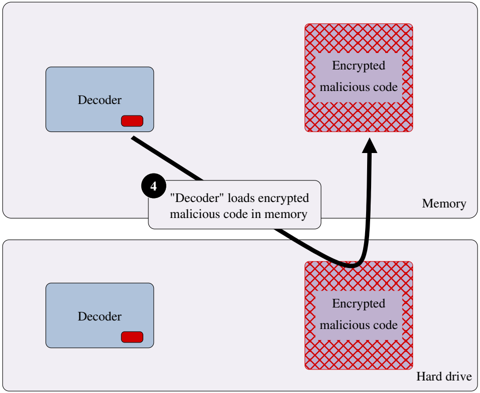

Figure: Principle of execution of an encrypted malware

Principle of execution of encrypted malware

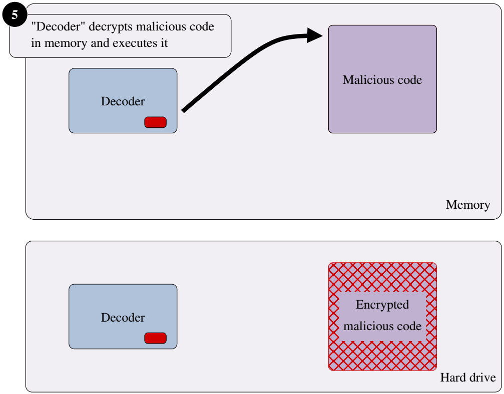

Figure: Principle of execution of an encrypted malware

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

  ⇒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 metadatas 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

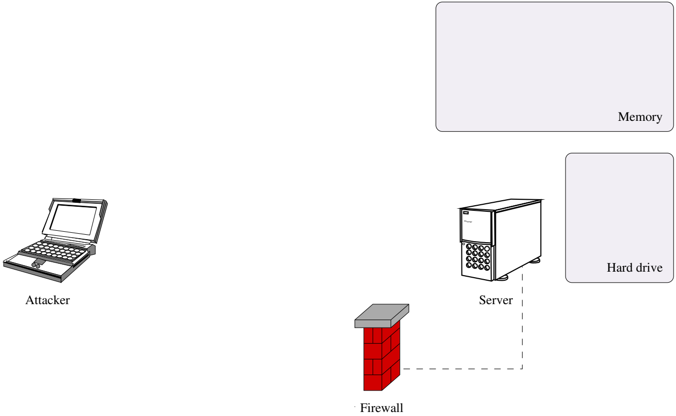

Figure: Principle of execution of malware only in memory

Principle of execution of malware only in memory

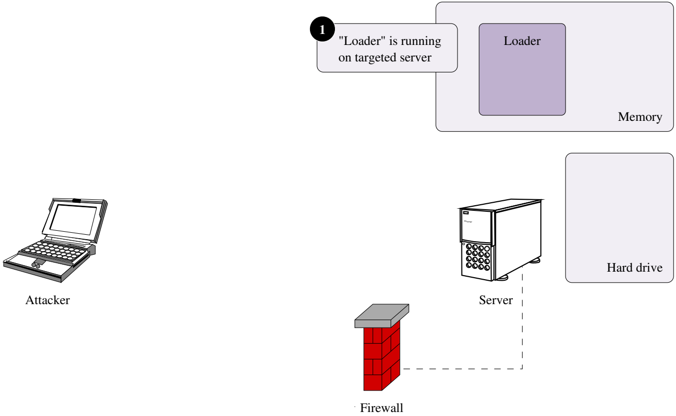

Figure: Principle of execution of malware only in memory

Principle of execution of malware only in memory

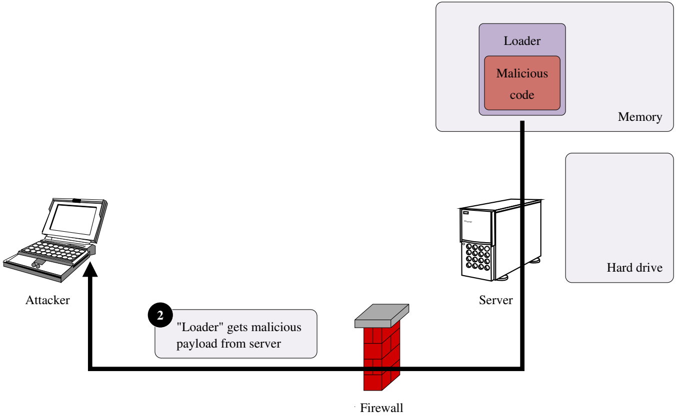

Figure: Principle of execution of malware only in memory

Principle of execution of malware only in memory

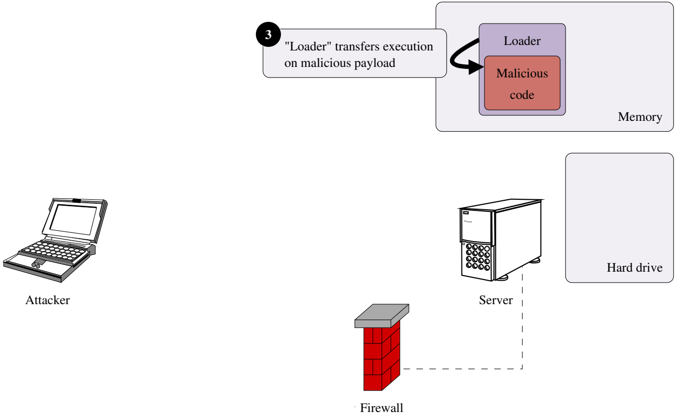

Figure: Principle of execution of malware only in memory

Execute only in memory - Implementation

- 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

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

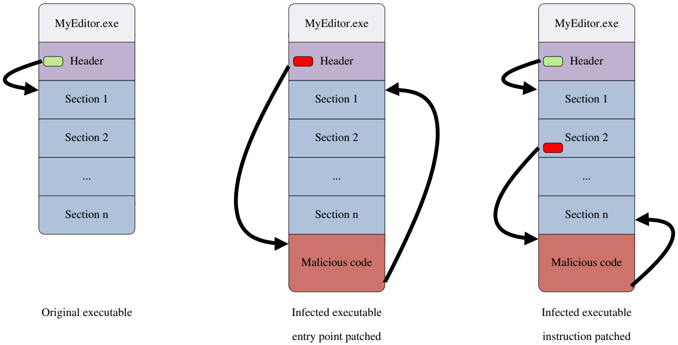

Figure: Principle of infection of an executable

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
- Intercept private data of user using infected computer: injection/API hooking/analysis of parameters
- Bypass bad implemented personal firewalls

Inject code into another process - Implementation

Code injection may be done in several ways:

- dll injection
- direct code injection

Each technique has pro and cons; we choose to use the second

Inject code into another process - Implementation

Figure: Principle of direct code injection

Inject code into another process - Implementation

Figure: Principle of direct code injection

Inject code into another process - Implementation

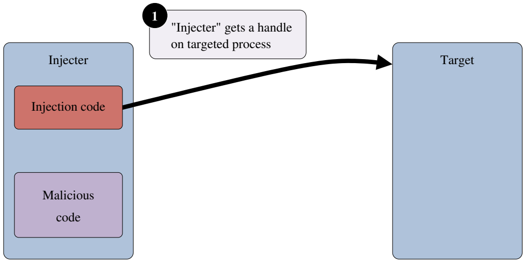

Figure: Principle of direct code injection

Inject code into another process - Implementation

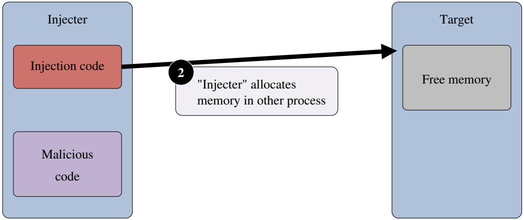

Figure: Principle of direct code injection

Inject code into another process - Implementation

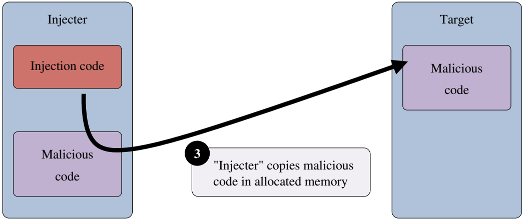

Figure: Principle of direct code injection

Inject code into another process - Implementation

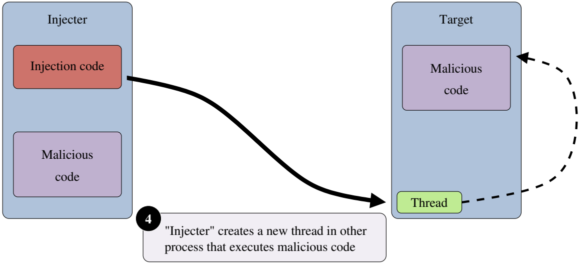

Figure: Principle of direct code injection

Inject code into another process - Implementation

- 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 ⇒ 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 ⇒ 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

⇒ 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

Objective of this part - 1

- Present an easy way to write the malicious code as a shellcode
- 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

Objective of this part - 2

- Binary code produced by normal compilation 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 distances are hardcoded
  - imported function calls rely on IAT
- Many ways to solve those problems (patch assembly, work in the stack. . .)
- Choose one technique: use a global data

Using a global data - 1

- 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

Using a global data - 2

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);
        ...
}
```

Using a global data - 3

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;
```

Using a global data - 4

Number of modifications can be considerably reduced by using C macros:

Definitions of macros

```
/* Add GLOBAL_DATA parameter in definitions of internal function */
#define DisplayFileTempDefinition(...) \\
        DisplayFileDefinition(PGLOBAL_DATA pGlobalData, __VA_ARGS__)

/* Add redirection and GLOBAL_DATA parameter in call of internal function */
#define PrintMsg(...)      pGlobalData->fp_PrintMsg(pGlobalData, __VA_ARGS__)
#define DisplayFile(...)   pGlobalData->fp_DisplayFile(pGlobalData, __VA_ARGS__)

/* Add redirection for imported functions */
#define CreateFile         pGlobalData->fp_CreateFile
#define HeapAlloc          pGlobalData->fp_HeapAlloc
#define GetProcessHeap     pGlobalData->fp_GetProcessHeap
#define ReadFile           pGlobalData->fp_ReadFile

/* Add redirection for strings */
#define STR_00000001(x)    pGlobalData->szString_00000001
```

Using a global data - 5

Patched function “DisplayFile” becomes:

Patched function DisplayFile with the macros

```
BOOL DisplayFileTempDefinition(IN CHAR * szFilePath)
{
        ...
        CreateFile(szFilePath, ...)
        pData = (UCHAR *) HeapAlloc(GetProcessHeap(), HEAP_ZERO_MEMORY, dwFileSize+1)
        ReadFile(hFile, pData, ...)
        PrintMsg(LOG_LEVEL_TRACE, STR_00000001("File successfully read: %s"), pData);
        ...
}
```

$\Rightarrow$ there are now very few modifications

Using a global data - 6

Call of the internal function "DisplayMessage"

```
        DisplayMessage(g_szMessage);
00412F99   8B45 08           MOV EAX,DWORD PTR SS:[EBP+8]  ; get address of g_szMessage in
00412F9C   05 58010000       ADD EAX,158                   ; GLOBAL_DATA
00412FA1   50                PUSH EAX                      ; push address of g_szMessage
00412FA2   8B4D 08           MOV ECX,DWORD PTR SS:[EBP+8]  ; get address of pGlobalData
00412FA5   51                PUSH ECX                      ; push address of pGlobalData
00412FA6   8B55 08           MOV EDX,DWORD PTR SS:[EBP+8]  ; get address of DisplayMessage
00412FA9   8B82 88000000     MOV EAX,DWORD PTR DS:[EDX+88]
00412FAF   FFD0              CALL EAX                      ; call DisplayMessage
```

Call of the internal function "DisplayFile"

```
        if(DisplayFile("test.txt") == FALSE)
00412FFC   8B45 08           MOV EAX,DWORD PTR SS:[EBP+8]  ; get address of pGlobalData
00412FFF   05 A1040000       ADD EAX,4A1                   ; get address of string
00413004   50                PUSH EAX                      ; push address of string
00413005   8B4D 08           MOV ECX,DWORD PTR SS:[EBP+8]  ; get address of pGlobalData
00413008   51                PUSH ECX                      ; push address of pGlobalData
00413009   8B55 08           MOV EDX,DWORD PTR SS:[EBP+8]
0041300C   8B42 78           MOV EAX,DWORD PTR DS:[EDX+78] ; get address of DisplayFile
0041300F   FFD0              CALL EAX                      ; call DisplayFile
```

Using a global data - 7

Call of the imported function "CreateFile"

```
        CreateFile(szFilePath, ...)
...
00412DE2    8B4D 08          MOV ECX,DWORD PTR SS:[EBP+8]  ; get address of pGlobalData
00412DE5    8B91 D8000000    MOV EDX,DWORD PTR DS:[ECX+D8] ; get address of CreateFile
00412DEB    FFD2             CALL EDX
```

Using a global data - 8

- Generated binary does not contain any hardcoded addresses
  ⇒ 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

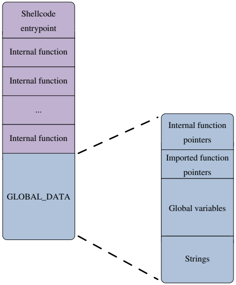

Figure: Overview of the structure of the shellcode

Summary

- This solution allows a shellcode to be created with little modification 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
  - 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

—

Presentation

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 one year
- 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 under active development
- 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

—

The shellcodisation process

The shellcodisation process in WiShMaster

Shellcodisation accomplished by WiShMaster is divided into 6 steps:

- **Analysis**: identifies code elements
- **Obtain the size of global variables**
- **Create environment**:
  - creates file global_data.h (GLOBAL_DATA structure and macros)
  - creates a patched copy of source files in a temporary directory
- **Generation**: builds patched sources, extracts binary data and generates the shellcode
- **Customization**
- **Integration**:
  - copy shellcode in a specific directory
  - or transform it in a C array and dump it in a C header file

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

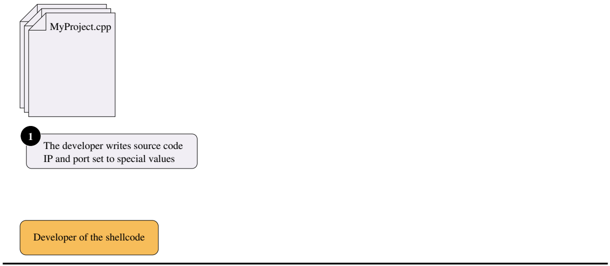

Figure: Principle of the separation between developer / user of a shellcode

The customization step - 3

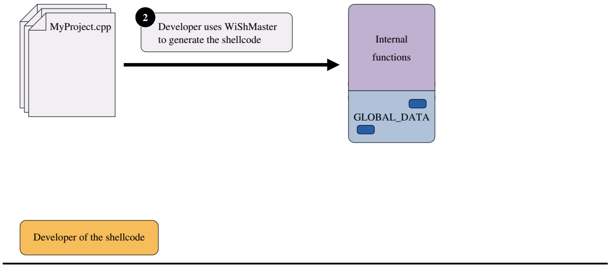

Figure: Principle of the separation between developer / user of a shellcode

The customization step - 3

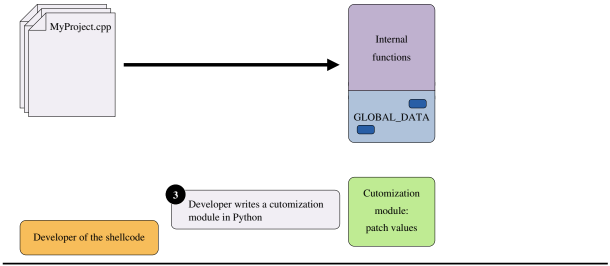

Figure: Principle of the separation between developer / user of a shellcode

The customization step - 3

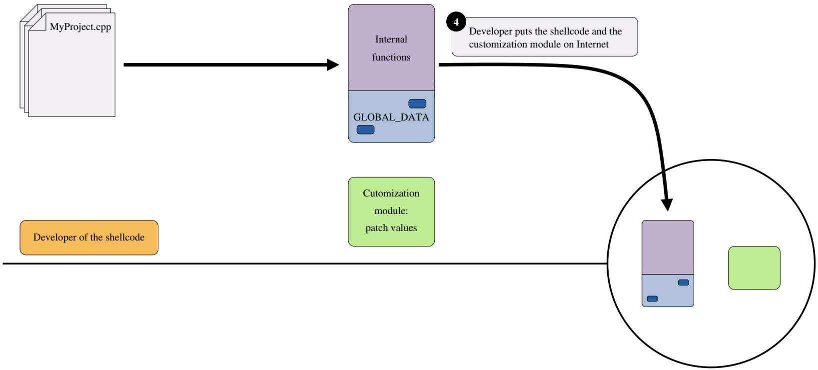

Figure: Principle of the separation between developer / user of a shellcode

The customization step - 3

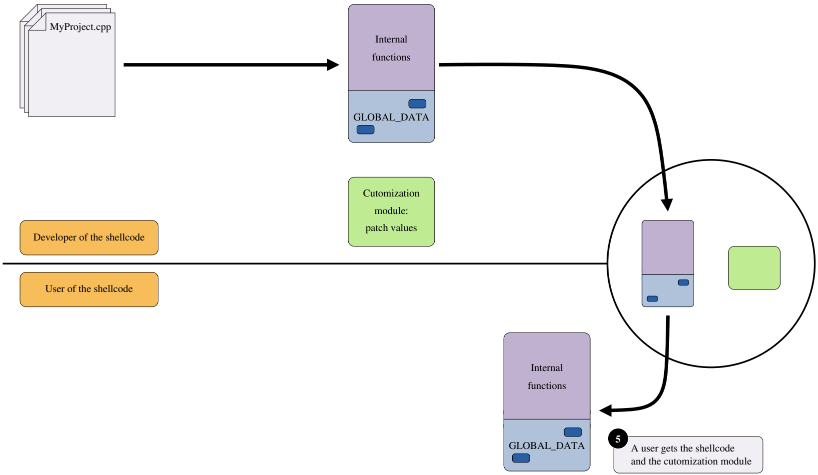

Figure: Principle of the separation between developer / user of a shellcode

The customization step - 3

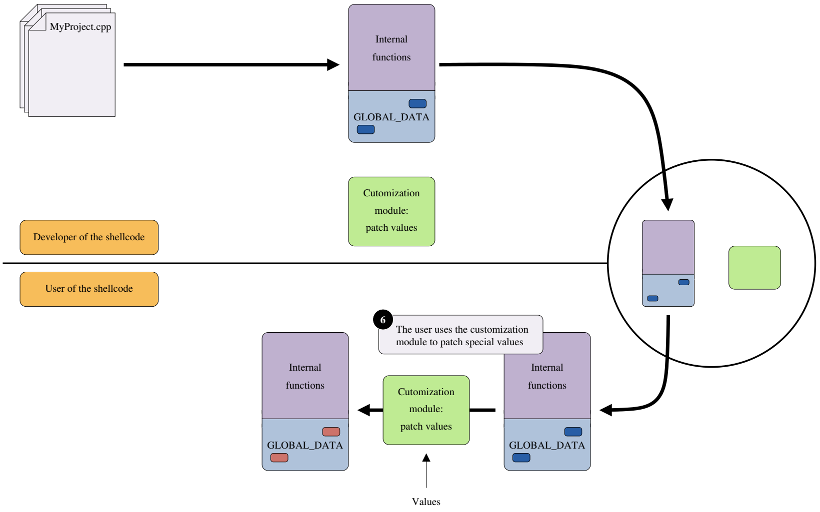

Figure: Principle of the separation between developer / user of a shellcode

The customization step - 3

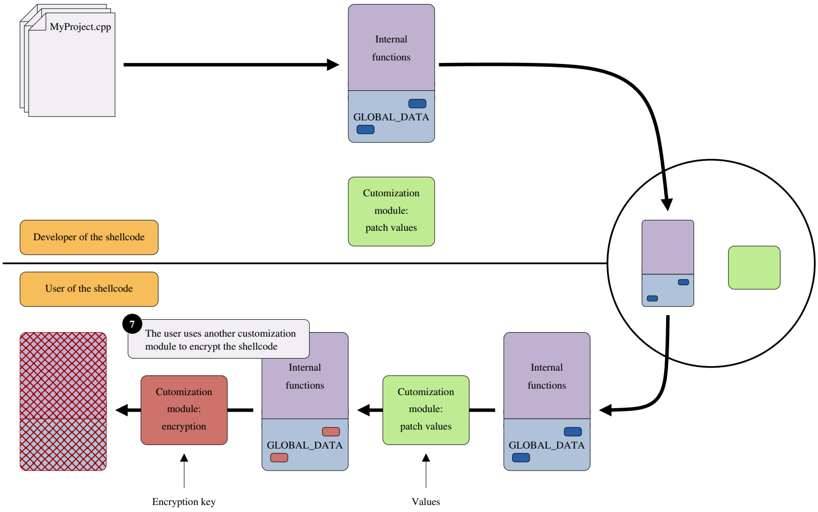

Figure: Principle of the separation between developer / user of a shellcode

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

Initialising the shellcode: principle

WiShMaster uses tips well-known by Windows shellcode writers:

- finds load address with call/pop instructions
- gets address of kernel32.dll through the PEB (Process Environment Block)
- resolves imported functions with LoadLibrary and an internal function that found the address of an exported function from a 32-bits checksum computed from its name

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

Presentation of simpletest

Very simple program:

- prints messages
- displays the content of a file “test.txt”

A few extracts of simpletest - 1

File user.h.txt

```
#define SIZE_USERNAME        32
#define SIZE_PASSWORD        32

typedef struct _USER
{
        CHAR szUsername[SIZE_USERNAME];
        CHAR szPassword[SIZE_PASSWORD];
} USER, *PUSER;
```

A few extracts of simpletest - 2

File display.cpp

```
CHAR g_szMessage[]="This is a message stored as a global variable";

VOID DisplayMessage(IN CHAR * szMessage)
{
        PrintMsg(LOG_LEVEL_TRACE, ">>> %s <<<", szMessage);
}

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);
        ...
}

BOOL DisplayData(VOID)
{
        DisplayMessage(g_szMessage);
        PrintMsg(LOG_LEVEL_TRACE, "Username: %s", g_User.szUsername);
        PrintMsg(LOG_LEVEL_TRACE, "Password: %s", g_User.szPassword);
        if(DisplayFile("test.txt") == FALSE)
                return FALSE;
        return TRUE;
}
```

A few extracts of simpletest - 3

File main.cpp

```
USER g_User ={"jmerchat","password"};

BOOL DisplayData(VOID);

int main(int argc, char * argv[])
{
        DisplayUser();
        return 0;
}
```

A few extracts of simpletest - 4

File print_msg.cpp

```
VOID PrintMsg(IN UINT uiMessageLevel, IN const CHAR * fmt, ...)
{
        CHAR szBuffer[SIZE_OF_LOCAL_LOG_BUFFER+1];

        UINT i = 0;
        if(uiMessageLevel == LOG_LEVEL_ERROR)
                i += _snprintf(&szBuffer[i], SIZE_OF_LOCAL_LOG_BUFFER-i, "[ERROR] : ");
        else if(uiMessageLevel == LOG_LEVEL_WARNG)
                ...

        va_list ap;
        va_start(ap, fmt);
        i += _vsnprintf(&szBuffer[i], SIZE_OF_LOCAL_LOG_BUFFER-i, fmt, ap);
        va_end(ap);

        printf("[%.4d] %s\n ", GetCurrentThreadId() , szBuffer);
        fflush(stdout);
}
```

A few extracts of simpletest - 5

To sum up, “simpletest” contains:

- 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...

⇒ not really useful but contains most elements of C program

Demonstrations

- Video “simpletest_exe.avi”: generation of “simpletest” as an executable
- Video “simpletest_shellcode.avi”: 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

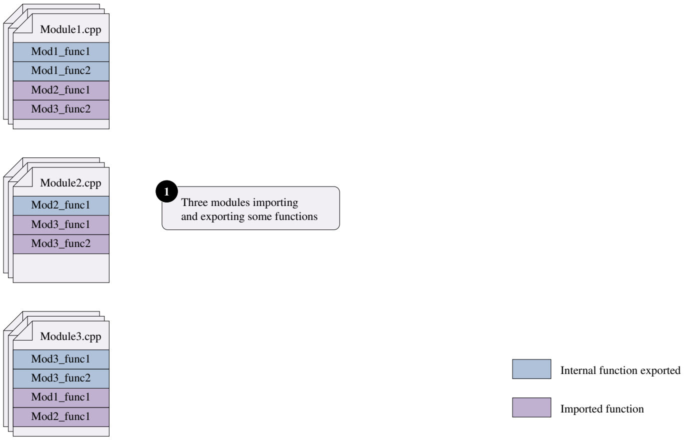

Figure: Structure of an application developed with WiShMaster v2

Overview of the application structure - 2

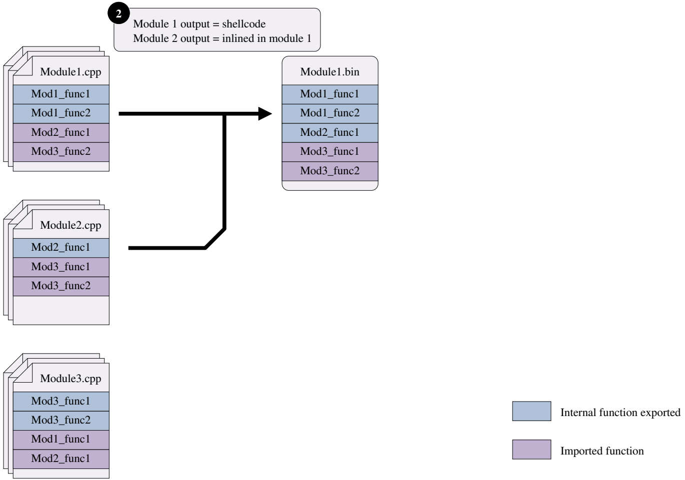

Figure: Structure of an application developed with WiShMaster v2

Overview of the application structure - 2

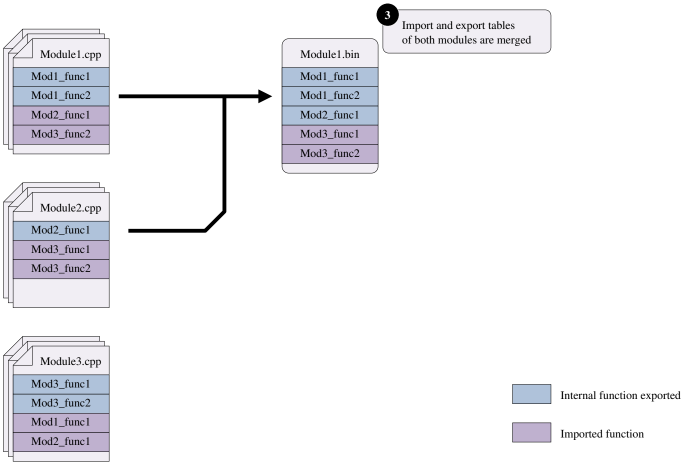

Figure: Structure of an application developed with WiShMaster v2

Overview of the application structure - 2

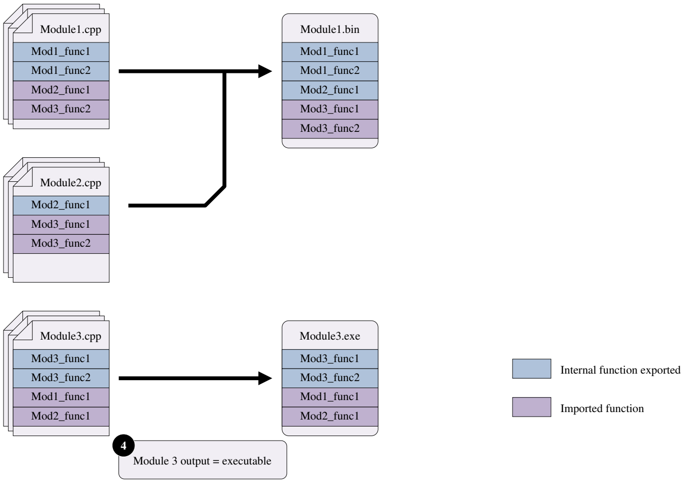

Figure: Structure of an application developed with WiShMaster v2

Overview of the application structure - 2

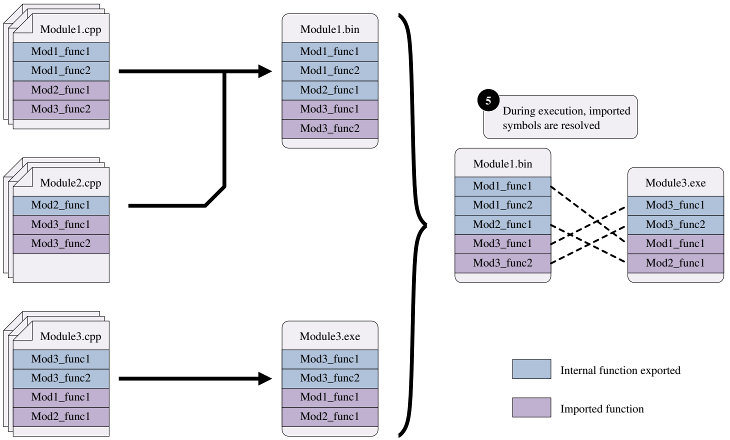

Figure: Structure of an application developed with WiShMaster v2

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 checksum of the names of each imported function
- an optional entry point: pointer on an internal function that must be called after module initialisation

Standard modules - 1

Presentation

WiShMaster comes with a few standard modules = modules that expose some functions frequently used by other modules

Standard modules - 1

Presentation

WiShMaster comes with a few standard modules = modules that expose some functions frequently used by other modules

Module "Log"

Exposes a function "PrintMsg" which allows the print of formatted messages

Standard modules - 1

Presentation

WiShMaster comes with a few standard modules = modules that expose some functions frequently used by other modules

Module "Log"

Exposes a function "PrintMsg" which allows the print of formatted messages

Module "InitSh"

Exposes all the functions needed to initialise a shellcode (notably InitialiseShellcode and GetProcAddressByCksumInDll)

Standard modules - 2

Module “Loader”

- Manages a set of modules
- Exposes a function “AddModuleToLoad”: handles all the load and the initialisation of a module (dll, shellcode, executable):
  - loads the module in memory
  - decrypts the module if this one is an encrypted shellcode
  - resolves all imported symbols (from standard libraries or other modules)
  - calls the entry point
- Note: “Loader” inlines “InitSh”

Shellcode encryption - Two kinds of encryption. . .

- "Loader" can handle shellcodes encrypted in AES-CBC with a 256-bits key
- Two kinds of encryption:
  - One secret key: all modules are encrypted with a secret key stored in "Loader"
  - Shared secret key

Shellcode encryption - Principle of shared secret key

- Following algorithm is used:
  - each module has a 256-bits private key
  - the shared key is the sum byte to byte of all private keys
  - all modules are encrypted with the final shared key
  - all modules contain their own private key (in clear)
- All modules are required to compute shared key
- Having N-1 private keys does not give any information on shared key

Plan

Plan

Demonstration: 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 compounded 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

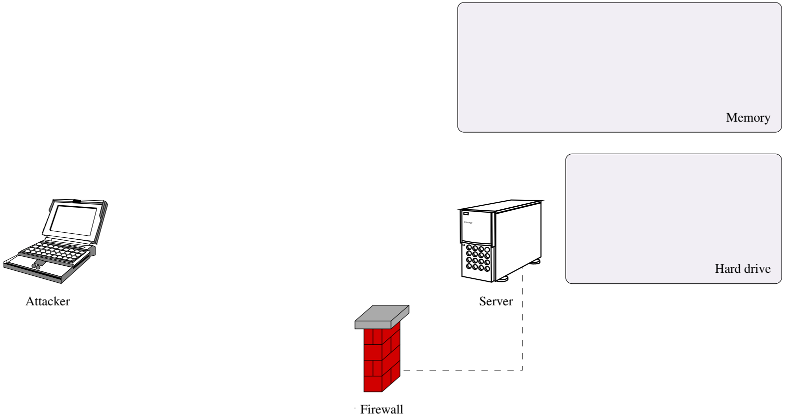

Figure: Working principle of RvShell

Presentation of RvShell - 2

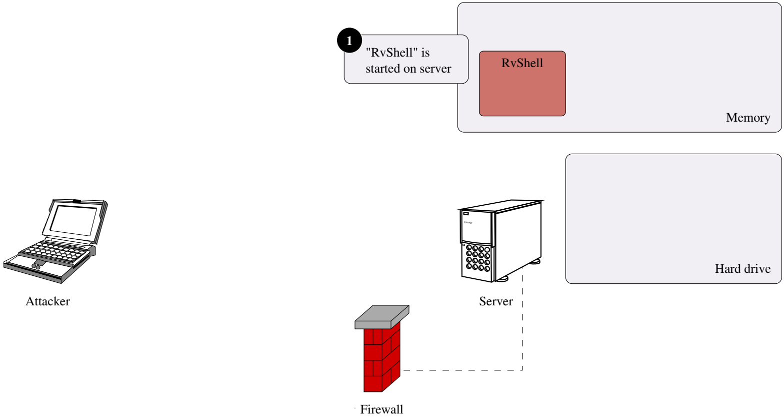

Figure: Working principle of RvShell

Presentation of RvShell - 2

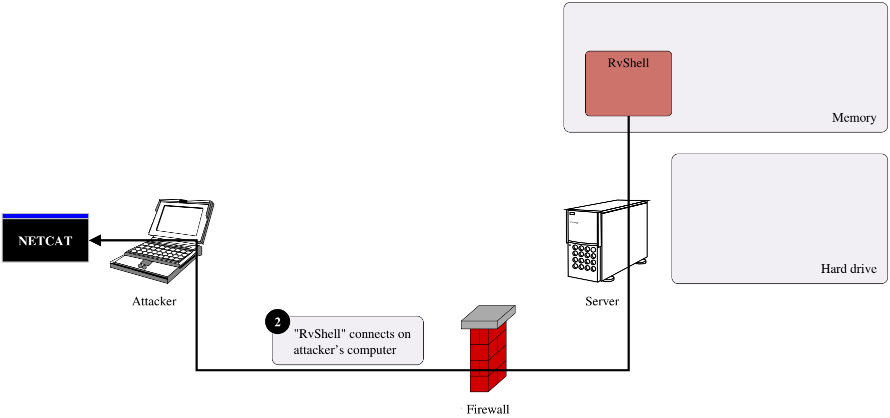

Figure: Working principle of RvShell

Presentation of RvShell - 2

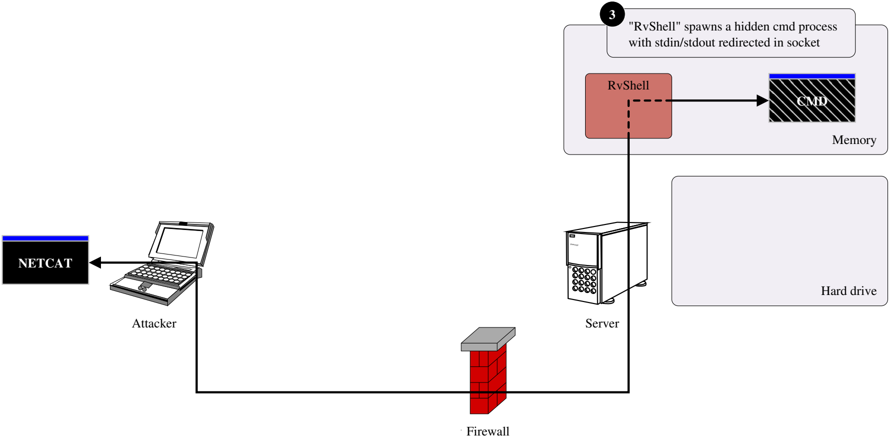

Figure: Working principle of RvShell

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 - 1

Configuration file used to generate RvShell as an executable

```
<solution>
        <module name="rvshell" config="rvshell/rvshell.cfg" input_type="code"
             specific_config="" output_type="exe"/>
        <module name="ntstacksmpl" config="ntstacksmpl/ntstacksmpl.cfg" specific_config=""
             input_type="code" output_type="inline"  inline_destination="rvshell"/>
        <module name="log" config="log/log.cfg" specific_config="" input_type="code"
             output_type="inline" inline_destination="rvshell"/>
</solution>
```

Generating RvShell as an executable - 2

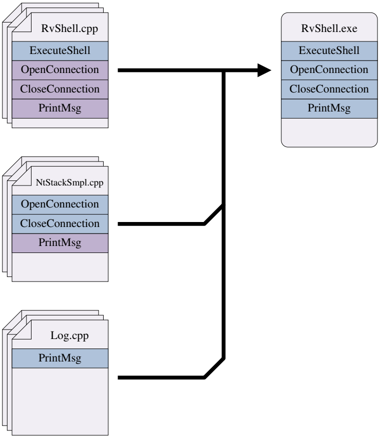

Figure: Result of the creation of the reverse shell as an executable

Generating a polymorphic RvShell - 1

“RvShell” is generated as a shellcode and then included in an executable that decrypts RvShell and jumps on it

Configuration file used to generate RvShell as a shellcode

```
<solution>
        <module name="rvshell" config="rvshell/rvshell.cfg" specific_config=""
             input_type="code" output_type="="shellcode"/>
        <module name="ntstacksmpl" config="ntstacksmpl/ntstacksmpl.cfg" specific_config=""
             input_type="code" output_type="="inline"  inline_destination="="rvshell"/>
        <module name="initsh" config="initsh/initsh.cfg" specific_config=""
             output_type="="inline" inline_destination="="rvshell"/>
        <module name="log" config="log/log.cfg" specific_config="" input_type="code"
             output_type="="inline" inline_destination="="rvshell" />
</solution>
```

Generating a polymorphic RvShell - 2

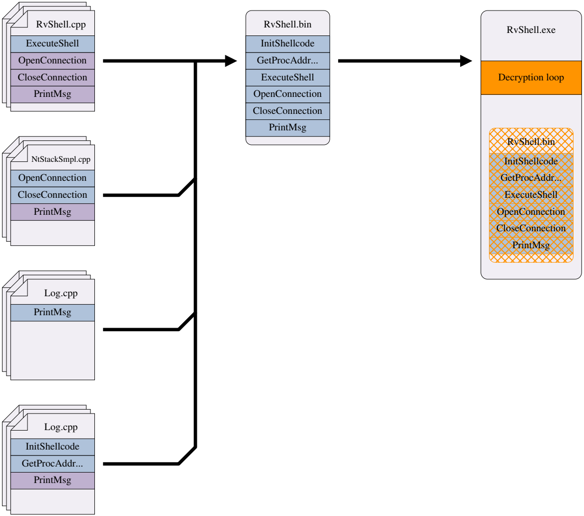

Figure: Result of the creation of a polymorphic reverse shell

Plan

Demonstration: 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

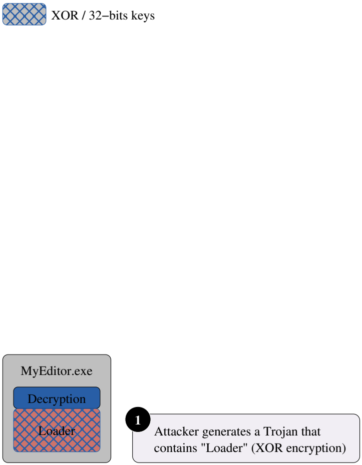

Figure: Principle of the attack with RvShell

Principle of the attack

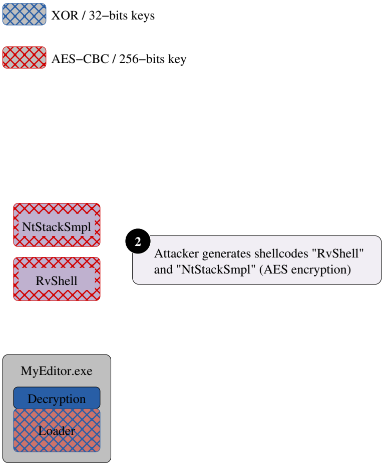

Figure: Principle of the attack with RvShell

Principle of the attack

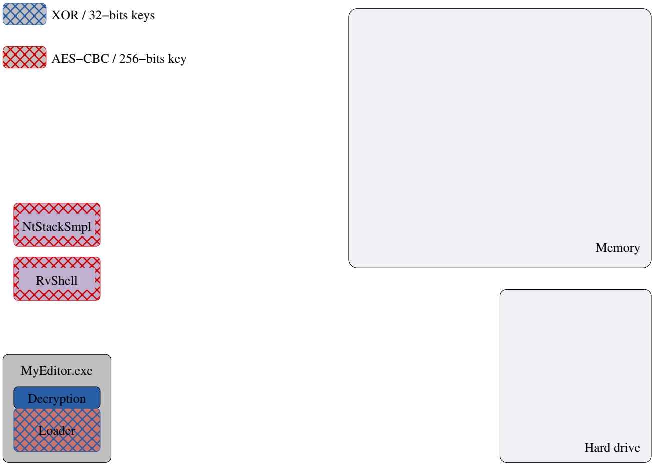

Figure: Principle of the attack with RvShell

Principle of the attack

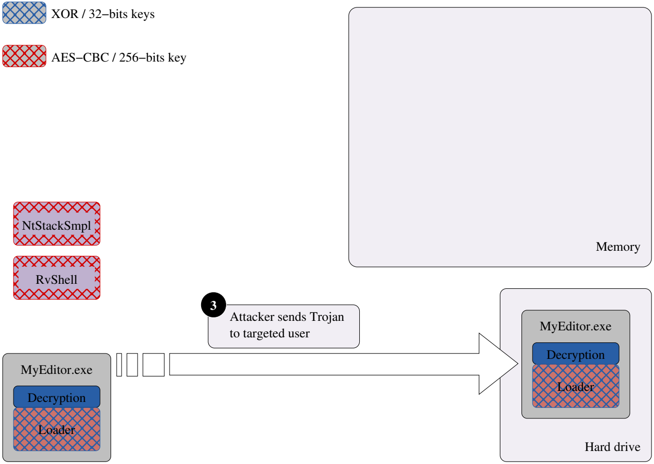

Figure: Principle of the attack with RvShell

Principle of the attack

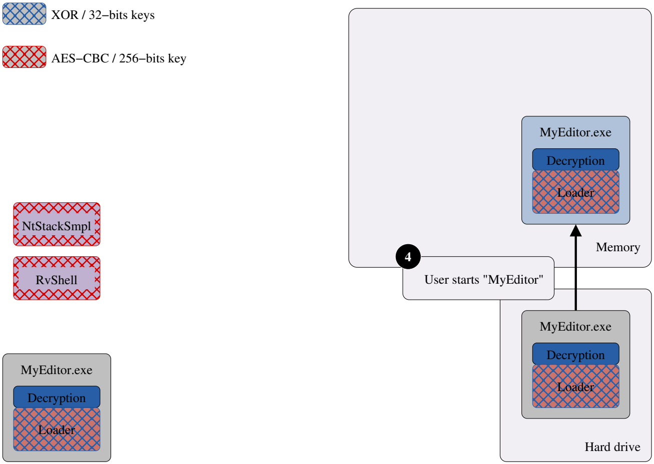

Figure: Principle of the attack with RvShell

Principle of the attack

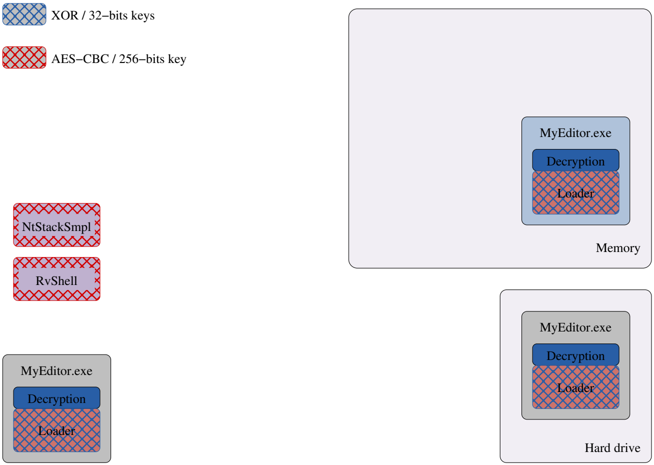

Figure: Principle of the attack with RvShell

Principle of the attack

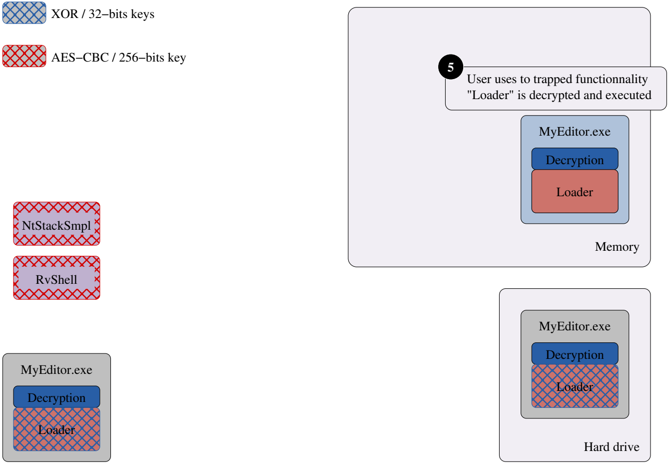

Figure: Principle of the attack with RvShell

Principle of the attack

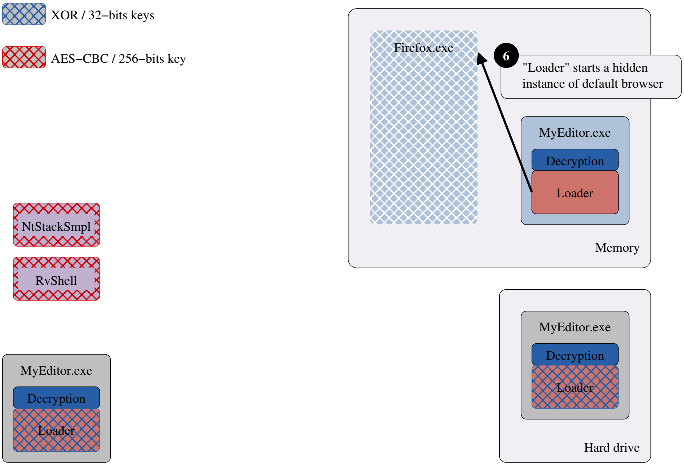

Figure: Principle of the attack with RvShell

Principle of the attack

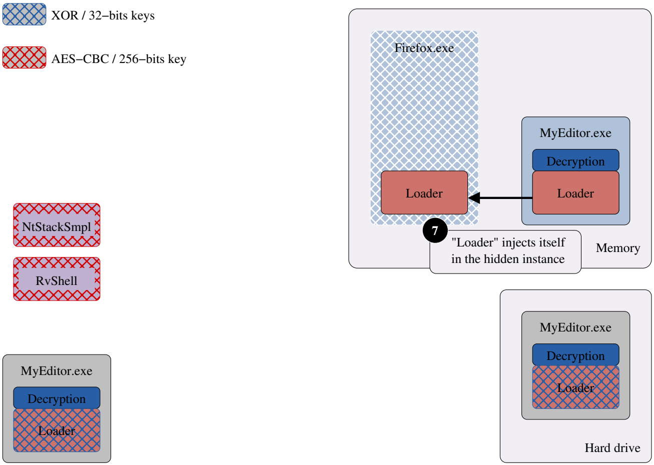

Figure: Principle of the attack with RvShell

Principle of the attack

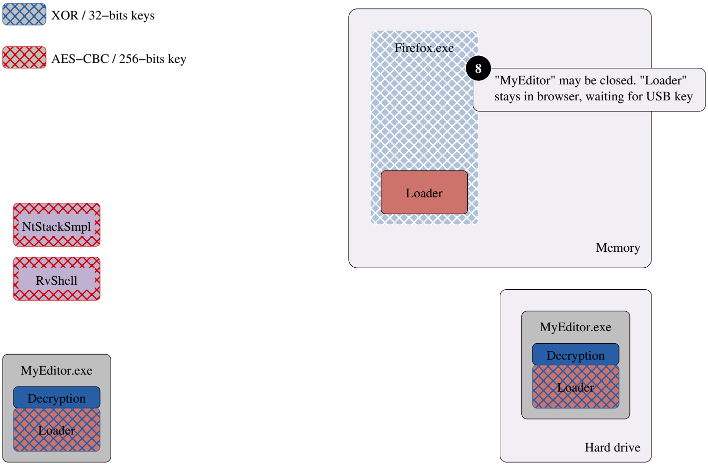

Figure: Principle of the attack with RvShell

Principle of the attack

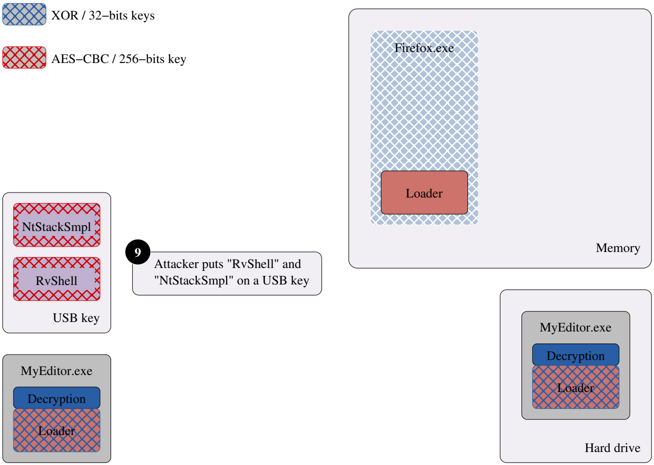

Figure: Principle of the attack with RvShell

Principle of the attack

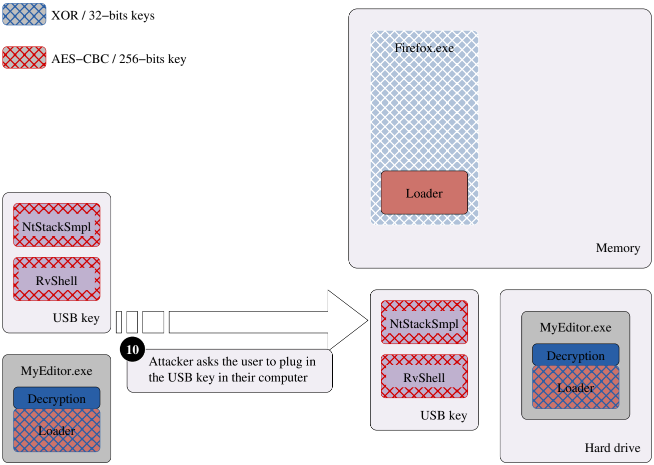

Figure: Principle of the attack with RvShell

Principle of the attack

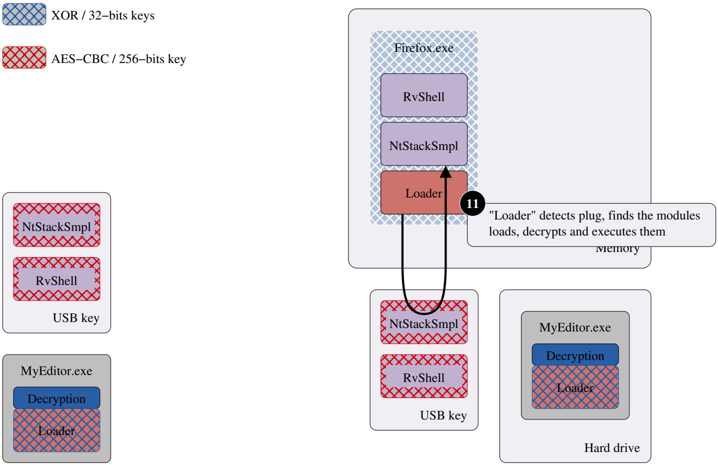

Figure: Principle of the attack with RvShell

Principle of the attack

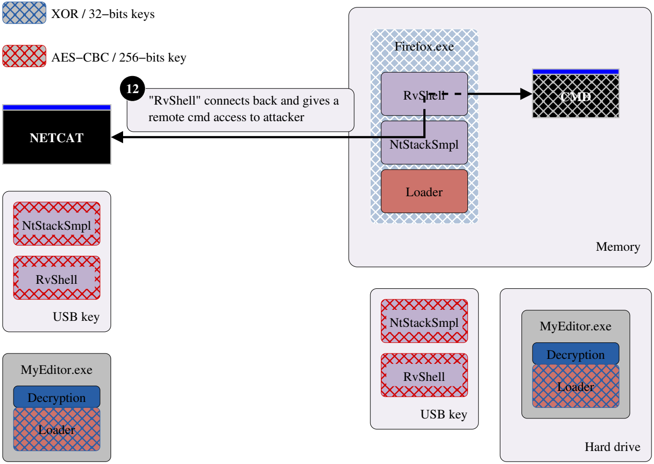

Figure: Principle of the attack with RvShell

Preparing attack - Generation of secret keys

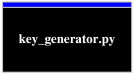

Figure: Principle of the generation of 256-bits keys

Preparing attack - Generation of secret keys

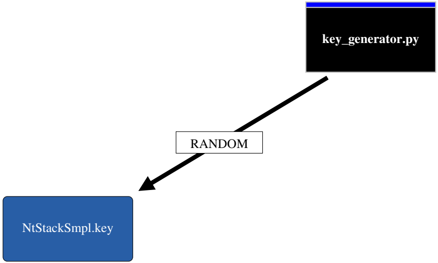

Figure: Principle of the generation of 256-bits keys

Preparing attack - Generation of secret keys

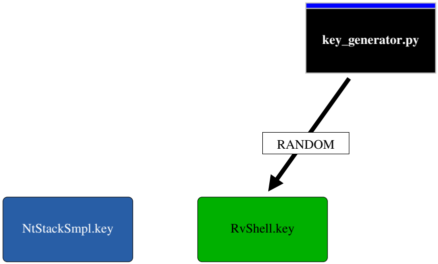

Figure: Principle of the generation of 256-bits keys

Preparing attack - Generation of secret keys

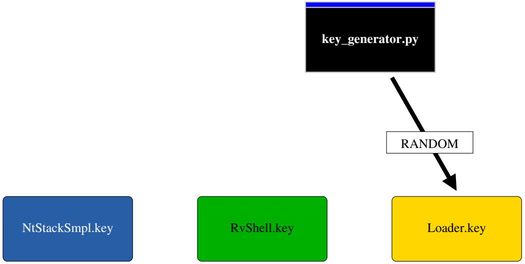

Figure: Principle of the generation of 256-bits keys

Preparing attack - Generation of secret keys

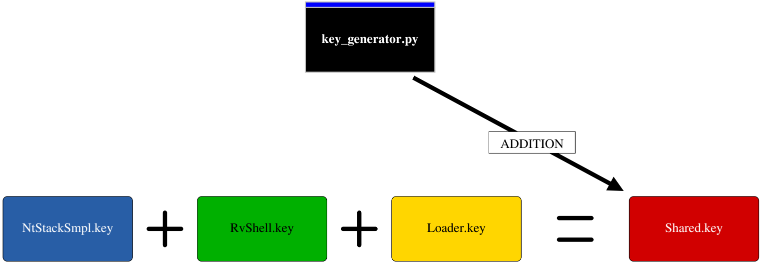

Figure: Principle of the generation of 256-bits keys

Preparing attack - key generation

Video "rvshell_1_genkey.avi": generation of encryption keys

Preparing attack - Generation of Loader

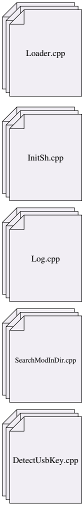

Figure: Generation of Loader

Preparing attack - Generation of Loader

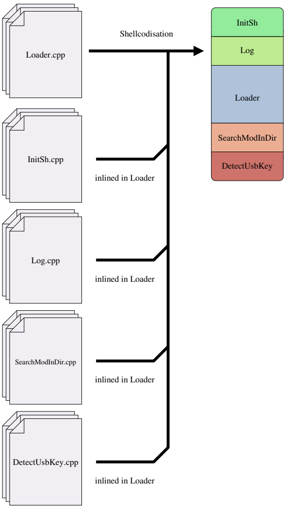

Figure: Generation of Loader

Preparing attack - Generation of Loader

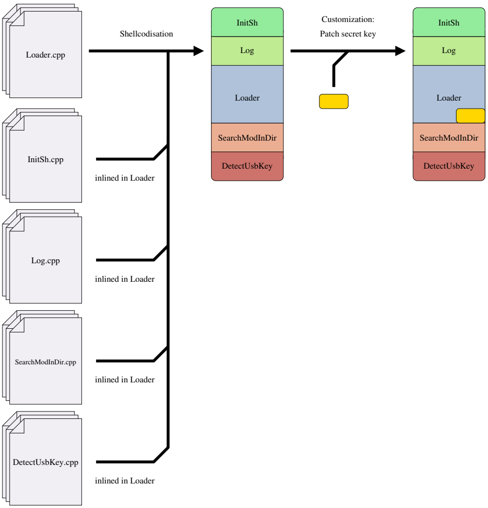

Figure: Generation of Loader

Preparing attack - generation of customized loader

video “rvshell_2_genloader.avi”: generation of customized loader

Preparing attack - Generation of RvShell and NtStackSmpl

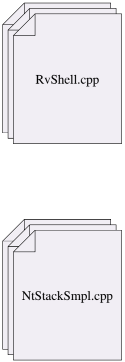

Figure: Generation of RvShell and NtStackSmpl

Preparing attack - Generation of RvShell and NtStackSmpl

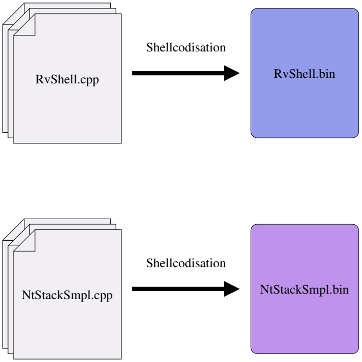

Figure: Generation of RvShell and NtStackSmpl

Preparing attack - Generation of RvShell and NtStackSmpl

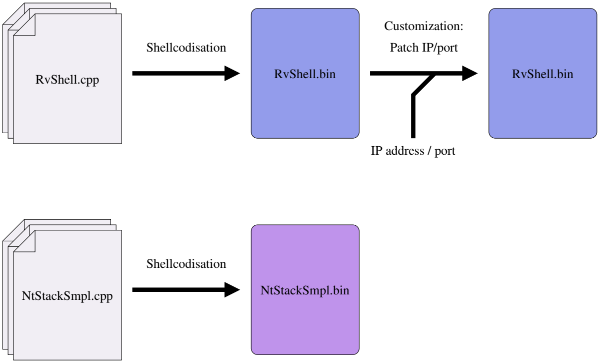

Figure: Generation of RvShell and NtStackSmpl

Preparing attack - Generation of RvShell and NtStackSmpl

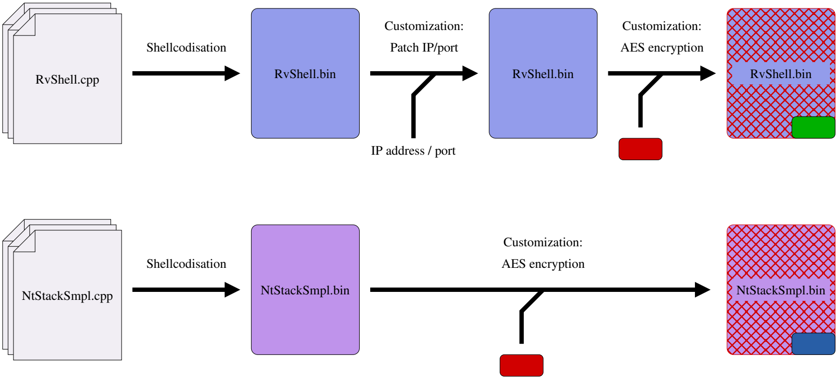

Figure: Generation of RvShell and NtStackSmpl

Preparing attack - generation of shellcode RvShell

video “rvshell_3_genrvshell.avi”: generation of shellcode RvShell

Preparing attack - Generation of Injecter

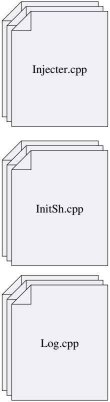

Figure: Generation of Injecter

Preparing attack - Generation of Injecter

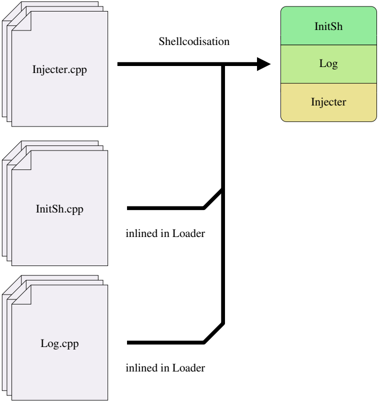

Figure: Generation of Injecter

Preparing attack - Generation of Injecter

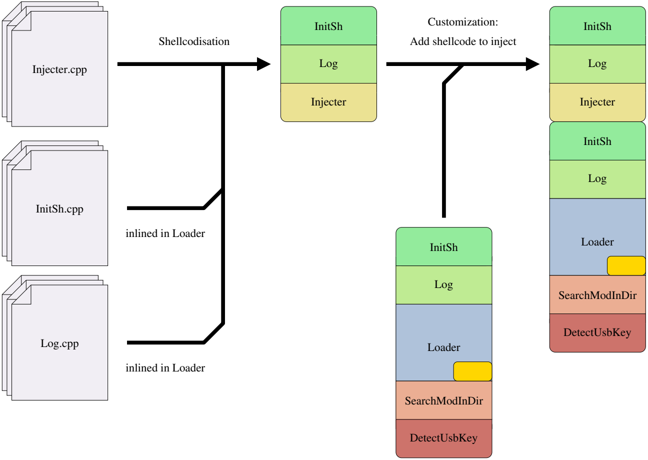

Figure: Generation of Injecter

Preparing attack - Generation of Injecter

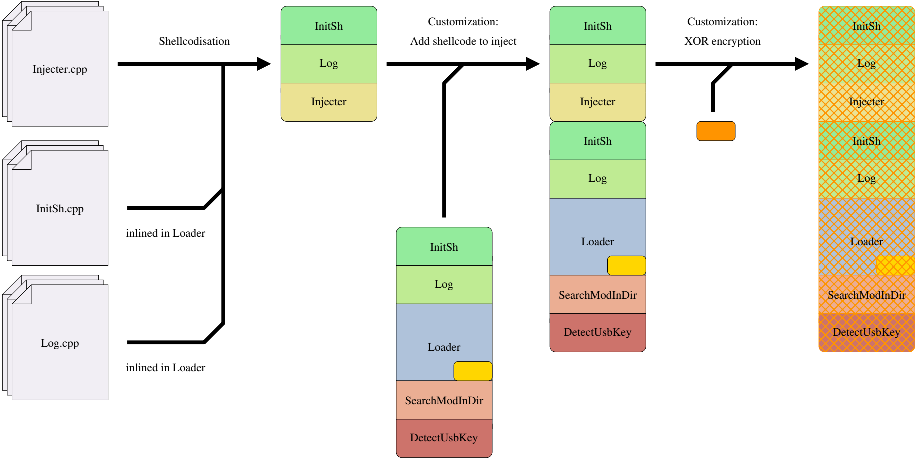

Figure: Generation of Injecter

Preparing attack - generation of injecter

video “rvshell_4_geninjecter.avi”: generation of injecter

Preparing attack - Generation of the Trojan

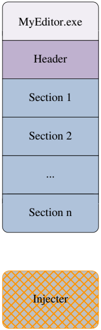

Figure: Generation of the Trojan

Preparing attack - Generation of the Trojan

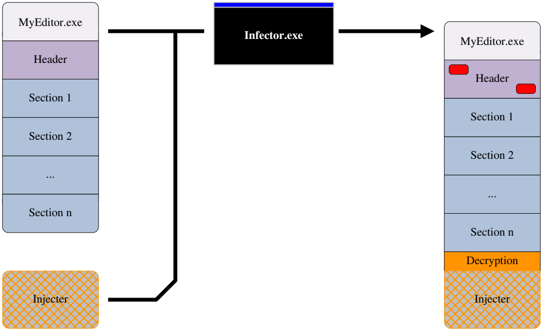

Figure: Generation of the Trojan

Preparing attack - Generation of the Trojan

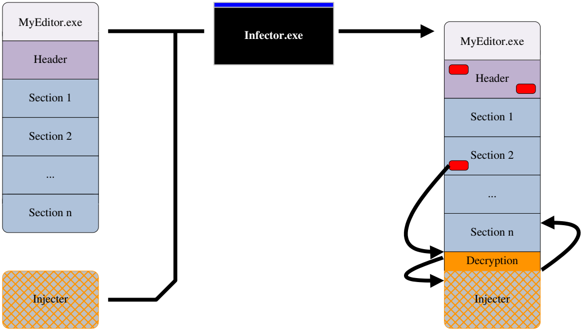

Figure: Generation of the Trojan

Preparing attack - generation of the Trojan

video “rvshell_5_gentrojan.avi”: generation of the Trojan

Attack - execution of Trojan

video "rvshell_6_executetrojan.avi": execution of Trojan

Attack - execution of RvShell

video “rvshell_7_executervshell.avi”: execution of RvShell

Attack - summary

Techniques used during this attack:

- Encryption of malicious payload:
  - "Injecter" in "MyEditor": polymorphism
  - "NtStackSmpl" and "RvShell": shared secret
- Execution only in memory : "NtStackSmpl" and "RvShell" loaded from USB key and 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 2003
- Two services:
  - Apache 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 2003
- Two services:
  - Apache 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" ⇒ 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

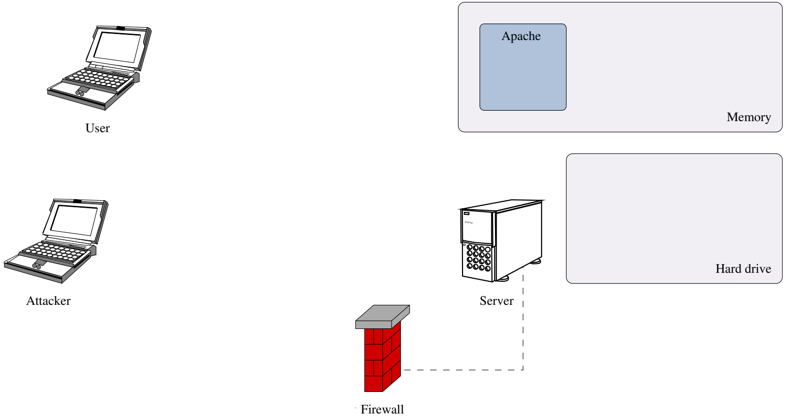

Figure: Principle of web server attack with WebDoor

Principle of web server attack

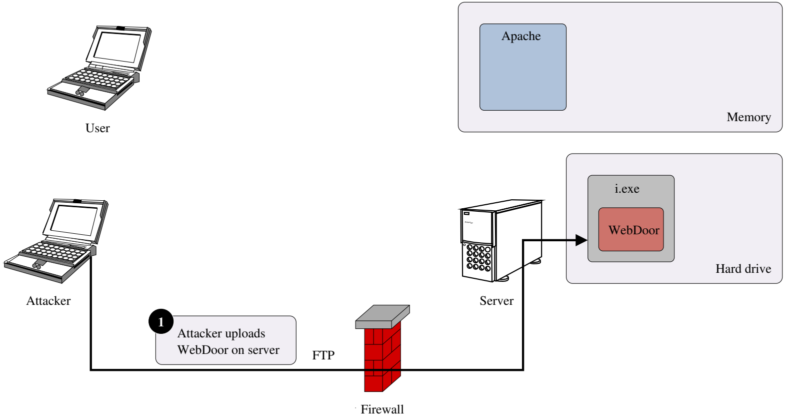

Figure: Principle of web server attack with WebDoor

Principle of web server attack

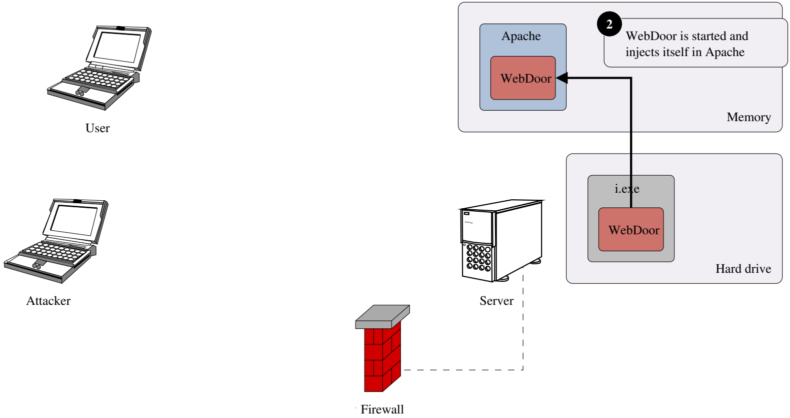

Figure: Principle of web server attack with WebDoor

Principle of web server attack

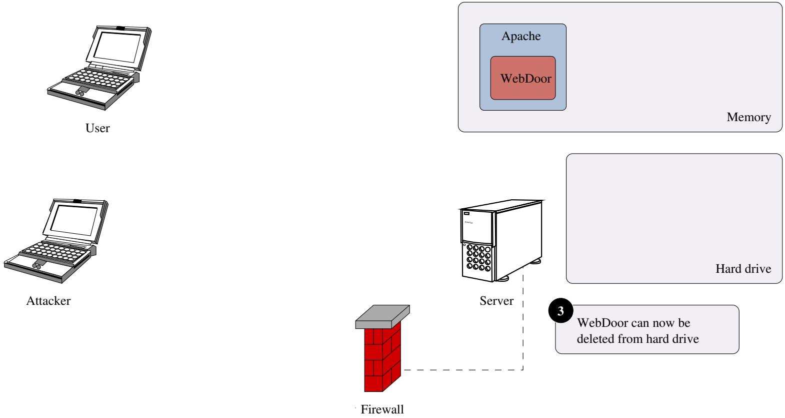

Figure: Principle of web server attack with WebDoor

Principle of web server attack

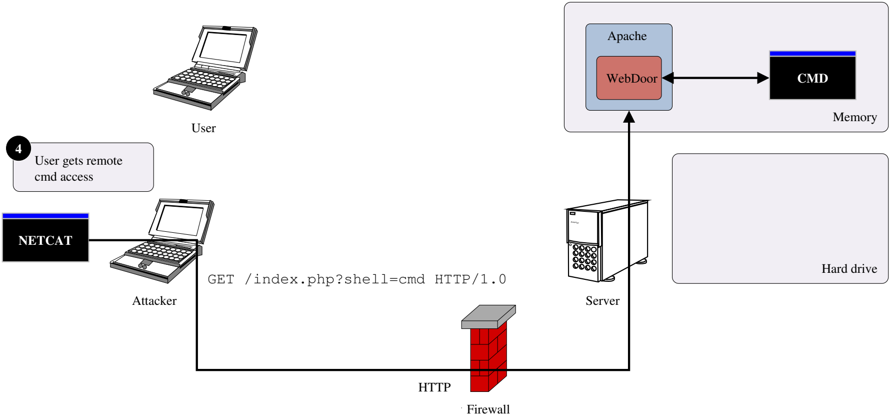

Figure: Principle of web server attack with WebDoor

Principle of web server attack

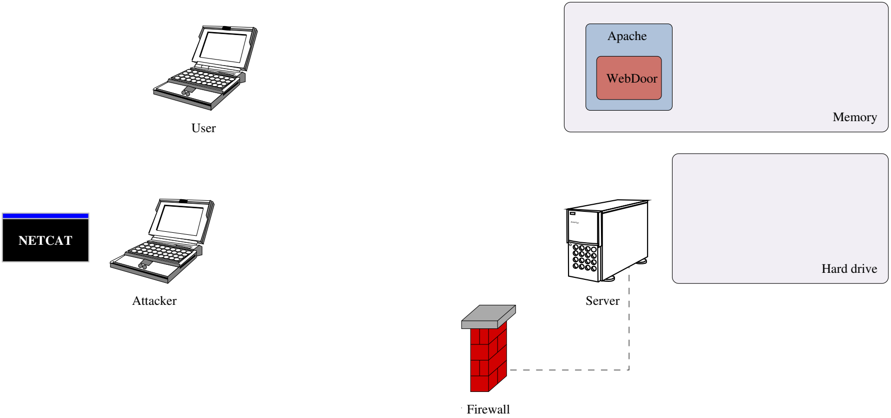

Figure: Principle of web server attack with WebDoor

Principle of web server attack

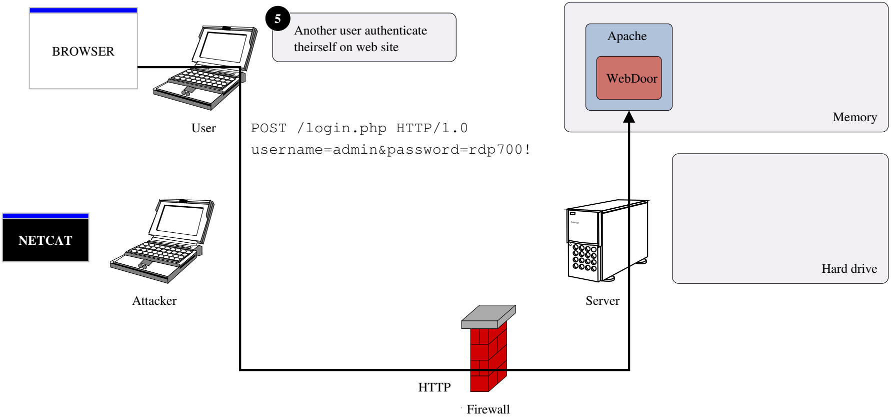

Figure: Principle of web server attack with WebDoor

Principle of web server attack

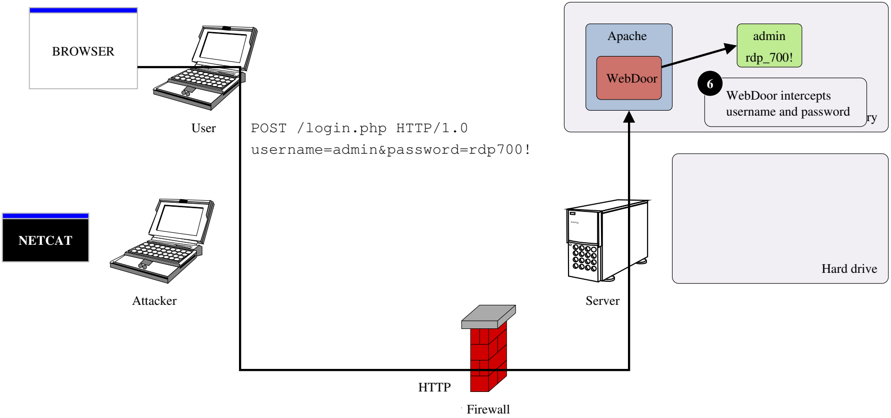

Figure: Principle of web server attack with WebDoor

Principle of web server attack

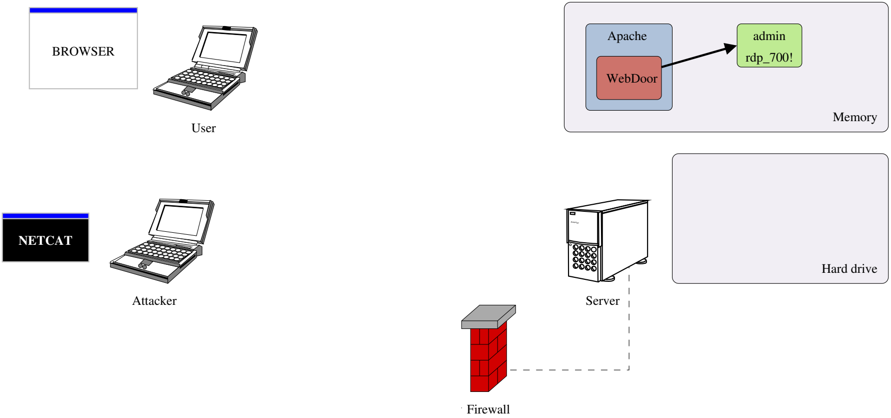

Figure: Principle of web server attack with WebDoor

Principle of web server attack

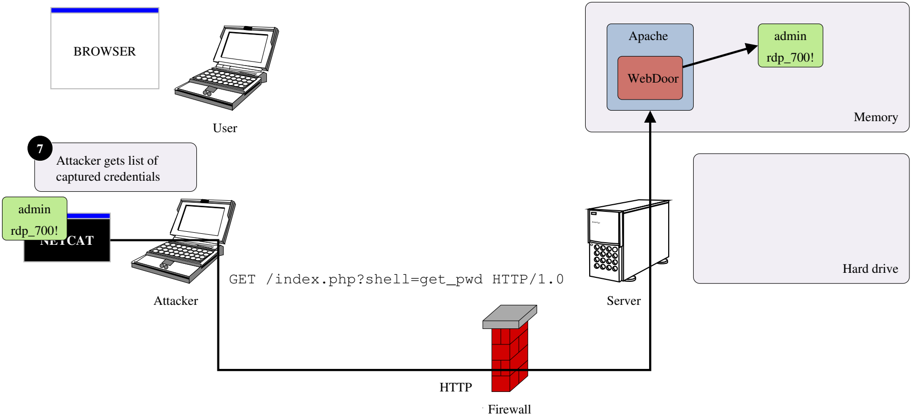

Figure: Principle of web server attack with WebDoor

Demonstration

- Video “webdoor_1_presentation.avi”: quick presentation of architecture
- Video “webdoor_2_attack.avi”: attack of web server
- Video “webdoor_3_still_working.avi”: web server work not disrupted
- Video “webdoor_4_control.avi”: getting remote cmd and stealing password

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

- Continue development of WiShMaster:
  - Main objective: improve analysis of C code and remove the latest constraints on the code imposed by the parsing with regular expressions
  - Example: integrate "pycparser": C parser and an AST generator
- 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