WiShMaster

Windows **Sh**ellcode **Master**y

BlackHat Europe 2009

Benjamin CAILLAT

ESIEA - SI&S lab

```
>>> print """
        caillat[NOSPAM]esiea\x2efr
        bcaillat[NOSPAM]security-labs\x2eorg
    """.replace("[NOSPAM]", "@")
```

Contents

List of Figures

Abstract

Malicious codes often need to manipulate their own code in order to implement some viral techniques, like executable infections, memory-only execution or polymorphism.
Such manipulations are considerably simplified if the program is a shellcode. There are a few solutions to obtain a shellcode: one is to write source code in assembly, but it quickly becomes a boring work. Another is to write source code in C language in a specific way, so that compiled code doesn't contain any hardcoded address.
However, writing C code like this is very boring too, and it quickly becomes obvious that the use of an automatic tool that generates "specific" code from "normal" code is indispensable.

WiShMaster is a tool that converts a set of C source files written "normally" (the compilation of those source files produces an executable) and generates a shellcode, that is a block of code without any hard-coded or external reference and that can run in any process at any address. If the execution is redirected to its first byte, the shellcode will accomplish exactly the same operation as the executable generated through normal sources compilation.

This article describes the principle of this transformation - called later "shellcodisation" - and gives some examples to present the facilities that it enables.

1 The use of shellcodisation in virology

1.1 Context definition

Generally, malicious codes try to do several things: stay undetected by antivirus, propagate to other hosts or executables and execute specific actions (e.g. capture some private user data, open a backdoor on the system . . .); they may use many techniques to do this.
This first part chooses a few specific objectives. It then explains how they can be implemented and tries to show the difficulties that may be encountered during this implementation.

In this paper, we will only consider the case of malicious programs for Windows, even if some techniques presented may be adapted to work on other operating systems.

1.2 Selected objectives of the malicious code

To take some concrete examples, we will consider that we want to create a malicious code that is able to:
- stay undetected by antivirus working with signature identification;
- leave as few traces as possible on an infected host;
- integrate a legitimate program to create a Trojan;
- intercept the private data of a user using the infected computer;
- encrypt itself to prevent (or complicate) a manual analysis.

Let us choose some precise techniques to implement this. Note that there are of course many other solutions, but these are good representatives of techniques that may be used by real malware.

1.2.1 Staying undetected by antivirus working with signature identification

To stay undetected, the malicious code can implement polymorphism. As a reminder, a polymorphic malicious program is made up of two parts:
- the real malicious payload which is encrypted;
- a decryption part, placed at the beginning of the virus, that decrypts the malicious payload and transfers execution to it.

The key used to encrypt the malicious payload is changed at each infection so that two copies of the same virus have completely different payloads.
Figure 1 represents a malicious code that does not implement polymorphism. An identification signature could be established. Once this one has been added to antivirus databases, the malicious code can be detected.

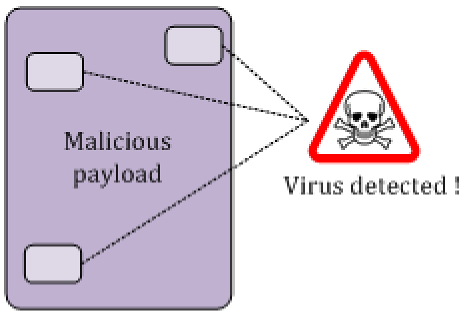


Figure 1: Detection of a malicious payload by signature

Figure 2 represents two copies of the same malicious code that implements polymorphism. Since the encryption key - stored in decryption part - is not the same, the encrypted payloads are completely different. It then becomes very difficult to establish an identification signature on the malicious payload.

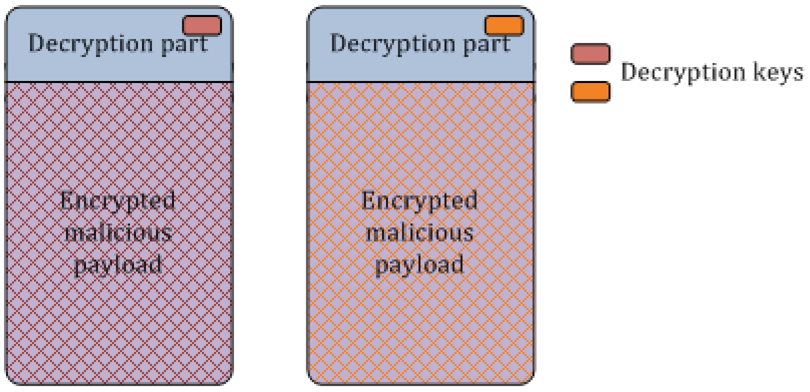


Figure 2: Two copies of the same virus that implements polymorphism

This prevents creating a signature identification on the malicious payload; however it might still be established on the decryption part. To avoid this, we can use metamorphism on this part. Note that this is only required in an automated propagation. If the malicious programme is specially developed for a targeted attack, we just have to rewrite the decryption loop manually.
Implementing metamorphism is hard work and is out of the scope of this article. We will only focus on the capability to implement polymorphism on the malicious payload.
Note that since the decryption key may be found in decryption part, this kind of polymorphism does not protect the malicious payload from a manual analysis. The encryption can rely on a simple XOR operation with a key of 32-bits.

1.2.2 Leave as few traces as possible on an infected host

To leave as few traces as possible on the infected system, we choose to add the capability to execute only in memory. Here is an example of a scenario:

- a code is running on the targeted system, for example after the exploitation of a buffer overflow;
- this code connects back to a server and downloads the malicious payload directly in the memory of the current process;
- the code transfers execution to the malicious payload which executes, without having being copied on the hard drive.

Figure 3 sums up this principle.

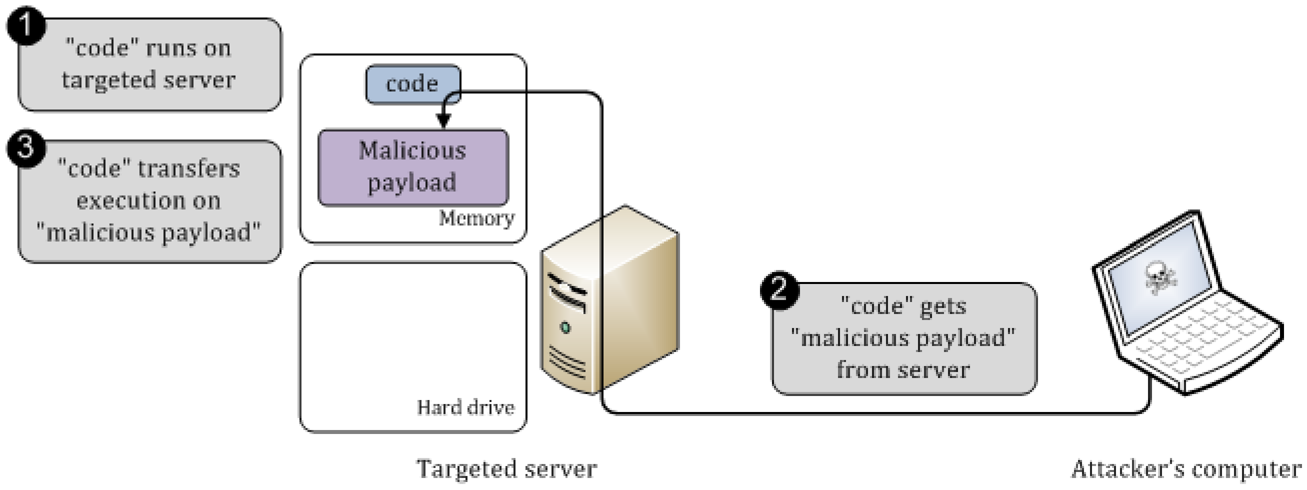

1
"code" runs on targeted server
3
"code" transfers execution on "malicious payload"
code
Malicious payload
Memory
Hard drive
2
"code" gets "malicious payload" from server
Targeted server
Attacker's computer

Figure 3: Execution of the malicious payload without writing it on hard drive

1.2.3 Integration into a legitimate program to create a Trojan

The aim is to integrate the malicious payload in a legitimate program so that it is executed when somebody uses the program. Of course, the behaviour of the program must not be disrupted so that the user is not aware of the infection.
This integration may be done in several ways. We consider the case where the malicious payload is added in the main executable. In order to limit the number of modifications on the PE header, the malicious payload will be added at the end of the executable, after the last section.
The redirection of execution flow may be done by patching the executable entry point in the PE header with the entry point of the malicious payload. A jump is then added in the malicious payload to jump on the original entry point after the execution.
However, this solution implies that the malicious payload is executed first. An antivirus with code emulation capability that scans the infected executable will then be able to emulate and analyse the malicious payload. A better solution is to patch some instructions that will probably be executed during the use of the infected program. For example, if the targeted program is a text editor, the function used to save the edited document. The malicious code will then be neither executed, nor analysed by an antivirus that scans the infected executable. Of course, this solution requires a manual analysis of the targeted executable to find suitable instructions and cannot be used in an automatic infection.
Figure 4 presents those different cases. The original executable is represented on the left. In the middle, we can see the executable after an infection where the executable entry point in PE header has been replaced by the entry point of the malicious payload. Finally, on the right side we can see the executable after infection where execution flow is redirected by patching an instruction.

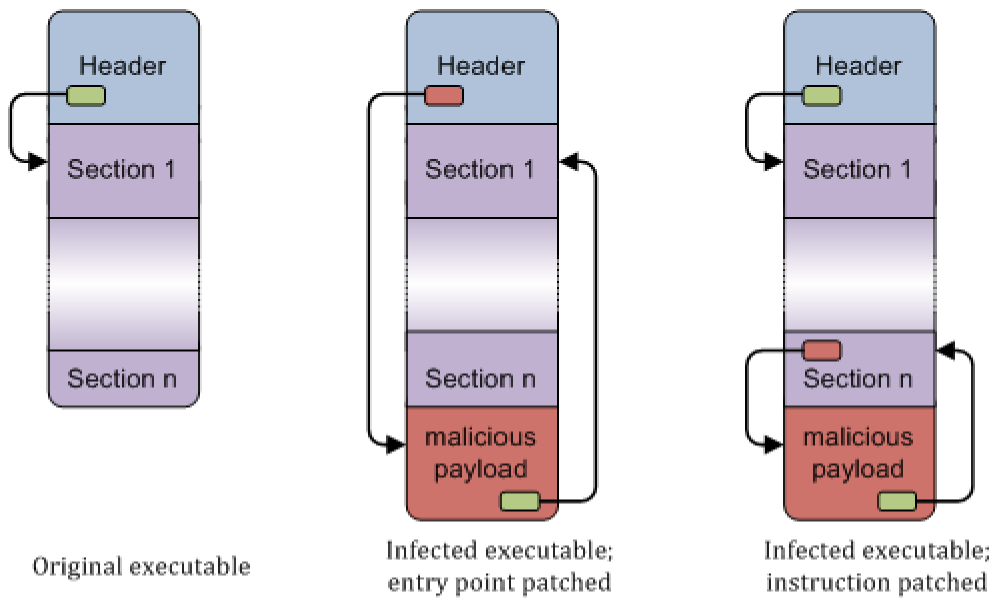


Figure 4: Infection of an executable

1.2.4 Interception of private data of a user using the infected computer

Processes are manipulating data that may be interesting and our malicious code can try to intercept them. However, from the operating system point of view, our malicious code is a simple process. Like other processes, it has its own memory space and its possibilities to interact with other processes are controlled by the security model of the operating system. If we want to intercept private data, we will have to exceed those limits.
This can be done in several ways:

– Working at kernel level. The security model of Windows is built in such way that if our malicious code manages to inject some code into the kernel, it will be able to analyse all input and output and to capture private data (keyboard stroke, network traffic, accesses to file system. . .). The disadvantage of this solution is that it requires administrative privileges.
– Targeting a specific application. For example if we want to capture the user credentials on a web site, we can develop a malicious plugin for the browser that will analyse all web traffic and extract the targeted data. However this technique is linked to a specific application.
– Using code injection and API hooking. The data to intercept is stored in buffers in the process memory which are passed in argument of function calls. For example, a server has a buffer where it receives data by calling the function “recv”and another to send data through the function “send”. In this solution, the malicious code uses standard function of Windows API to inject some code in the other process. The injected code installs some hooks on specific imported functions to intercept every call to those functions. It can then analyse the parameters before and after the call and look for interesting data. This solution is generic and does not require administrative privilege. However, note that if we consider that the malicious code is running in a limited session, it will of course not be able to inject some code into a process running in another session. Later, we will focus on this technique.

1.2.4.1 Code injection

The code injection may be done by several ways:

– The code to inject is built in a dll. The dll is then loaded in targeted process and “executed”. The dll injection mechanism may rely on several techniques (modifying the registry, using wide-system

Hooks, using CreateRemoteThread. . .) [1]

– The code is directly injected in remote process memory by using some standard functions of Win32 API:

  - OpenProcess: opens a handle with specific rights on a process designated by its PID;
  - VirtualAllocEx: allocates some memory in a process designated by a handle;
  - WriteProcessMemory: copies data from local memory to the memory of another process designated by a handle;
  - CreateRemoteThread: creates a thread in another process designated by a handle. It notably takes in parameter the execution address of the created thread;
  - CloseHandle: closes an opened handle.

Figure 5 sums up those operations

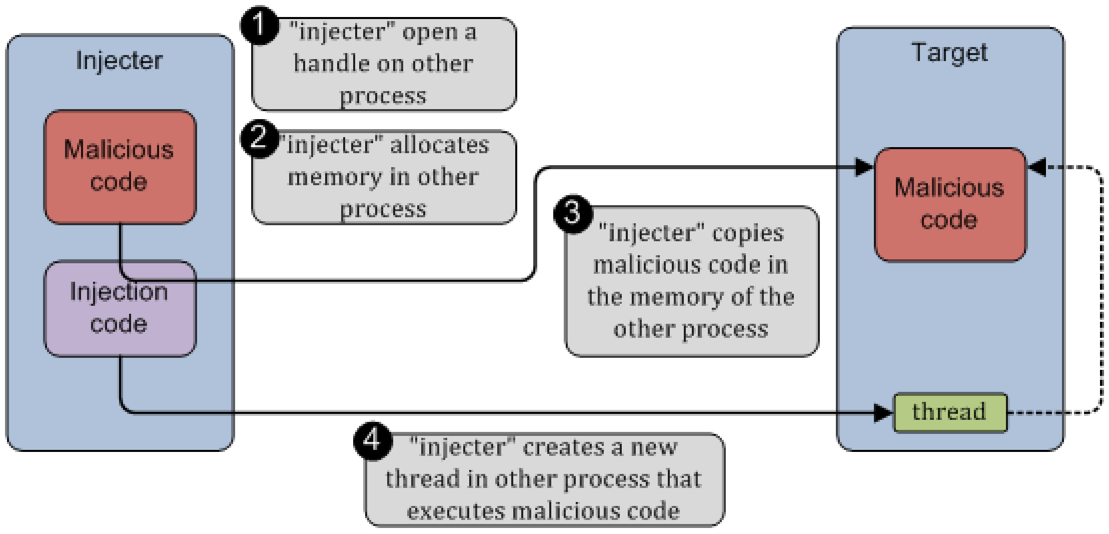

Injecter
Malicious code
Injection code
1 "injecter" open a handle on other process
2 "injecter" allocates memory in other process
3 "injecter" copies malicious code in the memory of the other process
4 "injecter" creates a new thread in other process that executes malicious code
Target
Malicious code
thread

Figure 5: Code injection with the technique VirtualAllocEx/WriteProcessMemory/CreateRemoteThread

This technique allows the execution of code without writing a dll on the hard drive. We finally choose this solution to implement code injection in our malware.
Note that this technique relies on specific services exposed by Windows kernel, which are particularly watched by protection programs like personal firewall. However, sometimes small tricks exist to bypass some of those programs. This aspect is out of the scope of this article and will not be considered later.

1.2.4.2 API hooking

The code injected in targeted process can then modify the memory to intercept function calls. This can be done by two ways:

– The code can patch the Import Address Table. The Import Address Table is a table that contains the address of all imported functions. It is filled when the process is created by Windows loader. Every call to imported functions (that is function in import table) is done through the addresses in this table. So by patching some of those values, we can intercept the wanted function calls.

– The first solution is quite easy to implement. The problem is that Windows offers another way to import function. Instead of using the function in the code directly - which creates a corresponding entry in the import table - we can declare a function pointer and resolve the function address during the execution by calling the functions LoadLibrary and GetProcAddress. Of course in this case, there is no entry for this function either in the import table or in the Import Address Table.
To solve this problem, instead of patching the Import Address Table, we can patch the header of

the function directly with a jump on our code. This solution creates many problems: memory rights of the section must be changed, instruction alignment must be computed to save the overwritten instructions, stack must be rebuilt. . . but it has the advantage to work all the time regardless of the resolution mechanism. The complete description of this solution is out of the scope of this article.

1.2.5 Encrypt itself to prevent manual analyse

The principle is similar to the one used for polymorphism, however the objective is different: polymorphism is used to protect against automated analysis executed for example by an antivirus. Here, we want to protect the malicious payload against a manual analysis. This creates a few differences:

– We have to use a real encryption algorithm. A simple XOR with a key of 32-bits is far from being enough. We can, for example, use AES with a key of 256-bits.
– The key should not be stored in the same file as the encrypted malicious code. For example, we can use another executable to decrypt the malicious code, or the key may be obtained from a server.

Figure 6 shows this scenario.

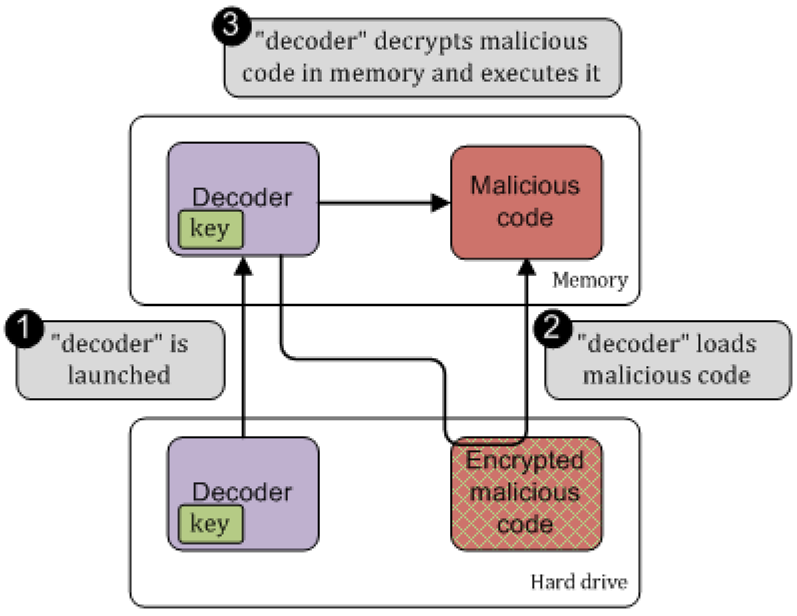


Figure 6: Execution of encrypted malware

Of course, it is easy to decrypt the malicious code if we extract the key from the decoder. However, we need to have both parts to decrypt the malicious code, which can be difficult if they are introduced by two different ways on the targeted system. If the decoder in captured alone, it does not contain any interesting information. If the malicious code is captured alone, it cannot be decrypted.

This principle may be extended by breaking the key into several pieces and storing them in several places. The algorithm of secret sharing of Shamir may be used for this [2]

1.3 Implementation from an executable

In this part, we consider that the malicious code is an executable “malware.exe”generated by the compilation of a set of C sources files written normally. This paragraph gives only a brief overview of the problems that may be encountered during the implementation the previously listed capabilities.

1.3.1 Implement polymorphism

The aim is to encrypt all binaries data in “malware.exe”that are characteristic of the malicious payload: every functions, initialized data and strings. The main problem is that those data are spread in whole executable. The decryption part will have to find and decrypt every blocks. As it may become complicated, it will be difficult to implement metamorphism on it. Furthermore the metadata of PE file cannot be encrypted, otherwise Windows will not be able to load the executable.
A best solution is to use a tool that will encrypt the whole executable, like a packer. The use of a public packer like the famous UPX [3] is not necessarily a good solution since generated executables have some very specific properties and may be recognized by an antivirus. We will have to develop our own packer that implements true polymorphism, which is a considerable job.

1.3.2 Execution only in memory

The piece of code running on remote server will be able to copy “malware.exe”in its address space. But the thing we have then in memory is a copy of the PE file, not a mapped executable ready to be executed. Before jumping on entry point of “malware.exe”, we will have to do all the initialization normally done by Windows loader:

– map the sections at the right address, since malware.exe probably contains hardcoded addresses. This can be done by using the function “VirtualAlloc”that allows to allocate memory at an address specified in parameter. Of course if this memory is already allocated, the function fails;
– resolved imported functions: load required libraries, find required functions and update the Import Address Table. Note that this step may be avoided by using dynamic address resolution.

This is not so complicated, but it requires some work. Furthermore, the size of the piece of code that gets “malware.exe”increases considerably so it may not be used as a shellcode in a buffer overflow exploitation.

1.3.3 Infecting other executable

The aim is to add the malicious payload extracted from “malware.exe”to a targeted executable, called “target.exe”in the following.
To execute the infection, we can simply try to add the sections of “malware.exe”after the last section of “target.exe”. The operation is however not as simple:

– several parts of the PE header of infected executable must be modified: the number of sections, the section table, size of image. . . ;
– since the code of “malware.exe”may contain hardcoded address, its preferred load address must be well chosen, so that the sections that represent malicious payload are loaded at the same address in a process “malware.exe”and in an infected “target.exe” ;
– the importation table in added code will not be parsed by Windows loader. So the required libraries will not be loaded and the Import Address Table will not be filled. As in last paragraph, a solution may be to use only dynamic address resolution during programming of “malware.exe”, but it quickly becomes boring.

1.3.4 Executing code injection

We will always encounter the same problems: “malware.exe”must be mapped at the right address and the imported functions must be resolved.

1.3.5 Implementing real encryption

As with polymorphism, the best solution is probably to use a specific packer that will get a secret key, for example, from a server and decrypt the malicious code. One again, it is a considerable job. I presented such a tool in a rump session at SSTIC in 2008. The slides (in French) may be found on SSTIC web site [4]. This tool is in beta version and will be available soon on my web site.

1.3.6 Summary

To sum up, it is of course always possible to add those capabilities to an executable, but it requires lots of work.
Those difficulties come from several properties of the executable:

- code and data are spread in the executable ;
- the process requires some of initialization - normally done by Windows loader - before execution can start: map sections in right place in memory, load required libraries, fill import address table. . . ;
- the code contains hardcoded address, so the section must be mapped at the right address. Otherwise, the relocation table must be used to patch every hardcoded address.

So all those capabilities could be implemented more easily if the code was constituted of only one block, if it was able to initialize the address space and if it contained no hardcoded address. That is if the code was a shellcode.

1.4 Implementation from a shellcode

Let us imagine now that our malicious code is a shellcode, that is a block of code completely autonomous. This shellcode executes exactly the same operations as the normal executable if we transfer execution to its first byte.
The implementation of previously listed capabilities becomes very easy.

1.4.1 Implement polymorphism

The decryption part becomes a loop that executes the decryption operation on the shellcode. If the encryption algorithm is a simple XOR with key of 32 bits, it becomes very simple:

Decryption loop of the polymorphic payload

```
UINT uiXorKey = 0xaabbccdd;
UINT i=0;
UINT j=0;

/* Decrypt shellcode */
for(i=0,j=0;i<sizeof(bShellcode)-1;i++,j=(j+1)&0x3)
{
        bShellcode[i] = bShellcode[i]^((CHAR *)&uiXorKey)[j];
}

/* Jump on shellcode */
((VOID (*) (VOID)) &bShellcode)();
```

1.4.2 Execution only in memory

By definition, the shellcode is able to execute in any process at any address. The external code running on the server just has to get the shellcode from a remote location to an allocated buffer and to jump on the beginning of this buffer. It does not have to do any initialization, since everything is handled by the shellcode itself. Furthermore, it does not need to map the shellcode at a specific address.

1.4.3 Infecting other executable

It becomes very easy to execute a simple infection. For example, the shellcode may be copied in the last section (which is increased by the corresponding size).

1.4.4 Executing code injection

Executing the shellcode in another process becomes very easy: we just have to allocate some memory in other process address space, copy the shellcode and create a new thread with the starting address set to the first byte of allocated memory.

1.4.5 Implementing real encryption

The implementation of real encryption relies on the same principle as polymorphism. The only difference is that the encryption will not be a simple XOR with a key of 32-bits, but a “true”encryption algorithm like AES.

1.4.6 Summary

We can see that the implementation of those capabilities is greatly simplified if the malicious code is a shellcode rather than an executable. The problem is now to find a way to generate a shellcode from a set of C source files.

2 Writing the shellcode

2.1 Analyse of binary data generated by compilation

To obtain malicious code as a shellcode, it is possible to write the source code directly in assembly. However, if the program is big, it may become quickly a long and boring job. This solution will be dismissed later and we will consider that source code must be written in C.
A first idea may be to write the program in C, to compile it, to extract binary data from the executable and to form the shellcode.
Let us take the example of a simple test program - called later "simpletest"- that only prints messages and displays the content of a file "test.txt".
Here are some extracts of the code:

File user.h

```
#define SIZE_USERNAME        32
#define SIZE_PASSWORD        32

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

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;
}
```

File main.cpp

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

BOOL DisplayData(VOID);

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

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);
}
```

This program contains:

- Two global variables;
  - A global variable “g_User”, declared and initialized in main.cpp. “User”is structure “USER”, a type defined in “user.h”
  - A global string “g_szMessage”
- Five internal functions:
  - “DisplayMessage”that displays “g_szMessage”
  - “DisplayFile”that tries to open a file “test.txt”and display its content
  - “DisplayData”which is the function that really executes all operation
  - “main”the program entry point that only calls “DisplayData”
  - “PrintMsg”which is used to display log messages.
- Several strings
- Several calls to imported functions: CreateFile, HeapAlloc, _snprintf...

This program is not very useful, but it regroups all the different things that we may have in a C program: global data, strings, internal function with or without parameters, imported functions...
We will now compile this program and have a look on the binary data produced:

```
Call of the function DisplayMessage

        DisplayMessage(g_szMessage);
00413119   68 00304300      PUSH 433000          ; ASCII "This is a message ..."
0041311E   E8 E8E7FFFF      CALL 0041190B        ; call jmp on DisplayMessage
00413123   83C4 04          ADD ESP,4
```

```
Call of the function DisplayFile

        if(DisplayFile("test.txt") == FALSE)
00413152   68 CCFF4200      PUSH 42FFCC          ; ASCII "test.txt"
00413157   E8 26E4FFFF      CALL 00411582        ; call DisplayFile
```

```
Call of the function CreateFile

        CreateFile(szFilePath, ...)
...
00412FD2   50               PUSH EAX
00412FD3   FF15 14724300    CALL DWORD PTR DS:[437214] ; call kernel32.CreateFileA
```

There are several points that prevent us from using this binary code directly to create a shellcode:

- There are a lot of hardcoded addresses: every reference to a string or a global data generates a hardcoded address.
- Internal functions calls are relative, but, the value included in instruction is the difference between the address of next instruction and the entry point of function to call. So we have to keep the internal functions at the same place from each other. This may be a problem, because if the compiler places some useless function or data between two functions that must be included in shellcode, we will have to keep this distance between functions. Furthermore, internal function calls does not directly jump on function entry point, but on a jmp instruction, placed at the beginning of the first section:

```
Call of the function DisplayMessage

0041190B   E9 50160000      JMP 00412F60        ; call DisplayMessage
```

  However, the compiler has perhaps an option to disable this behaviour.
- Imported function calls are done by jumping on the value contained in an entry of Import Address Table. Here again, we have a hardcoded address. Furthermore, since Windows uses an early binding mechanism, all required libraries have to have been loaded, all imported functions have to have been resolved and the Import Address Table have to have been filled.

To sum up, we notice that we are far from having a binary code that may be used in a shellcode.

2.2 Principle of the shellcode generation

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

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

2.2.1 First approach: patching assembly

A first solution may be to generate assembly with the compiler, to patch it with a transformation tool and to finally generate binary data.
However this approach suffers from several problems:

- firstly, the assembly generated by the compiler contains lots of hardcoded addresses, so we will have carry out lots of modification on it;
- secondly, to develop the transformation tool, we will have to work on assembly which is not really a natural language;
- finally the transformation tool will be linked to a specific assembly and then to a specific hardware platform.

I decided not to work at assembly level but directly on C code.

2.2.2 Second approach: using the stack

A second solution is to write C code in a specific way so that everything is handled in the stack.
For example, string may be rebuilt in stack:

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
...
```

To call an imported function, we use dynamic address resolution to get the address of desired function in a local function pointer:

Use of dynamic address resolution to call an imported function

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

However this technique creates several problems:

- writing code like this becomes very quickly boring;
- it does not solve the problem of calling internal functions;
- if we get some interesting code written normally for example from Internet, we will have to rewrite the major part of this code to adapt it.

2.2.3 Third approach: using a global data

A third solution is to work with one structure that stores all global data and that is transmitted in every internal function call. This structure, called later “GLOBAL_DATA”will contain:

- Pointers on internal functions
- Pointer on imported functions

- Global variables
- Strings

Once this structure has been defined, we modify the C code so that every reference to a previously listed element will be done through it.
Let us take the example of the function "DisplayFile":

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);
        ...
}
```

It becomes (modifications are colorized in red):

Patched function DisplayFile

```
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);
        ...
}
```

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;
```

We can reduce the number of modifications to do on source code considerably by using the power of C macros. We define the following 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
```

And the modified code 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);
        ...
}
```

As we can see there are now very few modifications. Let us analyse the generated assembly code:

Call of the function DisplayMessage

```
        DisplayMessage(g_szMessageUse);
00412F99   8B45 08          MOV EAX,DWORD PTR SS:[EBP+8]  ; get address of g_szMessageUse in
00412F9C   05 58010000      ADD EAX,158                   ; GLOBAL_DATA
00412FA1   50               PUSH EAX                      ; push address of g_szMessageUse
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

        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
```

Call of the 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
```

The generated binary does not contain any hardcoded address! We can extract this binary code and use it to form the shellcode.
The shellcode may be created simply by concatenating the extracted functions and adding the GLOBAL_DATA structure at the end. Figure 7 gives an overview of this structure.

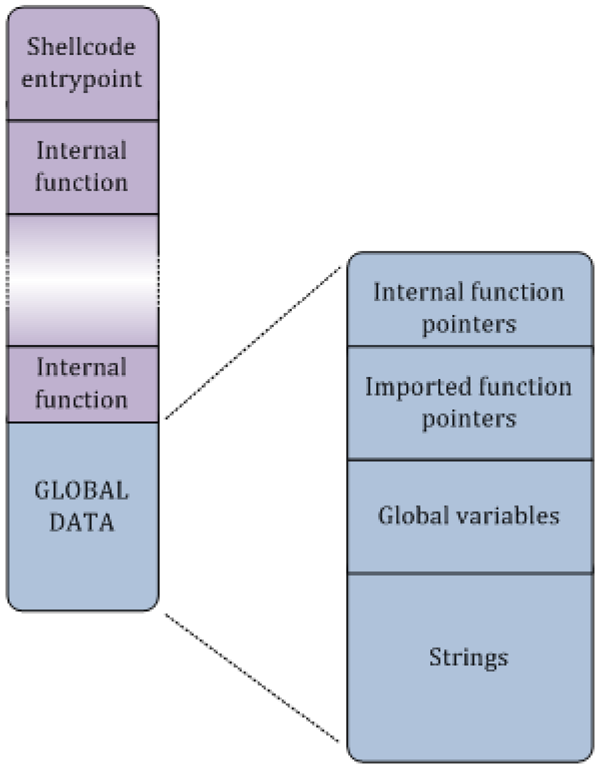


Figure 7: Overview of the structure of the shellcode

This is a first step; there are, however, still a few problems to solve:

– writing the definition of the GLOBAL_DATA structure is a long and boring task;
– we have to add some code to initialize this structure;
– the binary data must be extracted from the executable build by compilation and assembled to create the final shellcode.

I decided to write a tool that will execute all those operations automatically: WiShMaster.

3 WiShMaster: the shellcodisation process

3.1 Presentation

WiShMaster is a tool that automatically generates shellcodes, by using the previously described principle. It takes a set of C source files written "normally" in input (the compilation of this source files produce normally an executable) and generates a shellcode in output. If execution is transferred to its first byte, the shellcode will accomplish exactly the same operation as the normal executable. This transformation will be called later “shellcodisation”.

3.2 Development progress

3.2.1 WiShMaster version 1

A first version of WiShMaster has been developed and has been available on my web site for one year [5]. This is a graphical application developed in C#.
This version is quite stable but has several limitations:

– since it is a graphical application, WiShMaster is rather user-friendly. However it cannot be scripted or interfaced from script language;
– it parses source code with regular expressions. This one must, therefore, conform to a few syntax rules to be successfully analysed;
– the code should not contain more than one global variable;
– it is possible to extend the tool a little, but the additional modules must be written in C#, which is less well-known in the security community than script languages like Python or Ruby.

3.2.2 WiShMaster version 2

WiShMaster version 2 is under active development. This version corrects the previously enumerated problems:

– WiShMaster v2 is a console application written in Python. The shellcodisation process can be scripted;
– the user may intercede at any step of the shellcodisation process, view the results and correct the eventual mistakes;
– the parsing of source code with regular expressions has been considerably reduced. Most of the constrains on C syntax have then been removed;
– there is no longer any restriction on the number of global variables.

A beta version has been developed and will be soon available on my web site. This article only focuses on this version.

3.3 The shellcodisation process

3.3.1 Overview

The shellcodisation process accomplished by WiShMaster is divided into 6 steps:

– **Analysis**: the objective is to indentify internal functions, imported functions, global variables and strings.
– **Obtain the size of global variables**: this is an optional step where WiShMaster automatically obtains the size of eventual global variables (which must be included in GLOBAL_DATA).

– **Create environment**: WiShMaster creates the definition of GLOBAL_DATA structure and various macros. It then creates a copy of every source files in a temporary directory and does the small modifications presented previously on the code.
– **Generate**: WiShMaster builds patched sources and extracts binary data from produced executable. Finally it generates the shellcode. This shellcode may contain special values. For example, if the shellcode connects to a server, it will contain an IP address and a port that are filled for the moment with generic values.
– **Customize**: WiShMaster executes some transformations on the shellcode. For example, it may replace the generic values with values supplied by the user or encrypt the shellcode.
– **Integrate**: WiShMaster "integrates"the shellcode in the final project. It can be a simple copy of the shellcode in a specific directory or it can transform the shellcode in a C array and dump it in a C header file.

3.3.2 Description of step "analysis"

The objective of this step is to identify internal functions, imported functions, global variables and strings. To do this, a first approach is to parse all sources files with regular expressions and to look for those elements. This was the solution chosen in version 1 of WiShMaster. There are two problems with this solution: the list of source files must be selected manually by the user; above all the source code must conform to some syntax rules.
In version 2, WiShMaster relies on a special file that can be generated by the Microsoft compiler: the browse file.
The browse file contains all the symbols found by the compiler and the different places in source files where those symbols are defined and used (name of the file and line number).
The generation of the browse file is done in two steps:
– first the compiler must be called with the option "FR". It then generates a file ".sbr"for each source file parsed;
– the tool "bscmake"generates the final browse file (".bsc") from all files ".sbr".

The browse file can be parsed by using the "Microsoft Visual C++ Browser Toolkit", which can be downloaded from Microsoft web site [6].

By parsing this file, we will obtain:
– the list of files included in the project;
– the list of internal functions;
– the list of imported functions;
– the list of global variables.

This seems right, but in fact this analysis in not complete. Some information like the list of strings or the return type of internal functions is missing.
Therefore WiShMaster still has to parse code with regular expressions. If code is written in a special way, it is possible that WiShMaster would fail to find some information, even if the syntax is correct from the point of view of the C language.
The most restrictive convention is that every string must be placed in a macro "STR":

Use of the macro STR

```
PrintMsg(LOG_LEVEL_TRACE, STR("File successfully read: %s"), pData);
```

The definition of the macro STR is:

Definition of the macro STR

```
#define STR(x)     x
```

3.3.3 Description of step “Obtain the size of global variables”

WiShMaster has to know the size of global variables to reserve a corresponding space in GLOBAL_DATA structure. Thanks to the step “analyse”, we have the list of global variables and the location (file and line) where they are defined. We can try to compute the size of global variables by analysing this definition with a regular expression. However this is not a good solution because if the type of the variable is not standard, we will never manage to compute its size.
The chosen solution is to let the compiler do this compute: WiShMaster creates a copy of all sources file in a temporary directory and after the declaration of each global variable the definition adds of a global variable initialized with the size of the global variable.
For example, for the variable “g_User”:

Patched definition of the variable g_User

```
USER g_User = {"jmerchat","password"};
int GLOBAL_VAR_SIZE_g_User = sizeof(g_User);
```

WiShMaster compiles the patched sources and extracts the value of GLOBAL_VAR_SIZE variables from the generated executable. This extraction is done in the same way as the extraction of the shellcode in the “generate”step. It will be described in detail during the explanation of this step.

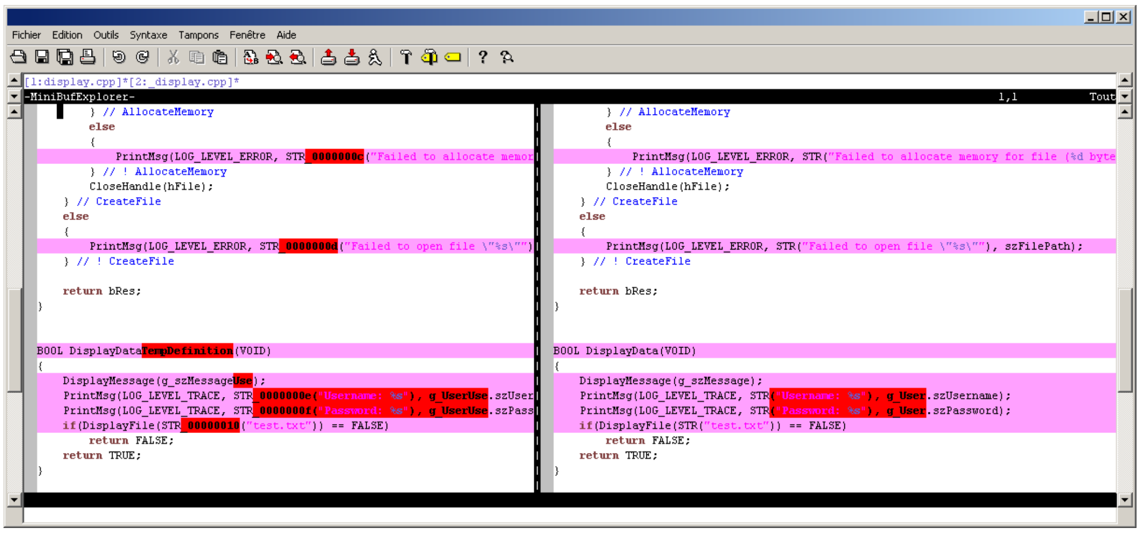
The execution of this step takes a few seconds since we have to copy and build all files. Furthermore, it is pointless to execute it if the size of global variables has been computed in a previous execution and if they have not been modified. WiShMaster integrates thus a mechanism of cache to keep those values. They may also be set manually by the user in a configuration file.

3.3.4 Description of step “generate”

During this step, WiShMaster generates a file named “global_data.h”that contains the definitions of the macros and of the structure “GLOBAL_DATA”. It then creates a patched copy of all files in a temporary directory.
The differences between those two copies are:

– the suffix “TempDefinition”is added at the end of every internal functions;
– the macro “STR”is replaced by a string STR_[NUM], where NUM is a number that uniquely identifies the string;
– the suffix “Use”is added to reference on global variables.

The following screenshot shows the differences for our program “simpletest”:


Fichier Edition Outils Syntaxe Tampons Fenêtre Aide
[1:display.cpp]*[2:_display.cpp]*
-MiniBufExplorer-
1,1
Tout

```
        } // AllocateMemory
        else
        {
            PrintMsg(LOG_LEVEL_ERROR, STR_0000000c("Failed to allocate memor
        } // ! AllocateMemory
        CloseHandle(hFile);
    } // CreateFile
    else
    {
        PrintMsg(LOG_LEVEL_ERROR, STR_0000000d("Failed to open file \"%s\"")
    } // ! CreateFile

    return bRes;
}


BOOL DisplayDataTempDefinition(VOID)
{
    DisplayMessage(g_szMessageUse);
    PrintMsg(LOG_LEVEL_TRACE, STR_0000000e("Username: %s"), g_UserUse.szUser
    PrintMsg(LOG_LEVEL_TRACE, STR_0000000f("Password: %s"), g_UserUse.szPass
    if(DisplayFile(STR_00000010("test.txt")) == FALSE)
        return FALSE;
    return TRUE;
}
```

```
        } // AllocateMemory
        else
        {
            PrintMsg(LOG_LEVEL_ERROR, STR("Failed to allocate memory for file (%d byte
        } // ! AllocateMemory
        CloseHandle(hFile);
    } // CreateFile
    else
    {
        PrintMsg(LOG_LEVEL_ERROR, STR("Failed to open file \"%s\""), szFilePath);
    } // ! CreateFile

    return bRes;
}


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

Figure 8: Difference between original and patched sources files

After having compiled the patched source, WiShMaster has to find and extract the internal functions and the global variables from the generated executable.
WiShMaster uses a special file that can be generated by Microsoft linker: the map file. This file is a text file that describes the memory mapping of a process created by launching the executable.
Here is an extract of the map file generated for our project "simpletest":

```
—————————————————————— Overview of a map file ——————————————————————

 Address          Publics by Value              Rva+Base       Lib:Object

0000:00000000       __except_list               00000000      <absolute>
...
0002:00001f60       ?DisplayMessage@@YAXPAD@Z   00412f60  f   display.obj
0002:00001f90       ?DisplayFile@@YAHPAD@Z      00412f90  f   display.obj
0002:000020d0       ?DisplayData@@YAHXZ         004130d0  f   display.obj
...
```

In this example, WiShMaster will consider that the function DisplayMessage will start at address 0x00412f60 and end at address 0x00412f90.
WiShMaster starts the executable with a special option (flag "CREATE_SUSPENDED"), which means that the process is created in memory but does not execute; It then uses the functions OpenProcess and ReadProcessMemory to read the process memory and to extract the required binary data.
Finally, the shellcode is created simply by concatenating the functions and adding the GLOBAL_DATA structure at the end.

3.3.5 Description of step "customize"

The customize step is compounded of a chain of functions that will execute some modification on the shellcode and transmit the modified shellcode to the next function.
The content of this chain is completely defined by the user through a configuration file. Each function is stored in a Python module. WiShMaster comes with some standard modules, but users can write their own modules.
To understand the use of this step, we will take a small example. Let us imagine that we want to write a shellcode that connects to a server. In the source file we will have two variables that represent the IP

address and the port of the server. If we put real value directly in those variables, the generated shellcode will contain those specific values. The shellcode will thus be linked with this server, which raises two problems:

– if we want to connect to another server, we have to change the values in the source and to regenerate the shellcode, which takes some time;
– we cannot distribute the shellcode in its binary form.

The solution of these problems is to initialize the IP address and the port with some special values, which generally never appear in binary code, for example 0xaaaaaaaa for the IP address and 0xbbbbbbbb for the port. The generated shellcode will thus contain these special values.
We then write a “customize”module in Python that reads the IP and port of the server from a configuration file, and replaces the special values 0xaaaaaaaa and 0xbbbbbbbb by them. We can now distribute our shellcode in binary form by giving the generated shellcode (with the special values) and the “customize”module.
If another user wants to use this shellcode, he just has to open the generated shellcode and to apply the customize module.
Of course “customize”modules can carry out other operations besides just patching a few values. For example, they can encrypt a shellcode. WiShMaster comes with two “customize”modules that can encrypt a shellcode: one executes a XOR encryption with a 32-bits key, and the other an AES-CBC encryption with a 256-bits key.
Figure 9 presents the principle of the creation of an encrypted shellcode. The upper part represents the developer and the bottom the user who customizes and encrypts the shellcode.

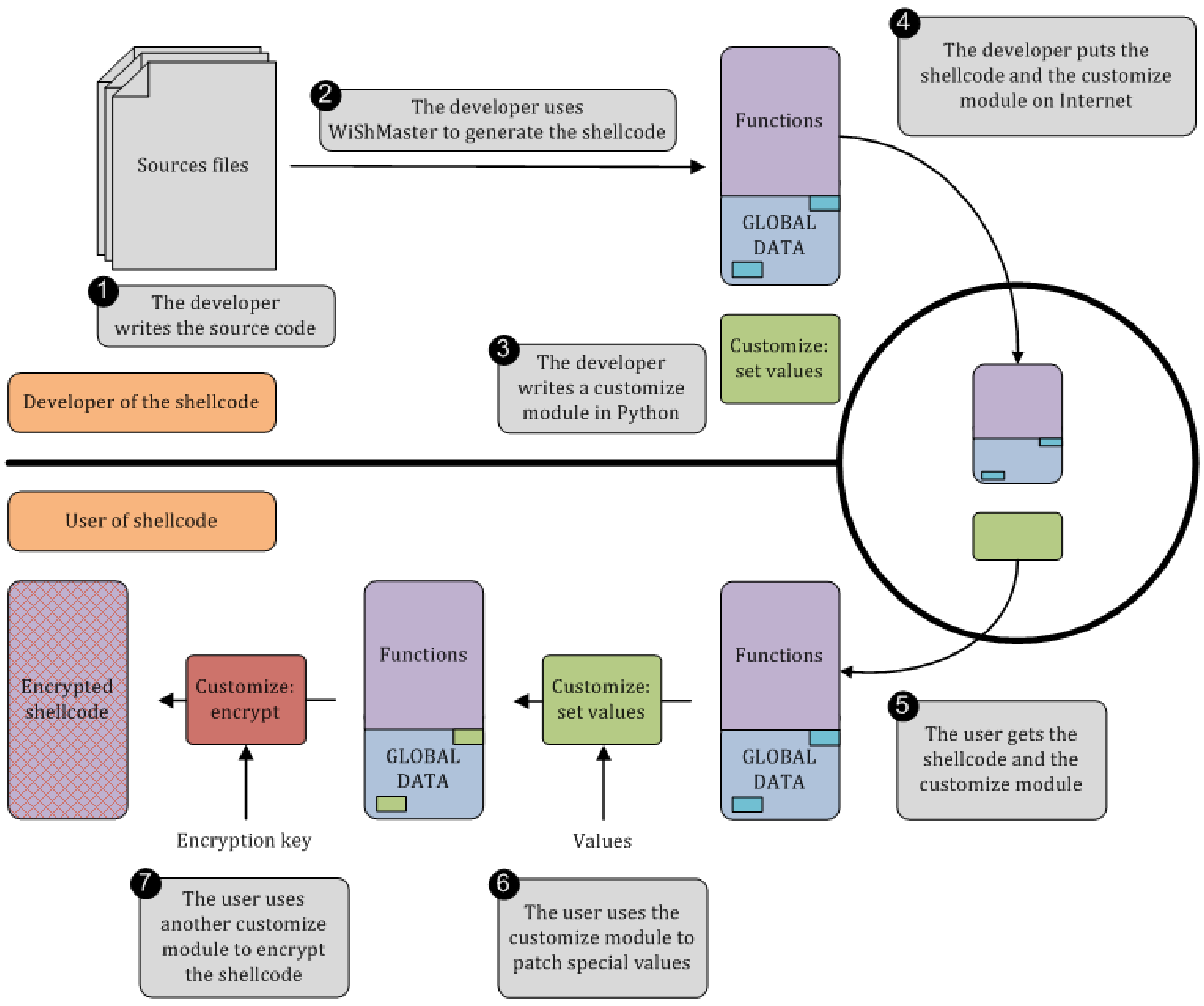

Sources files
2 The developer uses WiShMaster to generate the shellcode
Functions
GLOBAL DATA
4 The developer puts the shellcode and the customize module on Internet
1 The developer writes the source code
3 The developer writes a customize module in Python
Customize: set values
Developer of the shellcode
User of shellcode
Encrypted shellcode
Customize: encrypt
Functions
GLOBAL DATA
Customize: set values
Functions
GLOBAL DATA
Encryption key
Values
5 The user gets the shellcode and the customize module
7 The user uses another customize module to encrypt the shellcode
6 The user uses the customize module to patch special values

Figure 9: Principle of the separation between the developer and the user of a shellcode

3.3.6 Description of step "integrate"

This step integrates the customized shellcode in final project. This integration may be done either by copying the shellcode in a specified directory, either by transforming it in a C array and dumping it in a C header file:

```
Overview of a dumped shellcode

/*
        Shellcode generated by WiShMaster.
        Size=1022 bytes
*/

UCHAR bShellcode[]=
"\x55\x8B\xEC\x83\xEC\x40\x53\x56\x57\x8B\x45\x0C\x03\x45\x10\x5F"
...
"\x20\x2B\x20\x62\x20\x3D\x20\x25\x64\x00\x4C\x01\x00\x00";
```

This file can then be used in another project.

3.3.7 Implementation of shellcodisation in WiShMaster

The version 2 of WiShMaster has been developed with the following objectives:

– allow the user to execute the shellcodisation process step by step or in one time;
– allow the user to view the result of each step and to change them if he detects a mistake;
– allow the user to write their own shellcodisation script that will use the various functions exposed by WiShMaster.

WiShMaster can be launched in 3 modes: automatic, interactive and script

3.3.7.1 Automatic mode

In this mode, WiShMaster automatically executes the shellcodisation process:

WiShMaster in automatic mode

```
$ ./wishmaster.py -s ../Projects/CODE/simpletest.xml -a
[INFO    ] Configuration of module "simpletest" successfully loaded
...
[INFO    ] Final shellcode is in file "...\simpletest\shellcodes\simpletest.bin"
[INFO    ] Binary "...\simpletest\shellcodes\simpletest.bin" successfully
           copied to "...\integration\simpletest.bin"
```

3.3.7.2 Interactive mode

In this mode, WiShMaster starts a Python interpreter and lets the user drive the shellcodisation process. The shellcodisation is divided into 6 steps, each sub-divided into several sub-steps. Each sub-step is represented by a function that may be called from the interpreter. For example, the first step is "analyse"and the first sub-step is to generate the browse file, which is handled by the function "GenerateBrowse":

WiShMaster in interactive mode: execution of GenerateBrowse

```
$ ./wishmaster.py -s ../Projects/CODE/simpletest.xml -i
[INFO    ] Configuration of module "simpletest" successfully loaded
                    === WiShMaster interactive shell ===
(simpletest:manual mode)
>>> GenerateBrowse()
[INFO    ] Browse file successfully generated
True
```

The next sub-step is to analyse the generated browse file. This is handled by the function "Analyse-Browse":

WiShMaster in interactive mode: execution of AnalyseBrowse

```
(simpletest:manual mode)
>>> AnalyseBrowse()
[INFO    ] Browse file successfully analysed
True
```

Every result is kept in objects that may be displayed simply by calling "print"function on them. For example, if we display after the execution of those two sub-steps the object that represents the current project:

WiShMaster in interactive mode: display of the results

```
(simpletest:manual mode)
>>> print Solution.dicProjects['simpletest']

================================================================================

Project name     : simpletest
Input type       : code
Output type      : shellcode
Inlined projects : initsh,log
Fast mode        : True
--------------------------------------------------------------------------------
List of files

 - [H] ...\simpletest\headers\stdafx.h
 - [H] ...\simpletest\headers\global_data.h
 - [S] ...\simpletest\sources\global_data.cpp
 - [S] ...\simpletest\sources\main.cpp
 ...

--------------------------------------------------------------------------------
List of internal functions

DisplayFile [decorated name="DisplayFile(char *)" parameters="char *" return="BOOL"]
DisplayMessage [decorated name="DisplayMessage(char *)" parameters="char *" return="VOID"]
...

--------------------------------------------------------------------------------
List of imported functions
 (real name=fflush" return="", call="", parameters="")
 (real name=CreateFileA" return="", call="", parameters="")
 (real name=HeapAlloc" return="", call="", parameters="")
...

--------------------------------------------------------------------------------
List of global variables

g_szMessage type: CHAR; size: 0 bytes; is an array: True
g_User type: USER; size: 0 bytes; is an array: False
```

We can then check the results and alter them if there is a mistake before to continuing.

This mode offers lots of flexibility. However, the user has to remember all the function names that must be called, which is not really user-friendly. WiShMaster can be set in another mode where it automatically builds the execution flow. The user can then execute the next sub-step by calling the function “stepi()”. The function “step()”orders WiShMaster to execute all the sub-step in the corresponding step.
The keywords come from “gdb”interpreter, so WiShMaster too recognizes the functions:

– “restart()”: reinitialises the shellcodisation process;
– “cont()”: executes the shellcodisation process from the current step to the end;
– “run()”: executes the shellcodisation from the beginning.

The execution flow is represented by an object “ExecutionFlow”that may be displayed with the “print”function. Here is a small example:

WiShMaster in interactive mode: display of the execution flow

```
(simpletest:analyse/generate browse)
>>> print ExecutionFlow
  simpletest:analyse
>   generate browse
    analyse browse
    analyse code
    generate exported functions database
    load exported funtions database
    update modules' imports
  simpletest:get size of global variables
    fill patch
    apply patch
    build environment
    extract size of global variables
  simpletest:create environment
    create sorted lists
    create global header
    fill patch
    apply patch
  simpletest:generate
    Build created environment
    extract shellcode
    dump shellcode
    customize shellcode
  simpletest:integrate
    copy binary


(simpletest:analyse/generate browse)
>>> stepi()
[INFO    ] Browse file successfully generated
True

(simpletest:analyse/analyse browse)
>>> stepi()
[INFO    ] Browse file successfully analysed
True

(simpletest:analyse/analyse code)
>>> print ExecutionFlow
  simpletest:analyse
    generate browse
    analyse browse
>   analyse code
    generate exported functions database
    load exported funtions database
    update modules' imports
  simpletest:get size of global variables
    fill patch
    apply patch
    build environment
    extract size of global variables
  simpletest:create environment
    create sorted lists
    create global header
    fill patch
    apply patch
  simpletest:generate
    Build created environment
    extract shellcode
    dump shellcode
    customize shellcode
  simpletest:integrate
    copy binary

(simpletest:analyse/analyse code)
>>> step()
[INFO    ] Code successfully analysed
[INFO    ] Exported functions database successfully generated
[INFO    ] Exported functions database successfully loaded
[INFO    ] Imports successfully updated
True

(simpletest:get size of global variables/fill patch)
>>>
```

Note the small sign “>”that shows the next sub-step to execute.

3.3.7.3 Scripted mode

In this mode, WiShMaster executes a Python script supplied by the user. Here is an example of script:

Example of WiShMaster script

```
#!/usr/bin/python
# -*- coding: ISO-8859-15 -*-

# Standard importations
import sys

# Script entry point
print "Starting shellcodisation"
stepi()
stepi()
print Solution.dicProjects['simpletest']
```

The execution leads to the following output:

WiShMaster in script mode

```
$ ./wishmaster.py -s ../Projects/CODE/simpletest.xml -e simpletest
[INFO    ] Configuration of module "simpletest" successfully loaded
Starting shellcodisation
[INFO    ] Browse file successfully generated
[INFO    ] Browse file successfully analysed
...
```

3.3.7.4 Hook functions

WiShMaster allows the user to set “hook functions”on sub-step. A hook function is a Python script that will be called after the execution of the corresponding sub-step. Hook functions can typically be used to automate a patch operation in the shellcodisation process if WiShMaster makes a mistake. They can too be used to set the size of global variables, so that WiShMaster does not have to handle this operation.
For example, here is a script that inserts a hook after browse file analysis. The hook function manually sets the size of global variables, so that WiShMaster will not have to compute them during the step “Obtain the size of global variables”.

Example of script to set a hook function

```
#!/usr/bin/python
# -*- coding: ISO-8859-15 -*-

# Standard importations

# Personal importations
from output import Output
from shell import EnumSteps

def HookFunction_AnalyseBrowse(Project):
        Project.dicGlobalVariables['g_szMessage'].SetInformationManually(46, True)
        Project.dicGlobalVariables['g_User'].SetInformationManually(64, False)

        Output.debug('Hook point of step "analyse browse" successfully executed')
        return True

lstHookFunctions = {
                EnumSteps.ID_STEPI_ANALYSE_ANALYSE_BROWSE:HookFunction_AnalyseBrowse,
}
```

3.3.8 Initialising the shellcode

The shellcodisation process described previously creates a binary code that may run at any address. However, to work, the shellcode must initialise the GLOBAL_DATA structure, which means filling the pointers on internal functions and imported functions.
A few functions are included in the shellcode to do this initialisation. This paragraph gives a brief overview of how they work.

3.3.8.1 Finding address of GLOBAL_DATA

Firstly, WiShMaster finds the shellcode load address with a simple “call”followed by a “pop”:

Code used by the shellcode to find its load address

```
UCHAR * pShellcode = NULL;

/* 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));
if(i == 512)
        return FALSE;
pShellcode -= i;
```

Once the load address has been found, the shellcode just has to add the total size of shellcode minus the size of GLOBAL_DATA to get the address of the structure GLOBAL_DATA.

3.3.8.2 Finding the addresses of internal functions

During the shellcodisation process, WiShMaster integrates the size of each internal function in GLOBAL_DATA (in fact the size of binary data extracted from the generated executable). The addresses of internal functions can then be calculated very quickly 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;
}
```

3.3.8.3 Finding the addresses of imported functions

Finding the address of an imported function means to load the shared library that exposes this function and to find the function address in this library. This can be done very quickly with the functions of Win32 API LoadLibrary and GetProcAddress. But this does not solve the problem of finding the addresses of those two functions...
We know those functions are exported by the library “kernel32.dll”, but the load address of this library depends on the version of Windows.
WiShMaster uses tips well-known by Windows shellcode writers to solve this problem:

– firstly it get the address of kernel32.dll by analysing memory;
– secondly it walks through kernel32.dll exportation table to find the addresses of the functions.

Finding the address of kernel32.dll

This operation is handled by a function called “GetKernel32Address”. This function executes the following steps:

– It finds the address of the Process Environment Block (PEB). The PEB is a structure allocated in user-land address space of each process that holds a lot of information on it. A description of the structure of the PEB may be found on Microsoft web site [7]. The address of the PEB is stored at fs:[0x30].
– It extracts the address of a structure “PEB_LDR_DATA”that can be found at the offset 0xC in the PEB. This structure holds in particular three pointers on three linked lists of objects “LDR_DATA_TABLE_ENTRY”. Each object represents a loaded module and contains an UNICODE_STRING structure representing the library name and the module load address. The three linked lists are sorted in different order.
– It walks through the third list that is sorted by load order. Normally, the first object always represents “ntdll.dll”and the second “kernel32.dll”. However, to be sure, the function compares the first two letters of the module name with “ke”.
– When it is sure to have found the right structure, it just has to extract the load address.

Finding the addresses of LoadLibrary and GetProcAddress

The addresses of these functions are found by parsing the exportation table of “kernel32.dll”. The description of this operation is not useful (you only need to know the PE format) and will not be detailed in this document.

Finding the addresses of imported functions

To find the address of an imported function, we can use the “GetProcAddress”function. However, this implies storing the names of all imported functions in GLOBAL_DATA structure. Now, if we just have a look at the name of Windows functions, we will notice that they are rather big (“CreateRemoteThread”, “WriteProcessMemory”. . .).
Another solution is to write a function - called later “GetProcAddressByCksumInDll”- that will be able to find an exported function from the “checksum”of its name by parsing the PE format of a loaded library. The “checksum”refers here to the result of a transformation function that converts a string of any length into a 32-bits value. This technique has two advantages: firstly we have to store only a 32-bits value for each imported function in GLOBAL_DATA, instead of a long function name. Secondly this makes it easier to write a loop that walks through all the checksums as they are the same size.
The checksum transformation must be well chosen to avoid collision as much as possible (that is two functions with two different names that have the same hash). WiShMaster uses the same algorithm that Metasploit [8], which gives really good results.
Here is the code in Python:

Function used to compute the hash of a function name

```
def ComputeCksum(szString):
        """ Compute a 32-bits checksum for a string
Arguments
  szString = the string on which the checksum must be computed
Return
  int = checksum
        """
        uiCksum = 0
        for c in szString:
                uiCksum = ((uiCksum>>0xd)|(uiCksum<<(0x20-0xd))) & 0xffffffff
                uiCksum = (uiCksum + ord(c)) & 0xffffffff
        return uiCksum
```

Note that “GetProcAddressByCksumInDll”is not really a new function, since we already need such a function to find the addresses of “LoadLibrary”and “GetProcAddress”. In fact the same function is used to find the addresses of “LoadLibrary”and “GetProcAddress”and all the other imported functions.

The function “GetProcAddress”seems to be pointless, but in fact we may need it in special cases: sometimes, a function exported by a library is a simple forwarder to another function in another library. It this case, the export directory does not contain the address of the function, but a pointer to string “[name of dll].[name of function]”where [name of dll] is the name of the dll that exports the function [name of function].
For example, if we use the tool pedump [9] to display the function exported by “kernel32.dll”, we notice that “HeapAlloc”is a forwarder to “RtlHeapAlloc”in “ntdll.dll”(on my system):

Extract of the analysis the library kernel32.dll by pedump

```
C:\> pedump C:\WINDOWS\system32\kernel32.dll
  ...
  000090DA   518  HeapAlloc (forwarder -> NTDLL.RtlAllocateHeap)
  00036136   519  HeapCompact
  00012C46   520  HeapCreate
  0005F7D1   521  HeapCreateTagsW
  00010F88   522  HeapDestroy
  0005F7A0   523  HeapExtend
  000090F0   524  HeapFree (forwarder -> NTDLL.RtlFreeHeap)
  ...
```

The function used to resolve imported functions in WiShMaster is able to handle forwarders automatically: if it detects that an exported function is a forwarder, it simply uses LoadLibrary and GetProcAddress to find the real function.

3.3.8.4 *Summary of the shellcode initialisation*

The shellcode initialisation relies on three functions:

– “GetKernel32Address”: returns the load address of “kernel32.dll”;
– “GetProcAddressByCksumInDll”: find an exported function from the checksum of its name;
– “InitialiseShellcode”: entry point of the shellcode, which initialise the GLOBAL_DATA structure.

Prototypes of the functions used to initialise shellcode

```
UINT GetKernel32Address(VOID);
LPVOID GetProcAddressByCksumInDll(UINT uiChecksum, HMODULE hLib, \\
        LPVOID pLoadLibrary, LPVOID pGetProcAddress);
BOOL _InitializeShellcode(VOID);
```

4 Developing applications with WiShMaster

4.1 Objectives of WiShMaster

The version 1 of WiShMaster was only able to create monolithic shellcodes. With the version 2, the objectives of WiShMaster have been considerably broadened to:

- allow the development of modular applications, that is applications that can dynamically extend their capabilities by loading a module that implements some functions;
- allow the user to choose the format of the generated binary: an executable, a dll or a shellcode;
- allow code reusability, that is reusing some source code within several projects without having to make several copies of source code;
- allow the development of projects in the very powerful IDE Visual Studio;
- allow the distribution of projects either in source or in binary format.

4.2 Structure of an application in the version 2 of WiShMaster

4.2.1 Overview of the application structure

An application is compounded of one or several “modules”. Each module implements a set of functions that may be called by other modules. So each module contains an “export”table used to export internal functions and an “import”table used to resolve functions exported by other modules (just like in the PE format).
A module can be in one of the following 4 forms:

- an executable;
- a dll;
- a shellcode;
- inlined in another module.

When a module is inlined in another module the import and export table of both modules are merged.
Figure 10 sums up this principle. The application is compounded of three modules. Each exports some functions (in blue on the drawing) and imports some functions (in purple on the drawing).
Module 1 and module 2 are merged to create a single shellcode. Module 3 creates an executable. The imported symbols of the shellcode and the executable are resolved at execution time.

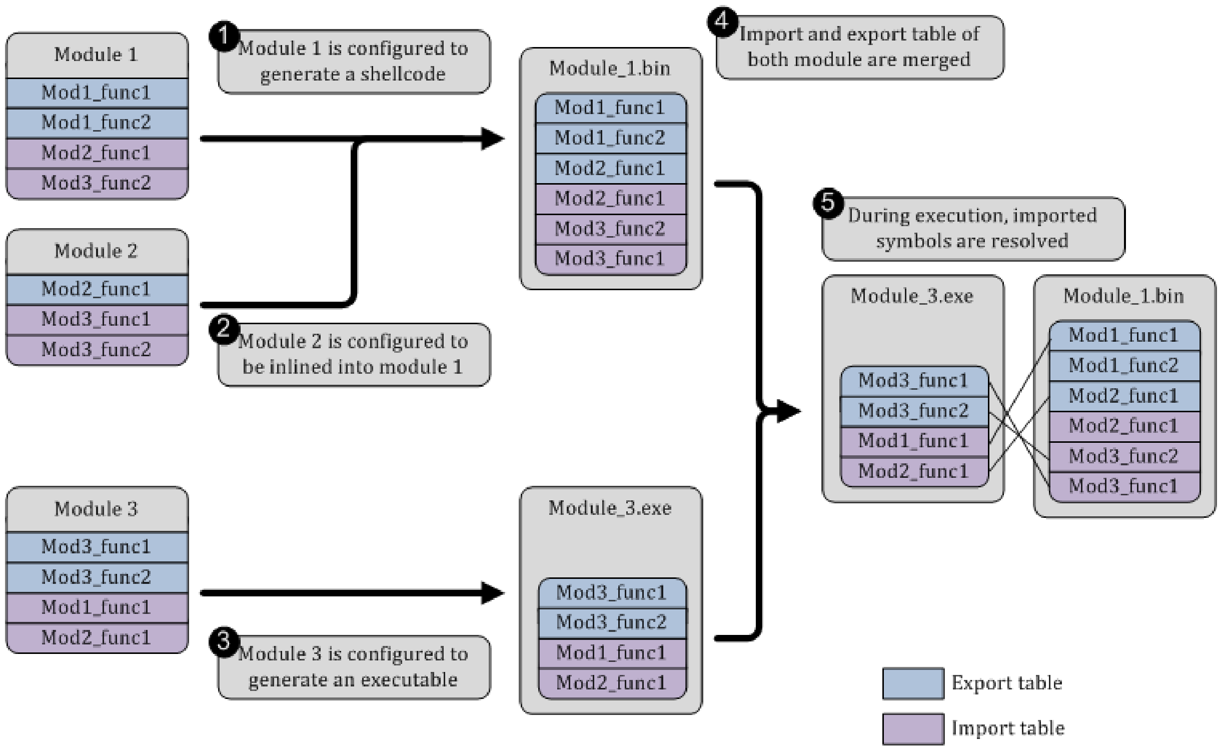


Figure 10: Structure of an application developed with WiShMaster 2

4.2.2 Export and import tables

Modules created as an executable or a dll could use the exportation table of PE format to export functions. However we cannot use this mechanism with a shellcode since it does not respect the PE format. Furthermore I wanted a module to be able to load even if some other required modules are missing, so that it can wait for those modules (or try to get them from Internet). Now, if you start an executable that imports an inexistent dll, you get a message box with an error.
Finally, I decided to create my own format. The export table is integrated in the structure GLOBAL_DATA. The structure contains the checksum of the name of each exported function.
The importation is done exactly in the same way as for normal dlls. From the point of view of the GLOBAL_DATA structure, there is no difference between a function imported from a standard dll and one imported from a module (which can be an executable, a dll or a shellcode).
Figure 11 gives an overview of the structure of import and export tables in GLOBAL_DATA.

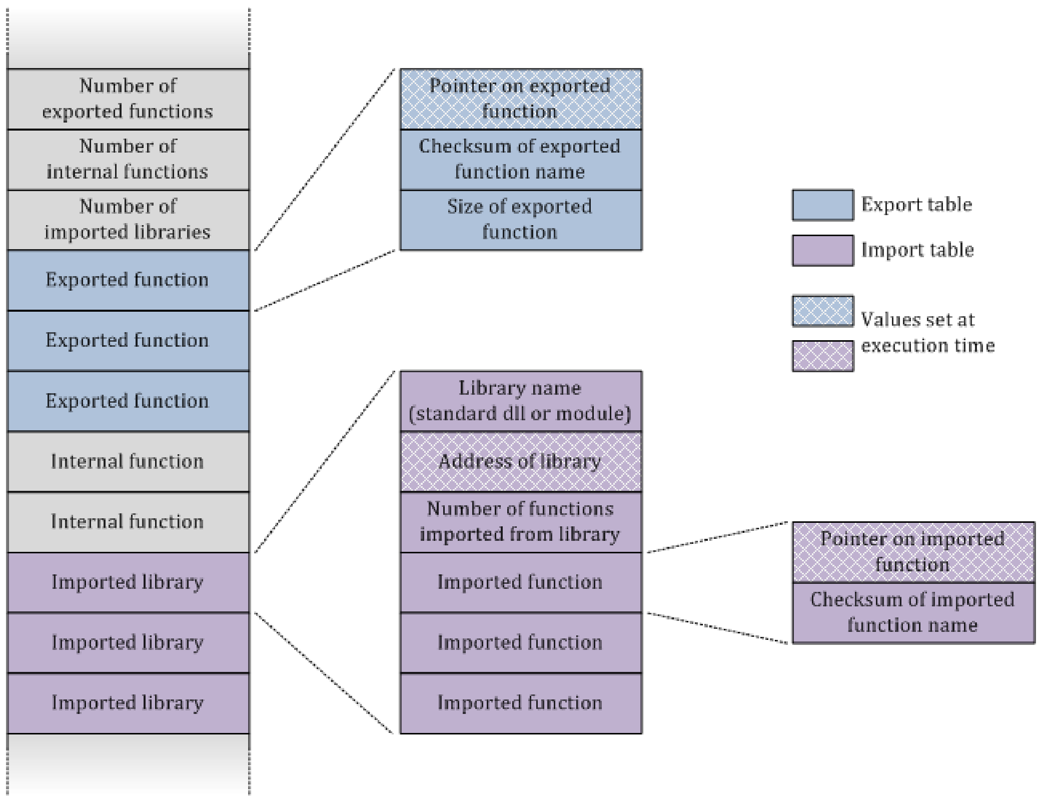

Number of exported functions
Number of internal functions
Number of imported libraries
Exported function
Exported function
Exported function
Internal function
Internal function
Imported library
Imported library
Imported library
Pointer on exported function
Checksum of exported function name
Size of exported function
Export table
Import table
Values set at execution time
Library name (standard dll or module)
Address of library
Number of functions imported from library
Imported function
Imported function
Imported function
Pointer on imported function
Checksum of imported function name

Figure 11: Overview of the structure of import and export tables in GLOBAL_DATA

4.2.3 Entry point

Each module may have an entry point, that is an internal function that must be called when the initialisation has been done. The GLOBAL_DATA holds a pointer on this function.

4.3 Standard modules

There are a few standard modules that expose some functions frequently used by other modules. This paragraph gives a brief description of those modules

4.3.1 Module "log"

This module exposes the function "PrintMsg"which allows the print of formatted messages. The message can be sent either to standard output or to attached debugger output. The tool DebugView [10] may be used to display the output sent to the debugger (if no debugger is attached to the process). This is particularly useful to debug the shellcode running in a graphical application.

4.3.2 Module "initsh"

This module exposes all the functions needed to initialise a shellcode:

– the function "InitialiseShellcode", the entry point of a shellcode that initialises GLOBAL_DATA structure;
– the function "GetProcAddressByCksumInDll"that finds the address of a function exported by a dll (as

describe previously);
- a function “ResolveImportedFunctions”that takes a GLOBAL_DATA structure in argument and resolves all imported functions;
- a function “GetProcAddressByCksumInModule”that finds the address of a function exported by a module.

4.3.3 Module “loader”

This module is used to manage a set of modules. It exposes a single function “AddModuleToLoad”that allows the loading of a module in memory.

```
Prototype of the function AddModuleToLoad
BOOL AddModuleToLoad(CHAR * szFilePathModule, MODULE_TYPE moduleType);
```

- “szFilePathModule”represents the full path of the module to load
- “moduleType”represents the type of module: executable, dll or shellcode

“loader”handles all the load and the initialisation of a module:
- the module is loaded in memory;
- it decrypts the module if this one is an encrypted shellcode;
- it resolve all imported symbols (from standard library or other module);
- it check if there is some module previously loaded that was waiting for function exported by this new module;
- it call the entry point of all initialised and resolved modules.

Of course, “loader”relies on the module “initsh”. In fact, “initsh”is inlined in “loader”

4.4 Shellcode encryption

WiShMaster integrates two customization modules that can encrypt a shellcode. The first executes a XOR encryption with 32-bits key and the other an AES-CBC encryption with a 256-bits key.

4.4.1 The XOR encryption with 32-bits key

This encryption is used only for polymorphism. It is generally used to create a single executable that contains the decryption loop - with the encryption key - and the encrypted shellcode.

4.4.2 The AES-CBC encryption with a 256-bits key

4.4.2.1 Principle of working

The module executes an AES-CBC with a key of 256 bits. It relies on the free cryptographic library PolarSSL [11] (formerly XySSL) and is used to protect the shellcode from a manual analysis. The decryption part (and the encryption key) and the shellcode are in two different files. The module “loader”implements all the mechanisms to decrypt an encrypted shellcode.
The attack is then compounded of two steps:
- the targeted computer is infected with a customized “loader”

– the real malicious payload - an encrypted shellcode - is transferred on the targeted computer

Those two elements are introduced on targeted system in different ways at different time. The main point is that one element alone does not give any information on the attack:

– “loader”is a generic framework and does not contain any code that represents the real attack;
– the malicious payload cannot be decrypted without the key in “loader”.

The two following paragraphs give two examples of attacks that relies on this principle

4.4.2.2 Example 1: getting encrypted shellcode from the cache of the browser

An e-mail with a web link is sent to the targeted user. The user clicks on the link and visits a fake web site that installs a customized version of “loader”. If this attack fails, the analysis of “loader”will only show that it executes some recursive searches of jpg files in some directories. Apparently, nothing really dangerous.
Later (maybe a few days), the user receives another mail with another link and clicks on it. He then visits an apparently innocent web site.
In fact, the real malicious payload has been transformed into an encrypted shellcode and has been hidden in a jpg file. This image has been put in the middle on the second web site. When the browser of the user displays the image, it stores the file in its cache directory.
“loader”scans this directory regularly and finds the image. It decrypts the module in memory and jumps on its entry point.
This attack has been successfully executed in a lab. The malicious payload was a backdoor that was able to bypass proxy with “basic”authentication. A set of videos presenting the whole attack are available on my web site [12].

4.4.2.3 Example 2: getting encrypted shellcode from an USB key

The objective is to manage to copy some files from a targeted computer to a USB key. The attacker can only plug the USB key (the autorun is disabled). Since this is a security area, the USB key could be manually analysed beforehand.
We consider that an extended version of “loader”is running on the targeted computer. If “loader”is captured by the security team, they will only find that it is waiting for USB events and is looking for files on mapped drive.
The attacker puts the encrypted shellcode on the USB key. If the security team analyses this key, they will only find a file with “random”data.
The attacker plugs the USB key into the targeted computer. “loader”detects the event, finds and loads the encrypted shellcode, decrypts it only in memory and jumps on its entry point. The shellcode could for example look for interesting files and copy them after encryption on the USB key.
This attack has been successfully executed in a lab. The malicious payload was a simple reverse shell.

4.4.3 Using shared secrets

If the malicious application is compounded of several modules, we can of course use the same key to encrypt every shellcode. Another solution is to use a secret sharing algorithm and to store a piece of the key in each part. Since the application needs all the modules to work, I decided not to implement a secret sharing algorithm with a threshold scheme like the Shamir’s Secret Sharing algorithm.
The secret key is then divided into N parts and the N parts are required to compute the key. First of all, we can imagine to split the key into N pieces and to use each piece as a part. So if the key is L bits length, each piece is L/N bits length.
This is a bad solution, because each time we have a piece, we get real information on the secret key. For example, if we get K ($0 \leq K \leq N$) pieces, we just have $\frac{L \times (N-K)}{N}$ bits to bruteforce to find the key.

I preferred to use the following algorithm:
- each module has a 256-bits private key;
- the secret key is the sum byte to byte of all private keys;
- all modules are encrypted with the final secret key;
- all modules contain their own private key (in clear).

4.5 Developing a reverse shell

4.5.1 Overview of the program structure

To take a practical example, let us suppose we want to develop a simple reverse shell, that is a backdoor that establishes a connection between a “cmd”process and a remote server.
The backdoor called later “rvshell”is compounded of two layer:
- the network layer that established the communication with the server;
- the application layer that creates the “cmd”process and uses the services exposed by the network layer

Figure 12 illustrates how rvshell works.

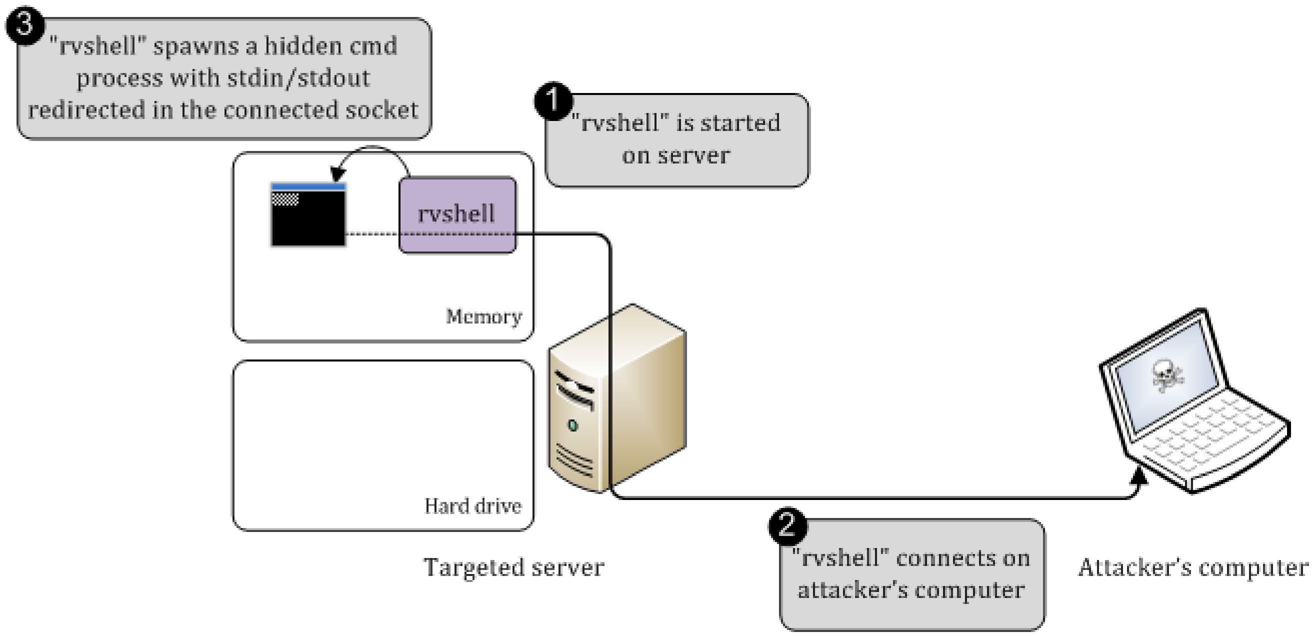

3 "rvshell" spawns a hidden cmd process with stdin/stdout redirected in the connected socket
1 "rvshell" is started on server
rvshell
Memory
Hard drive
Targeted server
2 "rvshell" connects on attacker's computer
Attacker's computer

Figure 12: Working principle of rvshell

We develop two modules:
- “ntstacksmpl”implements the network layer and exports two functions:

```
Prototypes of the functions exported by ntstacksmpl

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

The first opens a TCP connection on a server, the second closes a TCP connection.

– “rvshell”implements the application layer. It does not export any function but has an entry point, the function “ExecuteShell”that creates the “cmd”process and uses “OpenConnection”to open a TCP connection on the server.

Note that if we want to create a connection through a proxy by using the method “CONNECT”, we just have to develop a new module that exposes the same functions “OpenConnection”and “CloseConnection”. Instead of doing a simple TCP connect, the function “OpenConnection”of this module opens a CONNECT tunnel on the server.

4.5.2 Generating the reverse shell

4.5.2.1 Generating an executable

To generate an executable, we just give the following configuration file to WiShMaster:

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>
```

The module “log”and “ntstacksmpl”are inlined in “rvshell”, so it gains the capabilities to display formatted message and to open TCP connection.
Figure 13 shows this principle.

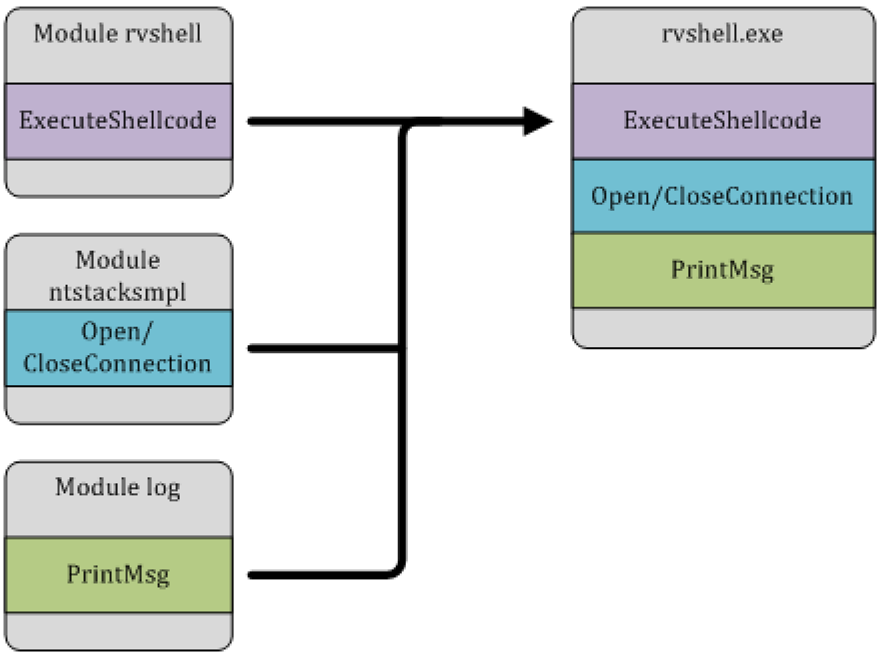


Figure 13: Result of the creation of the reverse shell as an executable

4.5.2.2 Generating a shellcode

To generate a shellcode, we give the following configuration file to WiShMaster:

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>
```

We just had the module initsh that contains all functions to execute shellcode initialisation. WiShMaster will then generate a shellcode that can be included after a XOR encryption in an executable: we just implement some kind of polymorphic reverse-connect shell.

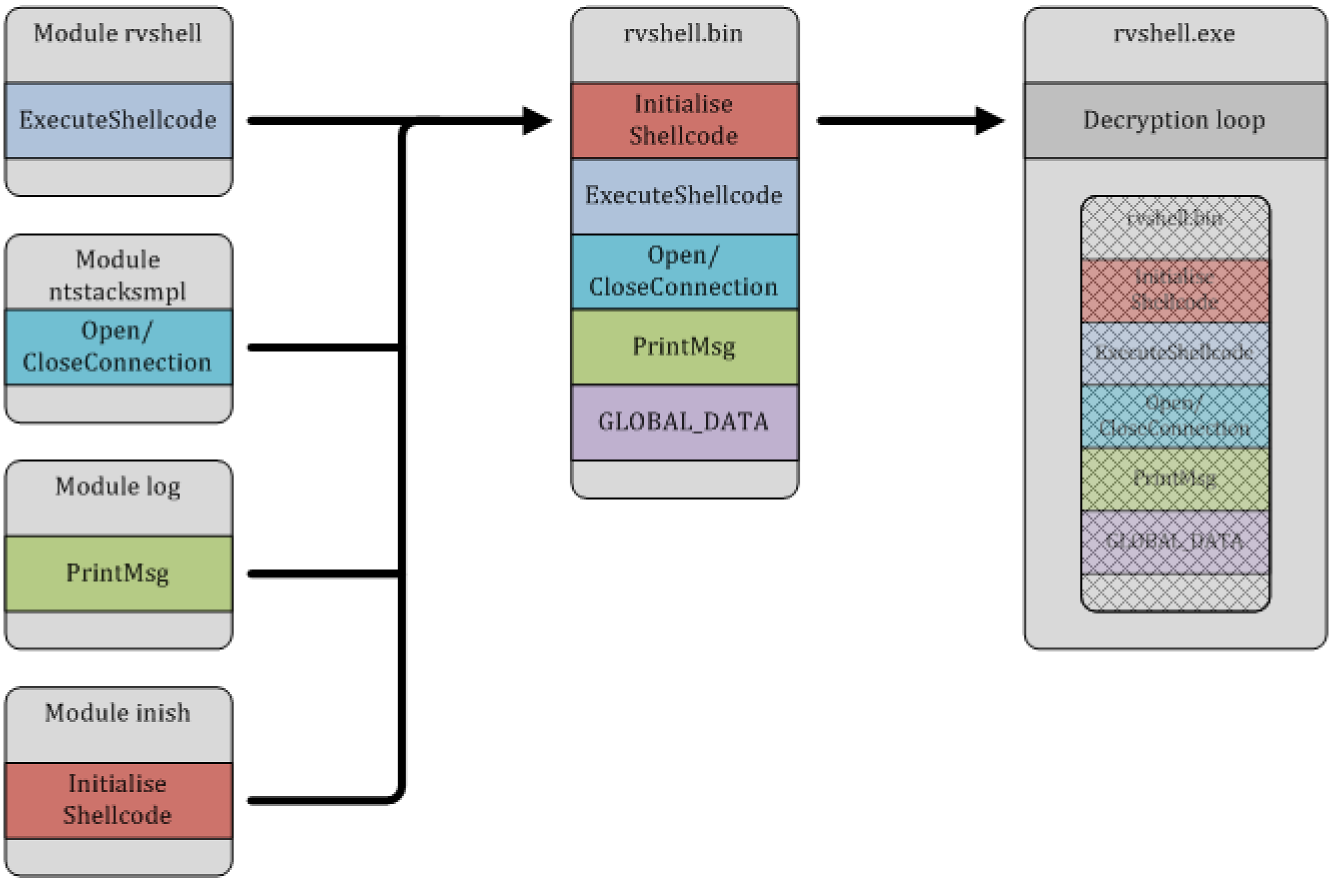


Figure 14: Result of the creation of the reverse shell as a shellcode

4.5.2.3 Using “loader”with a shellcode and a dll

In this case, “rvshell”is built as an encrypted shellcode, and “ntstacksmpl”as a dll. Those two modules will be loaded by “loader”. It is then pointless to include shared functions like “PrintMsg”or “Initialise-Shellcode”in each module. The module “log”and “initsh”are inlined in “loader”and considered as imported modules by “rvshell”and “ntstacksmpl”.
The shellcode “rvshell”is encrypted with a private key stored in “loader”. Figure 15 sums up this principle.

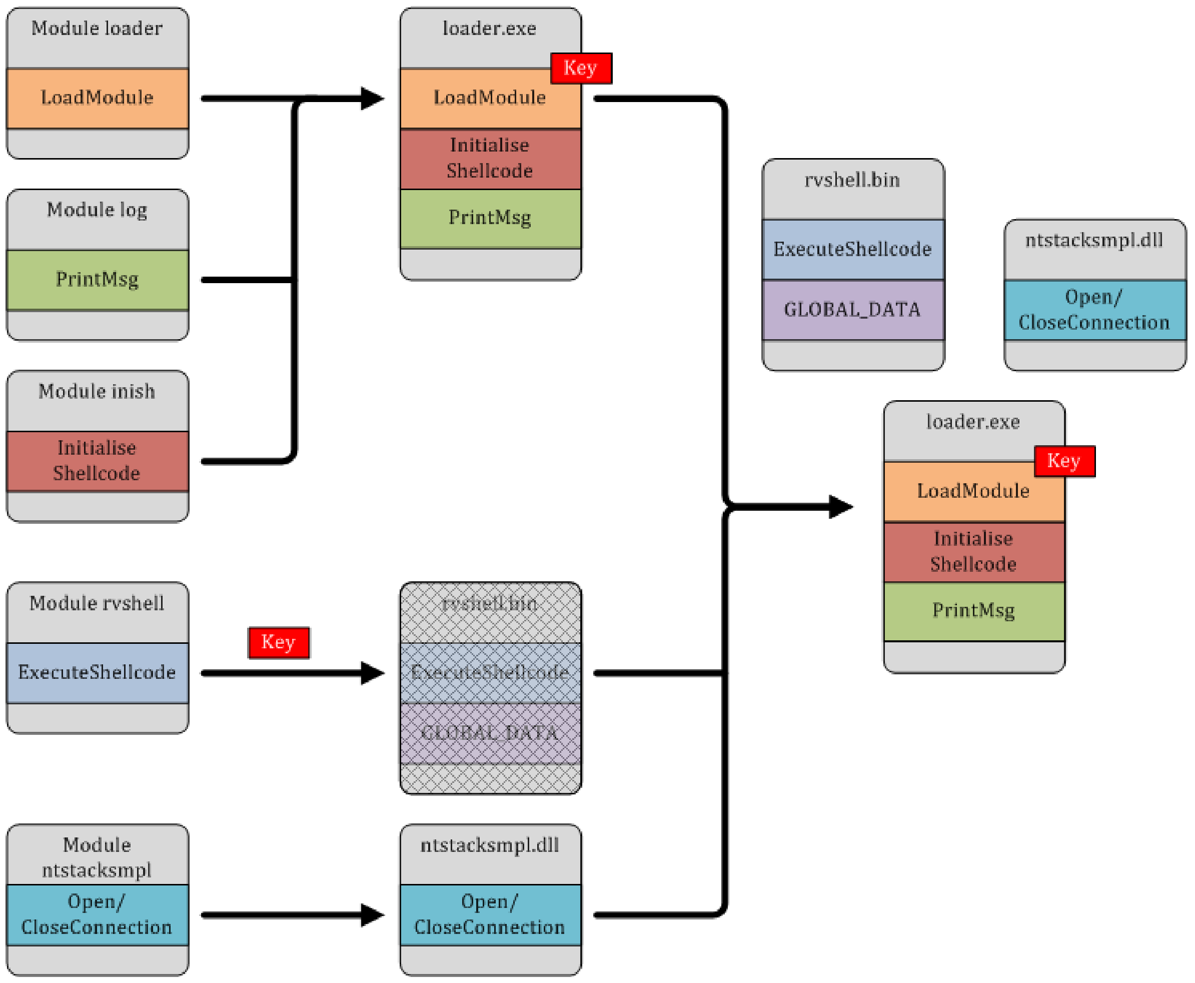


Figure 15: Execution of the reverse shell with loader; one shellcode encrypted by a secret key

4.5.2.4 Using “loader”with a two shellcodes and a shared encryption key

In this last case, we generate “rvshell”and “ntstacksmpl”as shellcode. These shellcodes are encrypted by a shared key (in red on figure 16) that is computed from three keys: one is stored in “loader”(the green), one in “rvshell”(the yellow) and the last in “ntstacksmpl”(the blue).

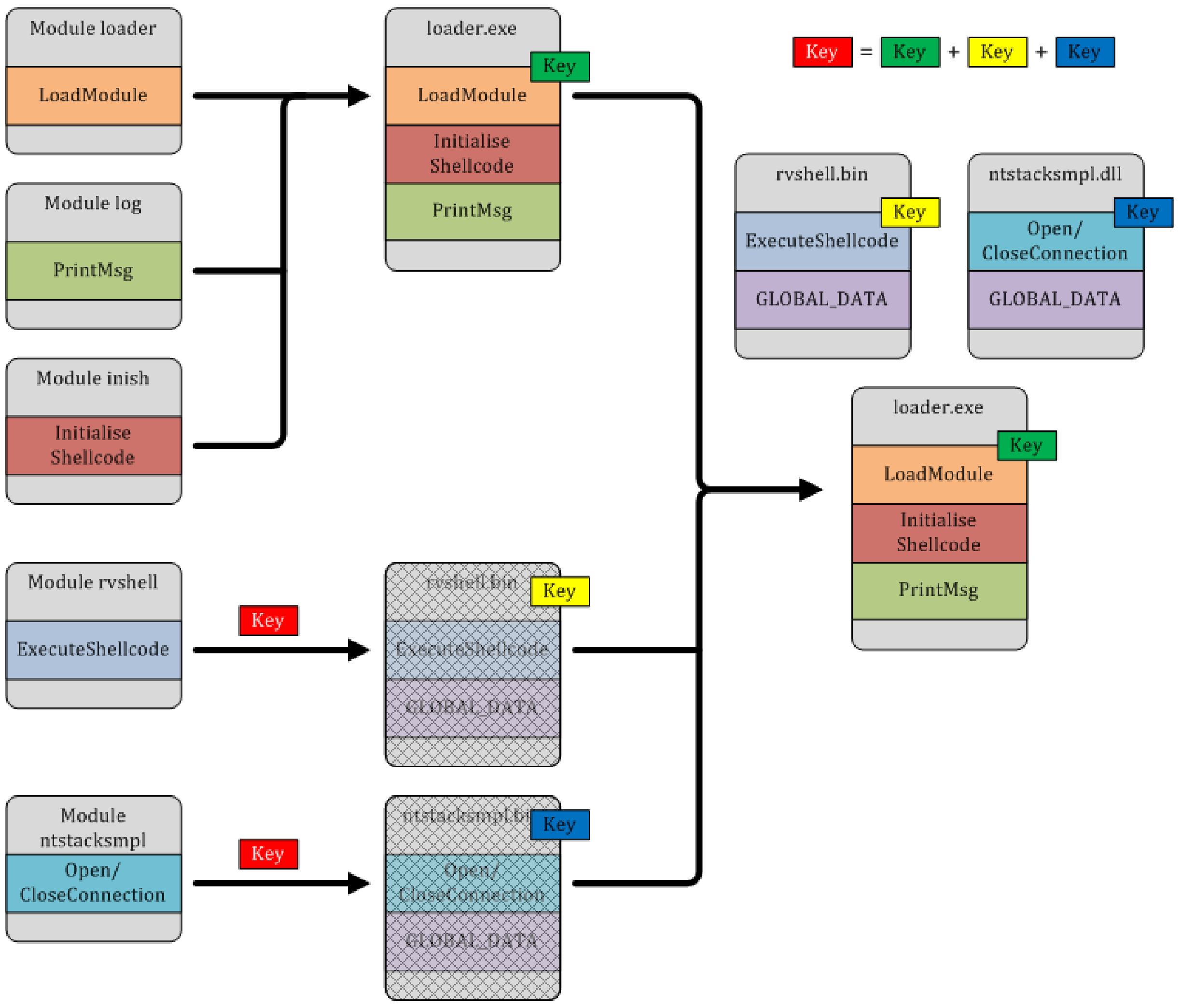


Figure 16: Execution of the reverse shell with loader; two shellcodes encrypted by a shared secret key

4.6 Implementing the USB attack to execute the reverse shell

4.6.1 Preparation of the attack

This part gives the main points of the implementation of an attack that relies on the plug of an USB key, as previously described.
Three additional modules have been developed:

- “searchmodindir”: executes recursive search of modules on directories. It exposes one function “AddDirectoryToAnalyse”that launches a new search of modules in a specified directory. When a module is found, “searchmodindir”calls the function “AddModuleToLoad”of “loader”with the full path to the corresponding file.
- “detectusbkey”: waits for the plug of USB keys. It uses standard functions of Windows API to inform the system that it wants to be warned when a USB key is plugged. It then calls the function “AddDirectoryToAnalyse”of “searchmodindir”with the corresponding drive letter.

– “injecter”: injects a shellcode in another process. The process may be a running process specified by name, a new process to start or a new instance of the default browser (launched hidden).

4.6.1.1 Generating secret key

Three private keys are generated. These keys are added to create a shared key “shared.key”.

4.6.1.2 Creating the reverse shell

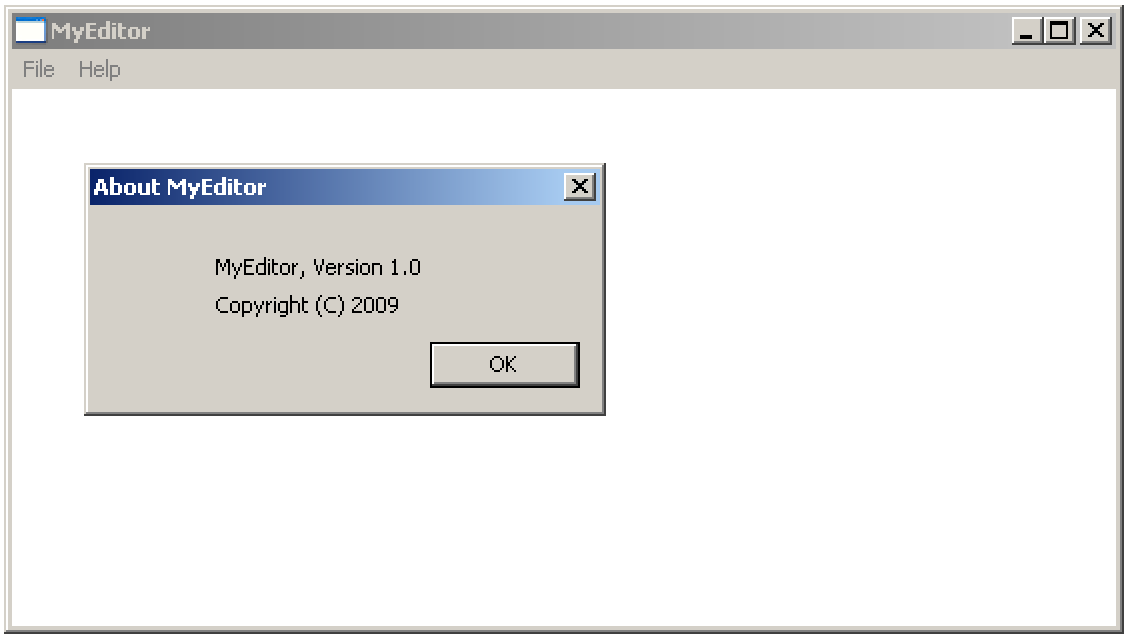
WiShMaster is used to transform “rvshell”and “ntstacksmpl”into two shellcodes.
These shellcodes are encrypted with “shared.key”. One private key is added to each module.
The two resulting files are placed on a USB key in two different directories.

4.6.1.3 Creating the loader

“loader”is created as a shellcode that inline the following modules: initsh, log, searchmodindir and detectusbkey. “loader”also contains the last private key.

4.6.1.4 Creating the Trojan

We will now create a Trojan that will contain “loader”. The application that will be infected is a simple test application called “MyEditor”that displays a message box when we click on the “Help”button in the tool bar.



MyEditor, Version 1.0
Copyright (C) 2009

Figure 17: The test application “MyEditor”

“loader”will be launched when the user tries to display this message box. For this, we will patch the “call”instruction that jumps on the function that displays the message box (called later “About()”).
Of course, we can jump directly on “loader”, but this has to disadvantages:

– firstly, we never return from “loader”. So the message box will not appear and the window will freeze;
– secondly the life of “loader”is then linked to the process “MyEditor”, which is not a good idea since the use may close this application.

We will rather use the module “injecter”to inject “loader”in a hidden instance of the default browser. We use WiShMaster to transform “injecter”into a shellcode. The final shellcode is then created by the concatenation of the shellcode injecter and the shellcode loader. This final shellcode is encrypted with a

XOR algorithm.
We then use an external tool - called “infector”- to infect MyEditor.exe:

– the last section is enlarged to add:
    - a decoding loop which contains the 32-bits key to decrypt shellcode injecter/loader;
    - the encrypted shellcode.
– the PE header is updated to reflect this modification;
– the call instruction used to jump on the function About() is patched by a call on decoding loop;
– a jump on the function About() is added at the end of the decoding loop.

Figure 18 sums up this principle

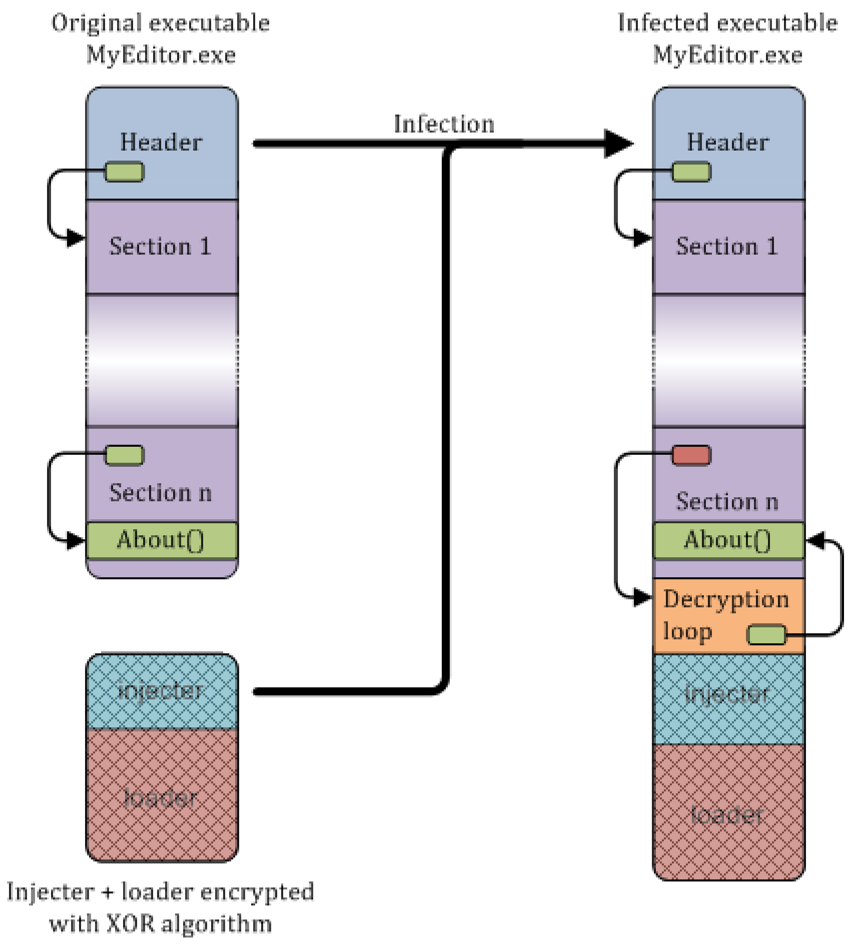


Figure 18: Principle of the infection of “MyEditor”

4.6.2 Sequence of the attack

The targeted user downloads “MyEditor.exe”on his computer. The local antivirus runs a verification by signature, emulates the executable and does not find any threat.
The user launches MyEditor. At a moment, he clicks on the About() button. The decryption loop is then called. It decrypts the shellcode infector/loader and jumps on infector. This one extracts full path to default browser from registry, starts a hidden instance of this application and injects it with loader. We return then from injector in decryption loop and jump on the function About(): the message box is displayed.
The user can now close MyEditor: loader is running in a hidden process and does not care.

Later, somebody comes to see the targeted user. He has an USB key. A quick scan shows that this key contains a few files with random data (maybe encrypted volume) but no executable or potentially dangerous files.
The man meets the targeted user and asks him if he can plug his USB key into his computer, of course under his surveillance. The key has been checked, the targeted user accepts. When the USB key is plugged in, Windows sends a message to “loader”. “loader”scans the USB key and finds the two modules. It loads them in memory, computes the shared key and decrypts them. It then jumps on the entry point of “rvshell”: the reverse shell is running.

5 Future work

WiShMaster version 2 is already able to generate shellcodes, but it is still under development. The main objective is to improve the analysis of C code to manage to remove the latest constraints on the code, imposed by the parsing with regular expressions.
The following paragraph sums up some of the ideas I am working on.

5.1 pycparser

pycparser is a C parser and an AST generator written in Python. It is available on google code web site [13]. It can be a better solution than using the browse file to parse the code: first it seems to give more information; secondly, it removes the link between WiShMaster and the Microsoft compiler.

5.2 Option fPIC of gcc

gcc integrates an interesting option “fPIC”that allow code to be generated without hardcoded address. PIC stands for “Position Independent Code”.
When this option is set, gcc inserts a call to a function that moves current eip to ebx before each access to a global data. Then the data is accessed relatively to ebx.
Let us take an example on Linux:

file test.c

```
#include <stdio.h>
char g_szMessage[]="Hello !";
void DisplayMessage(void)
{
        printf(">>> %s <<<\n", g_szMessage);
}

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

With a normal compilation the code contains hardcoded addresses:

Assembly generated with a standard compilation

```
$ gcc test.c -o test
$ objdump -d test
...
8048354 <DisplayMessage>:
 8048354:       55                        push   %ebp
 8048355:       89 e5                     mov    %esp,%ebp
 8048357:       83 ec 08                  sub    $0x8,%esp
 804835a:       c7 44 24 04 bc 95 04 08   movl   $0x80495bc,0x4(%esp)
 8048362:       c7 04 24 a8 84 04 08      movl   $0x80484a8,(%esp)
 8048369:       e8 22 ff ff ff            call   8048290 <printf@plt>
 804836e:       c9                        leave
 804836f:       c3                        ret
...
```

But if we use the fPIC option:

Assembly generated with the fPIC option

```
$ gcc test.c -o test -fPIC
$ objdump -d test
...
08048384 <DisplayMessage>:
 8048384:       55                      push   %ebp
 8048385:       89 e5                   mov    %esp,%ebp
 8048387:       53                      push   %ebx
 8048388:       83 ec 14                sub    $0x14,%esp
 804838b:       e8 4f 00 00 00          call   80483df <__i686.get_pc_thunk.bx>
 8048390:       81 c3 5c 12 00 00       add    $0x125c,%ebx
 8048396:       8b 83 f8 ff ff ff       mov    0xfffffff8(%ebx),%eax
 804839c:       89 44 24 04             mov    %eax,0x4(%esp)
 80483a0:       8d 83 0c ef ff ff       lea    0xffffef0c(%ebx),%eax
 80483a6:       89 04 24                mov    %eax,(%esp)
 80483a9:       e8 0a ff ff ff          call   80482b8 <printf@plt>
 80483ae:       83 c4 14                add    $0x14,%esp
 80483b1:       5b                      pop    %ebx
 80483b2:       5d                      pop    %ebp
 80483b3:       c3                      ret

080483df <__i686.get_pc_thunk.bx>:
 80483df:       8b 1c 24                mov    (%esp),%ebx
...
```

However, this option does not solve the problem of resolving imported function; Furthermore, the interval between the data is hardcoded, so we have to keep the same mapping in the shellcode. But most importantly, it seems this option does not work on Windows. I tried with gcc from Cygwin [14] and MinGW [15], and each time get the same message “this option is not supported”(or something a little bit less explicit…). In fact, this option has been added on Linux to support really shared libraries. Windows does not work in the same way and uses exclusively a relocation table to patch hardcoded addresses when a dll is not loaded at the preferred load address. Here is a link to an article that gives interesting information on this [16]
This option is notably used in ShellForge [17], a generator of shellcode for Linux developed by Philippe Biondi

5.3 Playing with pragma for fun...

We can use some pragma directives to group various elements together in a special section.
For example, if we change the code of “simpletest”a little:

Extract of patched code

```
#pragma code_seg("mysection")
__declspec(allocate("mysection"))
CHAR g_szMessage[]="This is a message stored as a global variable";
__declspec(allocate("mysection"))
CHAR sz00000001[]=">>> %s <<<";
__declspec(allocate("mysection"))
#define CreateFile fp_CreateFile
CreateFileTypeDef fp_CreateFile;

VOID DisplayMessage(IN CHAR * szMessage)
{
        PrintMsg(LOG_LEVEL_TRACE, sz00000001, szMessage);
}
```

Those directives indicate to place the function in a new section “mysection”. The “__declspec”allows the global data to be added in the code section.
Finally if we look at the generated .map file:

Extract of the generated map file

```
...
0003:00000000 0000127eH mysection               CODE
...
0003:00000000       ?fp_CreateFile@@...KKPAX@ZA 0042f000     display.obj
 0003:00000004       ?szS@@3PADA                0042f004     display.obj
 0003:00000010       ?g_szMessage@@3PADA        0042f010     display.obj
 0003:00000050       ?DisplayMessage@@YAXPAD@Z  0042f050 f   display.obj
 0003:00000080       ?DisplayFile@@YAHPAD@Z     0042f080 f   display.obj
 0003:00000200       ?DisplayData@@YAHXZ        0042f200 f   display.obj
...
```

We see that the functions and the global data are in the same section, so very close in address space.
If we look at the assembly:

Call of the function PrintMsg

```
        PrintMsg(LOG_LEVEL_TRACE, szS, szMessage);
0042F059  mov         eax, dword ptr [szMessage] ; szMessage = g_szMessage = 0x0042F010
0042F05C  push        eax
0042F05D  push        offset szS (42F004h)
0042F062  push        2
0042F064  push        0
```

It still contains hardcoded address, but if we extract the whole section “mysection”and map it at the same address in another process, it may work.

Call of the function CreateFile

```
        if((hFile = CreateFile(szFilePath, ..., NULL)) != INVALID_HANDLE_VALUE)
0042F0AC  push        0
...
0042F0C2  call        dword ptr [fp_CreateFile (42F000h)]
```

This technique suffers from a few limitations:

– we still have to handle imported functions;
– we still have some modification on code to do: replacing strings, adding pragma;
– we do not get a real shellcode since it contains hardcoded address and must be mapped at a specific address.

However, this might be interesting solution.

6 Conclusion

WiShMaster completely automates the shellcodisation process. A user can write their application in C and let WiShMaster handle all the transformation into a shellcode. Advanced viral techniques may be then very easily added.
WiShMaster tries however to do more than this. The proposed modular architecture allows the quick creation of an application that executes some specific operations from a library of modules. The developer just has to write few special modules and to assemble them with already existing modules to create a new application.

References

[1] Ivo Ivanov. Api hooking revealed. http://www.codeproject.com/KB/system/hooksys.aspx.

[2] A. Shamir. How to share a secret. *Communications of the ACM, 22: 612-613*, 1979.

[3] Upx, the ultimate packer for executables. http://upx.sourceforge.net/.

[4] Benjamin Caillat. Cracker, the **Cr**ypto **packer**. http://actes.sstic.org/SSTIC08/Rump_sessions/SSTIC08-Rump-Caillat-cracker.pdf.

[5] Benjamin Caillat. Wishmaster version 1. http://benjamin.caillat.free.fr/wishmaster.php.

[6] Microsoft browser toolkit. http://www.microsoft.com/downloads/details.aspx?FamilyID=621AE185-1C2A-4D6B-8146-183D66FE709D&displaylang=en.

[7] Description of the structure peb. http://msdn.microsoft.com/en-us/library/aa813706(VS.85).aspx.

[8] Metasploit. http://www.metasploit.com/.

[9] Pedump. http://www.wheaty.net/downloads.htm.

[10] Debugview. http://technet.microsoft.com/en-us/sysinternals/bb896647.aspx.

[11] Polarssl, small cryptographic library. http://polarssl.org/.

[12] Benjamin Caillat. Videos of x90re's backdoors. http://benjamin.caillat.free.fr/backdoorsvideos_en.php.

[13] pycparser. http://code.google.com/p/pycparser/.

[14] Cygwin. http://www.cygwin.com/.

[15] Mingw - minimalist gnu for windows. http://www.mingw.org/.

[16] Reji Thomas and Bhasker Reddy. Dynamic linking in linux and windows, part one. http://www.securityfocus.com/infocus/1872.

[17] Philippe Biondi. Shellforge. http://www.secdev.org/projects/shellforge/.