CS 5435 – Security and Privacy Concepts in the Wild

Homework 4

Instructor: Tom Ristenpart, TAs: Alaa Daffalla and Marina Bohuk *Spring 2024*

Instructions.

The deliverables in this homework are marked in blue boxes as exercises, and the end of the document includes a list of files that should be turned in. The task boxes are for your edification only, no deliverable is associated with tasks. You should submit your homework on Canvas as a PDF (we encourage using the provided LaTeX template) along with any code files indicated by the lab. You may complete the lab on your own or in a team of two, which you will indicate on Canvas and within the PDF.

Part 0: Getting Started

In this lab, we will be exploring control flow hijacking attacks including buffer overflows.

Setup. First log into the 64-bit Ubuntu VM we've set up in Google Cloud.

```
student@laptop:  ssh your-net-id@35.226.118.201
```

The password is also your netID, and you should be prompted to change it as soon as you log in.

Now retrieve the source code from the Github repository:

```
student@CS5435-VM:  git clone https://github.com/cs5435/cs5435-hw4.git
```

This will expand to a directory `demo` containing the part 1 target, a directory `targets/` containing the vulnerable targets for part 2 and the exploit templates `sploits/` for part 2. The targets have already been compiled for you and can be found at `/srv`.

We will grade your exploits by compiling them with

```
gcc -ggdb -m32 sploitX.c -o sploitX ; echo whoami | ./sploitX
```

and checking that the output is `targetuser`. We will test sploit4 with /bin/sh pointed to /bin/zsh and test sploitEC with /bin/sh symbolic linked to /bin/dash. We will also inspect the sploit source code to ensure that they are actually executing against the appropriate targets.

Part 1: Command Line Buffer Overflow of meet.c

We will start by walking you through the buffer overflow exploit shown in class. Investigate `meet.c`, reproduced in Figure 1, which is a slight simplification of the one from the Gray Hat Hacking textbook. Notice the blatant buffer overflow vulnerability on line 9 in `greeting()`. Recall from class that we can exploit

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


void greeting( char* temp1 )
{
  char name[400];
  memset(name, 0, 400);
  strcpy(name, temp1);
  printf( "Hi %s\n", name );
}


int main(int argc, char* argv[] )
{
  greeting( argv[1] );
  printf( "Bye %s\n", argv[1] );
}
```

Figure 1: The `meet.c` demo program.

this by smashing the stack. There are many sources online to understand this kind of basic buffer overflow exploit, you can read about it in Aleph1's seminal article from Phrack, in the Gray Hat Hacking book, and elsewhere.

The program expects a string from the command line. The string is copied into a local array `name` within `greeting()`. Because `strcpy()` will happily continue copy from the source string until it hits a null byte, a malicious input can force the program to copy more than 400 bytes into the local array. The local array is stored on the stack, but so is important control information such as the return address. When `greeting()` is called from `main()` the address of the next instruction of `main()` is placed on the stack, followed by the base pointer, followed by the local variables of `greeting()`. The convention on x86 systems is that the stacks grow towards lower addresses, so the array `name` is below the saved return address. Writing into the buffer, however, goes in the other direction, and overflowing the buffer will therefore begin overwriting the saved base pointer and then the saved return address.

The basic idea is that we will prepare a special input that arranges for the `strcpy()` to overwrite the stack with the combination of a NOP sled, a sequence of instructions to launch a shell, and (multiple copies of) an appropriate address that will, if calculated correctly, point back into the buffer somewhere in the NOP sled. For a diagram, see the slides from class or the Gray Hat Hacking book.

`meet.c` has already been compiled with the following command and can be found at `/srv/target0`:

```
gcc -ggdb -mpreferred-stack-boundary=2 -zexecstack -fno-stack-protector -no-pie -fno-pie
 -m32 meet.c -o /srv/target0
```

The first option tells the compiler to include debug symbols, which are useful when inspecting behavior of the program with the debugger `gdb`. The next option tells the compiler to align the stack in 4 byte chunks rather than optimizing it. The next four options turn off protections against stack smashing attacks: modern programs should not allow executing instructions from the stack and should include stack canaries. We turn these off to make writing exploits simpler, but even with them one can construct suitable, if more complicated, exploits.

We have turned off address space layout randomization (ASLR). This countermeasure has the kernel partially randomize the layout of the process's memory each time it is executed, which makes calculating the proper address to point back into the buffer more complicated.

Now we can try exploiting the buffer overflow. First, use `gdb` to inspect what happens in `greeting()`. Run

```
gdb -q /srv/target0
```

And type the command `run` `` `perl -e 'print "A"x399'` ``. Note that this uses backticks `` ` `` around the perl command. This tells amounts to telling the interpreter to first run the perl command and use its output as the second argument to `meet`. The perl command just prints out 400 copies of the character A, whose ASCII representation is 0x41.

The program should execute just fine. Now try incrementing 399 to 400 and re-running. (You can just press the up arrow in gdb's interactive console and then edit the command.) The program should now give a segfault. Let's find out why.

Set a breakpoint in `greeting()` right before executing the call to `memset()`. To do this in gdb type in `b 8`, which tells it to set a breakpoint to pause the execution at line 8 of the source code. Now run `run` `` `perl -e 'print "A"x400'` `` again. The program should pause for you right before the memset. Let's inspect the stack. First you can type `info frame` which has gdb print out information about the current stack frame. You'll see the address of the saved instruction pointer (EIP), the saved base pointer, etc. You can also print out the contents of memory near where the return address for this stack frame exists. Find the location of the saved EIP, it should be under the section "Saved registers" and look something like `0xffffff5cc`. Then type in `x/64x 0xffffff5cc-32`. This tells it to display the values of memory starting at the address 32 bytes before 0xffffff5cc. Remember stacks grow upwards, so we're showing some portion of — $7 * 4 = 28$ bytes to be exact — of the memory locations reserved for the local variable `name`. The next 4 bytes are the saved base pointer (EBP), and the next four bytes are the saved return address (EIP). Remember x86 is little-endian, so the bytes within a 4-byte address are in reverse order.

Single-step the program by one line of code, by typing 'n' and enter. Then inspect memory again via `x/64x 0xffffff5cc-32`. You should now see a bunch of zeros through the first 28 bytes of the printout — this is the result of memset, which just sets all those byte values to zero. Now single-step one more time, and you should see those replaced by 0x41 values (ASCII for "A"). But if you look carefully, one more value gets changed, the low byte of the saved EBP is set to zero. The reason is that `strcpy()` always places a null character at the end of the string, and so copying 400 bytes into `temp` actually results in 401 bytes being written to the array, and hence we've overflowed the buffer by one byte. If you keep stepping through, you should see a segfault occurring later, because munging the EBP interferes with proper operation of the stack after you return to `main()`.

Repeat the above procedure, with different numbers of A characters, and observe the resulting impact on the saved EBP and EIP.

Now we're ready to construct an exploit for this buffer overflow. We will use some shell code, together with a simple trick called the NOP sled. To do so you can just construct a new exploit, via:

```
perl -e 'print "\x31\xc0\x31\xdb\x31\xc9\x31\xd2\xb0\xa4\xb3\x1f\xb1\x1f\xb2\x1f\xcd\x80
\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b\x89\xf3\x8d\x4e\x08\x8d
\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd\x80\xe8\xdc\xff\xff\xff/bin/sh";' > sc
```

You can read about what the shellcode is doing in his paper on it. In short, it is a sequence of carefully crafted instructions that avoid using the null (0x00) byte while calling the system call `execve()` to launch a new shell.

Now you can construct an exploit on the command line as follows:

```
/srv/target0 `perl -e 'print "\x90"x200';``cat sc``perl -e 'print"\xf8\xf2\xff\xbf"x38'
;`
```

The last 38 repetitions are of an address that should, hopefully, fall within the first 200 bytes of the array `name`. You can calculate what address this should be by running the included `get_sp` program, which simply prints out the address of the top of the stack of a program that similarly calls one function from `main()`. We can then subtract some number from this, call it about 0x300. Why subtract? Well note that the command line arguments are, themselves, placed on the stack for `main()` and this pushes down the stack further in the address space. We can be a bit approximate though — our use of a NOP sled plus repetition of the address value should give us leeway about not quite being accurate.

That said, the exploit may not work the first time, because of alignment issues. We need one of the 4-byte buffer addresses to exactly overwrite the return address. Shifting it even just one byte in either direction will cause problems, as the resulting address will be (for example if we shift once too far left) `0xfff2f8ff`, as opposed to `0xffffff2f8`. So try running the command but replacing 200 with 201, 202, 203, and one of these should work.

You should get a shell. Type `id` or `whoami`. It should tell you what user your shell is now running as. Instead of a root shell, we've set the owner of the targets to be `targetuser`. Since you're working in a shared environment, make sure not to make any changes to the targets or their permissions; we will be checking these periodically.

Congratulations, you've now run your first control flow hijacking attack that resulted in privilege escalation.

More GDB commands. A couple further helpful GDB commands. There are plenty of guides about GDB on the web, too, and it will be helpful to familiarize yourself with some of them in order to fully understand the state of a program and step through it.

- `x/x $ebp` – displays the current register value for `$ebp`. Can also use with `$esp` and `$pc` (the program counter, also known as the instruction counter).
- `x/x *(char**)environ` – prints out the memory of environment variables in the current process. Note that `execve` allows you to control the environment variables passed to the launched process.
- `disassemble` – shows the machine instructions of the current line of code. You can also disassemble particular target functions by adding the function name as an argument to the command.
- `stepi` – step one instruction at a time through program.
- `p var` – prints the current value of the variable "var". This command can also be used for finding addresses of libc functions, e.g. `p system` will print the address for `system`.

Part 2: Stack Smashing Exploits

Targeting meet from a program. We will start by converting the demo above into the format of a more easy-to-grade format. We have included stub exploit files, in the form of `sploitX.c` along with `shellcode.h`. The former is a template for your solution. The latter includes shellcode similar to AlephOne's, which you can use to help you construct an exploit. You are welcomed to write your own shellcode, but the provided `shellcode.h` should be enough for exploits 0 to 3.

The value of exploiting a program via another program (as opposed to on the command line) is that you can more easily control the environment. Recall that the environment variables end up impacting the layout of memory.

Exercise 2.1: Implement a control flow hijack in `sploit0.c`. It must assume the setuid target is at /srv/target0. We will grade this by compiling `sploit0.c`, running the result, and observing if it obtains a shell for `targetuser`.

Hint: You may want to turn on debug tracing through the `execve`. Below are some helpful notes:

- Run `gdb -e sploitX -s /srv/targetX` in your bash to tell `gdb` reading symbols from `/srv/targetX`.
- In `gdb`, run `catch exec` to let `gdb` notify you on `exec()`.
- Once you are in the `targetX` process, you can break any line in `targetX.c` as in command-line attacks.

Hint: You will need to calculate a new return address to jump to, as compared to the command line version. You may need to try various offsets to find the right one—try both adding and subtracting.

Further targets. We have included three more targets compiled with an executable stack: `target1.c`, `target2.c`, `target3.c`.

Exercise 2.2: Implement exploits for `target1.c`, `target2.c`, `target3.c` that give you a shell for `targetuser`.

Hint: Some of the buffers are too small to include your exploit. Think about other locations you can store the exploit string that get passed to the target program when calling `execve()`.

Attacks that don't inject code. Now we turn to `target4.c` which we will compile as a non-executable stack. Our typical approach to building exploits, injecting code into the process and hijacking flow to execute it, won't work. That's because the system now marks memory pages associated with the stack as non-executable. But that doesn't mean that vulnerabilities can't be exploited, we will just need a different approach. For this target you will need to read about return-to-libc attacks.

Exercise 2.3: Implement a return-to-libc exploit for `target4.c` that runs a shell.

Assuming you built a typical return-to-libc exploit, you may not be able to get a shell. This is because of a countermeasure in modern shells, such as bash, in which they drop privileges automatically. You can confirm this by trying a different shell that lacks this countermeasure such as `/bin/zsh`. We will give you full credit for Exercise 2.3 even if you are only able to successfully obtain a shell using /bin/zsh.

As a special challenge, you can try to build an exploit that works despite the mechanism. This will be worth extra credit.

Exercise 2.4: [Extra credit] Implement an exploit for `target4.c` that gives a `targetuser` shell despite the shell drop privilege countermeasure. Put this in sploitEC.c.

Part 3: Short Answer questions

Exercise 3.1: Briefly explain the vulnerabilites in `target1.c`, `target2.c`, `target3.c`, and `target4.c`.

Exercise 3.2: Does ASLR help prevent return-to-libc attacks? If so, how? If not, why not?

Exercise 3.3: Explain the stack protector countermeasure, which we turned off via the "`-fno-stack-protector`" option to gcc. Which of your exploits would be prevented by turning on this stack protection?

Final deliverables. You will submit the following on Canvas:

- PDF of solutions to Exercises 3.*.
- The sploit*.c files

The submitted code will be tested within our own instance of the VM environment.