## Contents

# 1 PKU-ICS: The Attack Lab X

## 1.1 Introduction

In the previous part of the attack lab, you should have already been familiar with how to exploit buffer overflow vulnerabilities to transfer the control of a vulnerable program to a certain function. However, there is an important defense against this kind of attack which has not been applied yet.

You may want to try this simple program yourself. If your operation system is not too old and the following code snippet (saved in the file `test.c`) is compiled with default options (e.g. `gcc test.c -o test`), you are expected to see different outputs if you run the program multiple times.

```c
#include <stdio.h>

int main(void)
{
  printf("The address of main is %p.\n", &main);
  return 0;
}
```

One possible output you may see is:

```
$ ./test
The address of main is 0x565545e8f139.
$ ./test
The address of main is 0x559eb0bee139.
```

These two addresses are not related directly to the address you see via `objdump -S`:

```
$ objdump -S ./test
  ... ...
0000000000001139 <main>:
    1139:       55                          push   %rbp
    113a:       48 89 e5                    mov    %rsp,%rbp
  ... ...
```

The technique is known as Address Space Layout Randomization (ASLR). ASLR randomly arranges the address space positions of key data areas of a process (as a result, the function addresses will no longer be a constant) to effectively prevent attackers from overwriting the return address by a fixed address to jump to, for example, a particular exploited function.

Since ASLR randomly arranges the code, it requires the code to be position-independent (which means the code can be always executed correctly regardless where it is put in the memory). In x86_64, GCC generates position-independent code by default. The Linux kernel enables ASLR by default since the kernel version 2.6.12, released in June 2005. It implies that today you will almost always need to cope with ASLR if you want to exploit some practical programs.

For your information, if you do not want ASLR, you can disable ASLR in Linux kernel by `echo 0 | sudo tee /proc/sys/kernel/randomize_va_space` or instruct GCC not to generate position-independent executable by `gcc test.c -no-pie -o test`.

## 1.2 Exploit even with ASLR

Overwriting the return address by a fixed value doesn't simply work. However, the relative offset is still fixed regardless the use of ASLR.

You can try out this program yourself:

```c
#include <stdio.h>

int foo(int a, int b)
{
  return a + b;
}

int bar(int a[], int b)
{
  return a[b];
}

int main(void)
{
  printf("The relative offset between foo and bar is %lu.\n",
      (size_t) &bar - (size_t) &foo);
  return 0;
}
```

The expected output may be something like:

```
$ ./test
The relative offset between foo and bar is 20.
$ ./test
The relative offset between foo and bar is 20.
```

which is consistent with the output of `objdump -S`:

```
$ objdump -S ./test
  ... ...
0000000000001139 <foo>:
  ... ...
000000000000114d <bar>:
  ... ...
```

It means that if you can find out some way to mislead the program to print the address of any certain instruction (which is usually referred as the instruction address is leaked), you can add the relative offset and get the address of the function you want to transfer the control to. As for the remaining part, you should have been already familiar after completing the previous part of the attack lab.

In order to leak the instruction address, you have to communicate with the vulnerable program, get the address, perform the calculation and finally communicate with the program again to exploit the buffer overflow bugs. Therefore, in the final part of the attack lab, you have to submit a piece of program to automatically perform the whole attack process. But don't worry, we will provide a framework for you.

## 1.3 Handout

You will get three files in the handout:

1. `secret`
2. `handin.c`
3. `check.sh`

The vulnerable executable is `secret`. It contains some buffer overflow bugs and you should find them out and figure out a way to exploit them. Note there is a function named `YouWin` inside. Your final goal is to mislead `secret` to transfer control to `YouWin`.

We have prepared a framework in `handin.c` for you to make it easier. It should be compiled with the following instruction:

```
$ gcc handin.c -lpthread -o exp
```

The framework works as follow: It first runs `secret` in the background (the first part), then communicates with `secret` to exploit vulnerabilities (the second part), and finally puts `secret` in the foreground to allow you to communicate with it if you need (the third part).

A function named `exploit` acts as the second part (i.e. communicating with `secret` to exploit vulnerabilities). It takes two arguments, `rfile` and `wfile` respectively. When you write something into `wfile`, it goes into `secret` as the input. And when `secret` prints something, it goes into `rfile` as the output so that you can read from it (if you need).

```c
static int exploit(FILE *rfile, FILE *wfile)
{
  // TODO: Write your code to exploit the buffer overflow bugs

  return 0;
}
```

Note the function `exploit` is the only function you need to modify. You shouldn't need (although you are allowed) to modify any other functions in `handin.c`.

Here we illustrate the above process by several examples. First, if there is nothing in the function `exploit` (as shown in the above code snippet), the second part simply does nothing and the `secret` will be put in the foreground (in the third part). In this case, executing `exp` should be nothing more than just directly executing `secret` (except a few redundant lines):

```
$ ./exp
spawn pid=113178
Press <Enter> to continue...

exploit ok
What's your name?
Bob
Hello Bob
Tell me somethin. If you please me, I'll let you pass the test.
First, tell me how long is your story.
  ... ...
```

Here `spawn pid=<pid>` and `Press <Enter> to continue...` is used for debugging. `exploit ok` shows your `exploit` function completes successfully and returns zero. All other lines are the same as what you will see if you run `secret` directly:

```
$ ./secret
What's your name?
Bob
Hello Bob
Tell me somethin. If you please me, I'll let you pass the test.
First, tell me how long is your story.
  ... ...
```

Now you should already note `secret` will first print a line contained `What's your name?` and ask for your input as the name. We will present a demo to do this automatically with the provided framework. You can fill the function `exploit` with the following code:

```c
static int exploit(FILE *rfile, FILE *wfile)
{
  char buf[128];

  // Read a line from `secret`...
  fgets(buf, sizeof(buf), rfile);
  // Now `buf` should contain "What's your name?\n"
  // Let's see...
  printf("Got something from secret: %s", buf);

  // Ok, tell `secret` my name
  fprintf(wfile, "Bob\n");
  // You have to flush `wfile` after you output something
  // Don't forget this!
  fflush(wfile);

  // Read a line from `secret` again...
  fgets(buf, sizeof(buf), rfile);
  // You should see "Hello Bob" here
  printf("Got something again from secret: %s", buf);

  // After the function returns, `secret` will be put in the
  // foreground, and you can interact with it if you want.
  return 0;
}
```

As described in the comments, it communicates with `secret` in three rounds: first read out the line `What's your name?`, then tell `secret` the name, finally read out another line `Hello Bob`. As a result, you will see this if you run the program:

```
$ ./exp
spawn pid=114136
Press <Enter> to continue...

Got something from secret: What's your name?
Got something again from secret: Hello Bob
exploit ok
Tell me somethin. If you please me, I'll let you pass the test.
First, tell me how long is your story.
11
OK, let's get down to business.
  ... ...
```

Again, your goal is to modify the `exploit` function to exploit buffer overflow bugs in `secret` and finally mislead `secret` to run the function `YouWin`. Once you finish, you can verify your solution by running `./check.sh`. It will show `Success: Your string is correct` if you pass.

## 1.4 Handin

Your modified `handin.c` is the only file that is required.

## 1.5 Debugging

You may want to debug the vulnerable program while you are attacking. To do so, you need to first disable strict ptrace checks by

```
echo 0 | sudo tee /proc/sys/kernel/yama/ptrace_scope
```

Note it is not allowed in the class machines, so you have to do this in your own devices.

Then when you run `exp`, do not hurry to press the <Enter> key. For example, when you see something like:

```
$ ./exp
spawn pid=48338
Press <Enter> to continue...
```

You can remember the number followed by `pid=` and start another terminal to run `gdb -p pid` where `pid` is the number you have remembered.

```
$ gdb -p 48338
  ... ...
Attaching to process 48338
  ... ...
0x00007fd539563862 in read () from /usr/lib/libc.so.6
(gdb)
```

Here you can use any GDB commands that you have been familiar with. For example, you may have already known that there is a critical function named `vul` in `secret` (if you haven't, you may need to first perform some quick analysis by inspecting `objdump -S secret`). It is the function that first prints `What's your name?` by calling `puts` and ask for your input by calling `Gets`. You can set a breakpoint at the return of `Gets` to find out our input string (though you already know it is `Bob`, since the previous demo is used in this example).

You should first disassemble `vul` to find out where the instruction is located at run-time:

```
(gdb) disas vul
Dump of assembler code for function vul:
   0x000055aadb8c2343 <+0>:    endbr64
   0x000055aadb8c2347 <+4>:    push   %rbp
   0x000055aadb8c2348 <+5>:    mov    %rsp,%rbp
  ... ...
   0x000055aadb8c2377 <+52>:   lea    -0x100(%rbp),%rax
   0x000055aadb8c237e <+59>:   mov    $0xf8,%esi
   0x000055aadb8c2383 <+64>:   mov    %rax,%rdi
   0x000055aadb8c2386 <+67>:   call   0x55aadb8c2240 <Gets>
   0x000055aadb8c238b <+72>:   lea    -0x100(%rbp),%rax
  ... ...
```

You will find that `Gets` is called at instruction `0x000055aadb8c2386`, so our breakpoint will be set at the following instruction `0x000055aadb8c238b`:

```
(gdb) b *0x000055aadb8c238b
Breakpoint 1 at 0x55aadb8c238b
(gdb) c
Continuing.
```

Do not forget to continue the program by executing the last `c` command. Now go back to `exp` and press the <Enter> key.

```
$ ./exp
spawn pid=48338
Press <Enter> to continue...

Got something from secret: What's your name?
```

You can see `exp` is stuck, because our breakpoint has been triggered. Return back to the debugger and check our input string:

```
Breakpoint 1, 0x000055aadb8c238b in vul ()
(gdb) p (char *)($rbp - 0x100)
$1 = 0x7ffdf5f9a7d0 "Bob\n"
```