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 the default compiler 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 -d a.out
  ... ...
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.

1.3 Interact with the Program

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. In this final part, you have to submit a piece of program to automatically perform the whole attack process.

We have already provided a framework for you to make it easier. You do

not need to understand how the communication between your attack program and the vulnerable program is set up. You need only to modify one function named `exploit`. When the function starts to run, the communication has already been set up, and you could write something to `wfile` (don't forget to add the newline or call `fflush` to flush the stream, see the example below) or read something from `rfile` to communicate with the vulnerable program.

```
static int exploit(FILE *rfile, FILE *wfile)
{
  // Here is a demo on how to interact with the program:
  char buffer[0x10];

  fgets(buffer, 0x10, rfile);
  printf("Recive from target program: %s", buffer);

  memset(buffer, 'a', 0x10);
  fwrite(buffer, 0x10, 1, wfile);
  fflush(wfile);

  fgets(buffer, 0x10, rfile);
  printf("The target string is: %s", buffer);

  // TODO: Write your code to exploit the buffer overflow bugs

  return 0;
}
```

1.4 Handout

There are three files:

1. `secret`: the vulnerable executable, which should be analyzed in order to write your exploit code
2. `handin.c`: the framework to interact with the executable, where your exploit code should be written
3. `check.sh`: a bash script to check whether you pass the lab

The goal is to execute function `YouWin` in exectuable `secret` and obtain the secret string. Note simply printing the string will not work, because a different string is used when we are scoring your exploit.

1.5 Handin

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

1.6 Debugging

You may want to debug the vulnerable program while you are attacking. To do so, you need to first disable restricted 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` which is generated by `check.sh`, 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 -q -p 48338
  ... ...
Attaching to process 48338
  ... ...
0x00007fd539563862 in read () from /usr/lib/libc.so.6
(gdb) c
Continuing.
```

Here you can use any GDB commands that you have been familiar with. When you have done (e.g. after setting up some breakpoints), do not forget to send the final `c` command, return back to the program `exp` and press the <Enter> key.