LINGI2144: Secured System Engineering
Tutorial 4: Buffer Overflow

Academic year : 2020 - 2021

Teacher : Legay Axel
Course : LINGI2144
Collaborators :
Crochet Christophe
Duchene Fabien
Given-Wilson Thomas
Strebelle Sebastien

1 Prerequisite

Working directory: `~/SecurityClass/Tutorial-04`

Connection:

username	password
admin	nimda

1.1 Environment Configuration

On most Linux systems and with most compilers there are protections built in to prevent various exploits. For today's tutorial we may have to turn some of these off.

One is the randomisation of memory segments by the Linux kernel. We can see the current value with

```
sudo cat /proc/sys/kernel/randomize_va_space
```

This is "2" by default on Kali Linux (and most Linux systems). To turn this off for the rest of the sessions by setting the value to "0" we can run

```
echo 0 | sudo tee /proc/sys/kernel/randomize_va_space
sudo cat /proc/sys/kernel/randomize_va_space
```

The compiler can also insert various protections into code that is compiled. For the stack, `gcc` includes some stack protection by default on many versions. We can force this to be turned off with the argument

```
-fno-stack-protector
```

At times we may also wish to force gcc to compile code with executable instructions on the stack, we can enable this with

```
-z execstack
```

2 Exercise

2.1 Buffer Over ow on the Stack (from lecture)

Building the `password.c` file in a bad way:

```
gcc -fno-stack-protector -o password password.c
```

Note that we have disabled stack protection and are assuming no address space randomisation here.

Now let's see what we can observe from the behaviour by trying passwords of different length for the password program.

You should be able to provoke four different behaviours from this code:

1. `"access granted"` with the password `"good"`
2. `"bad password!"` with the wrong password
3. both `"bad password!"` and `"access granted"`
4. `"bad password!"` followed by a segmentation fault.

2.2 Buffer Overflow Minor Variations

Here we have the same code as before in `password.c`, but we can check and see the effect of the environment as configured in 4.1.

First let's build the code with stack protection enabled:

```
gcc -fno-stack-protector -o password password.c
```

Now if we run the same experiments as before we expect the code to report that "stack smashing" is detected and this should prevent both access granted after a bad password, and segmentation faults.

Inspect the two different versions of the code with `gdb` and see if you can find where and how the stack protection is implemented. Do you know which kind of stack protection this implementation is?

Next the code in "`reordered.c`" makes minor changes to the order of the declared variables in the "`check_authentication`" function. This code may or may not also have the same vulnerability, test on your system and with your compiler(s) to see what happens. Note that you may experience different results with different compilers, try:

```
gcc
clang
g++
```

Typically you can find what compilers (and compiler modules) are available with

```
dpkg --list | grep compiler
```

The VM only has the three above for now, but you can look on other systems.

2.3 Infinite Loop by Overflow

The code in "`simple-loop.c`" has a function that copies from an argument into a buffer. However, the bounds checking for the copy has an off-by-one error allowing an extra byte to be copied. This can cause an infinite loop for some compilers.

Compile the code with to enable debugging information and then use `gdb` to explore the values inside foo.

```
gcc -g -o simple-loop simple-loop.c
```

WARNING: Results here can be compiler dependent

Run with and then experiment with the argument to create the infinite loop.

```
./simple-loop test
```

When this has succeeded, use `gdb` to explore how this happens.

- HELP: The following command should create an infinite loop:

  ```
  ./simple-loop 0123456789abcdef
  ```

  if this does not, then you may can check with `gdb` to find out why (or why it is impossible).

CHEAT: (Read me if you cannot create an infinite loop!) The reason for the infinite loop is due to the null termination character from the input string being able to over-write the loop counter. This may not work, e.g. if the compiler re-orders the arguments. If this did not work for you, have a look and see what the compiler did to prevent it happening.

Does clang also allow an infinite loop? Can you fix the code so this doesn't happen?

2.4 Infinite Loop by Overflow (from lecture)

The code in "loop.c" does nothing interesting, except that the argument sent to the "foo" function can be changed by you inside main. You should be able to provoke an infinite loop in the code by using an overflow of the strcpy function inside "foo" to overwrite the EBP and EIP values on the stack.

Compile the code with to enable debugging information and then use gdb to explore the values inside foo.

```
gcc -g -o loop loop.c
```

HINT: you can use

```
x/24xw newbuffer
```

when inside the foo function to see values around the "newbuffer" as allocated on the stack. By comparing these with the known return point in the main function and the value of EBP you should be able to find what you're looking for.

HINT: You can change the value of TARGET and write other information into (or around) "buffer" in the "main" function

2.5 EBP abuse

The code in "exploitme.c" has an off-by-one error and very similar structure to the code in 4.4. Except now the value of the loop counter is different and so we cannot create the same infinite loop with a simple argument. However depending on the compiler we may be able to do something else.

To produce the exploitable code here compile with:

```
clang -g -o exploitme exploitme.c
```

Now we can run the code and explore in gdb to find what can be changed. You should be able to see a change to the stack where the value of EBP is stored.

BONUS: Can you find a way to exploit this?

The code in "vulnerable.c" allows you to overflow the buffer much more generously. Here you should be able to manipulate the stack (in particular EBP) to alter the state of the program after the return in a more complex way. Observe that the program prints the length of the argument. You should be able to overflow and overwrite the EBP so that it points somewhere else. Then in your argument that overflows, create a fake stack frame so that the rest of the main function prints an incorrect value (hiding that your argument was long enough to overflow by over-writing the value stored from "strlen").

2.6 Spawning a Shell

NOTE: This is not a buffer overflow. This section introduces you to how to spawn a shell from a "string" that you execute.

One thing we looked at in Tutorial 1 was spawning a shell in a program that was not designed to do this. As we saw, if the right program (with the right permissions) is used, then we can gain root shell access.

Now let's look at how we might create a shell in a program by exploiting our knowledge of the program itself. In "testshell.s" you will find the assembly instructions to spawn a shell under Linux.

ASIDE: If you want or need to build your own machine instructions for creating a shall follow these commends, but it has been made already (for the architecture and OS detailed at the beginning of the Tutorial):

```
as testshell.s -o testshell.o
ld testshell.o -o testshell
objdump -d testshell
```

which will show you the machine instructions. These can then be turned into the string you will find in "example.c".

In "example.c" you will find a simple C program that already has the code for a shell (or shellcode) defined as a string. The rest of the program merely points to this string and executes the "string". We can compile and run the program with

```
gcc -g -fno-stack-protector -z execstack -o example example.c
./example
```

Note that we compile to allow execution of the stack since the program we wish to execute is on the stack as the shellcode. When the program runs we are dropped into a /bin/sh shell that we can operate in. We can exit this shell now we have shown that it works.

Now if we redo the permissions from Tutorial 1:

```
sudo chown root:root example
sudo chmod 4755 example
```

and run the command without root

```
./example
```

we will be in a shell that has root privileges. This demonstrates how to spawn a (root) shell from executing a string stored on the stack

2.7 Overflow and Pwn

Now let's combine our shellcode with a buffer overflow and gain a shell. We can see in the code for "vulnerable.c" and we can observe that there is a copy of the first argument to the code on the stack. If we compile the code with

```
gcc -g -fno-stack-protector -z execstack -o vulnerable vulnerable.c
./vulnerable
```

that we could try and inject the shellcode into via a command line argument.

Open "vulnerable" with gdb and find where the copy takes place and what you would need to add to inject your shell code.

The goal here is to (at least in gdb) be able to create a new shell by using a command line argument to vulnerable.

HINT: You can create a "NOP sled" which is a sequence of non-instructions that you can land it before reaching the beginning of your shellcode. The instruction code for a non-operation is 0x90. For example: "\x90\x90\x90\x90\x90\x90\x90\x90\x90" before the shellcode creates a collection of instructions that the CPU will execute in order so you can jump to any of them and safely continue.

BONUS: Can you do something similar with "tinybuf.c" where the buffer is too small to contain the shellcode?

2.8 Play and Pwn

The goal for the rest of the tutorial is to take your experience with creating errors and combine them with the code to create a shell. Use the shellcode and exploits from 4.7 and 4.8 with the tutorial programs to inject an exploit using other buffer overflows.

Note that you may have to exploit a particular compiler and play with `gdb` a bit to find exactly how and where to insert your shellcode.

HINT: You may have to (re)compile the earlier programs with stack execution allowed to be able to run your injected shellcode.