Techniques of exploiting buffer overflow errors - Part II

In the previous section we have reviewed the memory organization, stack, calling the function to understand thoroughly why when the buffer overflow occurs, we can change the value of the command pointer register% eip, from which can execute any code. We also learned how to create a simple shellcode to insert in the middle of a corrupted program. In this next section, we will look at some basic techniques for exploiting buffer overflow.

Summary:

In the previous section we have reviewed the memory organization, stack, calling the function to understand thoroughly why when the buffer overflow occurs, we can change the value of the command pointer register `%eip` , from which can execute any code. We also learned how to create a simple shellcode to insert in the middle of a corrupted program. In this next section, we will look at some basic techniques for exploiting buffer overflow.

Exploiting buffer overflow error - part 2

1. 1. Root and setuid / setgid programs
2. 2. The program has a buffer overflow
3. 3. Organize shellcode on memory
4. 4. Specify the shellcode address
5. 5. Write a program to exploit buffer overflow
    1. 5.1. Transfer shellcode via buffer
    2. 5.2. Transfer shellcode via environment variable
6. 6. Conclusion
    1. Link

1. Root and setuid / setgid programs

On multi-user operating systems in general and in UNIX in particular, the traditional design allows user `root` (superuser) with supremacy to perform all operations on the system. In addition, there are some operations that require new `root` to be performed, such as changing the password (the file `/etc/passwd` must be updated). In order for the average user to be able to perform these operations, the UNIX system provides a mechanism to establish the actual permissions of the executing process through set-up `setuid()/setgid()` , `seteuid()/setegid()` , `setruid()/setrgid()` . Actual permissions will be automatically set by the system via the suid / sgid attribute bit of the program file. For example, the `passwd` program is suid `root` :

```
-rs - x - x 1 root root 12244 Feb 8 2000 / usr / bin / passwd
```

When the user normally executes the program, the actual right will be the owner's file, which is root. Due to usage requirements, there are often many program files set suid properties on UNIX systems (for owner, group). The following example will further illustrate this:

```
/ * suidsh.c * / void main () {setuid (0); system("/bin/sh"); system ("/ bin /
```

As can be seen, if the suid / sgid program has a security error, the hacker will take advantage of this to control the program to execute any code on the system with higher permissions and even with the highest `root` . That is the purpose of exploiting security vulnerabilities on local (local) machines.

2. The program has a buffer overflow

To illustrate how to organize and insert shellcode into a corrupted program, we will revise the `vuln.c` program for example in part 1:

```
/ * vuln1.c * / int main (int argc, char ** argv) {char buf [500]; if (argc>1)
```

The size of `buf` buffer is 500 bytes. From the previous sections, to exploit the buffer overflow in program `vuln1.c` we only need to override the value of the " *saved instruction pointer"* stored on the stack by the code address. The desired command, here is the start address of shellcode. So we need to sort the shellcode somewhere on the stack memory and determine its start address.

3. Organize shellcode on memory

The problem with organizing shellcode on memory is how the error-exploiting program can determine the start address of the buffer containing the shellcode inside the corrupted program. Normally, we cannot correctly know the address of the buffer in the faulty program (depending on the environment variable, the parameter when executing), so we will determine in an approximate way. This means that we have to organize the shellcode buffer so that when starting at an address it can deviate from the exact address that shellcode still executes without being affected. The command NOP (No OPeration) helps us achieve this. When a NOP command is encountered, the CPU will do nothing but increase the command pointer to the next command.

Thus, we will fill the first part of the buffer with NOP commands, then shellcode. Furthermore, in order not to calculate the exact location of the stored command pointer on the stack, we will only place the shellcode in the middle of the buffer, the rest will contain the full start address values ??of shellcode. Finally, the buffercode containing shellcode will look like this:

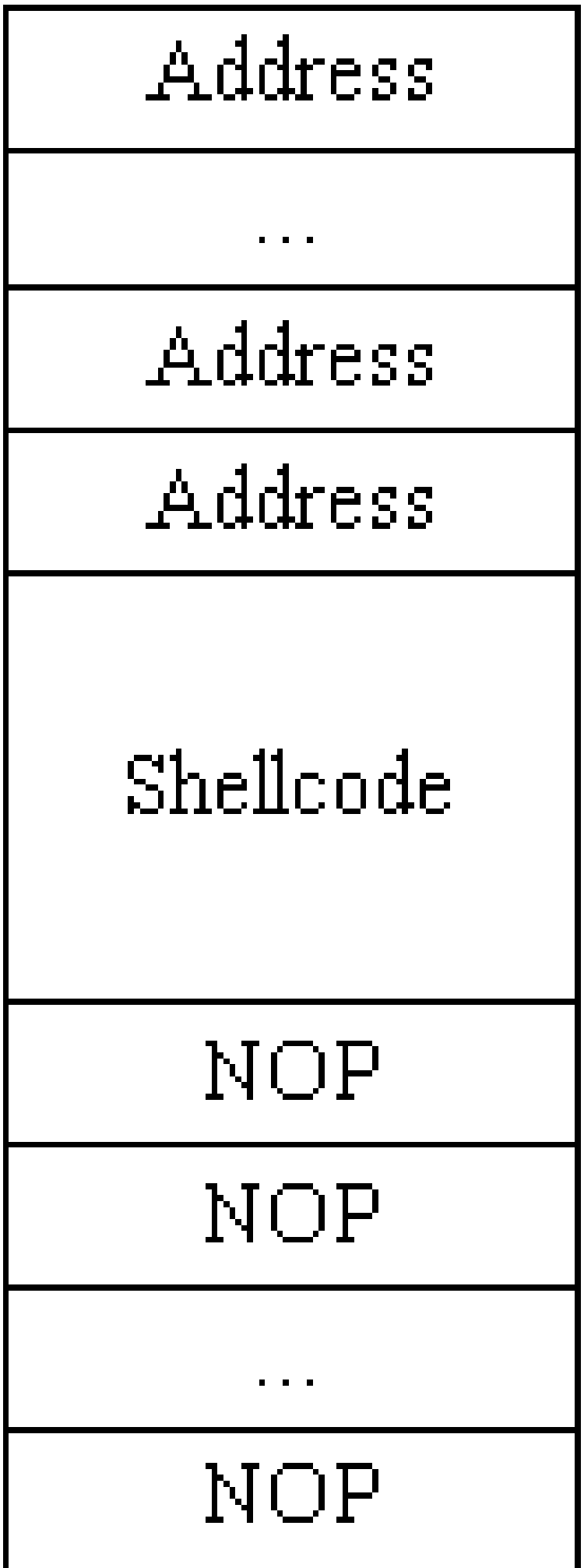
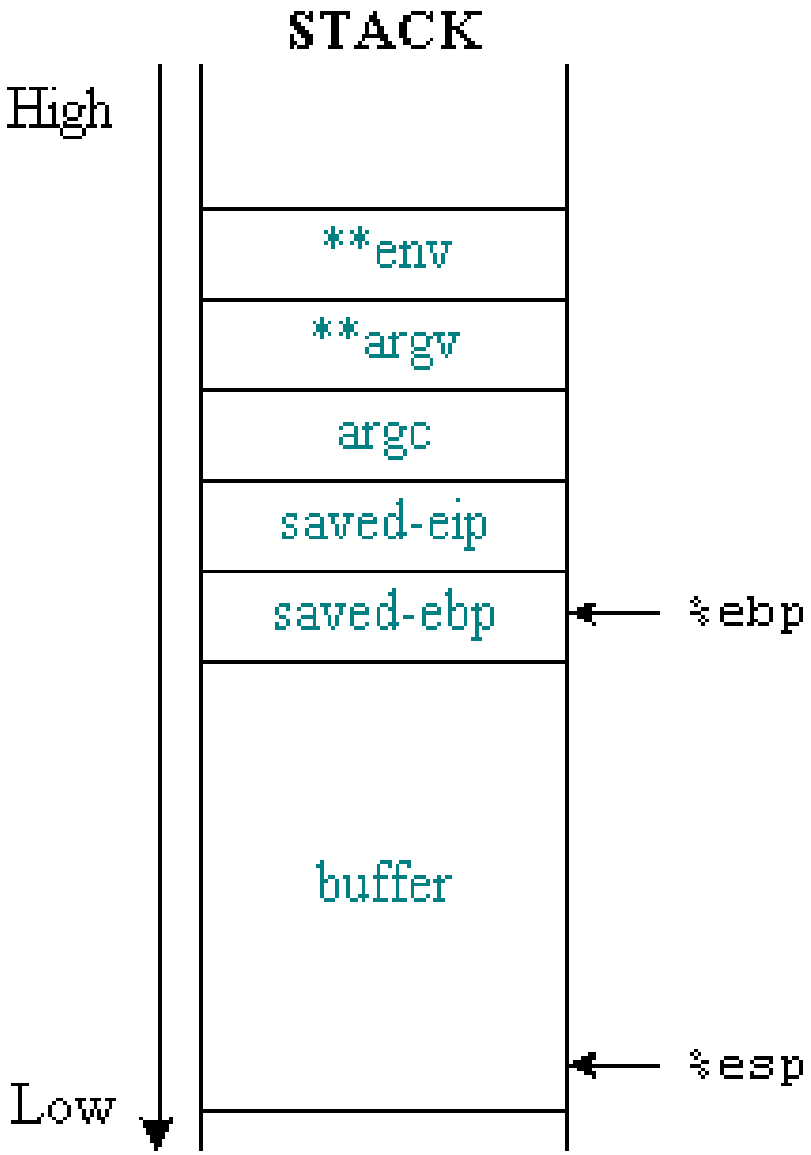

Address
...
Address
Address
Shellcode
NOP
NOP
...
NOP

Figure 1: Organizing shellcode on memory
The following figure shows the status of the stack before and after the buffer overflow occurs.



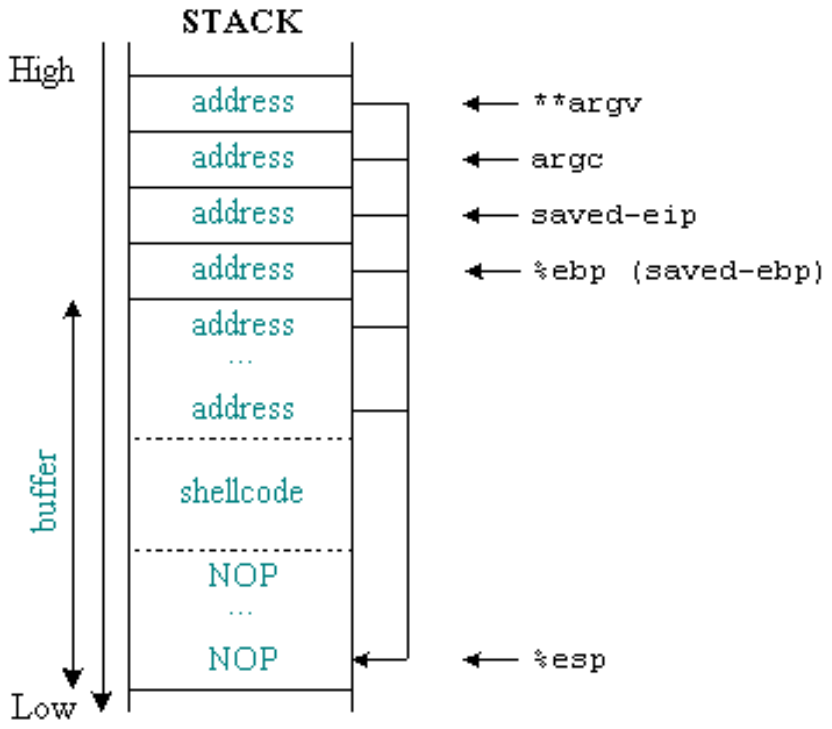


Figure 2: Status stack before and after buffer overflow
There is also a problem here: the alignment on the stack. The address value is 4 bytes (32 bits) long, so when it is placed on the stack, it is not always exactly as expected. In the previous section we already know that stack uses units of words with 4 bytes in length, so the deviation due to incorrect alignment will be 1, 2 or 3 bytes.

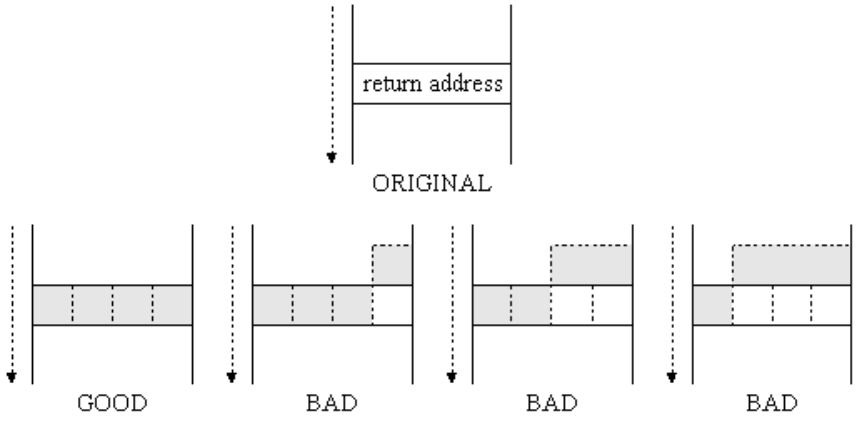


Figure 3: The ability to sort variables on the stack

Only one correct arrangement will work, others will result in a "segmentation violation" or "illegal instruction" error, but we can use the "trial and error" method to find the Arrange properly in memory is not difficult.

4. Specify the shellcode address

The most important issue is how to "anticipate" the start address of the buffer containing the shellcode inside the corrupted program. Thanks to the way to organize shellcode with the above NOPs, this address only needs to be approximate so that it falls between the NOP commands on the shellcode buffer.

A special point is that every program when executing has the same start address stack (note: on virtual address space. For example: this value on Linux is `0xbfffffff` , on FreeBSD is `0xbfbfffff`) and often programs rarely `push` into the stack at a few thousand bytes at a time. Therefore, it is possible to guess the start address of the buffer containing the shellcode on the stack in the faulty program based on the deviation from the current stack top address of the exploit program. This deviation may be negative or positive (see section 1).

The following program will print the value of the SP stack pointer:

```
  / * sp.c * / unsigned long get_sp (void) {__asm ??
__ ("movl% esp,% eax"); } void main() { printf("0x%xn", get_sp()); } void main (
```

The approximate address of the shellcode containing the buffer will be determined by the formula:

```
  SP + (-) OFFSET
```

5. Write a program to exploit buffer overflow

We already know what is needed to exploit the buffer overflow, now need to combine. The basic steps of the buffer overflow technique are: prepare the buffer used to overflow (as in the previous section), determine the return address (RET) and the deviation due to the upcoming variable, determine the address of the buffer containing shellcode, finally calling the program execution with a buffer overflow.

There are a number of ways to organize shellcode on memory and pass the program to an error, we will first look at the most basic method: shellcode is passed through the program's buffer. This method is not the easiest way to exploit buffer overflow on local machines, but this is the most general way to exploit local and remote buffer overflow errors.

In the above example, shellcode will be organized and transmitted via the `buf` buffer of the program `vuln1.c`

5.1. Transfer shellcode via buffer

Our following buffer overflow program will receive 3 parameter values: the program name is corrupted, the buffer size used to overflow and the displacement value compared to the current stack pointer (for example guess the buffercode containing shellcode).

```
  / * exploit1.c * / #include #define DEFAULT_OFFSET 0 #define DEFAULT_BUFFER_SI
??
__ ("movl% esp,% eax"); } void main(int argc, char *argv[]) { char *buff, *ptr;
```

The above program allocates the buffer used to overflow on the heap, why would it be for the reader to respond on their own.

The size of the buffer used to overflow is larger than the buffer overflowed by about 100 bytes. Then the overflow buffer has a fairly large header containing NOPs, the end contains shellcode and the address is enough to overflow and override the return address value (RET).

Try the newly written error extraction program.

```
  [SkZ0 @ gamma bof] $ ./exploit1 ./vuln1 600 Using address: 0xbffffa1c (.) bash
```

Try with displacement value:

```
  [SkZ0 @ gamma bof] $ ./exploit1 ./vuln1 600 100 Using address: 0xbffff9a8 (.)
```

5.2. Transfer shellcode via environment variable

Now, let's go back to the first example, program `vuln.c` (see section 1). It can be seen that the `exploit1.c` program cannot exploit the buffer overflow in `vuln.c` because the overflow buffer size is too small (16 bytes) not enough to fit shellcode. Then the return address will be overwritten by the code instead of the address value to jump to. To overcome this obstacle, we will use another "buffer" to store shellcode. It is usually possible to use environment variables or a program command line parameter (arguments) to contain shellcode because these variables are all on the stack, but using environment variables is a simple and effective method. than. With shellcode contained in the environment variable, the buffer used to overflow simply contains the full (guessed) address value of the environment variable containing shellcode.

The `exploit1.c` program was modified as follows (adding a parameter that is the size of the shellcode buffer).

```
  / * Exploit2.c * / #include #define #define DEFAULT_OFFSET 0 #define DEFAULT_BU
??
__ ("movl% esp,% eax"); } void main(int argc, char *argv[]) { char *buff, *ptr,
```

Try the new exploit program:

```
  [SkZ0 @ gamma bof] $ ./exploit2 ./vulnUsing address: 0xbffffa18 (.) bash $
```

You can see how to use environment variables quite effectively. The following method (**only for Linux x86**) uses the environment variable to contain shellcode but identifies the exact address of this environment variable. Therefore, we do not need to fill NOPs at the beginning of the buffercode containing shellcode, as well as the correctly defined shellcode address instead of having to guess.

The highest part of the address (equivalent to the bottom of the stack) of an ELF, Linux x86 program file is of the form:

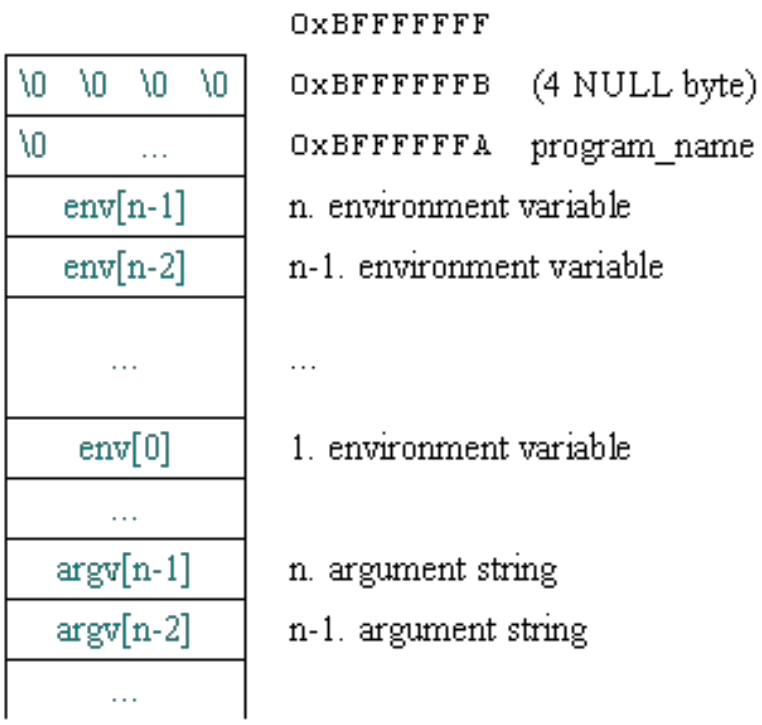


Figure 4: Linux stack architecture at x86

We see, the last environment variable address is calculated by the following formula:

```
envpn = 0xBFFFFFFF - 4 - // 4 NULL bytes strlen (program_name) - // program nar
```

Calling functions that execute programs like `execle` and `execve` allow the passing of an environment variable pointer to the called program. Taking advantage of this we can direct the buffer containing shellcode for the program to crash through the environment variable pointer, and calculate its address correctly.

The formula for shellcode's address:

```
addr = 0xBFFFFFFA - strlen (prog_name) - strlen (shellcode);
```

The new error extraction program is written as follows:

```
/ * exploit3.c * / #include #define DEFAULT_BUFFER_SIZE 512 #define NOP 0x90 /
```

In the above program, we passed the faulty program pointer to the environment only with a single variable that contains the shellcode, so the length of the environment variable is the length of shellcode. Try this new exploit program:

```
[SkZ0 @ gamma bof] $ ./exploit3 ./vulnUsing address: 0xbfffffd4 (.) bash $
```

6. Conclusion

Hopefully, what you have presented can help you understand the causes and consequences of the buffer overflow. The technique of exploiting buffer overflows is absolutely not difficult when there is a very clear

theoretical basis, although it requires a little knowledge of programming languages. Avoiding buffer errors can also be achieved without difficulty, which is to implement the principle of creating safe programs right from the design.

References:

1. Smashing The Stack For Fun And Profit - Aleph1
2. Avoiding security holes when developing an application - Frédéric Raynal, Christophe Blaess, Christophe Grenier
3. BUFFER OVERFLOWS DEMYSTIFIED - Murat Balaban
4. ?ang ghi b? ??m trên s? ph?c d?ng - m?t giao th?c cho b?t ??u - Mixter

Link

1. http://www.phrack.org
2. http://community.core-sdi.com/~juliano/

Previous article: Techniques for exploiting buffer overflows: Organizing memory, stack, calling functions, shellcode