System Security, Solution Buffer Overflow 0

George Cojocar

November 11, 2015

This exercise session will help you to gain understanding of calling stack frames and buffer overflow basics. It will also provide you with practical experience on how to use a debugger (gdb). The successful completion of this exercise will help you during the next exercise session, where you will be required to perform buffer overflow attacks.

The folder `bufov0` inside the VM contains a binary that you should use for this exercise. Furthermore, you might need a tutorial such as `http://sourceware.org/gdb/current/onlinedocs/gdb.html` or `http://www.cs.virginia.edu/~evans/cs216/guides/x86.html`.

The VULNAPP program provides two ways to read user input. The first is by supplying the user input as a program argument in the command line. The second is provided by the program that asks for user input during execution.

In particular, the program is only vulnerable while reading user input as a program argument. The function `cpybuf` handles the copying of the user input in an insecure way.

Stack frames

1. Analyze the stack before (before `CALL` instruction) and after (after `cpybuf` stack frame was set up) calling `cpybuf` using gdb. To analyze the stack during the program flow you must set appropriate breakpoints and run the program. By using the commands `info stack` and `info frame` you can examine the stack and the stack frame respectively. What are the four command outputs? What happens to the stackframes? Explain the meaning of the following outputs: `frame at ...`, `saved eip = ..`, `Saved registers:`, `ebp at ...`, `eip at ....`

   Solution:

```
Before calling cpybuf:

(gdb) info stack
#0  0x08048621 in main ()

(gdb) info frame
Stack level 0, frame at 0xbffff2f0:
 eip = 0x8048621 in main; saved eip = 0xb7e0e497
 Arglist at 0xbffff2e8, args:
 Locals at 0xbffff2e8, Previous frame's sp is 0xbffff2f0
 Saved registers:
  ebp at 0xbffff2e8, eip at 0xbffff2ec
```

`After calling cpybuf:`

```
(gdb) info stack
#0  0x08048541 in cpybuf ()
#1  0x08048626 in main ()

(gdb) info frame
Stack level 0, frame at 0xbffff2dc:
 eip = 0x804854b in cpybuf; saved eip = 0x8048626
 called by frame at 0xbffff2f0
 Arglist at 0xbffff2d4, args:
 Locals at 0xbffff2d4, Previous frame's sp is 0xbffff2dc
 Saved registers:
  ebp at 0xbffff2d4, eip at 0xbffff2d8
```

When the function `cpybuf` is called, a new stack frame is created a the top of the stack.

`frame at ...`: The starting memory address of the stack frame.

`saved eip = ..`: The address of the next frame up (caller of this frame). This is also called the `return address` which is pushed into stack upon `CALL` instruction.

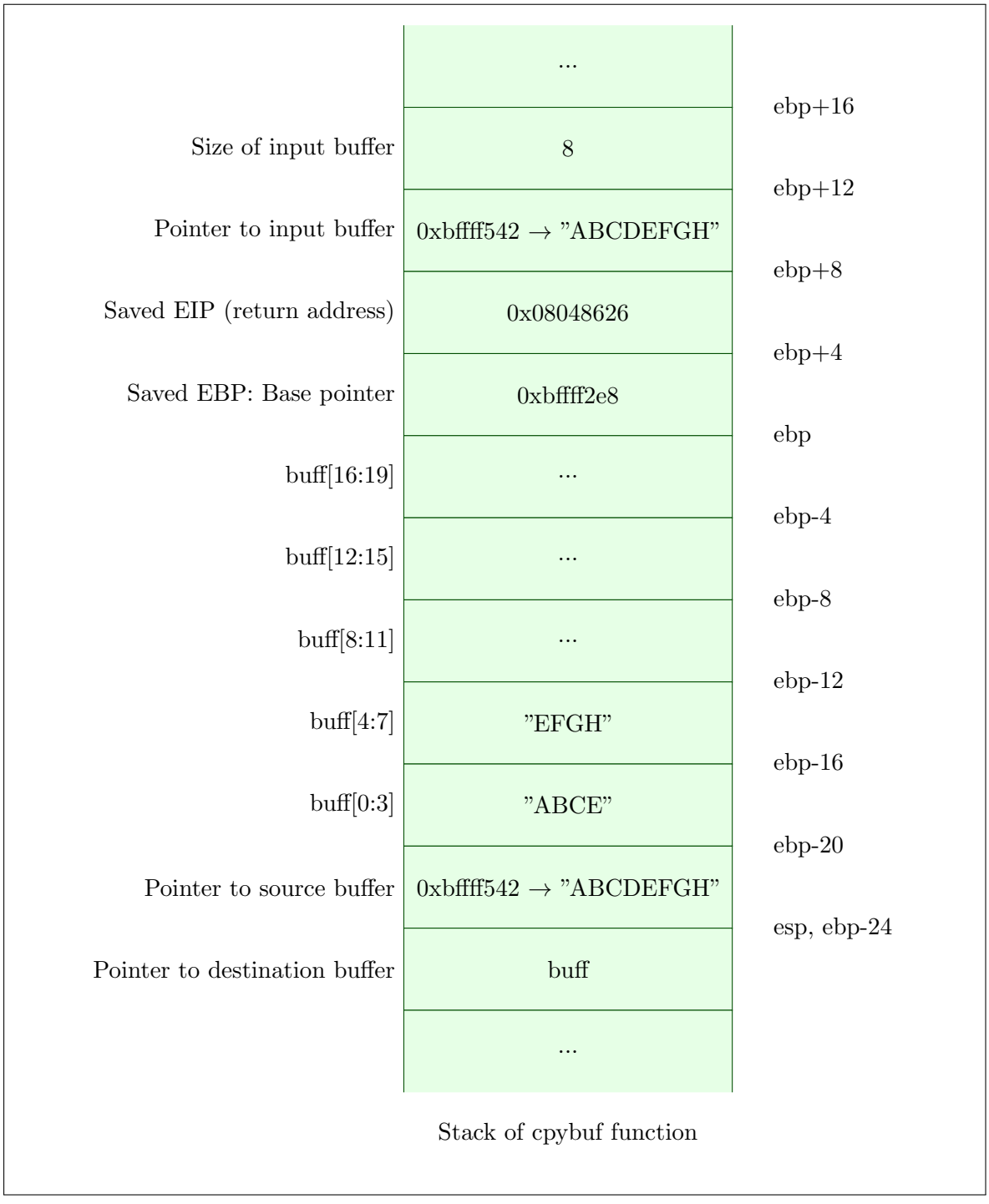
`Saved registers:` It indicates which registers where saved in the frame.

`ebp at ...`: The address where the `ebp` register of the caller function is saved. It represents the *base pointer* which indicates the top of the caller stack and it is used as a reference when accessing local variables and arguments of a function.

`eip at ...`: The address where the `eip` register of the caller function is saved. It represents the *extended instruction pointer* which points to the address of the next instruction of the caller function.

2. Use the information mentioned above and describe the stack frame of `cpybuf` straight after calling `strcpy`, i.e. right after the `CALL strcpy` has finished. Draw a diagram of the contents of the stack frame ([`esp`, `ebp + 11`]) of `cpybuf` for string input: **ABCDEFGH**, i.e. execute the application as `./vulnapp ABCDEFGH`.

   Solution:



3. What is the size of the vulnerable buffer in bytes?

Solution:

The size of vulnerable buffer is 20 bytes.

The buffer is allocated on the stack at line **cpybuf+3**.

```
0x804853e <cpybuf+3>      sub     esp,0x14
...
0x804854b <cpybuf+16>     lea     eax,[ebp-0x14]
0x804854e <cpybuf+19>     push    eax
0x804854f <cpybuf+20>     call    0x80483d0 <strcpy@plt>
```

Stack Buffer Overflow and Stack Overflows

1. How can you exploit a stack buffer overflow to change the program flow or run new code?

   Solution:

   A stack buffer overflow is the result of writing more data into a buffer that was allocated on the stack. One can exploit this vulnerability to change the program flow by overwriting the return address. This address is stored on the stack at the memory location EBP+4. It can be changed to point either at the beginning of the new code (shellcode) that one intends to execute, or anywhere in the memory. Typically the shellcode is written on the stack and then the returned address is changed to point to the stack.

2. What is the difference between a stack buffer overflow and a stack overflow?

   Solution:

   The *stack buffer overflow* refers to the case in which a program writes beyond the end of the memory allocated on the stack for a buffer.

   The *stack overflow* refers particularly to the case when the execution stack grows beyond the memory that is reserved for it. For instance an endless function recursion can lead to such a vulnerability. A function recursively calls itself without termination, it will cause a stack overflow as each function call creates a new stack frame and the stack will eventually consume more memory that is reserved for it.

References

[1] Debugging with gdb, `https://sourceware.org/gdb/current/onlinedocs/gdb.html`

[2] x86 Assembly Guide, `http://www.cs.virginia.edu/~evans/cs216/guides/x86.html`

[3] Stack frame layout on x86-64, `http://eli.thegreenplace.net/2011/09/06/stack-frame-layout-on-x86-64/`

[4] Smashing The Stack For Fun And Profit, `http://insecure.org/stf/smashstack.html`