Lab 2 - Buffer Overflow

Mirco Ghadri & August Holm

- An explanation of **how** your exploit gains root access in detail, including memory layout and screenshots. Graphical image(s) of the memory layout is appreciated, as it would help make the explanation more clear.
- Details on how you created the exploit, including how you found the return address (no bruteforcing), explaining the stack execution and how you managed to ensure that the exploit works without errors.

We created the exploit as a simple python script which printed a payload. The payload consisted of 4 string components: A NOP slide, the shellcode(given as part of lab instructions), a padding in the form of an alphabet, and a return address(repeated 20 times) that pointed back to the stack where the NOP slide was.

In order to get the payload correct so that the return address in the payload overwrote the return address on the stack and so that it pointed back towards the memory address in the stack where the NOP slide was stored, we had to do some exploratory work in GDB.

Finding the return address on the stack and writing over it was the difficult part. We knew that the formatbuffer which we wanted to write past(in order to overwrite the return address on the stack) had a size of 256 bytes.

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

We knew that the NOP slide and the shellcode contributed to the size of the payload. We constructed an alphabet string called padding which consisted of the alphabet with each letter repeated 10 times. We combined the NOP slide string with the shellcode string with the padding string to create our payload. We did not include the return address in the payload since we did not know where exactly to place it yet.

We debugged our program using GDB. We put a breakpoint inside the add_alias() function on the line that called the sprintf() function which wrote user input to the formatbuffer. We put the breakpoint here because it allowed us to inspect the stack and return address before we executed the sprintf() function which would overwrite the return address and data on the stack.

(gdb) break *0x8048564

We also defined a hook-stop using define hook-stop:

> x/1i $eip

> x/100wx $esp

The purpose of the x/1i $eip was to see the current instruction the instruction pointer was pointing to so that we could follow our breakpoint live and see that it works. The purpose of x/100wx $esp was to inspect the values written on the stack. This would allow us to see the

address on the stack where the NOP slide was located. It would also allow us to see how the alphabet had overwritten the stack and which letter in the alphabet overwrote the return address. However, for this we needed another instruction:

(gdb) info frame

The info frame instruction showed the instruction pointer (eip) and the saved instruction pointer(saved eip). The saved eip was precisely the return address and it pointed to the address of the instruction in the main function right below the call to add_alias(), as expected.

We started by running the code after setting the breakpoint and hooks.

(gdb) r $(python exploit.py) 2 3

We could now inspect the stack before the sprintf() function had been called with the input from exploit.py causing the buffer to overflow.

We proceeded to step to the next instruction, which effectively called the sprintf() function

(gdb) ni

After this, we could see the stack, but we also wanted to know what value the return address had been overwritten with. So we ran the command:

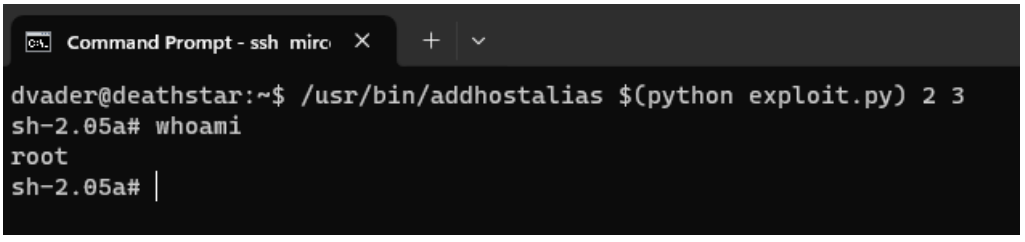
(gdb) info frame

This would show us a hex value which could be decoded into a sequence of ascii characters. The hex value could for example look like 0x52525252 which would be translated to the ascii sequence of characters “RRRR” since the hex value 0x52 represents “R” in ascii. This would let us know that in order to overwrite the return address, we would have to put our desired return address in the payload string where we have the sequence of R’s. So we delete all the letter after R (since their padding is not necessary to reach the return address - we already reach the return address with the padding given by the alphabet upto R). Then we delete the sequence of R’s and for safety measure also the sequence of Q’s (letter before R) in the padding string. And we concatenate a new address string to the end of the payload which contains the return address repeated 20 times. This makes sure that our return address effectively overwrites the return address stored on the stack and that we jump to the desired place. **But what should we put as return address?** The return address should be a memory address on the stack where the NOP slides are stored. Our hook command x/100wx $esp allows us to see where on the stack our NOP slide is stored after we execute the next instruction on the breakpoint.

```
Breakpoint 1, 0x08048564 in add_alias ()
(gdb) ni
0x8048569 <add_alias+41>:       add    $0x20,%esp
0xbffffa04:     0xbffffa3c      0x080486e0      0xbffffcd2      0xbffffe1b
0xbffffa14:     0xbffffe1d      0x04d6dad3      0x00000060      0xbffffb00
0xbffffa24:     0x4000736f      0x00000000      0x400272c1      0x4001432c
0xbffffa34:     0x40007099      0x08048241      0x90909090      0x90909090
0xbffffa44:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffffa54:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffffa64:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffffa74:     0x90909090      0xb9909090      0xffffffff      0x31b0c031
0xbffffa84:     0xc38980cd      0x46b0c031      0xc03180cd      0x80cd32b0
0xbffffa94:     0x31b0c389      0x80cd47b0      0xd231c031      0x2f2f6852
0xbffffaa4:     0x2f686873      0x896e6962      0x895352e3      0xcd0bb0e1
0xbffffab4:     0x40c03180      0x909080cd      0x90909090      0x90909090
0xbffffac4:     0x41419090      0x41414141      0x41414141      0x42424242
0xbffffad4:     0x42424242      0x43434242      0x43434343      0x43434343
0xbffffae4:     0x44444444      0x44444444      0x45454444      0x45454545
0xbffffaf4:     0x45454545      0x46464646      0x46464646      0x47474646
0xbffffb04:     0x47474747      0x47474747      0x48484848      0x48484848
0xbffffb14:     0x49494848      0x49494949      0x49494949      0x4a4a4a4a
0xbffffb24:     0x4a4a4a4a      0x4b4b4a4a      0x4b4b4b4b      0x4b4b4b4b
0xbffffb34:     0xbffffa54      0xbffffa54      0xbffffa54      0xbffffa54
0xbffffb44:     0xbffffa54      0xbffffa54      0xbffffa54      0xbffffa54
0xbffffb54:     0xbffffa54      0xbffffa54      0xbffffa54      0xbffffa54
0xbffffb64:     0xbffffa54      0xbffffa54      0xbffffa54      0xbffffa54
0xbffffb74:     0xbffffa54      0xbffffa54      0xbffffa54      0xbffffa54
0xbffffb84:     0x33093209      0x0804000a      0xbffffbc4      0x400300c0
0x08048569 in add_alias ()
```

We can see that at the stack memory addresses 0xbffffa44, 0xbffffa54 and 0xbffffa64, we have the NOP slide. So our return address could be any of these addresses. We decide for safety measure to select the one in the middle: 0xbffffa54.

So we have now made sure that our padding from the alphabet string(together with padding from NOP slide and Shellcode) is just enough so that when we place our return address at the end of the payload string, it will overwrite the return address on the stack. And we made sure that it our return address points to a memory address on the stack where the NOP slide is stored. Our attack should now work. We can not run the attack in GDB since it requires root priveleges to execute the full program. Instead we run the attack in normal mode(not debugging mode).



And as we can see, the exploit worked!

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

See ***exploit.py***

- Instructions how to make root access persistent. I.e. how can we keep root access even after rebooting. Clearly, rerunning the exploit is not a valid answer.

To make root access persistent, we changed the password of root when we were inside the root shell after the buffer overflow attack. We used the command *"passwd"* which asked for the new password that we wanted to change to:

```
dvader@deathstar:~$ /usr/bin/addhostalias $(python test.py) 2 3
sh-2.05a# passwd
Changing password for root
Enter the new password (minimum of 5, maximum of 127 characters)
Please use a combination of upper and lower case letters and numbers.
New password:
Bad password: too short.
Warning: weak password (enter it again to use it anyway).
New password:
Re-enter new password:
Password changed.
sh-2.05a#
```

There were 2 concerns here. The first concern was that the passwd command did not ask us enter the old password of root, before we could change it to a new password. This was the case when we for example wanted to change the password for dvader using the same *"passwd"* command:

```
dvader@deathstar:~$ passwd
Changing password for dvader
Old password:
Enter the new password (minimum of 5, maximum of 127 characters)
Please use a combination of upper and lower case letters and numbers.
New password:
```

This was a strange and major security flaw, because root is the most important account so if the system asks for the old password when trying to change the password of dvader, it should ask for the old password when trying to change the password of root also.

The other concern was that this method did not seem like the best way to get persistent access to root. The reason was that it would lock out the real root from their account and notify them that they have been hacked. So it was not a stealthy method. A better and more stealthy way would have been to perhaps install a backdoor that did not affect the real root account but gave the hacker root access also. We did not try this option.

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

Ok

- The shellcode is doing a couple of important instructions (see the explanation of the shellcode below) before starting the shell. What is the shellcode exploiting with how addhostalias is configured, why does it execute these instructions, and what would happen if those instructions were not executed? See the "important instructions" below to spot the important parts of the shellcode.

The important instructions mentioned are:

```
...
   1. "\x31\xc0" // makes a zero, to prepare the next instructions for
      setting real user id from effective user id (0).
...
   2. "\x89\xc3" // copies the value to ebx
...
   3. "\xb0\x47" // prepares the next system call for setting real group id
      from effective user id.
...
```

The shellcode exploits the fact that /usr/bin/addhostalias has the setuid bit set and is owned by root.

```
dvader@deathstar:~$ ls -l /usr/bin/addhostalias
-rwsr-xr-x    1 root     root        14196 Aug 27  2013 /usr/bin/addhostalias
```

This means that any user who executes the file will do so with root priveleges. In practice, this means that the users effective user id(EID) will be changed to root when executing the file. However, to fork a root shell, we want to change the users real user id(RUID) to root as well. This is necessary since the shell program runs as a different program, so what our effective user id is inside the addhostalias program does not matter for the shell, only what our real user id is. To change the real user id, this can be done inside the addhostalias program since the users effective user id is root. This is what the first `\x31\xc0` command helps us do.

The third command `\xb0\x47` helps to set the real group id(RGID) to root. Together with the first command, these command make sure that the users RUID and RGID are both changed to root so that they run the shell with full privileges of root.

- Include a comprehensive discussion of countermeasures. Give use cases for each countermeasure and discuss how they can be deployed for the scenario of the lab. The countermeasures should be from different levels, including:
    - Language

Canaries: Canaries are 32-bit values that are inserted between the functions return address and its variables in the stack. In case a of a buffer overflow vulnerability, where a variable of the function is written to with more data than its buffer can hold, this will overwrite the canary before it overwrites the return address. Before the program jumps to the return address, it will check if the canary has been altered. If that is the case, the program will crash and the buffer overflow attack will fail. However, this is not fail-proof. An attacker could bypass the canary and overwrite only the return address without changing the canary. By doing so, they would be able to change the return address without notifying the system so the buffer overflow attack would work.

    - Operating system

NX bit: This is a technology that allows the operating system to make certain parts of memory non-executable. This could be used to make the stack non-executable. By doing so, even if an attacker managed to successfully run a buffer overflow that overwrote the return

address pointing it to a payload on the stack(such as a shellscript), it would be meaningless since the payload on the stack would not be executable. However, this is also not fail-proof. A reason is that an attacker does not need to execute their malicious code on the stack, even though that might me more convenient. An attacker might for example overwrite the return address to point to a function somewhere else in memory. That function might in turn point to another function in memory. By creating a ROP chain(return oriented programming), the attacker could successfully execute a malicious script without executing anything on the stack.

- Run-time

ASLR: Address Space Layout Randomization is a third technique that can be used to defend against buffer overflow attacks. It works by randomizing the memory address of the stack and other components, such as gadgets. By doing this, it becomes harder for the attacker to perform buffer overflow attacks since they don't know where the return address is located on the stack since it changes with each run-time. It also becomes difficult to perform ROP-attacks since the memory address of the gadgets/system functions changes each time.