UNIK4270
Security in Operating Systems and Software

Oblig / Mandatory Exercise

Delivery Time Limit: Wednesday 2nd of November 2016 at 24:00

Delivery

This paper is divided into practical exercises to be done, with related questions that you will need to answer. The exercises are divided into two parts, which can be done independently. Upload the files requested and answers to the questions in the exercises, to the folder **Hand-in** in Fronter.

Tips

Besides reading the book and the lecture slides, do all the exercises given and look into the recommended reading material. Keep in mind that searching the Internet for information can be very helpful. All the resources needed are available at fronter.uio.no in the folder Oblig.

Contact information: trondaso@ifi.uio.no

Good luck!

Part 1: Buffer overflow

Preparation:

- Download the files **blocker** and **shellcode.py** from the Oblig folder.
- Make sure **blocker's** owner is root, that the SUID bit is set and that **blocker** is executable.
  Commands:
  ```
  sudo chown root blocker
  sudo chmod 4755 blocker
  ```
- Turn off ASLR (Address Space Layout Randomization). Remember that you have to turn off this feature every time you restart.
  Command:
  ```
  sudo sysctl -w kernel.randomize_va_space=0
  ```
- (Optional) Download the Linux evaluation copy of IDA from https://www.hex-rays.com/products/ida/support/download_demo.shtml. It might come in handy when debugging.
- http://unik4270.project.ifi.uio.no/overflow.mp4 is a screencast of the example from the memory corruption lecture, and might be a useful reference. Note that for the oblig, it is (a) ok to use fixed stack addresses for running our shellcode, and (b) the buffer we use is large enough to hold our entire shellcode.
- Now, pretend that your user account is not in the /etc/sudoers files, ie that you have no superuser privileges to the system. Open a terminal window, so we can start an attack to gain root access.

You are in a students' terminal room and the login screen in front of you is asking for your user id and authentication, which you submit to gain access to your user mode account. The system authorizes you, and shortly after you are greeted with its familiar desktop background.

Q1) Explain shortly the difference of identification, authentication and authorization

You are disappointed to discover that the system administrators have hardened the security around the boot process so that you cannot give yourself access from Grub. You only have access to the monitor and input devices, the rest of the computer hardware are locked in. You quickly discard the idea of breaking into the computer armed with an USB stick or a screwdriver.

Q2a) Why would physical access to the machine be handy for rooting it?
Q2b) Give an example on what you could do in an attempt to access the locked room by use of social engineering.

After poking around the system a bit, you find an interesting program called **blocker**. It is interesting because it is owned by root, and has the SUID bit set. This means it will be running with root privileges when you invoke it. You want to see if it has any help text available, so you invoke it with the standard help option **(./blocker –h**). Apparently, the program is supposed to read a domain name from a file, **block.txt**. The program then makes an entry in the system's hosts file, **/etc/hosts**, ensuring that every connection from our system to the specified domain name will be blocked. You decide to try the program out by creating the file block.txt in the same directory as the executable, containing just the line **slashdot.org**. Opening your web browser and trying to navigate to slashdot.org confirms that Slashdot has indeed been blocked.

> *Q3) Filling the hosts file with domain names, would make those domains inaccessible for the other users of the system. Would you consider this to be a Confidentiality, Integrity and/or Availability issue, and why/why not?*

You want to check if the program might be vulnerable to a buffer overflow, so you replace the contents of **block.txt** with a bunch of A's. (You can easily fill block.txt with A's by executing the command **`python -c 'print "A" * 100' > block.txt`** in the terminal). The program crashes, and by looking at the system log (**tail /var/log/syslog**), you see that the program tried to execute an instruction at the address 0x41414141. As 0x41 is the hexadecimal representation of the Ascii character 'A', you feel quite optimistic about your chances for exploiting the program.

The help text also mentioned an option **–d**, for running the program in debug mode.
You put just a few A's in **block.txt**, and run the program in debug mode. By updating **block.txt** with enough A's to make it crash, you notice that it calls the function **process_input** and copies a string just before crashing. The debug information even contains the addresses of the source and destination buffers! **(Note that while the buffer addresses do not change between invocations in the same terminal, they might change if you start the program in a debugger or in another terminal.)**

By using the techniques you've learned in class, you find that the copying of the string caused an overwrite of the return address of **process_input**. As debug mode reveals the address of the overflowing buffer, you could use that when overwriting the return address. You already have shellcode for popping a root shell, so now you just have to put in a NOP-sled in front of the shellcode, and some repeated return addresses after the shellcode. You draw yourself a mental image of how the input should look:

NOP-sled	Shellcode	Return addresses

> *Q4a) What is a NOP-sled?*
> *Q4b) Why are repeated return addresses (in combination with the NOP-sled) helpful when you deliver the shellcode?*

You open **shellcode.py** in another terminal, and modify it to write a **block.txt** that contains the right amount of NOPs and the return address you will use. After some trials, you finally get a root shell:

```
#
```

Running `whomai` in the prompt confirms that you have rooted the system.

Q4c) ***blocker*** *was compiled with –fno-stack-protector and linked with –z execstack. This disabled two important security features, what do these do, and would you be able to run the exploit if the features had been enabled?*
Q4d) What does address space layout randomization (randomize_va_space) do, and would you be able to run the exploit if it had been enabled?
Q4e) Do an Internet search and find an example of a program that is or has been vulnerable to a buffer overflow. Take note of interesting data e.g.: the url, the program name and what it does, vulnerability description etc.

Upload your modified, working shellcode.py and the answers to the previous questions.

Part 2. Password cracking

Preparation:

- Download a copy of the shadow and passwd file from the Oblig folder.
- Verify that John the Ripper is already installed on the virtual machine, by typing the command `john` in the terminal.
  If it isn't installed, you can install it by typing the following commands:
  ```
  sudo apt-get update
  sudo apt-get install john
  ```

After having rooted a system, you have access to the password files. There are several accounts listed in this file, and you want to try to crack their passwords. Before you can start with the actual cracking, you have to use the `unshadow` command first. You run unshadow on the **passwd** and **shadow** file, and save the output in the file **workfile**:

```
unshadow passwd shadow > workfile
```

Q5a) Why could it be useful for an attacker to crack passwords on a system he already got root access to?
Q5b) What does the unshadow command do?

You have heard that during a normal run of **john**, it will first enter the single crack mode, thereafter a wordlist attack and finally incremental mode. Since you are not familiar with password cracking, you decide to run the modes one by one so you can inspect how each of them work.

You start `john` in single crack mode on the workfile.

Q6a) What are the names of the persons with the weakest passwords on this system, and what is so bad about these passwords?
Q6b) Where does the information to crack these passwords come from, and what does the single crack mode do?

You now have two passwords, but you really want a few more. You start **john** again, this time using its default wordlist, which is located at **/usr/share/john/password.lst**. Then you continue using the wordlist in rules mode. You play around with pressing a random button when running `john` to see some status information.

Q7a) What is this kind of password attack called (according to Gollmann)?
Q7b) What is the difference between the wordlist with and without the rules?
Q7c) Inspect the Wordlist Mode Rules in the file /etc/john/john.conf

- *Which rule successfully cracked Dole's password?*
- *Which rule cracked Doffen's password?*

Q7d) What is salt and would use of salt prevent this attack? Why/Why not?

Since not all passwords are cracked by now, you figure it is a good idea to start **john** in the incremental mode. You quickly realize that this will take some time…

Q8a) What is this kind of password attack called (according to Gollmann)?
Q8b) Why is the password of Dolly stronger from the john point of view, even when it is only three characters long?
Q8c) If you have one password of the max length of 4 and only consisting of the lowercase from "a"to"z" and numbers from 0-9, how many combinations must you try at the max to crack the password?
Q8d) John can be configured to only search for passwords that satisfies the criteria given in the example above. By using the information from the status of john, calculate the maximum time it would take to crack this password on the system you are on, given a passwd file that only contains this one password.
Q8e) Skrue has a password that satisfies the criteria given above. Name one of the reasons why it takes much longer time to crack this password when running john with the default configuration than what your calculations showed. (hint: check the config file or/and keep pressing a button for status information to see what passwords john tries)
Q8f) Could you think of a rule you could apply to the wordlist attack that would crack Skrues password?
Q8g) If you were to set a password policy for an enterprise, which rules would you enforce if:

- *you want to enforce the strongest password policy possible, and the human factor is not to be considered. How would this policy be?*
- *you want to enforce a password policy takes human behavior into consideration?*
- *Why is the human factor a limitation when it comes to password security?*

The last password will require loads of time, and since you have no intention of sitting in front of this monitor for hours or even days you terminate John with ctrl+c. You log out of the computer and head for the exit.

Q9) Would you consider this attack a violation of Confidentiality, Integrity and/or Availability and why/why not?

Upload the answers to the previous questions.

The end