OT Lab 2 - Software Testing

In this lab, you will learn how to perform a buffer overflow attack. This is a phenomenon that occurs when a computer program writes data outside of a buffer allocated in memory. A buffer overflow can cause a crash or program hang leading to a denial of service (denial of service). Certain types of overflows, such as stack frame overflows, allow an attacker to load and execute arbitrary machine code on behalf of the program and with the rights of the account from which it is executed and thus gain root shell. In this lab, you will try to perform two types of buffer overflow attacks: a "classic" buffer overflow on a Linux machine, a "remote" buffer overflow on a modern version of Windows OS and finally perform a simple cracking of x64 program. On the other hand, you will work with exploiting vulnerabilities at the application and operating system level by implementing various types of attacks, such as RCE, Escalation Privilege, MITM and so on. Here you will also get acquainted with tools such as metasploit.

> Individual assignment with group work parts.
>
> Initial tasks distribution:
>
> - `st` number is even -> Choice 1
> - `st` number is odd -> Choice 2
>
> By the way, if you have some reason to work with opposite Choice, make an agreement with colleague to exchange with topics and fill this table. The main thing here to have 50/50 equal tasks distribution among students group.

Choice 1 - Buffer Overflow (low-level exploitations)

Task 1 - Theory

1. What binary exploitation mitigation techniques do you know?
2. Did NX solve all binary attacks? Why?
3. Why do stack canaris end with 00?
4. What is NOP sled?

Task 2 - Binary attack warming up

We are going to work `buffer overlow` attack as one of the main, most popular and widely-spread binary attack. You are given a binary `warm_up`. Link

Answer for the question and provide explanation details with PoC: "*Why in the* `warm_up` *binary, opposite to the* `binary64` *from Lab1, the value of* **i** *doesn't change even if our input was very long?*"

Task 3 - Linux local buffer overflow attack x86

You are given a very simple C code:

```
#include <stdio.h>
#include <string.h>
int main(int argc, char *argv[]){
char buf[128];
strcpy(buf, argv[1]);
printf("%s\n", buf);
return 0;
}
```

1. Create a new file and put the code above into it:

   ```
   touch source.c
   ```

2. Compile the file with C code in the binary with following parameters in the case if you use x64 system:

   ```
   gcc -o binary -fno-stack-protector -m32 -z execstack source.c
   ```

   Questions:

   - What does mean `-fno-stack-protector` parameter?
   - What does mean `-m32` parameter?
   - What does mean `-z execstack` parameter?

   If you use x64 system, install the following package before to compile the program:

   ```
   sudo apt install gcc-multilib
   ```

3. Disable ASLR before to start the buffer overflow attack:

   ```
   sudo echo 0 | sudo tee /proc/sys/kernel/randomize_va_space
   ```

4. Anyway, you can just download the pre-compiled binary `task 3`.
5. Choose any debugger to disassembly the binary. **GNU debugger (gdb) is the default choice**.
6. Perform the disassembly of the *required* function of the program.
7. Find the name and address of the *target* function. Copy the address of the function that follows (is located below) this function to jump across EIP.
8. Set the breakpoint with the assigned address.
9. Run the program with output which corresponds to the size of the buffer.
10. Examine the stack location and detect the start memory address of the buffer. In other words, this is the point where we break the program and rewrite the EIP.
11. Find the size of the writable memory on the stack. Re-run the same command as in the step #9 but without breakpoints now. Increase the size of output symbols with several bytes that we want to print until we get the overflow. In this way, we will iterate through different addresses in memory, determine the location of the stack and find out where we can "jump" to execute the shell code. Make sure that you get the *segmentation fault* instead the *normal* program completion. In simple words, we perform a kinda of fuzzing.

12. After detecting the size of the writable memory on the previous step, we should figure out the NOP sleds and inject out shell code to fill this memory space. You can find the shell codes examples on the external resources, generate it by yourself (e.g., *msfvenom*). You are also given the pre-prepared 46 bytes long shell code: `"\x31\xc0\xb0\x46\x31\xdb\x31\xc9\xcd\x80\xeb\x16\x5b\x31\xc0\x88\x43\x07\x89\x5b\x08\x89\x43\x0c\xb0\x0b\x8d\x4b\x08\x8d\x53\x0c\xcd\x80\xe8\xe5\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68"`
    The shell code address should lie down after the address of the return function.
13. Basically, we don't know the address of shell code. We can avoid this issue using NOP processor instruction: `0x90` . If the processor encounters a NOP command, it simply proceeds to the next command (on next byte). We can add many NOP sleds and it helps us to execute the shell code regardless of overwriting the return address. Define how many NOP sleds you can write: *Value of the writable memory - Size of the shell code*.
14. Run the program with our exploit composition: `\x90 * the number of NOP sleds + shell code + the memory location` that we want to "jump" to execute our code. To do it, we have to overwrite the IP which prescribe which piece of code will be run next.

    Remark: \x90 is is a NOP instruction in Assembly.
15. Make sure that you get the root shell on your virtual machine.

Bonus. Answer to the question: "It is possible that sometimes with the above binary `shell` is launched under gdb, but if you just run the program, it crashes with `segfault`. Explain why this is happening."

Task 4 - Windows remote buffer overflow attack x86

You can use a Linux virtual machine or your Linux host system (to save compute resources) as the attacker machine. And you might use Windows 10 virtual machine as victim.

1. Download Windows 10 VM. You can use this link to download Win10 VM for Oracle VirtualBox.
2. Connect your victim VM and attacker host to one network using, for example, *bridge adapter* in VirtualBox.
3. Download the software below to your Windows 10 VM:

- Vulnerable Server to make this instance vulnerable;
- Immunity Debugger to investigate into the memory addresses;
- Mona Modules to show the weakness to perform the exploitation;
- Install *Metasploit Framework* on your attacker machine to use some its modules to perform our attack.

    Install Immunity Debugger, put `mona.py` file into the ‘C:\Program Files (x86)\ImmunityInc\Immunity Debugger\PyCommands’ directory, run Vulnerable Server as admin, then run Immunity Debugger as admin and attach the Vulnerable Server process.

    Probably you should turn off the real-time Windows protection and Exploit protection in Windows Defender.

4. Test your connection between attacker and victim PC. For example, you can use `nc -nv` command.
5. In Immunity Debugger run the *Vulnerable server process*.
6. Using any programming language (Python or C are preferred), write the script to implement the *fuzzing*: using **TRUN** command of vulnerable server, you should crash the victim VM

sending a data array via vulnerable channel. Your script should also tell you how many bytes were committed at the time of the crash. Make your script executable and run it. After this make sure that victim VM was crashed (Immunity Debugger is set to pause mode).

7. Immediately after victim crashing stop your fuzzing script (CTRL + C) and note the crashing moment: remember at which bytes EIP starts.
8. Using **pattern_create.rb** from Metasploit Framework, generate the pattern of bytes a little bit more than you received on the previous step. Or you can use the following link.
9. Create/modify the script to overwrote the EIP value. Copy in this script the pattern that you got on the previous step. Also, you no longer need to get information on which bytes the fuzzing has crashed. Re-run Immunity Debugger. Make your script executable and run it.
10. Make sure that that Immunity Debugger again crashed and EIP value was overwrote.
11. Using **pattern_offset.rb** from Metasploit Framework, put your overwrote EIP value from Immunity Debugger and figure out the exact location (offset) of EIP that you can write.
12. Knowing the value of the EIP offset, create a script to overwrite the EIP value. Most likely, 4 bytes should be added to your script in addition to the offset value to overwrite the EIP value. Make your script executable and run it.
13. Make sure that you overwrite the EIP value with some specific characters.
14. To use Mona module, type `!mona modules` in the command line at the bottom of Immunity Debugger. This will let you know what programs there are without memory protection. Look at the resources without memory protections, find the *vulnerable server process* (most likely, *essfunc.dll)*, make sure that it has "*false*" statements.
15. Using **nasm_shell.rb** from Metasploit Framework, type `JMP ESP` into the *nasm* console. This will let you know code for JMP ESP in hex. Using **nasm_shell.rb** may require installing additional libraries.
16. Type `!mona find -s 'the code for JMP ESP in hex' -m essfunc.dll`. Copy the hex address of *essfunc.dll*.
17. Knowing the hex address of *essfunc.dll*, in Immunity Debugger search and find the address of *essfunc.dll*. Press F2 button on this address to create the breakpoint. Then run thevulnerable server process in Immunity Debugger.
18. Modify the script adding the address of *essfunc.dll* in little-endian format. Make your script executable and run it.
19. Make sure that EIP register overwritten in Immunity Debugger.
20. Using **msfvenom** from Metasploit Framework, generate a shellcode. Mark **LHOST** as your attacker machine IP address and **LPORT** to 4444, use `-a x86` parameter to set the architecture to x86.
21. In another terminal window, run the back connect (reverse shell): `nc -nvlp 4444`.
22. Create the final script to exploit the buffer overflow vulnerability. The exploit variable have to consist: `\x90 * pad size + shell code + the memory location + "some fuzzing input" * "EIP offset value"`. Make your script executable and run it.
23. Make sure that you get the shell root on the terminal for backdoor connection.

Bonus: implement another way to do buffer overflow attack on Win32 App & PE binary.

Task 5 - Catch the password x64

1. You are given a binary compiled for 64-bit architecture. Try to exploit buffer overflow vulnerability and find out the right password.

2. Questions:

   - What differences do you find between debugging 32-bit and 64-bit applications?
   - Do we need to use shell code to do this task? Justify your answer.

Bonus - Buffer overflow attack with memory protection enabled

> A challenge for those who want to defeat memory protection via stack canaries leaking and ROP. You need to deploy the docker container on your machine.

Task

An acquaintance of mine, a first-year student, began writing a program to exchange secrets between friends. Since he loves to show off, he opened access to his program on a public port, on his Arch Linux.
Help me teach him a lesson.

Link

Here

Choice 2 - Software Vulnerabilities (high-level exploitations)

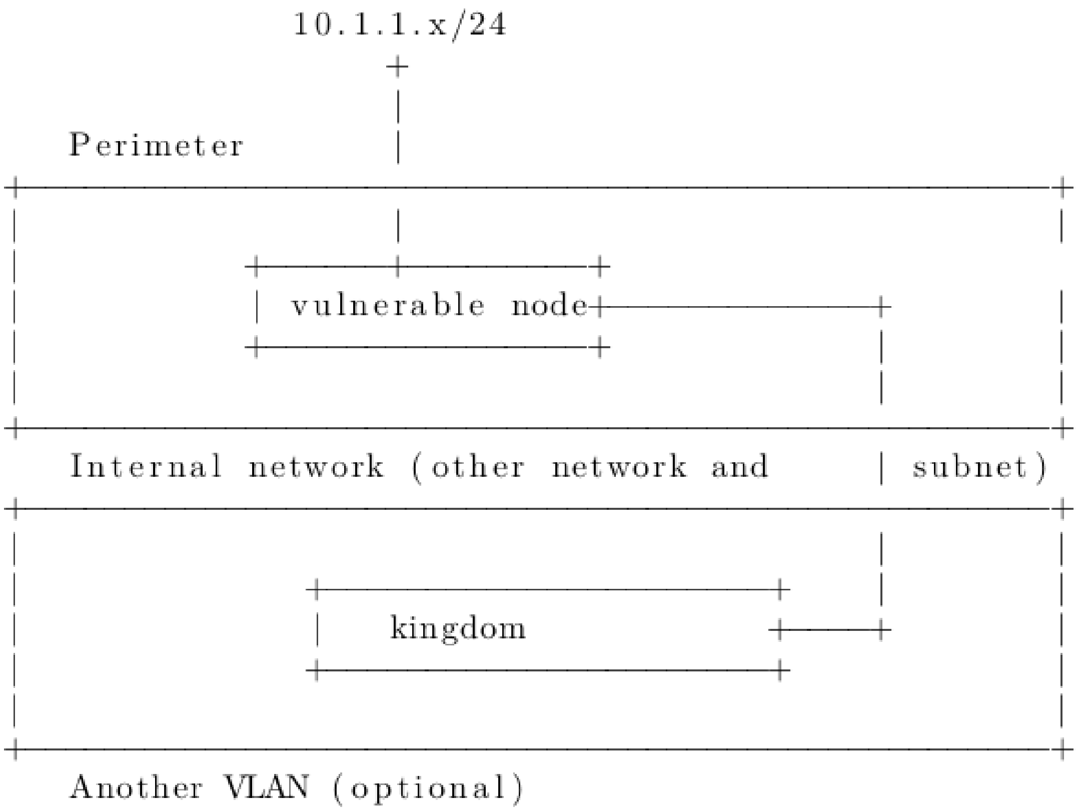
Task 1 - Setup infrastructure for penetration testing

Setup a minimal network architecture containing at least:

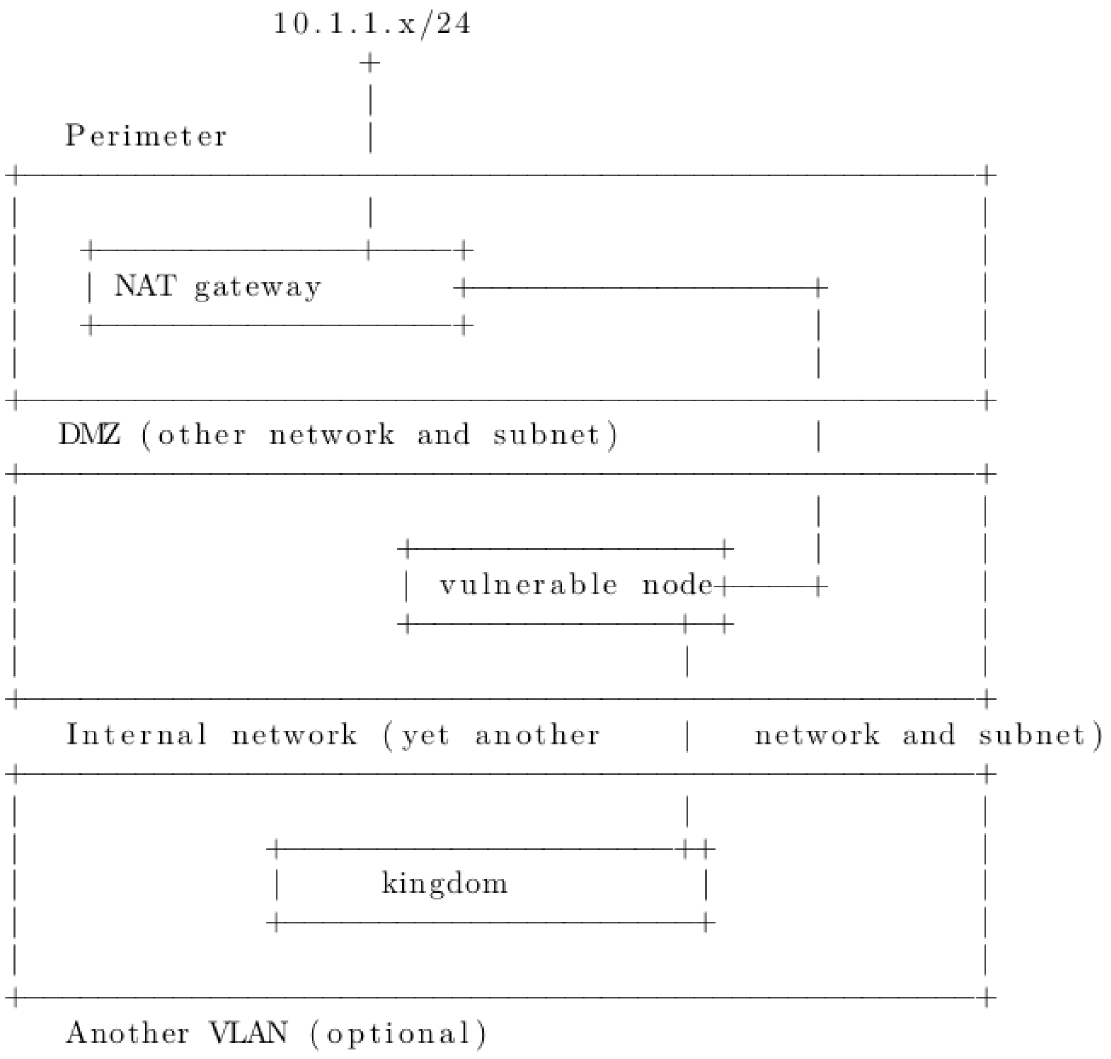
- An *attacker host* (could be your host OS in the goal to save compute resources)
- A *vulnerable node* (ideally with an RCE)
- A node that is not vulnerable (*kingdom*) but that we need to get credentials for (and it's the network link that is vulnerable) or implement any other techniques to steal sensitive data/get unauthorized access
- A isolated network for *kingdom* or in the DMZ (choose your favorite routing solution)

Some proposed network topologies that could be:

- **LAN/Perimeter example with only two members**



- **DMZ example with only two members**


10.1.1.x/24
Perimeter
NAT gateway
DMZ (other network and subnet)
vulnerable node
Internal network (yet another network and subnet)
kingdom
Another VLAN (optional)

This task does not need to be described in your individual report but quickly drawn and describe in your report (it's not an INR/AN lab).

Task 2 - Make an exploitation

Try and validate your chosen software vulnerability to attack a *vulnerable node*.

Example of old good ones working RCE vulnerabilities (you are free to find others and may be newer):

- CVE-2015-1635 (IIS)
- CVE-2017-0144 - EternalBlue
- CVE-2018-1000861 (Jenkins)
- CVE-2019-0708 - Bluekeep
- CVE-2020-7247 (OpenSMTPD)

Explore the environment and networks as much as possible with the given access to *vulnerable node*. To make a PoC, one vulnerability is enough, but you are free to implement more.

Task 3 - Attack a non-vulnerable node (kingdom)

Next, you are ready to proceed to get access on the internal network of the *kingdom*. To implement the attack, try to explore with which hacking tools you are able to implement this (*may be something about SSH MITM?*).

Kingdom-PC can be connected to any different from *vulnerable node* subnet, it doesn't matter. To make a PoC with one working hacking tool/technique is enough.

Task 4 - Privilege escalation flow (in group by two)

In this task you should chose a partner for you who has the same lab choice.

1. Choose privilege escalation scenario and setup a *vulnerable node*. You can use a *vulnerable node* from previous tasks. If your previously chosen vulnerability can provide you a privilege escalation attack, so just use it. Otherwise, define another one vulnerability and make it from scratch.
2. Understand the process of chosen vulnerability and describe how it works.
3. Test and validate it. If you do something with privilege escalation in Task 3, just extend the explanations here then.
4. Deliver your vulnerable instance for your partner to attack.

Note: Despite that attack type could be the same, make sure you have a different vulns!

5.1. After you received the partner's vulnerable node image, find out what kind of vulnerability you were proposed by your opponent teammate. *It is not necessary to do a black box exploration (surely you are already exhausted by this time), so just agree with your partner about the name of the vulnerability or about other tips.*

5.2. Understand the process and describe how it works.

5.3. Hack it.

Task 5 - Magic video

Let's make some fun at the end. You are given the file. Find out what kind of file is what and retrieve the message.

Download the video file

Bonus 1 - More penetration

Demonstrate an additional non-RCE vulnerabilities:

- Network pivoting / route fuzzing
- ARP spoofing
- DNS spoofing
- Pass the hash
- SSL MITM
    - Non-authenticated
    - Private CA
    - CVE-2020-0601: Curveball
- SSH MITM (needs to be persistent and something advanced...)

Bonus 2 - Persistent penetration

Once you compromised a vulnerable node, make sure that you stay there including after a reboot. It can be done using DIY methods (task scheduler & netcat) or using available malware and rootkits. Also establish a covert channel between the victim network and yours.