Introduction to Cyber Security
– System Security –

Deadline: 12th January, 2025

Introduction

Buffer overflow attacks have historically been one of the most common type of attacks used to break into computer systems and to spread malware like viruses, worms or Trojans. The overflow of buffers can affect any server and any operating system. Unfortunately, it proves to be a challenging task to avoid buffer overflows entirely, especially when using memory-unsafe languages such as C or C++. Fortunately, there are many modern countermeasures in place that make buffer-overflows much more difficult to exploit in contemporary operating systems, such as non-executable stack, stack canaries and random memory region offsets. Hence, nowadays, buffer overflow attacks have become rarer, harder to find and more difficult to develop. Nonetheless, a view into the national vulnerability database[1] will reveal that buffer overflows are still a feared vulnerability even to this day, with many tracked CVEs relating to such issues.

In the setting of this laboratory task, you will obtain a basic understanding of the problem of buffer overflows. You will execute your own attack in a simplified laboratory environment like it was the case some yearsago in many production environments. More precisely, you will execute a remote code execution attack based on a buffer overflow vulnerability in a time service so that you can extract the secret flag from the server. To ease your attack, the server is *not* using any of the modern protection measures such as stack randomization, stack protector or a non-executable stack.

[1] https://nvd.nist.gov/vuln/search

Notes

Please note that these practical tasks assume basic knowledge to have been learned in previous studies. This also holds true for topics that will be covered in the lecture or exercise classes of Introduction to Cyber Security at some point, but have not yet been held. If you find yourself missing the knowledge required to solve this task sheet, you must attain it on your own through the process of self study.

For the purposes of this task sheet, you need a good understanding of how program code is run on a computer on a low level (machine code). To this end, you must be familiar with C/C++ programming as well as x86 Assembly (for example, with nasm [4]). To develop your attack, you will need to know about the memory layout of a running process and how it changes throughout its runtime. To learn about buffer overflows, you will find many useful resources online, such as [6] or [1]. Furthermore, you will greatly benefit from familiarizing yourself with low-level debugging tools such as the GNU debugger (GDB [2]). Also, you should know the so called “swiss pocket knife for networks”, `netcat` (see A and [5]). Last but not least, your knowledge about the usage of UNIX/Linux systems, which was of particular importance in lab 2, will also come in handy for this lab.

1. Preparation

1.1. Laboratory Setup

In this laboratory exercise we consider two machines: the attacker called *Chuck*, and a publicly available time service running on the server *Time*. You have access to the computer of *Chuck*. Unfortunately, the secret flag you seek is saved on the server *Time*, which you can not access. That's why you want to find a way to attack time and run arbitrary code (in particular, an interactive shell) that allows you to extract the flag. You can connect to *Time*, e.g., by using netcat, to use the time service provided on port 2222.

The following steps will guide you through the process of connecting to *Chuck*, setting up the environment and validating that the provided time service works properly.

1.2. Connecting to Chuck

The machine used by Chuck can be accessed from *within the BTU network only*[2]. While last time, you connected to the server "neuseeland", this time, you are to connect to the server "mauritius" like so:

```
~$ ssh chuck@mauritius.informatik.tu-cottbus.de -p <your personal port>
```

Once again, you have received the necessary logininfo to log into chuck's account, including your personal port and chuck's password. You might be pleased to find that the port and password are the same as for Lab 2.

Once you successfully logged into Chuck's account, it is time to test out the time service. The host *time* provides a simple time service running on port 2222, which allows you to format string templates based on the current time. For example, you can connect to the service using:

```
        $ nc -u time 2222
```

Now, validate the response of the time service by entering, for example, "Today is %A." in the terminal. If the setup is configured correctly, you should obtain a response of the time service, echoing the string you entered, but with the term "%A" having been replaced by the current day. Note that while using `netcat` to connect to the time service the option `-u` is needed to use UDP instead of TCP. In case your `netcat` session closes after you entered a string, without receiving any echo, that implies that the time service is not running correctly (or you ran netcat with the wrong address / port).

[2] You can either connect directly to *eduroam* on campus or use a VPN connection. If you use Linux as your daily driver, the OpenConnect VPN tool may be of use to you.

1.3. Notes

To avoid data loss, it is recommended to backup the files on the attacker machine *Chuck* regularly. A common way to perform a backup is to use `scp` [7] to copy files from *Chuck* to your local computer.

To work more effectively with multiple terminals, you can have a look at `screen` and `tmux` (see Appendix B).

2. Main Task

The server *Time* provides the service `timeservice` on port 2222. Your task is to **obtain a secret file (`flag.txt`)** which is located at `/home/time/flag.txt` at the service provider. To gain access to this file, you must achieve arbitrary code execution using a buffer-overflow attack. To support you, the C-source-code `timeservice.c` of the service along with the Makefile used to build the service is given. You can find these files both in Moodle as well as on the challenge instance.

To execute the buffer overflow attack on the time server, the **buffer-overflow vulnerability has to be found**, first. To achieve this, you should first work locally on Chuck. You can uncover the vulnerability by analyzing the source code and / or by launching a local instance of the time service and finding the issue through trial and error.

Afterwards, you will have to develop your attack. To this end, you will **create appropriate shellcode** as well as an attack payload that, when sent to the timeservice, leads to the execution of your malicious shell code. In order to test the functioning of your shellcode, you can use the `shellcode_launcher.c` given both through Moodle and on your challenge instance.

After a successful implementation of the attack, **submit on Moodle** the file `secret.txt`, obtained from the server, a `README` describing your steps as well as the exploit source code (i.e. shellcode, executable and tested exploit), i.e. everything needed to understand how you created your attack and why you performed the respective steps.

Hints:

- For the development of the exploit, *Chuck* comes with Python and a C/C++ compiler preinstalled. However, the attack can also be mounted using `netcat` and a binary payload created in a hex editor only.
- To obtain `flag.txt` from *Time*, it is best practice (and the best attack) to establish a remote shell. Once this is achieved, you can execute remote commands on *Time* like viewing the contents of the time server and reading the secret flag. In this lab you are especially interested in a so called *remote bind shell* exploit.
- While writing the shellcode responsible to establish the above mentioned remote shell, you may find it useful to take a look into the options available for `netcat`. Hereby, take special care of the option `-e`, still provided by the `netcat-traditional` package installed on *Chuck* and *Time*.
- In order to exploit the time service, it is important to know that the Linux kernel has implemented the so called *address space layout randomization* since 2005. Due to it, the stack will not start at the top of the virtual memory address space. Instead, it will be shifted

downwards by a random offset. Thus, the address of where one needs to jump to in order to execute arbitrary code will change randomly and be unpredictable. In consequence, the most basic exploits will not work anymore. However, in this laboratory task, we have deactivated the randomization entirely. One should still carefully check which buffers are usable for the attack and which are not.[3]

- To develop and test your exploit, you can compile the provided `timeservice.c` file using the given Makefile and mount the attack locally. Avoid compiling the program manually, because you will receive a different binary if you do not use the same compilation options.
- The GNU debugger `gdb` is a very useful tool to investigate the memory layout of the timeservice and to obtain the return address. GDB automatically disables all address space layout randomization for the binaries it launches. Furthermore, it can be of great help for you to understand what happens once you send your payload to the timeservice, and will help you debug if things do not work as expected. Some additional hints on how to use `gdb` are given in the section D.
- If you want to test your exploit on a locally running instance of the timeservice without launching it through GDB, you must launch the service without address space randomization. You can use the following command to achieve this:

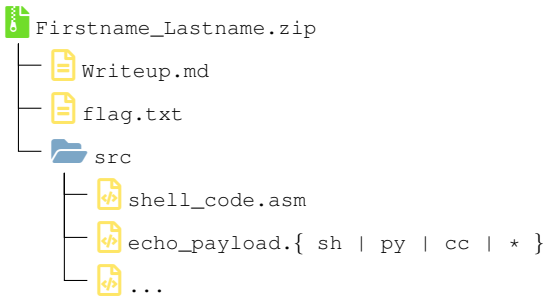
```
~$ setarch x86_64 --addr-no-randomize ./timeservice 127.0.0.1 2222
```

  Do remember to kill the timeservice process once you are done (using `htop` or `pkill`).
- To implement the exploit itself, we strongly recommend using the `nasm` assembler to write the necessary shellcode. This assembler compiles pure machine code if it is called without any additional parameter. Different assemblers often pack the compiled code into container formats compatible with the used operating system, which is unwanted for this attack.

[3] It is even possible to write the exploit in such a way that the attack would run with stack randomization enabled.

3. Submission and Lab Defense

Upload the contents of the secret `flag.txt` file along with a detailed writeup of your approach to the task, including how you analyzed the binary to find the vulnerability and the steps you took to develop the exploit. Include the assembly language source of your shell code `src/shell_code.asm`, a script that allows you to easily transmit the payload to the vulnerable time service `src/echo_payload.{py,sh,c,...}`, as well as any additional scripts you developed as part of your attack (explaining their functioning in your writeup). The zip file containing your submission should have the following structure:



Prepare yourself for a lab defense of up to 30 minutes. In the lab defense, we will go through your solutions and discuss the way in which you solved the tasks. Ensure that you are familiar with all of the concepts that play a role within this lab and are able to defend why you took each step you took.

For this particular defense, be ready for a live demonstration of your attack using the scripts that you uploaded to Moodle. During the defense you should be able to explain your attack, taking special care as to *why* the attack is working. For example, how did you find the correct memory addresses to use for your data and return pointer, and why are you sure they are the same in the real timeservice? How do different ways to prevent buffer overflow attacks work? In addition, you must be able to explain any written/used code/program as well as the provided ones that you have used. The examiners might also ask conceptual questions around the low-level functioning of programs and buffer overflow vulnerabilities.

Appendix

A. Netcat - the Swiss Pocket Knife of Network Administration

Netcat [5] is a command line tool which allows to pipe communication of a TCP or UDP connection to/from stdin/stdout. On the machines *Chuck* and *Time*, `netcat` is preinstalled. The version used is the traditional one still providing the `exec` parameter "-e".

Remember that both machines have an existing `netcat` installation. Think about how this will help you to apply your attack with respect to simplifying your shell code.

B. Parallel Execution of Multiple Command Interpreters Using GNU Screen

`Screen` [10] is a so called “commandline multiplexer”. The Tool allows you to execute multiple (text based) applications, and switch between them without opening multiple SSH sessions. Screen is already installed on *Chuck* and provides a comfortable way to work with multiple command line interpreters. Other advantages are the ability to keep the session open even if one logs out or the connection interrupts (by detaching the session first) as well as the possibility to view a single session in different terminals at the same time.

The command `screen` will start a new `screen` session, a virtual shell. With the keystroke **[ctrl]+[a]** (default), you can change to the command mode of `screen`. With further keystrokes, using the hot keys given below, screen can be interfaced. For example, hit **[ctrl]+[a], [c]** to spawn further virtual consoles. A full list of available hot keys can be found in the manual (`man screen`) or the internal help. The most important functions are listed in table 1 below.

With the keystroke **[ctrl]+[a], [d]** the active screen console will be detached. Thereby screen will fade out but will continue to run in the background. To bring back the session use the command `screen -r`. If there are multiple detached screen sessions you need to provide the PID of the screen session you want to bring forward. Information on the current screen session can be obtained by `screen -ls`. To terminate screen, close all active console windows.

A more modern alternative to `screen` is `tmux`, which can be used as a replacement with only slightly different functioning and features. One of the notable difference is that the default way to

Function	Command
Open help	[ctrl]+[a], [?]
Spawn a new virtual console	[ctrl]+[a], [c]
Switch to the next console	[ctrl]+[a], [n]
Switch to the previous console	[ctrl]+[a], [p]
Rename the current console	[ctrl]+[a], [A]
Get an overview of active consoles	[ctrl]+[a], ["]
Close console	[ctrl]+[a], [K]
Detach current session	[ctrl]+[a], [d]

Table 1: Screen commands

run a command starts with **[ctrl]+[b]** and that the command line arguments differ. If you prefer this variant, you will be pleased to find `tmux` is also already preinstalled on the servers.

C. From Assembly Language to Shellcode

To inject foreign executable code into the memory of a given program it is necessary to provide the instructions in the form of compiled binary code, just as it would be if it had actually been part of the original program. Therefore, we need to generate these binaries from assembly language code. You can either write this code directly, or generate it using another, higher-level tool that you developed yourself. You can obtain working shellcode from an assembly called `program_name.asm` in the following way:

1. Compile using `nasm`: `nasm -f elf32 program_name.asm`
2. Link the object file using `ld`: `ld -m elf_i386 -o program_name program_name.o`
3. Create an *objdump* of the binary using `objdumpa`: `objdump -d program_name`

The generated byte code can now be used within your exploit. In Moodle as well as on your lab instance, you will find an exemplary `exit.asm` program, which you can use to create shellcode that causes the program to exit. The resulting shell code from this example is the exact shellcode that is used in the provided `shellcode_launcher.c` file.

D. Working with the GNU Debugger

After a program is loaded by `gdb <program name>`, the debugger offers many features which can be used for statical analysis of binary code. Table 2 is just a short outline of some useful GNU

debugger commands. A good article on the usage of `gdb` in the context of buffer overflow exploits can be found in [8].

function	command (hot key)
print source code	`l [function name, line number]`
insert break point	`b <line number>`
set condition for break point	`condition <number> <expression>`
print value of variable, address or expression	`p <variable>`
print function in assembler syntax[1]	`disas <function>`
run program[2]	`r <parameter list>`
show register contents	`i r`
inspect memory at address	`x/<format> <address>`
print help to specific command	`h <command>`
switch into a tui layout (c code)	`layout src`
switch into a tui layout (assembly code)	`layout asm`
switch into a tui layout (with registers)	`layout reg`
disable tui	`tui disable`

Table 2: Useful gdb commands

The `p` and `x` commands may become a valuable tool for your exploit development. When working with memory addresses, they also provide the powerful options of formatting the output they provide. For instance, `p/x` will print the value in hexadecimal, while `p/d` prints it in decimal. To display four bytes in hex at a memory address, you can use `x/4bx <address>`. And finally, if you're curious about what the data at an address (for example, the value of the current program counter `$pc`) means when interpreted as 20 assembly instructions, you can run `x/20i $pc`.

Note 1: By default GDB follows the parent process, i.e. the process launched when directly running the program. Thus, it will not consider what happens in any child processes. With `(gdb) set follow-fork-mode child`, the setting is changed such that GDB follows the execution flow of the child process.

Note 2: By default the assembler code is printed in AT&T syntax. To change to Intel notation (as is used by `nasm`), one can change the format within gdb's interactive shell using `(gdb) set disassembly-flavor intel`.

References

[1] *Buffer Overflow - Wikipedia Article*. URL: https://en.wikipedia.org/wiki/Buffer_overflow (cit. on p. 2).

[2] *gdb - The GNU Debugger Project*. URL: http://sourceware.org/gdb/ (cit. on p. 2).

[3] E. A. Hall. *Internet Core Protocols*. O'Reilly & Associates, 2000. ISBN: 978-1565925724.

[4] *nasm - The Netwide Assembler*. URL: http://www.nasm.us/docs.php (cit. on p. 2).

[5] *netcat - The GNU Netcat Project*. URL: http://netcat.sourceforge.net/ (cit. on pp. 2, 19).

[6] A. One. *Smashing the Stack for Fun and Profit*. URL: http://insecure.org/stf/smashstack.html (cit. on p. 2).

[7] *Secure Copy (Manual Page)*. URL: http://www.unix.com/man-page/linux/1/scp/ (cit. on p. 4).

[8] P. Sobolewski. *Overflowing the stack on Linux x86*. Hakin9 magazine. 2004. URL: http://www-scf.usc.edu/~csci530l/downloads/stackoverflow_en.pdf (cit. on p. 20).

[9] *tcpdump (Manual Page)*. URL: http://www.tcpdump.org/tcpdump_man.html.

[10] *Terminal Multiplexer Screen*. URL: https://www.gnu.org/software/screen/ (cit. on p. 19).