67607 – Exercise 1 – Buffer Overflow

Instructed by Amit Klein, written by Agam Ebel,
(+thanks to Oriyan Hermoni for his feedback)

2024

Contents

1 Buffer Overflow **3**

1.1 Our Environment . . . 3
1.2 Intro . . . 3
1.3 General Information . . . 3
1.3.1 Notes about x64 assembly . . . 4
1.3.2 Linux x64 Calling Conventions and the Stack – Reminders . . . 5
1.3.3 Reminder on Sockets and Network Client . . . 6
1.4 Starting the Environment . . . 7
1.5 Tips and Tricks . . . 8
1.6 What to Submit . . . 9
1.6.1 Important Remarks . . . 9
1.6.2 Submission . . . 9
1.7 Resources . . . 10
1.8 A Note about our Server Side Implementation . . . 10

1 Buffer Overflow

In this section, we will introduce the concept of a "Buffer Overflow" attack and outline the steps involved in setting up the environment necessary for executing the attack.

Your goal is to exploit the buffer overflow vulnerability on the server using `execve` function and running a script named `success_script` with the submitters ID (the one who submits the files on Moodle) as a parameter.

1.1 Our Environment

Our server is vulnerable to a buffer overflow and is deployed within a Docker container.
The container is built by copying the server code, installing the necessary packages, and copying the server executable file that contains the vulnerability.
We also mounted a folder named "serverm" to the container, this directory is mapped from the host to the container and can be accessible via the following path `/home/cbof/Desktop/serverm` on the host. It is mounted to the following path in the container: `/tmp/serverm`.

All the files we upload to this directory are stored on the host (we can also access them from it) and can be accessible from inside the container without the need to run or build the container again.

The attacker will operate from another docker container and connect to the server via hostname *`server-IP-address`* and port `12345`.

1.2 Intro

The essence of a Buffer overflow attack is overwriting memory regions of the process with "unexpected" data. This can lead to undefined behavior of the process. A careful analysis of the process and crafting of the "unexpected" data can lead to e.g. forcing the process to execute machine instructions inside the data.
This kind of vulnerability (the ability of a user/attacker to "unexpectedly" modify memory regions of the process) can be exploited for example when buffers of type `char[]` (C strings) are used.

There are several types of Buffer Overflow attacks. We will focus on Stack-based buffer overflow. In this type, we overwrite the data on the stack, including a return address located within.

More information on the different types can be found here

1.3 General Information

- To log in to the environment, use username "cbof" and password 5260

- The server runs on a docker container and is connected to a virtual LAN named bofN, it is accessible via port `12345` and hostname *`server-IP-address`*.
- The server is vulnerable to a stack-based buffer overflow attack. This can be exploited by sending a payload of over 10 bytes to the server.
- We provide you the source code of the server. It can be found on the VM's desktop and named `bof_server.c`
- The object of the exercise is to exploit the server process with a buffer overflow attack and force it to run the file `/tmp/success_script` (which we have placed on the server container, of course).
- Insert server IP address, the stack address, and the offset from the return address to your code as command line arguments, so when we run the compiled code, the use will be like this:
  `./attacker` *`server-ip-address address-of-x x-offset-from-return-address`*
- When compiling your solution attempt, use `gcc -Wall -Wextra -Werror` *`program-name`*`.c -o` *`desired-compiled-file-name`*
  Compile your source code from within the client container, and make sure there are no compilation warnings!

1.3.1 Notes about x64 assembly

In this attack, we send a client request (that the server receives) which exploits a buffer overflow to hijack the return address from a legitimate function execution, eventually running the `execve` system call. We use assembly to generate the machine code that writes parameters to registers and invokes system calls.
There are 2 syntax conventions in Assembly, the Intel style and AT&T style. Herein, we use the latter.

In the x64 architecture, we have 16 registers that we can use for manipulating data and memory (there are many others registers as well, but those are mostly what we will use in this attack). Starting from 4 general registers: `RAX, RBX, RCX, RDX`, followed by 4 special registers `RSI, RDI, RBP, RSP` (`RBP` is the stack frame pointer and `RSP` is the stack pointer), and ending with 8 more general purpose registers (denoted by `R8` to `R15`).

We have two formats to represent data; those ways are called endianness. We have "big endian" and "little endian". The native x64 processor convention is "little endian".
In this method, the bytes are ordered from the least significant byte first and at the end the most significant byte. In contrast, TCP/IP uses "big endian" (also referred sometimes as network wire format) to represent data on the wire. In big-endian format, the lowest byte

address holds the start of the address (or the most significant byte of an integer) and the highest byte address holds the rightmost byte (or the least significant byte of an integer).

For example, in x64 memory, the 64-bit integer `0x1122334455667788ull` is stored as a sequence of 8 bytes thus: `{0x88, 0x77, 0x66, 0x55, 0x44, 0x33, 0x22, 0x11}`.

1.3.2 Linux x64 Calling Conventions and the Stack – Reminders

To pass function arguments (like pointers or integers) to the callee function, we use `RDI, RSI, RDX, RCX, R8, and R9` registers, in this order, for passing the 6 first arguments. If we have 7 or more arguments, the remaining arguments are pushed onto the stack immediately before the `CALL`.

The caller must save the values of `RAX, RCX, RDX, RSI, RDI, R8, R9, R10, R11` if needed, as the callee is not guaranteed to preserve them. The return value from the callee is provided in the `RAX` register (or `RAX` and `RDX` for functions returning two values). The callee function needs to preserve `RBX, RBP, R12, R13, R14, R15` and `RSP`. For example, the callee may write these registers on the stack if it uses them and restore the original values before returning.

Finally, we will do a short recap on system calls. A system call is an entry point into the Linux kernel. Those calls are generally not invoked directly, but rather via wrapper functions. For more information, see the Linux system call list here.
While they're usually invoked via wrappers, we can still invoke a system call directly (usually) by copying arguments and the system call number to the registers, trapping to a kernel mode, and checking whether the system call returns an error number when returning into user mode.

For example, here is a code we wrote to invoke the system call `sys_openat` directly (without going through its wrapper, `openat()`):

```
movq $257, %rax # Syscall number for openat
movq directory(%rip), %rdi # Load directory fd value into %rdi
leaq filename(%rip), %rsi # Pointer to the file name relative to %rip
movq $0, %rdx # Flags (e.g., O_RDONLY)
movq $0, %rcx # Mode, set to 0 when not creating a file
syscall # Make the syscall
```

Its assembled machine code looks like this:

```
0: 48 c7 c0 01 01 00 00 mov $0x101, %rax
7: 48 8b 3d 00 00 00 00 mov directory(%rip), %rdi
```

```
e: 48 8b 35 00 00 00 00 lea filename(%rip), %rsi
15: 48 c7 c2 00 00 00 00 mov $0x0, %rdx
1c: 48 c7 c1 00 00 00 00 mov $0x0, %rcx
23: 0f 05 syscall
```

We can also see here the different stages of calling a system call manually. We first copy the arguments in the order prescribed by the API of the system call function we invoke (in this case `sys_openat` – the API is documented in the `openat()` wrapper) to the registers and then we invoke the `syscall` machine instructions (at the last row of the disassembly).

Notice that the zeros at the end of each assembled machine code are placeholders in this case for the real variables (such as directory, filename...) at the memory. When we assembled this code, we used 0 for directory and filename offsets (for simplicity), which are translated to "0" in the 32-bit offset in the assembled code.

Make sure to plug correct offsets (relative to rip), or use machine instructions that have absolute addresses, and plug the absolute address to them. Please pay attention to this note in case the code looks right although it is not working as expected.

1.3.3 Reminder on Sockets and Network Client

To execute our attack, after creating the payload, we first need to establish a TCP connection to the server. We do that by creating a socket for connecting to the server. But even before that, need to know the server's IP address and to which port to connect.

We start by creating the socket. As the server uses TCP for communication, the socket we create is of type `SOCK_STREAM`. Recall that we can open a socket in the following way: `sockfd = socket(AF_INET, SOCK_STREAM, 0)`.
Now, we need to designate the destination's IP and port. We do that by creating a `struct` of type `sockaddr_in` and setting the values of the `sin_family, sin_addr, sin_port` fields.
When we assign a value to the 16-bit server port field, we need to use the `htons()` function to convert the 16-bit integer (`short`) from the host byte order ("little endian", in the x64 case) to "big endian" as required by TCP/IP.

Now, we can connect the client socket to the server socket using `int connect(int sockfd, const struct sockaddr *addr, socklen_t addrlen)`. We can then send the data to the server using `send(int sockfd, const void *buf, size_t len, int flags)`.

1.4 Starting the Environment

Our attack environment consists of a VM and 2 containers. First, download the environment at the following link.
Note – the environment disk requires at least 30GB free on your computer, make sure you to have enough free space to upload the disk.
Start the virtual machine using the instructions for uploading a virtual disk to a VM which we provided in the "Environment Setup" file. Next, run the server image to upload our vulnerable server. We can do it by calling "docker start" and running the executable file of the server.

- `docker start sbof-new3`
- `docker exec -it sbof-new3 /bin/bash`
- `./sbof` (from inside the container – meaning in the terminal that was opened by using the previous command)

Note: when trying to run your solution of the attack, first – upload the file to the "clientm" directory (it can be found in the following path `/home/cbof/Desktop/`) that mounted to the client container. Next, run the following commands:

1. `docker start cbof-new2`
2. `docker exec -it cbof-new2 /bin/bash`
3. navigate to "clientm" directory (from inside the container – meaning in the `bash` shell terminal that was opened by using the previous command)
4. compile your code using `gcc -Wall -Wextra -Werror` *`file-name`*`.c -o attacker`
5. run your attack attempt using `./attacker`

When try to run another attack attempt, do not forget to first run `./sbof` again inside the server container! (as the server terminates once it gets the data from the client or crashes in case fo unsuccessful attempt).
You can work on your source code file from within the VM (you should place you files under the directory `/home/cbof/Desktop/clientm` on the VM), and you'll be able to access it in the container under `/tmp/clientm` (which is mounted to the VM's `/home/cbof/Desktop/clientm`).

Therefore, it is enough to change the source code itself inside the "clientm" directory on the VM and the changes will also applied in the container. You must compile the source code from inside the container (with the following command: `gcc -Wall -Wextra -Werror` *`attack-attempt-file.c`* `-o attacker`) and ensure that there are no compilation warnings, and run the executable from the container. This is how we will compile and run your code ourselves (in our container environment which is practically identical to yours, so this guarantees you're working with the same compilation and library environment we use), and

we will take off points for compilation warnings (and lots of points if it fails to compile or do the job).

Note – compiling your code inside the container ensures that it will compile, link, and run in an environment similar to ours, increasing the likelihood that it will work seamlessly in our setup as well.

1.5 Tips and Tricks

- The server intentionally discloses the stack address (and the offset from the return address). You'll need to copy it from the server's terminal and use it in your code to execute the attack. When using these parameters (and the server IP address), please insert them to your code as command line arguments, so when we run the compiled code, the use will be like this:
  `./attacker` *`server-ip-address address-of-x x-offset-from-return-address`*
  (and not like the server tells you to run! The example there is when we assume the IP is hard-wired and shows the order of use in the parameters the server discloses).
  Note: This implementation is intentionally simplified to meet the exercise's objectives and should not be used as a reference for real-world applications, see also subsection 1.8.

- Check the system call table to get the right opcode for the system call we want to run. We can use this table for finding the right code for the system call function.
  To see the different system calls, please use Linux documentation of syscalls here

- We recommend you first "design" what the stack should look like. Look at the `execve()` documentation, find what parameters you need to insert and where they should be placed in the stack. Next, think which assembly instruction you will need to load the related data to the registers and invoke `sys_execve`, at end – assemble the commands and data to the payload, and send to the server.
  It can be helpful to draw a diagram of how the memory looks and use it to understand how to insert the parameters inside the stack.

- You can use godbolt online compiler for translating C code into assembly. This can be useful to check if the parameters which are inserted to the registers are as expected and for checking your payload.

- To find the server IP address, use `docker inspect bofN` and search for the IPv4 address of "sbof-new3" (which is our server).

- When compiling your solution attempt, use `gcc -Wall -Wextra -Werror` *`program-name`*`.c -o` *`desired-compiled-file-name`*
  Compile your source code from within the client container, and make sure there are no compilation warnings! We will take points off for any compilation warnings.

- Test your code! Try to run your attack on the provided server by sending the payload and check what is the output on the server container. Make sure this is consistent by running several attack cycles.
- The GDB ("the GNU debugger") disables ASLR by default. When you run the vulnerable software using GDB, the addresses of objects (including the stack) and functions will be identical in different runs of the same program.
  Notice that the vulnerable program runs with ASLR (by default). Therefore, do not rely on the addresses observed in GDB to determine whether addresses are fixed or to deduce the exact locations of functions and objects during the normal execution.
- Notice that we also have space to insert "our payload" above the word that includes the `return address` (meaning, we also have space in the stack frame of `main()` – you can see a hint for that in the `bof_server.c` code we provided).
- When you build your payload for the attack, try to avoid using functions like `strcpy` as they stop copying the string when the first instance of "0" or "NULL" appears (i.e. if the address you are copying starts with 0's, not all the data will be copied).
- Please note that we have installed Wireshark on the VM, which could be useful for checking if the connection to the server is successful, particularly when debugging your code. Of course, its use is entirely optional.

1.6 What to Submit

1.6.1 Important Remarks

1. Insert server IP address, the stack address, and the offset from the return address to your code as command line arguments, so when we run the compiled code, the use will be like this:
   `./attacker` *`server-ip-address address-of-x x-offset-from-return-address`*
2. When compiling your solution attempt, use `gcc -Wall -Wextra -Werror` *`program-name`*`.c -o` *`desired-compiled-file-name`*
   Compile your source code from within the client container, and make sure there are no compilation warnings!

1.6.2 Submission

The submission is done via Moodle in pairs. Please submit a zip file named "ex1.zip" which contains the following files:

1. Your attacker code (in `C`). The file should be named "ex1.c".
2. A readme.txt file that contains a single line with the submitters IDs (separated by a comma. i.e. 123456789,987654321).

3. A file containing a short explanation of your attack attempt, what assembly commands you used, in which order, why and how you used any of the commands.
   The file should be named "explanation.txt".

1.7 Resources

- `https://github.com/mschwartz/assembly-tutorial?tab=readme-ov-file#introduction`
- `https://medium.com/@0x-Singularity/exploit-tutorial-understanding-buffer-overflows-d017108edc85`
- `https://owasp.org/www-community/attacks/Buffer_overflow_attack`
- System calls and system calls table:
  `https://man7.org/linux/man-pages/man2/syscalls.2.html`
  `https://chromium.googlesource.com/chromiumos/docs/+/master/constants/syscalls.md`
- `https://defuse.ca/online-x86-assembler.htm`
- `https://www.man7.org/linux/man-pages/man2/intro.2.html`
- `https://godbolt.org/`
- `https://wiki.osdev.org/System_V_ABI`

1.8 A Note about our Server Side Implementation

Buffer overflow attacks are a relic from the 80s-90s. Nowadays, CPUs, operating systems and compilers employ multiple security mechanisms that defend against buffer overflows, and writing seemingly insecure C code (such as the vulnerable server in this exercise) does not necessarily guarantee that the software can be exploited. For this exercise, therefore, we had to turn off or actively bypass several such defense mechanisms. Specifically, for the server software (sbof):

- We turned off the GCC stack canary defense mechanism, by deliberately compiling sbof with the flag `-fno-stack-protector`
- We deliberately (in our server side code) made the stack pages executable, using the Linux `mprotect()` API.
- We deliberately disclose/leak the stack address by printing it to the screen (at the server side), de-facto annulling the ASLR protection.

Clearly these actions are completely unacceptable in any production code, and were carried out with the sole intention of making our server code vulnerable (like in the 90's...).