COS430/530 – Project Deliverable 2 – Buffer Overflow Project – Due Date: March 10th, 2024

Note: The materials presented in this project are inspired by and partially taken from SEED labs, with permission from the author, Dr. Du.

Description:

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 project is to gain practical insights into this type of vulnerability and learn how to exploit the vulnerability in attacks. You will be given a program with a buffer-overflow vulnerability and you are expected to develop a scheme to exploit the vulnerability and finally gain the root privilege. You will also learn how to protect the system and counter against buffer-overflow attacks.

This project covers the following topics:

- Buffer overflow vulnerability and attack
- Stack layout
- Address randomization, non-executable stack, and StackGuard
- Shellcode (32-bit and 64-bit)

Lab Environment:

This project has been tested on the pre-built Ubuntu 20.04 VM, which can be downloaded from the SEED website. However, most of the SEED labs can be conducted on the cloud, and you can follow our instructions to create a SEED VM on the cloud.

The Docker manual can be found here.

More details are found in the "`Project_2_2024.zip`" folder.

Environment Setup:

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 use address space randomization to randomize the starting address 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
```

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. 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.

Task 0 – 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.

1.1 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, you cannot just compile this code and use the binary code as our shellcode. The best way to write a shellcode is to use assembly code. In this project, we only provide the binary version of a shellcode, without explaining how it works (it is non-trivial).

1.2 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`.

- 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`".

1.3 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.

```
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 1 – Invoking the Shellcode (10 Points):

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, you will test the shellcode.

Listing 1: `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 you 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` in the `shellcode` folder, 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.

Note that: The compilation uses the `execstack` option, which allows code to be executed from the stack; without this option, the program will fail.

Submission – Report:

- **(10 Points)** You need to provide screenshots to demonstrate your investigation and explain your observation in detail.

Task 2 – Understanding the Vulnerable Program (10 Points):

The vulnerable program used in this project 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 `Project_2_2024.zip` 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.
 * 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, your goal 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, you need to make the program a root-owned `Set-UID` program. You can achieve this by first changing the ownership of the program to root (Line 1) and then changing 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=115 -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.

Submission – Report:

- **(10 Points)** You need to provide screenshots to demonstrate your investigation and the vulnerabilities and explain your observation and vulnerabilities in detail.

Task 3 – Launching Attack on 32-bit Program (Level 1) (20 Points):

3.1 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. In this task, you will use a debugging method to find it out. Since you have the source code of the target program, you can compile it with the debugging flag turned on. That will make it more convenient to debug.

You then need to 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. You need to use `gdb` to debug `stack-L1-dbg`. You 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 breakpoint 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 you print out the value of `ebp` here, you will get the caller's `ebp` value. You 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 that data, so the actual frame pointer value will be larger. You should keep this in mind when constructing your payload.

3.1 Launching Attacks

To exploit the buffer overflow vulnerability in the target program, we need to prepare a payload and save it inside `badfile`. You will use a Python program to do that.

A skeleton program called `exploit.py`, is provided in the `Project_2_2024.zip` file. The code is incomplete, and you need to replace some of the essential values in the code.

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!
```

Submission – Report:

- **(10 Points)** You need to provide screenshots to demonstrate your investigation and attack.
- **(10 Points)** You also need to explain how the values used in your `exploit.py` are decided.
    - These values are the most important part of the attack, so a detailed explanation is required to get a full grade.
    - Only demonstrating a successful attack without explaining why the attack works will not receive many points.

Task 4 – Launching Attack without Knowing Buffer Size (Level 2) (10 Points):

In the Level-1 attack, using `gdb`, you 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, you will not be able to get a copy of the binary or source code.

In this task, you 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. It is assumed that you know the range of the buffer size, which is from 100 to 200 bytes. Another fact that may be useful is that, due to the memory alignment, the value stored in the frame pointer is always multiple of four (for 32-bit programs).

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.

Submission – Report:

- **(10 Points)** In your report, besides the screenshots and description of your observations, you need to describe your method and provide evidence.

Task 5 – Launching Attack on 64-bit Program (Level 3) (10 Points):

In this task, you will compile the vulnerable program into a 64-bit binary called `stack-L3`. You will launch attacks on this program.

The compilation and setup commands are already included in `Makefile`. Similar to the previous task, a detailed explanation of your attack needs to be provided in the project report.

Using `gdb` to conduct an investigation on 64-bit programs is the same as that on 32-bit programs. The only difference is the name of the register for the frame pointer. In the x86 architecture, the frame pointer is `ebp`, while in the x64 architecture, it is `rbp`.

Challenges. Compared to buffer overflow attacks on 32-bit machines, attacks on 64-bit machines are more difficult. The most difficult part is the address. Although the x64 architecture supports 64-bit address space, only the address from `0x00` through `0x00007FFFFFFFFFFF` is allowed. That means for every address (8 bytes), the highest two bytes are always zeros. This causes a problem.

In your buffer overflow attacks, you need to store at least one address in the payload, and the payload will be copied into the stack via `strcpy()`. You know that the `strcpy()` function will stop copying when it sees a zero. Therefore, if zero appears in the middle of the payload, the content after the zero cannot be copied into the stack. How to solve this problem is the most difficult challenge in this attack.

Submission – Report:

- **(10 Points)** In your report, besides the screenshots and description of your observations, you need to describe your method and provide evidence.

Task 6 – Defeating `dash`'s Countermeasures (15 Points):

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. 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, you need 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 you invoke the shell program, you need to change the real UID to zero. You 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 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 or without the `setuid(0)` system call. Describe and explain your observations.

Launching the attack again. Now, using the updated shellcode, you need to 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, run the following command to prove that the countermeasure is turned on.

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

Submission – Report:

- **(5 Points)** In your report, describe and explain the observations for the experiment described above.
- **(10 Points)** Launch the attacks and describe your observation.

<u>Task 7 – Defeating Access Randomization (15 Points):</u>

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, you use such an approach to defeat the address randomization countermeasure on the 32-bit VM. First, you need to turn on Ubuntu's address randomization using the following command. Then you will run the same attack against `stack-L1`. Describe and explain your observation.

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

Next, you will use the brute-force approach to attack the vulnerable program repeatedly, hoping that the address we put in the `badfile` can eventually be correct. You should 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. This may take a few minutes, but if you are very unlucky, it may take longer. Describe your observation.

Submission – Report:

- **(7.5 Points)** In your report, describe and explain the observations for the attack described above.
- **(7.5 Points)** Launch the brute-force approach attack and 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
```

Task 8 – StackGuard Protection (10 Points):

Many compilers, 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 previous tasks, you disabled the StackGuard protection mechanism when compiling the programs. In this task, you need to 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, 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.

Submission – Report:

- **(10 Points)** In your report, besides the screenshots and description of your observations, you need to describe your method and provide evidence.

Submission for all Tasks: You need to submit a detailed project report, with screenshots, to describe what you have done and what you have observed. You also need to provide an explanation of the observations that are interesting or surprising. Also, list the important code snippets followed by an explanation. Simply attaching a code without any explanation will not receive credits.