2020-2021
BLG460E - Secure Programming

Term Project
CRN: 21165

Due Date: 01.06.2021, 18:00 PM

1 Introduction

This project aims to provide hands-on experience for students on buffer overflows, canonicalization issues and reverse engineering files. You are required to write a report which explains your code, your analyses and requirements of the assignment.

Merge the given source code pieces and compile an executable file in your test environment. Make sure the default security settings are removed in your test environment.

You can use a virtual box. A 32-bit Ubuntu-12.04-based virtual machine image is provided. You can access the provided virtual machine from lecture slide of the first week. Note that you have to download and install VirtualBox in order to run the virtual machine image. You need 10-GB of space in your HDD in order to run the virtual machine. After you installed the VirtualBox and also extracted the contents of the downloaded virtual machine image, create a new virtual machine in VirtualBox, select the operating system as *Linux*, distribution as *Ubuntu (32-bit)*, assign a RAM space greater than or equal to *1024-MB*.

In this project, you will working in `/user` folder at first step. Your aim is to access a secret file that is in `/admin` folder. The secret file contains a password. You will read the `password` in the secret file and use that password to extract the given zip file.

In the first part you will see a function that controls the `password` which is given as a parameter and returns an integer value. Your aim is passing the privilege mode control without using the right password. After you pass the privilege mode control you will try to read content of the `secretFile` which is under `/admin` folder.

After you read the `password` from `secretFile` you can extract the zip file and you should answer the questions in Part 3.

2 Part 1

You will deal with the main function that takes one argument `argv[1]` in this part. This argument will be used to overflow the stack in `IsPwOk` function. The input argument format is in hex format. The reason to that is to give any ASCII character easily. Therefore, you need to give an input in hex format. `hexToAscii` function will convert the hex input to ASCII characters. For example, `414243616263` will be converted to `ABCabc`. The aim of this part is modifying the return address. You are not supposed to change the given source code. The `IsPwOk` function simply copies the given argument that is finally copied to the password character array of the `IsPwOk` function. 8 octets space is allocated for the password parameter. So, anything given longer than 8 byte will cause an overflow. You need to give the appropriate input to the program so that `IsPwOk` function returns to the else statement on line `11`.

```c
1 int main(int argc, char **argv){
2   int admin_priv = 0;
3   int admin_pw_check = 0;
4   int size_of_argv = strlen(argv[1]);
5
6     hexToAscii(argv[1]);
7     admin_pw_check = IsPwOk(argv[1], size_of_argv);
8     if (!admin_pw_check) {
9       printf("Admin password is not accepted.\n");
10    }
11    else {
12      printf("Admin password is accepted.\n");
13      admin_priv = 1;
14    }
15
16    if(admin_priv == 1){
17          char fn[20], ln[20];
18          printf("Please enter the file name to read. ");
19          scanf("%s", fn);
20          printf("Please enter the log file to write, in case an unauthorized access is attempted. ");
21          scanf("%s", ln);
22          readFileIfPermitted(fn, ln);
23      }
24      else{
25          printf("You are not authorized. \n");
26      }
27  }
28
29
```

`IsPwOk` and `hexToAscii` functions can be seen below:

```
int IsPwOk(char * pw, int size_of_pw){
    char password[8];
    memcpy(password, pw, size_of_pw/2); // half-size character array is enough
    return 0 == strcmp(password, "1312");
}

/* This function converts hex string to ascii string
** Example: input=32334142 output=23AB
** Warning: Do not give any string that contains 00
*/
void hexToAscii(char *passInHex){
  char temp[3];
  char ch;
  int num;
  int i;
  for(i = 0; i < strlen(passInHex); i = i + 2){
    temp[0] = passInHex[i];
    temp[1] = passInHex[i + 1];
    temp[2] = '\0';
    sscanf(temp, "%x", &num);
    ch = num;
    passInHex[i / 2] = ch;
  }
  passInHex[i / 2] = '\0';

}
```

If you look at the code for the `IsPwOk` function, you can see that the password is compared with `"1312"`. So if the input argument is `"3133313200"`, which is equivalent to `"1312"` in ASCII format, then the else statement will be executed. But you are not supposed to give any input starting with `"3133313200"` for this assignment. Your aim is to make the program jump to the else statement that is 11th line of the main without using the right password.

3 Part 2

You are required to study the canonicalization issues in this part. You are required to analyze the code examples, try to bypass the naive security checks. Remember that, you are in the `/user` directory and your aim is to read a file on the `/admin` directory.

```c
/* int isalnum(char c) function is inside ctype.h library.
** It checks if character is alphanumeric or not.
*/
void readFileIfPermitted(const char *fileName, const char *logFileName){
  printf("Reading the file : %s\n", fileName);
  if(isalnum(fileName[0]) && isalnum(logFileName[0])){
    FILE *filePointer;
    int c;
    filePointer = fopen(fileName, "r");

    if(filePointer == NULL){
      FILE *logFile;
      logFile = fopen(logFileName, "w");
      fprintf (logFile, "An unauthorized access attempt has occurred.");
      printf("Error recorded in log file.");
    }

    while ((c = fgetc(filePointer)) != EOF){
        printf("%c", (char)c);
    }

    fclose(filePointer);
  }
  else{
    printf("Invalid filename has been entered.");
  }

  return;
}
```

After you read the password you can access the `files.zip` and you can use it for the next part of the project.

4 Part 3

You could be able to extract the provided zip file right after you learn its `password` by reading the contents of the `admin/secretFile.txt`.

The zip file includes ten files. Analyze these ten files and answer the following questions separately for each file.

- Is the file malicious or benign? Explain how you come to that conclusion.
- What is the entropy of the file? What can you say about the files when you compare their entropies?
- What are the basic information about the file (file type, architecture, etc.)? Is the file executable? How did you obtain these information?
- Does the file contain any debugging information or not? How did you find out this information?
- Can you disassemble the file? Explain why you cannot or how you can.
- When you inspect the strings in the binary file what information could you say about the file? Here are some examples:
  - The file may gather information from /proc/ folder.
  - The file may make HTTP requests.
  - The file may run a shell command.
  - The file may contain an IP address.
  - The file may manipulate the processes.
  - The file may make some file operations.

  Explain how did you conclude your answers.

- After taking precautions, try to run the file in an isolated environment. Were you be able to run the file? If it worked, what happened? If it did not work, why?
- Is the file packed or not? Explain how did you find out whether it is packed or not. If it is packed try to unpack the file and answer all the questions above again for the unpacked file. Compare the results. Make comment on the differences.
- Considering the answers for the questions above, what is the purpose of the file? What does it do?

Hints and Explanations

- The source code should be compiled with `gcc` compiler. In order to disable the stack protection `-fno-stack-protector` option should be given to the compiler. If you want to debug the program, you can use the GNU Project debugger, gdb. For that purpose `-g` option should be added as well. An exemplary compiling is given below.

  ```
  gcc -fno-stack-protector -g -o project project.c
  ```

- The compiled file can be run as below.

  ```
  ./project 31333132000408bff2
  ```

  Note that the input argument usually should be longer than this example for the buffer to overflow.

- Every time you run the program, the address spaces allocated for the process can change owing to ASLR (Address Space Layout Randomization). You may need to disable randomization temporarily with the command below.

  ```
  sudo sysctl kernel.randomize_va_space=0
  ```

  Note that this has a potential to harm the operating system. Use it only on a virtual machine.

- In the first part, you should not try any input that includes `00` because it is the NULL character and terminates any string. If the input contains `00`, the rest of the input will be ignored.
- When probing with probable inputs for the first part, you should avoid changing any part of the stack except the `password` array and the return address. Do not break the structure of the stack.
- For the third part, you can use any program to examine the files. Here are some programs that can help you with that: **ent, nm, radare2, ltrace, strace, file, strings, gdb, objdump, hexdump, readelf**.

gdb Usage

You can use any function of gdb, but some important ones are introduced in this section.

- When the program is compiled with `-g` option, it can be debugged with gdb as follows.

  ```
  gdb assignment
  ```

- The breakpoints should be created on the functions that will be examined thoroughly.

  ```
  break main
  break function1
  break function2
  ```

- The program can be run with one argument as follows.

  ```
  run 31333132000408bff2
  ```

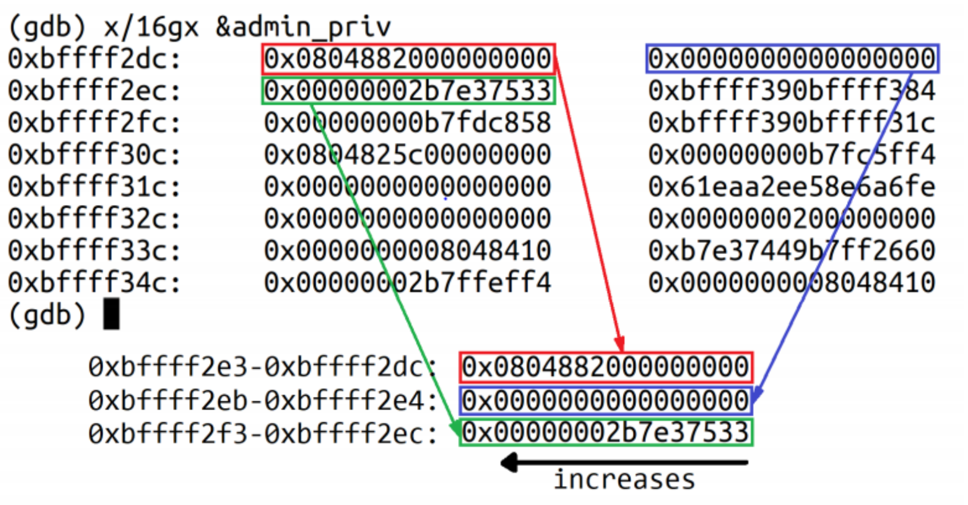
  It may be better to give small inputs for inspection at first.

- When the `run` command is executed, the program will run until the first breakpoint. At any stop on the program you can do followings.
  - `next`: Execute next program line (after stopping); step **over** any function calls in the line.

- `step`: Execute next program line (after stopping); step **into** any function calls in the line.
- `c`: Continue running your program (after stopping, e.g. at a breakpoint).
- `print` *`expr`*: Display the value of an expression. Some examples are:
  `print buffer`
  `print&buffer`
- `disassemble` *`func`*: Display a range of addresses or a function code as machine instructions. This command is useful to track the addresses of machine instructions when executing step-by-step.
- `x/`*`nfu addr`*: Examine memory. **n** is the number of memory parts. **f** is the format letter. **f** can be o(octal), x(hex), d(decimal), u(unsigned decimal), t(binary), f(float), a(address), i(instruction), c(char), s(string) and z(hex, zero padded on the left). **u** is the size letter. Possible size letters are b(byte), h(halfword, 2 bytes), w(word, 4 bytes), and g(giant, 8 bytes). Finally, **addr** is the starting address.

  Some exemplary memory examinations are as follows.

  ```
  x/12gx 0xbffff2ea
  x/20wx 0x0804564a
  x/4gx &password
  x/16wx &buffer
  ```



```
(gdb) x/16gx &admin_priv
0xbffff2dc:     0x0804882000000000      0x0000000000000000
0xbffff2ec:     0x00000002b7e37533      0xbffff390bffff384
0xbffff2fc:     0x00000000b7fdc858      0xbffff390bffff31c
0xbffff30c:     0x0804825c00000000      0x00000000b7fc5ff4
0xbffff31c:     0x0000000000000000      0x61eaa2ee58e6a6fe
0xbffff32c:     0x0000000000000000      0x0000000200000000
0xbffff33c:     0x0000000008048410      0xb7e37449b7ff2660
0xbffff34c:     0x00000002b7ffeff4      0x0000000008048410
(gdb) 

     0xbffff2e3-0xbffff2dc: 0x0804882000000000
     0xbffff2eb-0xbffff2e4: 0x0000000000000000
     0xbffff2f3-0xbffff2ec: 0x00000002b7e37533
```

Figure 1: Memory examination example

In the Figure 1, `x` command is used to show the memory parts of 16 giant words in hex format starting from the address of the `admin_priv` variable. Intel computers are little-endian, therefore the most significant value resides in the rightmost octet. For instance, the value **0x12345678** will reside in an 32-bit word as **0x78563412**. You should keep this in mind when trying the first part of this assignment.

Requirements

For the first part, you should

- give the appropriate input that realizes the desired attack and make comment on each part of each part of the input,
- explain how you found the return address and changed it,

- make comment on the stack before and after the overflow, and
- test and discuss what happens if you do not use `sudo sysctl kernel.randomize_va_space=0` command (in order to revert it back, you can run `sudo sysctl kernel.randomize_va_space=2`).

For the second part, you are required to

- write a report which explains the weaknesses with the code examples, sample attack strings which can bypass the checks and further discussions on your concerns.

For the third part, you are required to

- Obtain as much information as possible and prepare a report that contains your interpretations on the information you gathered from the files.

Important Notes:

- Date and the time of the submission will not be changed. Please be aware of the deadline.
- Interactions among groups are prohibited.
- Send an email to **sayinays@itu.edu.tr** for your questions.