BLG460E - Secure Programming

Take-Home Exam 1 : Buffer Overflows

Due Date: Feb 26, 2019, 23:59 PM

This assignment aims to provide hands-on experience for students on buffer overflows. You need to make some changes on the attached file "assignment1.c" as indicated below and you are required to write a report which explains your code, your analysis and requirements of the assignment.

We provided a 32-bit Ubuntu-12.04-based virtual machine image. You can download it from `https://drive.google.com/file/d/16Am5PDT4fcEkhMBjIqKVToC5gbGT9wJx/view?usp=sharing`. Note that you have to download and install VirtualBox in order to run the virtual machine image. All tests are done in VirtualBox 5.2.18, so this version is recommended. Note that 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* and direct VirtualBox to *ubuntu.vdi* we provided. When Ubuntu starts, you can login to the system with username **secure-programming** and password **12345**.

Part 1

In the first part, you need to modify the `get_uid` function provided in the "assignment1.c" file based on P, where $P = (Your\_Student\_ID \pmod{10} + 2)$.

```
int get_uid() {
    char buffer[P];
    ...
}
```

There are two aims of this function. The first aim is modifying the return address value so that the `uid` is not overwritten with `default_uid` after the function is called as it is seen below.

```
...
uid = get_uid();
uid = default_uid; // this line should not be executed
if (uid == 0) {
...
```

In order to change the return address of the function, you need to find the memory part where the return address is stored and change it accordingly. First, inspect the memory parts following the buffer and find the return address. Then, increase the address just enough to jump over the `uid = default_uid;` line by inspecting the assembly code. Hints are provided at the end of the assignment.

The second purpose is that the `uid` should take a value of 0 in order for the program to go in the `if` statement. Therefore, the function should return integer 0.

Part 2

The program takes one argument: `argv[1]`. This argument will be used in the second buffer overflow 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. `hex_to_ascii` function will convert the hex input to ASCII characters. For example, `414243616263` will be converted to `ABCabc`.

The aim of this part is similar to the first one: modifying the return address. But this time the function is given and you are not supposed to change it. The `IsPwOk` function simply copies the parameter given to the function to `password` character array. The character array is 8-byte long. So, anything given longer than 8 byte will overflow the `password` buffer. You need to give the appropriate input to the program so that the return address should jump to the `else` statement directly as it can be seen below.

```
...
admin_pw_check = IsPwOk(argv[1], size_of_argv);
if (!admin_pw_check) {
  printf("Admin password is not accepted\n");
}
else {
  printf("Admin password is accepted\n");
  admin_priv = 1;
}
...
```

If you look at the code for the `IsPwOk` function, you can see that the password is compared with `"1234"`. So if the input argument is `"3132333400"`, which is equivalent to `"1234"` in ASCII format, then the `else` statement will be executed. But you are not supposed to give any input starting with `"3132333400"` for this assignment. Your aim is to make the program jump to the `else` statement without using the right password.

Hints and Explanations

- The program 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 assignment1 assignment1.c
  ```

- The compiled file can be run as below.

  ```
  ./assignment1 323334350408bff2
  ```

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

- Every time you run the program, the address spaces allocated for the process can change owing to ASLR (Address Space Layout Randomization). In the second part, this can create some problems. Hence, 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 second part, you should not try any input that includes `00` because it is NULL character and ends any string. If the input contains `00`, probably the rest of the input will be ignored.

- When trying input for the second part, you should avoid changing any part of the stack except `password` array and the return address. If you break the structure of the stack, you may encounter some problems.

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 opened 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 323334350408bff2
  ```

  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
    ```

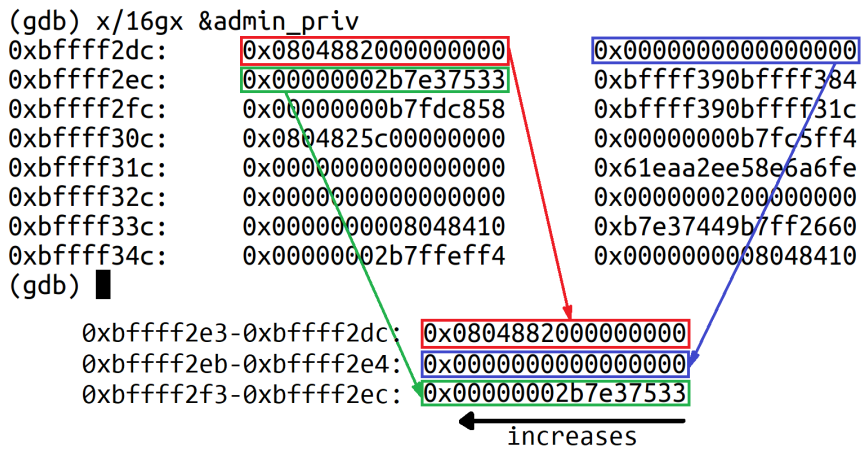

Figure 1: Memory examination example

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. In one giant word, the addresses increase from right to left. For instance, if you want to write `ab12cd34`, first `ab` will be written, then `12`, and so on. Finally, you should see the written part in the memory as `34cd12ab`. You should keep this in mind when trying the second part of this assignment.

Requirements

You should not edit any code except the `get_uid` function. You should submit the edited "assignment1.c" file along with a report.

In order for the assignment to be counted as successful, the output of the program should be as follows.

```
Logging in as Admin
Admin password is accepted
Logged in as Admin
Both buffer overflows succeeded!  YAY!
```

For the first part, you should

- explain your code,
- explain which method you used to implement the buffer overflow (note that you do not have to use gdb, but explain how you managed to achieve an overflow),
- draw the stack structure when the function is called (contents, addresses, and their purposes), and
- test and discuss what happens if you compile and run without `-fno-stack-protector` option.

For the second part, you should

- give the appropriate input that makes the overflow and make comment on each part of it,
- explain how you found the return address and changed it,
- make comment on the stack before and after the copying operation, 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`).