Lab 9.1: Buffer Remedies
IN618 Security

April 12, 2016

Introduction

When working in lower level languages like C we have to carefully manage the computer's memory. If do this incorrectly, buffer overflow vulnerabilities may result. We can take advantage of newer coding practices and compilers to help reduce our risk.

In this lavb exercise we will work with the `overflow` C program we used last week on `sec-student.foo.org.nz`. Log into that server and carry out the steps below on it. You will need a good copy of the original `overflow.c` source from last week's lab.

1 Use compiler features to guard against overflow

Recompile `overflow.c` with the command

```
gcc -fno-stack-protector -o overflow overflow.c
```

There are two things to notice about this command. First, it actually warns us that the `gets()` function that we are using is unsafe. We will deal with that in the next section. Second, note that we used the flag `-fno-stack-protector`. We actually disabled a feature in the compiler that builds executables with the capacity to guard against some stack overflows.

Run the program, giving it an over-log password (about 30 characters) and note the output. Now recompile the program, omitting the `-fno-stack-protector` option. Run the recompiled program, again giving it a 30 character password. Note the difference in the result.

Based on this, can you think of any reason why you would compile a C program without using the stack protection features?

AS a final test, run the program and give it a 20 character password. This is too long - our array is only 15 characters long, but the program doesn't abort. Can you think of why this is true?

2 Changing the code to remove vulnerabilities

Now it's time to fix the code. The compiler has already warned us that the `gets()` function is unsafe. It gets a string from standard input and saves it in the location specified, even if the string is too big to fit in that location. There is no safe way to use `gets()`, so we need a differnet function. One suitable replacement is `fgets()`. In your `overflow.c` file, replace the line

```
gets(buff);
```

with

```
fgets(buff, 15, stdin);
```

You can see that `fgets()` takes three arguments. The first is the variable where the string will be stored (`buff`). The second is the number of characters to get. We have room for 15 characters, so we makes sure we don't try to get any more. The third argument of the place from which we get the input. We specify standard input here (`stdin`), but `fgets()` can also read from a file.

There is actually one other risky line of code here:

```
if(strcmp(buff, "in618"))
```

This function compares two strings, but it has the potential to read off the end of a string, possibly into unsafe territory, if one of the strings is not terminated correctly. It also has a replacement, so we can replace the line above with

```
if(strncmp(buff, "in618", 15))
```

This compares two strings, but only up to a maximum of 15 characters in this case.

Since `strcmp()` doesn't write to memory it's less risky than functions like `gets()`, but using a function like `strncmp()` makes your code more robust overall in addition to being more secure.

Recompile your modified code and test it using over-long inputs to verify that the vulnerability has been corrected.