CPS221 Lecture: Secure Programming

last revised 12/2/14

Objectives

1. To introduce some key secure programming practices

Materials:

1. Projectable of before and after examples of buffer overflow
2. Arithmetic Demo
3. SQL Injection Demo + `injection` database in mysql
4. Unescaped version of 404 page from end of lecture

I. Introduction

A. Today we want to look at some programming and related practices pertinent to producing software that does not contain vulnerabilities that can be exploited by hackers.

B. In particular, we want to look at four categories of vulnerability that can be prevented by appropriate care in programming.

C. Actually, all four relate to a broad general principle whenever a program processes information that is supplied by the user (e.g. via a network packet, a URL, or input into a web form), take precautions to prevent malicious input from instituting harm.

II. Buffer Overflows

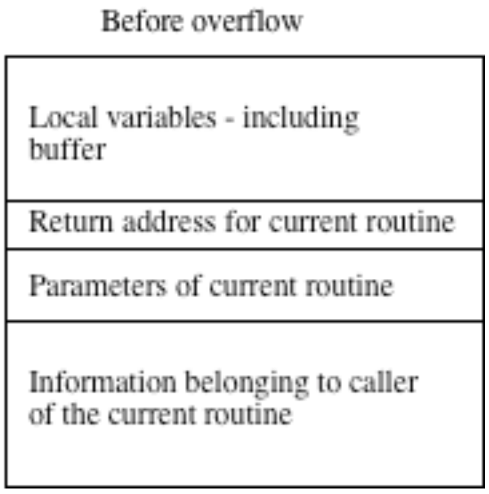
A. When a program needs to accept input data, or copies data from one location to another, it must specify a buffer in memory to hold the accepted or copied data.

1. Of course, the buffer has to be allocated with some amount of space capable of holding the largest expected legitimate data.

2. What if the data exceeds the size of the buffer? In this case, we have what is called a buffer overflow.

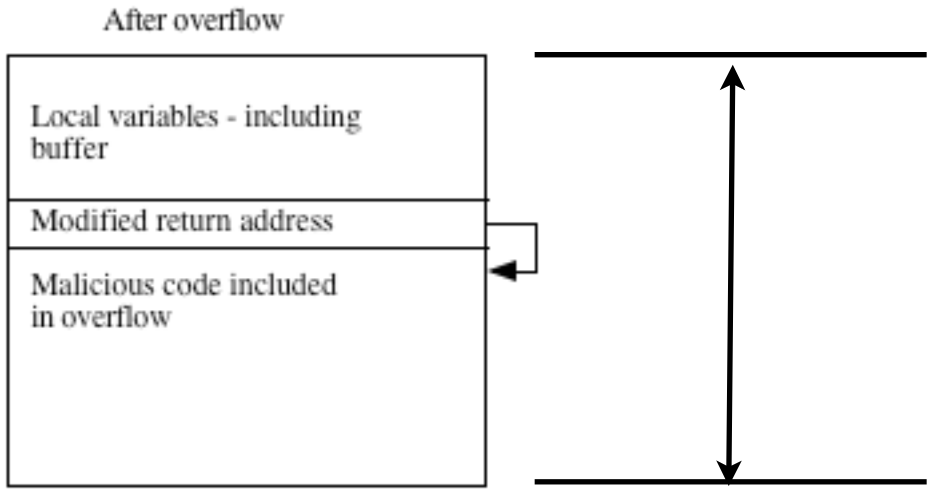
3. Historically, many of the library routines used by programs written in C or C++ simply stored the excess data beyond the end of the buffer, overwriting legitimate data that was there.

    a. When information is transmitted over the network, a malicious user can take advantage of this by knowledge of the way that local variables are allocated on the runtime stack

    
    Before overflow
Local variables - including buffer
Return address for current routine
Parameters of current routine
Information belonging to caller of the current routine

    PROJECT

    b. What if the information is longer than the size of the buffer? In this case, it will overflow the buffer, replacing information higher up the stack with malicious code. A malicious user might fill the overflow space with machine code of his/her own devising, and might change the return address to point to this code instead of the original caller of the routine. (All of the data in the range indicated was provided by the malicious user.)


After overflow
Local variables - including buffer
Modified return address
Malicious code included in overflow

PROJECT

(This, of course, requires that the attacker knows the architecture and stack structure of the machine the code will run on - which is why such attacks are typically made on widely used platforms.)

c. When the original routine completes, it returns to the address specified in the stack - which has now been modified by the malicious user. As a result, the CPU begins executing the malicious code provided by the user. This code can do anything the user could do.

   1. If the overflow occurs in a routine running in kernel mode (such as in networking code), the attacker can do essentially anything desired in the malicious code - e.g. installing a worm or virus, a keystroke logger, or a remotely-activated “bot” used for denial of service or sending spam email.

   2. Something similar could occur in a routine that is called by user-written code (e.g. a routine that is part of an API) and takes a parameter of some array type. If he caller is malicious, it is possible to provide a parameter that overflows buffer space allocated by the API routine.

B. Avoiding buffer overflow vulnerabilities involves various measures including using library routines that do not allow information to overflow allocated buffer space (i.e. either restricting the length of an input value or checking the length before copying).

   (Note that the C library, for example, provides routines called strcpy and strncpy for copying character strings. Both look for a null character (ASCII 0) to denote the end of the string. But strncpy has an extra parameter that specifies the maximum number of characters to be copied, and stops copying at that point whether or not the terminating null has been seen - but strcpy does not have this parameter or make this check.

C. The buffer overflow problem has been known for a long time, yet there are still myriads of vulnerabilities in existence due to programmer failure or laziness - e.g. using something like strcpy when strncpy should have been used.

   Given the present state of knowledge, this is totally unacceptable!

III. Integer Overflows

A. Mathematically, there are infinitely many integers - but standard data types impose an upper limit on the size of an integer - e.g. the largest signed integer a 32 bit int can represent is 2147483647.

   When an operation such as addition produces a result that is bigger than this, strange things happen.

   DEMO - Run Arithmetic Demo from CPS311 - Demo adding 1 to this number.

B. The book gave some examples of how integer overflows (and related consequences of the use of a finite number of bits to represent an integer) can be exploited to bypass security checks based on buffer lengths and the like. Note that it is not the integer overflow per se that causes the problem, but that this can be exploited to bypass checks built into the code.

III. **SQL Injection**

A. As you recall, facilities like JDBC allow SQL statements to be interpreted at run time, and applications can create SQL statements at run time by concatenating constant strings and run-time variables.

B. This, however, can create a vulnerability that can be exploited by a hacker. Consider the following simplified example:

1. We have a database containing user names and encrypted passwords.

   SHOW: passwords table in injection database

2. We have a program that gets a username and password from the user, and then queries the database using JDBC to see if the password is correct. The following is the

```
private void loginButtonActionPerformed(java.awt.event.ActionEvent evt) {
        String user = usernameField.getText();
        String password = passwordField.getText();
        String query = "select name from passwords where name = '" + user + "' " +
                       "and encrypted_password = PASSWORD('" + password + "')"
        System.out.println(query);
        try {
            ResultSet results = statement.executeQuery(query);
            // If the username exists and the password is correct, the query
            // will find one row.  Otherwise, it will find zero rows.
            if (results.next())
                resultsLabel.setText("Logged in as " + results.getString(1));
            else
                resultsLabel.setText("Login failed");
        }
        catch(SQLException e) {
            System.err.println("SQLException " + e);
        }
    }
```

   PROJECT

3. DEMO using `java -jar`, but don't show terminal

a. login to `aardvark` using `bad`

b. login to `aardvark` using `password`

c. How would we login to `root` if we didn't know the root password?

   ASK

   DEMO - enter username `root' or '1=1` and blank password

4. How does this work? Show query echoed on terminal

C. How can we avoid vulnerability to this kind of attack?

1. One approach - which is very hard to do completely right - is called "input sanitization" - that is, discarding characters like ' that appear in the input before constructing the query.

   Unfortunately, it's hard to catch all the possibilities, so this is at best a risky approach.

2. A better approach is possible by compiling the query at compile time rather than constructing and interpreting a query string at run time.

   In the case of JDBC, this involves using something called a PreparedStatement; other approaches to embedding SQL in a programming language generally offer something analogous.

   SHOW: Javadoc for prepareStatement() method of Connection.

IV. Cross-Site Scripting

A. Cross-site scripting builds on the fact that web browsers recognize - and execute - javascript scripts that appear between `<script>` and `</script>` tags. Thus, if we can contrive to include a script in a page that is displayed by a web browser, then we can get the browser to execute the script.

This, of course, makes it possible for a web page stored at a malicious site - or a legitimate page that has been modified by an adversary - to do harmful things.

B. Cross-site scripting builds on this by embedding <script> tags in places like URLs or user-inputted text that actually execute scripts located on a site controlled by the hacker. This allows the embedded script to be fairly short (and hence less likely to be noticed), since the bulk of the code resides elsewhere.

   The book gives several examples of how this might be done.

C. The best defense against cross-site scripting attacks is a technique known as "input sanitization". The basic idea is to replace special characters like "<" with escaped versions using hexadecimal codes.

   1. Example: enter the URL

      www.cs.gordon.edu/<script>alert("You've been hacked");</script>

      Note 404 page - then view source to notice how tags have been escaped.

   2. Now access unescaped version of page