SOFTWARE SECURITY SERIES

Foreword by Gary McGraw

SECURE PROGRAMMING WITH STATIC ANALYSIS

Brian Chess Jacob West

Praise for *Secure Programming with Static Analysis*

"We designed Java so that it could be analyzed statically. This book shows you how to apply advanced static analysis techniques to create more secure, more reliable software."

—Bill Joy
Co-founder of Sun Microsystems, co-inventor of the Java programming language

"If you want to learn how promising new code-scanning tools can improve the security of your software, then this is the book for you. The first of its kind, *Secure Programming with Static Analysis* is well written and tells you what you need to know without getting too bogged down in details. This book sets the standard."

—David Wagner
Associate Professor, University of California, Berkeley

"Brian and Jacob can write about software security from the 'been there. done that.' perspective. Read what they've written - it's chock full of good advice."

—Marcus Ranum
Inventor of the firewall, Chief Scientist, Tenable Security

"Over the past few years, we've seen several books on software security hitting the bookstores, including my own. While they've all provided their own views of good software security practices, this book fills a void that none of the others have covered. The authors have done a magnificent job at describing in detail how to do static source code analysis using all the tools and technologies available today. Kudos for arming the developer with a clear understanding of the topic as well as a wealth of practical guidance on how to put that understanding into practice. It should be on the required reading list for anyone and everyone developing software today."

—Kenneth R. van Wyk
President and Principal Consultant, KRvW Associates, LLC.

"Software developers are the first and best line of defense for the security of their code. This book gives them the security development knowledge and the tools they need in order to eliminate vulnerabilities before they move into the final products that can be exploited."

—Howard A. Schmidt
Former White House Cyber Security Advisor

"Modern artifacts are built with computer assistance. You would never think to build bridges, tunnels, or airplanes without the most sophisticated, state of the art tools. And yet, for some reason, many programmers develop their software without the aid of the best static analysis tools. This is the primary reason that so many software systems are

Continued

The creators of C++ provided a direct translation of the `gets()` functionality into C++ syntax, vulnerability and all. The following two lines of C++ are just as vulnerable as a call to `gets()`:

```
char buf[128];
cin >> buf;
```

Because the C++ problem is so faithful to the `gets()` problem, we find it odd that none of the compilers or runtime environments that we are aware of will give a warning about the danger inherent in this code.

C++ does provide a much better option. By reading into a string object instead of a character array, the buffer overflow problem disappears because the string object automatically allocates enough space to hold the input:

```
string str;
cin >> str;
```

But we're not out of the woods yet. Although the buffer overflow problem is gone, the code still doesn't place any limit on the amount of input it will accept. That makes it easy for an attacker to force a low-memory or out-of-memory condition on the program. Although the means of exploiting such a condition are not as obvious as with a buffer overflow, many programs can, at the very least, be forced to crash when they butt up against a memory limitation. The attacker might also take advantage of this code simply to slow the program down. Introducing lag is a great prelude to exploiting a bug related to time and state.

Java does bounds checking and enforces type safety, so buffer overflow is not a scourge in Java the way it is in C and C++, and there is no direct analog to the `gets()` problem. But just as with C++, Java provides an all-too-convenient way to read an unbounded amount of input into a string:

```
String str;
str = bufferedReader.readLine();
```

And just as with C++, an attacker can exploit this code to cause a low-memory condition from which many programs will have a hard time recovering.

We are fascinated by the similarity of the problems built into the libraries for these programming languages. It would appear that input-length checks are not a priority among language designers, so, in addition to avoiding the traps, you should expect you'll need design your own security-enhanced APIs.

Bound Numeric Input

Check numeric input against both a maximum value and a minimum value as part of input validation. Watch out for operations that might be capable of carrying a number beyond its maximum or minimum value.

When an attacker takes advantage of the limited capacity of an integral variable, the problem goes by the name integer overflow. In C and C++, integer overflow typically surfaces as part of the setup for a buffer overflow attack. See Chapter 7, "Bride of Buffer Overflow," for additional information about integer overflow in C and C++.

Some built-in classes and methods are well equipped to help with input validation. For example, this Java statement takes care of quite a few input validation problems:

```
x = Integer.parseInt(inputValueForX);
```

But if a program performs an arithmetic operation on a value of unknown magnitude, the mathematical result might not fit in the space allotted for the result—hence

```
Integer.MAX_VALUE + 1 == Integer.MIN_VALUE
```

and

```
1073741824*4 == 0
```

Integer overflow in Java doesn't lead to buffer overflow vulnerabilities, but it can still cause undesirable behavior. Example 5.25 gives a small code excerpt from a fictitious e-commerce site. The user provides the number of items to be purchased, and the code multiplies the number of items by the item price. Integers in Java are 32 bits, so if an attacker can cause `numPurchased*100` to be a larger number than can be represented by a signed 32-bit integer, the total price could end up being negative. For example, if an attacker supplies the value `42949671` for `numPurchased`, the total will be `-196`, which could potentially result in a `$196.00` credit given to the attacker.

Example 5.25 The following code excerpt from a fictitious e-commerce site demonstrates one way an integer overflow can occur: if an attacker supplies the value 42949671 for numPurchased, the total will be -196.

```
String numStr = request.getParameter("numPurchased");
int numPurchased = Integer.parseInt(numStr);
if (numPurchased > 0) {
  total = numPurchased * 100; // each item costs $100
}
```

The best way to avoid integer overflow problems is to check all integer input against both an upper bound and a lower bound. Ideally, the bounds should be chosen so that any subsequent calculations that are performed will not exceed the capacity of the variable being used. If such bounds would be too restrictive, the program must include internal checks to make sure the values it computes do not result in an overflow.

Java offers several different sizes of integral values: `char` (8 bits), `short` (16 bits), `int` (32 bits), and `long` (64 bits). However, Java does not offer unsigned integral types, so there is no way to avoid checking a lower bound as well as an upper bound. If you want to handle only non-negative numbers in Java, you must check to make sure that the values you receive are greater than or equal to zero.

By default, the Java compiler will complain if a value of a larger type is assigned to a variable of a smaller type, as shown in the following example:

```
int i = 0;
char c;
c = i;
```

In the previous code, javac will complain about a "possible loss of precision." However, you can easily shut up the compiler by modifying the last line as follows:

```
c = (char) i;
```

Be cautious about such casts! They can lead to unexpected loss of precision and possibly sign extension.

In C and C++, reading input using an unsigned integral type obviates the need to check against a lower bound. This luxury comes at a price, though: Programmers must be concerned about operations that mix signed and unsigned values because converting an unsigned value to a signed value could produce a negative result, and converting a signed value to an unsigned value could produce an unexpectedly large result.

Example 5.26 demonstrates a signed-to-unsigned conversion problem. The function `doAlloc()` takes a signed integer as a parameter and uses as the argument to `malloc()`. Because `malloc()` takes an unsigned argument, the call to `malloc()` implies a conversion from a signed value to an unsigned value. If `doAlloc()` is invoked with a negative argument, `malloc()` will try to allocate a very large amount of memory. This could cause the allocation to fail—or, perhaps worse, the allocation might succeed and deprive the program of a large chunk of memory.

Example 5.27 demonstrates an unsigned-to-signed conversion problem. The function `getFileSize()` returns a signed quantity but derives its return value from an unsigned field. This means that files larger than 2GB will appear to have a negative size.

Example 5.26 An implicit conversion from a signed value to an unsigned value. `doAlloc()` accepts a signed argument, but `malloc()` treats its argument as unsigned. A negative argument to `doAlloc()` results in `malloc()` attempting to allocate a very large amount of memory.

```
void* doAlloc(int sz) {
  return malloc(sz); /* Implicit conversion here: malloc()
                        accepts an unsigned argument. */
}
```

Example 5.27 An implicit conversion from an unsigned value to a signed value. `getFileSize()` returns a signed value, but it takes its return value from an unsigned struct field. This causes the value returned from `getFileSize()` to be negative for files larger than 2GB.

```
int getFileSize(char* name) {
  struct stat st;
  if (!stat(name, &st)) {
    return st.st_size; /* st.st_size is unsigned */
  } else {
    return -1;
  }
}
```

C error handling conventions make unsigned types harder to use. Many C functions return -1 on error, so a programmer who stores the result of such a function in an unsigned type runs the risk of missing an error condition. This is another example of the way mixing signed and unsigned operations can cause trouble.

5.3 Preventing Metacharacter Vulnerabilities

Allowing attackers to control commands sent to the database, file system, browser, or other subsystems leads to big trouble. Most programmers aren't dumb enough to intentionally give away direct control of these subsystems, but because of the way the interfaces are designed, it's easy to unintentionally blow it.

All scripting languages and markup languages that emphasize ease of use or that sometimes serve in an interactive capacity have at least one thing in common: They accept a fluid combination of control structures and data. For example, the SQL query

```
SELECT * FROM emp WHERE name = 'Brian'
```

combines the keywords SELECT, FROM, WHERE, and = with the data *, emp, name, and Brian. This natural combination of control structures and data is one of the things that make these languages easy to use.

The problem is that, without special attention, programmers might unwittingly give a user the capability to add, remove, or alter the meaning of control structures when they intend to allow the user to specify only some of the data. An attacker usually exploits this kind of vulnerability by specifying *metacharacters,* characters (or character sequences) that have special meaning in the input language. That makes the single quote (') a dangerous character in SQL queries and double periods (..) risky in file system paths. For command shells, the semicolon (;) and double ampersand (&&) are enormously powerful, while newline (\n) is critical in log files. And the list goes on. The problem is compounded further by the fact that there are often multiple ways to encode the same metacharacter and that different implementations of the same language might recognize different metacharacters or different encodings of metacharacters.

This section begins with a good answer to many metacharacter problems: Use parameterized commands. We continue the SQL example there. Then we look at three other types of metacharacter vulnerabilities:

- Path manipulation
- Command injection
- Log forging

We discuss many other types of metacharacter vulnerabilities in subsequent chapters. Format string errors are found in Chapter 6 as part of the discussion of buffer overflow, cross-site scripting and HTTP response splitting vulnerabilities are discussed in Chapter 7, and XML injection is covered in Chapter 10, "XML and Web Services."

Use Parameterized Requests

Many metacharacter vulnerabilities can be eliminated by keeping data and control information separate. Some interfaces have this capability built in. Use it! These interfaces allow the program to supply a parameterized request and a set of data values to be filled in as parameter values.

The common mistake is to instead form requests using string concatenation. The code in Example 5.28 assembles a SQL query by concatenating strings, leaving it vulnerable to a *SQL injection* attack.

Example 5.28 Forming a SQL query by concatenating strings leaves this code open to a SQL injection attack.

```
String userName = ctx.getAuthenticatedUserName();
String itemName = request.getParameter("itemName");
String query = "SELECT * FROM items WHERE owner = '"
               + userName + "' AND itemname = '"
               + itemName + "'";
Statement stmt = conn.createStatement();
ResultSet rs = stmt.executeQuery(query);
```

The programmer who wrote the code intended to form database queries that look like this:

```
SELECT * FROM items
      WHERE owner = <userName> AND itemname = <itemName>;
```