

Secure Coding

David Wong 2002-06-20

Secure Coding

by David Wong

last updated June 20, 2002

Building Secure Systems

Several months ago, Bill Gates announced that [security would be the number one priority at Microsoft](#). Several groups at Microsoft, such as the Trusted Computing Group and the Secure Windows Initiative strive to improve security in Microsoft products and ultimately improve security for individuals and corporations worldwide. These initiatives are not surprising, considering the major vulnerabilities found recently in Windows XP, Internet Information Server, Internet Explorer, and Outlook. Due to the popularity of Microsoft products and their market share, the vulnerabilities have caused havoc all across the Internet. If Microsoft, with it's billions of dollars of resources and talent, has all these security issues, how do you handle the problem of building trusted systems.

As any seasoned security professional will tell you, it's impossible to build bug-free, vulnerability free software. The resources required to create such software will be infinite, and financial analysts don't like seeing that on a balance sheet. The name of the game in the security industry is risk mitigation. That is, reducing the risk to an acceptable level. This article will provide a brief overview of some of the key issues of secure coding. It will identify some common mistakes made when developing software that lead to security vulnerabilities. This is followed by a list of best practices that, if followed religiously, will help you avoid 90% of all security vulnerabilities. The article concludes with a list of resources that will aid in your quest to build more secure software.

Secure Coding - Common Errors

Developing secure software is a process that requires vigilance at all stages and at all levels of the software development organization. It entails strict security requirements, application developers with a strong understanding of security issues, and a quality assessment (QA) team that can dig up security problems. However, in order to tackle this problem, you must learn to

walk before you run. It is unlikely that any software company will try to deal with security bugs by trying to address all problems at once. In this section, we identify the five problems that make up 90% of all security vulnerabilities. If you take steps to address these issues, you can increase the security of your application by an order of magnitude.

Buffer Overflow

The buffer overflow is probably the programming mistake that is most widely exploited by hackers. The details of how a buffer overflow can be exploited to cause a program to execute arbitrary code is beyond the scope of this article. This has been well documented by others. Suffice it to say that if there is a remotely exploitable buffer overflow condition in software, a hacker can potentially take complete control of the system. The root cause of this problem is that a static, fixed size variable is used to store input, and since a malicious attacker can overwrite these buffers, there is a potential for an exploit.

The key to avoiding this problem is to verify that all buffers in the program check the length of the input data before copying to the buffer. If the input data exceeds the buffer size, an exception should be logged and program flow changed. This can be a long and tedious process for any reasonably large program. Fortunately, there are many tools that can automatically identify this condition. However, these tools do not constitute a silver bullet: they may identify the problem, but the programmer still has to go through the results and fix each one.

Writing exploit code to attack a buffer overflow condition is often considered black magic. It does indeed require a lot of skill and knowledge of operating systems, compilers, assembly code. As a result, many people incorrectly assume that since it's so difficult to identify and exploit buffer overflow vulnerabilities, that there is minimal risk. A quick search through popular security sites such as [SecurityFocus](#) and [Packetstorm](#) reveal that numerous exploits for commercial software can be downloaded. Unskilled hackers can simply download the program and perform "Point and Click" hacking. But how does the original author of the exploit code find and exploit buffer overflow?

The author of this article is aware of several hacker groups that obtain software solely for the purposes of identifying security vulnerabilities. For example, one group is currently going through all of the [ISAPI extensions](#) in Microsoft IIS to identify buffer overflow conditions. They are methodically going through each ISAPI extension with debuggers, disassemblers, and specialized hacking programs that throw unexpected input to the Web server. Microsoft

programs are a favorite target of hackers due to the company's unpopularity in the open source world, and also because of the widespread deployment of its products. However, all companies that write software are at risk.

Format String vulnerabilities

Format string vulnerabilities are a new class of security problems that have recently been discovered over the past couple of years. Format strings are a programming construct used in the C and C++ programming languages used for formatting I/O. They contain special identifiers (such as %s for strings, %d for integers) that if used in malicious input, can reveal information about the call stack and variables used in functions. In particular, the dangerous %n identifier can be used to overwrite data in memory. Since overwriting memory allows hackers to do basically the same thing as buffer overflows, the results are the same: arbitrary code execution.

The root cause of format string vulnerabilities is the use of variable argument functions in C/C++. These problems can be eliminated by proper input validation and exception checking in the code. In addition, automated code testing tools can be used to identify format string bugs like: `printf(string);` and recommend that `printf("%s", string);` should be used instead.

Authentication

Authentication is the most critical component of any security system. Incorrectly authenticating a user will yield all other security functions such as encryption, audit, and authorization useless. The most common mistake with authentication is to use weak authentication credentials that would allow an attacker to brute force the credentials, such as a password. In addition, stringent password policies are required to mitigate password guessing attacks. Password composition requirements depend on the level of security required by the application and the user functionality requirements of the application. In general, it is always recommended that passwords are at least 8 characters in length and include alphanumeric and special characters.

Many applications, particularly Web applications, use authenticators to identify a user once they are logged in. Authenticators are like "tickets" that are issued once a user submits authentication credentials. These "tickets" can be used to verify authenticity instead of providing a username or password during a session. With Web applications, authenticators are often stored in cookies. During the design of an application, it is important to ensure that authenticators are not subject to brute force and prediction attacks. During our testing of

applications at Foundstone, we have often found that authenticators often contain weaknesses.

Authorization

Authorization is the process of allowing, or disallowing, access to a particular resource based on the identification of an authenticated principal. Authorization vulnerabilities are a common problem in many software applications. Mistakes often made are:

- **Authorization is not properly performed.** Once authenticated, the interface will grant access to the resource as long as authentication is provided. This is usually a case where a resource identifier is required, and it is assumed that the identifier cannot be guessed or changed. For example, during one of our tests of a Web banking application, the Web server set a Javascript variable to our account number. By simply changing our account number, we were able to view and edit other accounts.
- **Too much trust is based on user input.** For example, an HTTP cookie is a type of user input to a Web application that can be modified. If an application bases its authorization decision on the cookie, it may be relying on falsified data. The falsified data can be either the username or the resource requested. During a test of on-line collaboration tool, there was a cookie variable "admin". If it was set to "true", the application would provide the illegitimate user with an administrative interface.
- **Canonicalization errors.** Applications often make authorization decisions based on authentication credentials and the resource requested. Many security vulnerabilities have been related to canonicalization errors in the resource name. For example, although an application might deny access to `\secure\secret.txt` it may grant access to `\public\..\secure\secret.txt` based on the directory name. Unicode and hex encoding are in the same category. There are many more examples of canonicalization errors and we refer you to [Writing Secure Code](#) by Michael Howard and David Leblanc, which is also discussed in the resources section below.

Cryptography

If cryptography is performed in the application, the sensitivity of the data is assumed to be high. Skilled programmers who have a thorough understanding of the mathematics behind cryptographic algorithms are very rare. A mistake in custom designed cryptographic algorithm or the implementation can completely undermine the security of the application. During our security audits, we have found holes in several custom-built cryptographic algorithms using

standard cryptanalysis techniques, and we are by no means cryptography experts. Many applications use industry standard algorithms like RSA or blowfish, but do not properly handle memory and therefore leave the password and clear-text data susceptible to theft. We highly recommend that you use CryptoAPI and CAPICOM built inside the Windows operating system or buy a third party implementation of the libraries to avoid problems with the use of cryptography. It is too easy to mistakenly fall into the trap and think that data "looks" encrypted and is therefore safe.

Best Practices For Secure Coding

By now, you must be wondering, with all these potential problems, how do I find out if my software has these problems? More importantly, how do I fix them? The answer to the first question is easy. Perform security audits, security tests, and security code reviews to identify security bugs in the application. The answer to the second questions is much more difficult. Building secure software has been a problem many people have been trying to solve for a long time. We don't pretend to the answers here, but below are a list of guidelines that can help you avoid 90% of the commonly exploited security vulnerabilities.

Distrust User Input

A hacker will often attack an application through its external interfaces. Examples of external interfaces are sockets, Web forms, command line input, files, etc. Many exploits will send malicious input data to cause the application to behave in an unexpected manner, such as in a buffer overflow. Although this may sound like common sense, understanding and identifying all input may not always be straightforward. For example, in a Web application, the most common user input is the URI and form data. However, a hacker who is not using a standard Web browser can manipulate all HTTP headers, cookies, and hidden data. These types of user input should never be trusted to contain safe data.

Input validation

Once the input is identified, a description of valid data for each input should be developed. An example in the case of authentication is the username that should only contain alphanumeric characters from 6-20 characters and a password contains 6-20 printable characters. Then input validation and checking needs to be performed early in the code. If the client sends a buffer

overflow of more than 20 characters, the applications should disconnect the user. If SQL injection attacks are performed, character checks in the application should not allow the data to pass to the SQL server. This kind of distrust of user input combined with proper input validation will prevent most of the attacks at their roots. Your application could very well have SQL injection or buffer overflow vulnerabilities but they probably won't be exploitable if proper input validation is performed. The key to this technique is identifying all user input and properly restricting valid data.

Magic Switches

The list here is probably the informational gold mine you've been looking for – a list of magic switches you can flip on to improve the security of your application.

- VCC /G – Visual Studio .NET has a /G option that adds automatic bounds checking to prevent buffer overflows. Keep in mind there have been techniques to bypass automatic bounds checking. However, this does make it more difficult for hackers to exploit the buffer overflow and make it easier for you to identify the vulnerability.
- midl /robust – Using the /robust switch with the midl compiler will add strong parameter checking in RPC applications.
- DCOM encryption – Turn on wired level packet encryption for DCOM applications to encrypt network traffic.
- Avoid using Null DACLS – Never use NULL DACLS (Discretionary Access Control Lists) in an application.

Secure Coding Resources

Tools

The following is a list of tools that perform automatic source code analysis and identify flaws in the code. One strength of automated tools is that they require less initial work. Less experienced developers can run these tools and analyze the output and fix the problems. In addition, the tools listed below each have a large database (currently about 100) of common security flaws help identify vulnerabilities that the developer may be unaware of. Automated tools can also wade through large amounts of code in little time. As with every tool, it is wise to

know the tool's limitations. These tools produce very verbose output, with a high amount of false positives, and it takes a lot of time to sort through the output. In addition, automated tools work amazingly well with problems such as buffer overflow conditions, however, no automated tool will follow the logic of your program and determine if a resource is authorized or not. Specifically, the authentication and authorization vulnerabilities are not particularly well suited for automated analysis. Our recommendation is to use automated code analysis tools to identify buffer overflow conditions and sources of input validation attacks. And follow up with manual source code analysis by security professionals for any code related to authentication, authorization, and encryption. Some tools that can help in this process include:

- [ITS4](#)
- [RATS](#)
- [Flawfinder](#)

Secure Coding Books

In the past, looking for good resources on Secure Coding was almost impossible. There were a few Web sites with discussions of buffer overflows and how they work. But nothing that was comprehensive was available, and nothing that targeted the Windows environment existed. This year, two excellent books on secure programming were published. As mentioned earlier, [Writing Secure Code](#) is an excellent treatise on developing secure code with specific examples on the Win32 API and lessons learned from Microsoft. [Building Secure Software: How to Avoid Security Problems the Right Way](#) by John Viega, Gary McGraw is more UNIX oriented, but teaches lessons on secure programming that all developers should know.

Conclusion

Developing secure software is not an easy task. But based on the number of attacks on software today, it would be negligent to not build hacker resistant code. The resources required to build secure software is high. However, by educating programmers to avoid common security mistakes, performing security code reviews, and testing applications for security bugs, organizations can move quickly to eliminate many of the vulnerabilities commonly exploited today.

David Wong is a principal consultant with [Foundstone](#) and performs code reviews and security testing for

major software companies and other large corporations. He has taught Secure Programming crash courses to several major software companies. David is a contributing author to *Hacking Exposed Windows 2000* and *Hacking Exposed*, 3rd edition. David holds the designation of Certified Information Systems Security Professional (CISSP).

Relevant Links

[Security Zone Articles](#)

DevX

[Secure Programming Mailing List](#)

SecurityFocus

[Web Application Security Mailing List](#)

SecurityFocus

[The Open Web Application Security Project](#)

[The Tao of Windows Buffer Overflow](#)

Cult of the Dead Cow

[Privacy Statement](#)

Copyright 2006, SecurityFocus