



Secure Coding Principles

White Paper

Abstract

This paper describes the principles of Secure Coding, and why these principles are an absolute must know for every programmer.

Tometa creates custom software for you

Tometa Software designs and develops robust software solutions for virtually all industries including in-house (vertical market) and retail software, some of which is on the shelves at your local software store. We focus our unique combination of creative, technical, and problem-solving skills on meeting our client's objectives. Because of our clarity of purpose, commitment to process, and broad professional skill sets, we are able to provide our clients with world-class solutions that are functionally superior and fully aligned with our client's strategic focus.

Balancing development speed, quality and cost is what we are all about. Tometa combines agile development practices with fixed pricing, so you know what the cost, end product, and delivery time table look like—up front. If we underestimate the effort, we complete the overrun on our dime. Simple as that. That's why large enterprise firms like Alcoa and NASDAQ choose Tometa.

Tometa's agile development expertise and low-overhead US location keep our prices comparable to offshore vendors – without offshore challenges. Using a fixed pricing model, we provide upfront visibility into a project's ultimate costs, end product and delivery schedule. Our clients like knowing that we have “skin in the game” – a fixed price that aligns our goals with yours, incenting us to get the job done right and fast.

Lastly, as a Microsoft Certified Gold Partner, Tometa Software, can customize its products or create custom web, client/server, and traditional applications. With programming experience in C#, C++, Visual Basic, CGI, HTML, RPG, Delphi, Java and many others; Tometa Software is uniquely positioned to meet your needs as a development firm.

[Check us out today](#)

Overview on Secure Coding Principles

Secure coding is the topic of lectures, books, seminars, academic classes and even entire careers. Numerous articles, books, and websites are dedicated to helping programmers understand secure coding. This document is intended as a means to help experienced and novice programmers develop secure code. Good programmers write good code, bad programmers write bad code, but all programmers seem to write insecure code. This is because it is hard to write good secure code. Many programmers don't understand security as it relates to coding. Their main focus in software development is to make it work. Often they have no knowledge or regard for the principles of design, persistence, and exploitation avoidance of secure coding. This poses a problem for everyone who relies on the software they develop, and puts huge liability on the organization from which the code originated. Insecure code can allow an intruder to view sensitive information, change or delete valuable or important data, or to run programs or plant malicious code within the software system they have exploited. For these reasons, security must be a priority for every software developer and programmer.

Security in Design

Secure systems are built by employing the principles of secure coding into the design process and persistently evaluating the security of the application throughout the development process. Many security vulnerabilities could easily be prevented if security were taken into consideration at the beginning of the development process. While it is nearly impossible to come up with a list of every possible vulnerability that could exist as a result of coding oversights, it is possible to understand some of the more typical and common problems that often exhibit themselves as coding weaknesses and vulnerabilities. Some of the most common security exploits are:

- Weak file and group permissions
- Race conditions
- Problems with temporary files and session variables
- Buffer overflows and memory pointer exploits
- Overly complex and unnecessary code
- Hard-coding passwords
- User Input

Knowing these will help protect your applications from the most obvious security issues. Designing and implementing countermeasures to these most common exploits is a good place to start in developing secure applications. Also, enforcing programming standards and persistent regression testing will confirm that the application is secure.

Principle of Least Privilege

Program design should follow the principle of least privilege. This requires that a user be given no more privilege than necessary to perform a job. Ensuring least privilege requires identifying what the user's job is, determining the minimum set of privileges required to perform that job, and restricting the user to a domain with those privileges and nothing more. This prevents the disclosure of sensitive data, and prevents unauthorized users from gaining access to programs or areas where they were never meant to be.

Principle of Exclusive Rights

Race conditions occur when the outcome of interrelated events depend on a particular event ordering sequence that cannot be guaranteed making the final state of the system unpredictable. A typical example of this would be if commands to read and write to a particular file are received at the same time. This could result in the overwriting of data before it has been read, the return of incorrect data because it was read prior to writing the updated information, or worse yet, an operating system crash. Utilizing file locking mechanisms and asynchronous responses to called programs can prevent these race conditions from occurring.

Another specific type of race condition called “time of check to time of use” (TOCTTOU) can be created when using temporary files, or session variables. If the temporary file or session variable does not have secure permissions, it could be altered between the time it is created and the time the program later reads from it or writes to it again creating opportunities for exploitation.

The application must have exclusive rights to any files and data for which it relies on for information. Alterations and concurrent access cause unexpected program execution and vulnerabilities in security.

Principle of Secure Memory Management

Buffer overflows occur when arbitrary code is injected into assigned application buffer spaces causing unwanted executions of applications or malicious code. Buffer overflows can be prevented by making sure that bounds checking is done on the length of input variables, arrays, and arguments.

Memory pointers can also be used to execute malicious code or programs. Because memory pointers can be pointed at any memory structure or location, they enforce no ownership of memory and therefore can end up pointing at anything. It's common in attacks on the Windows OS to put malicious code at an address that a pointer references because the OS cannot tell that the code there is foreign. Minimizing the use of memory pointers and destroying them after they are done being used will help to reduce this security threat.

Buffer overflows and memory pointer exploits are of the most common system exploits, as well as the most dangerous. Not only can these exploits cause an application to act unexpectedly, but they can also compromise the security of the entire computer the application is run on, and possibly an entire network if the PC is connected to an unsecured LAN.

Principle of Simplicity

Coding standards are essential to secure programming. Coding standards keep things simple, ensure that security is implemented in the program, and provides the assurance that other developers can understand the code with the least amount of confusion.

The most important ideal to remember when designing a secure application is simplicity. Creating the simplest solution to a problem usually means it will create the least amount of security issues. Overly complex code is much more difficult to secure. Keep in mind that in developing future versions of the application, or during bug fixes, developers who didn't initially write the code may be the ones trying to secure it.

Your code should not be secure by obfuscation because with enough time and determination, the obfuscation can be deciphered and exploited. Rather the security in the program should be very transparent and even modularized. This will allow the code that implements the security and has been proven to be secure to be reused in other areas of the program, and even in other software systems.

Principle of Data Protection

Hard-coding passwords into programs is probably the worst coding sin a developer can commit. Developers should never hard-code passwords into programs. If passwords are hard-coded into programs, it is possible that unauthorized users can discover them using sniffers or protocol analyzers, or even a simple Hex editor.

Also, anytime a password or secure data is transmitted, the data should be encrypted with either a one way hash or some type of strong data encryption algorithm. An MD5 hash is a good example of this.

Principle of Distrust

All input is evil! This simple assumption will save you a lot of grief in the long run. Trusting a user to act in an expected manner is setting the security of the system up for failure. Because you cannot identify every type of input that may come your way, only allow what you know to be good through. This means that at every opportunity for the user to provide a value, click a button, open a file, or

any other user interaction with the program, the input must be scrubbed. Until it is proven secure, the input cannot be trusted.

The simplest way to test user input is through simple exception handling. The means of doing this will change depending on the language you are using. Most object oriented languages employ a try/catch model of exception handling. Combine this with the use of regular expressions. Often times these two steps will stop all sorts of attacks including:

- Data format exceptions
- SQL injection attacks
- Buffer overflow attacks
- Stack pointer switches

The ultimate goal of secure programming is detecting and recovering from any unforeseen situation. This means that if an error does occur during program execution, whether it was caused by bad user input or not, the application should handle the error in a kindly manner, or gracefully exit.

One thing to keep in mind when notifying a user that an error occurred is to not give away any information. A common example of this is with user login information. If an application tells the user that his password is incorrect when a correct username but a wrong password is entered and a different error message entirely when both are incorrect, this tells a malicious user that they have guessed a correct username. It may simply be a brute force guessing process to get a correct password; Hence, compromising the security of the entire system.

Lastly, the program should be secure by default. If an error or exception occurs and is caught, part of the graceful error handling should be to close open ports and database connections, release control of open files, and to overwrite the memory locations of secure data with null values.

S.T.R.I.D.E Principles of vulnerability

The acronym S.T.R.I.D.E describes all the different vulnerabilities a computer system may face. S.T.R.I.D.E stands for

- Spoofing
- Tampering
- Repudiation
- Information disclosure
- Denial of service
- Elevation of privilege

Spoofing refers to various forms of falsification of data. Simply put, spoofing is pretending that you're someone you're not (i.e. someone trusted). Spoofing can

take the form of using valid login information of a trusted person, a man in the middle attack where the communication between two entities is intercepted in both directions, or by changing HTTP referrer headers essentially fooling the system into thinking you are trusted.

Tampering simply means changing data. If tampering occurs through any means, security has been breached. Often tampering occurs when either an invalid user gains access to the system, or when information is intercepted in transit. Cryptographic hash functions and cryptographic signatures can be used to add a tamper-evident layer of protection to the data, often referred to as an electronic signature. A signed hash or Hex number is generated from the contents to be stored or transferred, and any change to the data, no matter how trivial, will cause it to have a different hash, which will make the signature invalid.

Repudiation simply means denial. This relates to secure coding principles usually through denial of responsibility. Repudiation must be prevented in all secure transactions. Proof of the integrity and origin of data and in the authentication of all users of a system are ways to prevent repudiation. This is especially important in eCommerce, legal contracts and exchanges, and other types of monetary transaction. Often times repudiation is used to circumvent the terms of a contract, and if identity cannot be proven the contract cannot be upheld.

Prevention of information disclosure is the main priority of secure coding. The principles and methods of doing so have already been discussed in this document.

Denial of service (DoS) attacks are all too common. A DoS is an attack on a computer system or network that causes a loss of service to users, typically the loss of network connectivity and services by consuming the bandwidth of the victim network or overloading the computational resources of the victim system. Denial of Service attacks can also lead to problems in the network 'branches' around the actual computer being attacked. For example, the bandwidth of a router between the Internet and a LAN may be consumed by a DoS, meaning not only will the intended computer be compromised, but the entire network will also be disrupted.

Elevation of privilege occurs when a trusted but restricted user gains access to data or programs which are meant only for system administrators or more privileged users. This obviously presents the same problems as the other exploits mentioned here. The exploits that a user uses to obtain an elevation of privilege cannot always be predicted. However like any other secure coding principle, this is one more thing to be aware of in the design and development process.

Conclusion

The secure programming principles need to be designed into your application from the very start and continuously evaluated throughout the development process. Not only does it make the applications being designed more robust and flexible, it ensures the safety and security of all programs, data, and the computer system or network. The basics of secure programming are to trust no user or input until it can be proven secure; common exploits must be known in order to take measures to prevent them; data protection is the most important purpose of secure coding; and if security cannot be ensured, then the program needs to take steps to recover or gracefully exit.

Hopefully, this has simplified what is to most people an incredibly complex subject. The fact-of-the-matter is that most beginning security mistakes can be solved by discontinuing lazy coding practices and creating a standard that is always followed. Most of today's security problems are caused by a combination of design flaws, poor programming standards, and programmer error.

This document is for informational purposes only. Tometa Software, Inc. MAKES NO WARRANTIES, EXPRESS OR IMPLIED, IN THIS DOCUMENT.

© 2004 Tometa Software, Inc. All rights reserved.