



Code (In)Security

Mano Paul, CSSLP, CISSP, AMBCI, MCAD, MCSA, Network+, ECSA

Introduction

Past (ISC)²® published whitepapers have discussed the basics of software security as it pertains to various stakeholders. In response to reader requests, this whitepaper will look at software security from the perspective of a software developer.

In the book, *Hacking: the Art of Exploitation*®, author Jon Erickson accurately and succinctly expresses that an understanding of writing code helps those who exploit it, and an understanding of exploitation helps those who write code. A software developer, in other words, must first know how their code can be exploited (insecure code) and then, in turn, use that knowledge to write code that is not exploitable (code in security). Just as when attempting to cure a malady, a physician needs to first diagnose the core issue before treating the symptoms, when developing hack-resilient software, one must first understand what constitutes insecure code, before attempting to address vulnerabilities.

Introducing Code (In)Security

It must be stated from the outset, however, that software security is more than writing secure code. In today's security landscape, considerations must go beyond mere functionality to take into account security as well. There are several noteworthy resources on software security; some of those worth mentioning are the *Hacking Exposed* series, *19 Deadly Sins of Software Security*, *Exploiting Software*, *Building Secure Software*, and *Writing Secure Code*. But even though these resources are considered must-reads for any software developer, evidence from sources such as the *Chronology of Data Breaches*® and security bug- and full disclosure-lists show that software applications produced today are still rife with vulnerabilities. Security breaches are merely the symptoms of insecure design and programming, and unless software developers are trained to architect secure software and identify what constitutes insecure code, the trend of software rampant with security weaknesses is likely to continue.

There is a lingering debate about who is ultimately responsible for software insecurity. Opinions vary. Is it the software publishers, or the organization as a whole? Some feel that the blame should rest with the coder. But without proper education being imparted to software coders as to how to write 'secure' code or how to not write 'insecure' code, it is unreasonable to put the blame totally on them.

It is the opinion of this author that software insecurity must be attributable to *all* stakeholders in the software development lifecycle, and software developers, who write code, can play a vital role in the development of secure software.

“Software insecurity must be attributable to all stakeholders in the software development lifecycle, and software developers, who write code, can play a vital role in the development of secure software.”

INSECURE Code

As the Chinese adage goes, a journey of a thousand miles begins with a single step. The journey to develop secure software begins with the first step of identifying what makes up insecure code.

So what is insecure code? Insecure code is code which is vulnerable to security attacks. For the benefit of the reader, the word “insecure” itself may be used as an acronym, to describe programming constructs and code that are vulnerable to security breaches. The following is by no means an all-inclusive list of everything that constitutes insecure code, but is a compilation of the most prevalent programming constructs that have been observed to render software insecure.

Characteristics of Insecure Code

Characteristic	
I	Injectable Code
N	Non-Repudiation Mechanisms not Present
S	Spoofable Code
E	Exceptions and Errors not Properly Handled
C	Cryptographically Weak Code
U	Unsafe/Unused Functions and Routines in Code
R	Reversible Code
E	Elevated Privileges Required to Run

I - Injectable Code

Injectable code is code that makes the infamous and prevalent injection attacks possible, allowing the user-supplied data to be executed as code. There are various types of injection attacks: attacks against databases (e.g., SQL injection), attacks against directory structures (e.g., LDAP injection), and even attacks against the operating system itself (e.g., OS command injection). Improper validation or filtration leads to injectable code attacks.

Lack of validation of user-supplied input is the primary reason why injection attacks are possible, resulting in serious consequences. Breaches such as the disclosure of sensitive information (confidentiality breach), tampering and manipulation of directory structures and trees (integrity breach), denial of service (availability breach), authentication and authorization check bypass, and even remote code execution are possible.

Coding constructs that depend on the user-supplied data to build the queries to be executed in the backend on-the-fly (dynamically) is another possibility that should not be overlooked.

There are many defensive coding techniques against injectable code. One option is input sanitization, which can be achieved either by allowing only a certain set of valid inputs, or disallowing and removing any invalid input patterns and characters.

A second option is the use of parameterized queries, those that are not dynamically generated. These take the user-supplied input as parameters. When architected correctly, they also aid with performance. Coding standards should disallow the dynamic construction of queries in code to mitigate injection attacks and this must be enforced.

"All Input Data is Evil-So Make Sure You Handle It Correctly and with Due Care"⁶ by Dino Esposito, published in CoDe magazine is a good reference article and as the latter half of the title rightfully suggests, it is extremely important to be able to identify injectable code and take necessary steps (due care) to address it.

N - Non-Repudiation Mechanisms not Present

In a highly interconnected and mobile world, it is imperative that the authenticity of code origin, and critical business and administrative transactions be indisputable. Non-repudiation is the guarantee that the origin of code is what it claims to be. It is the ability to verify the authenticity of the code's origin. This is particularly important because code can be downloaded from a remote location and executed on a local system. Referred to as "mobile" code, it must carry proof of its origin to verify authenticity, which can be achieved by signing the code. Code signing is the process of digitally signing code (executables, scripts etc.), to assure that the code has not been tampered with, and that the code is from a valid software publisher. Code signing is also known as digital shrink-wrapping.

Non-repudiation also applies to ensuring that the actions taken by the code cannot be denied.

Auditing functionality in code is a mechanism to ensure that repudiation is not possible. At a bare minimum, for critical business and administrative transactions, the code must be written to record the action taken, including the timestamp and other pertinent details such as the user or process that is performing the action.

S - Spoofable Code

Spoofing is the act of impersonating another user or process. Spoofable code is code that allows for spoofing attacks. Spoofing attacks allow the impersonator to perform actions with the same level of trust as a valid user, who has been impersonated. Disclosure of information to unintended users, tampering of data, resource exhaustion, bypassing authentication, circumventing authorization checks, deletion or manipulation of audit logs are all potential consequences of spoofing.

Spoofing has been observed in cases where the code base is not segmented to run under different execution contexts based on the trust (privilege) levels of the caller of the code. This violates the principle of "least common mechanism" which states that mechanisms common to more than one user/process are not shared⁷. With just one execution context for all code, an attacker can spoof an identity and execute the code as if the attacker was a valid user with permission.

Predictable session identifiers, hard-coded passwords, caching credentials, and allowing identity impersonation are common coding vulnerabilities that can result in spoofing attacks. Figure 1 is an example of a configuration entry that allows impersonation of a specific user. In addition to the fact that impersonation is allowed, the username and password is hard-coded inline in plain text, which is also not recommended. Figure 2 is an example that illustrates how an authenticated user's identity is impersonated in code.

Spoofable code can lead to several security compromises, the most common of which is session hijacking and replay attacks. An attacker can impersonate the identity of a valid user and take control over an established session between the client and the server and then replay the action.

In the event that there may be a valid business reason to allow for impersonation, such actions must be closely monitored and audited. Care must be taken to ensure that code written does not allow for impersonation and spoofing attacks.

E – Exceptions and Errors not Properly Handled

Any software developer understands that it is very hard to make software code error free. Exceptions and errors are inevitable. However, not handling exceptions and errors, or handling them improperly, are unacceptable options when it comes to software security.

Figure 1. Configuration setting to impersonate a specific user

```
<identity impersonate="true" userName="johnD03" password="p@ssw0rd"/>
```

Figure 2. Impersonating the authenticated user in code

```
//Creates a impersonation context object
System.Security.Principal.WindowsImpersonationContext impersonationCtxt;
//Sets the impersonation context object to that of the authenticated user (User.Identity)
impersonationCtxt = ((System.Security.Principal.WindowsIdentity)User.Identity).Impersonate();
//Code runs under the security context of the authenticated user
```

Figure 3. Error message discloses account name exception details

Login failed for SQL Server login 'SecuRiskLabUser'. The password for this login has expired.

Description: An unhandled exception occurred during the execution of the current web request. Please review the stack trace for more information about the error and where it originated in the code.

Exception Details: System.Data.SqlClient.SqlException: Login failed for SQL Server login 'SecuRiskLabUser'. The password for this login has expired.

Source Error:

```

Line 37:      SqlDataAdapter _oSqlDA = new SqlDataAdapter(_sCmdText, _oSqlConn);
Line 38:      DataSet _oDS = new DataSet();
Line 39:      _oSqlDA.Fill(_oDS);
Line 40:
Line 41:      //      SqlDataAdapter _oSqlCmd = new SqlDataAdapter("SecuRiskLab_GetLoginInfo", _oSqlConn);

```

Source File: c:\Users\Man_Paul\Documents\Visual Studio 2008\WebSites\SecuRiskLab\SignIn.aspx.cs **Line:** 39

Stack Trace:

```

[SqlException (0x80131904): Login failed for SQL Server login 'SecuRiskLabUser'. The password for this login has expired.]
  System.Data.SqlClient.SqlInternalConnection.OnError(SqlException exception, Boolean breakConnection) +800131
  System.Data.SqlClient.TdsParser.ThrowExceptionAndWarning(TdsParserStateObject stateObj) +186
  System.Data.SqlClient.TdsParser.Run(RunBehavior runBehavior, SqlCommand cmdHandler, SqlDataReader dataStream, BulkCopySimpleRe
  System.Data.SqlClient.SqlInternalConnectionTds.CompleteLogin(Boolean enlistOK) +33
  System.Data.SqlClient.SqlInternalConnectionTds.AttemptOneLogin(ServerInfo serverInfo, String newPassword, Boolean ignoreSniOpe
  System.Data.SqlClient.SqlInternalConnectionTds.LoginNoFailover(String host, String newPassword, Boolean redirectedUserInstance
  System.Data.SqlClient.SqlInternalConnectionTds.OpenLoginEnlist(SqlConnection owningObject, SqlConnectionString connectionOpti
  System.Data.SqlClient.SqlInternalConnectionTds..ctor(DbConnectionPoolIdentity identity, SqlConnectionString connectionOptions,

```

Code that reveals verbose details is an example of improper handling of exceptions and errors.

A simple example of a verbose error is "Username is invalid" during an authentication attempt. Even a simple message such as this is more information than necessary. A message such as "Login invalid" is sufficient. This non-verbose error message now leaves the attacker to guess whether it is the username or the password which is invalid, unlike in the previous case where the attacker knows that it is the username which is invalid. Non-verbose error messages and not displaying raw exception details will considerably increase the work factor for the attacker who is trying to break into your system. This can, however, have a negative impact on troubleshooting and customer support, and so design considerations should be factored in so that the software will display the appropriate verbosity of error messages, without revealing the details.

Improper handling of exceptions and error details can lead to disclosure of internal architecture, data flows, data types, data values, and code paths. During a reconnaissance exercise, an attacker can use the information gleaned from a verbose error message, or in the exception details, to profile the software. Figure 3 illustrates how an unhandled exception discloses that there is a login account name SecuRiskLabUser, besides revealing the exception details stack trace which can provide an attacker information that they can use to profile the software.

The absence of a catch-all exception-handling routine, and merely bubbling the raw exception information to the front end or client, is another example of improper exception or error handling. When an exception is caught, the code must handle the exception explicitly.

Additionally, in the event of failure, assets should not be put at risk, which is the principle of fail-safe or fail-secure. Decisions must be based on explicit permissions instead of exclusions.

It is important to ensure that errors and exceptions are handled so that they are non-verbose, do not reveal more information than is necessary, and do not violate the principles of fail-safe or fail-secure.

C – Cryptographically Weak Code

Developers are essentially creative problem solvers who use their skills and technological know-how to create solutions to solve business problems and needs. Developers seek to improve existing functionality. Unfortunately, this has been known to backfire, especially in the context of cryptography, as evidenced by the various poor custom cryptography implementations observed in code reviews.

Such reviews reveal that cryptographic functionality in code is, more often than not, custom-developed, rather than developed by leveraging existing proven and validated standard cryptographic algorithms. This contradicts the secure design principle of “leveraging existing components” to minimize the attack surface.

Additionally even when proven cryptographic algorithms are used, implementation detail have been found to be insecure. Cryptography algorithms use a secret value, known as a key, to encrypt (convert plain text to cipher text) and decrypt (convert cipher text to plain text). The key to the strength of a cryptographic implementation is not necessarily the strength of the algorithm itself, but the way the key is derived and managed. Using non-random numbers to derive the cryptographic keys renders the cryptographic protection weak and ineffective. Sometimes ASCII passwords that are easily guessable and non-random are used to derive the cryptographic key, which should be avoided. Another common problem is that keys are not stored in a secure manner. Keys have been observed to be hard-coded inline with the code. This is akin to locking your doors and leaving the keys in the lock, thus providing minimal protection if any protection at all.

Special attention must be paid when choosing the algorithm to see what the exploitability of the algorithm has been. Once chosen, Random Number Generators (RNG) and Pseudo-Random Number Generators (PRNG) must be used to derive the cryptography key for encryption and decryption. Derived keys must also be stored in a secure manner.

U – Unsafe/Unused Functions and Routines in Code

Unsafe functions are those that have been found to be inherently dangerous. These functions were developed without necessarily taking into account the security implications. The use of these functions gives the programmer no assurance of security protection. They can result in vulnerabilities that could allow an attacker to corrupt the system's memory and/or gain complete control over a system. One of the reasons attributable to the infamous buffer overrun attack is the use of unsafe functions in code. Several security bulletins and patches have been published to address these functions.

Unsafe functions are predominantly seen in legacy and older generation programming languages. Two common examples of these functions in the C programming language are *strcpy* and *strcat*. Since these functions do not perform length/size checks (also known as bounds checking), an attacker can supply inputs of arbitrary sizes that can overflow memory buffers. Figure 4 is an example of the unsafe C function *strcpy* being used to copy the input data into a local memory buffer. If there is no bounds checking, and the user-supplied input has more characters than the local memory buffer can hold, the result will be an overflow of the local memory buffer.

Figure 4. Use of unsafe 'strcpy' C function

```
void CopyData (char* inputData)
{
    char localBuffer [4];
    strcpy (localBuffer, inputData);
}
```

Today, software publishers of programming languages are avoiding unsafe functions in favor of safer alternatives such as the *strncpy* and *strncat*, which allow the developer to perform length/size checks.

Another insecure coding construct, observed in code reviews, is the presence of unused functions where redundant code that no longer addresses any business functionality is left remaining. Changes in the business and advancements in technology and the apprehension of breaking backward compatibility are reasons unused functions might be left in the code. This increases the relative attack surface of the code.

Another classic example of unused functions in code is something known as “Easter eggs”. An Easter egg in software is secretly hidden code that can result in a change of program behavior (e.g., displaying a message, playing a sound, etc.) when particular conditions (e.g., a series of mouse clicks, keystrokes, etc.) are met. Easter eggs are usually innocuous, but they also increase the attack surface of the code and hence can have potentially serious security consequences. Risk of losing customers, introducing new bugs, and performance impacts leading to resource exhaustion, i.e., availability impact, (along with the risk of looking like an

eggomaniac!) are some of the detrimental effects of Easter eggs. IT World's article "Favorite software Easter Eggs"^e is an interesting article that highlights the sinister side of Easter eggs in software.

It is a best practice to minimize the relative attack surface of the code, and this means avoiding the use of unsafe functions, removing unused functions that cannot be traced to a requirements traceability matrix, and avoiding coding in Easter eggs.

R – Reversible Code

The acclaimed IEEE paper entitled "Reverse Engineering and Design Recovery: A Taxonomy", defines reverse engineering as "the process of analyzing a subject system to identify the system's components and their interrelationships and create representations of the system in another form or at a higher level of abstraction." In simpler terms, software reverse engineering, or reversing, is the process of going backwards from system or code to determine the internal design and implementation details. Reversing can be performed on software at the system level or at the code level.

Un-obfuscated or unsigned code is easily reversible. Obfuscation of code is the process of making the code functionality difficult to understand, and is sometimes referred to as "shrouded" code. Obfuscation is convoluting the code so much that even if you have its source, it cannot be understood. Programs known as obfuscators operate on source code, object code, or both, mainly for the purpose of deterring reverse engineering. Figure 5 depicts an un-obfuscated and an obfuscated version of a simple print HelloWorld program^f. Signing code (discussed earlier) is another

anti-reversing technique, which offers more protection than obfuscation. Checking for the presence of debuggers in code and tricking de-compilers using non-instruction data or junk bytes^g are other anti-reversing techniques.



It must be recognized that all of these anti-reversing techniques can only deter reversing, not necessarily prevent it. But it is imperative that code be protected as much as is possible from the risks of reversing. Eldad Eilam's book, *Reversing - Secrets of Reverse Engineering*, is an excellent resource and a must-read for anyone interested in writing irreversible code.

E – Elevated Privileges Required to Run

The principle of least privilege states that a user or process is given explicitly only the necessary, minimum level of access rights (privileges) for the minimum amount of time required to complete the user's or process's operation. Sometimes, however, code that runs in a less secure development environment experiences hiccups or fails to run when deployed into a more restrictive production environment. The usual solutions in such cases are to increase the privilege level under which the code can execute, or remove the checks for privilege levels. Neither of these "solutions" is recommended from a security standpoint. They could lead to circumventing permissions, allowing users and processes with lower privilege to execute code that they are not authorized to.

An additional example is code that can be explicitly set to run with elevated (administrative) privileges programmatically, another classic indicator of insecure code.

Figure 5. HelloWorld program before and after code obfuscation

<pre>public class HelloWorld { public static void main(String args[]) { System.out.println("Hello World!"); } }</pre>		<pre>public class HelloWorld { public static void main(String args[]) { double d1 = 0.0134654879927; double d2 = 0.0234987519084; for (int i1 = 0; i1 < 72; i1++) { d1 = d2 + 0.00000001020102; } for (int i1 = 0; i1 < 59; i1++) { d2 = d1 + 0.00000001120102; } //System.out.println(d1+d2); if ((d1+d2) > 0.04699753441986) { System.out.println("Hello World!"); } else if ((d1+d2) < 0.04699753441186) { System.out.println("Goodbye World!"); } //This chain of alternatives could go on for a //long time... } }</pre>
		

Configuration differences between the development and production environments must be ensured so that code can execute with least privilege, irrespective of the environment. It is also imperative to ensure that code explicitly set to run with elevated privileges is carefully monitored (audited) and managed.

Conclusion

'Don't write insecure code' is one of the eight secure habits discussed in "8 Simple Rules for Developing More Secure Code".¹ Without a thorough understanding of what constitutes insecure code, it would be unfair to expect developers to write more secure code.

"It is critical to all those who write code to make it a habit to incorporate security in the code they write."

Should your code fall victim to a security breach, the code itself would be deemed innocent, but that kind of leniency will not be extended to the individual, team, or organization that wrote the code. So it is critical to all those who write code to make it a habit to incorporate security in the code they write.

Secure Code Characteristics

- Validates input
- Does not allow dynamic construction of queries using user-supplied data
- Audits and logs business-critical functions
- Is signed to verify the authenticity of its origin
- Does not use predictable session identifiers
- Does not hard-code secrets inline
- Does not cache credentials
- Is properly instrumented
- Handles exceptions explicitly
- Does not disclose too much information in its errors
- Does not reinvent existing functionality, and uses proven cryptographic algorithms
- Does not use weak cryptographic algorithms
- Uses randomness in the derivation of cryptographic keys
- Stores cryptographic keys securely
- Does not use banned APIs and unsafe functions
- Is obfuscated or shrouded
- Is built to run with least privilege

"Writing secure code has the added benefit of producing code that is of quality – functional, reliable, and less error-prone."

It would be far beyond the scope of this paper to enumerate all the ways that one can avoid writing insecure code, and/or write secure code. Instead, this should be treated as merely an eye-opener to how to code securely, and the tremendous risks of not doing so.

Just as a game of chess produces an infinite number of moves once the game begins, with a limited number of opening gambits, understanding the characteristics of insecure code is one of the first moves to ensure that code is written to be hacker-resistant. Insecure code means checkmate.

Summary of Insecure Concepts

Characteristic		What is it?	Insecure Code Examples	How to Fix It
I	Injectable Code	Code that makes injection attacks possible by allowing user supplied input to be executed as code.	No input validation, Dynamic construction of queries	Input Validation, Parameterized queries
N	Non-Repudiation Mechanisms not Present	Authenticity of code origin and actions are disputable.	Unsigned executables, Auditing not present	Code Signing
S	Spoofable Code	Code that making spoofing attacks possible.	Predictable session identifiers, hard-coded passwords, caching credentials and allowing identity impersonation	Session, Cache and Password Management, Managing identity impersonation
E	Exceptions and Errors not Properly Handled	Code that reveals verbose error messages and exception details, or fails-open in the event of a failure.	Verbose errors, Unhandled exceptions, Fails open	Non-verbose error messages, Explicit exception handling (Try-Catch-Finally) blocks, Fail-secure
C	Cryptographically Weak Code	Code that uses non-standard, weak or custom cryptographic algorithms and manages key insecurely.	Key not derived and managed securely	Do not use weak, non-standard algorithms, custom cryptography, Use RNG and PRNG for key derivation.
U	Unsafe/Unused Functions and Routines in Code	Code that increases attack surface by using unsafe routines or containing unused routines.	Banned API functions, Easter Eggs	Do not use banned APIs unsafe functions, Input validation, remove unused routines and Easter eggs.
R	Reversible Code	Code that allows for determination of internal architecture, design.	Unobfuscated code, Unsigned Executables	Code obfuscation (shrouding), Digitally signing code
E	Elevated Privileges Required to Run	Code that violates the principle of least privilege.	Administrative accounts	Environment configuration, Code set explicitly to run with least privilege

About (ISC)²[®]

The International Information Systems Security Certification Consortium, Inc. [(ISC)²[®]] is the globally recognized Gold Standard for certifying information security professionals. Founded in 1989, (ISC)² has now certified over 60,000 information security professionals in more than 130 countries. Based in Palm Harbor, Florida, USA, with offices in Washington, D.C., London, Hong Kong and Tokyo, (ISC)² issues the Certified Information Systems Security Professional (CISSP[®]) and related concentrations, Certified Secure Software Lifecycle Professional (CSSLP^{CM}), Certification and Accreditation Professional (CAP[®]), and Systems Security Certified Practitioner (SSCP[®]) credentials to those meeting necessary competency requirements. (ISC)² CISSP and related concentrations, CAP, and the SSCP certifications are among the first information technology credentials to meet the stringent requirements of ANSI/ISO/IEC Standard 17024, a global benchmark for assessing and certifying personnel. (ISC)² also offers a continuing professional education program, a portfolio of education products and services based upon (ISC)²'s CBK[®], a compendium of information security topics, and is responsible for the (ISC)² Global Information Security Workforce Study. More information is available at www.isc2.org.

About the Author

Mano Paul, CSSLP, CISSP, AMBCI, MCAD, MCSD, Network+, ECSC is CEO and President of Express Certifications and SecuRisk Solutions, companies specializing in professional training, certification, security products and security consulting. His security experience includes designing and developing software security programs from Compliance-to-Coding, application security risk management, security strategy and management, and conducting security awareness sessions, training, and other educational activities. He is currently authoring the Official (ISC)² Guide to the CSSLP, is a contributing author for the Information Security Management Handbook, writes periodically for Certification, Software Development and Security magazines and has contributed to several security topics for the Microsoft Solutions Developer Network. He has been featured in various domestic and international security conferences and is an invited speaker and panelist in the CSI (Computer Security Institute), Catalyst (Burton Group), TRISC (Texas Regional Infrastructure Security Conference), SC World Congress, and the OWASP (Open Web Application Security Project) application security conferences. He can be reached at mano.paul@expresscertifications.com or mano.paul@securisksolutions.com.

-
- a Erickson, J, Hacking: The Art of Exploitation, No Starch Press. ISBN 1593270070
 - b The Chronology of Data Breaches.
<http://www.privacyrights.org/ar/ChronDataBreaches.htm>
 - c CoDe Magazine – All Input is Evil- So Make Sure You Handle It Correctly and with Due Care.
<http://www.code-magazine.com/article.aspx?quickid=0705061>
 - d Saltzer, J.H and Schroeder, M.D, The Protection of Information in Computer Systems.
<http://web.mit.edu/Saltzer/www/publications/protection/>
 - e Favonte Software Easter Eggs, IT World.
<http://www.itworld.com/print/64689>
 - f Chikofsky, E. J & Cross, J. H Reverse engineering and design recovery: A taxonomy. IEEE Software, 7(1) 13-17, January 1990.
 - g HelloWorld Obfuscation Image Reference
<http://oreilly.com/pub/a/mac/2005/04/08/code.html>
 - h Eilam, E., Reversing: Secrets of Reverse Engineering, Wiley ISBN 0764574817
 - i 8 Simple Rules For Developing More Secure Code., Microsoft Developer Network (MSDN)
<http://msdn.microsoft.com/en-us/magazine/cc163518.aspx>