Global Information Assurance Certification Paper

CAUSES AND RECOMMENDATIONS TO RESOLVE VULNERABILITIES THAT CAN ARISE IN SECURITY CODE REVIEWS

GIAC GSEC Practical
Version 1.4
Option 1
April 15, 2003
By: Angela M. Goodwin

TABLE OF CONTENTS

CAUSES AND RECOMMENDATIONS TO RESOLVE VULNERABILITIES THAT CAN ARISE IN SECURITY CODE REVIEWS

ABSTRACT:

There are many information technology systems developed without security in mind. Security risks can exist in numerous forms and attack the information system in various ways. Systems connected to the Internet are especially vulnerable because of the global access to the Internet. For this reason, programmers for Web-enabled applications should be especially sensitive to the specific issues that are associated with this medium. Typically, a security code review consists of five major steps: application background review, compiling an analyst review sheet, reviewing source files, documenting the findings, and presenting the findings to the client. This document gives a broad overview of the process used by security analysts when conducting a security code review. This acts as background for the main focus of the document, which is to provide explanations of vulnerabilities commonly found in an application's security code review report. The document will assist programmer(s) with understanding of the findings, why they are considered vulnerabilities and recommendations for the resolving these issues in the follow-up review.

This document presents an overview of major security vulnerabilities commonly revealed through security code reviews. The vulnerabilities in this document are not programming language specific and do not include all potential vulnerabilities; instead this paper introduces a snapshot of the most common vulnerabilities associated with a code review.

1.0 SECURITY CODE REVIEW PROCESS

When the programmer(s) receives a security code review report, it is important that the programmer(s) understand the underlying process used by the security analyst identifying vulnerabilities. This knowledge of the code review methodology will assist programmer(s) with implementing the source code changes as outlines in the code review findings summary. This section defines security code review is and provides a general overview of a generic code review process.

1.1 WHAT IS A SECURITY CODE REVIEW?

A security code review is the process of analyzing an application's code base to identify security vulnerabilities that could lead to the loss of an application's integrity, confidentiality, or availability. Many methodologies vary within companies that provide security code review services. There are three broad categories of code review methodologies: manual source code analysis, semi-automated analyses, and fully automated analyses.

- **Manual Source Code Analysis –** This methodology consists of manually scanning each source code file, line by line. Prior to a visual scan of the source files, an analyst(s) uses sizing, metrics, and textual-search tools. The sizing tool counts the lines of code in each file; the metrics tool determines the code complexity; and the textual-search tool is a keyword search of high-risk malicious words or phrases. The advantages to this methodology are a greater understanding of the application's logical flow and a professional perspective on the exploitable vulnerabilities in the code. The disadvantages include eyestrain, oversight of potential issues due to fatigue, time to review, and cost.

- **Semi-Automated Analysis –** This methodology uses software tools designed to assist in the code inspection process. This software typically uses a combination of target word lists, database, graphical user interface, and report generation engine. The target word list uses a data repository to scan source code; and when a target word is found, a visual inspection of the word and its context in the application is reviewed by the analyst(s). The database tracks the target words and associated source code in the files. The graphical user interface gives the analyst(s) a means of searching, reviewing, and tagging the source files. The report generation engine includes the sizing and metrics of the entire application. The advantages of this approach are the efficiencies and traceability of the manual process and can be combined with fully automatic programs to identify threats. The only disadvantage is that it relies on manual review effort.

- **Fully Automated Analysis –** This methodology relies entirely on the program to test and report known coding vulnerabilities and malicious code. The advantages include the use of known, documented principles as their basis of identifying vulnerabilities and the ability to discover new attacks or mechanisms based on hostile characteristics. The disadvantage is the loss of human insights.

The scope of this document is limited to the manual source code analysis methodology. The backbone of automated analysis is the review process of the manual source code review. The tools used as part of both, semi-automated and fully automated review are built with the manual source code review process in mind.

1.2 APPLICATION BACKGROUND REVIEW

Understanding the application that is under analytical scrutiny is one of the most important elements of a security code review. This includes understanding the application's purpose, origin, and intended use. The analyst(s) must understand the type of environment in which the application will be placed. The potential for vulnerabilities is also closely related to the number of users and their level of knowledge. Frequently, internal users input incorrect information, due to lack of application knowledge, thus resulting in potential exploitable vulnerabilities. The client should provide the analyst(s) with all applicable documentation including application information from test plans and results, white papers, user interface diagrams, process flow diagrams, UML class diagrams, system specifications, and user manuals.

However, in many cases, the client does not have access to many of these documents. Sometimes, the documents do not exist because of poor application development methodology. After all system documentation is reviewed and the analyst(s) has a clear understanding of the application, the analyst(s) review sheet is ready to be compiled. The following is a description of each form of documentation, and its relevance to a security code review:

- Test Plans and Results. These test plans and their results provide the analyst(s) with information on testing procedures applied prior to the code review process.
- White Papers. These papers provide insight into the application's intended use, how it will fit into the end-user's environment, and how its output will make current processes more efficient, effective, and/or artificially intelligent.
- User Interface Diagrams. These diagrams demonstrate how end-users will interact with the application. This is extremely important to the analyst(s) because it shows where the application accepts user input. Filtering input is a possible security vulnerability that programmer(s) must address while building the application. These user interface diagrams point to the specific files that need to address input validation and filtering.
- Process Flow Diagrams. These diagrams show how information flows through the application. This is useful to the analyst(s) because it helps them get a better understanding of the application and how data flows. Generally, the exposure of the data is what makes an application insecure. Therefore, it is important to know how the application handles the data.
- UML Class Diagrams. Using JAVA and C++, these diagrams depict a process flow for individual classes used in the application. These diagrams can give the analyst(s) insight into what classes are necessary and if there are any classes that are not being used by the application. If classes exists that are not part of the process flow, they should be discarded.
- System Specifications. These specifications provide a detailed description on the tools used in application development. Also, they include a high-level description of how the application operates.
- User Manuals. These manuals assist the analyst(s) in demonstrating how the user will interface with the application, when user input is accepted, and when application output is expected. These are important elements in determining where it is necessary to include filtering and validation routines.

1.3 ANALYST REVIEW SHEET

The analyst review sheet is a compilation of security risks associated with the programming language of the application. The review sheet is ultimately used to compose a vulnerabilities/risk matrix for the client. The analyst review sheet offers a consistent rating system for each security risk and is applied to each source file. An analyst review sheet must be used when more than one analyst is involved in the security code review. The rating system enables a common technique to be used for evaluating various ways that the system could be compromised. The review sheet is a means to document the findings of each source file. Notes from each source file are

recorded on the corresponding analyst review sheet. In addition, it is important that the reviewer's name is included. If questions arise concerning specific source files, the analysts name and notes for each source file serve as a record in the future. An example of an analyst review sheet can be found at **Appendix A**.

1.4 SOURCE FILE REVIEW

When the analyst(s) and reviewer(s) agree on the analyst review sheet, the files are strategically divided into groups of related source files. This is important because many source files rely on others, and requires the same analyst(s) review those related files. Once the files are divided, the analyst(s) delve into the main concentration of code review. There are general vulnerabilities on which the source files are evaluated. These general vulnerabilities are the focus of this document.

2.0 GENERAL VULNERABILITIES AND RECOMMENDED ACTIONS FOR RESOLVING SECURITY CODE REVIEW FINDINGS

The previous section defines the process of a generic security code review. This section will describe potential vulnerabilities that could appear in the findings section of the security code review report. In the report, ratings will be assigned to the vulnerabilities and can act as a priority level for resolving the security issues. The following is an example a ratings that could potentially be assigned in the finding section:

a) Category I. Findings are vulnerabilities that provide an attacker immediate access into a machine, gain superuser access, or bypass a firewall.

b) Category II. Findings are vulnerabilities that provide information that has a high potential of giving access to an intruder.

c) Category III. Findings are vulnerabilities that provide information that potentially could lead to compromise.

d) Category IV. Vulnerabilities, when resolved, will prevent the possibility of degraded security.

These categories may change based on the threats that could potentially permeate the specific network, application, or local computers where the application resides. Once the programmer(s) defines a priority list for resolving the vulnerabilities, they must have a general understanding of the vulnerabilities and why they are considered risks. The subsequent subsections provide the description of the vulnerabilities and the reason it is considered a security compromise. In addition, recommendations are included to give the programmer(s) a reference for fixing the security vulnerability.

2.1 Input Validation

Input validation is an important element in a security code review, because users may not intentionally try to compromise a system. A mistake by a user can have the same effect as a malicious user intending to corrupt a system. Therefore, validation routines are used to assure data is "clean" before being committed to the database.
One should never trust user input. If input is accepted from users, either directly or indirectly, it is imperative that it is validated before it is used. Input validation is important because malicious users may attempt to make the application fail by tweaking the input to represent invalid data. All input is bad until proven otherwise. Typically, the moment this rule is overlooked, is the moment the system is attacked. It is imperative that indirect and direct user input is validated before it is used. Cross-site scripting, unbounded sizes, and injection attacks are different types of attacks that take advantage of non-validated user input.

Recommendation: It is recommended that all places in the source code that accept form data or end-user input, a routine should be called to verify that the data is in the correct format and within the size guidelines. "A secure program should know what it expects, and reject other input." (Shostack, 5) Write the source code in the manner that assumes the client is a hacker. This mindset will assist in writing secure source code.

The following are different types of attacks against user input:

- **Cross-site Scripting**: A cross-site scripting attack is a security vulnerability that is caused by trusting user input. This attack only applies to Web application. An attacker can send a link in an email to a user or otherwise points the user to a link to a Web site, and a malicious payload is in the query string embedded in the URL. This attack can be used against machines behind firewalls, if the firewall allows the attacker to communicate with the service running on the system on which the application is executed. Many corporate Local Area Networks (LANs) are configured such that client machines trust servers on the LAN, but do not trust servers on the outside Internet. If the attacker obtains the name of a Web server inside the firewall that doesn't check fields in forms for special characters, the application could be compromised. The following is an example of cross-site scripting:

Scripting via a malicious link
In this scenario, the attacker sends a specially crafted e-mail message to a victim containing malicious link scripting such as one shown below:

<A HREF=http://legitimateSite.com/registration.cgi?clientprofile=<SCRIPT>malicious code</SCRIPT>>Click here</A>

When an unsuspecting user clicks on this link, the URL is sent to legitimateSite.com including the malicious code. If the legitimate server sends a page back to the user including the value of client profile, the malicious code will be executed on the client Web browser.

Recommendation: The recommended action to combat this vulnerability is to filter and validate the input received and properly encode output returned to the end-user. Filtering input consists of checking for special characters that are defined in the HTML specification.(Lee, 4) When a special character is found, it will reject the input from the user and prevent the malicious code from being presented to the client. Also, in some cases it is necessary to include additional special characters in the filtering routine to take in account other characters that the application will not properly handle. Although this approach will filter some malicious code, it is important that all dynamic content is filtered just before the data output is sent as a portion of the dynamic page.

An alternative to filtering special characters is encoding each character in the input with its numeric entry value. “Server side encoding is a process where all dynamic content will go through an encoding function where scripting tags will be replaced with codes in the chosen character set.” (Lee, 4)

Generally, encoding is recommended because it does not require the analyst to make a decision about what characters could legitimately be entered and need to be passed through. Unfortunately, encoding all non-trusted data can be resource intensive and may have a performance impact on some Web servers.

- **Unbounded Sizes**: If the size of the client data is unbounded and unchecked, an attacker can send as much data as he/she wants. This could be a security issue if a buffer overrun exists in the program code and is called when invoking the SQL query. An attacker can easily bypass the maximum username and password size restrictions imposed by the previous client HTML form code, which restricts both fields to 32 characters, simply by not using the client code. Instead, attackers can change the client code by saving it as a text file, editing the file, and opening the revised file in a browser. This would allow the attacker to exceed the limits that were originally placed on the input. The following is such an example, which sends a valid HTML form to Logon.asp but sets the password and username to be 32,000 letter "A"s. (Microsoft Press, 6)

```
use HTTP::Request::Common qw(POST GET);
use LWP::UserAgent;

$ua = LWP::UserAgent->new();
$req = POST 'http://www.northwindtraders.com/Logon.asp',
[ pwd => 'A' x 32000,
name => 'A' x 32000,
];
$res = $ua->request($req);
```

Recommendation: A simple field validation rule is the only code necessary to check for an unbounded size. It is good programming practice to check the size of the input and assure that it does not exceed the maximum field size. This will eliminate relying on the user to supply the correct field parameters.

- **Injection Attacks**: Injection attacks happen when user input is used directly in SQL statements. When user input is trusted and has not been checked for validity, an attacker could change the structure of an SQL statement. For example, an attacker would have the ability to log into an application without a password because he/she could "comment out" the intended SQL statement that evaluates the password field. This common attack would allow the attacker to bypass the password validation. The following is an example of making a password field blank by injecting "Angela' --".

```
SELECT count(*)
FROM client
WHERE name='Angela' --and pwd=''
```

Recommendation: Common to all input validation issues, the first recommended rule is to determine which input is trusted and to reject all other input. There are many techniques to reduce the threat of user input. Some additional recommendations for combating injection attacks are using regular expressions, only display user input after it has been scrubbed, avoid using passwords and instead use Server.URLEncode and HttpServerUtility.URLEncode, don't use the system administrator account to log on to the database, and do not formulate SQL statements with direct user input.

2.2 Secure State and Session Management

Once a user has logged in, it is important to track the status and permissions of each user on the server side. The most common methods of tracking the user's session state are cookies, HTTP-BASIC authentication, server side session value storage, URL rewrite, and variables stored in hidden HTML form fields. The purpose of tracking the session state is to prevent "backdoor" / "deep linked" access to parts of a site, and ensures that only users with particular privilege levels are allowed access.

Recommendation: It is recommended to use server-side session variables to track the state of users and restrict unauthenticated users from accesses restricted pages. It is important that each protected page is authenticated by the application before the page loads. It is not recommended to use cookie-based authentication because cookies can be spoofed.

2.3 Sensitive Information in Non-Compiled Code

Sensitive information refers mainly to passwords, algorithms, cryptographic keys, and any information a product uses that is not intended for an end user to view. It is important that this type of information is not implemented into non-compiled code. The public can view non-compiled code. Putting information in this form would defeat the purpose of calling it sensitive. Programmers must keep in mind that information included in comments is viewable to the public. Users can right-click the web page and

select “View Source Code” to obtain view access to the code. Although server-side code will not be in view, comments will be shown.

Recommendation: To secure passwords, algorithms, cryptographic keys, and any information a product uses that is not intended for an end-user to view, it is recommended to never hard-code this information or put it into permanent memory. An alternative is to store the secret in your code in a way that is decryptable only by the programmer. In this case, the secret simply lies in the algorithm used by your code. Another option is to keep the sensitive data in a property file, and read from that file whenever necessary. If the data is extremely sensitive, the application should use some encryption/decryption techniques when accessing the property file.

2.4 Documentation / Comments

Inadvertent security issues can arise if new programmer(s) do not understand the code logic. It is important that the code is well documented. The primary purpose of comments and documentation is to allow for future maintainability to avoid security vulnerabilities being introduced as a result of programmer(s) not understanding the source code’s purpose or function in the application.

Recommendation: There are many guidelines that are recommended to follow when documenting and commenting source code. “Each source file should start with an appropriate header and copyright information.”(Sun Microsystems, 2) In addition, before each function, routine, method, and class in the source code, a comment block should be inserted to describe the functionality and purpose of the function block. Variable names should be descriptive and straightforward. This naming convention should be standard throughout the application. The function parameters that are used for input and output should be properly commented. Complex functions, algorithms and procedures should be properly documented to suffice as a guide for future programmer(s) maintaining the application.

2.5 User Authentication

Unauthorized access to an application can present security vulnerabilities. It is important that access is authenticated at a single entry point into the system. In addition to unauthorized users, it is important that authorized users have the least privileges necessary to conduct daily operations in the application.

Recommendation: It is recommended that a user authentication method be implemented in some form to provide the best protection of the application’s data. Assuring users have the rights and privileges to view information in the application are the main reason authentication is so important. User authentication can be implemented in various ways. The most common technique is a login page where users supply a user name and password to gain entry to the application. Client and server certificates with are passed to authenticate the user are another method of authentication. The

type of network architecture and sensitivity of the application's data will contribute to the authentication method chosen.

2.6 Principle of Least Privilege

The principal of least privilege dictates that the application only has the necessary set of privileges to perform tasks for users. All applications should be executed with the least amount of privileges required to get the job done. If a process runs as system administrator or some other high-privilege account and the process impersonates the user to "dumb down" the code's capabilities, an attacker might still be able to gain administrator's rights by injecting code. For example, a buffer overrun, that calls RevertToSelf, would stop impersonating and reverts to the process identity of system administrator. Therefore, it is important that not even the system administrator be granted system administrative privileges. Many privileges granted to an administrator account are never used and not necessary to be granted.

Recommendation: When determining privileges for authentication, the following are recommended actions to limit security vulnerabilities:

- Users should be granted the least privilege required to accomplish their tasks.
- Applications should be granted the least privilege to perform their functions.
- Systems should be granted the least privilege to fulfill their role in a larger network.

2.7 Test and Debug Code

Debug code is the code that is "commented out" or left over from the debug stages of the software development process. After the completion of the testing phase, instead of removing the code, programmer(s) sometimes "comment" those sections out. Commented code could also represent future functionality that has not been included in the current release. Therefore, some coders "comment out" this specific segment until future product releases require that particular functionality. This code sometimes exposes sensitive information such as user names, passwords, file names, memory locations, etc.

Recommendation: An explanation should be given for any code that is "commented out." Code that has been tested and has no purpose in future development should be removed. If the code is a temporary fix to a problem, it should be commented as such. The programmer(s) should never assume that other programmer(s) would understand why they have commented certain code. In addition, a comment should be included for all codes not completely implemented. The comment should describe why the code is not completed, and what needs to be done to implement the code into the application. The programmer(s) should also include a marker next to incomplete code, so that it is easy to find in the future.

2.8 Malicious Code

Malicious code is software that causes unauthorized or malicious behavior on a computer; such as, viruses, worms, and Trojan horse programs that threaten technology and tools. Source code is commonly “commented-out.” These code segments are either left from a previous implementation or purposely placed inside the program for future releases. If this code is uncommented, it could expose severe security holes. Moreover, malicious data can cause attacks that employ information that appears to represent input for an application program; such as, a word processor or spreadsheet. In reality, these inputs actually represent instructions that are carried out by the computer without the knowledge or approval of the computer's operator. This vulnerability comes from program flaws that allow data to act as instructions despite the programmer(s) intentions. An inspection for this type of security risk is performed to safeguard unauthorized access privileges to private data.

Recommendation: To help mitigate the malicious code risk, main() methods (except the one that launches the application), unused methods, and dead code should be removed from the source code. The programmer(s) should conduct a comprehensive code review of sensitive applications before the software is shipped. Another recommendation is to use "dummy" classes in place of the main() method for testing the functionality of the application. (Sahu, 2)

2.9 Public Variables, Methods, Objects

Objects, classes, methods, variables need to be declared at the lowest scope. Typically, private scope should be used whenever possible. Every class, method, and variable that is not private provides a potential entry point for an attacker.

Recommendation: Generally, it is good programming practice to use private scope whenever possible. Although this is a pain for the programmer(s) and requires more coding, it makes the application more secure. The programmer(s) should use assessor methods rather than public variables. On a case-by-case basis, compare coding convenience and cost against security needs when deciding which variables are material and should be declared private.

2.10 Buffer Overflow/Underflow

Buffer overruns occur when the data provided by the attacker is bigger than what the application expects, and overflows into internal memory space. They are a menace, but generally easy to fix. The key to fixing this vulnerability is assuring that the programmer(s) has taken into account that externally provided data is larger than the internal buffer. The overflow causes corruption of other data structures in memory, and this corruption can often lead to the attacker running malicious code. There are also buffer underflows and buffer overruns caused by array indexing mistake, but they are less common.

Recommendation: Generally, buffer overflows are easy to fix, but the programmer(s) just need to change the way they develop applications. Anytime input is accepted, they must check the input size against the buffer overflow before allocating the memory. The following is a code snippet to check for a buffer overflow:

```
Void CopyString(char *dest, char *source, int length){
        int i=0;
        while((*source) && (i < length)){
                *dest++ = *source++;
                i++;
        }
}
void Example (){
        char            buffer[16];
        CopyString(buffer, "This string is too long!", 16);
}
```

2.11 Error Handling

While the compiler will catch syntax errors, administering a visual inspection of code can be quite difficult without the proper documentation source code. It is also difficult to catch logical errors such as the mistaken use of the "+" sign instead of the "*" sign in the code. Writing error routines are a great coding practice, but it is useless unless it is properly tested.

Recommendation: To assure that error exceptions are handled properly, there should be a call stack function that grabs the error and releases it in the correct error path. It is always good programming practice to include error-handling routines throughout the application, where a system, application, or user error is possible. “Errors should be detected and handled if it affects the execution of the rest of a routine.” (Sun Microsystems, 2) An error would be considered “handled” if it cleanly exits the application or logs the error, depending on the possible actions that an application reverts to when an internal error occurs.

6.0 CONCLUSION

Security code reviews are vital to the safe operational use of applications. Although this document was written to assist the programmer(s) in fixing potential security holes, optimally, these guidelines should be exposed during the planning stage of application development. Application programmers must understand that “security” should be the first and foremost driving factor in the development of an application. They are at the cutting edge of the application development process and can influence the status quo mindset of just “developing and making an application operational”. Implementing a security plan, early in the design phase, can greatly reduce the amount of attacks on Internet applications. After reading this document, the programmer(s) should have a good understanding of why specific source code could potential be an open door for an attacker. Although there are many ways to reduce potential vulnerabilities, above

referenced recommendations should provide a good foundation to identifying weak security practices during the development of application code.

APPENDIX A

Overall Security Level: Confident Concern Uncertain

FOLDER NAME:	_____
FILE NAME:	_____
Lines of Code:	_____

Code Complexity:	Low	Medium	High		
Code Format/Variable Naming:	Safe	Concern			
Documentation / Comments:	None	Lacking	Sufficient	Good	

Session Management:	N/A	Safe	Concern		
Public Declarations:	N/A	Safe	Concern		
Protecting Sensitive Data:	N/A	Safe	Concern		
Principle of Least Privilege:	N/A	Safe	Concern		
Access and Authentication:	N/A	Safe	Concern		

Input Validation:	None	Safe	Concern		
Error Handling:	N/A	None	Lacking	Sufficient	Good
Debug/Commented Code:	None	Safe	Concern		
Malicious Code:	None Found		Concern		
Buffer Overflow:	None	Safe	Concern		

Additional Comments:

Reviewed By: _____

Review Date: _____

REFERENCES

Howard, Michael. “Protect Your Sensitive Web Data from Hackers.” 2001. URL: http://archive.devx.com/security/bestdefense/2001/mh0401-1/mh0401-1.asp .

Howard, Michael and LeBlanc, David. Writing Secure Code. Redmond, Washington: Microsoft Press, 13 November 2001.

Lee, Paul. “Cross-site Scripting.” September 2002. URL: http://www-106.ibm.com/developerworks/security/library/s-csscript/?dwzone=security .

Lussier, Stephane. “Part of Your Complete Breakfast: Code Review is a Source of Essential Vitamins and Minerals.” 20 November 2002. URL: http://macadamian.com/column/completeBreakfast.html .

Macadamian Technologies. “Code Review Checklist.” 2002. URL: http://macadamian.com/codereview.htm.

McGraw, Gary and Felton, Edward. Securing JAVA: Getting Down to Business with Mobile Code. Wiley, 1 March 1999.

McLaughlin, Connie. “Introduction to Buffer Overflows.” 2002. URL: http://vulcan.ee.iastate.edu/~cise/instructors/downloads/InterBO/InterBO.pdf.

Microsoft Corporation. ”Web-Based Security in Commerce Server 2002.” July 2000. URL: http://www.microsoft.com/technet/treeview/default.asp?url=/technet/prodtechnol/comm/comm2002/maintain/CS02WSec.asp.

Nitzberg, Sam. “Trusting Software: Malicious Code Analysis.” 1999. URL: http://www.iamsam.com/papers/milcom_malicious_code_analyses/MCAart16.htm .

Sahu, Bijaya Nanda. “Is your Java code secure- or exposed?.” 2001. URL: http://www-900.ibm.com/developerWorks/cn/java/j-staticsec/index_eng.shtml.

Shostack, Adam. “Security Code Review Guidelines.” 06 November 2001. URL: http://www.homeport.org/~adam/review.html .

Sun Microsystems, Inc. “Security Code Guidelines.” 2 February 2000. URL: http://java.sun.com/security/seccodeguide.html .