GROUP TECHNOLOGY & OPERATIONS

Secure Source Code Analysis Guide

Global ID: DBSG_CB_028

Scope/Coverage: Group-wide

Issuer: Reuven MAUTNER

Associated Unit: ISS Application Security

Last Review Date: 30 September 2020

Review Frequency: Biennially

Where applicable:

Reference Policies, Standards, Guides:

No.	Global ID	Name	Document Type
1	DBS_11_S_0027_GR	DBS System Development Life Cycle (SDLC) Standard	Standard
2	DBS_11_G_0032_GR	Code and Build Guide	Guide

Associated Applications

No.	Application Code	Application Name
		Not applicable

Associated Process Maps/Tools/Templates

No.	Name (as per the name in myDBS House repository)

CONTENTS

1 INTRODUCTION

1.1 Purpose

Secure source code analysis is the automated or manual review of a program's source code with the purpose of finding security defects and fixing them before the application is promoted to production.

Secure source code analysis is synonymous to static code analysis, where the source code is analyzed simply as code and the program is not running. This removes the need for creating and using test cases and may separate itself from feature-specific bugs (e.g. Usage of a non-Standard cryptography algorithms, weak random number generators etc.). The review focuses on finding security defects in the program that may be detrimental to its proper functioning (e.g. system crash).

It is recommended to use this guide as a supplement after reading the DBS SDLC Standard and Waterfall and Agile Procedures.

1.2 Audience

This guide is developed for: Developers, Development Leads, and Project Managers or Scrum Masters.

1.3 Scope

The guidelines and recommended practices included in this guide cover the following activities:

- Automated Source Code Analysis
- Manual Secure Code Review

This guide does not cover all the industry's best practices. It is highly recommended that Development Teams practice whatever works best for their team and project.

1.4 Key Assumption

These guidelines are recommended practices only, and not how-to's or user guides. It is assumed that the Developers and Development Leads have the necessary experience and knowledge to effectively perform code analysis activities.

2 AUTOMATED SOURCE CODE ANALYSIS

Automated source code analysis is basically automated code debugging using a code analyzer. The aim is to find security bugs and defects that may not be obvious to a programmer. It is meant to find defects like possible buffer overflows or untidy use of pointers and misuse of garbage collection functions, all of which may be exploitable by a hacker.

Code analyzers work using rules that tell it what to look for. With too little precision, an analyzer might spew out too many false positives and flood the user with useless warnings, while too much precision might result in the analyzer taking too long to complete a scan; therefore, must be a balance.

There are two kinds of analyzers:

- Interprocedural – Detects patterns from one function to the next, and these patterns are correlated so that the analyzer can create a model and simulate execution paths; and
- Intraprocedural – Focuses on pattern matching and depends on what kinds of patterns the user is looking for.

2.1 Static Code Analysis

Static code analysis is a method of analyzing and evaluating source code without executing a program. Static code analysis is part of what is called "white box testing" because, unlike “black box testing”, the source code is available to the testers. Many types of software testing involve static code analysis, where developers and other parties look for bugs or otherwise analyze the code for a software program.

Static code analysis is also known as static program analysis.

2.2 Dynamic Code Analysis

Dynamic code analysis is a testing procedure that is part of the software debugging process and used to evaluate a program during real-time execution. It is used during the development phase.

The main purpose of dynamic code analysis is to find errors while a program is running, functions are invoked, and variables contain values, versus checking each line of code, mentally applying values and guessing possible branching scenarios.

2.3 Secure Source Code Analysis - Review Items

2.3.1 API Abuse

An API is a contract between a caller and a callee. The most common forms of API abuse are caused by the caller failing to honor its end of this contract. For example, if a program fails to call chdir() after calling chroot(), it violates the contract that specifies how to change the active root directory in a secure fashion. Another good example of library abuse is expecting the callee to return trustworthy DNS information to the caller. In this case, the caller abuses the callee API by making certain assumptions about its behavior (that the return value can be used for authentication purposes). One can also violate the caller-callee contract from the other side. For example, if a coder subclasses SecureRandom and returns a non-random value, the contract is violated.

2.3.2 Encapsulation

Encapsulation is about drawing strong boundaries. In a web browser that might mean ensuring that your mobile code cannot be abused by other mobile code. On the server it might mean differentiation

between validated data and unvalidated data, between one user's data and another's, or between data users can see and data that they are not.

2.3.3 Environment

This section includes everything that is outside of the source code but is still critical to the security of the product that is being created. Because the issues covered by this kingdom are not directly related to source code.

2.3.4 Input Validation and Representation

Input validation and representation problems areas caused by metacharacters, alternate encodings and numeric representations. Security problems result from trusting input. The issues include: "Buffer Overflows," "Cross-Site Scripting" attacks, "SQL Injection," and many others.

2.3.5 Security Features

Software security is not security software. Here we're concerned with topics like authentication, access control, confidentiality, cryptography, and privilege management.

2.3.6 Time and State

Distributed computation is about time and state. That is, in order for more than one component to communicate, state must be shared, and all that takes time.

Most programmers anthropomorphize their work. They think about one thread of control carrying out the entire program in the same way they would if they had to do the job themselves. Modern computers, however, switch between tasks very quickly, and in multi-core, multi-CPU, or distributed systems, two events may take place at exactly the same time. Defects rush to fill the gap between the programmer's model of how a program executes and what happens in reality. These defects are related to unexpected interactions between threads, processes, time, and information. These interactions happen through shared state: semaphores, variables, the file system, and, basically, anything that can store information.

2.4 Open Source Software Risk Management

Open-source software (OSS) is computer software with its source code made available with a license in which the copyright holder provides the rights to study, change, and distribute the software to anyone and for any purpose. Open-source software may be developed in a collaborative public manner. Using OSS components has many benefit such as reduces costs, increase agility and drives competitive advantages. However, it also brings to organization a different set of problems.

Security vulnerabilities in OSS has a bigger risk over close source software as the source is available and ease the process of exploitation. With the tremendous growth of the component and its licenses, exposure of legal and business risk increases. In additional, the transitive dependency of components increases the complexity and reduce the visibility of the risk an organization is exposed to.

The Open Source Scanning tool will provide application team with the necessary information about the open source component used in their application, what are the potential security and license risk involved, therefore allowing application team to manage the risk.

2.5 Automated Security Testing in Quarantine Environment for Mobile Apps

All Mobile applications go through automated security testing in a quarantine environment before being promoted into production.

Among the security issues that will be checked are:

SSL Cert Pinning: To prevent Man-in-the-Middle attacks or sniffing of HTTPS traffic (applicable to Android and iOS devices)

Code Reverse Engineering: To verify if code obfuscation is enabled for code reverse engineering prevention (applicable to Android devices only)

Code Tampering: To detect if the app is able to run on a Rooted Android /Jailbroken iOS device (optional)

2.6 Automated Secure Source Code Analysis – Enterprise Tools

Automated Secure Source Code Analysis can be Performed using the Enterprise Static and Dynamic Analyzer which can be triggered automatically part of the CI/CD Pipeline or Manually triggered if the application is not onboarded to the Enterprise CI/CD Track.

Application that is not supported on the Enterprise platform, can proceed to use the Static Code Analyzer / Manual review methods that suits their Application environment to validate the above-mentioned review ITEMS and provide evidence's whenever necessary.

3 MANUAL SECURE CODE REVIEW

Manual secure code review is the process of reading source code line-by-line in an attempt to identify potential vulnerabilities. It is a tedious process that requires skill, experience, persistence, and patience. Vulnerabilities discovered, and subsequently addressed through the manual review process, can greatly improve an organization's security posture.

As with automated static analysis tools, manual secure code review has its strengths, limitations, and costs.

Manual review is great at unwinding business logic and understanding the intentions of a developer. Automated tools struggle in these areas, resulting in false positives and, worse, missed issues.

Flaws related to authentication, authorization, cryptography, and overall data validation are often best identified by manual analysis.

In the following sections, we will provide manual secure code review guidelines for critical application functionalities such as database access and file upload and a more generic overview of indicators of insecure or malicious code.

As a developer and code reviewer, it is important to know what to look for as missing an item might result in insecure or malicious code in your application. Importantly, regulatory requirements for the financial services sector dictate the use of code review as a method to improve code quality, especially security.

This guide has three main parts:

- **Critical Application Functionalities**: Provides code review guidelines for specific application functionalities;
- **Common Countermeasures**: Provides code review guidelines for the correct implementation of common countermeasures against typical application level vulnerabilities; and
- **Indicators of Insecure Code**: Provides indicators of insecure or malicious code.

By evaluating the proper implementation of critical application functionalities and common countermeasures, and looking out for indicators of insecure code, the developer and code reviewer will have a higher chance of identifying vulnerabilities introduced in the application code

3.1 Critical Application Functionalities

The following sections describe items to review for the following critical application functionalities:

- Database access
- Use of cryptography
- File upload

3.1.1 Functionality I: Database Access

3.1.1.1 *Considerations*

Nearly all applications need to store information in a database. Modern database management systems (DBMS) contain thousands of standard stored procedures, offer a lot of additional functionality such as direct access to the operating system and can even contain a complete development environment.

Database access provides a major attack surface and an attacker might have the following specific goals:

- Extracting confidential information from the database such as account statements;
- Accessing credentials such as user IDs and (hashed) passwords;
- Modifying the content of the database such as the balance of an account;
- Performing a Denial-of-Service (DoS) attack by deleting the whole or parts of the database; and
- Accessing the filesystem of the operating system with the goal of accessing configuration files, executing malicious code or taking over the server and using this to attack other servers in the environment.

Please refer DBS Access Control Standards in Appendix

The next sections give an overview of what code reviewers should look out for.

3.1.1.2 *Review Items*

3.1.1.2.1 Use of Prepared Statements with Parameterized Queries

The only real defense against SQL Injection (SQLi) is the use of prepared statements with parameterized queries. Code reviewers should verify that the code and framework make use of them and reject code which does not use them.

This defense can be improved by the application of input validation – using whitelisting and if necessary blacklisting.

3.1.1.2.2 Handle and Log Errors

Review the error-handling related to database access:

- Errors should be captured and logged to be reviewed or used in forensic investigations; and
- Any error-messages to the end-user should be generic and not contain any technical details or confidential information such as file locations.

3.1.1.3 *Additional Reading*

Link	Description
https://cheatsheetseries.owasp.org/cheatsheets/SQL_Injection_Prevention_Cheat_Sheet.html	The OWASP SQL Injection Prevention Cheat Sheet

3.1.2 Functionality II: Use of Cryptography

3.1.2.1 *Considerations*

Cryptographers are specialists in the study or creation of encryption protocols and ciphers. Developers should never attempt to “improve” on a protocol, chances are that you weaken it.

Regulatory standards might prescribe or forbid the use of specific algorithms or ciphers. This should be documented during the design stage of the application.

3.1.2.2 *Review Items*

3.1.2.3 *Use Approved Encryption Algorithms*

Code reviewers should verify that the code uses only algorithms, protocols and ciphers approved by the DBS Cryptography Standard and DBS Cryptography Usage Guidelines (Please refer Appendix).

3.1.2.4 *Use Standard Libraries*

Code reviewers should verify that the code uses mature encryption libraries to implement encryption. The code should implement any encryption technology according to the documentation of the library and in line with the DBS Cryptography Standard and DBS Cryptography Usage Guidelines (Please refer Appendix).

3.1.2.5 *Secure Random Number Generation*

Code reviewers should verify that random numbers are generated securely, according to the documentation of the library or framework used and in line with the DBS Cryptography Usage Guidelines. Random number generation starts with a seed, which should be generated securely.

3.1.2.6 *Encryption of Data and Traffic*

Code reviewers should verify that encryption is applied on sensitive and confidential data (both at rest and during transport, e.g. use of HTTPS instead of HTTP) on sensitive and confidential data. This should be based on the requirements and design of the application and in line with regulatory requirements such as those from MAS.

3.1.2.7 *Password Storage*

Code reviewers should verify that password storage is not used as developers should be using the Centralised AD Infrastructure.

In case there are specific requirements and design to store passwords, the current best practice is to apply a PBKBF2 (Password-Based Key Derivation Function).

Service passwords or API secrets are a special case and it is often necessary to store them. Code reviewers should verify that the following principles are applied:

- Never hardcode the secret or password;
- Use a separate configuration file (or keystore, environment variable) to store the secrets or passwords; and
- Use the method offered by the framework to encrypt the passwords and secrets.

3.1.2.8 *File Transfer*

Code reviewers should verify that an integrity check is performed when downloading codes from the server. This is usually based on checking the validity of the SSL certificate used during transfer:

- Server name is the expected one;
- Certificate is signed by a valid Certificate Authority;
- Certificate is not expired; and
- Certificate is not revoked.

3.1.2.9 *Temporary Buffers*

Code reviewers should verify that temporary buffers that hold cryptographic keys, passwords or other sensitive data are overwritten with zeroes or random bytes after use.

3.1.2.10 *Handle and Log Errors*

Code reviewers should verify the error-handling related to cryptography:

- Errors should be captured and logged to be reviewed or used in forensic investigations;
- Any error-messages to the end-user should be generic and not contain any technical details or confidential information; and
- Private keys should not be logged.

3.1.2.11 *Additional Reading*

Link	Description
https://wiki.owasp.org/index.php/Guide_to_Cryptography	The OWASP Guide to Cryptography
https://cheatsheetseries.owasp.org/cheatsheets/Cryptographic_Storage_Cheat_Sheet.html	The OWASP Cryptographic Storage Cheat Sheet

3.1.3 Functionality III: File Upload

3.1.3.1 *Considerations*

File upload provides a major attack surface and an attacker might have the following specific goals:

- Abusing the application to distribute malicious software or illegal items such as cracked software or porn;
- Uploading malicious software or virus infected files to attack a user viewing the content or downloading the file. The attack might be geared towards attacking the operating system of the user, or attempting to bypass authorization in the application itself, e.g. the file could include malicious JavaScript that executes when an administrator views the file;
- Uploading a backdoor to browse the file system of the application server, access critical files and probe for other vulnerabilities. This is a shell written in the language that the web server understands such as JSP and PHP;
- Performing a Denial-of-Service (DoS) attack to cause system disruption by uploading large files or overwriting the application or server configuration; and
- Adding a page to a website, e.g. for phishing purposes where the file is served by a web server and domain trusted by the victim.

The next sections give an overview of what code reviewers should look out for. Other considerations might be outside the domain of code review, but should have been considered in earlier development stages, such as:

- Restricting file upload as much a possible (e.g. authenticated users or administrators only);

- Performing virus scan after upload;
- Storing files outside the web root (i.e. the application and not the web server should be responsible);
- Using a different server and domain for storing and accessing uploads;
- Randomizing filenames; and
- Logging file upload and other potential problems.

3.1.3.2 *Review Items*

It is important to note that while Fortify will detect any issues with file upload, the scan might not be able to detect if the correct authorization is used.

3.1.3.3 *Implementation of Proper Authorization*

Code reviewers should verify that the upload functionality is restricted to authorized users only and according to the business requirements.

3.1.3.4 *Use of File Manipulation Functions Provided by the Framework*

Code reviewers should verify that the developer is using file manipulation functions provided by the framework whenever possible instead of coding their own functions.

It is often required to split a received filename into its components such as name and extension or join a file path with a filename. Where possible, the developer should use the framework provided functions to split filenames and extensions or adding a directory to a filename.

It is critical to understand that an attacker can bypass client-side restrictions and has control over the uploaded filename and extension. Developers implementing homebrew functions (e.g. checking for the last dot and splitting the received name and extension) might fail. In addition, simply adding a received filename to a directory might enable an attacker to manipulate the resulting path (e.g. putting ***../*** in the file-name).

Library functions understand differences between operating systems and will take these into account.

It is common to create a “location” of a file by adding a fixed directory name to a received filename and extension. The code reviewer should verify that framework or language provided functions is used to manipulate paths and filenames, instead of string manipulation by the developer.

If the framework or language does not provide specific functions, the string manipulation code should check that filenames and extensions are not empty, and that the resulting path is shorter than 255 characters.

3.1.3.5 *Whitelisting of Allowed Characters*

Code reviewers should verify that the developer has implemented whitelisting to ensure that the received filename and extension are free of any dangerous characters that might be abused to attack the operating system.

The code reviewer should check that:

- Only ASCII characters are allowed for file name and extension by whitelisting; and
- Filename can contain a maximum of one dot.

3.1.3.6 *Whitelisting of File Extensions*

Code reviewers should verify that the developer has implemented whitelisting of file extensions.

An attacker might abuse the file upload functionality to upload unwanted files such as malicious executables, backdoors or files containing JavaScript to perform Cross-Site Scripting (XSS) attacks.

The reviewer should verify that file extensions are restricted to what is necessary to implement the specific business requirements, e.g. “doc” or “pdf” only.

3.1.3.7 *Verification of User-Controlled Information*

An attacker can tamper with HTTP headers (such as MIME-TYPE) or upload lengths.

Code reviewers should verify that the code does not use this information in an insecure way, e.g. make security decisions on the MIME-TYPE only or consider the given length as true without verification that it is non-negative and greater than zero.

3.1.3.8 *Proper Handling and Logging of Errors*

Code reviewers should verify that the file upload functionality captures and logs errors to support review processes or forensic investigations.

The code review should also verify that any error-messages displayed to the end user should be generic and exclude any technical details or confidential information such as file locations.

3.1.3.9 *Additional Reading*

Link	Description
https://cheatsheetseries.owasp.org/cheatsheets/File_Upload_Cheat_Sheet.html	The OWASP recommendations and considerations for file uploads.

3.2 Common Countermeasures

The following sections describe items to review for the following common countermeasures against typical application level vulnerabilities:

- Data encoding
- Input validation
- Unique token

3.2.1 Countermeasure I: Data Encoding

3.2.1.1 *Considerations*

Together with input validation, encoding of data is the best protective mechanism against Cross-Site Scripting (XSS). Encoding of data renders harmless characters that have a special meaning to the browser.

3.2.1.2 *Review Items*

3.2.1.2.1 Determine Source and Sink

As a code reviewer, you should determine where the data enters the application (source) and where it is displayed (sink, where the potential data is displayed).

3.2.1.2.2 Determine HTML Context

Determine the HTML context of the sink and which characters pose a danger. Main contexts and dangerous characters are listed in this table:

Context	Dangerous Characters
HTML Context	< > = / ` ( )
HTML Attribute Name Context	< > = ` ( )
HTML Attribute Value Context	‘ “ = ( ) ` Note that in the special case of injection in an event attribute, the supplied input will be executed as JavaScript and encoding for that context should be applied.
HTML Comments Context	- ! >
JavaScript Context	The following characters should be encoded: \, /, ”, ’, `, {, }, [,], (,), %, ;, -, <, >, !, =

3.2.1.3 *Review Encoding*

Code reviewers should determine if proper encoding is used for the context. Standard functions from the used framework should also be preferred over homebrew methods.

If the sink is in JavaScript context, code reviewers should check if it is possible to implement the required functionality without echoing data in JavaScript context.

3.2.1.4 *Additional Reading*

Link	Description
https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html	The OWASP XSS Prevention Cheat Sheet documents counter-measures against XSS.

3.2.2 Countermeasure II: Input Validation

3.2.2.1 *Considerations*

Input validation is a key protection method against many typical web application attacks such as Cross-Site Scripting (XSS), SQL Injection (SQLi) or Path Manipulation.

3.2.2.2 *Review Items*

3.2.2.2.1 Validate Early

Review that input validation is applied as soon as the data enters the application, before any other processing.

3.2.2.2.2 Validate at least Server-Side

Review that input validation is applied at least server-side (client-side filtering can be bypassed).

3.2.2.2.3 White-list First, Black-List Second

Review that the input validation routine uses white-listing first, then black-listing. White-listing should use the following components:

- Limit the size of input;
- Verify the type;
- Verify the boundaries;
- Verify the permitted characters;
- Verify the syntax;
- Verify business rules

Black-listing is highly dependent on context. Review that any black-listed characters or string are not required by business rules. Review that black-listing is not applied on passwords (would make them weaker).

3.2.2.2.4 Handle and Log Errors

Review the error-handling related with input validation:

- Errors should be captured and logged to be reviewed or used in forensic investigations.
- Any error-messages to the end-user should be generic and not contain any technical details or confidential information and should not include the user input (the danger is Cross-Site-Scripting)

3.2.2.3 *Additional Reading*

Link	Description
https://cheatsheetseries.owasp.org/cheatsheets/Input_Validation_Cheat_Sheet.html	The OWASP Input Validation Cheat Sheet

3.2.3 Countermeasure III: Unique Token

3.2.3.1 *Considerations*

Criminals might create code on their website or a popular website they hacked and call functions in your application. This is called Cross-Site Request Forgery (CSRF).

If protection is not applied, an attacker could call critical functions such as those used for managing users or making financial transactions, thereby bypassing all access controls. The request will execute in the context of the victim.

A typical countermeasure is to include a unique and unpredictable token in each request. Since the attacker cannot predict the content of the token, the above attack will fail.

3.2.3.2 *Review Items*

3.2.3.2.1 Use Framework Provided Token

Many frameworks can automatically add anti-CSRF tokens. Depending on the application, unique tokens can be used for each request or for each session.

Code reviewers should verify that any function that modifies server-side context uses anti-CSRF tokens, either configured or using a homebrew solution.

3.2.3.2.2 Homebrew Anti-CSRF Solution

In case there is no standard framework-level function, a homebrew solution may be considered. This solution must be reviewed for the following:

- Check the Origin HTTP header: If present, the developer should block the request if the Origin header is not the expected one (i.e. the production URL of your application); and
- Check the Referrer HTTP header: if the Referrer header is not present, the developer should block the request. If the Referrer header is included it should match the production URL of the application.

It is important to note that the “production URL” in the above might need to be configured to make testing possible.

For any operation that changes state in the application, the request should include a CSRF token that is verified server-side. The token should be unique per session (or per request) and should be a large random value created by a cryptographically secure random number generator.

If the server-side verification fails, the request should be blocked, and the issue should be logged.

3.2.3.2.3 Handle and Log Errors

The code reviewers should check the error-handling related with missing anti-CSRF tokens:

- Errors should be captured and logged to be reviewed or used in forensic investigations (if the anti-CSRF token is wrong or not included in the request, an attack is ongoing); and
- Any error-messages to the end-user should be generic and not contain any technical details or confidential information.

3.2.3.3 *Additional Reading*

Link	Description
https://cheatsheetseries.owasp.org/cheatsheets/Cross-Site_Request_Forgery_Prevention_Cheat_Sheet.html	The OWASP CSRF Prevention Cheat Sheet

3.3 Indicators of Insecure Code

The following sections describe items to review for:

- Code readability and maintainability
- Debug codes
- Malicious code

3.3.1 Indicator I: Code Readability and Maintainability

Code readability is a critical component for the maintainability of code. When reviewing application code, the code reviewer must be able to understand what the code does and explain what he is reviewing.

A malicious developer who is trying to hide something may intentionally reduce the readability of the code to avoid detection by a code reviewer.

3.3.1.1 *Signs and Symptoms*

3.3.1.2 *Non-Adherence to Language Specific Guidelines*

Verify that the code uses the naming conventions and formatting of the language specific guidelines. The code reviewer should pay special attention to code indentation which might be deliberately or accidentally misleading.

3.3.1.3 *Unfinished Code Going into Production*

Verify that the code does not contain comments such as “TO DO” or “FIX ME”. Code under review should be production ready and should not contain any issues that still need fixing. The code reviewer should pay special attention to such code segments as they may be used to hide malicious code.

3.3.1.4 *Suspicious Coding Styles*

While different developers might have distinctive coding styles, the resulting code should be easily readable by a fellow developer. In general, the code reviewer should consider whether:

- Class names, methods, parameters and variable names are used to communicate the intent of the code;
- Comments are used sparingly, and code is self-explanatory (where feasible);
- Methods are short and consists of only a few statements;
- Nested conditionals are used sparingly and should be replaced by simpler code (where possible);
- Code does not contain duplicated code or logic; and
- Code does not contain unused or needless code.

3.3.1.5 *Code Portability*

Code reviewers should verify that the code is portable across different DBS environments, i.e., environment-specific configuration like database details, log configuration, and file path maintained should be maintained outside the code.

3.3.1.6 *Additional Reading*

Link	Description
https://owasp.org/www-project-code-review-guide/	The OWASP Code Review Project

3.3.2 Indicator II: Debug Codes

Debug codes should not be part of code in production:

- Debug codes are never part of functional testing and might introduce security bypasses or vulnerabilities;
- Debug codes might slow down the application;
- Debug codes might make the code harder to read; and
- An attacker might be able to control or access debug functionality.

Debug codes can be any code added by a developer that is not part of the design and is simply there for debugging, such as:

- Dumping application state to a debug log;
- Printing out control strings when the application reaches certain execution points or flows;
- Code that forces a specific state of the application; and
- Verbose logging to console or files.

Debug codes should be removed manually (delete from source) or by using conditional compilation for languages that support it.

3.3.2.1 *Signs and Symptoms*

3.3.2.1.1 Presence of Debug Code

Code reviewers should verify that the code does not include debug codes or that debug statements are removed by conditional compilation.

3.3.2.1.2 Debug Logging

Some application designs require that the application can be put in “debug” mode and that debug logs must be created.

The condition to put an application in debug mode, should never be part of user-controllable input such as parameters or HTTP headers.

3.3.2.2 *Additional Reading*

Link	Description
https://owasp.org/www-project-code-review-guide/	The OWASP Code Review Project

3.3.3 Indicator III: Malicious Codes

The Monetary Authority of Singapore (MAS) and other security standards requires source code to be reviewed for malicious codes, trojans, backdoors, logic bombs and other malware:

- A **trojan** is any piece of software that hides its true intent;
- A **backdoor** is an undocumented way to access an application or data (often created by developers and inadvertently pushed to production); and
- A **logic bomb** is malicious logic in an application which gets activated only on a certain predetermined date and time or only when specific circumstances happen.

Apart from the above, a malicious developer might try to deliberately introduce weaknesses and vulnerabilities by:

- Deviating from the design;
- Not implementing specific security controls and features; and
- Violating secure coding practices.

By looking out for signs and symptoms, code reviewers can be alerted to suspicious code that may warrant further examination.

3.3.3.1 *Signs and Symptoms*

3.3.3.1.1 Complex Code

If you, as a code reviewer cannot understand the code, there is a problem. A malicious code developer might try to hide certain malicious actions by using overly complex code with the goal of inserting trojans, backdoors, logic bombs or vulnerabilities into the application. (see Section 11)

3.3.3.1.2 Deviation from the Design

Code must be written according to the design. The code reviewer should refuse any code that does not follow the design.

Specifically, code reviewers should review the implementation of any security related features, such as authentication, authorization, logging and encryption.

3.3.3.1.3 Missing Security Features

A malicious developer might deliberately not implement specific security controls to make the application vulnerable to typical web-application vulnerabilities, such as SQL Injection (SQLi), Cross-Site Scripting (XSS), etc.

Code reviewers should verify that all countermeasures for typical application-level vulnerabilities are implemented.

3.3.3.1.4 Unusual Control Flow

While this is closely related to the readability of the code, a malicious developer might try to insert unwanted code paths by manipulating the control flow of the function.

Unwanted code paths might enable access control bypasses (authentication, authorization, dual control) or bypass audit logging.

3.3.3.1.5 Hardcoded Constants

Hardcoded constants should never be part of an application but moved to the database or configuration files.

They are often used to insert logic bombs. Code reviewers should be especially wary of hardcoded usernames and/or passwords, magical strings that enable specific functionalities, hardcoded dates, and hardcoded account numbers.

3.3.3.1.6 Calls to Operating System Functions

Code reviewers should verify that calls to operating system functions are part of the design. It is important to make sure that proper access controls (i.e. authentication, authorization, logging) and security features (e.g. input validation) are implemented.

If an attacker can access operating system level commands, he can access all functionalities on the server with the access rights of the application server. Any “elevation of rights” bug in the operating system might enable the attacker to take full control over the system.

3.3.3.1.7 External Communications

Code reviewers should verify that all external communications are part of the design and are implemented accordingly using proper access controls and logging.

A malicious developer might try to send confidential data to another system, or he might also try to enable insecure communications (e.g. using HTTP instead of HTTPS).

3.3.3.2 ***Additional Reading***

Link	Description
https://owasp.org/www-project-code-review-guide/	The OWASP Code Review Project
https://www.owasp.org/images/a/ae/OWASP_10_Most_Common_Backdoors.pdf	The OWASP Top 10 of most common backdoors

APPENDIX 1 ADDITIONAL REFERENCES

NIL

APPENDIX 2 VERSION HISTORY

Version	Date of Issue	Summary of Key Changes
1.0	30 Sep 2018	Initial version
1.1	30 Sep 2020	Biennial review and update of Additional Reading