GROUP TECHNOLOGY & OPERATIONS

Secure Coding Practice Guide

Global ID: DBSG-CB-027

Scope/Coverage: Group-wide

Issuer: Reuven MAUTNER

Associated Unit: ISS Application Security

Last Review Date: 30 September 2020

Review Frequency: Biennially

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

This document provides insights into the hacker mindset, the potential goals and attack methods of an adversary and link to secure coding practices.

As a developer, it is important to understand these goals (why) and methods (what) of attackers. It enables you to better develop secure code and defensible applications; by understanding why certain practices are needed, you will acquire a better appreciation of application security and the thinking required for writing secure code.

This guide has three main parts:

- **Inside the Hacker Mindset**: what type of hackers exist, what are their motivations and what are their potential targets;
- **Attack Stages and Methods**: how do attackers stage their attack, what methods do they use, and which secure coding practices will counteract their attempts; and
- **Secure Coding Practices**: what are the countermeasures and the recommended implementation methods independent of in a language independent way.

Developing secure code requires a good understanding of potential threats and the available countermeasures to defend your application against attacks. As it is not possible or practical to comprehensively address all current and future threats and mitigations, this guide serves as a good starting point to develop a secure coding mindset.

2 INSIDE THE HACKER MINDSET

Without proper understanding of what criminals are targeting and how they work it will be impossible to implement proper defences for your system.

2.1 What is at stake for DBS

A successful attack might result in severe damages for DBS:

- **Business disruption**: An attack might bring down a critical system, impairing customer transactions or make it impossible to make payments. This might result in loss of business and customer trust.
- **Intellectual property loss**: A criminal might steal specific information about internal processes or proprietary knowledge that give DBS a competitive advantage. The loss of this information (or the sharing with unauthorized parties) might reduce business opportunities or competitiveness.
- **Customer privacy loss**: Customers' private and personal data must be protected according to the law and regulatory standards for all jurisdictions that DBS operates in. If a criminal gains access to such information, this might lead to regulatory actions and legal fines.
- **Reputation loss**: If an attack against DBS succeeds, the reputation and trustworthiness of the bank might be at stake, resulting in loss of business and customers.
- **Financial loss**: Criminals might conduct fraud or manipulate financial transactions that results in significant financial impact on the bank. This can lead to additional financial impact in the form of customer compensation and regulatory penalties.

In all cases, a successful attack imposes financial costs on DBS. This might include internal human resources required to manage, resolve and recover from the attack; service providers to provide advisory, remediation and investigation; and security solutions to strengthen the bank's defences; and public relations and marketing initiatives to regain the trust of the customers.

2.2 Motives for Attacks

Attackers might target the bank for many different reasons:

- **Fun**: Many technically-oriented people are interested in discovering how things work. Doing so, they might accidentally break things or stumble upon vulnerabilities and exploit them. In most cases, they are not after monetary rewards but are out there trying to make a name for themselves in the hacker scene or are just fine with the acquired knowledge.
- **Hacktivism**: Usually the motive is social or political and directed at governments or major companies that might act contrary to what the attacker believes in. Most hacktivists are not interested in stealing data unless exposure of the information can damage the victim's reputation. The common tactic is to deface the victim's public website or to launch a Denial-of-Service (DOS) attack. Unlike other attacks, the hacktivist seeks high visibility of the result and is usually not interested in building upon this attack.
- **Cybercrime**: The goal is to make as much money as possible in the shortest period. This include direct attacks on financial or payment systems, ATMs, or tricking users into sending money to accounts controlled by the criminals.

- **Advanced Persistent Threats (APTs)**: Such threats are usually state-sponsored as the attacks require a lot of resources and skills to be successful. The attackers are not interested in short time financial gains but wants to gain a foothold in the victim to conduct sophisticated long-term campaigns. Targets are typically organizations providing essential services or critical infrastructure, including financial institutions.

2.3 Types of Hackers

Hackers or usually classified into skill set categories and the methods they apply (legal or non-legal):

- **Script kiddies**: This is a derogatory term used by "advanced" hackers for beginners or hackers that only use public available tools or scripts without much knowledge. They might not fully understand how the tools work, the limitations of the tools, or the effects on a system or application.
- **Black hats**: These are hackers that operate outside the law, and might work alone or as part of a criminal organization. The motive is usually for financial gains.
- **Gray hats**: Grey hats operate outside the law, but their intent is usually not financial gains, but is limited to hacktivism or fun. They might probe or attack a website without approval and might or might not warn the owner of discovered vulnerabilities.
- **White hats**: Also referred to as "ethical hackers". White hats operate within the laws, such as penetration testers or bug bounty hunters. They attack systems to find vulnerabilities and help make systems more secure, and never attack or probe a system without having approval or a binding contract.

2.4 Target of Attacks

An attack can be **deliberate** and **targeted**, or the application might have a vulnerability that is hit by **chance** when an internet worm accesses the application.

Criminals directing a targeted attack at the bank incur considerable time, resources and effort. While they may target your specific application system, there are also other potential targets that are at risk:

- **Application users**: This includes customers, partners, suppliers, regular users or application administrators. In certain cases, criminals might target specific customers (e.g. famous persons, politicians and other high-valued customers) for the following reasons:
  - **Identity theft**: The criminal is interested in obtaining personal data such as names, phone numbers, addresses, and other confidential information. Similarly, certain information such as passwords might be reused on other sites and could be abused to target another application.
  - **Blackmail**: The criminal might abuse any information gained to blackmail or extort a victim and might threaten to release the information if the victim doesn't pay.
  - **Monetary gains**: The criminal might obtain information about private banking customers or companies, including their identity and account balance and use this information for targeted attacks against wealthy customers or companies.
- **Application infrastructure**: Criminals might not be interested in exploiting your application but might want to abuse the infrastructure to house malware or illegal files. Criminals will often

attack vulnerable applications or servers and use this compromised infrastructure for storing and distributing malware or illegal files.

- **Internal access**: In certain cases, criminals might attack a specific application to gain access into the internal network to target other connected servers and applications. As applications are typically shielded from external networks by firewalls; if an application can be attacked, the criminal is already behind your firewall. Criminals are known to spy for months on internal networks to make certain their presence is persistent.

3 ATTACK STAGES AND METHODS

In the complex world of the global internet, a criminal might attack network and security systems (e.g. firewall, routers, etc.), organisations or companies in control of the DNS servers (e.g. top-level domains, ISPs, etc.), the domain registrars, weak or non-existing management processes or the human. As it is impractical to provide a comprehensive discussion of all attacks, for this document, we will focus on application-level attacks where developers can implement countermeasures.

Independent of the type of hacker or motives and whether it is an authorised penetration test or criminal attack, the following stages of attacks are commonly applied:

- **Information gathering**: Perform reconnaissance by gathering as much information about the application and its users as possible;
- **Vulnerability detection**: Scan the target using a vulnerability scanner or execute manual tests to detect exploitable security holes or weaknesses; and
- **Exploitation**: Exploit the weakness and continue to build upon a successful exploitation.

In the case of a criminal attack, the attacker might want to try to maintain continued access and cover his tracks. In the case of a penetration test, the tester might install scripts or backdoors and create accounts, which will be removed before providing a report explaining how the tester was able to penetrate the application system.

In all stages, your task as a developer is to make it as hard as possible for an attacker or tester to reach the objective. The harder it is for an attack to succeed, the more time, effort and skills an attacker will need to be successful. And this provides the organization with a higher chance of detecting and stopping the potential attack.

3.1 Information Gathering

A criminal might gather information for weeks or months prior to attacking the target. Many sources are available such as internet searches, social engineering, DNS system or simply browsing the application.

There are two main methods an attacker could use to probe an application in the initial stages of an attack:

- **Communications level**: An attacker might be able to see communications between the user and the application. Most of the time this involves sniffing network traffic but might also be executed at intermediary proxies between the browser and the application. The proper use of encryption makes this impossible (refer to Section 8).
- **Application (software) level**: An attacker can “fingerprint” the application by browsing as a regular authenticated or public user. This approach might reveal a lot of internal details (e.g. software version and patch levels of frameworks used, etc.). An attacker can then make use of this knowledge (e.g. vulnerable frameworks, functions that did not implement authentication, etc.) in a later attack.
- An attacker will typically make use of automated tools to spider the application and try to detect more functionality by brute-forcing common locations (e.g. directory indexing).
- At this stage, the attacker might also try to provoke detailed error messages or stack traces by sending invalid input to the application. Implementing proper input validation (refer to Section 6) and preventing information leakage (refer to Section 5) can make this a lot harder.

3.2 Vulnerability Detection

Based on the information gained during the Information Gathering stage, attackers or penetration testers will use automated tools (e.g. attack proxies such as OWASP Zap or Burp Pro) to scan the application for vulnerabilities and weaknesses they can later exploit for the targeted attack.

In this stage, automated tools might generate a high volume of requests, with targeted payloads for typical application level vulnerabilities such as SQLi or XSS. The tools might fail to detect issues or indicate that further manual research is needed, and may also indicate that certain information (e.g. credentials, session tokens, etc/) are sent in the clear without proper encryption of the communications channel.

Developers can implement typical countermeasures such as input validation (refer to Section 6) or measures that may slow down brute-forcing (refer to Section 5).

3.3 Exploitation

Based on the results of the Information Gathering and Vulnerability Detection stages, the attacker will now exploit the identified vulnerabilities.

The type of exploits used will depend on the goals of the attacker and will include specific exploits against typical web application vulnerabilities such as SQL Injection (SQLi) and Cross-Site Scripting (XSS) attacks. There are tools that are freely available to the attacker, and the tools become more powerful every year.

Developers can implement typical Countermeasures such as input validation (refer to Section 6), output encoding (refer to Section 8) and parameterized queries (refer to Section 9).

4 OVERVIEW OF SECURE CODING PRACTICES

With an understanding of the hacker mindset and methods, as a developer, you are in a better position to develop secure code that can counter the potential threats to your application.

The following sections provide the considerations, possible approaches, and specific countermeasures for common issues that should be implemented to address the attack methods discussed in Section 3. For every countermeasure, we describe the recommended implementation method (how) in a language independent way and where possible, provide reference to existing DBS language specific coding guidelines.

This document provides guidance for the following secure coding practices:

No.	Practice	Purpose
I	**Prevent Information Leakage**	Criminals use automated tools or perform manual assessment to detect as much information about your users and the application. Preventing information and slowing down an attacker (might enable detection) is key.
II	**Input Validation**	Correctly implemented input validation will prevent application errors from unexpected input due to user typing errors and hacking attempts.
III	**Encryption**	Proper use of encryption makes it impossible for an attacker to spy on the communication or interfere with it.
IV	**Output Encoding**	Output encoding, together with input validation, is the recommended countermeasure to thwart a Cross-Site-Scripting (XSS) attack.
V	**Parameterized Queries**	Parameterized queries are the number one protective measure against SQL Injection (SQLi) attacks.

5 PRACTICE I – PREVENT INFORMATION LEAKAGE

5.1 Introduction

Attackers need to gather as much information as possible about the application and the users of the application at the initial stages of their attack.

This section provides guidance on how developers can make information gathering more difficult for the attacker. By making this stage more difficult, the attacker will require more time and a higher set of skills to succeed, hence increasing the chance that the organisation will detect the attack.

5.2 Considerations

To attack an application, the attacker will require access to the application just like a regular user. Exposing too much information to unauthenticated users makes it easier for the attacker to gather the information he requires for the attack. To slow down the attacker, the application can limit access to certain information for authenticated users only.

Typically, attackers will use automated tools to detect hidden information leaked by an application. These tools use long lists of commonly used application requests. Developers can implement different request names to at least stop the script kiddies and slow down the black hat.

5.3 Approaches

Developers can consider the following potential approaches to preventing information leakage:

- Avoid exposing information about users and administrators;
- Limit access to information about application functionality assigned to specific roles, or at the very least discern between unauthenticated users, authenticated users and administrators; and
- Obfuscate information about application functionality.

5.4 Countermeasures

5.4.1 Avoid Exposing User Information

Typically, the application might leak user information when the authentication system reveals whether a specific username is available in the application.

5.4.1.1 Implement a Generic Response in case of Authentication Failure

The application should respond with a generic message (e.g. "Invalid username or password, please try again") when authentication fails. The application should never reveal which of the specific credential (username or password) was wrong.

5.4.1.2 Implement a Generic Response for Password Reset

The "password reset" function might reveal if a username exists in the application or not. If the function is implemented via email, the application should again respond with a generic message (e.g. "A password reset token has been send to your registered email address. In case you did not receive this email, please contact the helpdesk.”). By using this generic message, we do not reveal whether the username or registered email address exists in the application.

5.4.1.3 Slow Down Potential Brute-Force Attempts

When the attacker knows the valid username(s), he might attempt to brute-force the password. Depending on application, the developer can:

- Implement a limit on the maximum number of unsuccessful authentication attempts after which the account is locked. This might however result in a Denial-of-Service attack as the attacker might just perform many failed authentication attempts to lock the user account(s). In this situation, the helpdesk may have a resource issue if an application has many users.
- Implement a limit on the maximum number of failed authentication attempts within a given period. This is often implemented by making the user wait a certain period before the next authentication attempt.

5.4.2 Limit Information about Application Functionality

5.4.2.1 Split JavaScript Files per Role

In a Single Page Application (SPA), it is common to send the whole application to the user’s browser even before authentication is performed. This allows anyone to detect all possible requests (most commonly REST or JSON requests) in the application, even if the functionality is only supposed to be available to certain roles after a successful authentication.

The application should only provide a login page and avoid sending the JavaScript file containing the SPA before authentication is completed. After a successful authentication, the application should only send the SPA containing the available requests based on the user role (e.g. a different SPA for administrators and regular users).

5.4.2.2 Remove the WSDL File

A Web Services Description Language file (WSDL) details all publicly exposed methods and parameters. Instead of brute-forcing, an attacker can immediately abuse this information and target the documented methods. It is also common for this file to include internal methods, which should not be exposed.

Developers should remove the WSDL file if the service is for server-to-server communication, and it is not necessary for everyone to access the services. It is always possible to separately distribute this file to developers or partners that need it. This approach will stop the script kiddie and make it more difficult for the black hat. If the WSDL is not available, the attacker will have to either reverse engineer the client or brute-force the methods or parameters.

5.4.3 Hide or Obfuscate Information

While obfuscation is not real security, a dedicated attacker with available time and resources might still be able to extract the information necessary to facilitate his attack. However, this will require the attacker to possess a higher level of skills and stop script kiddies.

5.4.3.1 Minify your JavaScript

The process of minifying JavaScript will remove comments and make the code a lot harder to understand. There are standard tools available to completely automate this task.

Obfuscators for JavaScript do exist but are also known to cause problems and is probably not worth the effort to implement them.

5.5 Additional Reading

Link	Description
https://cheatsheetseries.owasp.org/cheatsheets/Authentication_Cheat_Sheet.html	The OWASP Authentication Cheat Sheet, detailing best practices to implement authentication.

6 PRACTICE II -INPUT VALIDATION

6.1 Introduction

Many applications are vulnerable because of weaknesses related to input validation, including:

- Injection type vulnerabilities, such as SQL injection (SQLi) and Cross-Site Scripting (XSS)
- Path manipulation attacks (file upload and download)

These attacks might succeed when developers do not check that entered data is of the expected format such as a phone number or NRIC. This allows attackers to inject deliberately malformed data that is targeted to attack back-end systems such as databases and LDAP servers, the browser or the operating system of the server.

It is important not to consider input validation as the only line of defines as many of the possible web-based attacks will require additional countermeasures. However, proper input validation will make the attacker work much harder, defy automated scans, buy you some time or – in some cases –block real-world attacks.

An additional benefit of input validation is that malformed data from a malfunctioning component or service can be detected and handled early.

6.2 Considerations

6.2.1 What to Validate

The general principle is to validate all *untrusted* data. But what data is trustworthy? Even if contracts with service providers or partners make them accountable for security breaches or the consequences thereof, it is better to be safe than sorry. Criminals have been known to attack service providers first with the goal of attacking their customers.

To keep the design of an application simple - the KISS ("Keep it simple, stupid") principle – it is better to validate *all* data even if this comes from other banks, SWIFT, or regulatory authorities. After all, they might have been compromised, or might be sending malformed data accidentally because of a bug or problem in their systems.

6.2.2 When to Validate

As a best practice, validate the input as soon as it enters the application, before any further processing. When data is processed, it might be too late to stop the injection of malicious data.

You should also apply input validation at the point of entry when data crosses a boundary, such as one application to another.

6.2.3 Where to Validate (Client-Side or Server-Side)

At the very least, you should apply input validation at the server-side as client-side filtering is never enough to block an attack on the server interface. Remember:

- An attacker can bypass all client-side filtering, e.g. by disabling or modifying JavaScript; and
- An attacker does not have to use your client-side application at all to attack the server.

However, client-side validation can still be useful to prevent additional round-trips to the server, and if the server is still receiving malformed or malicious data, there can only be two reasons:

- A bug in the client-side validation, which must be fixed; and
- An active attack where the attacker is intentionally bypassing client-side validation, which should be logged for further investigation or forensic purposes.

In case of a browser client, validation can be implemented in JavaScript and aided by HTML input tags. HTML 5 tags now include build-in checks for email, month, number, colour, date and range.

6.2.4 What to Log

The OWASP Top 10 – 2017 includes *Insufficient logging and monitoring* as one of the ten most critical web application security risks.

"Ensure all login, access control failures, and ***server-side input validation*** *failures can be logged with sufficient user context to identify suspicious or malicious accounts and held for sufficient time to allow delayed forensic analysis."*

Typical application security and control assessments will require auditors, penetration testers and source code reviewers to evaluate what is logged by your application.

6.2.5 Whitelist or Blacklist

There are two approaches to input validation: Whitelist or Blacklist.

Whitelist validation accepts what is "expected" by the application. For every possible input, the application validates and confirms that the data is received according to the application's specific requirements. If the validation fails, the application does not perform further processing and the input is refused. In other words, processing only happens when the validation succeeds. The potential downside of whitelisting is that the developer must have a good idea of what is considered valid input for every possible input field (e.g. form, API, etc.) of an application.

Blacklist validation accepts any data apart from input that contains "forbidden" characters or combinations of characters. If the input does not contain blacklisted characters or strings, the application accepts the input as valid.

"…because as we know, there are known knowns; there are things we know we know. We also know there are known unknowns; that is to say we know there are some things we do not know. But there are also unknown unknowns – the ones we don't know we don't know."

Donald Rumsfeld

U.S. Secretary of Defense, 2001-2006

A blacklist will only protect against the things we know are bad. Any new attack (the unknowns) will result in the application being vulnerable and the blacklist must be updated. On the contrary, a whitelist is based on what the application accepts as valid input. This only needs updates when some valid formats change, or some new functionality is introduced.

The correct order to implement input validation is:

- Whitelist: accept what is valid input (this will also catch a lot of ‘unknown’ new attacks); and
- Blacklist: block what is known to be bad (blacklist anything that still might pass the whitelist).

6.3 Approaches

6.3.1 Whitelisting

Whitelisting is best implemented at the initial coding stages as it can be difficult to put in place for an existing live application. Detailed requirements and documentation of data formats and flow is key to a successful implementation.

In general, a developer should consider the following options to implement the best possible (most strict) whitelist validation:

- Limit the size of input;
- Verify the type;
- Verify the boundaries;
- Verify the permitted characters;
- Verify the syntax;
- Verify business rules; and
- Implement proper error-handling and logging.

This is further explained in the sections below.

6.3.1.1 Limit the Size of Input

The main principle here is to only accept the shortest maximal size possible. By accepting more characters, attackers will be able to use more malicious input.

The application should never accept as valid the size provided by HTTP fields or HTML parameters as an attacker can tamper with these values. Typical attacks include:

- Putting zero or negative sizes in the client-controlled fields, which might result in unhandled error conditions; and
- Putting a longer length than the input really is, which might result in the application waiting forever for input that never comes – a potential Denial of Service (DoS) situation;

6.3.1.2 Verify the Type

The application should verify the expected type, e.g. numeric, alphanumeric, etc. Some languages implement stronger type-checking, and the developer should apply the strongest possible. It is also important to be careful of languages that perform automated type conversion.

Allowing alphanumeric input in numeric fields is still one of the root causes of SQL Injection (SQLi).

6.3.1.3 Verify the Boundaries

The application should verify that numeric input is within boundaries and are correctly signed (positive or negative) to ensure that later processing does not lead to an overflow of bits or unexpected totals.

6.3.1.4 Verify the Permitted Characters

Typically, an attacker need ‘special’ characters to stage an attack. Limit the allowed characters as much as possible. Limiting to [AZ][az][09] is good but might still be too lenient if the expected input is a FIN number.

6.3.1.5 Verify the Syntax

The application should verify inputs with specific syntax, e.g. postal code, phone number, NRIC, etc.

6.3.1.6 Verify Business Rules

Many applications expect data to conform to specific business, e.g. a financial penalty could be determined as a certain percentage with maximum and minimum value. It will be nearly impossible for an attacker to bypass certain business rules.

6.3.1.7 Implement Proper Error-Handling and Logging

The application should catch and log all input validation errors. In addition, the application should continue to work, even if unexpected data is received.

6.3.2 Blacklisting

Consider the situation where you have just received a vulnerability report, how can you find the optimal approach when blacklisting seems like the best strategy?

When a vulnerability report is received, and blacklisting is an option, the following process should be followed to eliminate the bug:

Step 1: Research

- Determine the source of the input
- Determine the sink (where the input does the damage)

Step 2: Review

- Password field? Choose other option.
- Business required input? Choose other option.
- Determine unwanted input (characters or string)

Step 3: Fix

- Implement client-side code to refuse the unwanted input
- Implement server-side code to refuse the unwanted input
- Modify client-side display or error-handling to not display the unmodified input if blacklisted characters or strings where detected

In the next sections, we will discuss the implementation approaches in greater detail.

6.3.2.1 Filtering Bad Characters or Blocking the Input

When blacklisting certain characters or combinations of characters, two approaches are possible:

- Filter the input and remove unwanted and potentially dangerous characters; and resume regular processing with the cleaned input; or
- Check the input for unwanted characters and stop further processing when unwanted input is detected.

The first option could be written as this:

```
GET input-string
REMOVE blacklisted characters or string
PROCEED
```

However, this might not be the solution: what would happen when the string **`<SCRIPT>`** is blacklisted, but **`<SCRI<SCRIPT>PT>`** is entered?

A solution would be to perform the filtering until the unwanted string is not found anymore, such as:

```
GET input-string
WHILE blacklisted characters or string in input-string
REMOVE blacklisted characters or string
END WHILE
PROCEED
```

This does the job but takes longer and might enable an attacker to enter a very long string, that takes several loops to be cleaned. When the pattern-matching is implemented with regular expressions, attackers have been known to attack this mechanism, resulting in a Denial-of-Service attack. This is known as the “ReDOS” attack.

Another problem is that user-supplied input is modified, and this might be unwanted for audit reasons.

A better approach is to check the input, and block further processing when blacklisted input is detected:

```
GET input-string
IF input-string contains blacklisted characters or string
REFUSE input-string
LOG error
RETURN to user
ELSE
      PROCEED
END-IF
```

6.3.2.2 Anti-Pattern

There are a few cases where you should not use blacklisting:

- For a password field, secure passwords should contain a lot of entropy, limiting certain characters makes it easier for an attacker to brute-force the password. A possible solution would be to use BASE64 encoding to enable further processing.

- When the specific characters or combinations thereof must be allowed according to user requirements.

6.4 Countermeasures

While we should always consider white-listing as the preferred approach, in this section, we will discuss in detail the best options for blacklisting as a countermeasure for the most common vulnerabilities such as Cross-Site-Scripting (XSS) and Path Manipulation.

The main problem with the earlier documented process is to determine the list of unwanted characters or input. A vulnerability report might provide a exploit string such as **“<IMG SRC=1 ONERROR=alert(1)”>**, but which characters should be blocked? Blocking only the malicious characters used by the penetration testers, might not be enough to eradicate the bug.

In the following sections, we provide the background needed to determine dangerous characters or strings for Cross-Site-Scripting (XSS) and Path Manipulation vulnerabilities.

6.4.1 Cross-Site Scripting (XSS)

A Cross-Site-Scripting vulnerability is caused by echoing user-supplied input in an HTML page without proper validation. The result is that malicious JavaScript will run in the context of the victim.

Depending on where (e.g. HTML, CSS or JavaScript) your code echoes the malicious input, different characters pose a danger and must be blacklisted.

The following table provide the most common locations where user-supplied input might be echoed, and an example on how an attacker would attack this.

Context	Example

HTML Context	`<tag>echoed_input<tag>`
	
	Any valid HTML will be accepted, the attacker could enter:
	`<img src=x onerror=alert(1)>`
	
	Which would result in:
	`` <tag><img src=x onerror=alert`1`></tag> ``
	
	Consider blacklisting the following characters:
	`` <, >, =, /, `, (, ) ``

HTML Attribute Name Context	`<tag echoed_input attribute=”xyz”>`
	
	An attacker could add an event-handler name, followed by the equal sign, and followed by JavaScript. This would result in something like this:
	`<tag onclick=ask(1) attribute=”xyz”>`
	
	Consider blacklisting the following characters:
	`` <, >, =, `, (, ) ``

Context	Example
	and maybe all event-handlers (this might be impossible).
HTML Attribute Value Context	There are three variants: `<tag attribute=”echoed_input”>` `<tag attribute=’echoed_input’>` `<tag attribute=echoed_input>` If the attribute is a regular attribute, the following inputs will result in code execution: `<tag attribute=”” onclick=alert(1)”>` `<tag attribute=’‘ onclick=alert(1)”> ” onclick=alert(1)”>’>` `<tag attribute= onclick=alert(1)”>>` Secure coding guidelines should prescribe the use of double-quoted attributes. Consider blacklisting: **‘, “, =, (,), `**  If not a regular attribute, code execution is still possible: a. Event attribute: Supplied input will be executed as JavaScript b. URL attribute: Input such as **`javascript:prompt(1)`** could work Again, it will be difficult to block the attack in [a] by just blacklisting (see notes on JavaScript context). For [b], consider blacklisting (case-insensitive comparison) the string **`javascript`**.

HTML Comments Context	`<!-- comment echoed_input comment -->`
	
	Code execution is possible by ending the comment:
	`<!-- comment --><img src=x onerror=alert(1) comment -->`
	
	Consider blacklisting the following characters:
	-, !, >

JavaScript Context	JavaScript has its own contexts, which have “dangerous characters” then HTML context. Even more dangerously, it is possible to switch to HTML context.
	Consider the following code:
	`<script>`
	`    a = “echoed_input”`
	`</script>`
	
	An attacker can switch to HTML context by simply inserting this:
	`<script>`
	`    a = “</script><script>alert(1)</script>\\”`
	`</script>`

Context	Example
	The recommendation is to never echo user-supplied input in JavaScript. If a short-term solution must be implemented, make it very difficult for the attacker – the attacker needs to break out of wherever the insertion is, and the resulting code must be valid JavaScript or the browser will prevent it from executing. The following characters should be considered for blacklisting: \, /, ”, ’, `, {, }, [,], (,), %, ;, -, <, >, !, =
Older contexts	With older browsers, it might be possible to insert VBScript or put JavaScript in CSS context. This is not possible anymore with recent browsers.

6.4.2 Path Manipulation

A Path manipulation vulnerability happens when your application expects a file-name, but the attacker sends a full path, or tries to get access to other directories by using “../” (go to one directory higher).

Characters to be blacklisted are: “..”,”/” and “\”. It might be prudent to blacklist “:” too (for Windows paths) and even “.” (many important file-names under Unix start with a “.”, e.g. the Apache .htaccess file which might contain user-ids and passwords).

6.5 Additional Reading

Link	Description
https://cheatsheetseries.owasp.org/cheatsheets/Error_Handling_Cheat_Sheet.html	The OWASP Error Handling
https://cheatsheetseries.owasp.org/cheatsheets/Logging_Cheat_Sheet.html	The OWASP Logging Guide
https://cheatsheetseries.owasp.org/cheatsheets/SQL_Injection_Prevention_Cheat_Sheet.html	The OWASP SQL Injection Prevention Cheat Sheet documents counter-measures against SQLi.
https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html	The OWASP XSS Prevention Cheat Sheet documents counter-measures against XSS.
https://cheatsheetseries.owasp.org/cheatsheets/Input_Validation_Cheat_Sheet.html	The OWASP Input Validation Cheat Sheet

7 PRACTICE III – ENCRYPTION

7.1 Introduction

Encryption is often mandated by regulatory authorities or the law. Sensitive customer data and confidential information at DBS must be protected against theft, unauthorized modification or destruction.

The following techniques can be used:

- **Encryption**: Symmetrical or asymmetrical cryptography (public key cryptography) can be applied to encrypt data during transfer or at rest. The protection is dependent on keeping the encryption key secret and provides confidentiality between sender and receiver. Attackers will focus on getting access to the symmetrical key or private key in the case of asymmetrical cryptography.
- **Hashing**: A hashing function uses a mathematical formula to create a unique result from an input. It is not, or it should not be possible to recreate the input by studying the resulting hash. If the input is relatively short (e.g. passwords), attackers have pre-computed lists (rainbow lists) of all possible passwords of a certain length. If they get access to the hashes, they can do a simple and fast lookup in the rainbow list.
- **Digital Signatures**: A combination of hashing and public key cryptography that is used to protect the integrity of information and provide proof of the authenticity of the sender. It can also be used to provide non-repudiation. Attackers will focus on gaining access to the private key of the certificate authority (CA), users and systems, or exploit weaknesses in the protocols and ciphers used.

Breaking encryption protocols or attacking weak implementations is generally the domain of organized crime or governments.

Standards prescribe what are the best practices to be applied and might dictate specific protocols or mechanisms. Developers should refer to the DBS Encryption Policy Guidelines which is aligned with the relevant requirements.

While it is possible for an attacker of any skill level to detect whether data (e.g. personal data, session keys, credentials, etc.) is properly encrypted during communication or storage, many of the vulnerabilities related to the implementation can only be exploited by expert hackers since most of the available tools are geared towards brute-forcing. In most cases, once cryptographers document weaknesses in protocols or ciphers, automated tools will be freely available shortly thereafter.

7.2 Considerations

Cryptographers are specialist 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.

Using a real case study based on an application review, developers tried to improve the strength of encryption by individually encrypting every single bit of a password. Since the protection offered by encryption protocols depends on the length, type and entropy of data, and passwords are relatively short and contain a predictable input (i.e. alphanumeric and special characters), the result is trivially reversible. Since a bit can only have the value 1 or 0, each encrypted bit can only have the value of unencrypted (0) or encrypted (1).

Many protocols depend on random numbers. Computers are however deterministic and can only create pseudo-random numbers. Protocols exist to implement this in a secure way and similarly, any attempt at improving these protocols will most probably result in a weaker pseudo-random number.

Developers should always use a framework’s mature encryption libraries to implement encryption. Attempts to copy random “sample” implementations of encryption ciphers will most likely result in a poor implementation that can be easily broken. Mature libraries are studied or in some cases even certified. Even if these implementations contain weaknesses, it is always easier to upgrade a library than to update your code.

Developers should be aware that choosing a good cipher but using the wrong key-length can still result in an insecure result.

7.3 Approaches

If encryption needs to be applied, always follow this approach:

- Choose the protocol recommended by the DBS Encryption Policy Guidelines
- Review the proper use of the protocol or cipher and use recommended key lengths according to policy
- Use functions from a standard library to code

7.4 Countermeasures

7.4.1 Passwords

Password storage is discouraged, and developers should use the Centralised AD Infrastructure.

In case it is necessary to store passwords, the current best practice is to apply a PBKBF2 (Password-Based Key Derivation Function), although the winner of a recent password hashing algorithm is Argon2, which may gain popularity soon.

PBKBF2 is now part of standard libraries and toolkits (such as “javax.crypto”), which make implementation relatively straightforward.

PBKBF2 and similar algorithms make it infeasible for an attacker having access to advanced hardware to brute-force the result, while still making it possible in a short time to verify if any given password corresponds with a given result.

7.4.2 Service Passwords

It is often necessary to store API secrets or service account passwords. The following principles should be applied:

- Never hard-code the secret or password
- Use a separate configuration file (or keystore, environment variable) to store the secrets or passwords
- Use the method offered by the framework to encrypt the passwords and secrets
- Configure the version control system to not check in the configuration file or keystore with the encrypted secrets

Multiple frameworks or servers have dedicated methods to protect secrets in configuration files:

- NET - https://msdn.microsoft.com/en-us/library/zhhddkxy%28v=vs.140%29.aspx
- Websphere - https://www.ibm.com/support/knowledgecenter/en/SSAW57_8.0.0/com.ibm.websphere.nd.doc/info/ae/ae/tsec_protplaintxt.html
- Jetty – http://www.eclipse.org/jetty/documentation/current/configuring-security-secure-passwords.html

7.5 Additional Reading

Link	Description
http://megaweb1.sgp.dbs.com/advisor/policyrepository/documents/DBS-11-S-0006-GR-Cryptography-Standard.pdf	DBS Cryptography standards
http://megaweb1.sgp.dbs.com/advisor/policyrepository/documents/DBS-11-G-0056-GR-Cryptography-Usage-Guidelines.pdf	DBS Cryptography Usage Guidelines
https://cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html	The OWASP Password Storage cheat sheet
https://cheatsheetseries.owasp.org/cheatsheets/Choosing_and_Using_Security_Questions_Cheat_Sheet.html	The OWASP Choosing and Using Security Questions Cheat Sheet
https://cheatsheetseries.owasp.org/cheatsheets/Forgot_Password_Cheat_Sheet.html	The OWASP Forgot Password Cheat Sheet
https://cheatsheetseries.owasp.org/cheatsheets/Cryptographic_Storage_Cheat_Sheet.html	The OWASP Cryptographic Storage Cheat Sheet
https://cheatsheetseries.owasp.org/cheatsheets/Transport_Layer_Protection_Cheat_Sheet.html	The OWASP Transport Layer Protection Cheat Sheet

8 PRACTICE IV – OUTPUT ENCODING

8.1 Introduction

As we have seen earlier, Cross-Site Scripting (XSS) attacks can be blocked by implementing input validation. Another possibility is to use output to renders characters that might have a special meaning to the browser harmless.

The problem is that proper output encoding is depending on the context where the input is echoed. As an example:

- White-space characters are ignored in HTML context; and
- Specific white-space characters (CR, LF) in JavaScript are line terminators and start a new command.

This makes it necessary to know exactly where the injection happens before proper and context-sensitive output encoding can be applied.

In practice, to fix a bug related to XSS, the best approach is a combination of input validation and output encoding.

8.2 Considerations

Output encoding, if applied properly, is a very effective method to block XSS attacks. However, if user-supplied input is echoed into a JavaScript context, an attacker could switch to another context which will need a different output encoding mechanism. It is never possible to fully block an XSS attack when unfiltered input is echoed into the JavaScript context (refer to Section 6.4.1).

If user-supplied input needs to be echoed into JavaScript, a redesign of the application is preferred. If that is not feasible, developers should apply the strictest possible input validation methods combined with the encoding required for JavaScript (refer to Section 8.4).

8.3 Approaches

Consider the situation where you have just received a vulnerability report related to XSS, how can you find the optimal approach for using output encoding as one of the counter-measures?

The following process should be followed to eliminate the bug:

Step 1: Research

- Determine the source of the input
- Determine the sink (where the input does the damage)

Step 2: Review

- Determine the context of the sink (e.g. HTML, attribute, JavaScript, etc.)

Step 3: Fix

- Encode the output according to the correct context
- Review the code if stricter input validation (server-side and client-side) is possible.

8.4 Countermeasures

The countermeasure is to encode characters that have a special meaning for the context they are echoed in into a harmless representation. Please refer to the table in Appendix 3.

Depending on the language and framework used, the following standard libraries and or functions should be considered:

- The OWASP ESAPI - https://wiki.owasp.org/index.php/Category:OWASP_Enterprise_Security_API
- Microsoft .NET AntiXSS - https://www.microsoft.com/en-us/download/details.aspx?id=43126

8.5 Additional Reading

Link	Description
https://owasp.org/www-project-java-encoder/	The OWASP Java Encoder Project
https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html	The OWASP XSS Prevention Cheat Sheet documents counter-measures against XSS (DOM-based, Reflected and Stored XSS).

9 PRACTICE V – PARAMETERIZED QUERIES

9.1 Introduction

The use of parameterized queries is the most effective method to counteract a SQL Injection (SQLi) vulnerability. This is a potentially devastating attack where an attacker can abuse your application to communicate using SQL with the database backend. A successful attack might even result in access to the underlying operating system using default stored procedures or DBMS-specific SQL extensions to the standard SQL language.

The root cause of a SQLi vulnerability is the use of dynamic database queries that include user supplied input.

In practice, the best approach to fix a SQLi bug is the use of prepared statements with parameterized queries, potentially supplemented with input validation. As the protection of parameterized queries is based on properties of the DBMS and its drivers, a bug there might still allow an attacker to bypass the protection provided by parametrized queries. Hence, server-side input validation is still recommended.

9.2 Considerations

In general, it is highly recommended to use a standard framework to implement parameterized queries. Most modern frameworks provide such implementations.

Relying on input validation alone will never be enough to protect the DBMS against SQLi attacks, and advanced tools exist to exploit SQLi-related weaknesses.

Even if the attacker does not get direct answers from the DBMS (e.g. Blind SQLi attack), it is still possible for information to be extracted, deleted or inserted when an attacker can make a difference between a “true” and a “false” result. Once that situation has been established, the attacker can add additional queries (e.g. is the first character of the database name an ‘a’). Potential “true” or “false” answers can be derived from the content of the response (e.g. value, length, etc.) or the time it takes to generate a response.

The application should never display any technical information (e.g. DBMS version, content of the query, etc.) related to the error to the attacker. Just based on this information leakage, existing attack tools can speed up attacks by up to a thousand-fold.

9.3 Approaches

Consider the situation where you have just received a vulnerability report related to SQLi, how can you find the optimal protection?

Step 1: Research

- Determine the source of the input
- Determine the sink (where the input does the damage)

Step 2: Review

- Review the specific query that caused the injection

Step 3: Fix

- Implement parameterized queries and bind variables

- Review the code for proper error-handling (e.g. the message displayed to the end user should be generic and not include any technical details)
- Review the code if stricter input-validation (server-side) is possible.

9.4 Countermeasures

The implementation is highly dependent on the programming language, framework and DBMS used:

- **Hibernate**: Use **`createQuery()`** with named parameters
- **Java EE**: Use **`PreparedStatement()`** with bind variables
- **.NET**: Use parameterized queries like **`SqlCommand()`** or **`OleDbCommand()`** with bind variables or parameterized queries with LINQ
- **PHP**: Use PHP Data Objects (PDO) with strongly typed parameterized queries using **`bindParam()`**
- **SQLite**: Use **`sqlite3_prepare()`** to create a statement object

The application should never connect to a DMBS using an account with root or DBA access, but rather an application user account that gives only the access needed to implement the required business functionality should be used.

9.5 Additional Reading

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

APPENDIX 1 ADDITIONAL REFERENCES

Reference	Description
DBS The SDLC Portal	You can find the portal at https://dbs1bank.sharepoint.com/sites/sg_EASRE/SitePages/SDLC/SDLC.aspx
DBS Access Control Standards	You can find the standards at http://mydbs.net/uploadedFiles/Departments/Technology_and_Operations/Singapore/Technology_Services/Info_Security_Services/Information/Policies,_Standards_and_Guidelines/access_control_std.pdf
DBS eBanking Standard	You can find the standards at http://megaweb1.sgp.dbs.com/advisor/policyrepository/documents/DBS-11-S-0009-GR-eBanking-Standard.pdf
DBS Application Security Engineering Guidelines	You can find the standards at http://megaweb1.sgp.dbs.com/advisor/policyrepository/documents/DBS-11-G-0066-GR-Application-Security-Engineering-Guide.pdf
The OWASP Cheat Sheets Series	A list of all OWASP Cheat Sheets, each describing what must be done to counteract a specific web application vulnerability. You can find the list at https://cheatsheetseries.owasp.org/index.html
The OWASP Code Review Guide	The OWASP Code Review Guide provides recommendations for a code review process and provides detailed information about what to check. You can find the guide at https://owasp.org/www-project-code-review-guide/migrated_content
The OWASP Secure Coding Practices	The OWASP Secure Coding Practices Quick Reference Guide provides a set of set of general software security coding practices. You can find the guide at https://owasp.org/www-project-go-secure-coding-practices-guide/
The OWASP Top 10 Proactive Controls	The OWASP Top 10 Proactive Controls documents the most important control and control categories that every architect and developer should include in every project. You can find the guide at https://owasp.org/www-project-proactive-controls/

APPENDIX 2 ACRONYMS

Acronym	Description
OWASP	The **Open Web Application Security Project** is a non-profit organization focused on improving the security of software. They are well-known for the **OWASP Top 10 list**, used as the yard-stick by penetration testers and IT security auditors. They also publish the Developer Guide and Code Review Guide. You find them at https://www.owasp.org.
SPA	A **Single Page Application**, consisting of a client-side JavaScript.
SQLi	SQL Injection
WSDL	A **Web Services Description Language** file details publicly exposed methods and parameters for Web Service requests.
XSS	Cross-Site-Scripting

APPENDIX 3 CHARACTER ENCODING

[redacted]

Char #	ASCII Char	Encode for HTML	Encode for HTML Attribute	Encode for CSS	Encode for JavaScript	Encode for URL
0				\0	\x00	%00
1				\1	\x01	%01
2				\2	\x02	%02
3				\3	\x03	%03
4				\4	\x04	%04
5				\5	\x05	%05
6				\6	\x06	%06
7				\7	\x07	%07
8				\8	\x08	%08
9		&#x9;	&#x9;	\9	\x09	%09
10		&#xa;	&#xa;	\a	\x0A	%0A
11				\b	\x0B	%0B

Char #	ASCII Char	Encode for HTML	Encode for HTML Attribute	Encode for CSS	Encode for JavaScript	Encode for URL
12				\c	\x0C	%0C
13		&#xd;	&#xd;	\d	\x0D	%0D
14				\e	\x0E	%0E
15				\f	\x0F	%0F
16				\10	\x10	%10
17				\11	\x11	%11
18				\12	\x12	%12
19				\13	\x13	%13
20				\14	\x14	%14
21				\15	\x15	%15
22				\16	\x16	%16
23				\17	\x17	%17
24				\18	\x18	%18
25				\19	\x19	%19
26				\1a	\x1A	%1A
27				\1b	\x1B	%1B
28				\1c	\x1C	%1C
29				\1d	\x1D	%1D
30				\1e	\x1E	%1E
31				\1f	\x1F	%1F
32				\20	\x20	%20
33	!	&#x21;	&#x21;	\21	\x21	%21
34	"	&quot;	&quot;	\22	\x22	%22
35	#	&#x23;	&#x23;	\23	\x23	%23
36	$	&#x24;	&#x24;	\24	\x24	%24
37	%	&#x25;	&#x25;	\25	\x25	%25
38	&	&amp;	&amp;	\26	\x26	%26

Char #	ASCII Char	Encode for HTML	Encode for HTML Attribute	Encode for CSS	Encode for JavaScript	Encode for URL
39	'	&#x27;	&#x27;	\27	\x27	%27
40	(	&#x28;	&#x28;	\28	\x28	%28
41	)	&#x29;	&#x29;	\29	\x29	%29
42	*	&#x2a;	&#x2a;	\2a	\x2A	*
43	+	&#x2b;	&#x2b;	\2b	\x2B	+
44	,	,	,	\2c	,	%2C
45	-	-	-	\2d	\x2D	-
46	.	.	.	\2e	.	.
47	/	&#x2f;	&#x2f;	\2f	\x2F	/
48	0	0	0	0	0	0
49	1	1	1	1	1	1
50	2	2	2	2	2	2
51	3	3	3	3	3	3
52	4	4	4	4	4	4
53	5	5	5	5	5	5
54	6	6	6	6	6	6
55	7	7	7	7	7	7
56	8	8	8	8	8	8
57	9	9	9	9	9	9
58	:	&#x3a;	&#x3a;	\3a	\x3A	%3A
59	;	&#x3b;	&#x3b;	\3b	\x3B	%3B
60	<	&lt;	&lt;	\3c	\x3C	%3C
61	=	&#x3d;	&#x3d;	\3d	\x3D	%3D
62	>	&gt;	&gt;	\3e	\x3E	%3E
63	?	&#x3f;	&#x3f;	\3f	\x3F	%3F
64	@	&#x40;	&#x40;	\40	\x40	@
91	[	&#x5b;	&#x5b;	\5b	\x5B	%5B

Char #	ASCII Char	Encode for HTML	Encode for HTML Attribute	Encode for CSS	Encode for JavaScript	Encode for URL
92	\	&#x5c;	&#x5c;	\5c	\x5C	%5C
93	]	&#x5d;	&#x5d;	\5d	\x5D	%5D
94	^	&#x5e;	&#x5e;	\5e	\x5E	%5E
95	_	_	_	\5f	_	_
96	`	&#x60;	&#x60;	\60	\x60	%60
123	{	&#x7b;	&#x7b;	\7b	\x7B	%7B
124	|	&#x7c;	&#x7c;	\7c	\x7C	%7C
125	}	&#x7d;	&#x7d;	\7d	\x7D	%7D
126	~	&#x7e;	&#x7e;	\7e	\x7E	%7E
127				\7f	\x7F	%7F
128				\80	\x80	%80
129				\81	\x81	%81
130	‚			\82	\x82	%82
131	ƒ			\83	\x83	%83
132	„			\84	\x84	%84
133	…			\85	\x85	%85
134	†			\86	\x86	%86
135	‡			\87	\x87	%87
136	ˆ			\88	\x88	%88
137	‰			\89	\x89	%89
138	Š			\8a	\x8A	%8A
139	‹			\8b	\x8B	%8B
140	Œ			\8c	\x8C	%8C
141				\8d	\x8D	%8D
142				\8e	\x8E	%8E
143				\8f	\x8F	%8F
144				\90	\x90	%90

Char #	ASCII Char	Encode for HTML	Encode for HTML Attribute	Encode for CSS	Encode for JavaScript	Encode for URL
145	‘			\91	\x91	%91
146	’			\92	\x92	%92
147	“			\93	\x93	%93
148	”			\94	\x94	%94
149	•			\95	\x95	%95
150	–			\96	\x96	%96
151	—			\97	\x97	%97
152	˜			\98	\x98	%98
153	™			\99	\x99	%99
154	š			\9a	\x9A	%9A
155	›			\9b	\x9B	%9B
156	œ			\9c	\x9C	%9C
157				\9d	\x9D	%9D
158				\9e	\x9E	%9E
159	Ÿ			\9f	\x9F	%9F
160		&nbsp;	&nbsp;	\a0	\xA0	%A0
161	¡	&iexcl;	&iexcl;	\a1	\xA1	%A1
162	¢	&cent;	&cent;	\a2	\xA2	%A2
163	£	&pound;	&pound;	\a3	\xA3	%A3
164	¤	&curren;	&curren;	\a4	\xA4	%A4
165	¥	&yen;	&yen;	\a5	\xA5	%A5
166	¦	&brvbar;	&brvbar;	\a6	\xA6	%A6
167	§	&sect;	&sect;	\a7	\xA7	%A7
168	¨	&uml;	&uml;	\a8	\xA8	%A8
169	©	&copy;	&copy;	\a9	\xA9	%A9
170	ª	&ordf;	&ordf;	\aa	\xAA	%AA
171	«	&laquo;	&laquo;	\ab	\xAB	%AB

Char #	ASCII Char	Encode for HTML	Encode for HTML Attribute	Encode for CSS	Encode for JavaScript	Encode for URL
172	¬	&not;	&not;	\ac	\xAC	%AC
173		&shy;	&shy;	\ad	\xAD	%AD
174	®	&reg;	&reg;	\ae	\xAE	%AE
175	¯	&macr;	&macr;	\af	\xAF	%AF
176	°	&deg;	&deg;	\b0	\xB0	%B0
177	±	&plusmn;	&plusmn;	\b1	\xB1	%B1
178	²	&sup2;	&sup2;	\b2	\xB2	%B2
179	³	&sup3;	&sup3;	\b3	\xB3	%B3
180	´	&acute;	&acute;	\b4	\xB4	%B4
181	µ	&micro;	&micro;	\b5	\xB5	%B5
182	¶	&para;	&para;	\b6	\xB6	%B6
183	·	&middot;	&middot;	\b7	\xB7	%B7
184	¸	&cedil;	&cedil;	\b8	\xB8	%B8
185	¹	&sup1;	&sup1;	\b9	\xB9	%B9
186	º	&ordm;	&ordm;	\ba	\xBA	%BA
187	»	&raquo;	&raquo;	\bb	\xBB	%BB
188	¼	&frac14;	&frac14;	\bc	\xBC	%BC
189	½	&frac12;	&frac12;	\bd	\xBD	%BD
190	¾	&frac34;	&frac34;	\be	\xBE	%BE
191	¿	&iquest;	&iquest;	\bf	\xBF	%BF
192	À	&Agrave;	&Agrave;	\c0	\xC0	%C0
193	Á	&Aacute;	&Aacute;	\c1	\xC1	%C1
194	Â	&Acirc;	&Acirc;	\c2	\xC2	%C2
195	Ã	&Atilde;	&Atilde;	\c3	\xC3	%C3
196	Ä	&Auml;	&Auml;	\c4	\xC4	%C4
197	Å	&Aring;	&Aring;	\c5	\xC5	%C5
198	Æ	&AElig;	&AElig;	\c6	\xC6	%C6

Char #	ASCII Char	Encode for HTML	Encode for HTML Attribute	Encode for CSS	Encode for JavaScript	Encode for URL
199	Ç	&Ccedil;	&Ccedil;	\c7	\xC7	%C7
200	È	&Egrave;	&Egrave;	\c8	\xC8	%C8
201	É	&Eacute;	&Eacute;	\c9	\xC9	%C9
202	Ê	&Ecirc;	&Ecirc;	\ca	\xCA	%CA
203	Ë	&Euml;	&Euml;	\cb	\xCB	%CB
204	Ì	&Igrave;	&Igrave;	\cc	\xCC	%CC
205	Í	&Iacute;	&Iacute;	\cd	\xCD	%CD
206	Î	&Icirc;	&Icirc;	\ce	\xCE	%CE
207	Ï	&Iuml;	&Iuml;	\cf	\xCF	%CF
208	Ð	&ETH;	&ETH;	\d0	\xD0	%D0
209	Ñ	&Ntilde;	&Ntilde;	\d1	\xD1	%D1
210	Ò	&Ograve;	&Ograve;	\d2	\xD2	%D2
211	Ó	&Oacute;	&Oacute;	\d3	\xD3	%D3
212	Ô	&Ocirc;	&Ocirc;	\d4	\xD4	%D4
213	Õ	&Otilde;	&Otilde;	\d5	\xD5	%D5
214	Ö	&Ouml;	&Ouml;	\d6	\xD6	%D6
215	×	&times;	&times;	\d7	\xD7	%D7
216	Ø	&Oslash;	&Oslash;	\d8	\xD8	%D8
217	Ù	&Ugrave;	&Ugrave;	\d9	\xD9	%D9
218	Ú	&Uacute;	&Uacute;	\da	\xDA	%DA
219	Û	&Ucirc;	&Ucirc;	\db	\xDB	%DB
220	Ü	&Uuml;	&Uuml;	\dc	\xDC	%DC
221	Ý	&Yacute;	&Yacute;	\dd	\xDD	%DD
222	Þ	&THORN;	&THORN;	\de	\xDE	%DE
223	ß	&szlig;	&szlig;	\df	\xDF	%DF
224	à	&agrave;	&agrave;	\e0	\xE0	%E0

Char #	ASCII Char	Encode for HTML	Encode for HTML Attribute	Encode for CSS	Encode for JavaScript	Encode for URL
225	á	&aacute;	&aacute;	\e1	\xE1	%E1
226	â	&acirc;	&acirc;	\e2	\xE2	%E2
227	ã	&atilde;	&atilde;	\e3	\xE3	%E3
228	ä	&auml;	&auml;	\e4	\xE4	%E4
229	å	&aring;	&aring;	\e5	\xE5	%E5
230	æ	&aelig;	&aelig;	\e6	\xE6	%E6
231	ç	&ccedil;	&ccedil;	\e7	\xE7	%E7
232	è	&egrave;	&egrave;	\e8	\xE8	%E8
233	é	&eacute;	&eacute;	\e9	\xE9	%E9
234	ê	&ecirc;	&ecirc;	\ea	\xEA	%EA
235	ë	&euml;	&euml;	\eb	\xEB	%EB
236	ì	&igrave;	&igrave;	\ec	\xEC	%EC
237	í	&iacute;	&iacute;	\ed	\xED	%ED
238	î	&icirc;	&icirc;	\ee	\xEE	%EE
239	ï	&iuml;	&iuml;	\ef	\xEF	%EF
240	ð	&eth;	&eth;	\f0	\xF0	%F0
241	ñ	&ntilde;	&ntilde;	\f1	\xF1	%F1
242	ò	&ograve;	&ograve;	\f2	\xF2	%F2
243	ó	&oacute;	&oacute;	\f3	\xF3	%F3
244	ô	&ocirc;	&ocirc;	\f4	\xF4	%F4
245	õ	&otilde;	&otilde;	\f5	\xF5	%F5
246	ö	&ouml;	&ouml;	\f6	\xF6	%F6
247	÷	&divide;	&divide;	\f7	\xF7	%F7
248	ø	&oslash;	&oslash;	\f8	\xF8	%F8
249	ù	&ugrave;	&ugrave;	\f9	\xF9	%F9
250	ú	&uacute;	&uacute;	\fa	\xFA	%FA

Char #	ASCII Char	Encode for HTML	Encode for HTML Attribute	Encode for CSS	Encode for JavaScript	Encode for URL
251	û	&ucirc;	&ucirc;	\fb	\xFB	%FB
252	ü	&uuml;	&uuml;	\fc	\xFC	%FC
253	ý	&yacute;	&yacute;	\fd	\xFD	%FD
254	þ	&thorn;	&thorn;	\fe	\xFE	%FE
255	ÿ	ÿ	ÿ	ÿ	ÿ	%FF

APPENDIX 4 VERSION HISTORY

Version	Date of Issue	Summary of Key Changes
1.0	29 Mar 2018	Initial version
1.1	30 Sep 2018	Aligned with revised code review process to assign with new document reference
1.2	30 Sep 2020	Biennial review and update of Additional Reading