Table of Contents

Mitigating OWASP 2021 Insecure Design

Narration

On screen text

DES 235

Mitigating OWASP 2021 Insecure Design

Course Overview and Objectives

Narration

This course on mitigating cryptographic failures, is defined by the OWASP 2021 Top 10 web application security risks and applies to the Software Developer, Secure Software Assessor, and Systems Security Analyst roles as defined in the NICE Workforce Framework.

After completing this course, you will understand: The secure software development lifecycle and how it can improve software security, how to use secure design patterns, the importance of threat modeling, and how to prevent excessive resource consumption.

On screen text

Course Overview and Objectives

Designed for the Software Developer, Secure Software Assessor, and Systems Security Analyst roles.

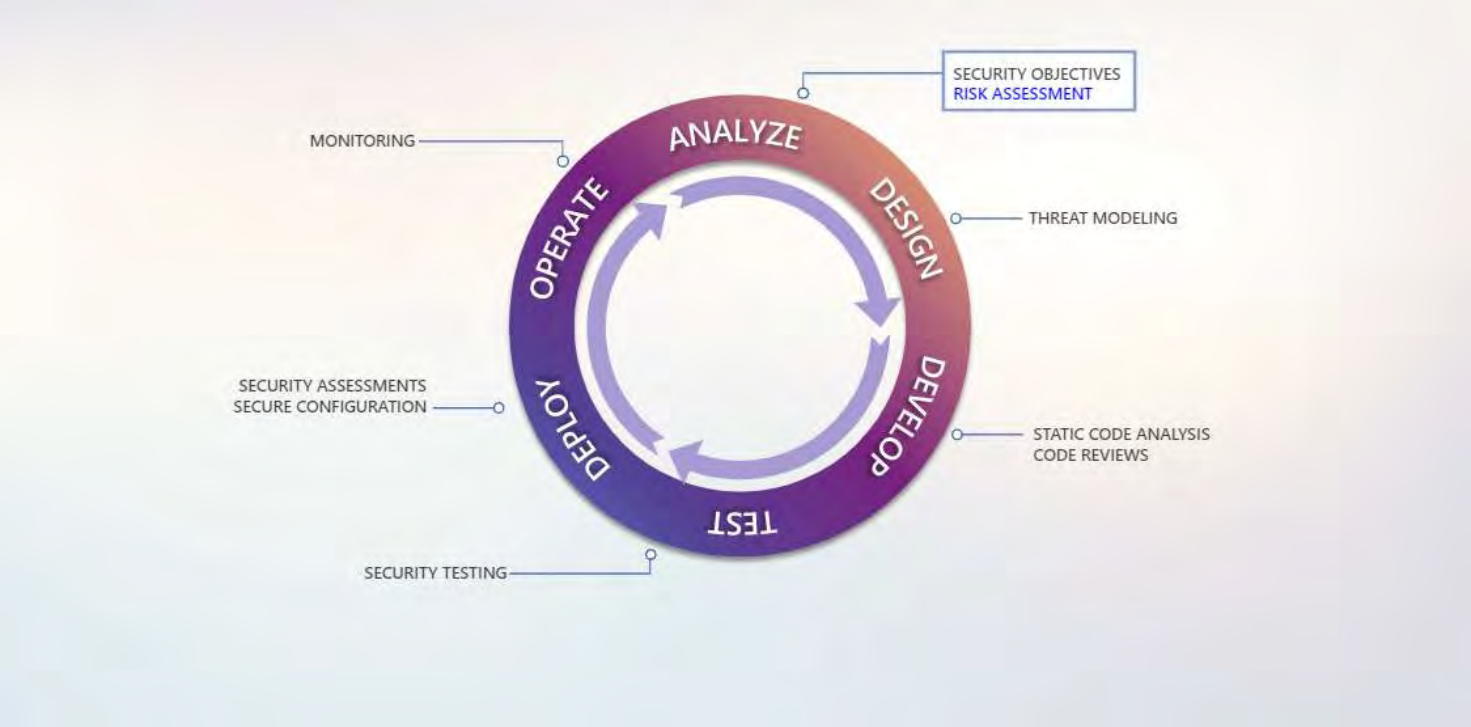
Objectives align with the OWASP 2021 Top 10 and NICE Workforce Frameworks.

After completing this course, you will understand:

- The secure software development lifecycle and how it can improve software security
- How to use secure design patterns
- The importance of threat modeling

- How to prevent excessive resource consumption

Secure Development Life Cycles



Narration

A secure software development life cycle—or SSDLC—ensures that proper security controls and safeguards become an intrinsic part of an application's development.

By treating security as a design requirement, it becomes a core part of every component of the application.

Integrating security-related tasks in each phase of the SSDLC. SSDLCs facilitate the development of a system aligned with security and compliance requirements.

Putting into action the principles of secure software engineering and architecture, an SSDLC consists of stages such as requirements analysis, design, development, testing, deployment, and operations. While the development life cycle of your organization may vary some, or be part of a larger DevOps life cycle, the principles remain the same.

The primary benefit of an SSDLC is to ensure that all stages of development follow any security controls in place. Furthermore, it shifts security left, enabling early detection of problems and significant cost savings.

Each phase in an SSDLC includes add-on security tasks critical not only in that step but that will have an impact on subsequent steps.

For example, in the initial analysis phase where you would normally determine requirements, you would also determine the security objectives and perform a risk assessment.

Continue this process with each subsequent step, integrating security activities into normal development processes.

On screen text

Secure Development Life Cycles

Secure Software Development Life Cycles

SSDLC

Security
Compliance

- SECURITY OBJECTIVES
  RISK ASSESSMENT
- THREAT MODELING
- STATIC CODE ANALYSIS
  CODE REVIEWS
- SECURITY TESTING
- SECURITY ASSESSMENTS
  SECURE CONFIGURATION
- MONITORING

Secure Design Patterns

Narration

Software security is complex.

Secure design patterns are generic methodologies for handling common security-sensitive tasks.

They help ensure that application code is secure, robust, and resilient against attacks. The patterns are

reusable, well-designed and tested, commonly accepted as the best available solution, and are flexible enough to adapt to different scenarios. Secure design patterns are necessary to avoid common security mistakes in software development.

Secure design patterns are similar to and overlap both security principles and best practices. The main difference is that secure design patterns are the best method for solving one specific problem or task, rather than general advice. Secure design patterns might cover tasks such as user authentication, security logging, or implementing cryptography.

The primary benefit of using secure design patterns is to make sure you perform common tasks in the most secure manner.

Other benefits are to:

- Identify potential security issues and likely points of failure.
- Educate developers by example and help them in implementing best practices in their own code, and

- Provide software architects and developers with a common language for expressing complex tasks.

Secure design patterns are good candidates for reusable code libraries and using a well-established library can be a suitable alternative to designing your own.

On screen text

Secure Design Patterns

Secure design patterns are generic methodologies
for handling common security-sensitive tasks.

Reusable
Well-designed
Best Solution
Flexible

Secure
Design
Patterns

Security
Principles

Best
Practices

AUTHENTICATION
SECURITY LOGGING
CRYPTOGRAPHY

Benefits

- Identify security issues
- Educate developers
- Provide a common language

Threat Modeling

Narration

Threat modeling is a process of identifying threats to a software system design, understanding the risks created by the identified threats, and understanding the appropriate security controls to mitigate those risks.

Threat modeling occurs at different levels of a software system, from individual functions to the entire system itself. Creating a threat model is a key step in any security planning process and is especially important for secure application development.

Applications use a variety of technologies, such as REST interfaces or mobile applications, and external applications such as databases or cloud services. Each of these technologies introduces additional threats, risks, and complexities that development teams must understand and address.

In addition, applications must also contend with non-technology-based risks, such as loss, theft, or even supply chain risks. A threat model can help to understand and plan for these threats.

The threat modeling process consists of five steps.

The first of these steps is to identify clear security objectives. With this step, you can determine the extent to which you should focus the threat modeling activity on subsequent steps.

Having established your objectives, create an application overview, which is important for later in the process.

Next, you will decompose your application to better understand its mechanics, inputs, and outputs.

Then, identify threats relevant to the scenario and context of your application based on your clear understanding of the application. And finally, identify your application's weaknesses corresponding to your threats and use vulnerability categories to target those areas where developers tend to make mistakes.

As you move through your application development life cycle and discover more about your application, continue to add more detail to your threat model.

Repeat the entire process as needed to continuously gain more understanding of your application's strengths and weaknesses. It is important to not get caught up in the details or implementation, rather, emphasize the overall approach. Stay focused on the primary objective: to improve security design.

On screen text

Threat Modeling

APPLICATION
THREATS

Creating a threat model is important for secure software
Application technologies introduce attack surface
There are also non-technical risks

Objectives
Identify objectives to predict security efforts

Overview
Create an application overview

Decompose
Decompose the application to understand the mechanics

Threats
Identify threats relevant to the application

Weaknesses
Identify your application's weaknesses

Security Testing

Narration

Quality assurance is key for ensuring that there are no vulnerabilities in your code. Testing in the context of software development refers to a variety of verification and validation activities that ensure an application, product, or system performs as expected and meets requirements for quality. A proper testing strategy should use a combination of techniques to ensure sufficient vulnerability coverage.

Here are the most common types of software security testing:

- **Custom test cases** – Unit, integration, and system tests that you write to validate that all critical flows address the issues on your threat model.
- **Static analysis** – Tools that automatically analyze your code to identify common patterns that might lead to security vulnerabilities. These might take the form of IDE add-ins or standalone tools you integrate into your build or deployment process.
- **Vulnerability scanning** – Automated tools that look for specific vulnerabilities, usually in web interfaces of running applications.
- **Penetration testing** – Manual testing by security experts that involves multi-step attacks that emulate a real-world intrusion, and
- **Fuzzing** – Automated tools that continuously send random or crafted data to an application and watches for exceptions and anomalous behavior. Fuzzers are good at identifying failures in input validation and bounds checking.

On screen text

Security Testing

Quality Assurance

- Custom test cases
- Static analysis
- Vulnerability scanning
- Penetration testing
- Fuzzing

Component Isolation

Narration

Segregating software components and defining their operational scope is crucial to limiting the impact of software security vulnerabilities. If there is a vulnerability in one component, proper isolation can limit the possibilities for lateral movement or leveraging one component to compromise another.

This creates a zero-trust environment that does not rely only on perimeter boundaries.

From a developer's and administrator's perspective, the main advantage of isolation is to insulate software components from each other and provide a single communication path between various components. This makes it easier to develop, manage, and monitor security controls.

Some techniques for isolation are:

- Virtual machines,
- Containers,
- Sandboxes,
- Process isolation,
- Separate physical servers,
- Physical or virtual networks, and
- Firewalls or filtering gateways.

On screen text

Component Isolation

Isolation can limit lateral movement

Zero Trust

Example Techniques:

- Virtual machines
- Containers
- Sandboxes
- Process isolation
- Physical servers
- Physical or virtual networks
- Firewalls or gateways

Resource Consumption

Narration

In a computing environment, a resource is anything that has limited availability or capacity. As a general term, a resource can refer to CPU, memory, and storage, but it can also refer to anything that might have physical capacity limits or other limitations imposed by the software or operating system.

Some other resources might include:

- Network capacity
- File handles
- Network sockets, or
- I/O operations

A resource exhaustion attack causes an application to fail because it cannot allocate enough resource to handle a load. These attacks exploit system defects, insufficient server resources, inefficient coding practices, or lack of rate limiting features.

By not controlling resource consumption, you might allow an attacker to cause the system to slow down, momentarily hang, or crash, making it unusable. A failure in one system could lead to a

cascading effect, causing an increased load or the failure of other systems.

Every application consumes system resources while running. Examples of resource exhaustion are excessive CPU usage, RAM depletion, or filling up a hard drive.

Here are some tips to limit resource consumption:

- Monitor and identify excessive consumption,
- Perform input validation,
- Improve code efficiency,
- Improve system scalability,
- Throttle or block abusive users, and
- Improve error recovery.

On screen text

Resource Consumption

CPU

RAM

Storage

Network capacity

File handles

Network sockets

I/O operations

Resource Exhaustion Attack

- System defects
- Insufficient resources
- Inefficient code
- No rate limiting

CPU usage

RAM depletion

Filling hard drive

Monitor & identify excessive usage

Perform input validation

Improve code efficiency

Improve system scalability

Throttle or block abusive users

Improve error recovery

What Causes Resource Consumption?

Narration

There are a variety of methods that lead to resource consumption vulnerabilities, listed above are some of those methods. Select all that apply.

On screen text

What Causes Resource Consumption?

There are a variety of methods that lead to resource consumption vulnerabilities, listed above are some of those methods. Select all that apply.

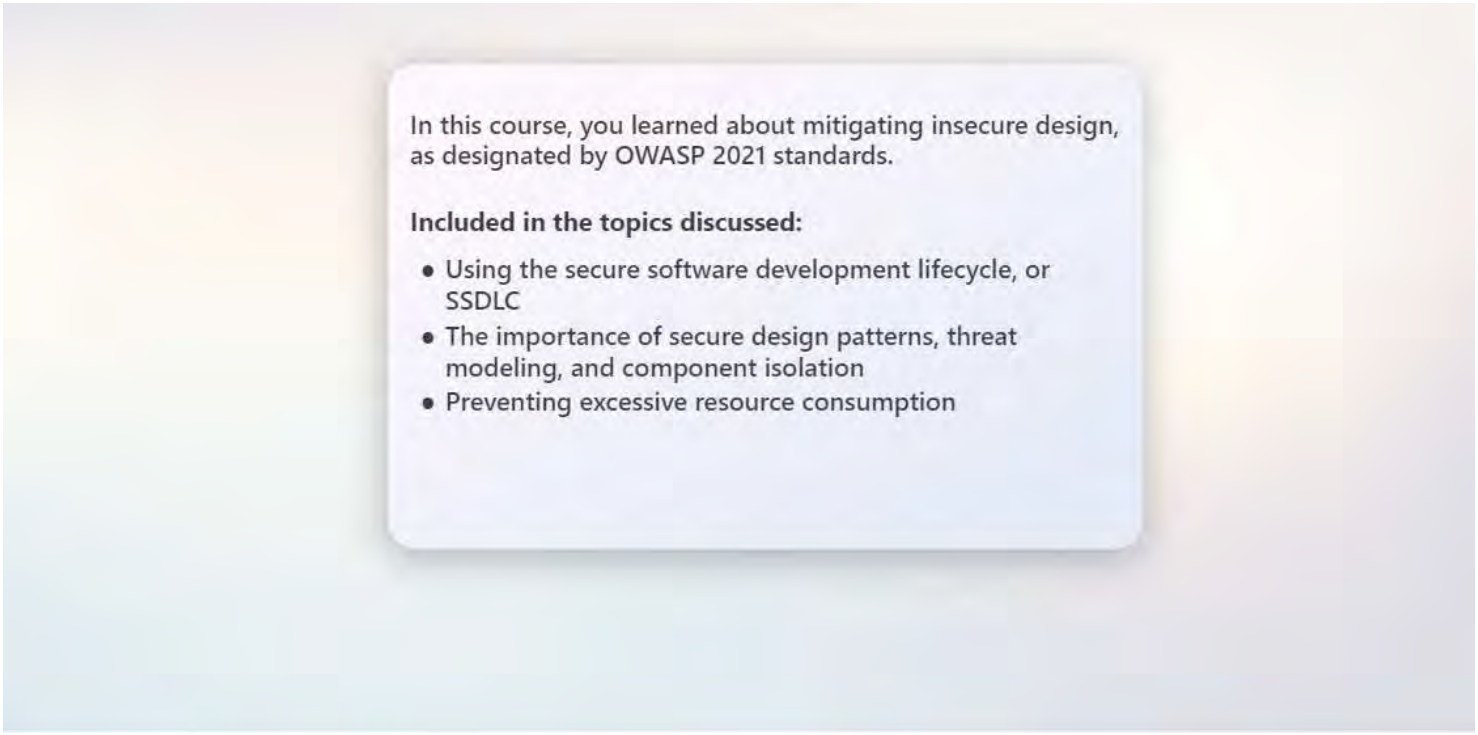
Poor input validation - **Correct**

Inefficient coding - **Correct**

Poor error recovery - **Correct**

Set CPU usage limits

Course Summary



Narration

In this course, you learned about mitigating insecure design, as designated by OWASP 2021 standards.

Included in the topics discussed:

- Using the secure software development lifecycle, or SSDLC,
- The importance of secure design patterns, threat modeling, and component isolation, and
- Preventing excessive resource consumption.

On screen text

Course Summary

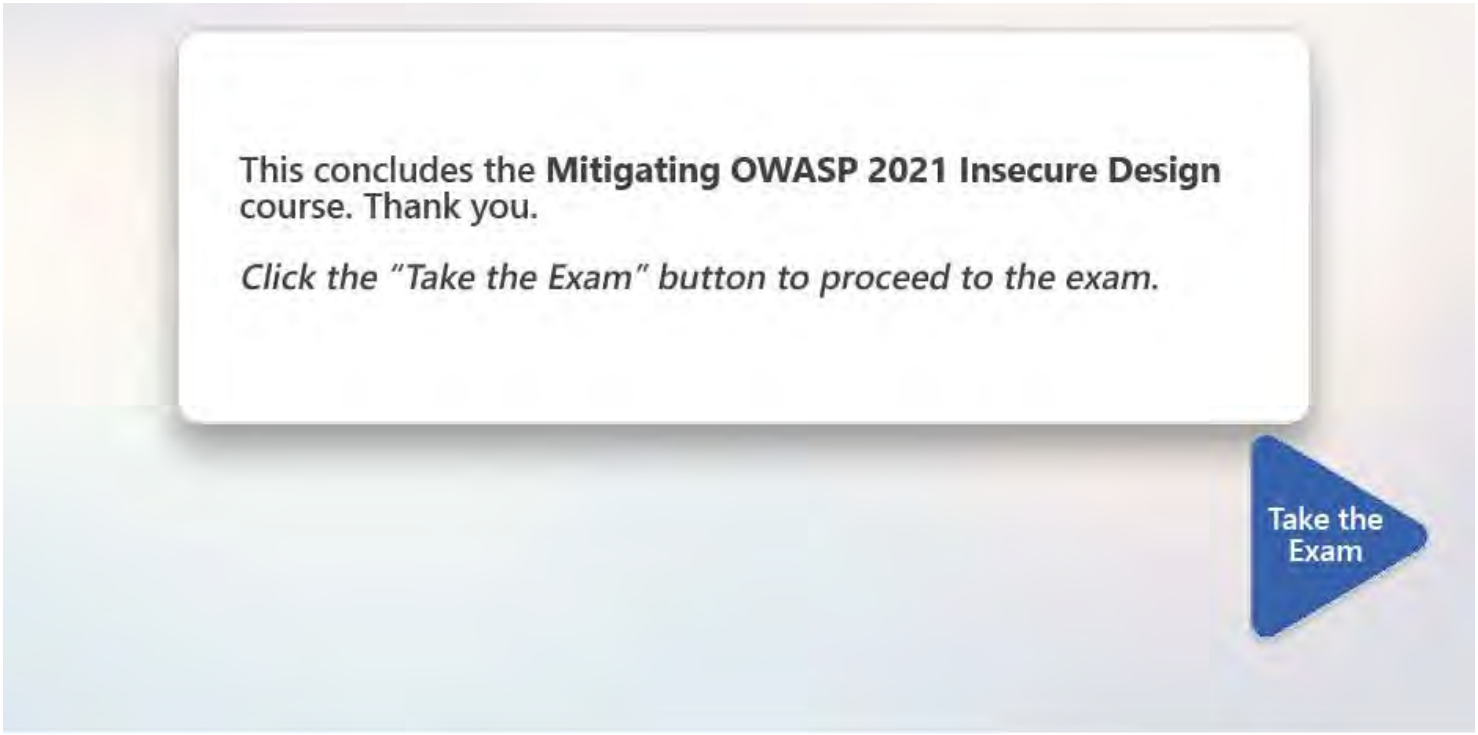
In this course, you learned about testing for Cryptographic Failures, as designated by OWASP 2021 standards.

Included in the topics discussed:

- Using the secure software development lifecycle, or SSDLC
- The importance of secure design patterns, threat modeling, and component isolation

- Preventing excessive resource consumptionHow to secure data in transit through TLS and best practices to prevent the exposure of sensitive data.

Thank You



Narration

On screen text

Thank You

This concludes the **Mitigating OWASP 2021 Insecure Design** course. Thank you.
Click the "Take the Exam" button to proceed to the exam.