Developing Secure Applications

Training Reference Guide

June 2017

About this Training Reference Guide

Purpose

This Training Reference Guide is intended to be used as a reference when taking the **Developing Secure Applications** online course.

This guide provides an outline of key points and is intended to complement the online course and to aid in note taking. In addition, links to helpful resources are provided for learners who would like more information about a particular topic.

In this Document

This document contains the following topics.

Developing Secure Applications

Training Reference Guide

June 2017

1. Secure Design

Introduction

This lesson outlines key methodologies and best practices for designing secure software. At the end of this lesson, learners will be able to:

- What an attack surface is, and how to mitigate their presence.
- The basics of threat modeling different types of attacks.
- Strategies for designing layered, in-depth security.
- The principle of least privilege.

Attack Surfaces

An attack surface is any part of a program accessible from another program.

House analogy:

- Attack surfaces would be any entry point into that home:
  - Window.
  - Garage.
  - Front door.

To mitigate attack surfaces:

- Identify all entry points
- Rank them for potential risk, and consider security factors, such as:
  - Authentication.
  - Remote vs. local access.

Tips and tricks:

- Iterative process—some features may have sub-features and other functionality that needs to be thoroughly analyzed.
- Managing attack surfaces doesn't necessarily mean eliminating them in their entirety.
- House analogy: Having a door with a lock and key might be better than having no door at all.
- Developer example: Allowing imports, but limiting the number of file formats allowed.

Attack surface mitigation examples:

Higher Attack Surface	Lower Attack Surface
On by default	Off by default
Open socket	Closed socket
UDP	TCP
Anonymous access	Authenticated user access
Constantly on	On as needed
Administrative access	User access
Internet accessible	Local subnet accessible
Running as SYSTEM	Running as user, network service or local service account
Uniform defaults	User defined settings
Large code	Small code
Weak access controls	Strong access controls

Real life example—Microsoft Windows:

Microsoft Product	Attack Surface Reduction
Windows	• Authenticated Remote Procedure Call (RPC) • Firewall on by default
Internet Information Services 6.0 and 7.0	• Off by default • Running as network service by default • Static files by default
SQL Server 2005 and 2008	• xp_cmdshell stored procedure off by default • CLR and COM off by default • Remote connections off by default
Visual Studio 2005 and 2008	• Web server localhost only • SQL Server Express localhost only

---

Privacy vs. Security

Let's define privacy and security:

- **Privacy:** Allows users control over their information
- **Security:** Establishes protective measures against hostile acts or influences, can protect privacy

Both Security and Privacy are necessary for building trusted applications. While security measures are essential in preserving privacy, a secure system does not guarantee privacy in regards to the data stored within it.

House analogy:

- A closed and locked window may provide security, but it does not necessarily provide privacy—potential attackers could see who is home and wait for the right time to try and compromise the security system.

Key factors in developing privacy-aware applications:

- **Legal obligations:** Company and government policies.
- **Customer Trust:** Customers like control over their information.
- **Blocked Deployments:** Privacy concerns can limit user's willingness to install the application.

Privacy sensitive behaviors and their associated laws:

Application Behavior	Privacy Concern
Target children	Children Online Privacy Protection Act (COPPA)
Transfer sensitive PII	Gramm-Leach-Bliley Act (GLBA), Health Insurance Portability and Accountability Act (HIPAA)
Transfer non-sensitive PII	European Union (EU) or Federal Trade Commission (FTC)
Modify system	Computer Fraud and Abuse Act (CFAA)
Continuous monitoring	Anti-Spyware Legislation, Deployment Blocker
Anonymous transfer	Deployment Blocker

Resources for developing privacy-aware applications:

- Laserfiche corporate privacy policy
- Microsoft Security Development Lifecycle site https://www.microsoft.com/en-us/sdl/default.aspx

---

Threat Modeling

Threat modeling is a process used to understand threats to an application by simulating or "modeling" their behavior.

Defining the purpose of threat modeling brings up an important distinction between threats and vulnerabilities:

- **Threats** are what a malicious user may attempt to do in order to compromise a system.
- **Vulnerabilities** are specific ways that a threat is exploitable, such as a coding error

Going back to our example of a home, the threat may be a break-in, and the vulnerability is a loose door lock.

In this scenario, threat modeling could involve asking a locksmith to try and pick the lock, and use any inferences as a way to locate and identify vulnerabilities in the home's security system.

Defense in Depth

Defense in depth is the idea that if one defense layer is breached, other layers should provide additional security to the application.

Real-life example—car in a garage:

- First layer of defense—the garage itself.
- Second layer—car's locks.
- Third layer—car's alarm system should activate.

As a developer, always assume that any single layer of defense could fail at some point. Although a singular defense mechanism may be penetrable, a layered approach reduces the likelihood of a compromised system.

For an application to be trusted, it must implement multiple layers of security and privacy protection.

Least Privilege

The principle of least privilege follows the idea that if an application is compromised, then potential damage that a malicious person can inflict should be contained and minimized.

Thus, this principle falls on the assumption that the application can and will be compromised. By limiting privileges of users, malicious users will likely inherit those limitations.

Home analogy:

- The principle of least privilege may allow most users to access the house, but only a handful of users will be granted access to the kitchen, the garage, bedrooms, and the like, depending on the chores those users have to do.

Applying the Principle of Least Privilege:

- Give users the minimal number of capabilities to perform their functions—you only want to give administrative and system level access to a select few users who need them.

- This way, if a malicious attacker tries to enter the system, the possibility that they'll have the rights necessary to cause significant damage, is minimized.

Tips and tricks:

- Evaluate your application and focus on its essential tasks.
- Ask yourself: "What is the minimum access level this application needs to perform its functions?"
- Elevate privileges only when needed, and release these privileges when they're no longer needed by users.

Secure Defaults

Secure defaults are security-conscious default settings for applications that come out-of-the-box.

Secure defaults help customers get a safer experience, and leaves it up to them to reduce security and privacy levels.

Home analogy:

- When selling a home, a realtor likely wouldn't sell it without locks on the doors or with cracks in the windows.
- The buyer may open windows or keep the door open once they move in, but by default they are given the more secure option.
- On the other hand, the user also doesn't have to install their own locks or fix up the windows to make their house secure.

In developer's terms, this means enabling or disabling features in order to make the initial user experience as safe as possible.

Here's a list of common secure defaults:

Application Component	Secure Defaults Principle
Firewall	Firewall ON by default
SSL/TLS socket	Older versions of SSL and insecure ciphers blocked by default
User can access application anonymous or authenticated	Application requires authenticated user sessions by default
Password complexity can be enforced	Password complexity is required by default
Store user passwords as hashes or clear text	Store user passwords as hashes by default

---

2. Secure Implementation

Introduction

This lesson identifies common security issues and which tools and strategies developers can use to address them. At the end of this lesson, users will be able to:

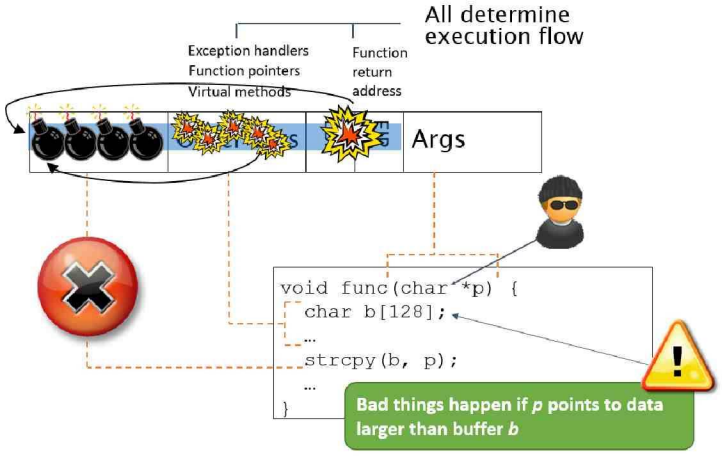
- Identify and remedy various input validation issues
- Implement strategies to address non-input validation issues

Buffer Overflow

Buffer overflow occurs when an input is larger than the buffer, or the intended length of the input. When the input's length exceeds that of a buffer, a malicious user can add their own instructions. These kinds of attacks work on both process stacks and heaps.

Example—function that copies oversized string into string with smaller buffer size.

- Without proper constraints, this oversized string can take up space in other parts of system memory.
- In the case of malicious attackers, they could insert their own code and instructions into this extra space, which could compromise the system.



Remedies:

- Use more secure libraries and classes (Strsafe, Safe CRT, or STL.)
- Search for risky functions and determine data origins—if a function's caller isn't at risk, the function itself probably isn't either.
- Enable compiler switches—use a compiler switch such as /GS to check buffer security at compile time.
- Use static code analyzers—tools such as PREFast and SAL allow for detection of vulnerable code before it compiles.
- Reduce attack surfaces—attack surfaces are entry points a user may use to access a program.
- Perform fuzz testing to test system's vulnerability to input validation issues.

Integer Arithmetic Errors

These type of errors involve common integer arithmetic mistakes that can result in buffer overflow conditions.

Example—a function that checks an input's size to make sure it doesn't overflow the buffer.

```
void Example(char* str, int size)
{
  char buf[80];

  // Check to make sure size is valid
  if (size < sizeof(buf))
  {
     // Should be safe to copy str
     strcpy(buf,str);
  }
}
```

If we use simple arithmetic to compare sizes, a size of -1 or smaller may pass the check even though it's technically out of bounds. Thus, a small arithmetic oversight may leave room for system vulnerability.

Remedies:

- Identify any calculations used to determine array offsets or memory locations—is there room for misinterpretation of an input?
- Use unsigned variables for array indexes and buffer sizes. Signed variables may cause out of bounds scenarios that lead to system vulnerability.
- Make sure to watch out for compile errors, or compiler warnings associated with any compiler switches you enabled.

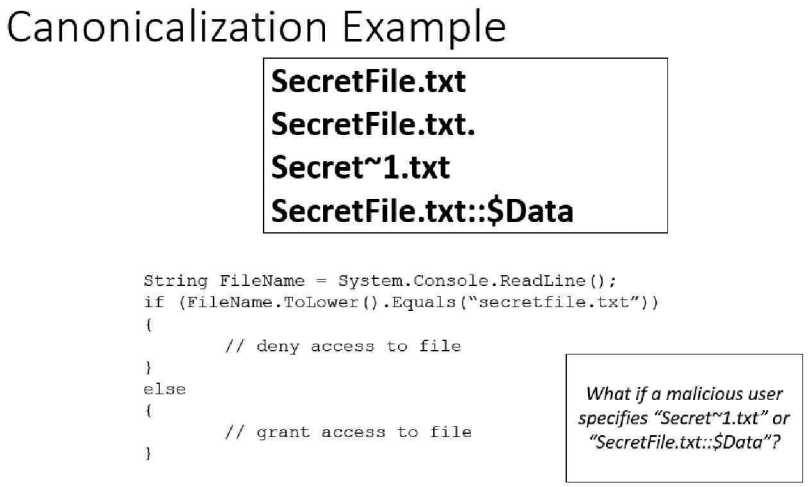
Canonicalization Errors

Canonicalization issues occur when security checks look for one form of data, but not the one actually used by the application.

Data can be represented in different forms, but have the same meaning.

Like a "fake ID", malicious users can present information that could be correct, but isn't, in order to bypass weak security checks.

Example—Program that reads from a file, "SecretFile.txt."



In this case, a malicious attacker may present a different file using one of these alternate filenames in order to compromise the system.

Remedies:

- Avoid making security decisions based on names—use operating system functionality such as ACLs, or user, file, and process IDs to confirm file and data validity.
- If names must be used, make sure you're only accepting the final or intended version of the filename.
    - This can be accomplished through the operating system using canonicalization APIs, or by simply restricting allowed file names manually in your code.

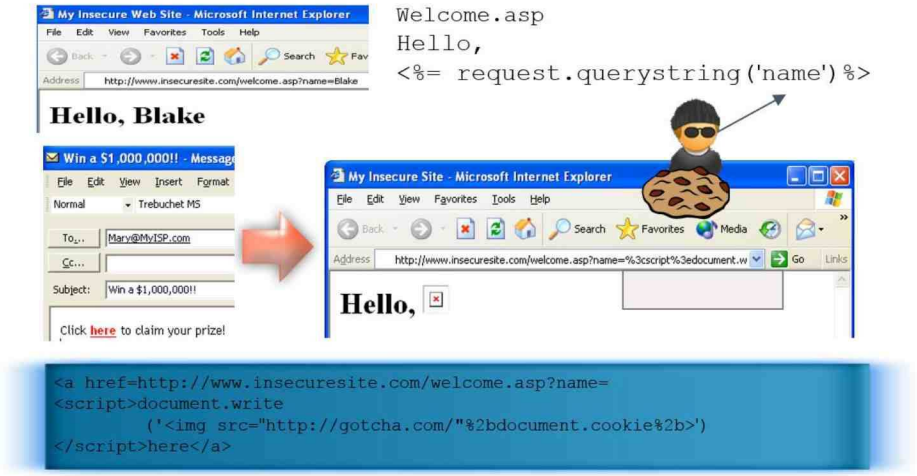
Cross-Site Scripting (XSS)

Cross-site scripting (XSS) occurs when a user input is not validated and echoed back as part of web responses. A very common type of attack, XSS exploits flaws in web applications that leads to the client being compromised.

Example—User dashboard page. In this case, the parameter for "user" is in the URL.

- An attacker could include Javascript code as the username. This way, when the web page displays the username, it will run the code, allowing the attacker to steal cookies, usernames, passwords, or other confidential information.

- How the attacker gets a user to visit is simpler—they usually use a phishing scam, whether via e-mail or social media.



Remedies:

- Validate all inputs from web pages or API requests—is the format suspicious? Is this code?
- Never directly echo user input in the browser—encode html output to mitigate the chance of echoing executable code, using tools such as the HttpUtility Class from Microsoft's .NET Framework.
- Use the HTTPOnly tag, which makes sure that cookies are only accessible by the server. This makes sure any cookie-stealing scripts end up empty-handed, provided the client's browser supports and properly implements the tag. (Most modern web browsers do.)
- Use the built-in request validation in ASP.NET to examine HTTP requests and identify any potentially dangerous inputs.
- Utilize the Content Security Policy features provided by modern browsers—they whitelist certain URLs, to ensure scripts are only loaded from trusted sources.

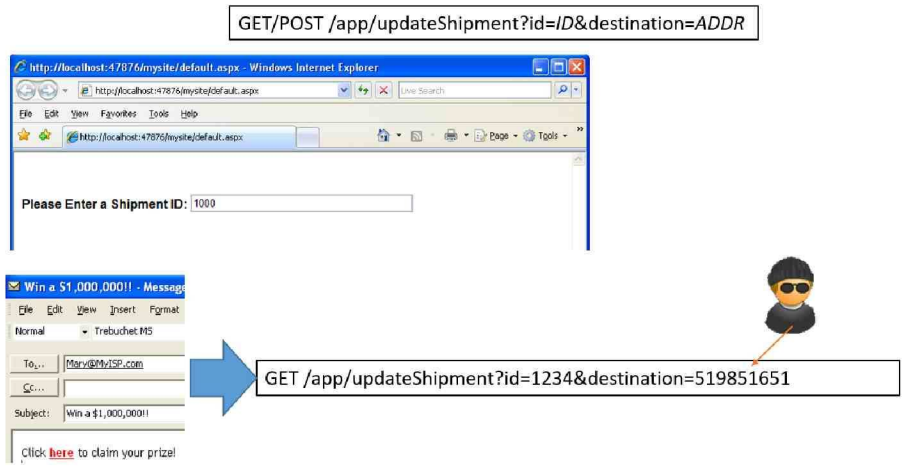
Cross-Site Request Forgery (CSRF)

Cross-site request forgeries (CSRF) occur when a legitimate client is tricked into sending a request crafted by an attack to modify server-side state

A common web attack, cross-site request forgeries exploit flaws in web applications in order to modify its server-side state.

For example, let's use a GET request that ships items to a particular address.

A malicious attacker may use a phishing link to get a user to send the request unknowingly and ship items to the attacker's address.



However, note that these types of attacks only work if the victim of the attack is authenticated. Unauthenticated users may fail this check, but these attacks can be broad enough in their approach to at least catch a few unsuspecting users.

Remedies:

- Use a hidden HTML form field with an unpredictable token—validate this token with each state-modifying request.
- This token must be unpredictable; make sure to use appropriate number generators and other tools to keep these tokens cryptographically random.
- Make sure to also generate new tokens each session, or for even more security, after the execution of each request.
- Immunity to replay attacks (when an attacker "listens in" on a communication channel in order to impersonate a user) implies immunity to CSRF attacks, but this does not work the other way around. (As attackers have current session information in a replay scenario.)

Broken Authentication and Session Management

Rather than a single flaw, **broken authentication and session management** encompasses a collection of flaws that expose user credentials or session information. These include:

- Exposing session IDs in URLs
- Predictable session IDs
- Cross-site scripting (XSS) flaws
- Other flaws allowing attackers to submit requests that appear authenticated

For more information on broken authentication and session management, as well as remedies for these flaws, consult the OWASP wiki.

Unvalidated Redirects and Forwards

Unvalidated redirects and forwards occur when an attacker exploits redirects and forwards frequently used by applications.

Example—Login page that redirects to a user's desired content page of a website.

- If the redirect address is included in the URL, an attacker could have the login page redirect to their page instead.
- Although the attacker's URL might be suspect, users might be falsely assured of security as they started with a legitimate site, and unknowingly give their credentials and other sensitive information to them.

Remedies:

- Refrain from using redirect addresses in page URLs
- If redirects are necessary, thoroughly validate the given address.
    - Quick way to do this—check if the redirect URL is part of your site or on a whitelist of external URLs.

Injection Attacks

Injection attacks occur when malicious data is inserted into strings that are later passed to a software system for parsing or execution

Usually due to a lack of input validation, these types of attacks issue commands, typically through SQL, to an internal system that users normally wouldn't access.

Although injection attacks are primarily associated with SQL, LDAP, XPath, and OS command lines are also potential targets of this technique.

Example— a webpage where a user enters a shipment ID. This site queries the SQL database for the desired shipment, then displays the information back to the user.


```
SELECT * FROM ShipmentOrders
WHERE ID = '<ID_ENTERED_BY_USER>'
```

If we give this page an expected input such as the number 1000 for a shipment ID, the database searches for a shipment with the ID of 1000.

However, if we appended an SQL command to the initial input, such as "drop table" this input will follow these extra "injected" instructions, dropping the table.

In an even worse case scenario, a malicious user may be able to inject system-level commands using a procedure such as XP CMD shell.

```
SELECT * FROM ShipmentOrders
WHERE ID = '<ID_ENTERED_BY_USER>'
```

INPUT #1: ID = 1000
Database Executes:
SELECT * FROM ShipmentOrders WHERE ID = '1000'

INPUT #2: ID = 1000'; DROP TABLE ShipmentOrders;--
Database Executes:
SELECT * FROM ShipmentOrders WHERE ID = '1000';
DROP TABLE ShipmentOrders;--'

INPUT #3: ID = 1000'; exec xp_cmdshell <*any_command*>;--
Database Executes:
SELECT * FROM ShipmentOrders WHERE ID = '1000';
exec xp_cmdshell <*any_command*>;--'

Remedies:

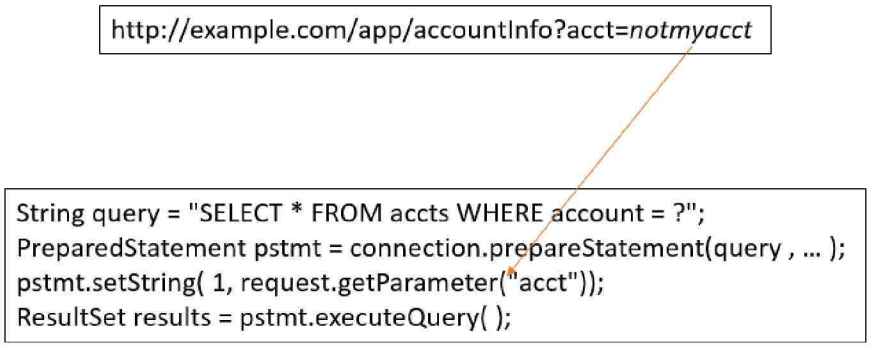
- Validate all input—input validation flaws are how injection attacks happen.
- Preprocess all input and escape control/meta-characters—make sure inputs don't have any telltale signs of being code.
- When using SQL, use parameterized queries—they preprocess requests, rendering any injection attack attempts as irregular but harmless input.
- As another SQL consideration, use stored procedures and disallow access to underlying tables—these measures limit the damage potential of any one injection attack.

Insecure Direct Object References

Insecure direct object references occur when applications process requests containing object IDs without performing object-level authorization checks

This lack of authorization verification allows malicious users to execute commands against resources they are not authorized for.

Example—site that upon logging in, uses a user's account number in the URL. Without proper validation, a malicious user may be able to input a different account number and access the site as someone else.



Remedies:

- Perform an authorization check against any referenced objects before each request—is this user authorized to do this or access this content?
- You can also convert direct references to indirect references:
    - Create a map of authorized objects and actions with a key to each one, when the user is initially authorized.
    - Have users select only from these pre-defined lists of authorized options.

Missing Function Level Access Control

Missing function level access control occurs when a function is executed despite the function's caller being unverified.

Attackers can potentially exploit these insecure functions via:

- Web requests.
- Remote procedure calls.
- Other methods, even if these options aren't available via an application's user interface.

To prevent these types of issues, add a check at the start of the function to make sure the caller is authorized to run it.

Sensitive Data Exposure

Sensitive data exposure occurs when sensitive information is unnecessarily exposed—examples include displaying a full credit card number on a user's bank dashboard, or storing passwords in plain text.

Remedies:

- Avoid storing passwords using your own infrastructure. If necessary, use salted hashes—this method increases a password's security by adding a user-unique random string, or salt, to their password before hashing and storing it.
- Encrypt sensitive data, both at rest and over the network using standard protections and protocols.
- When no longer needed, be sure to discard sensitive data immediately.
- Disable browser autocomplete forms collecting any sensitive data such as passwords or credit card numbers.

Cryptographic Flaws

Cryptographic flaws occur when using outdated or insecure algorithms for cryptographic primitives. For example, security experts have been able to "break" algorithms for MD5 and DES, rendering them obsolete.

As these changes in technology can happen fast, be sure to keep up with data cryptography trends. If you're unsure on what algorithms can be used, be sure to check company listings of approved algorithms.

In addition, be sure to use company approved implementations; in many cases, a cryptographic tool's implementation can be just as important as the algorithm itself.

Cryptographic primitives can also be combined together to enhance data security. However, be sure to use established, company approved patterns, or gain approval for any new patterns you would like to use.

Insecure OS and Application Configurations

Insecure OS and application configurations occur when an application has improperly configured security settings relative to operating systems it interacts with.

Remedies:

- Check the following:
  - ASP.NET debug pages—are they enabled? Keeping debugging tools available leaves pages vulnerable to attackers.
  - Your application's services—do they run at the system level? If so, is it necessary?
  - Vendor-supplied passwords—are they used for installed software?
  - Secure design principles play a large role—ensure secure defaults are used to establish a secure out-of-box state for your application.
- Testers should add configuration checks to their test plans—although it seems daunting to test for every type of system, it is important to test for common configurations and operating systems to avoid vulnerabilities.

Using Components with Known Vulnerabilities

While checking the security of an application, it is also important to make sure that any third-party components or libraries used within it do not contain security vulnerabilities.

Remedies:

- Record all third-party software being used and its version.
- Before shipping any code, research any third-party components to ensure there are no published vulnerabilities.
- Minimize functionality exposed in third-party components. If a particular functionality isn't used by the application, and it can be turned off, stripped off, or disabled, do so.

---

3. Threat Modeling

Introduction

This lesson guides learners through the steps needed to model threats to the system. At the end of this lesson, learners will be able to:

- Define the process of threat modeling
- Understand the key steps in threat modeling

Threat Modeling Principles

Threat modeling is the process used to understand security threats to a system, determine risks from those threats, and establish appropriate mitigations.

- Threat modeling is a key requirement of the Microsoft Security Development Lifecycle (SDL) and similar methodologies.
- Threat modeling follows a series of four sequential steps: modeling, enumeration, mitigation, and validation.

When to threat model:

- During the design phase.
- Agile development is an iterative process of design and implementation, so these modeling processes should be repeated in a similar way.

Events in the process where you'd want to threat model include:

- Architectural changes.
- New features that are adding data flows between components.

Who performs threat modeling:

- Threat modeling can be performed regardless of security expertise.
- Process should be driven by:
    - Application designers.
    - Testers.

Although not all developers and testers will be driving the process, they should still in some way be involved in it.

What should be threat modeled:

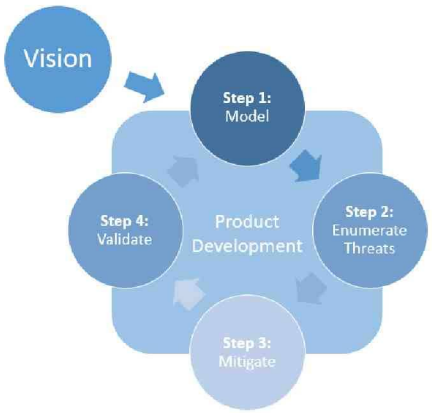
- Threat modeling processes should be performed on:
    - The application as a whole.
    - Any security and privacy features.
    - Features whose failures have security or privacy implications for your application.
    - And any features that cross trust boundaries.

Advantages and disadvantages of threat modeling:

- Advantages:
  - Threats can be found early in the development process.
  - Threat models can be performed with or without security expertise.
  - Can be and be used to guide security assessment activities.
- Disadvantages:
  - Threat modeling requires a good level of training, software, and setup to be properly implemented

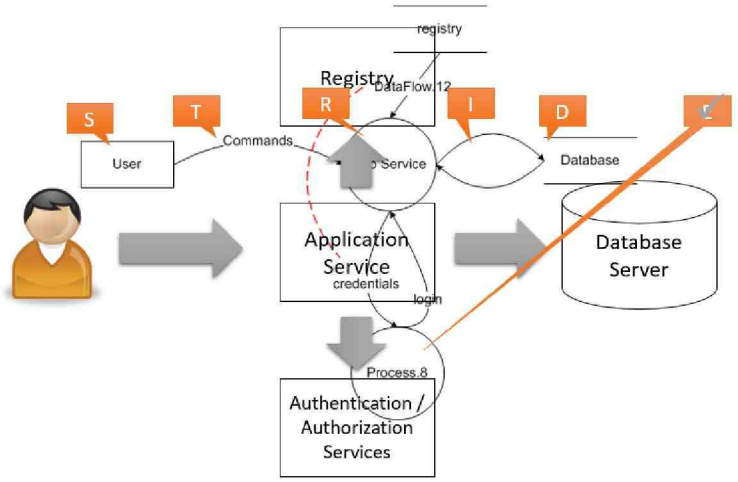
The threat modeling process generally follows these steps:

- **Vision:** Before going into any of the core steps, first make sure you have a vision of what kind of threat you'd like to model and the general approach you'll take as you go through the core steps.
- **Modeling:** The first core step in the process is modeling. This involves diagramming and laying out the groundwork for the other steps.
- **Enumeration**: This step involves identifying threats to each component in the model.
- **Mitigation:** This step addresses any enumerated threats.
- **Verification:** Lastly in this step, the model is thoroughly verified for accuracy and validity.



Step 1: Modeling

The modeling step's objective is to drive threat analysis through modeling an application's design. This is done using a **Data Flow Diagram, or DFD.**



What is a Data Flow Diagram, or DFD? Each DFD highlights the following:

Element	Represented By	Description
External Entity		Any entity not within the control of the application, such as people and external systems
Process		Code, such as native code executables and .NET assemblies
Data Store		Data at rest, such as registry keys and databases
Data Flow		How data flows between elements, such as function calls and network connections
Trust Boundary		A point within an application where data flows from one privilege level to another, such as network sockets, external entities and processes with different trust levels

Step 2: Enumeration

The enumeration step's objective is to identify threats for each element in the data flow diagram, or DFD.

Although security experts can get away with brainstorming possible threats to DFD elements, we recommend all those performing threat modeling processes to identify threats using the **STRIDE threat types.**

STRIDE threat types include:

Desired Property	Threat	Definition
Authentication	Spoofing	Impersonating something or someone else
Integrity	Tampering	Modifying code or data without authorization
Non-repudiation	Repudiation	The ability to claim to have not performed some action against an application
Confidentiality	Information Disclosure	The exposure of information to unauthorized users
Availability	Denial of Service	The ability to deny or degrade a service to legitimate users
Authorization	Elevation of Privilege	The ability of a user to elevate their privileges with an application without authorization

For information on STRIDE and threat modeling, visit Microsoft's Developer Network site.

Different elements in the data flow diagram are more vulnerable to certain types of attacks than others. The following table lists how each type of DFD element maps to STRIDE threat types:

Element	S	T	R	I	D	E
External Entity	✓		✓			
Process	✓	✓	✓	✓	✓	✓
Data Store		✓	✓	✓	✓	
Data Flow		✓		✓	✓	

Note that data stores face a special kind of threat from repudiation attacks. A common approach to mitigating repudiation attacks is to keep an activity log. This means that a primary goal of repudiation attacks will be deleting or altering these logs; wherever they are stored will be a target.

Step 3: Mitigation

The mitigation step's objective is to **address any identified threats in an application's design.**

Approaches to threat mitigation include:

- Redesigning the application.
- Asking "what have similar software packages done and how has that worked out for them? And applying those standard mitigations.
- Using unique mitigation solutions.
- Accepting a certain level of risk, as long as it's in accordance with company policies.

What mitigations you use depends on the type of threat addressed. The following table maps STRIDE threat types to their appropriate mitigations:

Threat	Example Standard Mitigations
Spoofing	IPsec Digital signatures Message authentication codes Hashes
Tampering	ACLs Digital signatures Message authentication codes
Repudiation	Strong authentication Secure logging and auditing
Information Disclosure	Encryption ACLs
Denial of Service	ACLs Quotas High availability designs
Elevation of Privilege	ACLs Group or role membership Input validation

---

Step 4: Validation

The validation step's objective is to **help ensure that threat models accurately reflect the application's design and address potential threats.**

In the validation step, be sure to ask the following questions in order to validate your threat model:

- Does the data flow diagram match the application?
- Are threats enumerated and accounted for?
- Were the identified threats mitigated or addressed?
- Have any assumptions made at the early stages in the modeling process been proven wrong? Proven true?

---

4. Secure Verification

Introduction

This lesson outlines various methods for developers to test and verify a system's security configuration. At the end of this lesson, learners will be able to:

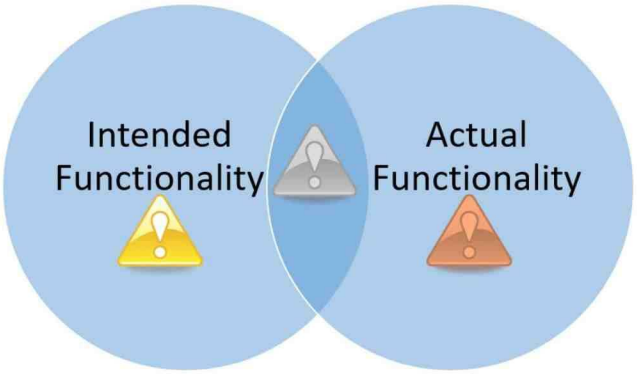
- Define and implement various security testing methods.
- Track and manage results of security tests.

Functional Testing vs. Security Testing

Functional testing is the process of verifying an application can be used by legitimate users. This is done by testing for intended functionality outside of actual functionality.

Security testing is the process of verifying that an application cannot be misused by malicious users. This is done by testing for actual functionality outside of intended functionality.

Using both types of tests, the desired result is an application with intended functionality that matches its actual functionality.



Tips and tricks:

- Remember the first rule of security testing: there are no rules!
- Malicious users do not play by the rules, and your testing techniques should recognize that.
- Whenever you think "malicious users wouldn't try that," remind yourself that they could, and in many cases will try anything to compromise the system. This includes:
    - Circumventing application components.
    - Attempting to compromise application dependencies.

In summary, perform your security testing as if an attacker will use any and all means necessary to reach their ultimate goal, which is to compromise the system.

Verification Phase Testing Techniques

Fuzz testing is a testing methodology that can help identify input validation issues, which may leave a system vulnerable to potential attackers.

There are two common approaches to fuzz testing:

- "Smart Fuzzing" involves tweaking valid inputs slightly to make them invalid inputs.
- "Dumb Fuzzing" uses random inputs for testing. Despite its name, dumb fuzzing is valuable, and in general both approaches should be used in order to fully test the system.

How to fuzz test:

- Identify all entry points, then determine valid inputs for each entry point.
- Using these valid inputs, create a set of invalid inputs by modifying them.
- Lastly, feed the inputs into the program and observe.

Use automation if possible, as this both streamlines the process and allows for a wider range of test inputs, which can help discover more bugs and security flaws.

Penetration testing involves actually simulating attacks against an application

With this approach, testers take the role of a malicious user, either manually or using tools to automate attacks. When performing this type of test, please note:

- Penetration testing requires an expert. They need to be as resourceful and determined as an actual attacker to properly perform a penetration test.
- These tests only provide a snapshot of your security posture at the time of testing. Results should not be considered completely conclusive or heavily relied on.

Run-time verification is the process of examining application behaviors at run-time.

- The goal of run-time verification is to observe how an application behaves under certain conditions to detect specific security issues.

- Primarily done through automated tools, such as Microsoft's AppVerifier.

Manual review means to manually review the source code to identify common weaknesses.

- Although this should be done with all high-priority code, and is very effective, note that manual review is, by its nature, an incredibly labor-intensive task.
- Even with manual review, there are some automated tools out there to assist reviewers. Be sure to use them to make the process of manual review smoother and more efficient.

Verification Phase Automation

There are many tools available to testers wanting to automate the verification process. The automation tools used in this process fall into a few main categories:

- **Static source code analysis tools:** Analyze the code before it compiles.
- **Binary analysis tools:** Detect flaws at compile-time.
- **Run-time analysis tools:** Test the system at run-time.

Handling Security Vulnerabilities

When addressing security vulnerabilities, it is important to note that our software's main function isn't just security—its primary function is to solve a business problem. Therefore, developers must perform a delicate balancing act between application functionality and security.

To help decide what code can ship, create a **security bug bar** to define the maximum severity level of flaws and threats you're willing to accept. More information on security bug bars can be found on Microsoft's Security Development Lifecycle site.

When creating a security bug bar, be sure to include and enumerate threats, in order of severity, such as:

- Elevation of privileges.
- Denial of service.
- Targeted information disclosure.
- Spoofing.
- Tampering.
- Other threats and vulnerabilities that need to be addressed.

Once potential threats and vulnerabilities are identified, they should be scored by severity level to determine what is acceptable to ship. Although there are many ways to score components, whether as low-medium-high, or a numeric scale such as one through five, we recommend using the **Common Vulnerability Scoring System (CVSS)**, or CVSS, in order to identify and classify threats.

Currently developed under the Forum of Incident Response and Security Teams, or FIRST, the CVSS scale scores vulnerabilities on a scale of one to ten. Scores are based on three key security factors:

- Exploitability.
- Scope.
- Impact.

For more information about CVSS, visit the FIRST website.

Make sure to identify all vulnerabilities and possible threats to the system. When identifying vulnerabilities and threats, be sure to keep in mind:

- All components, even third-party components, should be thoroughly reviewed and analyzed.
- Use a comprehensive and sorted list of threats and vulnerabilities to determine a maximum acceptable level of severity.

Step 3: Mitigation

The mitigation step's objective is to **address any identified threats in an application's design.**

Approaches to threat mitigation include:

- Redesigning the application.
- Asking "what have similar software packages done and how has that worked out for them? And applying those standard mitigations.
- Using unique mitigation solutions.
- Accepting a certain level of risk, as long as it's in accordance with company policies.

What mitigations you use depends on the type of threat addressed. The following table maps STRIDE threat types to their appropriate mitigations:

Threat	Example Standard Mitigations
Spoofing	IPsec Digital signatures Message authentication codes Hashes
Tampering	ACLs Digital signatures Message authentication codes
Repudiation	Strong authentication Secure logging and auditing
Information Disclosure	Encryption ACLs
Denial of Service	ACLs Quotas High availability designs
Elevation of Privilege	ACLs Group or role membership Input validation

---

Step 4: Validation

The validation step's objective is to **help ensure that threat models accurately reflect the application's design and address potential threats.**

In the validation step, be sure to ask the following questions in order to validate your threat model:

- Does the data flow diagram match the application?
- Are threats enumerated and accounted for?
- Were the identified threats mitigated or addressed?
- Have any assumptions made at the early stages in the modeling process been proven wrong? Proven true?
