WEB APPLICATION SECURITY

Stephen Schaub

Attacks

- Cross-Site Scripting
- Cross-Site Request Forgery

Cross-Site Scripting (XSS)

- A type of injection attack in which an attacker uses a vulnerable web application to force a victim to execute malicious code
- Discussion:
  - https://excess-xss.com/

Cross Site Scripting Prevention

- Appropriately encode data from untrusted sources before outputting it to HTML responses
- Proper encoding depends on context of information on page
- Review: How should *value* be encoded for each of the following contexts?
  - Document Body:
    - <html><body> … {{{value}}} … </body></html>
  - Element Attribute:
    - <input type="text" name="somename" value="{{{value}}}">
  - Attribute Containing URL: <img src="{{{value}}}">

XSS Prevention, Continued

- OWASP Cross Site Scripting Prevention Cheat Sheet
  - https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html
  - Good advice
  - Some recommended escaping can be reduced if care is taken with proper quoting

XSS Prevention, Continued

- Note Rule 0, places never to put untrusted data
- Rule 1: HTML Escape untrusted data in HTML Element Content
  - Example: … (HTML Escape data put here) …
- Rule 2: Attribute Escape untrusted data in Typical HTML Attributes
  - Overkill if the attribute is properly quoted
  - When inserting data into a double-quoted attribute, simply ensure that the data has any " characters replaced with "
  - Example:

```
let escapedData = untrusted.replace("", "&quot;");
<textarea width="(... insert escapedData here...)">
```

XSS Prevention, Continued

- Rule 3: Unnecessary if you follow good practices
  - Generating JavaScript code that contains dynamically inserted data is rarely needed when using modern AJAX techniques
- Rule 4: CSS Escape and Validate untrusted data inserted into Style Property values
  - Like rule 2, escaping can be reduced if using properly quoted values
- Rule 5: URL Escape untrusted data inserted into URLs
  - Important!
- Rule 6: Sanitize HTML Markup with a Sanitizer Library
  - Good advice

XSS Prevention, Continued

- Consider implementing a Content Security Policy that forbids inline script execution on your site (see content-security-policy.com)

Cross-Site Request Forgery (CSRF)

- An attack that forces an end user to execute unwanted actions on a web application in which they're currently authenticated
  - https://owasp.org/www-community/attacks/csrf
- Discussion
  - https://reflectoring.io/complete-guide-to-csrf/

CSRF Prevention

- Use POST for state-modifying requests
- Add confirmation screens for sensitive operations
- Use SameSite session cookies
- Generate and use secure CSRF tokens for state-modifying requests
- Many frameworks provide built-in support for CSRF tokens
  - Example: Express csrf-sync module
  - See examples/webapps/carscsrf
- See
  - https://cheatsheetseries.owasp.org/cheatsheets/Cross-Site_Request_Forgery_Prevention_Cheat_Sheet.html

OWASP Top Ten

- Highlights "Top Ten" vulnerabilities in web applications
- Our focus:
  - Injection
  - Identification and Authentication Failures
  - Broken Access Control
  - Cryptographic Failures
  - Security Misconfiguration
  - Vulnerable and Outdated Components

Injection

- Server-side code interacts with API's that execute strings containing commands
  - SQL statements
  - Command Shell commands
  - eval() functions
- Often the command strings must be built dynamically using data that originated with user
  - executeQuery("select * from users where username = \"" + username + "\"")
  - exec("cp uploads/" + filename + " /tmp")
- Injection occurs when users provide input values that alter the intended effect of the command strings

Preventing Injection

- Validate and/or Sanitize inputs before using them to build command strings
- Use SQL parameterized query mechanisms

Authentication vs. Authorization

- Authentication – associating a client with a user account
  - Client present credentials
  - Server validates credentials against an entry in the account database
  - An authenticated client is known as a **principal**
- Authorization – verifying a user has permission to access requested info / perform requested action
  - Actions / data items are **securables**
  - Authorization involves verifying that a **principal** has access to a **securable**

Identification and Authentication Failures

- Includes a variety of vulnerabilities related to authentication and session management
  - Allows use of weak passwords
  - Poor forgot-password procedures
  - Poor password management practices
  - Poor session ID management
- Prevention:
  - Know and use best practices

Broken Access Control

- Access Control: Restrictions that prevent users from accessing data or functions outside what is allowed by their intended permissions
- Applications often rely on view-level protections to implement access control
  - Don't show links to admin functions to regular users
  - Display a subset of records for the current user to view / edit
- Fail to account for possibility that users may modify URL to attempt to
  - Directly access restricted areas of application
  - Access data that they should not be able to see (example: insecure direct object reference)
- Fail to consider that users may attempt to access application using special browser developer tools, or non-browser tools like curl / wget

Broken Access Control Exploits

- Privilege elevation
  - Accessing data / functionality intended only for authenticated users without being logged in
  - Accessing data / functionality intended only for administrators when logged in as a non-admin user
- Bypassing access control checks through URL modification or use of tools like curl
  - http://example.com/app/accountInfo?acct=notmyacct
- Force browsing
  - Accessing a specific page of the application via direct URL entry without navigating to it through the application

Broken Access Control Prevention

- Application must not rely on view-level security
- Application must check that the user has access to
  - Each page
  - Each record

Cryptographic Failures

- Sensitive data needs to be protected
  - In transit (using https)
  - At rest (encrypting data in database)
- Prevention
  - Identify sensitive data
    - Authentication information (passwords)
    - Credit card information
    - Health data
    - Personally identifiable information (PII)
  - Ensure the data is protected in transit and at rest

Security Misconfiguration

- Includes:
  - Default accounts and passwords enabled and unchanged
  - Error handling that reveals stack traces or overly informative error messages
  - Security settings in application servers, frameworks, databases not set to secure values
  - Application / web servers with unnecessary demo applications / features turned on in production
    - Example: Web server has directory listing feature unnecessarily enabled
  - Out of date software

Vulnerable and Outdated Components

- Includes:
  - Third-party libraries with vulnerabilities
  - Application server with vulnerabilities
  - Unpatched OS
- Prevention:
  - Regularly review and update third-party components
    - Example: use **npm audit** to identify vulnerable components in your app
  - Keep OS and application server up to date with security patches

References

- OWASP Top 10
  - https://owasp.org/www-project-top-ten/
- OWASP Cheat Sheets
  - https://cheatsheetseries.owasp.org/
- OWASP Code Review Guide
  - https://owasp.org/www-project-code-review-guide/