

SEMINAR 5

INPUT VALIDATION AND SECURE CODING TECHNIQUES



Strategies to reject invalid data



JOSÉ MANUEL REDONDO LÓPEZ "S-81 ISAAC PERAL" PROJECT V4.0



Departamento de Informática
Universidad de Oviedo



Universidad de Oviedo

Logo: @creative_vanesa (Instagram)

WHAT IS THIS SEMINAR?

- In this seminar we will give practical examples of how to do certain security tasks linked to the coding phase of the SSDLC
 - As complement to the theory **Topic 6**
 - Many in the form of a checklist that allow you to easily check if they have been implemented
- These examples are shown in the technologies that are used in other courses of the degree on purpose, to favor integration among courses
 - Node and Spring Boot (**SDI**)
 - PHP (**SEW**)
- We will also mention concepts of
 - **TPP**: Concurrency handling
 - **DLP/AMD**: Handling and validating regular expressions
- The full (and editable) checklist can be downloaded here
 - https://github.com/jose-r-lopez/SSI_Extra_Materials/wiki/5.-%E2%9C%85-Secure-Coding-Checklist
 - It aims to be used in your TFG or future software projects



INDEX

- ▶ Security requirements
- ▶ Input validation and output encoding
- ▶ Access control
- ▶ Cryptography and data security
- ▶ General secure code creation practices
- ▶ System configuration, development tools and dependencies

[< Go to Index](#)

SECURITY REQUIREMENTS

In the Requirements Engineering (IR) course format



Departamento de Informática
Universidad de Oviedo



Universidad de Oviedo

SAMPLE SECURITY REQUIREMENTS



- This is a list of potential security requirements

- They can be used as a base to develop your own
- Adapt them to your application purpose, users, and restrictions

- Once you master it, you **must use a more complete source**

- The OWASP* Application Security Verification Standard (ASVS)
 - Free, international, and validated
 - <https://github.com/OWASP/ASVS/tree/v4.0.3#latest-stable-version---403>
- Also usable in the next phases: **design, implementation, and testing**

- RS1 Validation

- RS1.1 **Do not trust anyone**: Validate and sanitize (if required) all data, including those in the DBMS
- RS1.2 **Do all validation on the server**
 - Favor an allow-list strategy
- RS1.3 **Encode** all outputs

- RS2 Encryption

- RS2.1 **Encrypt all saved data**
- RS2.2 **Encrypt all data "in transit"**
 - Sent / received by the user, databases, APIs, etc.
- RS2.3 **Hash and add salt bits** (at least 28 characters) to all stored passwords (or avoid storing them)
- RS2.4 **HTTPS** is mandatory on all the application
- RS2.5 Use the **latest TLS version** for encryption

- RS3 Dependency Management

- RS3.1 **Scan all third-party code** (SCA)
 - To find known vulnerabilities before using it
 - And periodically once the code is incorporated
- RS3.2 **Minimize the attack surface**
 - Including as few dependencies as possible

SAMPLE SECURITY REQUIREMENTS



● RS4 Data Management

- RS4.1 **Classify and tag** all application data (stored, collected, created...)
 - According to its **criticality**
- RS4.2 Save all app secrets to a **secret store**
- RS4.3 Strictly prohibit **secret "hardcoding"**
- RS4.4 **NO sensitive data** in comments
 - Connection strings, API keys, passwords...
- RS4.5 **Error messages never** contain technical information
- RS4.6 **Only parametrized queries are allowed** for data access
- RS4.7 Do not pass important data in URL parameters

● RS5 Technological

- RS5.1 Use all applicable HTTP security headers and **CSP (Seminar 6)**
- RS5.2 Use **secure cookie settings**
- RS5.3 Use **existing security features**
 - E. g.: Those of the frameworks you use
- RS5.4 Use stable, supported, and updated framework/third-party modules **versions**
- RS5.5 Keep **dependencies up to date**
- RS5.6 Use **security tools (SAST, DAST...)**
 - Before the application goes to production
- RS5.7 **Code review** the application
 - Also, before it goes production
- RS5.8 **Capture and handle all errors**
 - Without falling into unexpected states
- RS5.9 **Disable caching** on pages containing private information (e. g. passwords)

SAMPLE SECURITY REQUIREMENTS



● RS6 User Accounts

- RS6.1 All application accounts must be **service accounts**
 - Never user accounts, a unique account per application
- RS6.2 Use password managers and **prohibit key reuse**
- RS6.3 Force **using of long passwords** (pass phrases)
 - With a minimum of complexity
- RS6.4 Verify passwords have not already been **filtered**
 - Using an external service that collects leaks
- RS6.5 **Enable MFA** on all important accounts
- RS6.6 **Do not force periodic password change**
 - Only after a breach
- RS6.7 Assign **roles** to users and permissions to roles
- RS6.8 Use a **unique authentication and identity** management method for the entire application
 - **Avoid storing passwords** (use a third-party auth service)
- RS6.9 Force the **least privilege** for all accounts
- RS6.10 Enable copy/paste on **password UI elements**
 - Allows using password managers
 - Also disable autocomplete on password fields

● RS7 Particular features

- RS7.1 Use application **threat modeling**
- RS7.2 Avoid **user file uploads** (too problematic)
 - If not possible, follow OWASP recommendations
 - https://cheatsheetseries.owasp.org/cheatsheets/File_Upload_Cheat_Sheet.html

● RS8 Log

- RS8.1 **Record all access attempts**
- RS8.2 **Do not include sensitive information** in log files
- RS8.3 Use a log format **compatible** with your log analysis tools (SIEMs...)
- RS8.4 Send your application logs to a **central storage**
 - Favors analysis and correlation of application logs
- RS8.5 Log **improper access attempts**
 - Detect malicious attempts (e. g.: malicious user input) with application code (**proactive defense**)
 - **Generate security alerts** if the application is under attack

[< Go to Index](#)

INPUT VALIDATION AND OUTPUT ENCODING

Practical ways to deal with input or output information



Departamento de Informática
Universidad de Oviedo



Universidad de Oviedo

INPUT VALIDATION



● Inputs from external users / files / APIs should be treated with extreme caution

- It is one of the most important attack vectors

● Input must be validated

- With centralized tested APIs and according to their context

● Input validation must

- Apply to at least all input data
- Define an allowed set of characters to be accepted
- Define a minimum and maximum length
 - Decimal and non-decimal digits, maximum string size...
- Define an expected type or range

1. Input Validation Checklist

Ensure compliance with mandatory input validation principles

- Conduct all data validation on a trusted system (e.g., The server)
- Identify all data sources and classify them into trusted and untrusted. Validate all data from untrusted sources(e.g., Databases, file streams, etc.)
- There should be a centralized unique input validation routine for the application

Perform proper data encoding

- Determine if the system supports UTF-8 extended character sets and if so, validate after UTF-8 decoding is completed
- Encode data to a common character set before validating (Canonicalize)
- Specify proper character sets, such as UTF-8, for all sources of input
- Verify that HTTP header values in both requests and responses contain only ASCII characters

Implement secure management of data contents

- If any potentially hazardous characters must be allowed as input, be sure that you implement additional controls like output encoding, secure task specific APIs, and implement accounting of the utilization of that data throughout the application. Examples of common hazardous characters include: < > " ' % () & + \\' \'
- If your standard validation routine cannot address the following inputs, then they should be checked discretely
 - Check for null bytes (%00)
 - Check for new line characters (%0d, %0a, \r, \n)
 - Check for "dot-dot-slash" (../ or ..\) path alterations characters. In cases where UTF-8 extended character set encoding is supported, address alternate representations like: %c0%ae%c0%ae/ (Utilize canonicalization to address double encoding or other forms of obfuscation attacks)

- Validate all input against a "white" list of allowed characters, whenever possible

Validate data types and limits

- Validate data length
- Validate data range
- Validate for expected data types

Perform data validation in a secure way

- All validation failures should result in input rejection
- Validate all client provided data before processing, including all parameters, URLs and HTTP header content (e.g. Cookie names and values). Be sure to include automated post backs from JavaScript or other embedded code
- Validate data from redirects (an attacker may submit malicious content directly to the target of the redirect, thus circumventing application logic and any validation performed before the redirect)

DATA VALIDATION STRATEGIES



- **Data that comes from an external entity should **never** be trusted**
 - *"All input is evil"* – Michael Howard (*"Writing Secure Code"*)
- **Data validation must be syntactic and semantic**
 - **Syntactic** forces the data to have correct syntax and structure: Dates, currency, DNIs, telephones...
 - **Semantic** forces values to be correct in a context, or according to given business rules
 - The start date must be earlier than the end date, the price of an item is within a supported range, the type of residential road is within the admitted types (street / square /...) ...
- **If these things are checked as soon as possible, unauthorized entries can be detected quickly and act upon it**
 - Must be done in a planned and thorough manner, as part of the system requirements
 - If it is a clear malicious case, responses against the user must be implemented in the software (account suspension, block ...) and create security alerts, as saw in theory (**not just discard the data**)
- **Always favor **allowlists** (list of allowed items, the rest is blocked)**
 - Over **blocklists** (list of blocked items, the rest are allowed)

ALLOWLIST VS BLOCKLIST



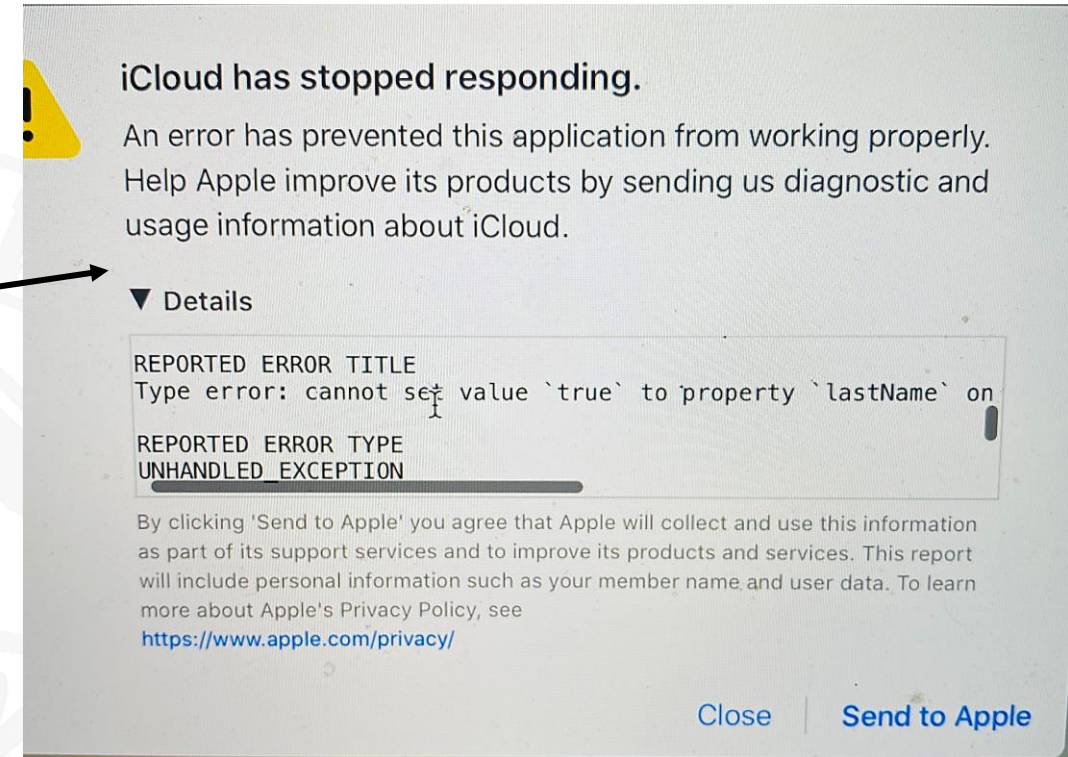
- **Use blocklists when the set of items to block is finite, fully known, and not large**
 - **Blocking known specific clients/networks** with a restricted/known number of clients (LAN)
 - **Regional blockades** (by country/area): countries known to be the source of many attacks
 - **Client connections from the ToR network**: The number of output nodes in the ToR network is known
 - They can be blocked individually (<https://www.dan.me.uk/torlist/?exit>)
 - **Blocking domains known as malicious**: <https://github.com/StevenBlack/hosts>
 - **Blocking of specific pages for different reasons**, if its number limited
- **Validating with an allowlist is usually more suitable for user input**
 - It requires **defining exactly what is authorized**
 - Anything that is not in that definition **is NOT AUTHORIZED**
 - In case of **well-structured data** (dates, social security numbers, postal codes, emails...) it is feasible to define a very strong validation pattern, **usually with regular expressions**
 - These are generally predefined and standardized for common data types
 - If the input comes from a **fixed set of options**, such as a drop-down list, it **must match exactly one of the values offered to the user**

IMPLEMENTING INPUT VALIDATION STRATEGIES

- **Input validation can be implemented with any technique that ensures syntactic and semantic correctness; usually one of these**
 - **Native validators** of any modern development framework used to create the website
 - Django validators: <https://docs.djangoproject.com/en/2.2/ref/validators/>
 - Apache Commons validators: <https://commons.apache.org/proper/commons-validator/>
 - PHP (**SEW**): https://www.w3schools.com/php/php_form_validation.asp
 - Node.js (**SDI**): <https://www.npmjs.com/package/validator?activeTab=readme>
 - Spring Boot (**SDI**): <https://reflectoring.io/bean-validation-with-spring-boot/>
 - ...
 - **Validation with JSON and XML (XSD) schemas** for inputs that follow these formats
 - **Type conversion** (e.g., `Integer.parseInt()` in Java, `int()` in Python...) with **strict exception handling**
 - **Checking the minimum and maximum** value for numeric parameters and dates, and **Length** for strings
 - **Strict checking of a set of allowed values**: especially for string parameters that can only have a small number of possible values (days of the week, months...)
 - **Regular expressions** for any other structured data (see the input validation seminar)
 - The entire input string must be validated, **avoiding the use of wildcards**

TYPE CONVERSION IS IMPORTANT

- Type conversion may be neglected, thinking that syntactic/semantic validation is enough
 - But this might cause errors in special conditions
- **iCloud: The case of "Rachel True"**
 - Someone forgot to force `lastName` type to `string`
 - Therefore, it was inferred that it was `boolean`
 - They didn't add " to the value so...
 - The program failed to run on a lower layer
 - **Consequence:** Rachel was unable to access iCloud for more than 6 months
 - His only "sin" is to have "True" as surname...
- The guy that put "null" as his car license plate
 - It appeared even in the news!
 - He had to pay 12k in parking tickets from others!



Source: <https://blog.elhacker.net/2021/03/usuario-con-apellido-TRUE-bloquea-sistema-iCloud-Apple-boolean.html>



Source: <https://www.wired.com/story/null-license-plate-landed-one-hacker-ticket-hell/>

VALIDATION OF FREE-FORM UNICODE TEXT

- **Free-form text, especially with Unicode characters, is often difficult to validate**
 - It has a large set of characters
 - In these cases, it is also important to **encode** the text according to the context of the application
- **If your text legitimately supports potentially dangerous characters (' , < , or >)...**
 - They must be encoded correctly throughout the data lifecycle
- **To validate free-form text inputs these techniques are used**
 - **Normalization**: Make sure that there is a canonical encoding that is used throughout the application
 - And that there are no invalid characters present at any time
 - **Character category allowlists**: Unicode allows categories such as "decimal digits" or "letters"
 - They not only cover the Latin alphabet, but also other global alphabets (e.g., Arabic, Cyrillic...)
 - **Allowlists of individual characters**: E.g.: if letters and apostrophes are allowed (Irish names)
 - But not the entire category of punctuation marks for a given data
- **This is beyond the scope of the course, but you can learn more here**
 - <https://ipsec.pl/python/2017/input-validation-free-form-unicode-text-python.html>

- Regular expressions you've seen in **AMD** or **DLP** are one of the most common data validation tools
 - Creating regular expressions for data validation can be complicated
 - It is beyond the scope of this course
 - But there are many resources that provide us with **already created regular expressions!**
 - **Appropriate for validating common data types**
 - One of the best known is: https://www.owasp.org/index.php/OWASP_Validation_Regex_Repository
 - Some frameworks can also define default regular expressions for certain data types
- The PCRE (Perl-Compatible Regular Expression) regular expression format is one of the most widely used formats
 - Many languages have libraries that process these regular expressions

EXAMPLES OF REGULAR EXPRESSION WHITELISTS

● Regular expression for URLs

- `![CDATA[^(((https?|ftp|gopher|telnet|nntp)://)|(mailto:|news:))(%[0-9A-Za-z]{2}|[-()_.!~*' ;/?:@&=+$,A-Za-z0-9])+)([!.~*' ;/?:,][[:blank:]])?$]]`

● Regular expression for IPv4 addresses

- `![CDATA[^(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\. (25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\. (25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\. (25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)$]]`

● Regular expression for emails

- `![CDATA[^[a-zA-Z0-9_+&* -]+(?:\.[a-zA-Z0-9_+&* -]+)*@(?:[a-zA-Z0-9-]+\.)+[a-zA-Z]{2,7}$]]`

● Regular expression for English numbers expressed in words

- `![CDATA[^(zero|one|two|three|four|five|six|seven|eight|nine)$]]`

PROTECTION AGAINST CODE INJECTION ATTACKS

- All user-controlled data received on the server must be encoded when it is part of the HTML response to a request
 - E. g.: `<script>` would be returned as `<script>`
- If we don't do this, we may be facilitating code injection attacks.
- The type of encoding that data received from users needs depends on the context of the page where that data is inserted
 - For example, **HTML encoding** is appropriate for data placed in the HTML body of the response
 - However, user data placed within the scripts of an HTTP response page would require **JavaScript-specific encoding**
 - The same happens with user data placed within CSS, URLs...
 - **Encoding HTML entities alone is not enough!**
 - You can find a complete guide on how to do it here (**XSS prevention rules**)
 - https://github.com/OWASP/CheatSheetSeries/blob/master/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.md

PROTECTION AGAINST CODE INJECTION ATTACKS

- Although there are many XSS attack vectors, following the mentioned XSS prevention rules allows us to prevent them
- We must treat HTML pages as a "template" with "placeholders"
 - The developer can only place untrusted data within these "placeholders"
 - For each of them, we implement slightly different security rules
- *Why?*
 - Each "placeholder" has a different context
 - Actions to prevent content from being interpreted as code change based on that context
- **Untrusted data should never be used on parts of the page other than those "placeholders" defined in the OWASP URL on the previous slide**
 - The effort to make sure they're not a problem is too high, and not worth it in most cases
 - So, it is **necessary** to keep user input restricted to the "placeholders" defined by the rules
 - **Rules 2 and 3 of the previous OWASP URL cover most common cases**

- **Manually encoding data is not recommended**
- There are public, free, and tested libraries that will do a better job
- The likelihood of skipping validations due to implementation details is much lower
- Some examples are
 - ASP .NET: <https://docs.microsoft.com/es-es/aspnet/core/security/cross-site-scripting?view=aspnetcore-3.1>
 - OWASP Java Encoder Project: https://www.owasp.org/index.php/OWASP_Java_Encoder_Project
 - PHP (**SEW**): <https://www.virtuesecurity.com/preventing-cross-site-scripting-php/>
 - NodeJS (**SDI**) uses **template engines** for XSS prevention through character encoding
 - Spring Boot (**SDI**): <https://stackabuse.com/securing-spring-boot-web-applications/>
 - Spring security should also be checked and used to prevent multiple attack types: <https://spring.io/projects/spring-security>
 - ...

- Another alternative is the **OWASP Java Encoder Project**

- https://www.owasp.org/index.php/owasp_java_encoder_project
- Provides numerous coding features to help defend against XSS in a **variety of different contexts**: HTML, JavaScript, XML, and CSS, with no dependencies and high performance

- Encoding is done with the `Encode` class

- Contains methods for the multiple supported encoding contexts

- Examples of context-dependent coding

- HTML

- **Basic**: `<body><%= Encode.forHtml(UNTRUSTED DATA)%></body>`
- **Content**: `<textarea name="text"><%= Encode.forHtmlContent(UNTRUSTED DATA)%></textarea>`
- **Attributes**: `<input type="text" name="name" value="<%= Encode.forHtmlAttribute(UNTRUSTED DATA)%>" />`

- **CSS**: `<div style="width:<%= Encode.forCssString(UNTRUSTED DATA)%>">`

- JavaScript

- **Block**: `<script type="text/javascript"> var txt= "<%= Encode.forJavaScriptBlock(UNTRUSTED DATA)%>"; ... </script>`
- **Variable**: `<button onclick="alert('<%= Encode.forJavaScriptAttribute(UNTRUSTED DATA) %>');"> Submit</button>`

- URL

- Parameter values: `<a href="/search?value=<%= Encode.forUriComponent(UNTRUSTED DATA)%>&order=1#top">`
- Parameters of a REST URL: `<a href="/public/<%= Encode.forUriComponent(UNTRUSTED DATA)%>">`

- **Handling an untrusted full URL**

- When controlling a fully qualified URL, first verify that it is a legal URL
 - `String url = validateURL(untrustedInput);`
- Then encode the URL as an HTML attribute when you type it on the page
- Linkable text must be encoded in a different context

```
<a href="<%= Encode.forHtmlAttribute(Untrusted URL) %>">
<%= Encode.forHtmlContent(Name of the untrusted link) %>
</a>
```

- **HTML needs to be cleaned up when only a subset of it is valid in a given context**
 - Again, we must rely on proven third-party tools to get this job done
- **One of the most popular is OWASP Java Html Sanitizer**
 - <https://github.com/owasp/java-html-sanitizer>
 - Defines a series of policies that determine the allowed HTML elements
 - You can define custom elements
 - **BLOCKS**: Allows common block elements, including `<p>`, `<h1>` etc.
 - **FORMATTING**: Allows common formatting elements, including `<b>`, `<i>` etc.
 - **IMAGES** : Allows `<img>` HTTP, HTTPS and relative sources
 - **LINKS** : Allows HTTP, HTTPS, MAILTO, and relative links
 - **STYLES** : Allows certain secure CSS properties in `style= "..."`
 - **TABLES** : Allows common table elements

```
import org.owasp.html.Sanitizers;
import org.owasp.html.PolicyFactory;
PolicyFactory sanitizer = Sanitizers.FORMATTING.and(Sanitizers.BLOCKS);
String cleanResults = sanitizer.sanitize("<p>Whatever <b>it takes</b>");
```

- **Many application frameworks (especially web frameworks) provide templates that automatically encode context-based elements**
 - Obviously, this type of technologies saves us a lot of problems: forget encoding operations
- **Examples**
 - Strict Contextual Scaping (from AngularJS): [https://docs.angularjs.org/api/ng/service/\\$sce](https://docs.angularjs.org/api/ng/service/$sce)
 - Go Templates (from Go): <https://golang.org/pkg/html/template/>
 - Node.js Template Engines (for SDI) : <https://www.veracode.com/blog/secure-development/nodejs-template-engines-whydefault-encoders-are-not-enough>
 - Spring Boot (for SDI): Use Spring Security features (<https://spring.io/guides/gs/securing-web/>)
- **It is always convenient to see if our framework has this type of technology**
 - Or choose it only if you own these technologies, as you take security seriously

ENCODING THE OUTPUTS



- Our data must be encoded to not cause problems to other systems that depend on our data
 - Encoding avoids taking advantage of our flaws to deploy attacks on third-party software
 - To do this, we must follow the same rules that we saw in the previous slides
- Try to avoid using OS commands as recipients of our data
- Remember: Encoding is not encryption
 - It only converts character encoding to ASCII, the format supported by HTTP
 - It is easily decoded by multiple tools/APIs, even online (www.base64decode.org, Cyberchef)
 - One popular encoding method is Base64

2. Output Encoding

Ensure compliance with mandatory output encoding principles

- Conduct all encoding on a trusted system (e.g., The server)
- Use a standard, tested routine for each type of outbound encoding performed over the data that the application emits

Encode output according to its context

- Encode all data returned to the client that originates outside the application's trust boundary. HTML entity encoding is one example, but does not work in all cases
- Contextually sanitize all untrusted outputs that are going to be used in SQL, XML, and LDAP queries
- Encode all characters, unless they are known to be safe for the intended interpreter
- Sanitize all output of un-trusted data that is going to be used with OS commands

https://www.base64decode.org

BASE64

Decode and Encode

Do you have to deal with Base64 format? Then this site is perfect for you! Use our super handy online tool.

Decode from Base64 format

Simply enter your data then push the decode button.

RXN0byBubyBlcyByZWVudGUgdW4gc2VjcmV0bw==

For encoded binaries (like images, documents, etc.) use the file upload form a little further down on this page.

UTF-8 Source character set.

☐ Decode each line separately (useful for when you have multiple entries).

☒ Live mode OFF Decodes in real-time as you type or paste (supports only the UTF-8 character set).

< DECODE > Decodes your data into the area below.

Esto no es realmente un secreto

[< Go to Index](#)

ACCESS CONTROL

Authorization. MFA



Departamento de Informática
Universidad de Oviedo



Universidad de Oviedo

ACCESS CONTROL (AUTHORIZATION)



- It is best to have a single centralized authorization mechanism for the entire application
- Each request must be explicitly authorized
- Restrict access to anything valuable to authorized users
 - Functions, information...
- Allowed user accounts and their privileges must be set in the design phase

3. Access Control (Authorization)

Ensure compliance with mandatory authorization principles

- Access controls should fail securely
- Deny all access if the application cannot access its security configuration information
- Use a single application-wide component to check access authorization. This includes libraries that call external authorization services
- Use only trusted system objects (e.g., server-side session objects), for making access authorization decisions

Enforce proper restrictions only to authorized users

- Restrict access to application data to only authorized users
- Restrict access to files or other resources, including those outside the application's direct control, to only authorized users
- Restrict access to protected functions to only authorized users
- Restrict access to protected URLs to only authorized users
- Restrict access to security-relevant configuration information to only authorized users
- Restrict access to services to only authorized users
- Restrict access to user and data attributes and policy information used by access controls
- Restrict direct object references to only authorized users

Implement security best access control practices in business logic code

- Enforce application logic flows to comply with business rules
- If state data must be stored on the client, use encryption and integrity checking server side to catch state tampering
- Limit the number of transactions a single user or device can perform in a given period of time. The transactions/time should be above the actual business requirement, but low enough to deter automated attacks
- Server-side implementation and presentation layer representations of access control rules must match
- Use the "referer" header as a supplemental check, it should never be the sole authorization check: it can be spoofed

Implement proper authorization management

- Create an Access Control Policy to document an application's business rules, data types and access authorization criteria and/or processes so that access can be properly provisioned and controlled. This includes identifying access requirements for both the data and system resources
- Enforce authorization controls on every request, including those made by server-side scripts, "includes" and requests from advanced client-side technologies like AJAX
- If long authenticated sessions are allowed, periodically re-validate a user's authorization to ensure that their privileges have not changed and if they have, log the user out and force re-authentication
- Segregate privileged logic from other application code

Manage user accounts securely

- Implement account auditing and enforce the disabling of unused accounts (e.g., After no more than 30 days from the expiration of an account's password)
- Service accounts or those supporting connections to/from external systems must have the least privilege possible
- The application must support disabling of accounts and terminating sessions when authorization ceases (e.g., Changes to role, employment status, business process, etc.)

4. Authentication and Password Management			
Ensure compliance with mandatory authentication principles		Implement secure authentication mechanism practices	
	All authentication controls must be enforced on a trusted system (e.g., The server)		Validate the authentication data only on completion of all data input, especially for sequential authentication implementations
	All authentication controls should fail securely		Authentication failure responses should not indicate which part of the authentication data was incorrect. For example, instead of "Invalid username" or "Invalid password", just use "Invalid username and/or password" for both. Error responses must be truly identical in both display and source code
	Establish and utilize standard, tested, authentication services whenever possible		If using third party code for authentication, inspect the code carefully to ensure it is not affected by any malicious code
	Require authentication for all pages and resources, except those specifically intended to be public		Use Multi-Factor Authentication for highly sensitive or high value accounts
	Segregate authentication logic from the resource being requested and use redirection to and from a centralized authentication control		Utilize authentication for connections to external systems that involve sensitive information or functions
	Use a centralized implementation for all authentication controls, including libraries that call external authentication services	Manage account passwords securely in GUIs / application logic	
Implement proper password storage policies			All administrative and account management functions must be at least as secure as the primary authentication mechanism
	Authentication credentials for accessing services external to the application should be encrypted and stored in a protected location on a trusted system (e.g., The server). Source code is NOT a secure location		Disable "remember me" functionality for password fields
	If your application manages a credential store, it should ensure that only cryptographically strong one-way salted hashes of passwords are stored and that the table/file that stores the passwords and keys is writeable only by the application. Do not use the MD5 algorithm		Enforce account disabling after an established number of invalid login attempts (e.g. five attempts is common). The account must be disabled for a period of time sufficient to discourage brute force guessing of credentials, but not so long as to allow for a denial-of-service attack to be performed
	Password hashing must be implemented on a trusted system (e.g., The server).		Enforce changing temporary passwords on the next account usage
Enforce secure password transmission			If using email-based resets, only send email to a pre-registered address with a temporary link/password
	Only send passwords over an encrypted connection or as encrypted data, such as in an encrypted email		Notify users when a password reset occurs
	Use only HTTP POST requests to transmit authentication credentials		Password entry should be obscured on the user's screen (e.g., on web forms use the input type "password")
Enforce a secure password policy			Password reset and changing operations require the same level of controls as account creation and authentication
	Enforce password complexity requirements established by the application policy or regulation. Authentication credentials should be strong enough to withstand typical attacks (e.g., requiring the usage of alphanumeric and special characters)		Password reset questions should support sufficiently random answers. (e.g., "favorite book" is a bad question because "The Bible" is a very common answer). However, it is best not to use this reset mechanism
	Change all vendor-supplied default passwords and user IDs or disable the associated accounts		Re-authenticate users prior to performing critical operations
	Enforce password changes based on requirements established in the application policy or regulation. Critical systems may require more frequent changes. The time between resets must be controlled		Temporary passwords and links should have a short expiration time
	Enforce password length requirements established by the application policy or regulation. Consider the use of multi-word pass phrases	Implement a password log	
	Passwords should be at least 1 day old before they can be changed, to prevent attacks on password re-use		Implement monitoring to identify attacks against multiple user accounts, utilizing the same password (password spray). This attack pattern is used to bypass standard lockouts, when user IDs can be harvested or guessed
	Prevent password re-use		The last use (successful or unsuccessful) of a user account should be reported to the user at their next successful login

IMPLEMENTING AN MFA



Computer Security

- **Implementing an MFA can seem to be a very advanced and difficult operation**
 - But the usage of MFA is so common today that there are APIs that considerably simplify their implementation
- **We must look for the best API for our framework / language**
- **And, if possible, use existing third-party MFA solutions**
- **Some examples of implementation**
 - Google OAuth2: <https://developers.google.com/identity/protocols/OAuth2>
 - Google Authenticator: https://medium.com/@richb_/easy-two-factor-authentication-2fa-with-googleauthenticator-php-108388a1ea23
 - PHP (for **SEW**): https://medium.com/@richb_/easy-two-factor-authentication-2fa-with-googleauthenticator-php-108388a1ea23
 - Node .js (for **SDI**): <https://developer.okta.com/blog/2018/05/22/simple-multifactor-authentication-innode>
 - Spring Boot (for **SDI**): <https://sultanov.dev/blog/multi-factor-authentication-with-spring-boot-andoauth2/>

[< Go to Index](#)

CRYPTOGRAPHY AND DATA SECURITY

How to protect your private data



Departamento de Informática
Universidad de Oviedo



Universidad de Oviedo

PRACTICAL TIPS ON CRYPTOGRAPHY



Computer Security

- Never expose secret information
- Always encrypt private information
- Never use unreliable, outdated or unvalidated cryptography modules
- Follow the principles described in cryptography and application security theory topics

5. Cryptographic Practices

Ensure compliance with mandatory cryptography management policies

All cryptographic functions used to protect secrets from the application user must be implemented on a trusted system (e.g., The server)

Establish and utilize a policy and process for how cryptographic keys will be managed

Protect master secrets or passwords from unauthorized access

Use only adequate cryptography modules

All random numbers, random file names, random GUIDs, and random strings should be generated using a cryptographic module's recommended random number generator when these random values are intended to be un-guessable

Cryptographic modules should fail securely

Cryptographic modules used by the application should be compliant to FIPS 140-2 or an equivalent standard. (See <http://csrc.nist.gov/groups/STM/cmvp/validation.html>)

- To avoid problems and to ensure maximum protection, **always use TLS**
 - Regardless of the content of the transmitted data
 - **HTTPs is mandatory on production!**
- Always use a secure TLS version
- Follow the principles described in the cryptography topic
- Use proper TLS settings on the web server
 - This is not difficult if we follow Mozilla's recommendations
 - <https://ssl-config.mozilla.org/>

6. Communication Security	
Ensure compliance with mandatory communication security principles	
	Filter parameters containing sensitive information from the HTTP referrer, when linking to external sites
	Implement encryption for the transmission of all sensitive information (or for all information, in general). This should include TLS for protecting the connection and may be supplemented by discrete encryption of sensitive files or non-HTTP based connections
	Specify character encodings for all connections
Implement secure TLS management policies	
	Failed TLS connections should not fall back to an insecure connection
	TLS certificates should be valid and have the correct domain name, not be expired, and be installed with intermediate certificates when required
	Utilize a single standard TLS implementation that is configured appropriately
	Utilize TLS connections for all content requiring authenticated access and for all other sensitive information
	Utilize TLS for connections to external systems that involve sensitive information or functions

[< Go to Index](#)

GENERAL SECURE CODE CREATION PRACTICES

Concrete actions when creating your code



Departamento de Informática
Universidad de Oviedo



Universidad de Oviedo

GENERAL CODE CREATION PRACTICES



- **Update everything**
 - A vulnerability in our software (and its CVE) can be the problem
- **Use measures to prevent the forgery of third-party dependencies**
 - File hashes (**Topic 2**) or SCA tools (**Topic 6**)
- **Don't use metaprogramming**
 - **TPP**: `eval`, `exec`...
- **Expose yourself as little as possible**
 - Minimize privileged access
 - Implement measures to hinder code reverse engineering (code obfuscation)
- **Program defensively**
 - <https://latteandcode.medium.com/qu%c3%a9-es-eso-de-la-programaci%c3%b3n-defensiva6a89d3bcab8e>

7. General Coding Practices

Ensure compliance with mandatory secure coding practices

Use tested and approved managed code rather than creating new unmanaged code for common tasks

Implement a secure policy to use external APIs / Libraries

Implement safe updating. If the application will utilize automatic updates, then use cryptographic signatures for your code and ensure your download clients verify those signatures. Use encrypted channels to transfer the code from the host server

Review all secondary applications, third party code and libraries to determine if they are necessary and validate its safe functionality, as these can introduce new vulnerabilities

Use checksums or hashes to verify the integrity of interpreted code, libraries, executables, and configuration files

Utilize task specific built-in APIs to conduct operating system tasks. Do not allow the application to issue commands directly to the Operating System, especially using application-initiated command shells

Use secure programming techniques

Avoid calculation errors by understanding your programming language's underlying representation and how it interacts with numeric calculation. Pay close attention to byte size discrepancies, precision, signed/unsigned distinctions, truncation, conversion and casting between types, "not-a-number" calculations, and how your language handles numbers that are too large or too small for its underlying representation

Do not pass user supplied data to any dynamic execution function

Explicitly initialize all your variables and other data stores, either during declaration or just before the first usage

In cases where the application must run with elevated privileges, raise privileges as late as possible, and drop them as soon as possible

Protect shared variables and resources from inappropriate concurrent access

Restrict users from generating new code or altering existing code

Utilize locking to prevent multiple simultaneous requests or use a synchronization mechanism to prevent race conditions

(EXTRA) counter-reverse engineering features

Use a packer

Use code obfuscation

Use code trimming

DATA PROTECTION



- **Always apply the least necessary privilege to data access**
- **Do not give information about yourself**
 - Users, staff, developers..., the structure of the application, or the used technologies
- **Avoid remembering sensitive data in the application's GUI**
- **Protect source code at all costs**
- **These techniques were discussed in**
 - **Topic 2** (Cryptography)
 - **Topic 3** (OS Security)
 - **Topic 6** (Application Security)

8. Data Protection

Implement secure data permission handling

- Implement appropriate access controls for sensitive data stored on the server. This includes cached data, temporary files and data that should be accessible only by specific system users
- Implement least privilege, restrict users to only the functionality, data and system information that is strictly required to perform their tasks
- Protect all cached or temporary copies of sensitive data stored on the server from unauthorized access and purge those temporary working files as soon as they are no longer required.
- Protect server-side source-code from being downloaded by a user

Encrypt data securely

- Encrypt highly sensitive stored information, like authentication verification data, even on the server side. Always use well vetted algorithms, and see "Cryptographic Practices" for additional guidance

Implement secure information storage

- Do not store passwords, connection strings or other sensitive information in cleartext or in any non-cryptographically secure manner on the client side. This includes embedding data in insecure formats like: Microsoft viewstate or compiled code
- The application should support the removal of sensitive data when that data is no longer required. (e.g., personal information or certain financial data)

Remove any information the application gives about itself

- Do not include sensitive information in HTTP GET request parameters
- Remove comments in user accessible production code that may reveal data about the backend system or other sensitive information
- Remove file metadata and any kind of additional information files may give
- Do not use default templates of structures that may reveal technologies you are using
- Remove unnecessary application and system documentation as this can reveal useful information to attackers

Implement secure GUI practices to deal with sensitive data

- Disable auto complete features on forms expected to contain sensitive information, including authentication
- Disable client-side caching on pages containing sensitive information. Cache-Control: no-store, may be used in conjunction with the HTTP header control "Pragma: no-cache", which is less effective, but is HTTP/1.0 backward compatible

SESSION MANAGEMENT



● Stealing sessions allow taking other users identities: **it MUST be avoided**

- Never reveal a valid session ID
- Follow the recommended session ID secure handling rules

● Follow the guidelines to implement cookies securely

- Session IDs are typically stored as cookie attributes

● Control user suspicious behavior

- Simultaneous sessions from different geolocations...

● Make sure that the logout function invalidates the session

9. Session Management

Ensure compliance with mandatory session management practices

- Session identifier creation must always be done on a trusted system (e.g., The server)
- Session management controls should use well vetted algorithms that ensure true random session identifiers
- Use the server or framework's session management controls. The application should only recognize these session identifiers as valid

Implement secure cookie management

- Do not expose session identifiers in URLs, error messages or logs. Session identifiers should only be in the HTTP cookie header. For example, do not pass session identifiers as GET parameters
- Set cookies with the HttpOnly attribute, unless you specifically require client-side scripts within your application to read or set a cookie's value
- Set the "secure" attribute for cookies transmitted over an TLS connection
- Set the domain and path for cookies containing authenticated session identifiers to an appropriately restricted value for the site

Implement secure login management

- Do not allow concurrent logins with the same user ID, unless it is a requisite
- Generate a new session identifier on any re-authentication
- If a session was established before login, close that session and establish a new one after a successful login

Implement secure logout management

- Disallow persistent logins and enforce periodic session terminations, even when the session is active, especially for applications connecting to critical systems. Termination times should support business requirements, and the user should receive a notification with sufficient time to mitigate negative effects
- Establish a session inactivity timeout that is as short as possible, based on balancing risk and business functional requirements. In most cases it should be no more than several hours
- Logout functionality should be available from all pages protected by authorization
- Logout functionality should fully terminate the associated session or connection

Handle session data securely

- Generate a new session identifier and deactivate the old one periodically. (This can mitigate certain session hijacking scenarios where the original identifier was compromised)
- Generate a new session identifier if the connection security changes from HTTP to HTTPS, as can occur during authentication. Within an application, it is recommended to consistently utilize HTTPS rather than switching between HTTP to HTTPS
- Protect server-side session data from unauthorized access, by other users of the server, by implementing appropriate access controls on the server
- Supplement standard session management for highly sensitive or critical operations by utilizing per-request, as opposed to per-session, strong random tokens or parameters
- Supplement standard session management for sensitive server-side operations, like account management, by utilizing per-session strong random tokens or parameters. This method can be used to prevent Cross Site Request Forgery attacks

ERROR HANDLING



● Avoid errors showing too much information

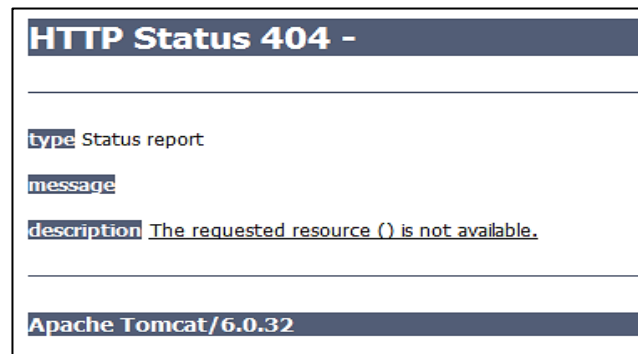
- Generic messages for the user, detailed for developers
- Avoid errors such as the one in the image

● Log all important error messages

- But make sure they don't reveal private information!

● Limit access to logs

- Only authorized users can access



10. Error Handling and Logging

Ensure compliance with mandatory logging practices

All logging controls should be implemented on a trusted system (e.g., The server)

Do not disclose excessive information on error messages

Do not disclose sensitive information in error responses, including system details, session identifiers or account information

Implement generic error messages and use custom error pages

Use error handlers that do not display debugging or stack trace information

Implement secure error handling

Error handling logic associated with security controls should deny access by default

Properly free allocated memory when error conditions occur

The application should handle application errors and not rely on the server configuration

Enable secure error log creation and management

Do not store sensitive information in logs, including unnecessary system details, session identifiers or passwords

Ensure log entries that include un-trusted data will not execute as code in the intended log viewing interface or software

Ensure logs contain important log event data

Ensure that a mechanism exists to conduct log analysis

Logging controls should support both success and failure of specified security events

Restrict access to logs to only authorized individuals

Use a cryptographic hash function to validate log entry integrity

Utilize a unique "master" function for all logging operations

Log the adequate types of information related to security

Log all access control failures

Log all administrative functions, including changes to the security configuration settings

Log all apparent tampering events, including unexpected changes to state data

Log all authentication attempts, especially failures

Log all backend TLS connection failures

Log all input validation failures

Log attempts to connect with invalid or expired session tokens

Log all system exceptions

Log cryptographic module failures

FILE MANAGEMENT



- Try to avoid a file upload functionality in the application
- If it is mandatory, follow recommended file upload rules
 - https://cheatsheetseries.owasp.org/cheatsheets/File_Upload_Cheat_Sheet.html
 - Especially those that prevent malicious file uploads
 - Or, better, **use a tested and verified third-party module that implements this**
- Always check access to external files
- Never allow users to specify routes
 - Even partially!
- Use allowlists for restricting file types

11. File Management

Ensure compliance with mandatory file management practices

Ensure application files and resources are read-only

Securely handle includes and references to files

Do not pass user supplied data directly to any dynamic include function

Do not pass directory or file paths, use index values mapped to pre-defined list of paths

Do not pass user supplied data into a dynamic redirect. If this must be allowed, then the redirect should accept only validated, relative path URLs

Never send an absolute file path to the client

When referencing existing files, use a whitelist of allowed file names and types. Validate the value of the parameter being passed and if it does not match one of the expected values, either reject it or use a hard-coded default file for the content instead

Implement secure file uploads, if this functionality is necessary

Do not save files in the same web context as the application. Files should either go to the content server or in the database

Implement safe uploading in Linux by mounting the targeted file directory as a logical drive using the associated path or a chrooted environment

Limit the type of files that can be uploaded to only those types that are needed for business purposes

Prevent or restrict the uploading of any file that maybe interpreted by the web server

Require authentication before allowing a file to be uploaded

Scan user uploaded files for viruses and malware

Turn off execution privileges on file upload directories

Validate uploaded files are the expected type by checking file headers. Checking for file type by extension alone is not sufficient

VALIDATING FILES THAT ARE UPLOADED TO AN APPLICATION

- Many applications (especially websites) allow users to upload files, such as profile pictures
- This functionality can be used to attempt to load an unauthorized file type
 - Putting executable code or another type of malicious file on a server
- Therefore, we must correctly check the uploaded files
 - The uploaded **file name** must be validated to make sure it has an allowed extension
 - The uploaded **file size** should not be larger than the maximum allowed file size
 - Which should always be set!
 - If **uploading zip files** is supported, you must perform the following checks before you unzip it
 - Consistent and not excessively long destination route
 - Reasonable compression level (extreme compression levels can overwhelm server processing)
 - Estimating the uncompressed size: avoiding attacks such as **zip bombs**

VALIDATING FILES THAT ARE UPLOADED TO AN APPLICATION

- **Typical uploaded Office documents (Word, Excel...) can be malicious**
 - They can contain malicious code that runs when a user opens them
- **The most common types of malicious file attack vectors are**
 - Word/Excel/PowerPoint documents: Using VBA Macros and the OLE Package
 - **PDF** documents: with malicious code inserted as an attachment
 - **Images**: Malicious code embedded in the file, executable binary files with an image extension
- **To avoid problems of this type we must**
 - Word/Excel/PowerPoint/PDF: Detect when a document contains OLE or JavaScript objects
 - If this is detected, it is safer to block the upload process and **do NOT to try to “clean” the file**
 - **Images**: blocking the upload, as there is no legitimate reason why the user should upload images with embedded code
- **All validations must be coded manually or (better) look for reputable third-party components that do so. Examples in Java:**
 - https://github.com/OWASP/CheatSheetSeries/blob/master/cheatsheets/Protect_FileUpload_Against_Malicious_File.md

STORING FILES THAT ARE UPLOADED TO AN APPLICATION

- A new file name must be used to save all uploaded files
- The name must never contain any data provided by the user, and must be random
 - E.g. : If you upload a file `photo.jpg` you can rename it to something like `as345ajrbbe.jpg` (random)
 - You get the following advantages
 - That the uploaded file cannot be directly accessed
 - Avoid malicious names created to evade filters, such as `photo.jpg;.asp` or `/../../../../../../../../photo.jpg`
 - Every time you download an image from a social network this is exactly what happens
- Uploaded files should be scanned for malicious content
 - Anti-malware, static scanning, etc.
- The file path must not be specified by the client, **the server must set it**

SERVING FILES THAT ARE UPLOADED TO AN APPLICATION

- **When serving user-uploaded content you must ensure that the **correct MIME type is used****
 - For example, images should be used as an `image/jpeg` type
- **There are files with special significance and, if uploaded, they could lead to vulnerabilities**
 - `crossdomain.xml/clientaccesspolicy.xml`: allows cross-domain data loading in Java
 - `.htaccess` and `.htpasswd`: Contain web server configuration options for the directory that has them, and may overwrite the default server settings
- **Never allow uploading files with executable code**
 - Examples: `.aspx`, `.asp`, `.css`, `.xhtml`, `.rhtml`, `.shtml`, `.jsp`, `.js`, `.pl`, `.php`, `.cgi...`

VERIFYING FILES THAT ARE UPLOADED TO AN APPLICATION

- **When uploading images, use image rewrite libraries that verify that the image is valid and remove “foreign” content**
 - These libraries also allow you to find out the type of content by analyzing the image
 - This type is the one to use to determine the extension of the saved image file
 - And not the one detected in the image header
 - **Images can be maliciously altered**
 - Once this data is obtained, the image should be checked again to see if its type is within the list of allowed image types (.jpg, .png, .gif...)
- **The following link can help with these tasks**
 - <https://stackoverflow.Com/questions/55869/determine-file-type-of-an-image>

[< Go to Index](#)

SYSTEM CONFIGURATION, DEVELOPMENT TOOLS AND DEPENDENCIES

Security in application configuration



Departamento de Informática
Universidad de Oviedo



Universidad de Oviedo

SYSTEM CONFIGURATION



● On the production server...

- Do not upload unnecessary content
- Do not enable unnecessary functionality
- Configure it securely
 - Following validated automated hardening procedures (topic 4)
 - CIS Benchmarks, STIGs, OpenSCAP...

● If it is a web server, Apache 2 / IIS installation is reviewed in [ASR](#)

13. System Configuration

Use a secure software update policy

- Ensure servers, frameworks and system components are running the latest approved version
- Ensure servers, frameworks and system components have all patches issued for the version in use

Deploy a secure web server configuration regarding software elements

- Define which HTTP methods, GET or POST, the application will support and whether it will be handled differently in different pages in the application
- Disable unnecessary HTTP methods, such as WebDAV extensions. If an extended HTTP method that supports file handling is required, utilize a well-vetted authentication mechanism
- If the webserver handles both HTTP 1.0 and 1.1, ensure that both are configured in a similar way or that you understand any difference that may exist (e.g., handling of extended HTTP methods)
- Remove unnecessary information from HTTP response headers related to the OS, web-server version and application frameworks
- Restrict the web server, process and service accounts to the least privileges possible
- Turn off directory listings

Minimize and remove unnecessary files in the server

- Prevent disclosure of your directory structure in the robots.txt file by placing directories not intended for public indexing into an isolated parent directory. Then "Disallow" that entire parent directory in the robots.txt file rather than Disallowing each individual directory
- Remove all unnecessary functionality and files
- Remove test code or any functionality not intended for production, prior to deployment

Deploy a secure application configuration policy

- Implement a software change control system to manage and record changes to the code both in development and production
- Implement an asset management system and register system components and software in it
- Isolate development environments from the production network and provide access only to authorized development and test groups. Development environments are often configured less securely than production environments and attackers may use this difference to discover shared weaknesses or as an avenue for exploitation
- The security configuration store for the application should be able to be output in human readable form to support auditing
- When exceptions occur, fail securely

DATABASE SECURITY



- **ALWAYS USE PARAMETERIZED QUERIES**
- **Treat the DB like any external system**
 - Properly encode the input
 - Do not trust the data you return; validate them
 - Update the DBMS software
- **Use a security management policy**
 - Passwords (strong, non-default...)
 - Remove unnecessary content
 - And use the corresponding CIS Benchmark (or similar) for the secure configuration of the DBMS

14. Database Security	
Ensure compliance with mandatory secure database management principles	
	Use strongly typed parameterized queries
Securely handle types	
	Ensure that variables are strongly typed
	Utilize input validation and output encoding and be sure to address meta characters. If these fail, do not run the associated database command
Enforce proper secure database permission	
	The application should connect to the database with different credentials for every trust distinction (e.g., user, read-only user, guest, administrators)
	The application should use the lowest possible level of privilege when accessing the database
Implement secure database management policies	
	Close the connection as soon as possible
	Connection strings should not be hard coded within the application. Connection strings should be stored in a separate configuration file on a trusted system, and they should be encrypted.
	Disable any default accounts that are not required to support business requirements
	Remove or change all default database administrative passwords. Utilize strong passwords/phrases or implement multi-factor authentication
	Remove unnecessary default vendor content (e.g., sample schemas)
	Turn off all unnecessary database functionality (e.g., unnecessary stored procedures or services, utility packages, install only the minimum set of features and options required (attack surface reduction))
	Use secure credentials for database access
	Use stored procedures to abstract data access and allow for the removal of permissions to the base tables in the database

SECURE USE OF DEVELOPMENT FRAMEWORKS/LANGUAGES



● Modern frameworks/languages have built-in anti-XSS protections

- They must be correctly used to create our applications (**never validate manually**)
- **Do not deactivate them**

● Always use the latest possible stable version (patches)

- Deploy to production using the framework options (e.g.: Angular 2+: `ng build-prod`)

● Many frameworks/languages integrate security protections

- The image lists common PHP ones (**SEW**)
- Next slide, for Node.js and Spring Boot (**SDI**)
- HTML5 has many security features
 - https://github.com/OWASP/CheatSheetSeries/blob/master/cheatsheets/HTML5_Security_Cheat_Sheet.md

REV 2 / 2010-Apr-12

PHP Application Security Checklist

BASIC

- ☐ Strong passwords are used.
- ☐ Passwords stored safely.
- ☐ `register_globals` is disabled.
- ☐ Magic quotes is disabled.
- ☐ `display_errors` is disabled.
- ☐ Server(s) are physically secure.

INPUT

- ☐ Input from `$_GET`, `$_POST`, `$_COOKIE`, and `$_REQUEST` is considered tainted.
- ☐ Understood that only some values in `$_SERVER` and `$_ENV` are untainted.
- ☐ `$_SERVER['PHP_SELF']` is escaped where used.
- ☐ Input data is validated.
- ☐ `\0` (null) is discarded in input.
- ☐ Length of input is bounded.
- ☐ Email addresses are validated.
- ☐ Application is aware of small, very large, zero, and negative numbers. Sci. notation too.
- ☐ Application checks for invisible, look-alike, and combining characters.
- ☐ Unicode control characters stripped out when required.
- ☐ Outputted data is sanitized.
- ☐ User-inputted HTML is sanitized with `HTMLPurifier`.
- ☐ User-inputted CSS is sanitized using a white-list.
 - ☐ Abusable properties (position, margin, etc.) are handled.
 - ☐ CSS escape sequences are handled.
 - ☐ JavaScript in CSS is discarded (expressions, behaviors, bindings).
- ☐ URLs are sanitized and unknown and unwanted protocols are disallowed.
- ☐ Embedded plugins are restricted from executing JS.
- ☐ Embedded plugin files (Flash movies) are embedded in a manner so that only the intended plugin is loaded.
- ☐ The application uses a safe encoding.
 - ☐ An encoding is specified using a HTTP header.
 - ☐ Inputted data is verified to be valid for your selected encoding if using an unsafe encoding.

FILE UPLOADS

- ☐ Application verifies file type.
 - ☐ User-provided mime type value is ignored.
- ☐ Application analyzes the content of files to determine their type.
- ☐ It is understood that a perfectly valid file can still contain arbitrary data.
- ☐ Application checks the file size of uploaded files.
 - ☐ `MAX_FILE_SIZE` is not depended upon.
 - ☐ File uploads cannot "overtake" available space.
- ☐ Content is checked for malicious content.
 - ☐ Application uses a malware scanner (if req.).
 - ☐ Uploaded HTML files are displayed securely.
- ☐ Uploaded files are not moved to a web-accessible directory.
- ☐ Extensive path checks are used when serving files.
- ☐ Uploaded files are not served with `include()`.
- ☐ Uploaded files are served as an attachment using the `Content-Disposition` header.
- ☐ Application sends the `X-Content-Type-Options: nosniff` header.
- ☐ Files are not served as "application/octet-stream", "application/unknown", or "plain/text" unless necessary.

DATABASE

- ☐ Data inserted into the database is properly escaped or parameterized/prepared statements are used.
 - ☐ `addslashes()` is not used.
- ☐ Application does not have more privileges to the database than necessary.
- ☐ Remote connections to the database are disabled if they are unnecessary.

SERVING FILES

- ☐ User input is not directly used in a pathname.
 - ☐ Directory traversal is prevented.
 - ☐ Null (`\0`) in paths filtered.
 - ☐ Application is aware of "..."

- ☐ PHP streams are filtered.
- ☐ Access to files is not restricted by hiding the files.
- ☐ Remote files not included with `include()`.

AUTHENTICATION

- ☐ Bad password throttling.
 - ☐ CAPTCHA is used.
- ☐ SSL used to prevent MITM.
- ☐ Passwords are not stored in a cookie.
- ☐ Passwords are hashed.
 - ☐ Per-user salts are used.
 - ☐ `crypt()` is used with sufficient number of rounds.
 - ☐ MD5 is not used.
- ☐ Users are warned about obvious password recovery questions.
- ☐ Account recovery forms do not reveal email existence.
- ☐ Pages that send emails are throttled.

SESSIONS

- ☐ Sessions only use cookies. (`session.use_only_cookies`)
- ☐ On logout, session data is destroyed.
- ☐ Session is recreated on authorization level change.
- ☐ Sites on the same server use different session storage dirs.

3RD-PARTIES

- ☐ CSRF issues are prevented with tokens/keys.
 - ☐ Referrers are not relied upon.
- ☐ Pages that perform actions use POST.
- ☐ Important pages (logout, etc.) are protected.
- ☐ Your pages are not written in a way (i.e. JSON, JS-like) where they can be included and read on a remote website successfully.
- ☐ Aware that Flash can bypass referrer checks to load images and sound files.
- ☐ The following things will not reveal significant information if included remotely:
 - ☐ Images.
 - ☐ Pages that take a longer time to load.

- ☐ CSS files.
- ☐ Existence or ordering of frames.
- ☐ Existence of a JS variable.
- ☐ Detected visit of a URL.

- ☐ Inclusion of your website in an inline frame with JS disabled does not reveal a threat.
- ☐ Application uses frame bursting code and sends the `X-Frame-Options` header.

MISCELLANEOUS

- ☐ A cryptographically secure PRNG is used for secret randomly-generated IDs (activation links, secret IDs, etc.).
 - ☐ Suhosin is installed or you are not using `rand()` or `mt_rand()` for this.
- ☐ Anything that consumes a lot of resources should be throttled and limited.
 - ☐ Pages that use 3rd-party APIs are throttled.
- ☐ You did not create your own encryption algorithm.
- ☐ Arguments to external programs (i.e. `exec()`) are validated.
- ☐ Generic internal and external redirect pages are secured.
- ☐ Precautions taken against the source code of your PHP pages being shown due to misconfiguration.
- ☐ Configuration and critical files are not in a web-accessible directory.

SHARED HOSTING

- ☐ Using a secure shared host where users cannot access the files of other users.
- ☐ Aware that fellow shared hosting users:
 - ☐ Can, if on the same IP address, issue requests against your site with `XMLHttpRequest` in IE6.
 - ☐ Can access your website from 127.0.0.1 or ::1.
 - ☐ Can host a server on the same IP address.
 - ☐ Are not "remote" as far as your DB is concerned.
- ☐ Session & file upload directories are not shared.



Find the annotated original at <http://sk89q.com/phpsec/>
© 2010 sk89q. You are free to reproduce this without modification.



1. Use HTTPS in Production

To use HTTPS in your Spring Boot app, extend `WebSecurityConfigurerAdapter` and require a secure connection (Note: this forces HTTPS in development also):

```
@Configuration
public class WebSecurityConfig extends
    WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http)
        throws Exception {

        http.requiresChannel().requiresSecure();
    }
}
```

2. Test Your Dependencies

Ensure your application does not use dependencies with known vulnerabilities. Use a tool like Snyk to:

- ✓ Test your app dependencies for known vulnerabilities.
- ✓ Automatically Fix issues that exist.
- ✓ Continuously Monitor for new vulns.

3. Enable CSRF Protection

Spring Security enables [CSRF support](#) by default. If you use a JavaScript framework, configure the `CookieCsrfTokenRepository` so cookies are not HTTP-only.

```
@EnableWebSecurity
public class WebSecurityConfig extends
    WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http)
        throws Exception {

        http.csrf()
            .csrfTokenRepository(CookieCsrfTokenRepository
                .withHttpOnlyFalse());
    }
}
```

4. Use a Content Security Policy

Enable to avoid XSS attacks.

Spring Security provides a number of security headers by default, but not CSP. Enable in your Spring Boot app as follows:

```
@EnableWebSecurity
public class WebSecurityConfig extends
    WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http)
        throws Exception {

        http.headers().contentSecurityPolicy("script-src
            'self' https://trustedscripts.example.com; object-src
            https://trustedplugins.example.com; report-uri /csp-
            report-endpoint/");
    }
}
```

5. Use OpenID Connect

OpenID Connect (OIDC) provides user information via an ID token in addition to an access token.

Query the `/userinfo` endpoint for additional user information.

6. Use Password Hashing

Don't store passwords in plain text. Spring Security doesn't allow plain text passwords by default.

`PasswordEncoder` is the main interface for password hashing in Spring Security:

```
public interface PasswordEncoder {
    String encode(String rawPasswd);
    boolean matches(String rawPasswd, String encPasswd);
}
```

7. Use the Latest Releases

The [start.spring.io](#) site automatically uses the latest versions of libraries for new apps.

For existing apps, when upgrades aren't possible, consider patches from a security vendor, like Snyk.

8. Store Secrets Securely

Store secrets in [Vault by HashiCorp](#) or [Spring Vault](#)

Extract secrets from the Spring Vault using annotations.

```
@Value("${password}")
String password;
```

9. Pen Test Your App

The [OWASP ZAP](#) security tool is a proxy that performs penetration testing. It runs in Spider and Active Scan modes to identify and map all hyperlinks in your app, and automatically test your selected targets against a list of potential vulnerabilities.

10. Have Your Security Team do a Code Review

Code reviews are essential. Ensure all your code changes undergo:

- ✓ a security team code review.
- ✓ Automatic testing on every pull request using Snyk

Authors:

[@sjmable](#)

Java Champion and Developer Advocate at Snyk

[@mraible](#)

Java Champion and Developer Advocate at Okta



1. Avoid publishing secrets to the npm registry

- 1 Run `npm publish --dry-run` to review the package before publishing
- 2 Put sensitive files in `.gitignore`
- 3 Use the `files` property in `package.json` to whitelist files and directories

2. Enforce lockfile

Freeze lockfile and ensure the npm CLI installs per lockfile only, without changing it. In CI and build environments favor:

- 1 `$ npm ci`
- 2 `$ yarn install --frozen-lockfile`

3. Minimize attack surface—ignore run-scripts

Malicious packages take advantage of key lifecycle events when an npm install runs arbitrary commands.

To minimize this attack surface:

- 1 Assess a project's health status and credibility before installing a package
- 2 Disable run-scripts during install such as:

```
$ npm install <package> --ignore-scripts
```

4. Assess npm project health

Review a project for outdated dependencies, and assess environment health with CLI commands:

```
$ npm doctor
$ npm outdated
```

5. Scan and monitor for vulnerabilities in open source dependencies

Don't let vulnerabilities in your project dependencies reduce the security of your application. Make sure to:

- 1 Connect Snyk to GitHub or other SCMs for optimal CI/CD integration with your projects
- 2 Run `snyk test` to scan a new project from the CLI
- 3 Run `snyk monitor` to track and open PRs to automatically fix security vulnerabilities in open source dependencies.

6. Use a local npm proxy

A local private registry such as [Verdaccio](https://verdaccio.org/) will give you an extra layer of security, enabling you:

- 1 Full control of lightweight private package hosting
- 2 To cache packages and avoid being affected by network and external incidents

Easily spin up verdaccio using docker:

```
$ docker run verdaccio/verdaccio
```

7. Responsible disclosure

Publicly disclosed security vulnerabilities without prior warning and proper coordination pose a potentially serious threat.

We are happy to collaborate on responsible security disclosures for the npm community:

- 1 Report a security issue via the [vulnerability disclosure form](https://snyk.io/vulnerability-disclosure-form)
- 2 Email us at security@snyk.io

8. Enable 2FA

Enable two-factor authentication on npm with

```
$ npm profile enable-2fa auth-and-writes
```

9. Use npm author tokens

Make use of restricted tokens for querying npm packages and functionalities from CI by creating a read-only and IPv4 address range restricted token:

```
$ npm token create --read-only -cidr=192.0.2.0/24
```

10. Understand typosquatting risks

Typos in package installation can be deadly.

- 1 Be mindful when copy-pasting package install instructions to the terminal and verify authenticity.
- 2 Opt to have a logged-out npm user in your developer environment
- 3 Favor npm install with `--ignore-scripts`

Authors:

 [@liran_tal](#)
Node.js Security WG & Developer Advocate at Snyk

 [@jotadeveloper](#)
Core maintainer at Verdaccio

FRAMEWORK-SPECIFIC SECURITY TECHNOLOGIES



- Each framework / technology has its security functionalities
- We can find secure development cheat sheets in: <https://cheatsheetseries.owasp.org/Glossary.html>

- Ajax:
https://cheatsheetseries.owasp.org/cheatsheets/AJAX_Security_Cheat_Sheet.html
- .Net:
https://cheatsheetseries.owasp.org/cheatsheets/DotNet_Security_Cheat_Sheet.html
- REST:
https://cheatsheetseries.owasp.org/cheatsheets/REST_Security_Cheat_Sheet.html
- ...

- If the application is hosted in the cloud, the threat landscape changes

- <https://cloudsecurityalliance.org/press-releases/2019/08/09/csa-releases-new-research-top-threats-to-cloud-computing-egregious-eleven/>



Pragmatic Web Security
Security for developers

SECURITY CHEAT SHEET

Version 2020.001

ANGULAR AND THE OWASP TOP 10

The OWASP top 10 is one of the most influential security documents of all time. But how do these top 10 vulnerabilities resonate in a frontend JavaScript application?

This cheat sheet offers practical advice on handling the most relevant OWASP top 10 vulnerabilities in Angular applications.

DISCLAIMER This is an opinionated interpretation of the OWASP top 10 (2017), applied to frontend Angular applications. Many backend-related issues apply to the API-side of an Angular application (e.g., SQL injection), but are out of scope for this cheat sheet. Hence, they are omitted.

1 USING DEPENDENCIES WITH KNOWN VULNERABILITIES

OWASP #9

- ☐ Plan for a periodical release schedule
- ☒ Use **npm audit** to scan for known vulnerabilities
- ☒ Setup automated dependency checking to receive alerts
Github offers automatic dependency checking as a free service
- ☒ Integrate dependency checking into your build pipeline

2 BROKEN AUTHENTICATION

OWASP #2

From an Angular perspective, the most important aspect of broken authentication is maintaining state after authentication. Many alternatives exist, each with their specific security considerations.

- ☒ Decide if a stateless backend is a requirement
Server-side state is more secure, and works well in most cases

SERVER-SIDE SESSION STATE

- ☐ Use long and random session identifiers with high entropy
OWASP has a great cheat sheet offering practical advice [1]

CLIENT-SIDE SESSION STATE

- ☐ Use signatures to protect the integrity of the session state
- ☒ Adopt the proper signature scheme for your deployment
*HMAC-based signatures only work within a single application
Public/private key signatures work well in distributed scenarios*
- ☐ Verify the integrity of inbound state data on the backend
Explicitly avoid the use of "decode-only" functions in libraries
- ☐ Setup key management / key rotation for your signing keys
- ☒ Ensure you can handle session expiration and revocation

COOKIE-BASED SESSION STATE TRANSPORT

- ☐ Enable the proper cookie security properties
*Set the **HttpOnly** and **Secure** cookie attributes
Add the **_Secure** or **_Host-** prefix on the cookie name*
- ☐ Protect the backend against Cross-Site Request Forgery
*Same-origin APIs should use a **double submit cookie**
Cross-Origin APIs should force the use of CORS preflights by only accepting a non-form-based content type (e.g. **application/json**)*

AUTHORIZATION HEADER-BASED SESSION STATE TRANSPORT

- ☐ Only send the authorization header to pre-approved hosts
*Many custom interceptors send the header to **every** host*

[1] <https://bit.ly/2U8kJWc>

3 CROSS-SITE SCRIPTING

OWASP #7

PREVENTING HTML/SCRIPT INJECTION IN ANGULAR

- ☐ Use interpolation with `{{ }}` to automatically apply escaping
- ☐ Use safe property binding such as `[href]`, `[src]`, `[style.color]`
- ☐ Use binding to `[innerHTML]` to safely insert HTML data
- ☐ Do not use `bypassSecurityTrust*`() on untrusted data

PREVENTING CODE INJECTION OUTSIDE OF ANGULAR

- ☐ Avoid direct DOM manipulation
*E.g. through **ElementRef** or other client-side libraries*
- ☐ Do not combine Angular with server-side dynamic pages
- ☐ Use Ahead-Of-Time compilation (AOT)

4 BROKEN ACCESS CONTROL

OWASP #5

AUTHORIZATION CHECKS

- ☐ Implement proper authorization checks on API endpoints
*Check if the user is authenticated
Check if the user is allowed to access the specific resources*
- ☐ Do not rely on client-side authorization checks for security

CROSS-ORIGIN RESOURCE SHARING (CORS)

- ☐ Prevent unauthorized cross-origin access with a strict policy
- ☐ Avoid accepting the **null** origin in your policy
- ☐ Avoid blindly reflecting back the value of the origin header
- ☐ Avoid custom CORS implementations
Origin-matching code is error-prone, so prefer the use of libraries

5 SENSITIVE DATA EXPOSURE

OWASP #3

DATA IN TRANSIT

- ☐ Serve everything over HTTPS
- ☐ Ensure that all traffic is sent to the HTTPS endpoint
*Redirect HTTP to HTTPS on endpoints dealing with page loads
Disable HTTP on endpoints that only provide an API*
- ☐ Enable **Strict Transport Security** on all HTTPS endpoints

DATA AT REST IN THE BROWSER

- ☐ Encrypt sensitive data before persisting it in the browser
- ☐ Encrypt sensitive data in JWTs using JSON Web Encryption

MICROSOFT APPLICATION INSPECTOR



Computer Security

- Tool that helps minimize risks when using third-party software
- It is a source code parser for open-source code that identifies "interesting" features and metadata
 - Use of cryptography, connections to remote sites, platforms running...
- Not only does it identify flawed programming practices
 - But it also identifies and reports code features that may be difficult to detect by manual analysis
 - They are not evaluated; they are only reported so that you better understand what the application does
- More information
 - <https://www.microsoft.com/security/blog/2020/01/16/introducing-microsoft-application-inspector/>
 - <https://www.genbeta.com/desarrollo/microsoft-libera-herramienta-open-source-que-permite-analizar-todocodigo-fuente-aplicacion-busca-amenazas>
- Download
 - <https://github.com/MICROSOFT/APPLICATIONINSPECTOR>

MICROSOFT APPLICATION INSPECTOR



Computer Security

- It is cross-platform and uses hundreds of feature detection patterns, covering many popular programming languages
- Supports identification of the following types of features
 - **Application frameworks:** development, testing...
 - **Cloud / Service APIs:** Microsoft Azure, Amazon AWS, and Google Cloud Platform
 - **Cryptography:** symmetric, asymmetric, hashing and TLS
 - **Types of data:** sensitive, personally identifiable information.....
 - **Operating System Functions:** platform identification, file system, registration and user accounts
 - **Security Features:** Authentication and Authorization

MICROSOFT APPLICATION INSPECTOR



- It has many modes of operation, but the main one is **"Analyze"**
 - Inspects source code in compressed directories / files (.tgz, .zip) looking for defined characteristics
- This is the analysis of the source code of **Angular v1.7.9**
 - The "features" category shows the different feature types used in the Angular code, classifying it
 - If we select a feature, the tool navigates to the source code where it think it is implemented
 - This makes it very easy to locate and inspect the behavior of the program
 - Specially on big programs whose code is unknown to us

Application Inspector Overview Summary Features About

Application Features

View key characteristics organized by Feature Groups. Click any active icon below or expand a feature group to view additional details. A disabled icon on the right to view where a specific feature was found. The full set of unique identified features can be found under the Summary menu.

Feature Groups			
— Select Features			
	Feature	Confidence	Details
	Authentication		View
	Authorization		View
	Cryptography		View
	Data deserialization		N/A
	AV media parsing		N/A
	Dynamic command execution		View

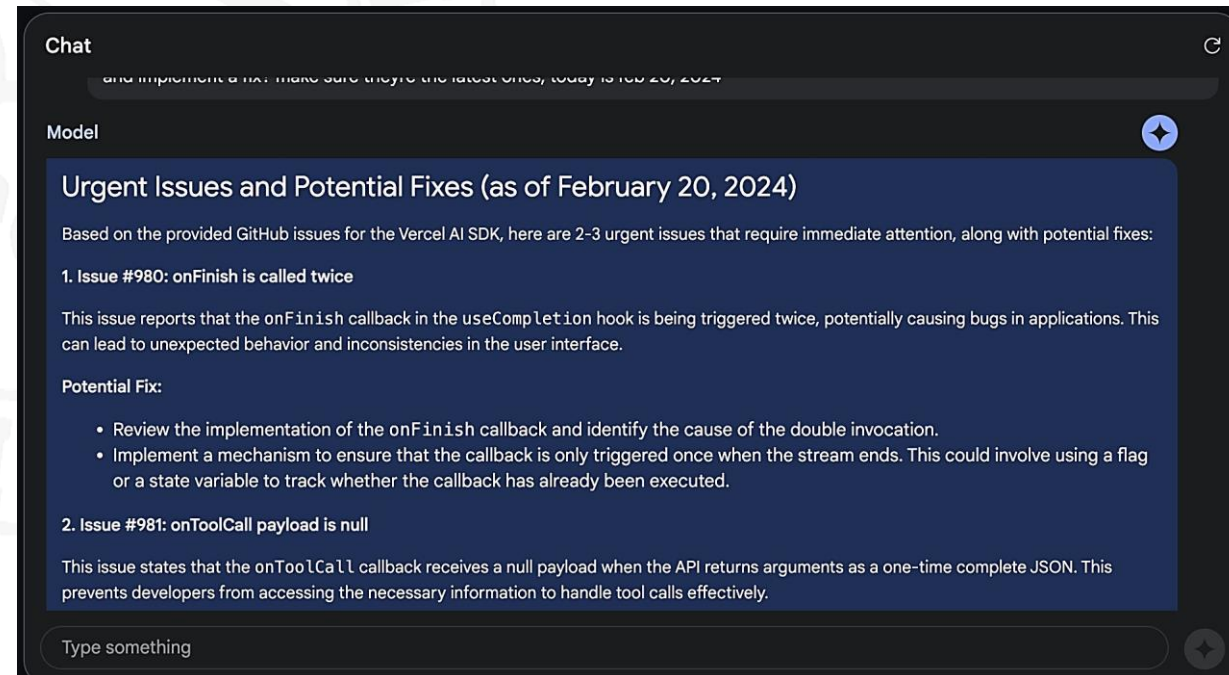
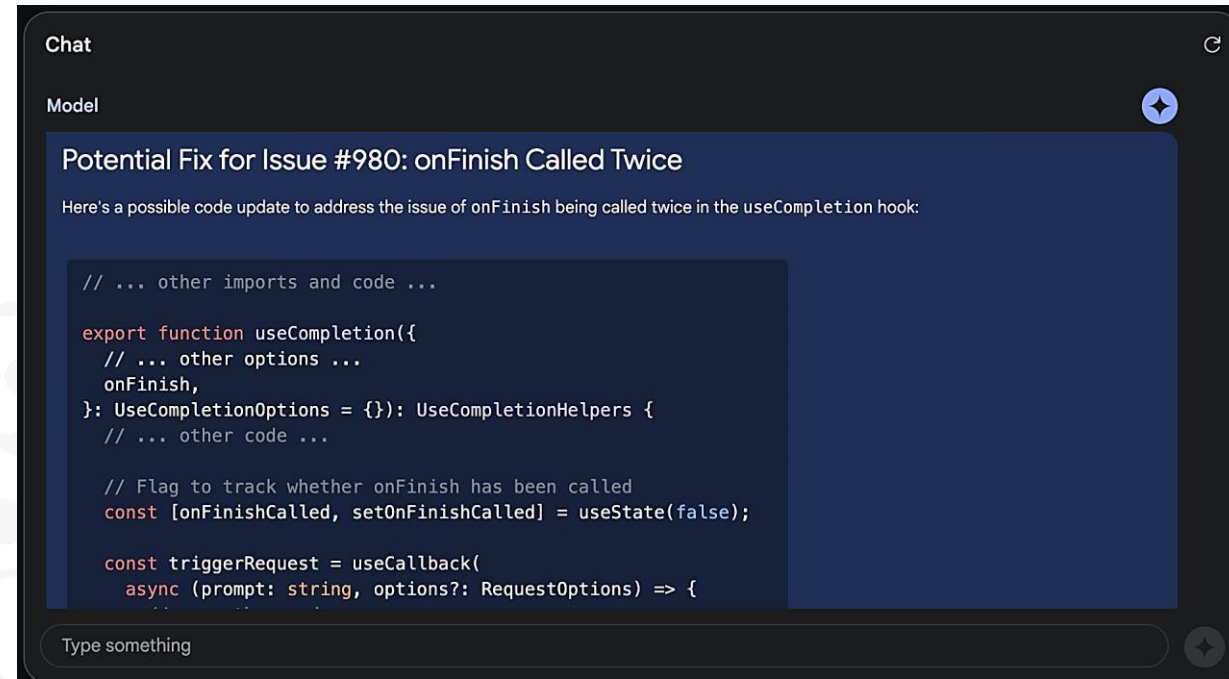
General Features			
	Feature	Confidence	Details
	Network communications		View
	File read		N/A
	File write		View
	File delete		N/A
	Multi-threaded processing		View

Associated Rules	
Name (click to view source)	
Network Connection: HTTP	
Network Connection: General	
Network Connection: FTP, Telnet or Similar	
Network Connection: HTTP Content (Ajax)	

AI-ASSISTED SAST



- This user has used AI to use it as an advanced SAST tool
 - <https://twitter.com/SullyOmarr/status/1760066335898513655>
 - This is Gemini 1.5 from Google
- The user uploaded an entire codebase directly from GitHub, and all the issues
- It able to understand the entire codebase
 - It identified the most urgent issue
 - And implemented a fix



SELF-EVALUATION ACHIEVEMENT LIST: SEMINAR 5

- Know the difference and the application of a whitelist and a blacklist
- Be able to perform appropriate type conversions and range checks
- Know how to use encoding libraries
- Use proper encoding libraries and HTML sanitizers
- Know how to choose the most appropriate security requirements
- Know the validation strategies adapted to each context
- Understand the validation mechanisms of modern web frameworks
- Know all the general aspects to create a secure code
- Know how to handle the functionality of uploading files to an application
- Know rules to prevent XSS attacks



SEMINAR 5

INPUT VALIDATION AND SECURE CODING TECHNIQUES

