More Web Security Considerations

Thomas L. "Trey" Jones, CISSP, CEH

Introduction

- A web application or web app is a client-server software application in which the client (or user interface) runs in a web browser.
- Uses database in the backend for storage of credentials and other information.
- Web applications are tricky
  - There is no way to understand the intention of the logged in user to determine whether he is a true user or a malicious user.
  - HTTP creates opportunities for security problems as it is a stateless protocol
  - Malicious users will try to use the application as a springboard for attacks against other targets.

INPUT AND OUTPUT VALIDATION FOR THE WEB

Input and output validation for the Web

- Perform the validation on the server; do not rely on the client
- Attackers may
  - change cookies, hidden fields, and post parameters.
  - post to URLs in the wrong order, at the wrong time, and for the wrong reasons.
  - For example, a typical HTTP User-Agent header might look something like this:
    - Mozilla/4.0 (compatible; MSIE 9.0; Windows NT 5.1; SV1)
    - The "User-Agent" header field contains information about the user agent originating the request, which is often used by servers to help identify the scope of reported interoperability problems, to work around or tailor responses to avoid particular user agent limitations, and for analytics regarding browser or operating system use.

Static Analysis: Audit cookies, Hidden Fields and HTTP Headers

- Use static analysis to identify places where the application uses these input sources and manually audit the ways they are used
- A tool will not be able to automatically discern the level of trust implicit in the way values from these sources are used, but it will do a very good job of making sure you review every instance throughout the application

Assume the Browser Is an Open Book

- Server-side input validation isn't the whole story.
- Attackers view, study, and find the patterns in every piece of information you send them, even if the information isn't rendered in the browser, so you must also be mindful of the data you reveal.
- All an attacker needs to do to examine static Web page content is select the Show Source option in the browser.
- Basic example is to show the message "Invalid Credentials" instead of "Username is incorrect" OR "Password is incorrect".

Assume the Browser Is an Open Book (Cont'd)

- Attackers will look for patterns in URLs, URL parameters, hidden fields and cookie values
- They can contemplate the meaning of your naming conventions, and they will reverse engineer your JavaScript
- Attacker will decipher the data transfer protocol and message formats that the client is trading with the server if the web application uses Asynchronous JavaScript and XML (Ajax).

HTTP Response Splitting

- An HTTP response splitting vulnerability allows an attacker to write data into an HTTP header.
- If the application allows input that contains CR (carriage return, also given by %0d or \r) and LF (line feed, given by %0a or \n) characters in the header, it can give the attacker the ability to
  - control the remaining headers and the body of the current response
  - craft an entirely new HTTP response
- It may lead to a number of potential attacks
  - Cross-user defacement
  - Web and browser cache poisoning
  - Cross site scripting
  - Page hijacking

HTTP Response Splitting (Cont'd)

- If an attacker submits a malicious string, such as Wiley Hacker\r\n\r\nHTTP/1.1 200 OK\r\n..., the HTTP response will be split into two responses of the following form:

```
HTTP/1.1 200 OK
Set-Cookie: author=Wiley Hacker


HTTP/1.1 200 OK
...
```

- Many Web browsers and Web proxies will mishandle a response that looks like this.

HTTP Response Splitting (Cont'd)

- An attacker might be able to use the vulnerability to do the following:
  - Provide malicious content to the victim browser. This is similar to a reflected cross-site scripting attack.
  - Confuse a Web proxy. This can result in the Web proxy sending the second HTTP response to a different user, or in the Web proxy sending a different user's data to the attacker.

Prevent HTTP Response Splitting

- Perform validation on the data before it gets included in the HTTP headers.
- Do not allow modification of header information
  - Carefully checking the input for the type and length
- Remove characters such as line feed (\r, or LF, and other variants) and new line (\n, CR, and other variants) in the HTTP header
- Ensure your web server and web scripting code are up to date.

Summary of Input and Output validation

- Do not trust the User-Agent header or any other data that come directly from the client
- Do not trust values stored in cookies or hidden fields
- Do not wager the security of your application on radio buttons, drop-downs, or JavaScript functioning properly
- Do not assume that the referrer header actually points to a referring page

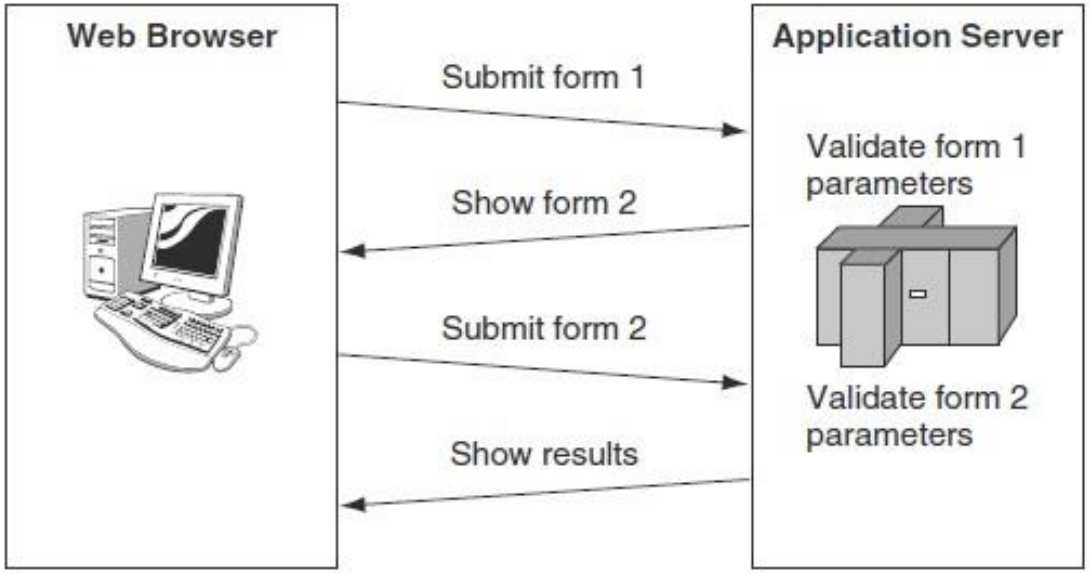
HTTP REQUEST ORDERING

Request Ordering

- HTTP is a stateless protocol.
- It does not provide any way to enforce the order of requests.
- Attackers can make requests in any order that is advantageous to them (not the order intended by the program logic).

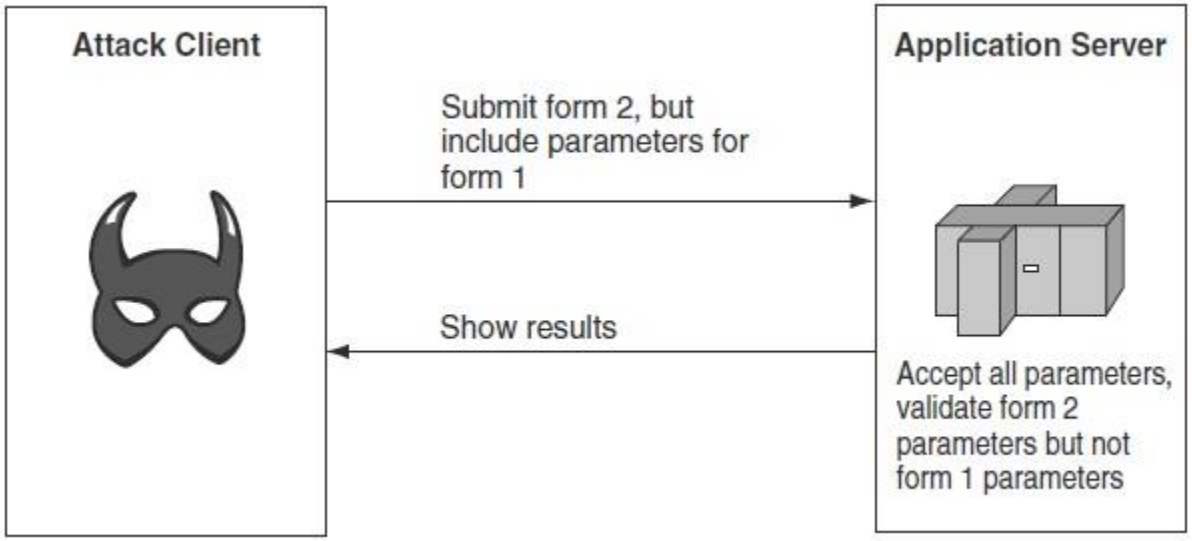
Request Ordering (Cont'd)

- The figure shows an expected request sequence.
  - The user submits a first form, and the application validates the form data and responds with a second form.
  - The user submits the second form, and the application responds with the results of the transaction.



Request Ordering (Cont'd)

- This image shows an attack.
  - The attacker submits the second form
  - But includes the parameters for the first form in order to bypass the validation for the first form.



MAINTAING SESSION STATES

Introduction to Maintaining Session States

- HTTP was not designed with applications in mind
- HTTP is stateless, building almost any sort of sophisticated application requires passing a *session identifier* back and forth to associate a user's previous requests with her next. Session identifiers can be passed back and forth as URL parameters, but today most applications handle them with cookies.
- The most common reason to use a session identifier is to allow a user to authenticate only once but carry on a series of interactions with the application. That means the security of the application depends on it being very difficult for an attacker to make use of the session identifier for an authenticated user.

Handling Session Identifiers

- Good HTTP session management means picking strong session identifiers and ensuring that they're issued and revoked at appropriate points in the program.
  - Select an application container that offers good session management facilities
  - Make sure the session identifier contains at least 128 bits of random data, to prevent attackers from hijacking users' session identifiers.
  - Enforce a maximum session idle time and a maximum session lifetime.
  - Make sure the user has a way to terminate the session.
  - Ensure that whenever a user is authenticated, the current session identifier is invalidated and a new one is issued.

Use Strong Session Identifiers

- Use session identifiers that include at least 128 bits of data generated by a cryptographically secure random number generator.
  - A shorter session identifier leaves the application open to brute-force session-guessing attacks
- The expected number of seconds required to guess a valid session identifier is given by this equation:

$$\frac{2^B + 1}{2 \times A \times S}$$

B is the number of digits/bits/characters of entropy in the session identifier.
A is the number of guesses an attacker can try each second.
S is the number of valid session identifiers that are valid and available to be guessed at any given time.

Why at least 128 bits?

- With a 64-bit session identifier, assume 32 bits of entropy. For a large Web site, assume that the attacker can try 10 guesses per second and that there are 10,000 valid session identifiers at any given moment. Given these assumptions, the expected time for an attacker to successfully guess a valid session identifier is less than 6 hours.
- Now assume a 128-bit session identifier that provides 64 bits of entropy. With a very large Web site, an attacker might try 10,000 guesses per second with 100,000 valid session identifiers available to be guessed. Given these somewhat extreme assumptions in favor of the attacker, the expected time to successfully guess a valid session identifier is greater than 292 years.

Use Strong Session Identifiers (continued..)

- No standardized approach exists for controlling the length of the session identifier used by a Servlet container. Example below shows how to control the session identifier length for BEA WebLogic.

```
<session-descriptor>
 <session-param>
   <param-name>IDLength</param-name>
   <param-value>25</param-value> <!--Specified in characters-->
 </session-param>
 ...
 </session-descriptor>
```

Use Strong Session Identifiers (continued..)

Static Analysis: Avoid Weak Session Identifiers

Use static analysis to identify programs configured to use weak session identifiers. The following rule will flag session identifiers configured to be less than 25 characters long in `weblogic.xml`:

Configuration rule:

```
File Pattern: weblogic.xml
XPath Expression: /weblogic-web-app/session-descriptor/
  session-param[normalize-space(param-name)= 'IDLength' and param-value < 25
```

Enforce a Session Idle Timeout and a Maximum Session Lifetime

- It limits the period of exposure for users who fail to invalidate their session by logging out.
- It reduces the average number of valid session identifiers available for an attacker to guess.
- It makes it impossible for an attacker to obtain a valid session identifier and then keep it alive indefinitely.

Enforce a Session Idle Timeout and a Maximum Session Lifetime (continued..)

- Session Idle Timeout
  - For any container that implements the Servlet specification, you can configure the session timeout in web.xml like this:

```
<session-config> <!-- argument specifies timeout in minutes -->
      <session-timeout>30</session-timeout>
</session-config>
```

Enforce a Session Idle Timeout and a Maximum Session Lifetime (continued..)

- Session Idle Timeout
  - You can also set the session timeout on an individual session using the setMaxInactiveInterval( ) method:

```
// Argument specifies idle timeout in seconds
session.setMaxInactiveInterval(1800);
```

Enforce a Session Idle Timeout and a Maximum Session Lifetime (continued..)

- Maximum Session Lifetime
  - The Servlet specification does not mandate a mechanism for setting a maximum session lifetime, and not all Servlet containers implement a proprietary mechanism. You can implement your own session lifetime limiter as a Servlet filter.
  - The doFilter() method below stashes the current time in a session the first time a request is made using the session.
  - If the session is still in use after the maximum session lifetime, the filter invalidates the session.

Enforce a Session Idle Timeout and a Maximum Session Lifetime (continued..)

```
public void doFilter(ServletRequest request,
                     ServletResponse response,
                     FilterChain chain)
                     throws IOException, ServletException {

   if (request instanceof HttpServletRequest) {
      HttpServletRequest hres = (HttpServletRequest) request;
      HttpSession sess = hres.getSession(false);
      if (sess != null) {
         long now = System.currentTimeMillis();
         long then = sess.getCreationTime();
         if ((now - then) > MAX_SESSION_LIFETIME) {
            sess.invalidate();

         }
      }
   }
   chain.doFilter(request, response);
}
```

Begin a New Session upon Authentication

- Always generate a new session when a user authenticates, even if an existing session identifier is already associated with the user.
- If the application does not generate a new session identifier whenever a user authenticates, the potential exists for a *session fixation* attack, in which the attacker forces a known session identifier onto a user.
  - an attacker creates a new session in a Web application without logging in and records the associated session identifier. The attacker then causes the victim to authenticate against the server using that session identifier, which results in the attacker gaining access to the user's account through the active session.

Begin a New Session upon Authentication (continued..)

- An attacking scenario
  1. The attacker walks up to a public terminal and navigates to the login page for a poorly built Web application. The application issues a session cookie as part of rendering the login page.
  2. The attacker records the session cookie and walks away from the terminal.
  3. A few minutes later, a victim approaches the terminal and logs in.
  4. Because the application continues to use the same session cookie it originally created for the attacker, the attacker now knows the victim's session identifier and can take control of the session from another computer.

Begin a New Session upon Authentication (continued..)

- This attack scenario requires several things for the attacker to have a chance at success:
  - Access to an unmonitored public terminal,
  - Capability to keep the compromised session active,
  - A victim interested in logging into the vulnerable application on the public terminal.

Begin a New Session upon Authentication (continued..)

- An attacker can do away with the need for a shared public terminal if the application server makes it possible to force a session identifier on a user by means of a link on a Web page or in an e-mail message.
- Apache Tomcat allows an attacker to specify a session identifier as a URL parameter like this: https://www.example.com/index.jsp?jsessionid=abc123. If the value of the jsessionid parameter refers to an existing session, Tomcat will begin using it as the session identifier.

Begin a New Session upon Authentication (continued..)

- To limit session fixation
  - A Web application must issue a new session identifier at the same time it authenticates a user.
- The Java Servlet specification requires a container to provide the URL j_security_check, but it does not require that the container issue a new session identifier when authentication succeeds. This leads to a vulnerability in the standard recommended method for setting up a login page, which involves creating a form that looks like this:

Begin a New Session upon Authentication (continued..)

- If the application has already created a session before the user authenticates, some implementations of j_security_check (including the one in Tomcat) will continue to use the already established session identifier. If that identifier were supplied by an attacker, the attacker would have access to the authenticated session.

```
<form method="POST" action="j_security_check" >
    Username: <input type="text" name="j_username">
    Password:<input type="password" name="j_password">
    <input type="submit" name="action" value="Log In">
</form>
```

Begin a New Session upon Authentication (continued..)

- Maintain state across an authentication boundary

```
public void doLogin(HttpServletRequest request) {
  HttpSession oldSession = request.getSession(false);
  if (oldSession != null) {
    // create new session if there was an old session
    oldSession.invalidate();
    HttpSession newSession = request.getSession(true);
    // transfer attributes from old to new
    Enumeration enum = oldSession.getAttributeNames();
    while (enum.hasMoreElements()) {
      String name = (String) enum.nextElement();
      Object obj = oldSession.getAttribute(name);
      newSession.setAttribute(name, obj);
    }
  }
  authenticate(request); // username/password checked here
}
```

Summary of Maintaining Session States

- If an attacker can learn a user's session identifier, the attacker can gain control of the user's session.
- Make sure that the session identifiers you use are long enough and random enough that attackers cannot guess them.
- To prevent an attacker from forcing a session identifier on a user, issue a new session identifier as part of the authentication process.
- Enforce a session idle timeout and a maximum session lifetime.

WORKING WITH XML

Issues Related to Handling XML (1/2)

- XML uses a Standards-Compliant XML Parser.
- The following bit of XML shows both deep nesting of name nodes and the strange appearance of a member node inside a group element.

  <member> <name> <name> <name> <name> ...
- A home-brew parser is more likely to use an excessive amount of memory or run into other unexpected performance bottlenecks when faced with documents that have an unusual shape

Issues Related to Handling XML (2/2)

- XML entities are another major source of problems
  - Examples: < and
- XML allows a document to define its own set of entities.
- Entities can kill your front end with recursion.
- It also greatly increases the chance that two different parsers will have slightly different interpretations of the same document.

Answer to the Parsing Problem

- Don’t underestimate the parsing problem.
- You’ll almost always want to reuse an existing parser that has a good track record for standards compliance and for security
- Validating against an XML schema (or even a DTD) is a good way to limit an attacker’s options
- Validation isn’t necessary to parse XML, but skipping the validation step gives an attacker increased opportunity to supply malicious input.

Turn on Validation (1/5)

- OrderXMLHandler will process the expected input just fine, but unexpected input is a different matter. When it processes the following order XML, the price of the book drops from almost $20 to just a nickel.

```
<order>
  <title>Magic tricks for all ages</title>
  <price>20.00</price>
  <shipTo><price>0.05</price>P. O. Box 510260 St. Louis, MO
63151-0260 USA</shipTo>
</order>
```

Turn on Validation (2/5)

- This vulnerability sometimes goes by the name XML injection because, presumably, the attacker has tricked some frontend system into generating an order document without validating the shipping address, thereby allowing the attacker to inject an unexpected XML tag.
- The simplest way to ensure that the order document has the form we expect is to validate it against a DTD.

Turn on Validation (3/5)

- Document Type Definition (DTD) Example:

```
<?xml version="1.0" encoding='UTF-8'?>
<!ELEMENT order (title, price, shipTo) >
<!ELEMENT title (#PCDATA) >
<!ELEMENT price (#PCDATA) >
<!ELEMENT shipTo (#PCDATA) >
```

- To enforce that the order XML matches the schema, two things need to change. First, the OrderXMLHandler class needs to enable validation. Second, it needs to stop processing if validation fails.

Turn on Validation (4/5)

- Simple XML schema for book order documents.

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" >
<xs:element name="order">
  <xs:complexType>
    <xs:all>
      <xs:element name="title‟ type="xs:string‟/>
      <xs:element name="price"  type="xs:string"/>
      <xs:element name="shipTo‟ type="xs:string"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

Turn on Validation (5/5)

- Example 10.3 A more rigorous XML schema for book order documents.

```
<xs:element name="order">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="title">
        <xs:simpleType>
          <xs:restriction base="xs:string">
            <xs:pattern value="[a-zA-Z0-9 '\-]*"/>
          </xs:restriction>
        </xs:simpleType>
      </xs:element>
      <xs:element name="price" type="xs:decimal"/>
      <xs:element name="shipTo">
        <xs:simpleType>
          <xs:restriction base="xs:string">
            <xs:pattern value="[a-zA-Z0-9 ,#\-\.\t\n]*"/>
          </xs:restriction>
        </xs:simpleType>
      </xs:element>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

Be Cautious About External References (1/2)

- DTDs, a document type declaration typically looks something like this:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/strict.dtd">
```

  - The last portion of the declaration, the part that reads http://www.w3.org/TR/xhtml1/DTD/strict.dtd, is called the system identifier.
  - It is a URI that both names the DTD and provides a location where the DTD can be found.

Be Cautious About External References (2/2)

- Do not trust external references that arrive cloaked in XML
- Attacker can change the parsed contents of the document as desired. This is called an XML External Entity (XXE) Attack.
- This is one of many reasons that it is preferable to use XML Schemas instead of DTDs.

Keep Control of Document Queries

- Grubbing around directly in XML tags can be tiresome. Languages such as XPath can take much of the tedium out of manipulating XML, but if attackers can control the contents of XPath query, they could end up with more access than you intended to give them.
- This attack is known as XPath injection, and both the attack and the vulnerable code constructs look quite similar to SQL Injection.

WORKING WITH REST APIS

REST API Security Top Vulnerabilities

- Broken Object Level Authorization (BOLA)
- Broken User Authentication
- Excessive Data Exposure
- Lack of Resource and Rate Limiting
- Broken Function Level Authorization
- Mass Assignment
- Security Misconfigurations
- Injections
- Improper Assets Management
- Insufficient Logging & Monitoring
- Information Disclosure
- Business Logic Vulnerabilities

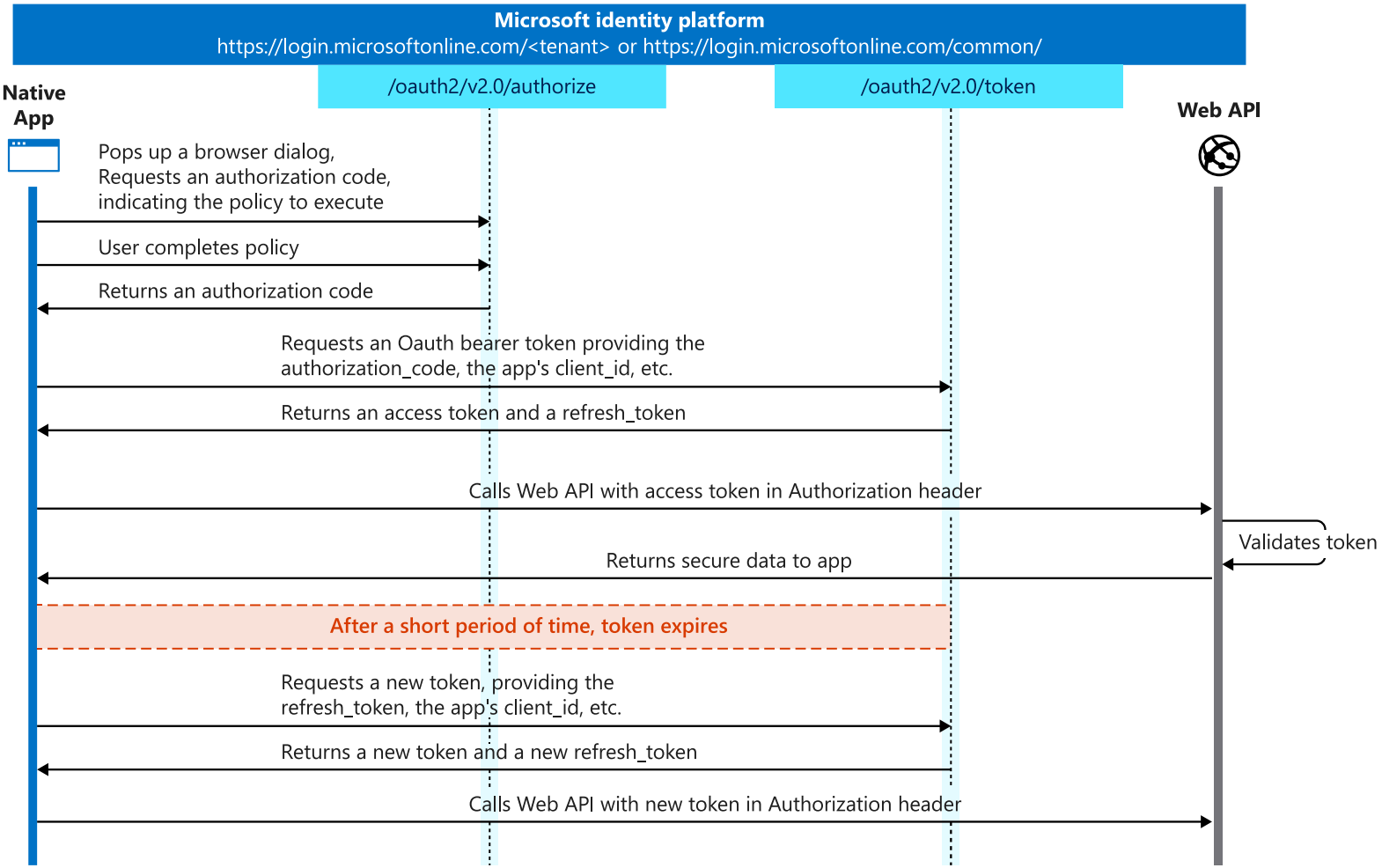
Broken Object Level Authorization (BOLA)

- Insufficient object access control checks leading to ability for caller to access resources/data belonging to others.
- How to Prevent:
  - Implement proper authorization mechanism relying on user policies and hierarchy.
  - Use authorization mechanism to check if logged-in user has access to perform requested action on the record in every function that uses input from the client to access a record in the database.
  - Prefer to use random and unpredictable values as GUIDs for records' IDs.
  - Write tests to evaluate the authorization mechanism. Do not deploy vulnerable changes that break the tests.

Broken User Authentication (1/2)

- Any aspect of the authentication solution that might be poorly implemented including generation, transmission, storage, and verification of tokens/certificates/secrets/etc.
- How to Prevent (partial list):
  - Make sure you know all the possible flows to authenticate to the API (mobile/ web/deep links that implement one-click authentication/etc.)
  - Read about your authentication mechanisms. Make sure you understand what and how they are used. OAuth is not authentication, and neither is API keys.
  - Don't reinvent the wheel in authentication, token generation, password storage. Use the standards.

OAuth 2.0 Flow



More Information: [Microsoft identity platform and OAuth 2.0 authorization code flow - Microsoft Entra | Microsoft Learn]

Broken User Authentication (2/2)

- How to Prevent (partial list continued):
  - Credential recovery/forget password endpoints should be treated as login endpoints in terms of brute force, rate limiting, and lockout protections.
  - Use the OWASP Authentication Cheatsheet.
  - Where possible, implement multi-factor authentication.
  - Implement anti brute force mechanisms to mitigate credential stuffing, dictionary attack, and brute force attacks on your authentication endpoints. This mechanism should be stricter than the regular rate limiting mechanism on your API.
  - Implement account lockout / captcha mechanism to prevent brute force against specific users. Implement weak-password checks.
  - API keys should not be used for user authentication, but for client app / project authentication.

Excessive Data Exposure (1/2)

- API returns more data than is expected or needed with expectation the client will filter to what is needed.
- How to Prevent:
  - Never rely on client side to filter sensitive data.
  - Review responses from the API to ensure they contain only legitimate data.
  - Backend engineers should always ask themselves "who is the consumer of the data?" before exposing a new API endpoint.
  - Avoid using generic methods such as `to_json()` and `to_string()`. Instead, cherry-pick specific properties you really want to return
  - Classify sensitive and personally identifiable information (PII) that your application stores and works with, reviewing all API calls returning such information to see if these responses pose a security issue.

Excessive Data Exposure (2/2)

- How to Prevent (continued):
  - Implement a schema-based response validation mechanism as an extra layer of security. As part of this mechanism define and enforce data returned by all API methods, including errors.

Lack of Resource and Rate Limiting (1/2)

- Missing or poorly implemented throttling of calls leading to exhaustion of available resources ([Distributed] Denial of Service Attack).
  - Execution timeouts
  - Max allocable memory
  - Number of file descriptors
  - Number of processes
  - Request payload size (e.g., uploads)
  - Number of requests per client/resource
  - Number of records per page to return in a single request response

Lack of Resource and Rate Limiting (2/2)

- How to Prevent:
  - Docker makes it easy to limit memory, CPU, number of restarts, file descriptors, and processes.
  - Implement a limit on how often a client can call the API within a defined timeframe.
  - Notify the client when the limit is exceeded by providing the limit number and the time at which the limit will be reset.
  - Add proper server-side validation for query string and request body parameters, specifically the one that controls the number of records to be returned in the response.
  - Define and enforce maximum size of data on all incoming parameters and payloads such as maximum length for strings and maximum number of elements in arrays.

Broken Function Level Authorization (1/2)

- A user in one role being able to use functionality that is supposed to be limited to users in another role.
  - Can a regular user access administrative endpoints?
  - Can a user perform sensitive actions (e.g., creation, modification, or erasure) that they should not have access to by simply changing the HTTP method (e.g., from GET to DELETE)?
  - Can a user from group X access a function that should be exposed only to users from group Y, by simply guessing the endpoint URL and parameters (e.g., /api/v1/users/export_all)?

Broken Function Level Authorization (2/2)

- How to Prevent:
  - Your application should have a consistent and easy to analyze authorization module that is invoked from all your business functions. Frequently, such protection is provided by one or more components external to the application code.
    - The enforcement mechanism(s) should deny all access by default, requiring explicit grants to specific roles for access to every function.
    - Review your API endpoints against function level authorization flaws, while keeping in mind the business logic of the application and groups hierarchy.
    - Make sure that all your administrative controllers inherit from an administrative abstract controller that implements authorization checks based on the user's group/role.
    - Make sure that administrative functions inside a regular controller implements authorization checks based on the user's group and role.

Mass Assignment (1/2)

- Request includes more parameter assignments than expected, leading to overwriting internal state data that wasn't intended to be modified by caller.
- An API endpoint is vulnerable if it automatically converts client parameters into internal object properties, without considering the sensitivity and the exposure level of these properties. This could allow an attacker to update object properties that they should not have access to.

Mass Assignment (2/2)

- Examples for sensitive properties:
  - Permission-related properties: `user.is_admin`, `user.is_vip` should only be set by admins.
  - Process-dependent properties: `user.cash` should only be set internally after payment verification.
  - Internal properties: `article.created_time` should only be set internally by the application.
- How to Prevent:
  - If possible, avoid using functions that automatically bind client's input into code variables or internal objects.
  - Whitelist only properties that should be updated by the client.
  - Use built-in features to blacklist properties that should not be accessed by clients.
  - If applicable, explicitly define and enforce schemas for the input data payloads.

Security Misconfigurations (1/2)

- Lack of security patches and misconfiguration of technology stack (e.g. missing or insecure encryption in transit or at rest, misconfiguring headers).
  - Appropriate security hardening is missing across any part of the application stack, or if it has improperly configured permissions on cloud services.
  - The latest security patches are missing, or systems are out of date.
  - Unnecessary features are enabled (e.g., HTTP verbs).
  - Transport Layer Security (TLS) is missing.
  - Security directives are not sent to clients (e.g., Security Headers).
  - A Cross-Origin Resource Sharing (CORS) policy is missing or improperly set.
  - Error messages include stack traces, or other sensitive information is exposed.

Security Misconfigurations (2/2)

- How to Prevent:
  - The API life cycle should include:
    - A repeatable hardening process leading to fast and easy deployment of a properly locked down environment.
    - A task to review and update configurations across the entire API stack. The review should include: orchestration files, API components, and cloud services (e.g., S3 bucket permissions).
    - A secure communication channel for all API interactions access to static assets (e.g., images).
    - An automated process to continuously assess the effectiveness of the configuration and settings in all environments.

Injections (1/2)

- Includes various attacks such as SQL/NoSQL injection, XSS, etc.
- The API is vulnerable to injection flaws if:
  - Client-supplied data is not validated, filtered, or sanitized by the API.
  - Client-supplied data is directly used or concatenated to SQL/NoSQL/LDAP queries, OS commands, XML parsers, and Object Relational Mapping (ORM)/Object Document Mapper (ODM).
  - Data coming from external systems (e.g., integrated systems) is not validated, filtered, or sanitized by the API.

Injections (2/2)

- How to Prevent:
  - Preventing injection requires keeping data separate from commands and queries.
    - Perform data validation using a single, trustworthy, and actively maintained library.
    - Validate, filter, and sanitize all client-provided data, or other data coming from integrated systems.
    - Special characters should be escaped using the specific syntax for the target interpreter.
    - Prefer a safe API that provides a parameterized interface.
    - Always limit the number of returned records to prevent mass disclosure in case of injection.
    - Validate incoming data using sufficient filters to only allow valid values for each input parameter.
    - Define data types and strict patterns for all string parameters.

Improper Assets Management (1/3)

- The API might be vulnerable if:
  - The purpose of an API host is unclear, and there are no explicit answers to the following questions:
    - Which environment is the API running in (e.g., production, staging, test, development)?
    - Who should have network access to the API (e.g., public, internal, partners)?
    - Which API version is running?
    - What data is gathered and processed by the API (e.g., PII)?
    - What's the data flow?
  - There is no documentation, or the existing documentation is not updated.
  - There is no retirement plan for each API version.
  - Hosts inventory is missing or outdated.
  - Integrated services inventory, either first- or third-party, is missing or outdated.
  - Old or previous API versions are running unpatched.

Improper Assets Management (2/3)

- How to Prevent:
  - Produce and Maintain Documentation (either automated or manual):
    - Inventory all API hosts and document important aspects of each one of them, focusing on the API environment (e.g., production, staging, test, development), who should have network access to the host (e.g., public, internal, partners) and the API version.
    - Inventory integrated services and document important aspects such as their role in the system, what data is exchanged (data flow), and its sensitivity.
    - Document all aspects of your API such as authentication, errors, redirects, rate limiting, cross-origin resource sharing (CORS) policy and endpoints, including their parameters, requests, and responses.
    - Generate documentation automatically by adopting open standards. Include the documentation build in your CI/CD pipeline.
    - Make API documentation available to those authorized to use the API.
  - Use external protection measures such as API security firewalls for all exposed versions of your APIs, not just for the current production version.

Improper Assets Management (2/2)

- How to Prevent (cont):
  - Avoid using production data with non-production API deployments. If this is unavoidable, these endpoints should get the same security treatment as the production ones.
  - When newer versions of APIs include security improvements, perform risk analysis to make the decision of the mitigation actions required for the older version: for example, whether it is possible to backport the improvements without breaking API compatibility or you need to take the older version out quickly and force all clients to move to the latest version.

Insufficient Logging & Monitoring (1/2)

- The API is vulnerable if:
  - It does not produce any logs, the logging level is not set correctly, or log messages do not include enough detail.
  - Log integrity is not guaranteed (e.g., Log Injection).
  - Logs are not continuously monitored.
  - API infrastructure is not continuously monitored.

Insufficient Logging & Monitoring (2/2)

- How to Prevent:
  - Log all failed authentication attempts, denied access, and input validation errors.
  - Logs should be written using a format suited to be consumed by a log management solution, and should include enough detail to identify the malicious actor.
  - Logs should be handled as sensitive data, and their integrity should be guaranteed at rest and transit.
  - Configure a monitoring system to continuously monitor the infrastructure, network, and the API functioning.
  - Use a Security Information and Event Management (SIEM) system to aggregate and manage logs from all components of the API stack and hosts.
  - Configure custom dashboards and alerts, enabling suspicious activities to be detected and responded to earlier.

Additional Common API Vulnerabilities

- Information Disclosure
  - Via API responses or public sources (e.g. search results, source code repos, social media, web sites, etc.)
  - Verbose messaging.
- Business Logic Vulnerabilities
  - Any number of vulnerabilities in the code that could be exploited such as improper validation of uploaded file payloads.
  - Incorrectly assuming caller will use APIs as intended.
  - Callers may lose credentials or have them stolen, leading to an adversary ("wolf in sheep's clothing") masquerading as a trusted client and misusing the API.

References

- Book: “Hacking APIs” by Corey J. Ball, 2022, No Starch Press.
- Web Site: OWASP API Security Project | OWASP Foundation