How to BEST Use This Material...

We're thrilled to be able to release a portion of our Application Security training material to the community! In doing so, we're cognizant that the content within is unable to meet all team needs, skill levels, or contexts (e.g. software languages).

To make the most out of this content, we'd offer a few tips:

1) Take what you like and leave the rest behind -- there's no one-size-fits-all
2) Transfer the content valuable to your organization into more familiar branding
3) Replace the merit of examples shown with ones more akin to your tech stack
4) Share internal examples of previous security issues to increase the impact
5) Throw in details about your security program and application security team
6) Mix-and-match content from other sources to build a training for *your* needs

Available Lab Resources

It's **highly** encouraged that as part of your curriculum, hands-on labs are given across the training to help attendees gain a direct, technical understanding of topics being shared. This will maximize their engagement and increase learning.

Below are a list of potential lab resources that may be beneficial to consider:

- Hunter2 Community (Hosts Duo's own lab module contributions for free)
- OWASP Juice Shop extremely popular vulnerable-by-design web service
- OWASP WebGoat provides a lesson-based capability along with hacking
- Google Gruyere is an older set of challenges but still widely applicable today
- flAWS.cloud & flAWS2.cloud focused on AWS-centric security challenges

Additional Training Tips

- Create a feedback form that is filled out by your attendees each offering
- Be sure to provide many breaks during the day
- Small (~15) class sizes
- Set “Ground Rules” so students are not getting distracted from the class
- Handle IT setup logistics prior to the class date
- Take questions regularly

Advanced Application Security

Application Security Team, Duo Security

Duo Security is now part of Cisco.

Cryptography

Duo Security is
now part of Cisco.

Cryptography Can Be Difficult

- So many options, so little time
  - NIST standards: AES, DES
  - CBC, ECB, CTR block types
  - Authenticated encryption
- The “mystery” around cryptography, especially for newer developers
- Just “rolling your own” can be tempting (but dangerous)

Crypto Issue #1

Security Expertise

- Assumption that developers have necessary security knowledge
  - Different developers, different experience
- Knowing the difference between regulatory-compliance and effective data protection
- What to protect & when to protect it
  - Encrypted at rest
  - Encrypted during transmission
- Standards like FIPS 140-2 can help
  - Security standard used to approve cryptographic modules

Crypto Issue #2

Poor Key Management

- No centralized source for key storage
- Insufficient or non-existent key rotation policies
- Allowing “weak keys” to be used
- Key reuse
- Ensuring key is available when requested
- Insider threats or accidental key exposure

Crypto Issue #3

Third-Party Protection

- Using encryption to protect data living on non-encrypted, third-party systems (think S3)
- Relying on third-party to protect data correctly
  - Be sure to confirm their standards and policies
- Hardware security module (HSM) for key storage
  - For example, Amazon offers HSM as a service - CloudHSM

Cryptography Tips for Developers

- Avoid key reuse
- Only generate keys using a cryptographically strong random source
- Use only known and widely used standard algorithms
- When possible, make use of authenticated encryption over non-authenticated
- Define robust key storage mechanisms
- Plan a key rotation policy

Content Security Policies

What are Content Security Policies?

- Security control designed to prevent injection (like XSS) and external code execution attacks
- “Agreement” between the client and the server
  - Supported in most major browsers
- Configured via a standard HTTP header: `Content-Security-Policy`

What are Content Security Policies?

- Allows control over several aspects of the content
  - Use of inline CSS and Javascript
  - Control of dynamic Javascript code execution (eval)
  - Using dynamic CSS statements
  - Form actions
  - Iframe sources
- Allows definition of the “origin” of content
  - `self`
  - domain(s) including wildcard support

Content Security Policy Examples

```
Content-Security-Policy: default-src 'self'
*.trusted.com
```

```
Content-Security-Policy: default-src 'self'; img-
src *; script-src https://userscripts.example.com
```

Content Security Policy Examples

```
Content-Security-Policy: default-src 'self';
script-src 'self' https://www.google-
analytics.com; img-src 'self' https://www.google-
analytics.com
```

Content Security Policy Features

- Provides a reporting feature to log violations
  - `report-uri`
  - Sent as a POST request to the defined endpoint
  - Includes URI of requested resource, policy that caused the violation, portion of script causing the issue, section of the policy violated
- Allows for “reporting only” versus enforcement via `Content-Security-Policy-Report-Only` header
- While most major browsers support policies, Internet Explorer (and Edge) support many less directives

Content Security Policy Tips

- Older browsers may require the `X-Content-Security-Policy` header
- Some frameworks include a CSP by default, also easy to inject via middleware
- The `nonce` option allows for even further lockdown

Client-Side Injection

What is Client-Side Injection?

- Injection of malicious code or content via data stored on the client-side only
- Data sources include:
  - Local SQLite databases on mobile devices
  - Localstorage in browsers
  - Content from other locations in the page (forms)
- Most often caused by missing or broken input validation and poor output escaping

Client-Side Injection Example: Javascript

```
<script>
function append() {
        var input = document.getElementById(‘input’).value;
        var output = document.getElementById(‘output’);
        output.innerHTML = output.innerHTML + string;
}
</script>

<html>
        <body>
                <form>
                        <input type=”text” id=”input” value=””>
                        <button type=”submit” onclick=”append()”>Submit</button>
                </form>
    </body>
</html>
```

Client-Side Injection Example: CSS

```
<a id="input">Click me</a>
<script>
if (location.hash.slice(1)) {
     document.getElementById("input").style.cssText = "color: " +
     location.hash.slice(1);
}
</script>
```

For http://mysite.com/#red:

```
<style>
 #input{
 color: red;
 }
</style>
```

---

https://www.url/#red; @import "https://externalsource/navigation.css”

Client-Side Injection Example: Redirect

```
<script>
var redir = location.hash.substring(1);
if (redir)
    window.location='http://'+decodeURIComponent(redir);
</script>
```

Valid: http://url/?#www.url/login

Malicious: http://url/?#www.attackerurl/phishing-page

Client-Side Injection Example: Cookies

```
Set-Cookie: sessionid=38afes7a8; HttpOnly; Path=/

Set-Cookie: id=a3fWa; Expires=Wed, 21 Oct 2015 07:28:00 GMT; Secure;
HttpOnly

Set-Cookie: qwerty=219ffwef9w0f; Domain=somecompany.co.uk; Path=/;
Expires=Wed, 30 Aug 2019 00:00:00 GMT
```

Cookies are completely user controlled!

Mitigations

- Use well-vetted templating libraries, don't roll your own
- Avoid the use of "workarounds" in templating (ex: outputting raw data)
- Avoid the direct use of user controllable input in client-side resources
- When querying local databases, ensure the use of prepared statements/parameterized queries

Broken Authentication Controls

Duo Security is now part of Cisco.

What are authentication controls?

- Control types
  - Single-factor
  - Multi-factor
  - Continuous
- Common authentication controls
  - Credential-based (identifier)
  - Token based (JSON tokens)
  - Out-of-band methods
  - Third-party services
  - Origin of request

Planning

- Base authentication controls around risk
  - Risks involved with information disclosure
  - Risks of data tampering or loss of integrity
- Design and document based on risk assessment
  - Provides a unified perspective on what's covered and where
  - Threat modeling
- Completely implement and update control
  - Improper control usage
  - Failure to update all locations when making changes could result in logic differences
- Enforce strong standards for credential/identifier storage

Common Flaws: Technology

Poor login protection

- No rate limiting
- Additional controls not implemented after X number of failures
- No authentication on sensitive resources!

No quality enforcement

- Permission policy not in place/weak
- Passwords not evaluated for reuse (i.e. haveibeenpwned)

Poor session state handling

- Session not recycled on permission change
- Session IDs not validated or transmitted in an insecure way

Common Flaws:
Human

- Phishing
- Vishing (Voice)
- Social Engineering
- Poor credential handling habits

Good security education is key!

Authentication Control Testing

- Test common weaknesses
  - Unencrypted credential transmission
  - Default credentials and other “insecure by default” settings
  - Weak secondary controls (remember me, password reset, etc)
  - Bypass via “forceful browsing” or parameter manipulation
- Testing with more than one type of user
  - Determine major access control differences (make a matrix)
  - Static files with predictable filenames and available via direct web request
- Automated testing via tooling

Authentication Control Testing

Key:
Text = Should they be able to do it?
Color=Could they do it?
Green=Positive
Red=Negative

Role	Update Passwords	Update Email	Change Account Data	Upgrade Account to Admin	View Logs
Admin	Yes	Yes	Yes	Yes	Yes
User	Yes	Yes	No	No	No
Unauthenticated	No	No	No	No	No

Broken Authorization Controls

Broken Authorization Controls

- AuthN proves identity, AuthZ proves what they can do
- Multiple levels of controls can be more effective
  - Split by user level (ex: normal user vs admin)
  - Split by permission set
  - Split by access control list of users
    Issues with consistent data access across application
- Multiple controls can add to overall complexity
  - Potential for bypass via partially implemented controls

What are Authorization Controls?

- Control types
  - Discretionary - Based on user identity and “has” or “in” relationship, owns own data
  - Mandatory - Based on protection requirements of resource (must match)
- Common authorization controls
  - Access list
  - Permissions (has, has not)
  - Permissions grouping (roles)
  - Other: restrictions on location, time of day, system accessing from

Common Flaws: Technology

- Direct Object Reference
  - Changing values in the URL
  - Changing values in `POST/PUT/DELETE` request body content
  - ID, GUID, hash
- Missing function-level access control
  - “Forceful browsing” to locate misconfigurations
  - Static files with predictable filenames and resources available via direct web request
- Logic Flaws
  - Beginning a workflow in the wrong place
  - Bypass of controls by forceful browsing to other resources rather than the one directed to

Common Flaws: Technology

XML External Entity Injection (XXE)

What is XML External Entity Injection?

- Entities are objects in XML documents
  - Defined in the base XML specification
  - Acts as a reusable value
  - Replaced as a part of parsing the XML document
- Injection caused by malicious entity definition
  - User-provided content injected directly into the document
  - XML parser misconfigured to expand entities
- Multiple potential vulnerabilities
  - Accessing local files
  - Loading external resources

How does XXE happen?

1.
User input is allowed into the XML content.

2.
This content is passed into a parser for evaluation and extraction.

3.
Due to misconfiguration, the parser expands the entities by default.

4.
The malicious user XML content is then activated as a part of processing.

5.
The malicious XML then executes the attack, potentially causing:

1. Denial of service
2. Information leakage
3. Code execution

XXE Example #1: Billion Laughs

```
<?xml version="1.0"?>
<!DOCTYPE lolz [
 <!ENTITY lol "lol">
 <!ELEMENT lolz (#PCDATA)>
 <!ENTITY lol1 "&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;">
 <!ENTITY lol2 "&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;">
 <!ENTITY lol3 "&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;">
 <!ENTITY lol4 "&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;">
 ........
]>
<lolz>&lol100;</lolz>
```

XXE Example #2: File Injection

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE foo [ <!ELEMENT foo ANY >
<!ENTITY xxe SYSTEM "file:///etc/passwd" >]>
<creds>
    <user>&xxe;</user>
    <pass>mypass</pass>
</creds>
```

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE foo [ <!ELEMENT foo ANY >
<!ENTITY xxe SYSTEM "http://192.168.0.1/secret.txt" >]>
<creds>
    <user>&xxe;</user>
    <pass>mypass</pass>
</creds>
```

XXE Example #3: Code Execution

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE foo [ <!ELEMENT foo ANY >
<!ENTITY xxe SYSTEM "expect://id" >]>
<creds>
    <user>&xxe;</user>
    <pass>mypass</pass>
</creds>
```

“expect” is a PHP extension allows interaction with processes through PTY.

Mitigation

- Disable auto-entity expansion on the parser
  - Wide range of different parsers for many different languages, each with its own method
  - Ensure the setting is the default
  - Access
- If access to the entities is required, use parser controls
  - Prevent access to external entities
  - Take extra care when using these values
- Don’t allow unvalidated user input into your XML
  - Ensure that user input is valid and only contains what you expect
  - If there is a requirement, escape it correctly for the XML context

Lack of Sufficient Logging & Monitoring

Duo Security is now part of Cisco.

Lack of Sufficient Logging & Monitoring

- You can’t detect what you can’t measure
  - Provides context into the use and abuse of your environment
  - Logging based on failures is important, but sometimes successes are too
  - Real-time detection is crucial to response times and approach
- Logging must be analyzed
  - Useless without analysis
- Multiple benefits of good logging practices
  - Easier to locate information in case of an investigation
  - Potentially simpler identification of the origin of an attack
  - Log evaluation post-incident can provide additional context to correct similar issues

Security Logging Issues

Difficult to detect anomalies

- Tooling can help (machine learning, thresholds, pattern matching)
- No replacement for humans

Log Overload

- Too many logs to be overly useful without heavy evaluation

Monitoring alerts

- May require “threat hunters”
- False positives can eclipse actual problems

Security Logging Recommendations

- Plan on what to log and how to log it
  - Log on purpose, “everything” is usually too much
  - Define high risk areas that may require more detailed logs
- Define the important indicators
  - Not all log information is useful
- Ensure the logging and monitoring solution is sustainable
  - Will the new feature or system you’re adding greatly increase the log content?
  - Does it measure up to the continuing requirements of your organization?
  - Can it easily be integrated with centralized logging?

Security Logging Recommendations

Logging locations

Access level changes (login, assume user role, logout)

Higher-risk data changes

Details about secondary controls (forgot password, 2FA)

Files added or accessed

Authentication failures

Authorization failures

User input validation failures

Session management issues

Changes in permissioning

Third-party errors

Exceptions

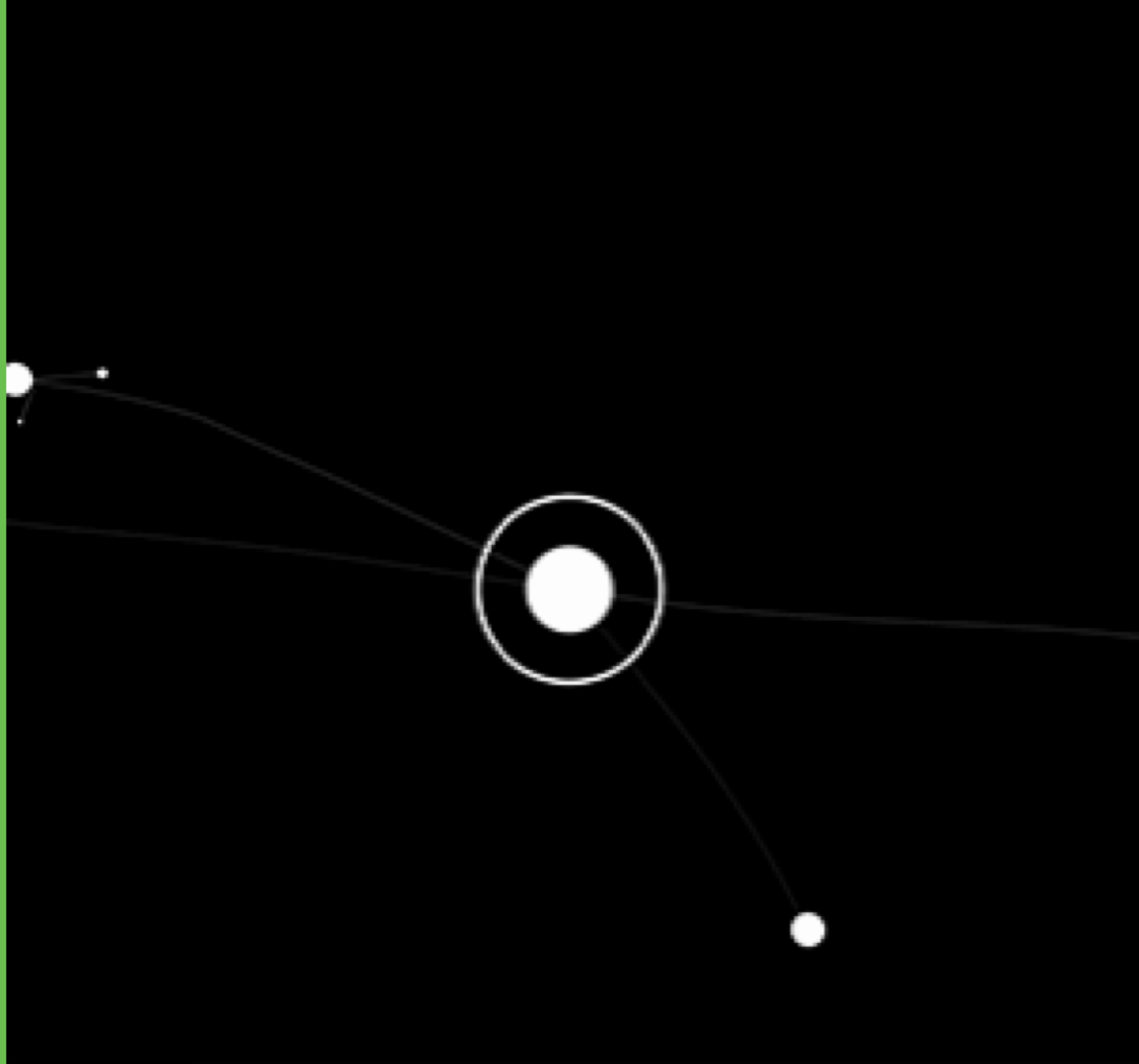
Using Vulnerable Components

What are Vulnerable Dependencies?

- Any piece of software (component) that has known vulnerabilities, published or unpublished
- Vulnerability sources
  - Public disclosure via independent researcher
  - Disclosure by the project itself
  - Results of automated scanning (e.g. Black Duck, SourceClear)
  - Internally known issues

Why is this so important?

- 90% of all applications make use of third-party components
- Dependencies of dependencies of dependencies...



A Flavor for Every Language

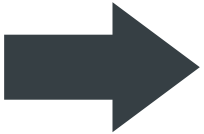
- Python packages
- Ruby gems
- NPM packages (Node.js)
- Microsoft .NET packages
- WordPress plugins
- Composer packages (PHP)
- Maven (Java)

...and many,
many more.

What are the risks?

- Risk depends on the functionality the component provides
  - Templating libraries: XSS injection
  - Database library: SQL injection
  - Encryption library: Weak or incorrect cipher use
  - Authentication library: Bypass of controls
- Integration of one or more components could also increase the risk

What are the risks?



Disaster Rank	OWASP Top 10	# of Breaches Root Cause	% of Breaches Root Cause
1	A9	12	24.%
2	A5	10	18%
3	A1	4	8%
4	A2	3	6%
4	A6	3	6%
5	A7	2	4%
6	A4	1	2%
--	A3	0	0%
--	A8	0	0%
--	A10	0	0%
	Other	15	30%
	Unknown	32	

snyk

Breach Examples by Vulnerable Components

- VerticalScope/Techsupportforum.com breach
  - 45 million passwords and IP addresses were stolen from a network of over 1,100 websites and forums
  - Caused by vulnerability in an old version of vBulletin (SQL injection)
- Ubuntu forums breach
  - 2 million usernames, IP addresses and passwords were compromised
  - Caused by a vulnerability in the Forumrunner component (SQL injection)

Path to Upgrade

Mitigations

- **Rely on the project to fix the issues**
  - Lack of control over when the fix happens
  - Quality of the fix might be questionable depending on project developers
- **Fork and fix the issues yourself**
  - Time to fix greatly increases
  - Requires extensive in-house knowledge of the component
  - Puts a maintenance burden on the team to keep up with future releases (and patch those if needed)

HTTP Header Injection

What is HTTP Header Injection?

- User-defined content is included in the HTTP headers of the request/response
  - Source could be from application data
  - Also could be from related information (such as referrer URI)
- Injection of a newline (CRLF) then adding a custom header
  - Also called “HTTP Request Splitting” where headers are “split” with additional newline
- Potential vulnerabilities depend on how header information is used in application
  - Any value that can be controlled via headers
  - Most languages support “last in wins” parsing of multiple headers

HTTP Header Injection Example

URL: http://mysite.com/user?username=testuser1

```
GET /user?username=testuser1 HTTP/1.1
Host: mysite.com
X-Username: testuser1

username=testuser1&id=1
```

HTTP Header Injection Example

URL: http://mysite.com/user?username=%0d%0a%0d%0aLocation:http://google.com

```
GET /user?username=%0d%0a%0d%0aLocation:http://google.com HTTP/1.1
Host: mysite.com
X-Username:
Location: http://google.com

username=testuser1&id=1
```

Impactful Headers

Some headers are more useful than others

- `Set-Cookie`
- `Access-Control-Allow-Origin`
- `Location`
- `X-XSS-Protection`
- `Strict-Transport-Security`
- `X-Frame-Options`
- `X-Content-Type-Options`
- `Content-Security-Policy`

Mitigations

- Prevent the use of user-definable values in your headers as much as possible
  - Sometimes unavoidable (as in the case of `Referer`)
- If user input is required, ensure that all newlines are correctly escaped
  - `\n` encodes to `%0A`
  - `\r` encodes to `%0D`
- Use language level functions to prevent injection
  - PHP includes checks on inputs to the header() function for newlines
  - Flask (Python web framework) includes newline checks in functions setting headers

JSON Injection

What is JSON Injection?

- Injection of user-defined values into a JSON document or data
- On its own, could potentially be used to overwrite existing data or corrupt the JSON structure
- Side effects
  - Corrupted JSON could trigger an exception, leading to information exposure
  - Poor JSON parsing could lead to malicious values being used in other locations (such as XSS in an item's value)
  - Potential of additional false information being appended to the document, possibly allowing for an auth bypass if used in logic.

What does JSON Injection look like?

```
{
   "username":"user1",
   "password":"$2y$10$VBOVO3XJHT7.jwykGyWMk.4Pvg7qgEba.YLcU3xBV7Vp9IqzRswcO",
   "name":"Chris",
   "email":"me@example.com"
}
```

What does JSON Injection look like?

```
{
   "username":"user1",
   "password":"$2y$10$VBOVO3XJHT7.jwykGyWMk.4Pvg7qgEba.YLcU3xBV7Vp9IqzRswcO",
   "name":"Chris", "password":"known-string",
   "email":"me@example.com"
}
```

Duplicate key names could do funny things but might just throw a warning from the parser.

What could cause this?

- Malicious user input making its way into the data (unverified)
- Poor parsing and creation practices
- Poor output escaping or direct use of values without sanitization

Mitigations

- Use as little user input in JSON as possible, validating the input to ensure correctness
- Use only well-established and vetted parsing libraries and tools.
- Avoid the direct use of values pulled from JSON, escaping according to the correct context

Replay Attacks

What are Replay Attacks?

- Repeating the same content in an effort to reproduce the effect
- Could be performed by valid user or attacker
- Sometimes captured as a part of a Man-in-the-Middle attack
- Could be sent to the same location or a different location that accepts the same data
- Not just restricted to web-based requests

Why do they work?

- No uniqueness in the session or request
- No detection of changes in the session (like IP address)
- Plain-text messages not protected in transit
- Does not require knowledge of the payload context but impact is difficult to determine without it

How to prevent the attack

- Include uniqueness into the request
  - Header with dynamic value verified on the server
  - Time-bound session identifiers rotated frequently
  - One-time protection (such as a password)
- Reducing risk by limiting window a request could be valid
- Preventing interception with encryption in transit
- HTTPOnly on (session) cookies can prevent access

Secure JSON Web Tokens

Duo Security is now part of Cisco.

JSON Web Tokens (JWT)

- Based on a standard: RFC 7519
- Provides more context than a basic session identifier
- Relies on:
  - JWS - JSON Web Signature
  - JWE - JSON Web Encryption
- Signed to ensure integrity of the token's contents
- Often sent as a `Bearer` header for reauthorization on following requests

Building JWT

- Three sections:
  - Header defines type of token and algorithm to use for the signature
  - Claims are the details (default and custom)
  - Signature (None, HMAC, RSA)
- All sections are defined as JSON structures
- Each section is (URL safe) base64 encoded and combined with “.”

```
base64urlEncoding(header)+'.'+base64urlEncoding(payload)+'.'+base64urlEncoding(signature)
```

JWT Claims

Default claims

- `iss`: Issuer
- `sub`: Subject
- `aud`: Audience
- `exp`: Expiration
- `nbf`: “Not before”
- `iat`: Issued At
- `iti`: JWT ID

Custom claims

- No sensitive data! Remember, these can be decoded

JWT Example

Encoded PASTE A TOKEN HERE

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.SflKxwRJSMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c
```

Decoded EDIT THE PAYLOAD AND SECRET

HEADER: ALGORITHM & TOKEN TYPE

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

PAYLOAD: DATA

```
{
  "sub": "1234567890",
  "name": "John Doe",
  "iat": 1516239022
}
```

VERIFY SIGNATURE

```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  your-256-bit-secret
) ☐ secret base64 encoded
```

Common JWT Issues

- Sensitive information in the claims
- Individual tokens cannot be easily revoked (until timeout defined by claim)
  - By user or by service
- “Stale” values if the token is not regenerated when data is updated
- Intended to be stateless, using them as stateful is basically a glorified session token
- Using the `None` algorithm
- Not verifying key length before signature generation

Want to Learn More?

- **Defense in Depth:** https://gist.github.com/maxvt/bb49a6c7243163b8120625fc8ae3f3cd
- **Developer's Guide to Encryption:** https://crypto.stanford.edu/cs142/papers/web-session-management.pdf
- **Broken Access Control:** https://www.owasp.org/index.php/Top_10-2017_A5-Broken_Access_Control
- **Mobile Top 10 - Client Side Injection:** https://www.owasp.org/index.php/Mobile_Top_10_2012-M4_Client_Side_Injection
- **HTTP Header Injection:** https://dzone.com/articles/crlf-injection-and-http-response-splitting-vulnera
- **JSON injection:** https://blog.qualys.com/technology/2016/08/03/testing-ajax-applications-with-json-input-for-vulnerabilities-using-qualys-was
- **Client-side reflected JSON injection:** https://portswigger.net/kb/issues/00200371_client-side-json-injection-reflected-dom-based
- **OWASP Logging Cheatsheet:** https://github.com/OWASP/CheatSheetSeries/blob/master/cheatsheets/Logging_Cheat_Sheet.md
- **Security Logging Best Practices:** https://reciprocitylabs.com/audit-log-best-practices-for-information-security/
- **OWASP XXE Prevention Cheatsheet:** https://cheatsheetseries.owasp.org/cheatsheets/XML_External_Entity_Prevention_Cheat_Sheet.html