API Penetration Testing

Course Overview

Alexis Ahmed

Offensive Security/Red Team Instructor @INE
Red Team Lead @HackerSploit

Key Concepts

- Introduction to Web Services & APIs
- Introduction to Web API Security
- API Reconnaissance
- API Authentication Testing
- API Injection Vulnerabilities

MAJOR TOPICS

- Fundamentals of web services and APIs
- Web API Security
- API Pentesting Methodology
- API Reconnaissance
- API Authentication Testing
- API Injection Vulnerabilities

LEARNING OUTCOMES

- Understand the purpose and structure of APIs, including REST, SOAP, and GraphQL.
- Recognize common API security risks and their impact.
- Apply a systematic methodology for API penetration testing.
- Perform reconnaissance to discover API endpoints and resources.
- Test API authentication mechanisms for vulnerabilities.
- Identify and exploit API injection vulnerabilities.

PREREQUISITES

- Familiarity with the OWASP Top 10 & OWASP WSTG
- Experience in using web proxies like Burp & ZAP

LET'S GO!

Introduction To Web Services

What Are Web Services?

- Web services are software components designed to facilitate communication and data exchange between different applications or systems over the internet.

- They allow disparate applications to work together, even if they are developed on different platforms, using different programming languages, or running on different servers.

What Are Web Services?

- They are usually intended to facilitate:
  - Integration between applications: Application ‘A’ uses features implemented in application ‘B’.
  - Separation within an application: Front-end scripts that use web service functionality to dynamically update the content.

Web Services vs Web Applications

Web Services:

- Web services are designed to facilitate communication and data exchange between different software systems over the internet.
- They provide a standardized way for different applications to interact with each other, often using protocols like SOAP (Simple Object Access Protocol) or REST (Representational State Transfer).
- Web services are typically used for machine-to-machine communication and are not meant for direct human interaction.

Web Services vs Web Applications

Web Applications:

- Web applications are software programs that are accessed through a web browser and are designed to perform specific tasks or provide services directly to end-users.
- They are meant for human interaction and can range from simple websites to complex web-based applications like email clients, social media platforms, or online shopping sites.

Web Services vs Web Applications

Aspect	Web Services	Web Applications
Purpose	Facilitate data exchange between applications.	Provide services or perform tasks directly for end-users.
User Interaction	No user interface; meant for machine-to-machine communication.	Has a user-friendly interface for human interaction.
Data Exchange	Exchanges structured data between applications.	Involves user data input, processing, and result presentation.
Communication Protocol	Uses protocols like SOAP, REST, XML-RPC, or JSON-RPC.	Primarily uses HTTP/HTTPS for communication.
Security Focus	Focuses on securing data during transmission and access control.	Broader security aspects, including authentication, authorization, data validation, and protection against web vulnerabilities.
Examples	Payment gateways (e.g., PayPal API), weather data services.	Online banking, e-commerce (Amazon), email (Gmail), social networking (Facebook).

Key Characteristics of Web Services

- Interoperability: Web services promote interoperability by providing a standardized way for applications to communicate. They rely on open standards like HTTP, XML, SOAP, REST, and JSON to ensure compatibility.

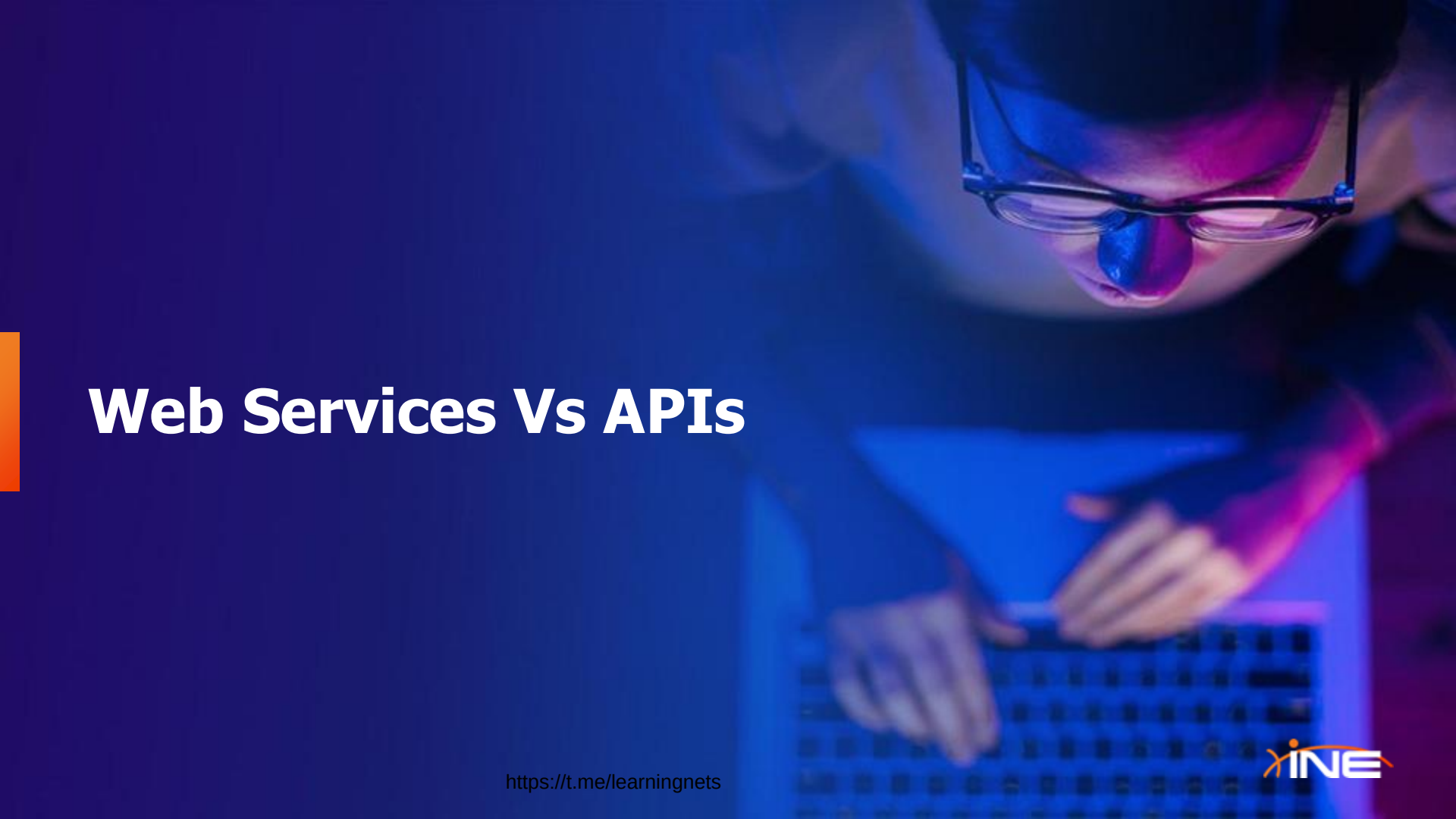
- Platform-agnostic: Web services are not tied to a specific operating system or programming language. They can be developed in various technologies, making them versatile and accessible.

Key Characteristics of Web Services

- Loose Coupling: Web services allow for loosely coupled interactions between systems. This means that changes in one system's implementation do not necessarily disrupt the functionality of other systems.

- Location Independence: Web services operate over the internet, making them location-independent. They can be hosted on different servers and accessed from anywhere with an internet connection.

Web Services Vs APIs



Web Services Vs APIs

- Web services and APIs (Application Programming Interfaces) are related concepts in web development, but they have distinct differences.

- Web services are a broader category of technologies used to enable **machine-to-machine** communication and data exchange over the internet. They encompass various protocols and data formats. APIs, on the other hand, are a set of rules and tools that allow **developers** to access the functionality or data of a service, application, or platform.

Web Services Vs APIs

- **Web services** are a broad category of technologies and protocols designed to facilitate communication and data exchange between different software systems over the internet. They are meant to provide a standardized way for various applications, often on different platforms and using different programming languages, to interact with each other.

- **APIs (Application Programming Interfaces)**, on the other hand, refer to a set of rules, protocols, and tools that allow different software applications to communicate with each other. They provide access to the functionality or data of an application or service for developers to use in their own applications.

Web Services vs APIs

Aspect	Web Services	APIs (Application Programming Interfaces)
Purpose	Facilitate data exchange between software systems.	Provide access to the functionality or data of an application or service for developers.
Communication Protocol	Can use various protocols, including SOAP, REST, XML-RPC, JSON-RPC, and more.	Can use various protocols, often associated with RESTful APIs but not limited to them.
Data Format	Use multiple data formats, such as XML and JSON.	Can work with different data formats, including JSON, XML, or custom formats.
Interface	Do not have a user interface for direct human interaction.	Do not have a user interface but are used by developers to build applications with user interfaces.
Scope	A subset of APIs, specifically focused on web-based data exchange.	A broader concept encompassing various types of interfaces for software interaction.
Standards	WS-Security, WS-Policy for SOAP-based web services.	OpenAPI (formerly Swagger) for documenting RESTful APIs, GraphQL for querying APIs.

Web Service Implementations

Web Service Implementations

- Web service implementations refer to the different ways in which web services can be created, deployed, and used.
- There are several methods and technologies available for implementing web services. We will be exploring them in the next set of slides.

Web Service Implementations

- SOAP (Simple Object Access Protocol): SOAP is a protocol for exchanging structured information in the implementation of web services. SOAP-based web services use XML as their message format and can be implemented using various programming languages.
- JSON-RPC and XML-RPC: JSON-RPC and XML-RPC are lightweight protocols for remote procedure calls (RPC) using JSON or XML, respectively. These are simpler alternatives to SOAP for implementing web services.
- REST (Representational State Transfer): REST is an architectural style for designing networked applications, and it uses HTTP as its communication protocol.

XML-RPC

- XML-RPC (Extensible Markup Language - Remote Procedure Call) created in 1998, is a protocol and a set of conventions for encoding and decoding data in XML format and using it for remote procedure calls (RPC).
- It is a simple and lightweight protocol for enabling communication between software applications running on different systems, often over a network like the internet.
- XML-RPC has been used as a precursor to more modern web service protocols like SOAP and REST.
- It works by sending HTTP requests that call a single method implemented on the remote system.

XML-RPC - Request Example

```
POST /xml_rpc/web_service.php HTTP/1.1
User-Agent: Zend_XmlRpc_Client
Host: wp.site
Content-Type: text/xml
Content-length: 179
…

<?xml version="1.0"?>
<methodCall>
   <methodName>My.Method</methodName>
   <params>
      <param>
         <value>www.google.com</value>
      </param>
   </params>
</methodCall>
```

Request headers such as user agent, hosts, content type etc.

Payload format in XML.

Must contain <methodCall> with the <methodName> sub-item.

XML-RPC encodes data in XML format, which is both human-readable and machine-readable. It uses XML tags to represent data types and method calls.

XML-RPC - Requests & Responses

```
<?xml version="1.0"?>
<methodCall>
  <methodName>sampleMethod</methodName>
  <params>
    <param>
      <value><int>42</int></value>
    </param>
  </params>
</methodCall>
```

REQUEST

```
<?xml version="1.0"?>
<methodResponse>
  <params>
    <param>
      <value><string>Hello, World!</string></value>
    </param>
  </params>
</methodResponse>
```

RESPONSE

JSON-RPC

- JSON-RPC (Remote Procedure Call) is a remote procedure call (RPC) protocol encoded in JSON (JavaScript Object Notation).
- Like XML-RPC, JSON-RPC enables communication between software components or systems running on different machines or platforms.
- JSON-RPC is known for its simplicity and ease of use and has become popular in web development and microservices architectures.
- JSON-RPC is very similar to XML-RPC, however, it is usually used because it provides much more human-readable messages and takes less data to for communication.
- JSON-RPC allows a client to invoke methods or functions on a remote server by sending a JSON object that specifies the method to call and its parameters.

JSON-RPC

- The message sent to invoke a method is a request with a single object serialized using JSON. It has three properties:
  - method: name of the method to invoke
  - params: an array of objects to pass as arguments
  - id: request ID used to match the responses/requests

JSON-RPC - Request Example

```
POST /json_rpc/web_service.php HTTP/1.1
User-Agent: my_user_agent
Host: wp.site
Content-Type: application/json-rpc
Content-length: 57
…

{"method": "MyMethod", "params": ["www.google.com"], "id": 1}
```

Request Headers like User-Agent, Host etc

This is a single object serialized using JSON. Note the three properties: method, params and id.

SOAP

- SOAP (Simple Object Access Protocol) is a protocol for exchanging structured information in the implementation of web services.
- It is a protocol that defines a set of rules and conventions for structuring messages, defining remote procedure calls (RPC), and handling communication between software components over a network, typically the internet.
- SOAP is seen as the natural successor to XML-RPC and is known for its strong typing and extensive feature set, which includes security, reliability, and transaction support.
- SOAP Web Services may also provide a Web Services Definition language (WSDL) declaration that specifies how they may be used or interacted with.

SOAP - Request Example

```
POST /soap_rpc/web_service.php HTTP/1.1
User-Agent: PHP-SOAP
Host: wp.site
Content-Type: text/soap+xml
Content-length: 179
…

<?xml version="1.0"?>
<soap:Envelope
xmlns:soap="http://www.w3.org/2003/05/soap-envelope"
soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">
<soap:Body xmlns:m="http://ws.site/soap_ws/ws">
  <m:MyMethod>
    <m:Host>www.google.com</m:Host>
  </m:MyMethod>
</soap:Body>
</soap:Envelope>
```

Request headers such as user agent, hosts, content type etc.

Body of the request (XML)

SOAP - Requests & Responses

```
<soapenv:Envelope
xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
                  xmlns:web="http://www.example.com/webservice">
   <soapenv:Header/>
   <soapenv:Body>
      <web:sampleMethod>
         <web:inputParameter>42</web:inputParameter>
      </web:sampleMethod>
   </soapenv:Body>
</soapenv:Envelope>
```

REQUEST

```
<soapenv:Envelope
xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
                   xmlns:web="http://www.example.com/webservice">
   <soapenv:Header/>
   <soapenv:Body>
      <web:sampleMethodResponse>
         <web:result>Hello, World!</web:result>
      </web:sampleMethodResponse>
   </soapenv:Body>
</soapenv:Envelope>
```

RESPONSE

REST (RESTful APIs)

- REST, which stands for Representational State Transfer, is an architectural style for designing networked applications. It is not a protocol or technology itself but rather a set of principles and constraints that guide the design of web services and APIs (Application Programming Interfaces).
- REST is widely used for building scalable, stateless, and easy-to-maintain web services/APIs that can be accessed over the internet.
- REST web services generally use JSON or XML, but any other message transport format like plain-text can be used.

REST (RESTful APIs)

HTTP Method	Action
GET	Retrieve a resource on the server, list a collection of records • http://ws.site/book (e.g. List all books available) •http://ws.site/book/1 (e.g. View a specific book)
PUT	Change the state of a resource, replace or create it if it does not exist •http://ws.site/book/1 (e.g. Replace a book)
POST	Create a new resource or record •http://ws.site/book (e.g. Create a specific book)
DELETE	Delete a resource or record •http://ws.site/book/1 (e.g. Delete a book)

WSDL Language Fundamentals

Introduction

A web service is characterized by:

One or more Methods

Each reflects a service provided by the server application

A Protocol

It defines:

- The structure of each message used to request a service
- The structure of a message sent by the web service in response
- The transport method used to transmit the messages

WSDL

- WSDL, which stands for Web Services Description Language, is an XML-based language used to describe the functionality and interface of a web service.

- WSDL documents serve as contracts between service providers and consumers, specifying how a web service can be used.

- WSDL is commonly used in conjunction with SOAP (Simple Object Access Protocol) to define and document SOAP-based web services.

WSDL Versions

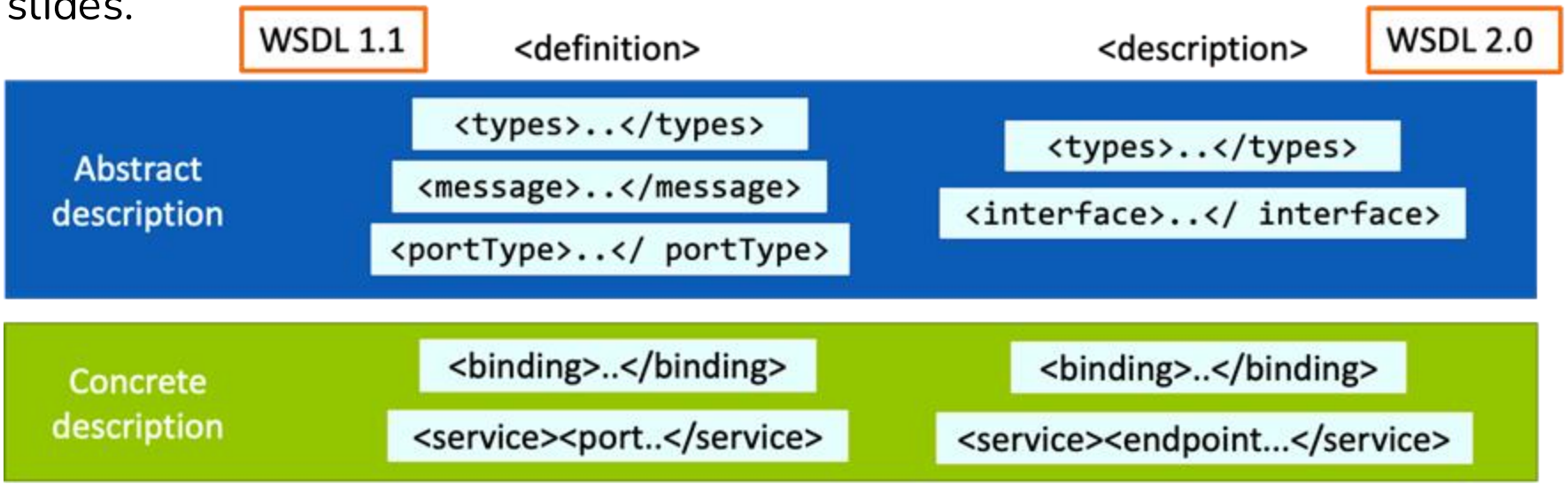
- At the time of writing, WSDL can be distinguished in two main versions: 1.1 and 2.0.
- Although 2.0 is the current version, many web services still use WSDL 1.1 therefore, in the next slides we will see both WSDL specifications.

WSDL

- First of all, it is important to know that WSDL documents have abstract and concrete definitions:
    - Abstract: describes what the service does, such as the operation provided, the input, the output and the fault messages used by each operation
    - Concrete: adds information about how the web service communicates and where the functionality is offered

WSDL

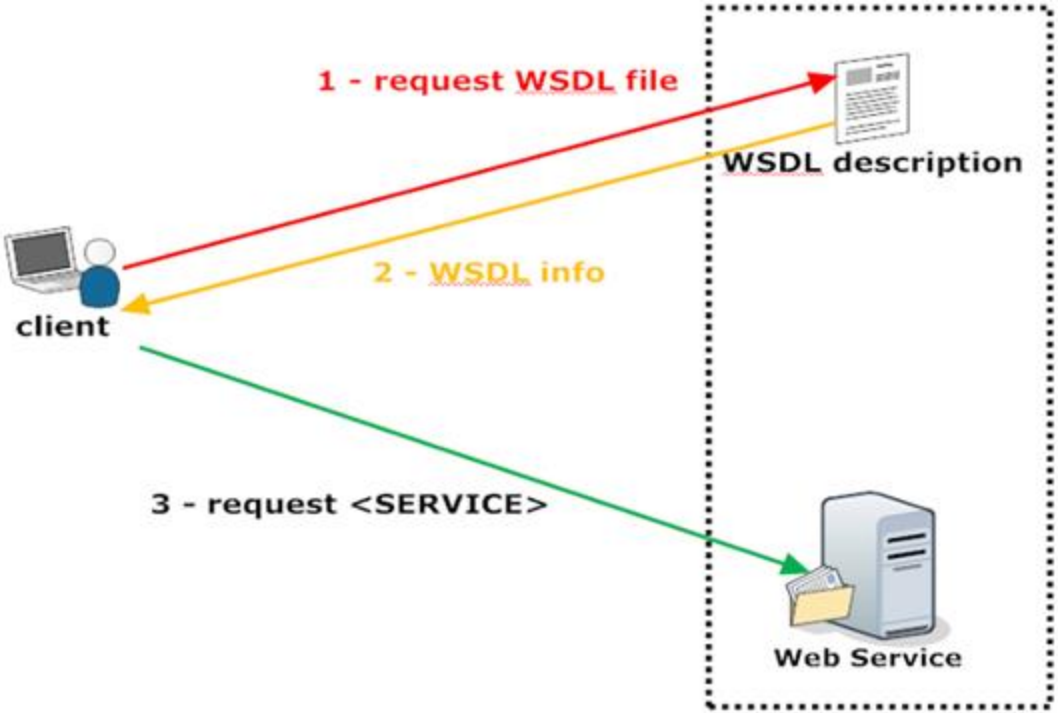
The following image shows the main differences between WSDL 1.1 and WSDL 2.0 definitions. We will inspect the most important elements in future slides.



WSDL Documents

- A WSDL document is typically created to describe a SOAP-based web service. It defines the service's operations, their input and output message structures, and how they are bound to the SOAP protocol.
- The WSDL document effectively documents the API provided by the service.
- The WSDL document serves as a contract between the service provider and consumers. It specifies how clients should construct SOAP requests to interact with the service. This contract defines the operations, their input parameters, and expected responses.

Interaction Between Client & Web Service



WSDL Components

- Types: The <types> section defines the data types used in the web service. It typically includes XML Schema Definitions (XSD) that specify the structure and constraints of input and output data.
- Message: The <message> element defines the data structures used in the messages exchanged between the client and the service. Messages can have multiple parts, each with a name and a type definition referencing the types defined in the <types> section.
- Port Type: The <portType> element describes the operations that the web service supports. Each operation corresponds to a method or function that a client can invoke. It specifies the input and output messages for each operation.

WSDL Components

- Binding: The <binding> element specifies how the service operations are bound to a particular protocol, such as SOAP over HTTP. It defines details like the protocol, message encoding, and endpoint addresses.

- Service: The <service> element provides information about the service itself. It includes the service's name and its endpoint address, which is the URL where clients can access the service.

Binding

- The <binding> element specifies how the service operations are bound to a particular protocol, such as SOAP over HTTP. It defines details like the protocol, message encoding, and endpoint addresses.

```
<wsdl:binding name="HelloServiceSoap11Binding"
    type="ns:HelloServicePortType">
    <soap:binding transport="http://schemas.xml
            soap.org/soap/http" style="document"/>
    < . . [WSDL OPERATIONS] . . />
</wsdl:binding>
```

PortType

- The <portType> element describes the operations that the web service supports. Each operation corresponds to a method or function that a client can invoke. It specifies the input and output messages for each operation.

```
<wsdl:portType name="HelloServicePortType">
      <wsdl:operation name="sayHello">
            <wsdl:input message="ns:sayHelloRequest"/>
            <wsdl:output message="ns:sayHelloResponse"/>
      </wsdl:operation>
</wsdl:portType>
```

Operation

- The operation object defined within a port type, represents a specific action that a service can perform. It specifies the name of the operation, the input message structure, the output message structure, and, optionally, fault messages that can occur during the operation.

```
<wsdl:operation name="sayHello">
      <soap:operation soapAction="sayHello" style="document"/>
      <wsdl:input>
            <soap:body use="literal"/>
      </wsdl:input>
      <wsdl:output>
            <soap:body use="literal"/>
      </wsdl:output>
</wsdl:operation>
```

Interface

- Instead of portType, WSDL v. 2.0 uses interface elements which define a set of operations representing an interaction between the client and the service. Each operation specifies the types of messages that the service can send or receive.

- Unlike the old portType, interface elements do not point to messages anymore (it does not exist in v. 2.0). Instead, they point to the schema elements contained within the types element.

Types of Authentication Mechanisms

Authentication Mechanisms

- Authentication mechanisms are methods or processes used to verify the identity of a user or system attempting to access a web application or service.
- These mechanisms ensure that only authorized users can gain access to sensitive resources, enhancing security.
- In the next couple of slides we will explore some of the key types of authentication mechanisms used in web applications.

Types of Authentication Mechanisms

Password-Based Authentication

- Users provide a username and a password to verify their identity.

Multi-Factor Authentication (MFA)

- Combines two or more independent credentials, such as:
  - Something you know (password or PIN).
  - Something you have (smartphone, security token).
  - Something you are (biometric verification like fingerprint or facial recognition).

Types of Authentication Mechanisms

Two-Factor Authentication (2FA)

- A specific type of MFA that requires exactly two factors for authentication, often a password and a one-time code sent via SMS or an authenticator app.

Token-Based Authentication

- Uses tokens (e.g., JSON Web Tokens or OAuth tokens) that are issued upon successful login and are used for subsequent requests, reducing the need to repeatedly enter credentials.

Types of Authentication Mechanisms

Single Sign-On (SSO)

- Allows users to log in once and gain access to multiple applications or services without needing to re-enter credentials, often using protocols like SAML or OAuth.

One-Time Passwords (OTP)

- A temporary password sent to the user via SMS or email for a single login session, often used in conjunction with other authentication methods.

Authentication Testing Methodology

Authentication Testing

- Authentication testing is the process of probing and exploiting weaknesses in a web application's identity verification mechanisms.
- This involves testing various authentication controls, like login forms, password reset functionality, multi-factor authentication, and account lockouts, to discover any vulnerabilities that could allow unauthorized access or account compromise.
- Authentication testing targets flaws that may permit bypassing of login controls, such as weak password policies, susceptibility to brute force or credential stuffing attacks, session fixation, and inadequate token handling.

Authentication Testing

- The objective is to identify and exploit weaknesses in authentication to gain unauthorized access, elevate privileges, or hijack legitimate user sessions, thus demonstrating the real-world impact of compromised authentication security.

OWASP WSTG

- For web application penetration testers, the OWASP WSTG serves as both a training resource and a methodological guide, particularly in the critical area of authentication testing.

- By providing standardized, comprehensive guidance on testing steps, it enables testers to conduct effective, consistent, and impactful assessments of authentication controls and other security areas within web applications.

🔗 OWASP WSTG: https://owasp.org/www-project-web-security-testing-guide/

OWASP WSTG - Authentication Tests

Test Name	ID	Description
Testing for Credentials Transported over an Encrypted Channel	WSTG-ATHN-01	Verifies that user credentials are transmitted securely over HTTPS to prevent interception or tampering.
Testing for Default Credentials	WSTG-ATHN-02	Checks if any default credentials are still in use, which attackers could exploit to gain unauthorized access.
Testing for Weak Lock Out Mechanism	WSTG-ATHN-03	Assesses the application's lockout mechanisms to prevent brute-force attacks on user accounts.
Testing for Bypassing Authentication Schema	WSTG-ATHN-04	Identifies flaws in the authentication process that allow attackers to bypass authentication altogether.
Testing for Vulnerable Remember Password Function	WSTG-ATHN-05	Evaluates if the "Remember Me" functionality is implemented securely, without exposing sensitive data that could aid in unauthorized access.

OWASP WSTG - Authentication Tests

Test Name	ID	Description
Testing for Browser Cache Weaknesses	WSTG-ATHN-06	Ensures sensitive information is not stored insecurely in the browser cache, where attackers could retrieve it.
Testing for Weak Password Policy	WSTG-ATHN-07	Examines the application's password policy to determine if it enforces sufficient password complexity and expiration requirements.
Testing for Weak Authentication in Alternative Channels	WSTG-ATHN-08	Tests alternative authentication channels (e.g., APIs, mobile applications) for weaker or inconsistent security practices that could lead to account compromise.

Introduction to Token-Based Authentication

Token-Based Authentication

- Token-based authentication is a modern method used to securely validate and authorize users or applications in web environments.
- Instead of maintaining session state on the server, tokens are issued to clients and passed back and forth to authenticate requests.
- These tokens encapsulate user or application identity and relevant permissions, enabling seamless and scalable authentication.

Types of Tokens

Bearer Tokens

- A simple token format that grants access to resources when presented.
- Usage: Often used in APIs; the server assumes the bearer of the token has the authority to access the resource.

```
GET /user/profile HTTP/1.1
Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...
```

Security Considerations:

- Must be kept secret; easily exploitable if intercepted.
- Typically short-lived to minimize the risk of misuse.

Types of Tokens

JSON Web Tokens (JWT)

- A self-contained token format that includes a header, payload (claims), and signature.
- Usage: Widely used in modern web applications for stateless authentication.
- Advantages: Portable and stateless, allowing servers to offload session management.

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwI
iwibmFtZSI6IkpvZSIsImlhdCI6MTUxNjIzOTAyMn0.SflKxwRJSMeKKF2QT4f
wpMeJf36POk6yJV_adQssw5c
```

Types of Tokens

oAuth Tokens

- Tokens used in the OAuth 2.0 protocol to authorize applications or users to access resources.
- Types of Tokens:
  - Access Tokens: Grant permission to access resources.
  - Refresh Tokens: Obtain new access tokens without re-authentication.

- Example Flow:
  - User authenticates with an OAuth provider (e.g., Google).
  - Access token is issued to the client application.
  - Client uses the token to request resources from the server.

Token Placement

Authorization Header

- Usage: The most common and secure method.
- Why: Keeps tokens separate from the request body or URL, reducing exposure risks.

```
GET /api/v1/resource HTTP/1.1
Host: example.com
Authorization: Bearer <token>
```

Benefits:

- Works seamlessly with APIs and browsers.
- Prevents token leakage through logs (as tokens aren't in the URL).

Query Parameters

- Usage: Tokens are appended as part of the URL.

```
GET /api/v1/resource?access_token=<token> HTTP/1.1
Host: example.com
```

Risks:

- Tokens might get exposed through browser history, logs, or referral headers.
- Vulnerable to being cached by proxies or servers.

Request Body

- Usage: Used primarily in POST requests when submitting data.

```
POST /api/v1/resource HTTP/1.1
Host: example.com
Content-Type: application/json

{
  "token": "<token>"
}
```

Considerations:

- Suitable for scenarios requiring data submission.
- Avoid using this method in GET requests since GET requests are generally not designed to carry sensitive data.

Cookies

- Usage: Tokens are stored as cookies and automatically included in requests to the same domain.

```
Set-Cookie: auth_token=<token>; Secure; HttpOnly
```

Considerations:

- Security Features:
  - HttpOnly: Prevents JavaScript access to the token.
  - Secure: Ensures the cookie is only sent over HTTPS.
- Risks: Vulnerable to Cross-Site Request Forgery (CSRF) if not paired with proper protections (e.g., CSRF tokens).

Custom Headers

- Usage: Some systems may define a custom header for token transmission.

```
GET /api/v1/resource HTTP/1.1
Host: example.com
X-Auth-Token: <token>
```

Considerations:

- Less standard than the Authorization header.
- Useful in systems with specific security needs or existing conventions.

Token Placement - Best Practices

- Use the Authorization Header: Preferred for most scenarios, especially for RESTful APIs.
- Avoid Query Parameters: Only use when absolutely necessary.
- Secure Tokens in Cookies: If cookies are used, configure them with **HttpOnly**, **Secure**, and **SameSite** attributes.
- Encrypt Data in Transit: Always use HTTPS to protect tokens regardless of placement.
- Minimize Token Exposure: Avoid placing tokens in locations where they can be easily accessed, logged, or cached.

Token-Based Authentication: Applications & Use Cases

Token Placement - Best Practices

- Modern Web Applications: Token-based authentication is lightweight, stateless, and aligns well with the architecture of SPAs, Progressive Web Apps (PWAs), and other modern web applications.
- RESTful APIs: Tokens can be passed easily in HTTP headers, making them suitable for APIs that need to authenticate multiple clients (e.g., web, mobile, IoT).
- Microservices Architectures: Tokens eliminate the need for centralized session storage, allowing each service to validate the token independently, enabling better scalability and reliability.
- Cross-Domain Authentication and SSO: Tokens (e.g., JWTs, OAuth tokens) allow for secure sharing of authentication information across different domains or systems, a core requirement for SSO.

Token Placement - Best Practices

- Modern Web Applications: Token-based authentication is lightweight, stateless, and aligns well with the architecture of SPAs, Progressive Web Apps (PWAs), and other modern web applications.
- RESTful APIs: Tokens can be passed easily in HTTP headers, making them suitable for APIs that need to authenticate multiple clients (e.g., web, mobile, IoT).
- Microservices Architectures: Tokens eliminate the need for centralized session storage, allowing each service to validate the token independently, enabling better scalability and reliability.
- Cross-Domain Authentication and SSO: Tokens (e.g., JWTs, OAuth tokens) allow for secure sharing of authentication information across different domains or systems, a core requirement for SSO.

JSON Web Tokens (JWT)

JSON Web Tokens (JWT)

- JSON Web Tokens (JWTs) are a compact, URL-safe, and self-contained method for securely transmitting information between parties.

- JWTs are commonly used for authentication, authorization, and information exchange in modern web applications.

- They consist of three parts:
  - Header: Metadata about the token (e.g., signing algorithm, token type).
  - Payload: Claims (statements) about the user or session (e.g., user ID, roles, expiration).
  - Signature: Ensures the token's integrity by cryptographically signing the header and payload.

JSON Web Tokens (JWT)

- JWTs were created to address the need for a lightweight, stateless, and scalable method for managing authentication and session data in modern, distributed systems.
- Traditional session-based authentication methods often require server-side storage to manage session states, which can be resource-intensive and less scalable.
- JWTs eliminate this need by embedding session-related data directly into the token itself.

Example JWT (Base64-encoded)

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI
6IkpvaG4gRG9lIiwiYWRtaW4iOnRydWUsImV4cCI6MTcxNjQ1NDY2MH0.SflKxwRJSMeKKF2
QT4fwpMeJf36POk6yJV_adQssw5c
```

While this looks convoluted, it is actually a very compact, printable representation of a series of claims, along with a signature to verify its authenticity.

```
{
     "alg": "HS256",
     "typ": "JWT"
}
{
     "sub": "1234567890",
     "name": "John Doe",
     "admin": true
}
```

Role of JWTs in Authentication & Session Management

1. Authentication

- JWTs are widely used for authenticating users in web applications. Upon successful login:
  - The server generates a JWT containing user-specific claims.
  - The token is signed with a secret key or private key to prevent tampering.
  - The JWT is sent to the client and stored (e.g., in a cookie or local storage).
- For subsequent requests:
  - The client includes the JWT in the Authorization header or a cookie.
  - The server validates the token to authenticate the user.

```
GET /protected-resource HTTP/1.1
Host: acme.com
Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...
```

Role of JWTs in Authentication & Session Management

2. Session Management

- JWTs enable stateless session management, meaning the server does not need to store session data. All necessary information (e.g., user roles, expiration) is embedded within the token itself.

- Benefits for Session Management:
  - Scalability: Eliminates the need for server-side session storage, reducing resource usage.
  - Cross-Domain Authentication: Works well in distributed systems and APIs.
  - Decentralization: Allows third-party services to validate tokens without contacting the issuer.

JWT Structure

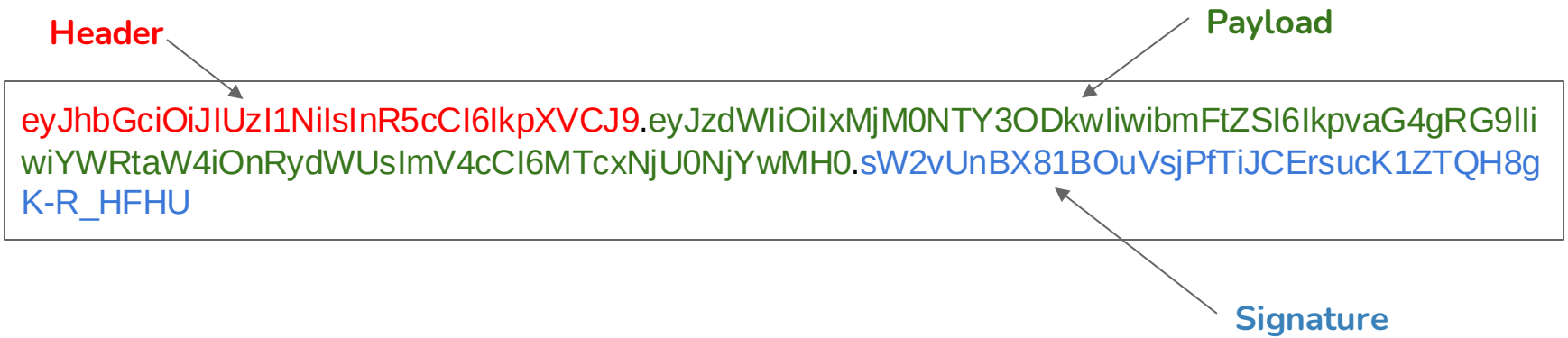
JWT Structure

- The header always comes first, followed by the payload, and finally the signature.
- This order is crucial because the signature is generated by hashing the header and payload together.
- Reversing or altering this order would invalidate the token.

JWT Structure

A JSON Web Token (JWT) is a compact, URL-safe token consisting of three parts, separated by dots (.):

HEADER.PAYLOAD.SIGNATURE



JWT Header

- Contains metadata about the token, such as the signing algorithm (e.g., HS256, RS256) and token type (JWT).

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

JWT Payload

- The payload is the second part of the JWT and contains claims that provide information about the user or session.
- Claims can be registered (iss, exp, iat), public (application-specific), or private (agreed-upon).
- This section includes information like user roles, permissions, and token metadata.
- It is base64-encoded, not encrypted, meaning it is readable if decoded.

```
{
  "sub": "1234567890",
  "name": "John Doe",
  "admin": true,
  "exp": 1716454560
}
```

JWT Signature

- The signature is the final part of the JWT and ensures token integrity and authenticity.
- It is created by signing the encoded header, payload and a secret key (for symmetric algorithms) or a private key (asymmetric algorithms).
- The signature ensures that the token has not been tampered with.

Signature Formula (HS256 Example)

```
HMACSHA256(
  base64UrlEncode(header) + "." + base64UrlEncode(payload),
  secret
)
```

JWT Claims

What Are Claims in JWTs?

- Claims in JSON Web Tokens (JWTs) are key-value pairs in the payload section of the token.
- These claims carry information about the user, session, or other relevant data and are used by the application or service to process the token.
- JWT claims are not encrypted by default (unless the JWT is encrypted using JWE). Therefore, claims are base64-encoded but easily readable, so they should not include sensitive information like passwords.

What Are Claims in JWTs?

Claims enable JWTs to:

- Provide Context: Include information about the user or session.
- Authorize Actions: Define user roles and permissions for resource access.
- Support Application Logic: Share data between services in a distributed system.

Registered Claims

- These are predefined, optional claims that provide a standardized way of describing common information. Examples include:
  - iss (Issuer): Identifies who issued the token (e.g., a server or authentication service).
  - sub (Subject): Identifies the subject of the token, often a user ID.
  - aud (Audience): Indicates the intended recipient(s) of the token (e.g., a specific API).
  - exp (Expiration Time): Specifies when the token expires (in Unix timestamp format).
  - iat (Issued At): Indicates when the token was created (in Unix timestamp format).
  - nbf (Not Before): Specifies when the token becomes valid.

Registered Claims

```
{
  "iss": "auth.example.com",
  "sub": "1234567890",
  "aud": "example-app",
  "exp": 1716546600,
  "iat": 1716543000
}
```

Public Claims

- These are custom claims defined by the application developer. Public claims must be unique to avoid collisions with other claim names. They typically store user-specific or application-specific data.
- Examples:
  - role: Defines the user's role (e.g., admin, user).
  - email: Stores the user's email address.
  - permissions: Lists access rights for the user.

```
{
  "role": "admin",
  "email": "user@example.com",
  "permissions": ["read", "write", "delete"]
}
```

Private Claims

- Private claims are custom claims that are agreed upon between parties exchanging the JWT. They are not standardized and are typically used for application-specific data.
- Examples:
  - department: Specifies the user's department.
  - cartId: Stores a shopping cart ID for an e-commerce app.

```
{
  "department": "sales",
  "cartId": "abc123"
}
```

WSDL Disclosure & Method Enumeration

WSDL Disclosure

- When dealing with web service security, accessing the WSDL file is the first step; this gives us the full list of operations and types allowed by the server as well as the correct syntax to use, inputs, outputs and all the useful information we may need to run successful attacks.

- Before we can enumerate the WSDL file for the SOAP web service, we need to identify the SOAP web service and endpoints.

WSDL Disclosure

- Once the SOAP service has been identified, another way to discover WSDL files is by appending ?wsdl,.wsdl or ?disco to the end of the service URL:

WSDL Disclosure

- Once we find WSDL files, we can start inspecting them and gather valuable information about the web service.
- As you already know, this allows us to gather information such as operations, data, syntax and much more.

Demo: WSDL Disclosure & Method Enumeration

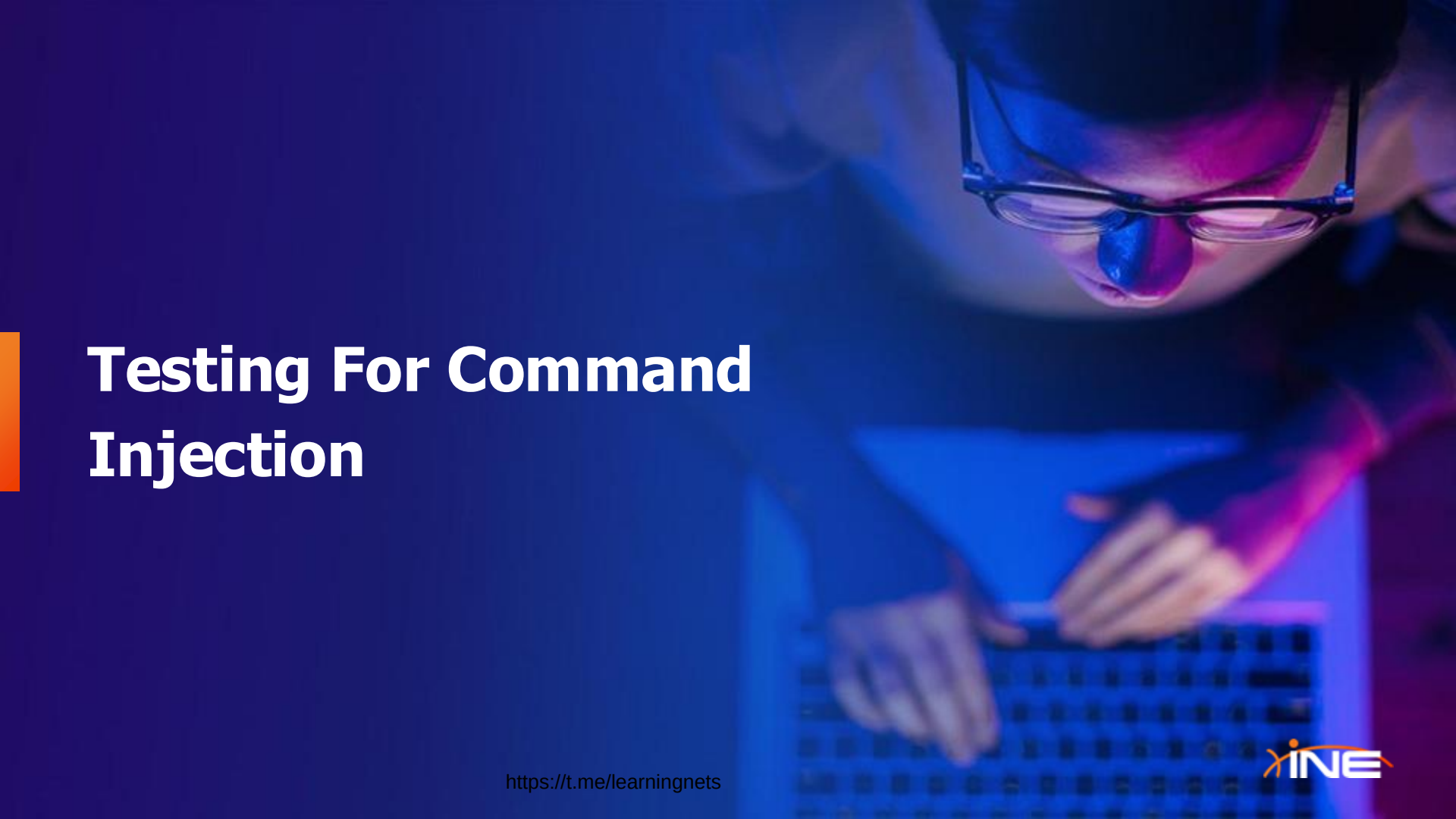
Invoking Hidden Methods

Demo: Invoking Hidden Methods

Testing For SQL Injection

Demo: Testing For SQL Injection

Testing For Command Injection



Demo: Testing For Command Injection

API Penetration Testing

Course Summary

Key Concepts - Recap

- Introduction to Web Services & APIs
- Introduction to Web API Security
- API Reconnaissance
- API Authentication Testing
- API Injection Vulnerabilities

Learning Outcomes Recap

- Understand the purpose and structure of APIs, including REST, SOAP, and GraphQL.
- Recognize common API security risks and their impact.
- Apply a systematic methodology for API penetration testing.
- Perform reconnaissance to discover API endpoints and resources.
- Test API authentication mechanisms for vulnerabilities.
- Identify and exploit API injection vulnerabilities.

Real-World Applications

- Securing APIs in Real-World Environments: Use your understanding of API fundamentals to identify and address vulnerabilities in modern web services.
- Enhancing Organizational Security: Apply API security principles to strengthen the overall security posture of web applications.
- Executing Professional Penetration Tests: Leverage a structured methodology to conduct thorough and effective API penetration tests.
- Identifying API Weak Points: Perform detailed reconnaissance to uncover exposed endpoints and assess their risk.
- Hardening Authentication Systems: Evaluate and improve API authentication mechanisms to prevent unauthorized access.

Next Steps

+ Explore Advanced API Security Topics: Dive into areas like OAuth 2.0, OpenID Connect, API rate limiting, and secure API design principles.

+ Learn API Testing Automation: Master tools like Postman, Burp Suite, and OWASP ZAP for automated API security testing.

+ Study Cloud API Security: Understand API security in cloud environments, focusing on AWS, Azure, and Google Cloud APIs.

THANKS FOR WATCHING!

EXPERTS AT MAKING YOU AN EXPERT