Server-Side Attacks

Course Overview

Alexis Ahmed

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

Key Concepts

+ Modern Web Application Architecture
+ Server-Side Request Forgery (SSRF)
+ Insecure Deserialization

MAJOR TOPICS

- Modern Web Application Architecture Models
- Server-Side Vulnerabilities
- Server-Side Request Forgery (SSRF)
- Insecure Deserialization
- Java, PHP & .Net Deserialization

LEARNING OUTCOMES

+ **Understand Modern Web App Architecture:** Describe the components and layers of modern web application infrastructure and explain the role and interaction of the components that make up a web application.
+ **Recognize Server-Side Vulnerabilities:** Understand what server-side attacks are and how they exploit weaknesses in the back-end infrastructure.
+ **SSRF:** Define SSRF and explain its causes, attack anatomy, and variations. Demonstrate practical exploitation of SSRF vulnerabilities, including blind and advanced attack techniques.
+ **Insecure Deserialization:** Explain what insecure deserialization is and how it can lead to vulnerabilities like remote code execution (RCE).
+ **Exploit Language-Specific Deserialization Vulnerabilities:** Identify and exploit deserialization vulnerabilities in Java, PHP and .NET Applications..

PREREQUISITES

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

LET'S GO!

Modern Web Application Architecture

Modern Web Application Architecture

When interacting with a web application through a browser, it may seem like a simple, one-dimensional system where a single entity serves your requests. ***However, the reality is far more complex.***

Modern web applications are built on **sophisticated architectures** with multiple interconnected components working together to deliver a seamless experience.

Understanding modern web application architecture is essential for web application penetration testers because it provides a clear view of how various components, such as web servers, application servers, APIs, and databases, interact and function together.

Modern Web Application Architecture

Understanding modern web app architecture will enable you to better understand and map out the functionality of a web application, identify the various components that make up it's architecture and the technology stack it uses. Furthermore, this knowledge is also useful in understanding **layer** specific vulnerabilities, how they can propagate across layers, and the potential impact of exploiting said weaknesses or vulnerabilities.

In the context of this course, "**Server-Side Attacks**", understanding web application architecture, more specifically, server-side infrastructure is particularly important, as vulnerabilities like **SSRF** and **insecure deserialization** often exploit the architecture's internal trust relationships, data flows, and component interactions.

By understanding a web app's architecture, you can approach these attacks with precision, uncover hidden flaws, and recommend targeted mitigations.

What is Web Application Architecture?

Web application architecture refers to the **framework** and **structure** that defines how the components of a web application interact with each other and function together to deliver the application to users.

It encompasses the **design** of the application's **front-end, back-end, data storage**, and the **communication** between them, ensuring they work cohesively to provide a seamless user experience.

In essence, web application architecture is the foundation that determines the application's functionality, performance, and adaptability, making it a critical element in the development process. ***Without a solid architecture, even the most innovative web applications can fail to meet user expectations or business goals.***

Modern Web Application Architecture

At their core, web applications often consist of:

- **Application Servers:** Process business logic and handle dynamic requests from users.
- **Database Servers:** Dedicated servers for storing and retrieving application data, typically secured behind firewalls and isolated from direct external access.
- **Performance Optimization Mechanisms:** Services like Content Delivery Networks (CDNs) that cache static assets (e.g., images, JavaScript, CSS) closer to the user to reduce latency and server load.

Modern Web Application Architecture

This architectural organization allows for scalability, security, and high availability. For instance, in such configurations, the IP address associated with the application's domain name often points to intermediary services like load balancers or CDNs, which obscure the actual application servers from direct exposure to users.

While the client-side experience appears straightforward, the underlying architecture is a complex, layered system that ensures performance, resilience, and reliability.

Each component is meticulously designed to handle specific roles, making modern web applications robust and adaptable to varying demands.

Web Application Architecture Models

Web Application Architecture Models

Model	Description	Best Use Case
Layered	Organizes components into distinct layers (e.g., presentation, business logic, data).	Traditional monolithic applications.
Microservices	Small, independent services that handle specific business functionalities, communicating via APIs.	Large, complex, and scalable applications.
SOA	Reusable, centralized services managed via an Enterprise Service Bus (ESB).	Enterprise-level integrations.
Event-Driven	Processes triggered by user actions or system events, often asynchronously.	Real-time systems like e-commerce or IoT.
Serverless	Event-triggered functions managed by a cloud provider (FaaS).	Lightweight, event-driven workloads.
Monolithic	All application components bundled into a single unit.	Small applications or early-stage startups.
Distributed	Application spread across multiple interconnected systems or nodes.	Highly scalable and fault-tolerant systems.

Layered Architecture

This architecture model organizes the application into distinct layers, each responsible for specific tasks.

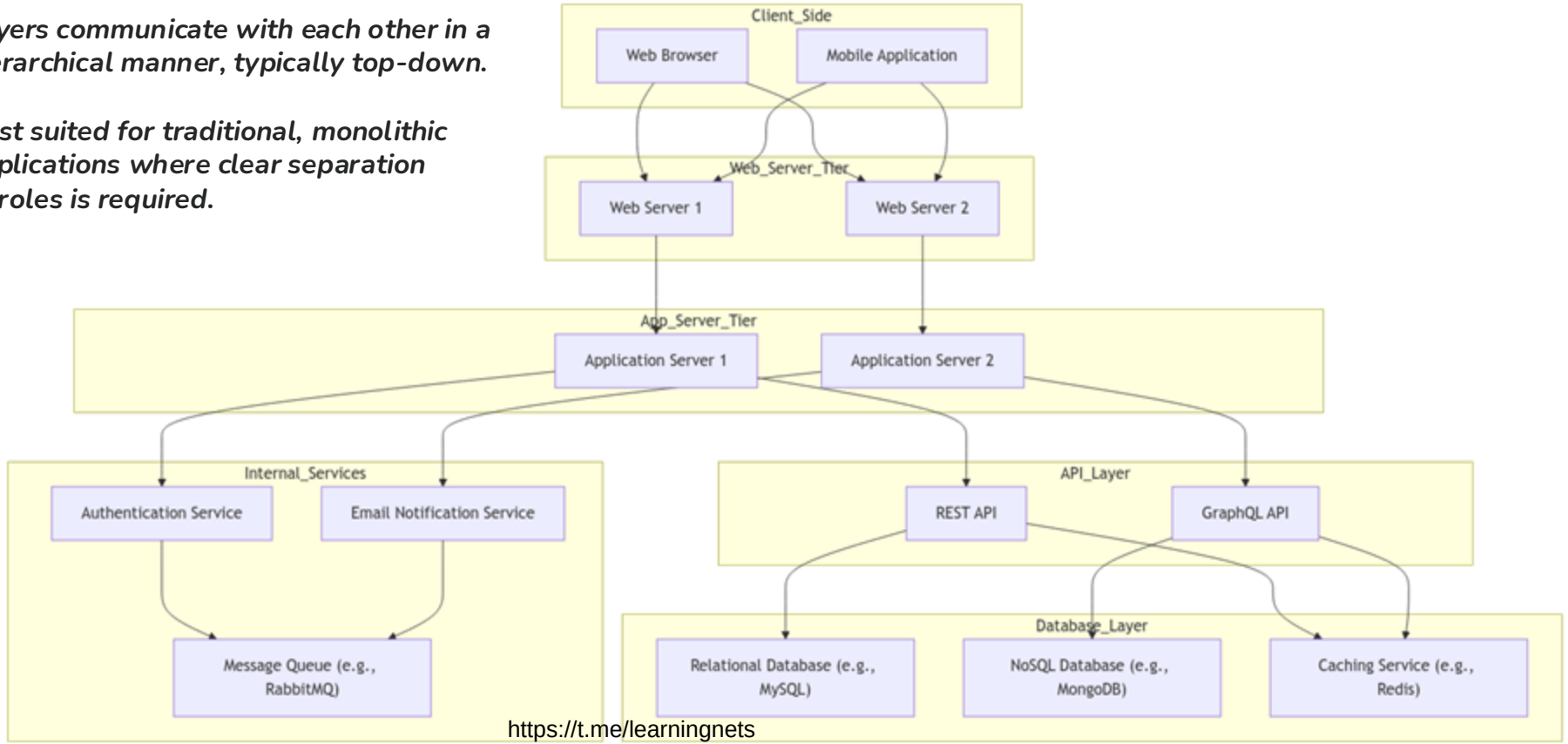
Common layers include:

- Presentation Layer (Client-Side): Handles user interactions (e.g., browsers, mobile apps).
- Business Logic Layer (Application Server): Processes requests, applies logic, and interacts with the database.
- Data Access Layer (Database Layer): Manages data storage and retrieval.
- Layers communicate with each other in a hierarchical manner, typically top-down.

Layered Architecture

Layers communicate with each other in a hierarchical manner, typically top-down.

Best suited for traditional, monolithic applications where clear separation of roles is required.



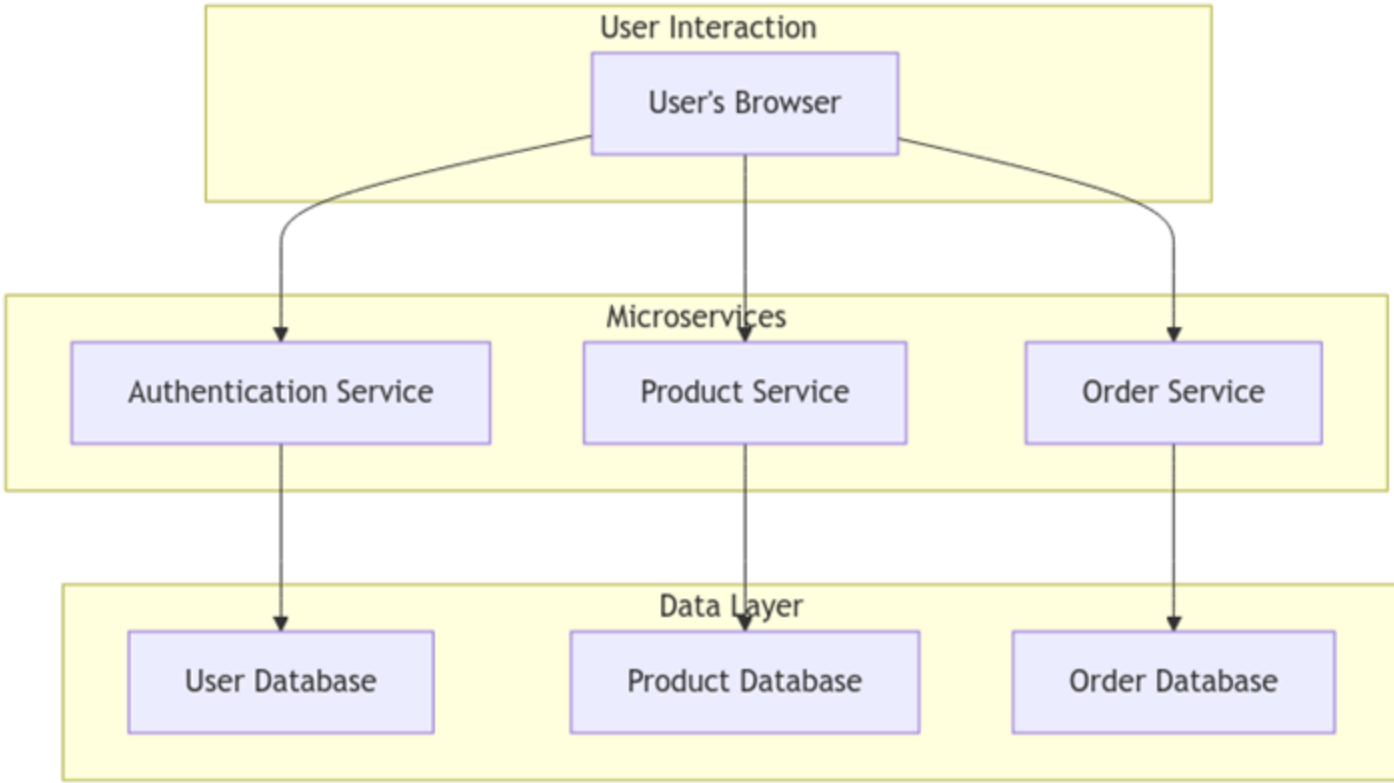
Microservices Architecture

This model divides the application into small, **independent** services, each handling a specific business capability (e.g., user authentication, payment processing).

Each service is loosely coupled and **independently deployable**, Communicates with other services through APIs (e.g., REST, gRPC) and often uses its own database or data storage mechanism.

This model is ideal for large-scale, complex applications requiring **flexibility**, **scalability**, and **continuous deployment**. It enables scalability and independent updates for each service and promotes the use of different technologies for different services ("polyglot programming").

Microservices Architecture



Modern Web Application Architecture Example

Layered Architecture Example

Here's a detailed example of an e-commerce website that uses a layered architecture model.

The architecture comprises the following:

- Presentation Layer: This layer includes the user's browser or front-end interface, which handles the application's visual elements and user interactions. It is responsible for sending requests to and receiving responses from the back-end.
- Business Logic Layer:The heart of the application resides here. The Application Server processes requests from the presentation layer, applies business rules (e.g., validating user inputs, calculating discounts, or managing order workflows), and orchestrates interactions with the data access layer.
- Data Access Layer: This layer connects to the Database, which stores critical data such as user accounts, product catalogs, and order histories. It ensures data integrity and efficient retrieval of information.

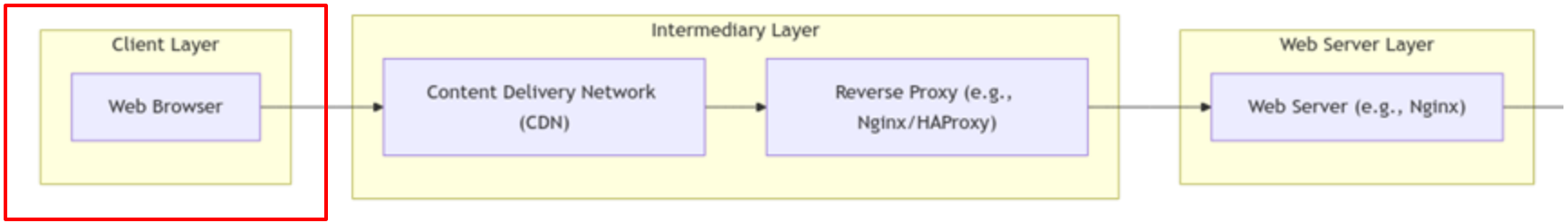
Layered Architecture Example

Client Layer: This is the front-end interface that customers interact with.

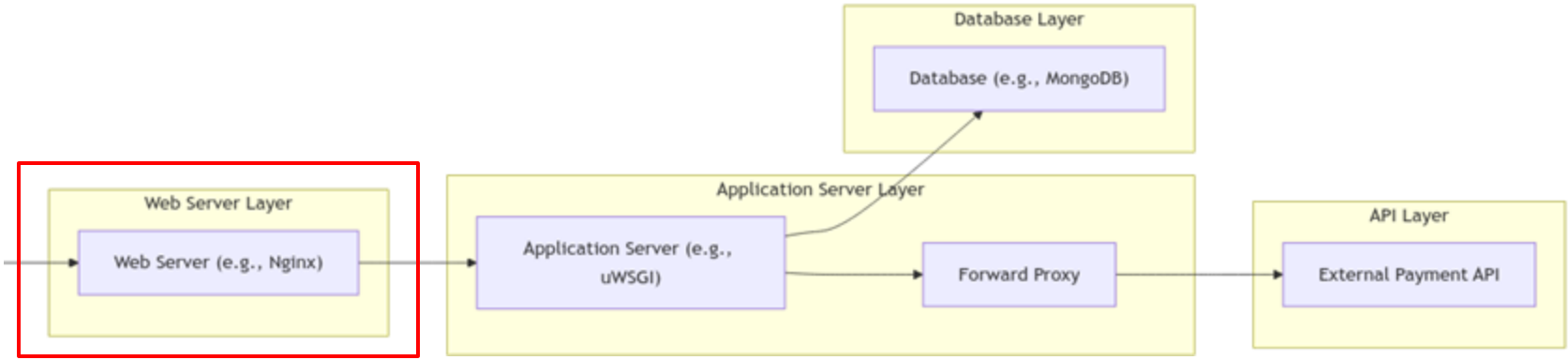
Components:

- ➔ An e-commerce website that provides:
- ➔ Product browsing and search.
- ➔ Shopping cart functionality.
- ➔ User account management.

The website runs in web browsers and sends HTTP requests to the web server.



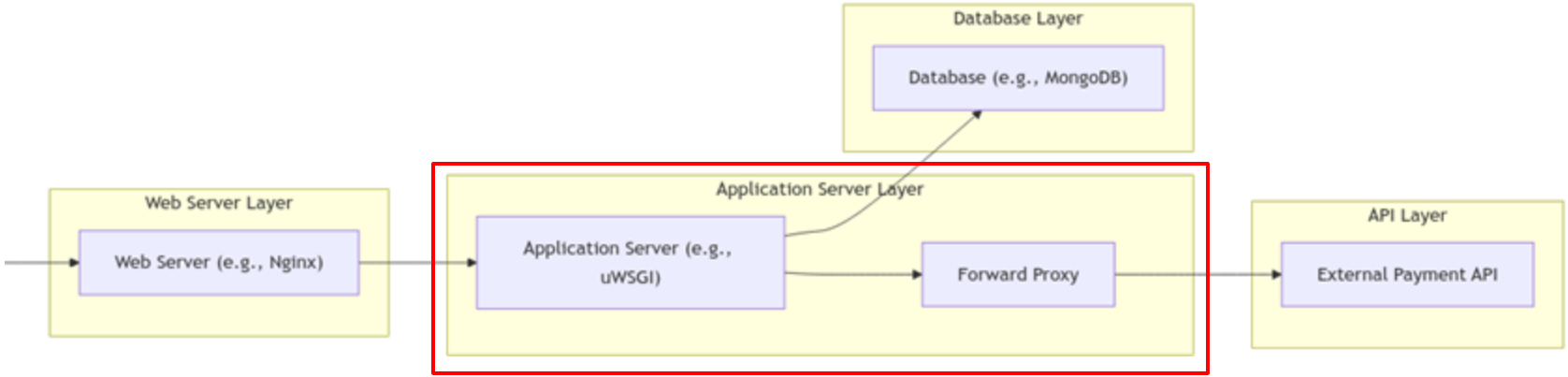
Layered Architecture Example



Web Server Layer: Nginx

- ➔ Handles incoming HTTP/HTTPS requests from clients.
- ➔ Serves static files (e.g., images, CSS, JavaScript).
- ➔ Acts as a reverse proxy, forwarding dynamic requests to the application server (uWSGI).
- ➔ Provides load balancing if multiple application servers are deployed.

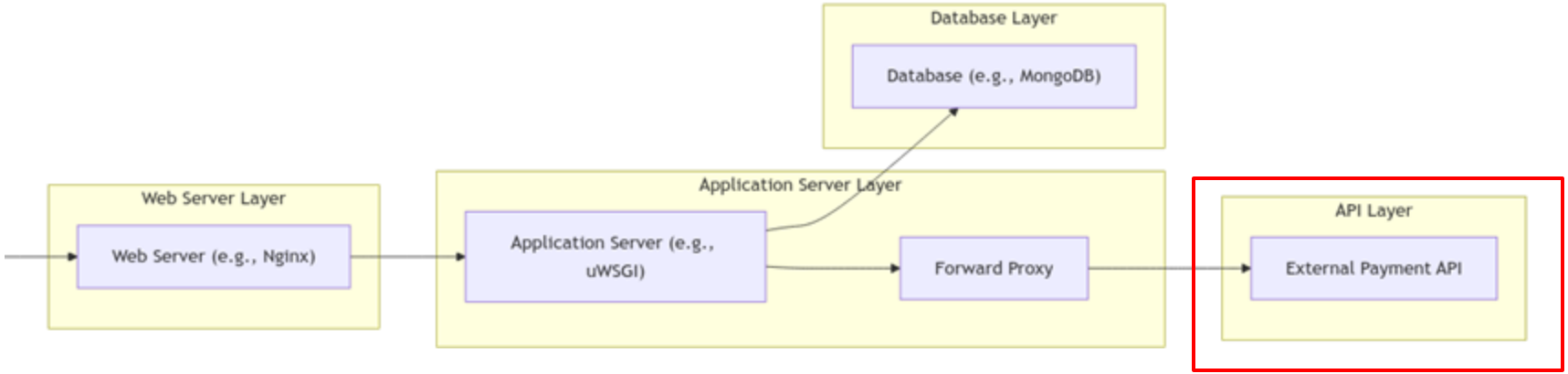
Layered Architecture Example



Application Server Layer: uWSGI

- ➔ Processes dynamic requests received from Nginx.
- ➔ Executes business logic and handles the back-end operations for the e-commerce website.
- ➔ Manages user authentication, shopping cart logic, order processing, etc.

Layered Architecture Example

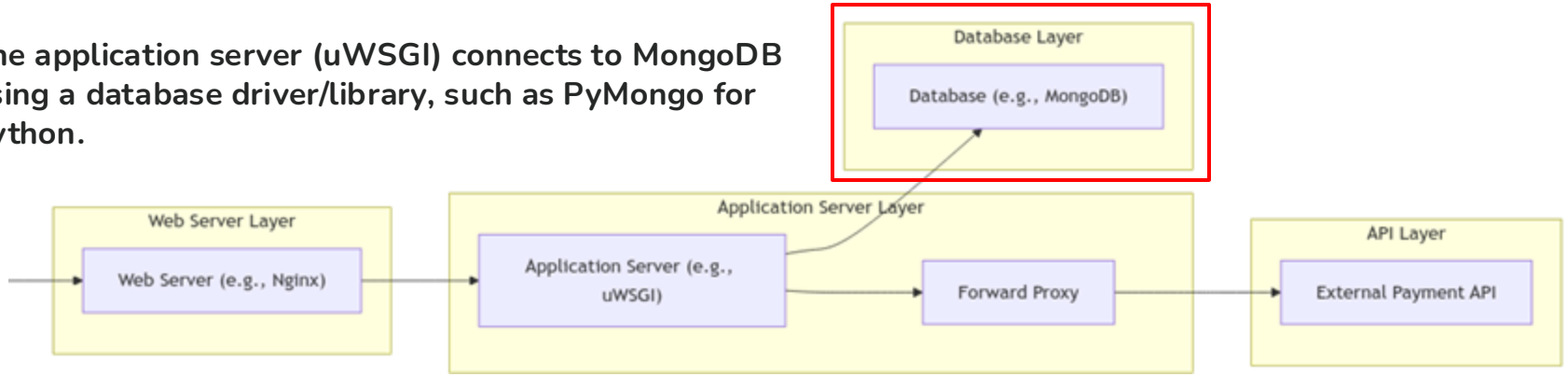


API Layer: Provides structured interfaces for communication between the application server and external or internal systems.

- ➔ Product Information API: Provides up-to-date information on product availability, pricing, and descriptions.
- ➔ Order Management API: Tracks customer orders and provides order status updates.

Layered Architecture Example

The application server (uWSGI) connects to MongoDB using a database driver/library, such as PyMongo for Python.



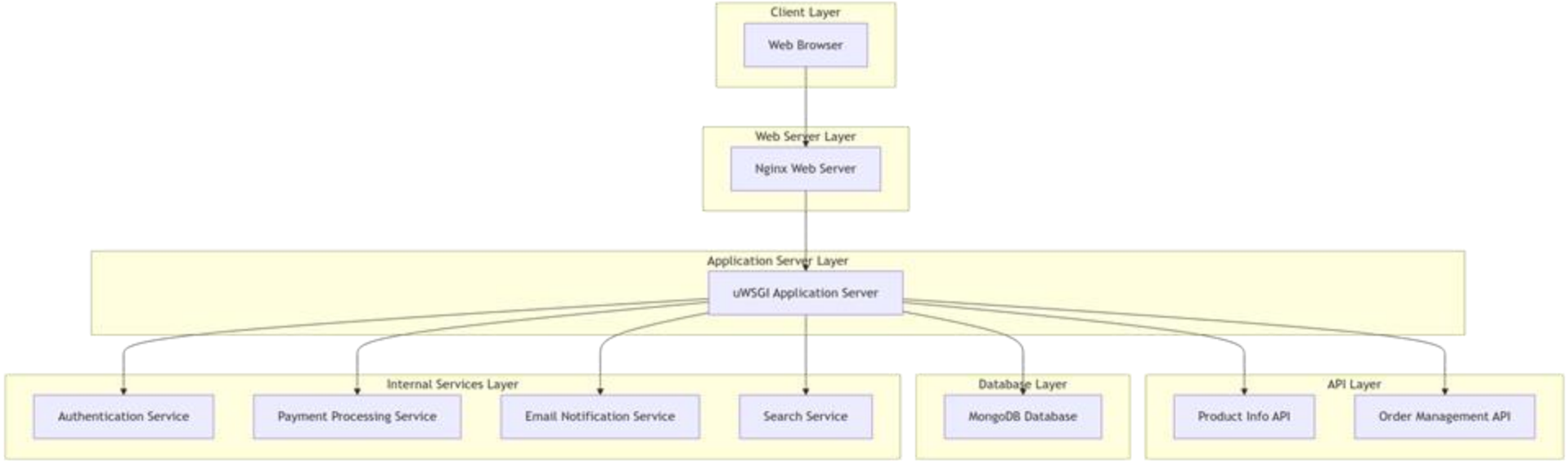
Database Layer: MongoDB

Stores e-commerce data, including:

- ➔ Product catalog.
- ➔ Customer information.
- ➔ Order history and transaction records.

Hosted on a separate server for scalability and security.

Layered Architecture Example



Server-Side Attacks

Server-Side Attacks

What does “server-side” mean?

In the context of web application penetration testing, **server-side** refers to the **processes**, **logic**, and **infrastructure** that **operate on the server** rather than the client.

Server-side functionality handles critical tasks such as **request processing**, **business logic execution**, communication with databases and APIs, and integration with internal services.

Unlike client-side vulnerabilities, which target the user-facing components, server-side vulnerabilities exploit weaknesses in how the server processes inputs, communicates with other systems, and manages resources.

Server-Side Attacks

Server-side attacks are exploits that target the **back-end components** of a web application, including **web servers**, **application servers**, APIs, and databases.

Unlike client-side attacks, which focus on the user's browser or front-end, server-side attacks manipulate or exploit vulnerabilities in the **server-side infrastructure**.

These attacks can lead to unauthorized access, data breaches, or even full system compromise, as they leverage weaknesses in how servers process requests, communicate with other components, and manage data.

In this course, we will only be focusing on vulnerabilities specific to the Application/Business layer as we have already explored attacks against the data layer (SQLi, NoSQl Injection etc)

Server-Side Attacks

In order to **differentiate server-side vulnerabilities** from other types of web application vulnerabilities like injection vulnerabilities, you need to understand the specific functional characteristics and contexts of server-side functionality in a modern web application infrastructure. Here are the key criteria and factors to consider:

Execution Context

- ➔ Server-side vulnerabilities occur in the code or logic **executed on the server** rather than on the client (e.g., browser).
- ➔ This includes processes and interactions within components such as the web server, application server, and back-end services.

Server-Side Attacks

Scope

- ➔ These vulnerabilities typically affect the server's infrastructure, configuration, or inter-component communication rather than the client-side or database alone.
- ➔ Examples: ***File system manipulation, internal network access, and configuration exposure.***

Interaction with Components

- ➔ Vulnerabilities that exploit how the server interacts with other components (e.g., APIs, microservices, or external systems).
- ➔ ***These do not directly target the database layer (e.g., SQL injection) but may involve communication with it indirectly.***

Server-Side Attacks

Trust Assumptions

- ➔ Server-side vulnerabilities often **exploit implicit trust** between the server and its resources or components.

For example:

- ➔ Trust in user input leading to **insecure deserialization**.
- ➔ Trust in internal network requests leading to **SSRF**.

Server Resources

- ➔ Vulnerabilities that manipulate or exploit the resources and capabilities of the server, such as its file system, memory, or CPU.
- ➔ Example: Exploiting file upload functionality to gain unauthorized access to the server's file system.

Layers Targeted by Server-Side Attacks

In a typical web application architecture, server-side attacks primarily target the following layers:

Web Server Layer

- ➔ Handles HTTP requests and serves static content.
- ➔ Acts as a gateway to the application server.

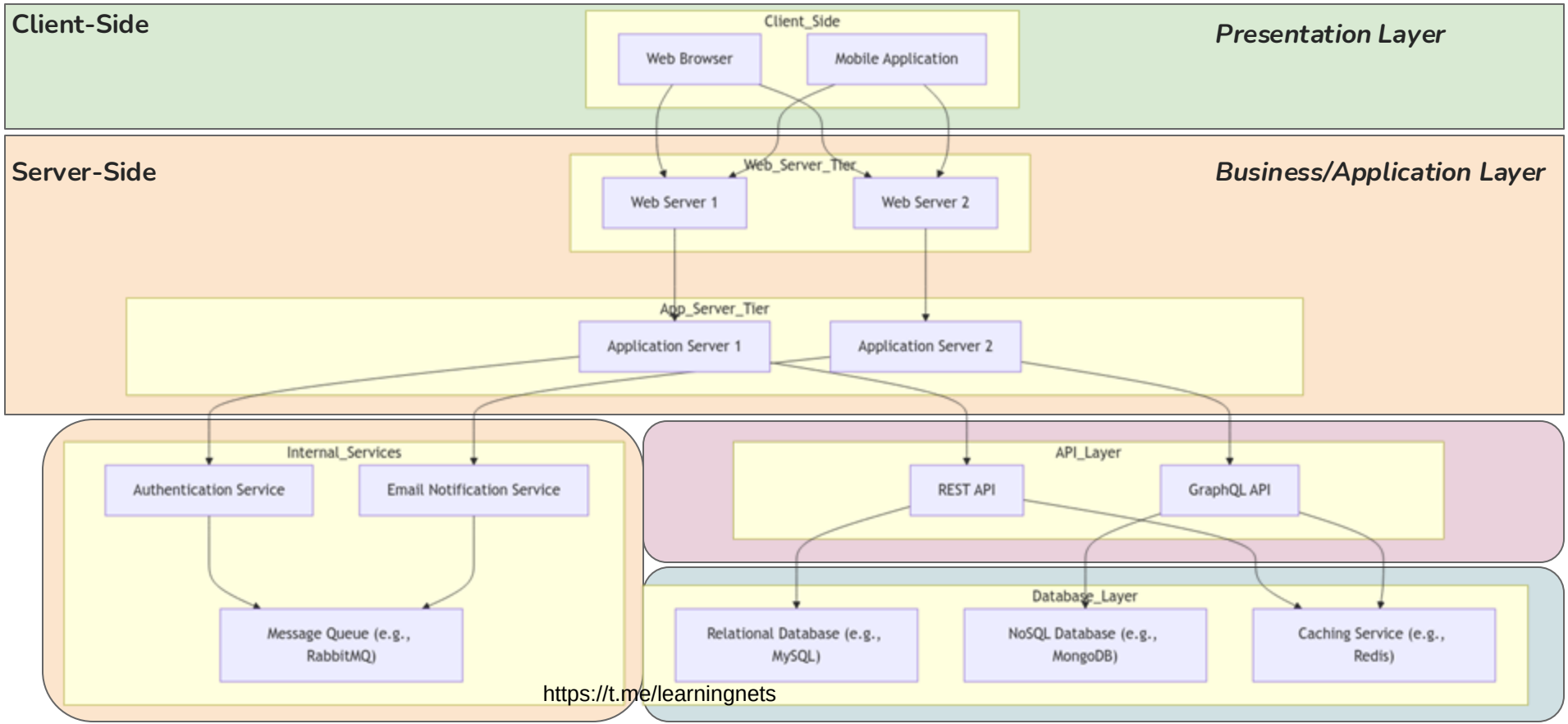
Application Server Layer

- ➔ Processes dynamic content and executes business logic.
- ➔ Communicates with the database and APIs to fulfill client requests.

Data Access Layer (Indirectly)

- ➔ Exploits from the application or web server may manipulate data or compromise database integrity.

Server-Side Attacks



Web Server vs. App Server

Aspect	Web Server	Application Server
Role	Serves static content, routes requests.	Executes business logic, generates dynamic responses.
Interaction	Communicates with clients (HTTP/HTTPS).	Communicates with databases, APIs, and services.
Processing	Handles lightweight tasks (e.g., static file serving).	Handles heavy processing tasks (e.g., data processing).
Output	Serves files (HTML, CSS, JavaScript).	Serves computed data (JSON, HTML).
Examples	Apache, Nginx, IIS.	Node.js, Django, Spring Boot.

Vulnerabilities in the Web Server Layer

Directory Traversal

Exploits improper validation of file paths to access sensitive files outside the intended directory (e.g., `../../etc/passwd`).

HTTP Response Splitting

Manipulates HTTP headers to inject malicious responses or redirect users to malicious sites.

Misconfigured SSL/TLS

Exploits weak or outdated configurations (e.g., supporting SSLv3 or weak ciphers), allowing man-in-the-middle (MITM) attacks.

File Inclusion Vulnerabilities

Exploits improper handling of file paths to execute unauthorized files on the server.

Vulnerabilities in the App Server Layer

Server-Side Request Forgery (SSRF)

Exploits the server's ability to make HTTP requests, enabling attackers to access **internal** or **external** resources.

Insecure Deserialization

Allows attackers to inject malicious payloads during the deserialization process, leading to remote code execution or privilege escalation.

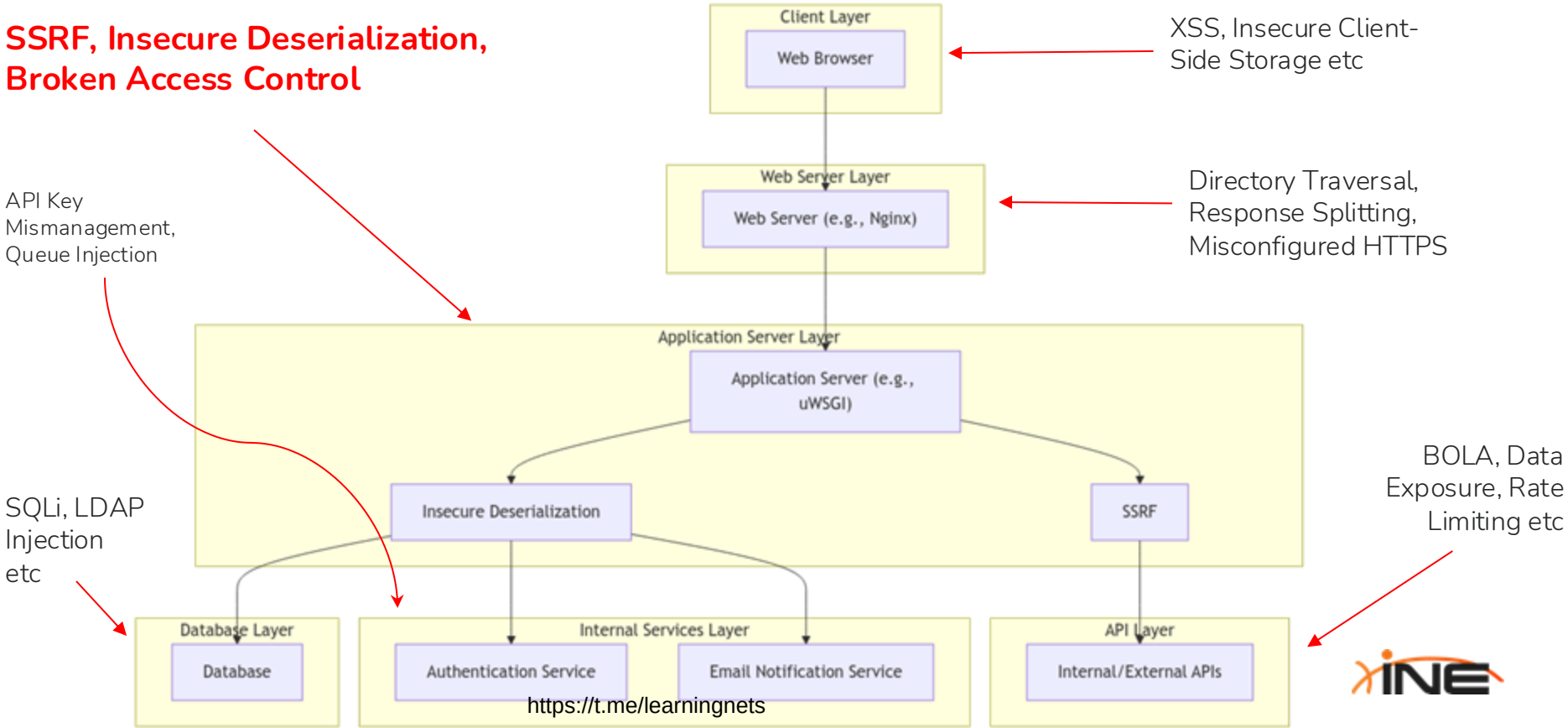
Broken Access Control

Flaws in authorization logic allow attackers to perform actions or access resources without proper permissions.

Command Injection

Exploits vulnerabilities in the application server to execute arbitrary system commands on the host.

Server-Side Vulnerabilities Visualized



Server-Side Request Forgery (SSRF)

Server-Side Request Forgery (SSRF)

Server-Side Request Forgery (SSRF) is a vulnerability that allows an attacker to **trick** a server into making unauthorized requests to **internal** or **external** resources on its behalf.
These requests are initiated by the vulnerable application but controlled by the attacker, often leading to exposure of sensitive data or access to restricted systems.

Successful exploitation of SSRF can lead to:

- ➔ Sensitive data disclosure
- ➔ Theft of authentication information
- ➔ Internal network reconnaissance
- ➔ RCE → SSRF Can be used as an entry point for more advanced attacks such as RCE or privilege escalation
- ➔ File read/inclusion

What Causes SSRF?

SSRF vulnerabilities are typically caused by:

- **Unvalidated Input:** Applications accept user-supplied URLs or resources without properly validating or sanitizing them.
- **Server Trust:** The server has access to resources not directly available to the attacker, such as internal APIs, databases, or cloud metadata services.
- **Improper Access Controls:** Applications do not enforce strict policies to prevent unauthorized or unintended requests.

Examples of vulnerable scenarios include:

- ***Image fetchers or URL preview generators that download content from user-supplied URLs.***
- ***API integrations where the server processes user input as part of an external request.***

Anatomy of an SSRF Attack

Anatomy of an SSRF Attack

1 - Identify a Vulnerable Endpoint

- ➔ Look for functionality where the application fetches resources or content from a user-provided URL (e.g., file downloaders, webhooks, or API testers).

2 - Manipulate the Input

Craft a malicious URL that points to an internal or unauthorized resource, such as:

- ➔ Internal servers (**http://localhost**, **http://127.0.0.1**).
- ➔ Cloud metadata endpoints (e.g., **http://169.254.169.254** on AWS).

3 - Server Processes the Request

- ➔ The application server processes the attacker-controlled URL and executes the request.
- ➔ The server's trust in its network allows access to otherwise restricted resources.

Anatomy of an SSRF Attack

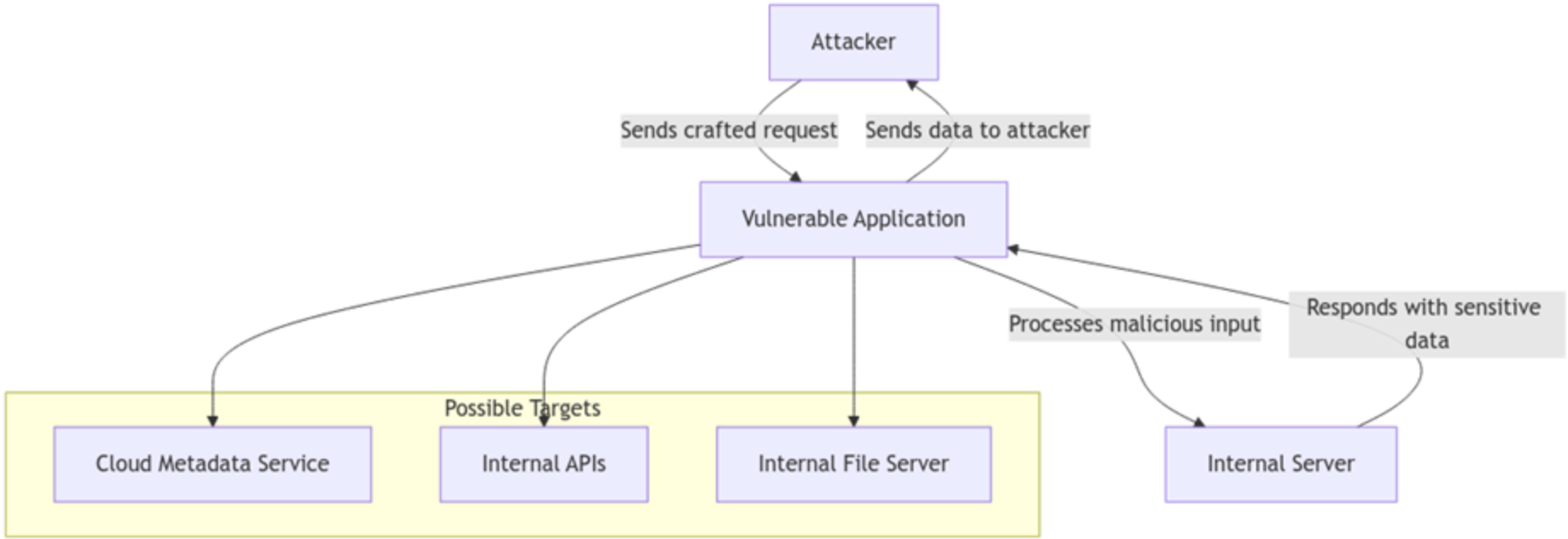
4 - Extract or Exploit the Response

The attacker may retrieve sensitive information (e.g., internal service data, credentials).

In advanced cases, the attacker may use SSRF to:

- ➔ Perform port scanning on internal systems.
- ➔ Chain attacks with other vulnerabilities (e.g., deserialization or RCE).

Anatomy of an SSRF Attack



Types of SSRF Attacks

Types of SSRF Attacks

Basic SSRF

The attacker uses a malicious URL to fetch internal resources or external data via the server.

Example:

- Vulnerable URL: **http://example.com/fetch?url=http://attacker.com/malicious.png**
- Attacker modifies it to: **http://example.com/fetch?url=http://localhost/admin**

Blind SSRF

The attacker cannot directly see the server's response but can infer the results through side effects (e.g., server logs, DNS resolution).

Example:

- Sending a DNS query to a controlled domain (**http://example.com/fetch?url=http://attacker-controlled-domain.com**) and monitoring the DNS logs.

Types of SSRF Attacks

SSRF to Internal Reconnaissance

The attacker uses SSRF to scan internal networks or services that are not exposed externally.

Example:

- Testing for open ports: **http://example.com/fetch?url=http://127.0.0.1:80**

SSRF Chaining

SSRF is combined with other vulnerabilities for advanced exploitation:

- Cloud Exploitation: Access cloud metadata services (e.g., AWS IAM credentials via **http://169.254.169.254/latest/meta-data/**).
- Privilege Escalation: Access sensitive internal services that lead to privilege escalation or lateral movement.

Basic SSRF Exploitation

Demo: Basic SSRF Exploitation

SSRF to RCE

Lab Demo: SSRF to RCE

Introduction to Insecure Deserialization

Serialization & Deserialization

What is Serialization?

Serialization is the process of **converting** an **object** or **data structure** (*e.g., a Python object, a Java object, or a C# object*) into a format that can be easily **stored** or **transmitted** and then **reconstructed** (**deserialized**) later.

The **serialized format** is often a byte stream, text-based format (e.g., JSON, XML), or a binary representation.

Objects are typically serialized to convert them into a format that can be easily stored, transmitted, and reconstructed later, enabling data persistence and communication across different systems or components.

The recipient of serialized data should be able to reconstruct(deserialize) the object back to its unchanged original form.

How Serialization Works

1. **Converting the Object:**

The in-memory representation of an object (its properties, methods, and state) is transformed into a specific format.

Example formats include:

- ➔ **Binary**: Compact and fast for storage and transmission.
- ➔ **Text**: Human-readable formats like JSON, XML, or YAML.

1. **Storing or Transmitting:**

Once serialized, the data can be stored (e.g., in a file or database) or transmitted (e.g., over a network or through an API).

1. **Reconstructing the Object (Deserialization):**

The serialized data is parsed to recreate the original object or data structure in memory.

What is Serialization?

What are objects?:

In the context of serialization, an **object** refers to a **structured data entity** in programming that encapsulates properties (data/attributes) and potentially behaviors (methods/functions), representing a specific instance of a class or data type that can be converted into a storable or transmittable format for later reconstruction.

What is Deserialization?

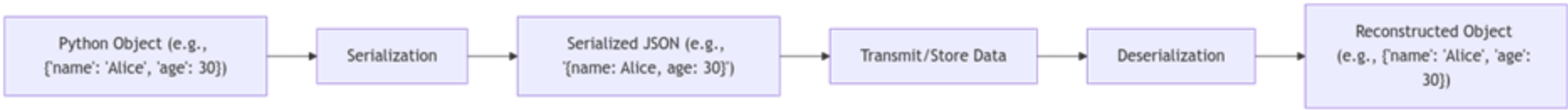
Deserialization is the process of converting serialized data (e.g., a byte stream, JSON string, or binary format) back **into its original** in-memory representation, such as an object or data structure, so it can be used again in the application.

This process **reconstructs** the data's structure and state, enabling operations or interactions as if the data had never been serialized.

The most popular and widely used data format used for data serialization is JSON, prior to which was XML.

Many programming languages provide built-in capabilities for object serialization, often offering more features than formats like JSON or XML, including greater flexibility and customization of the serialization process.

Serialization & Deserialization



The diagram illustrates the process of serialization and deserialization using a structured flow of data transformation:

- **Serialization**: A PythonObject with attributes (e.g., name and age) is converted into a SerializedData format, such as JSON or binary. This step prepares the object for storage or transmission.
- **Transmit or Store**: The serialized data is either transmitted over a network (e.g., via an API) or saved to persistent storage (e.g., a file or database).
- **Retrieve or Load**: The serialized data is retrieved from storage or received from the network, maintaining its serialized format.
- **Deserialization**: The serialized data is transformed back into a ReconstructedObject, restoring its original structure and attributes (e.g., name and age).

Considerations

It is important to note that serialized data is typically **neither encrypted nor signed** by default. This means that, once intercepted, it can often be freely tampered with, potentially leading to unexpected or malicious behavior in the component responsible for deserialization.

In some cases, transport protocols may incorporate serialization alongside compression or encryption, which can conceal or secure the serialized data. However, serialized data may also be encountered in an unprotected, plaintext format.

The latter scenario is particularly significant for penetration testers and attackers, as it provides an opportunity to analyze and manipulate the data.

Insecure Deserialization

Insecure Deserialization

Insecure deserialization is a vulnerability that occurs when **untrusted** or **user-controlled** data is **deserialized** by an application **without** proper validation or security checks.

This can lead to severe consequences such as **remote code execution (RCE)**, **privilege escalation**, or **data manipulation**.

Serialization converts an object into a format for storage or transmission, while deserialization reconstructs the object from that format. If deserialization processes input that has been tampered with, it may execute unintended actions or compromise the application.

How it Works

- **Serialization:** The application serializes an object into a format (e.g., JSON, XML, or binary) and stores or transmits it.
- **Deserialization:** Later, the application reconstructs the object by deserializing the stored or transmitted data.
- **Exploitation:** If an attacker intercepts or controls the serialized data, they can modify it to inject malicious payloads or manipulate object behavior.

The vulnerable application deserializes the tampered data without validation, leading to unexpected or dangerous outcomes.

What Causes the Vulnerability?

Lack of Input Validation:
The application blindly accepts serialized data from untrusted sources without validating its authenticity or integrity.

Overly Permissive Deserialization Libraries:
Many deserialization frameworks automatically instantiate objects, call methods, or execute code as part of the deserialization process, increasing attack vectors.

Trusting User Input:
Serialized data from users or external sources is inherently untrusted but is often treated as safe by applications.

Complex Object Hierarchies:
Deserializing data into complex objects or hierarchies can introduce unexpected behaviors, especially if deserialization invokes constructors or methods.

Lack of Security Controls:
Missing cryptographic signing or validation mechanisms allow attackers to tamper with serialized data undetected.

Impact(s)

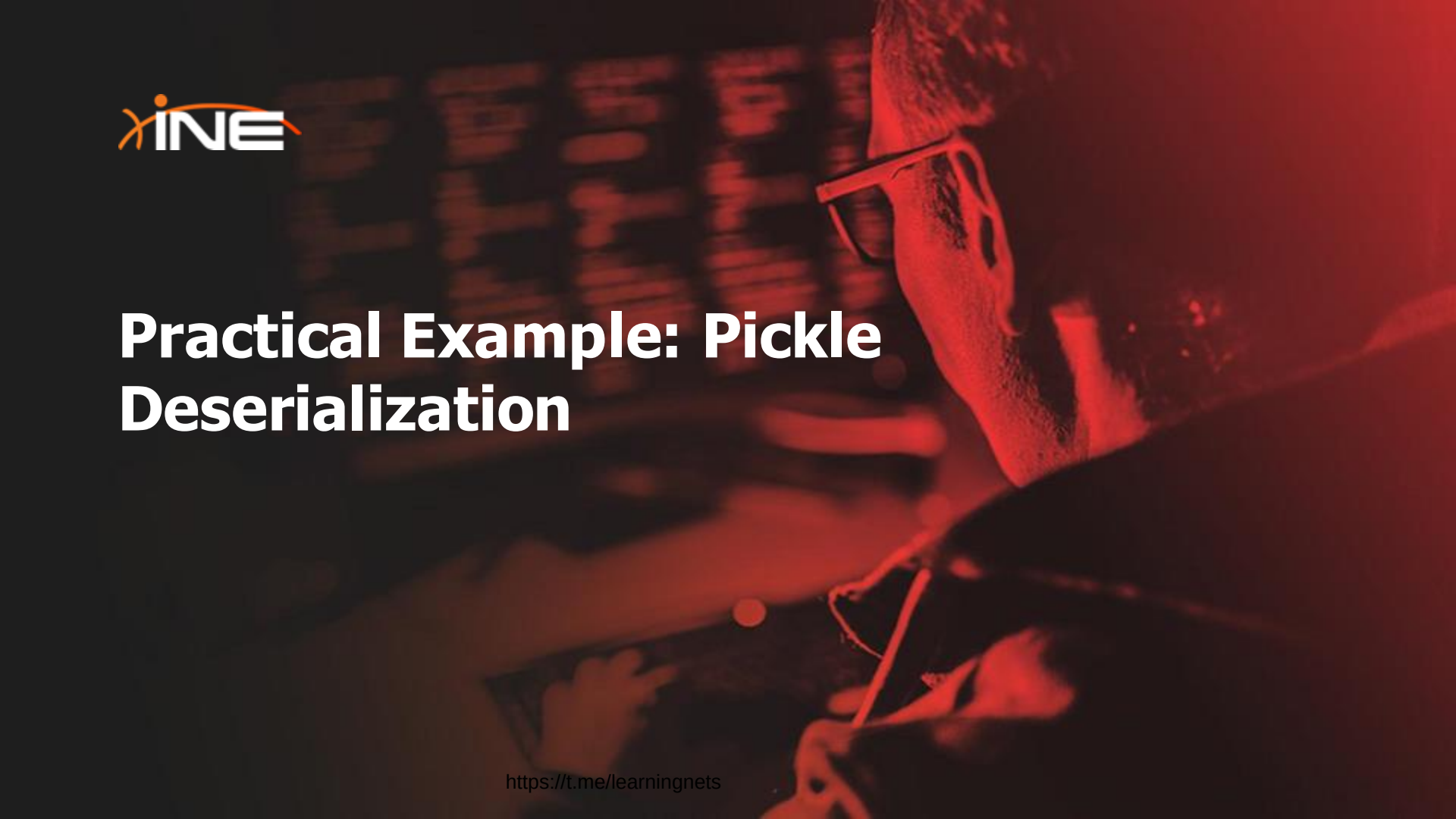
Remote Code Execution (RCE)
Attackers inject payloads that exploit insecure deserialization processes to execute arbitrary code on the server.
Example: Replacing serialized objects with ones containing malicious code.

Privilege Escalation:
Tampering with serialized data to escalate privileges or impersonate users by modifying access tokens or session objects.

Data Tampering
Changing serialized data values to manipulate application behavior, such as increasing account balances or bypassing restrictions.

Application Logic Abuse
Injecting objects that invoke unintended application functionality, such as triggering sensitive methods during deserialization.

Denial of Service (DoS)
Crafting serialized data to exhaust server resources during deserialization, causing application crashes.



Practical Example: Pickle Deserialization

Pickle Deserialization

Pickle is a Python module used for **serializing** and **deserializing** Python objects.

Serialization, in this context, refers to the process of **converting a Python object into a byte stream**, which can then be stored in a file, database, or transmitted over a network.

Deserialization is the reverse process—reconstructing the original Python object from the byte stream.

How Pickle Works

Serialization

- `pickle.dump(obj, file)` or `pickle.dumps(obj)` converts a Python object into a byte stream.

The resulting byte stream can be stored or transmitted.

Deserialization

- `pickle.load(file)` or `pickle.loads(data)` reads the byte stream and recreates the original Python object in memory.

Example

```
import pickle

# Example object
data = {"name": "Alice", "age": 30, "roles": ["admin",
"user"]}

# Serialize the object
serialized_data = pickle.dumps(data)
print("Serialized Data:", serialized_data)

# Deserialize the object
deserialized_data = pickle.loads(serialized_data)
print("Deserialized Data:", deserialized_data)
```

Demo: Pickle Deserialization

Java Insecure Deserialization

Java Serialization & Deserialization

In **Java**, serialization is the process of converting an object's state into a byte stream so it can be stored or transmitted.

The serialized data can later be deserialized to recreate the original object in memory. Java uses the **Serializable** interface to mark objects that can be serialized.

Java provides built-in support for serialization and deserialization through **`ObjectOutputStream`** and **`ObjectInputStream`** classes.

Java Serialization & Deserialization

Before we dig into exploiting insecure deserialization in Java, let's take a look at how to create a serialized object to understand the process better.

If you want to recreate this process, you will need a Java compiler and a text editor.

In order to get the Java compiler, install the latest **JDK (Java Development Kit) and JRE (Java Runtime Environment)** on your operating system.

Java Serialization & Deserialization

In this example, we will create two files:

- **Item.java** → Holds code for a class named “Item”.
- **Serialize.java** → Contains the serialization logic.

In order to use serialization, the program must import the **java.io.Serializable package**.

Moreover, for the class to be serialized, it must implement a **Serializable** interface.

Java Serialization & Deserialization

```
//Item.java
import java.io.Serializable;
public class Item implements
Serializable {
    int id;
        String name;
        public Item(int id, String
name) {
        this.id = id;
        this.name = name;

    }
}
```


Item.java is a simple class that has two fields: **id** and **name**.

Serializable classes can contain many fields and methods.

Now, we will use another file (in Java one class should be contained in one file) that will make use of that class.

Java Serialization & Deserialization

```
//Serialize.java
import java.io.*;
class Serialize{
 public static void main(String args[]){
  try{
  //Creating the object
  Item s1 = new Item(123,"book");
  //Creating stream and writing the object
  FileOutputStream fout = new
FileOutputStream("data.ser");
  ObjectOutputStream out = new ObjectOutputStream(fout);
  out.writeObject(s1);
  out.flush();
  //closing the stream
  out.close();
  System.out.println("Serialized data saved to
data.ser");
  }catch(Exception e){System.out.println(e);}
 }
}
```


First, it will create an instance of the Item class, and then serialize that instance as well as save it to a file.

The Serialize class makes use of the Item class, as it converts the instance of the Item class to a Stream of Bytes.

Then, it is saved to a file called “data.ser”.

Java Serialization & Deserialization

Before compilation, make sure that the two .java files are in the same directory.

```
root@0xluk3:~/java# javac Item.java Serialize.java
root@0xluk3:~/java# java Serialize
Serialized data saved to data.ser
root@0xluk3:~/java# cat data.ser
��srItemZ,��t��IidLnametLjava/lang/String;xp{tbookroot@0xluk3:~/java#
```

We can see that the saved data.ser file is in binary format. Apart from some strings that disclose what the serialized data might be, there are also some non-ASCII characters.

```
root@0xluk3:~/java# strings data.ser
ItemZ,
namet
Ljava/lang/String;xp
book
root@0xluk3:~/java# file data.ser
data.ser: Java serialization data, version 5
root@0xluk3:~/java# cat data.ser | hexdump -C
00000000  ac ed 00 05 73 72 00 04  49 74 65 6d 5a 2c 80 f7  |....sr..ItemZ,..|
00000010  74 f7 b8 1d 02 00 02 49  00 02 69 64 4c 00 04 6e  |t......I..idL..n|
00000020  61 6d 65 74 00 12 4c 6a  61 76 61 2f 6c 61 6e 67  |amet..Ljava/lang|
00000030  2f 53 74 72 69 6e 67 3b  78 70 00 00 00 7b 74 00  |/String;xp...{t.|
00000040  04 62 6f 6f 6b                                    |.book|
00000045
root@0xluk3:~/java#
```

Java Serialization & Deserialization

The file begins with the “ac ed 00 05” bytes, which is a standard java serialized format signature.

Wherever you see binary data starting with those bytes, you can suspect that it contains serialized java objects.

Java Serialization & Deserialization

```
//Deserialize.java
import java.io.*;
class Deserialize{
 public static void main(String args[]){
  try{
  //Creating stream to read the object
  ObjectInputStream in=new ObjectInputStream(new
FileInputStream("data.ser"));
  Item s=(Item)in.readObject();
  //printing the data of the serialized object
  System.out.println(s.id+" "+s.name);
  //closing the stream
  in.close();
  }catch(Exception e){System.out.println(e);}
 }
}
```


This code will retrieve the serialized data from the binary file.

The file will be named **Deserialize.java**.

Java Serialization & Deserialization

After compilation and running the Deserialize class, we can see that the object was properly reconstructed.

```
root@0xluk3:~/java# javac Deserialize.java
root@0xluk3:~/java# java Deserialize


123 book
root@0xluk3:~/java#
```

Insecure Deserialization Conditions

A potentially exploitable condition in Java occurs when **readObject()** or a similar function is called on user-controlled object and later, a method on that object is called.

An attacker is able to craft an object containing multiple, nested properties, that, upon being called, will do something completely different, e.g. ***hijack the called method by implementing a Dynamic Proxy and an Invocation handler in the serialized object's properties.***

Gadgets

Every property or method that is part of a nested exploit object is called a **gadget**.

There are some **specific Java libraries** that were found to contain some universal gadgets used to build **serialized exploit objects**.

These libraries are called **gadget libraries**. The concept of gadgets was first presented at the following talk.
https://frohoff.github.io/appseccali-marshalling-pickles/

Gadgets

There is a set of **common libraries** that have been identified as **gadget libraries**.

This does not mean that they are insecure by design. It only means that in the event insecure deserialization is performed while these libraries are loaded into the classpath of the running application, the attacker can abuse them to construct a known gadget chain that will result in successful exploitation.

For example, common libraries that were identified as vulnerable are **CommonsCollections** (versions 1-6). There is a powerful tool called **ysoserial** that can be used to perform exploitation of insecure java deserialization vulnerabilities. Ysoserial contains multiple modules that are suited to various Java deserialization exploitation scenarios.

Ysoserial

Ysoserial is a Java-based tool designed to generate **malicious serialized payloads** that can be used to exploit insecure deserialization vulnerabilities.

It leverages known vulnerabilities **in popular Java libraries** (e.g., Apache Commons Collections, Spring, etc.) to create payloads that execute arbitrary commands during deserialization.

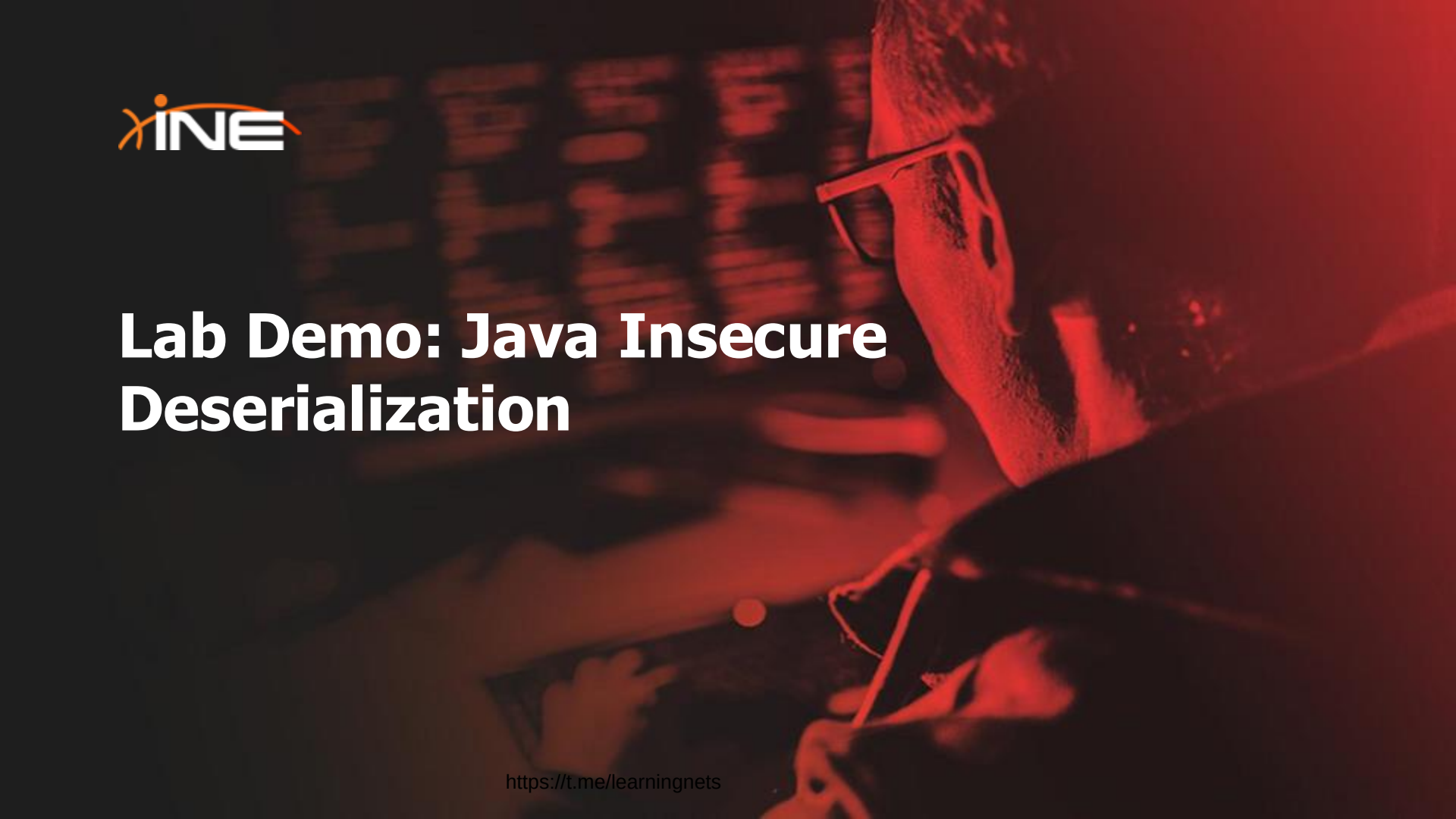
How Ysoserial Works

Ysoserial **creates serialized payloads** that **trigger** malicious behavior when deserialized.

```
java -jar ysoserial.jar CommonsCollections1 "whoami" > payload.ser
```

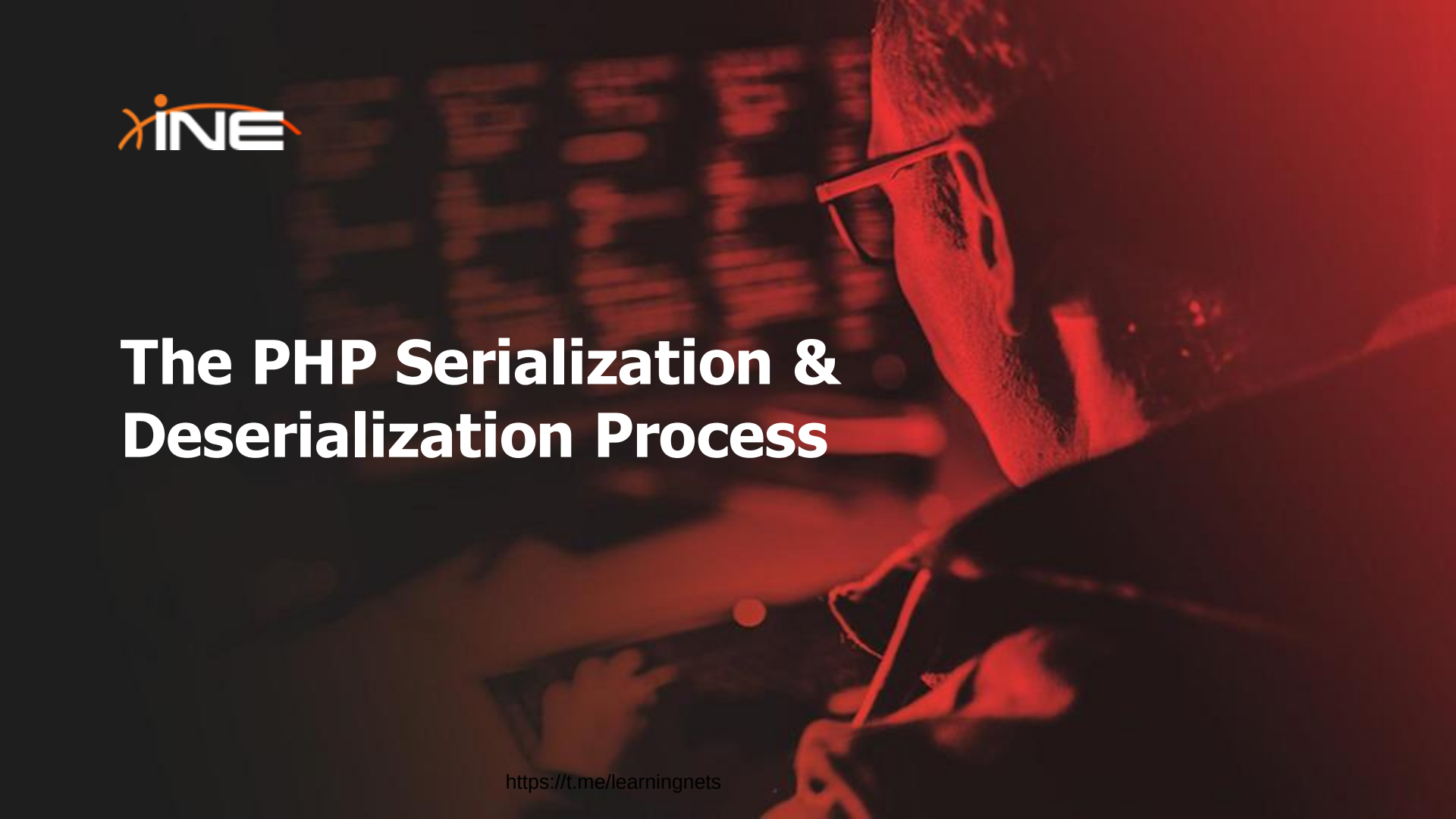
This command generates a payload that executes a system command (whoami) when deserialized.

The generated payload is sent to the vulnerable application through attack vectors such as cookies, HTTP requests, or files. When the application deserializes the payload, the command is executed.



Lab Demo: Java Insecure Deserialization

PHP Insecure Deserialization



The PHP Serialization & Deserialization Process

PHP Serialization

PHP Serialization is the process of converting a PHP object, array, or value into a string representation that can be stored, transmitted, or reused later.

The serialized string **contains all** the necessary information to reconstruct the original object or data structure.

The **serialize()** function handles serialization, and the **unserialize()** function handles deserialization.

PHP Serialization Process

The **PHP serialization process** is achieved using the **serialize()** function in PHP.

Serialization converts the object or data structure into a string format **following a strict syntax**. This serialized string contains:

- **Object type:** Identifies the data type (e.g., object, array, or scalar).
- **Length and metadata:** Includes lengths of names, values, or keys.
- **Properties and values:** All property names and their respective values are included in the serialized form.

PHP Serialization Syntax

The serialized string follows this general pattern:

<length> is the length of the class name.

<num_properties> is the number of properties in the object.

```
O:<length>:"<class_name>":<num_properties>:{<property_name>;<property_value>;}
```

O indicates the serialized type is an Object.

<class_name> is the name of the class being serialized.

<property_name> and **<property_value>** pairs represent the object's properties and values.

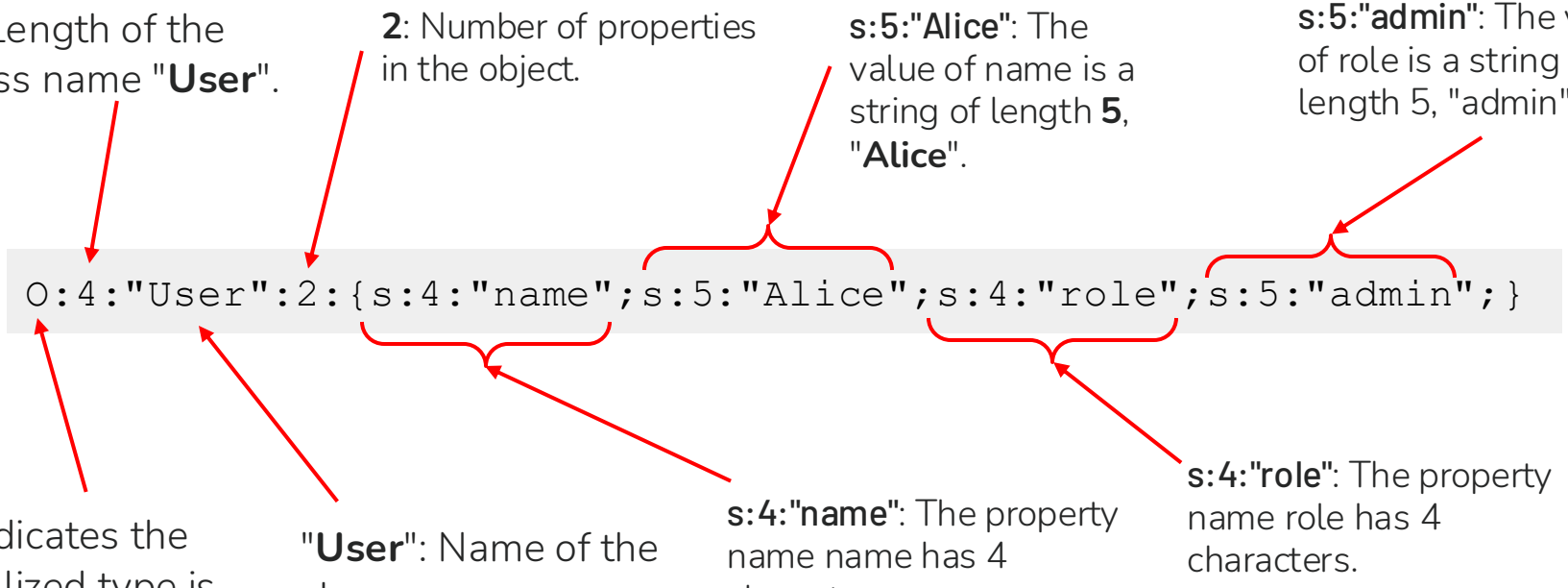
PHP Serialization Process

```php
<?php
class User {
    public $name;
    public $role;

    public function __construct($name, $role) {
        $this->name = $name;
        $this->role = $role;
    }
}

// Create a User object
$user = new User("Alice", "admin");

// Serialize the object
$serialized = serialize($user);
echo "Serialized Object: " . $serialized;
?>
```

The **serialize()** function converts PHP objects, arrays, or values into a storable string format.

PHP Serialization Example

The output of the **serialize()** function for the above code will look like this:



PHP Deserialization Process

The **unserialize()** function reconstructs the serialized string into the original object.

```php
<?php
$serialized =
'O:4:"User":2:{s:4:"name";s:5:"Alice";s:4:"role";s:5:"admin";}';
$user = unserialize($serialized);

echo "Name: " . $user->name . ", Role: " . $user->role;
?>
```

Output:

```
Name: Alice, Role: admin
```

Summary of the Process

Object Inspection: PHP inspects the object's class name, properties, and values.

Type Mapping: Each data type is encoded:

- O: Object
- s: String
- i: Integer
- b: Boolean
- a: Array

Length Tracking: The length of strings and arrays is calculated to ensure precise encoding.

Serialization String Generation: The object is transformed into the serialized string format, combining all metadata, property names, and values.

PHP Insecure Deserialization

PHP Insecure Deserialization

Insecure deserialization in **PHP** occurs when user-supplied serialized data is deserialized without proper validation. This can lead to unintended behaviors, such as code execution, data manipulation, or object injection.

How It Works

- A **PHP application** accepts serialized data from untrusted sources (e.g., cookies, forms, or HTTP requests).
- The application uses **unserialize()** to reconstruct objects without verifying the data's integrity.
- An attacker crafts a malicious serialized string that triggers unintended functionality during deserialization.

Causes of PHP Insecure Deserialization

- **Blind Trust in User Input**: Accepting serialized data from untrusted sources without verification.
- **Misuse of unserialize()**: Using **unserialize()** directly on user-supplied data.
- **Vulnerable Magic Methods**: Exploiting magic methods like **__wakeup()** or **__destruct()** for malicious code execution.
- **Lack of Input Validation**: Not verifying or sanitizing serialized input.

Magic Methods in PHP

Magic methods in PHP are special methods prefixed with __ that are automatically called during **certain object operations**. In the context of deserialization, the following magic methods are critical:

__wakeup(): Automatically invoked during deserialization.
Often used to reinitialize object properties.

__destruct(): Called when an object is destroyed (e.g., at the end of a script).

__toString(): Invoked when an object is treated as a string.

__sleep(): Invoked during serialization to clean up or prepare object properties.

__call(): Triggered when an undefined method is called.

Attackers exploit magic methods by embedding malicious logic in the serialized payload. When the application unserializes the payload, magic methods like __wakeup() or __destruct() execute automatically.

PHP Object Injection

PHP Object Injection is an attack vector that arises when user input is passed to **unserialize()** without proper validation.

By **manipulating** serialized objects, attackers can exploit **vulnerable magic methods** to execute arbitrary code, modify application behavior, or access sensitive data.

PHP Object Injection Example

```php
<?php
class User {
    public $name;
    public $role;

    public function __destruct() {
        if ($this->role === "admin") {
            echo "Access Granted! Admin
privileges.";
        }
    }
}

if (isset($_GET['data'])) {
    $data = $_GET['data'];
    $user = unserialize($data); // Vulnerable
}
?>
```

Here's an example of PHP code that is vulnerable to Object Injection

PHP Object Injection Example

Exploit: The attacker crafts a malicious serialized payload:

```php
<?php
class User {
    public $name = "EvilUser";
    public $role = "admin";
}

echo urlencode(serialize(new User()));
?>
```

Generated Payload:

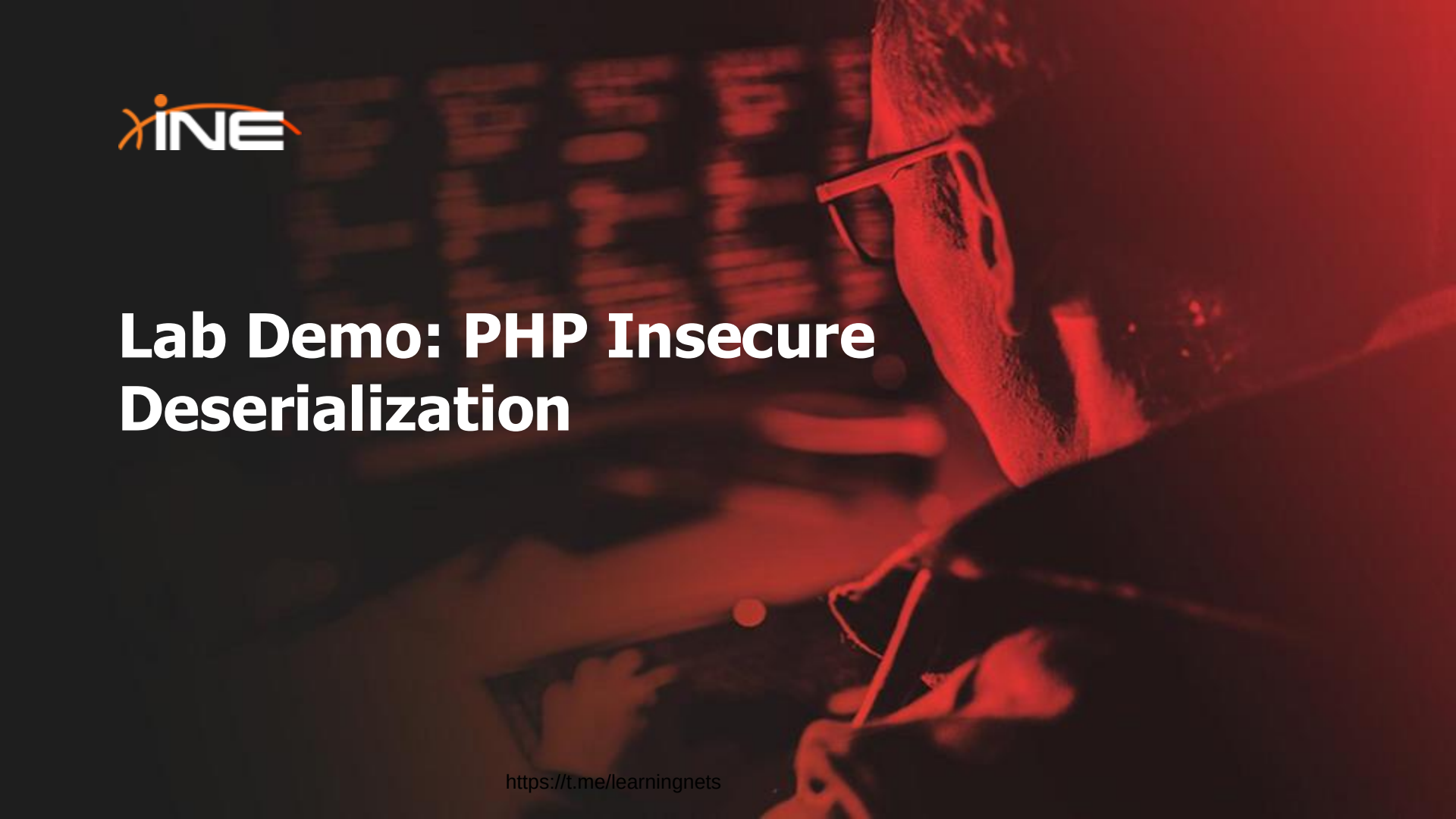
```
O:4:"User":2:{s:4:"name";s:8:"EvilUser";s:4:"role";s:5:"admin";}
```

PHP Object Injection Example

The attacker sends the serialized payload via **GET**:

```
http://example.com/vuln.php?data=O:4:"User":2:{s:4:"name";s:8:"Evil User";s:4:"role";s:5:"admin";}
```

- ➔ The **unserialize()** function reconstructs the object.
- ➔ The **__destruct()** magic method is triggered, **granting the attacker admin privileges.**



Lab Demo: PHP Insecure Deserialization

Server-Side Attacks

Course Summary

Key Concepts - Recap

- Modern Web Application Architecture
- Server-Side Request Forgery (SSRF)
- Insecure Deserialization

Learning Outcomes - Recap

+ **Understand Modern Web App Architecture:** Describe the components and layers of modern web application infrastructure and explain the role and interaction of the components that make up a web application.
+ **Recognize Server-Side Vulnerabilities:** Understand what server-side attacks are and how they exploit weaknesses in the back-end infrastructure.
+ **SSRF:** Define SSRF and explain its causes, attack anatomy, and variations. Demonstrate practical exploitation of SSRF vulnerabilities, including blind and advanced attack techniques.
+ **Insecure Deserialization:** Explain what insecure deserialization is and how it can lead to vulnerabilities like remote code execution (RCE).
+ **Exploit Language-Specific Deserialization Vulnerabilities:** Identify and exploit deserialization vulnerabilities in Java, PHP and .NET Applications..

Real-World Applications

- Prevalence in Real World Applications: SSRF is commonly found in applications integrating third-party APIs or processing user-supplied URLs and has been highlighted in vulnerability disclosures affecting major cloud providers (e.g., AWS and Google Cloud).
- Sophisticated Exploitation Techniques: This course provides you with the skills required to facilitate advanced attacks that are rarely standalone and are often used as entry points for advanced attack chains, for example; *SSRF -> Internal API -> Privilege Escalation*
- Custom Tooling & Payloads: This course provides you with practical knowledge and experience in developing custom payloads and tools that are required for the successful exploitation of vulnerabilities like Insecure Deserialization.

Next Steps

- Extend your learning by exploring other critical server-side vulnerabilities like SSTI.
- Practice chaining vulnerabilities for real-world scenarios, such as combining SSRF with privilege escalation or insecure deserialization with remote code execution.
- Build custom payloads for complex environments, such as deserialization in less-documented frameworks.
- Start applying your skills to bug bounty programs such as HackerOne, Bugcrowd, read reports and practice creating detailed reports, documenting your findings, and proposing actionable remediation steps for server-side vulnerabilities.

THANKS FOR WATCHING!

EXPERTS AT MAKING YOU AN EXPERT