Introduction of Command Injection and Practice

Student Jeongmin Lee

Introduction

This report covers command injection attacks, referencing lecture materials and YouTube videos provided in class. Where necessary, additional information was supplemented from online resources. The goal of this report is to provide an in-depth understanding of the concept, types, real-world cases, and best practices for preventing command injection attacks.

What is a Command Injection Attack?

A command injection attack is a security vulnerability that allows an attacker to execute arbitrary system commands by exploiting weaknesses in a web application's input handling. This type of attack occurs when user input is not properly validated or filtered. Attackers can input malicious commands to perform unintended actions on the system or steal sensitive information.

Command injection is similar to SQL injection attacks. However, the main difference is that SQL injection targets databases by injecting SQL commands, while command injection allows execution of system-level commands on the host operating system. In other words, SQL injection manipulates SQL queries to target the database, whereas command injection exploits vulnerabilities that permit arbitrary command execution at the OS level.

In the worst case, an attacker can exfiltrate information from the system or achieve Remote Code Execution (RCE). This enables the attacker to add a reverse shell to the system, gain full control, and execute any command on the web server.

Simple Example of Command Injection

For example, let's assume the server executes a ping command using the IP address provided by the user.

- Normal Input: `ping 127.0.0.1`
- Malicious Input: `127.0.0.1; cat /etc/passwd`
- Actual Command Executed: `ping 127.0.0.1; cat /etc/passwd`

In this case, the server not only executes the ping command but also the additional `cat /etc/passwd` command, potentially exposing the contents of the system's password file to the attacker. Below is an example of vulnerable code written in Python.

```
user_input = request.args.get("ip")  # e.g., "127.0.0.1; cat /etc/passwd"
cmd = "ping␣" + user_input
os.system(cmd)  # Executes the entire command via shell
```

Listing 1: Vulnerable Code Example — Directly Combining User Input into Ping Command

Types of Command Injection Attacks

Command injection attacks can be broadly divided into three types:

First, **In-band Command Injection** refers to cases where the output or result of the injected command is directly included in the application's response (such as HTTP Response). This type allows attackers to quickly and easily confirm information (such as directory listings, file contents, etc.). The simple example discussed earlier falls into this category.

Second, **Blind Command Injection** refers to cases where the command is executed on the server, but the result (output) is not directly returned in the HTTP response. Attackers use various techniques to infer the success/failure of the command or obtain system information without direct output. Because the result is not visible in the response body, it is called 'blind.'

Third, **Out-of-Band (OOB) Command Injection** refers to cases where the output of the command is sent to a separate channel controlled by the attacker (such as DNS, HTTP callback, email, etc.), and the attacker checks the response through that channel. Some sources consider OOB a subcategory of Blind, while others treat it as a separate type. In this report, it is classified as a separate type.

Impact of Command Injection Vulnerabilities

Command injection vulnerabilities can have very serious impacts on systems and services. The main impacts and their severity are as follows:

- **Sensitive Information Disclosure**: Attackers can read or exfiltrate important files (e.g., /etc/passwd, configuration files, etc.) on the server through system commands. This can lead to the exposure of personal information, authentication credentials, system settings, and more.
- **Remote Code Execution (RCE) and Privilege Escalation**: Attackers can execute arbitrary commands to install malware (backdoors, reverse shells, etc.) on the system or escalate privileges to gain full control over the server. This can cause severe damage to service operators.
- **Denial of Service (DoS) and System Disruption**: If attackers execute infinite loops, large file creations, or resource-consuming commands, it can lead to service interruptions or server crashes.
- **Data Integrity/Availability/Confidentiality Compromise**: File deletions, data tampering, and system configuration changes can all threaten the integrity, availability, and confidentiality of data.
- **Stepping Stone for Attacking Other Systems**: Through command injection, attackers can use the server as a foothold to further attack other systems on the internal or external network. For example, network scanning and additional malware distribution are possible.
- **Legal/Trust Issues**: If information leaks or service disruptions occur, companies or organizations may face legal liabilities, reputational damage, and loss of trust.

In this way, command injection vulnerabilities can seriously threaten not only the security and stability of the system as a whole but also the trustworthiness of the organization, so prevention and prompt response are essential.

Prevention and Defense Strategies

The key to preventing command injection vulnerabilities is to **avoid directly invoking shell commands of the operating system**. Whenever possible, application logic should not pass through the shell, and only safe APIs or dedicated functions should be used. Additional defense techniques include:

- **Input Validation and Sanitization**: Strictly validate user input and normalize it to prevent unexpected metacharacters. Limit allowed character sets using regular expressions or schemas.
  - **Whitelist-based Validation**: Only accept permitted values or patterns. For example, IPv4 addresses should only allow digits and dots (`.`).

```
import re, subprocess

def is_valid_ipv4(ip):
  pattern = r"^([0-9]{1,3}\.){3}[0-9]{1,3}$"
  return re.match(pattern, ip) is not None

user_input = request.args.get("ip")
if is_valid_ipv4(user_input):
  subprocess.run(["ping", user_input], check=True)  # shell=False
else:
  raise ValueError("Invalid␣IP␣address")
```

Listing 2: Python Example: IPv4 Input Validation and Safe Execution

- **Use Safe APIs and Library Functions**: Instead of composing system calls as strings, use safe function calls that only perform the intended operation. For example, use `os.mkdir()` for directory creation.

```
import os

# Vulnerable way (allows command injection)
user_input = "new_directory;␣rm␣-rf␣/"  # Malicious input
os.system("mkdir␣" + user_input)

# Safe way (only creates directory)
os.mkdir(user_input)
```

Listing 3: Unsafe vs. Safe Method

- **Principle of Least Privilege**: Do not run applications with unnecessarily high privileges (e.g., root). Minimize the privileges of the execution account to reduce the impact even if an attack succeeds.
- **Security Options and Frameworks**: Use the `shell=False` option in Python's `subprocess` module, or argument-list-based execution in other languages to avoid shell interpretation. Apply modern security frameworks or libraries for systematic input validation, logging, and error handling.

```
import re
import subprocess

def is_valid_ipv4(ip: str) -> bool:
  pattern = r"^([0-9]{1,3}\.){3}[0-9]{1,3}$"
  return re.match(pattern, ip) is not None

user_input = request.args.get("ip", "")

if not is_valid_ipv4(user_input):
  raise ValueError("Invalid␣IP")

subprocess.run(["ping", "-c", "4", user_input], check=True, shell=False)
```

Listing 4: Python: Using subprocess.run (shell=False) with Input Validation

- **Output Encoding**: When returning command execution results or error messages to users in HTML, JSON, etc., always encode them properly. This provides an additional security layer to prevent command injection from combining with other vulnerabilities such as XSS.

Practical Application

I conducted hands-on exercises on command injection attacks using the labs at `https://portswigger.net/web-security/all-labs#os-command-injection`.

During the exercises, I added further explanations for aspects not covered above.

Lab: OS Command Injection, Simple Case

Lab Description : This lab covers an OS command injection vulnerability in a product stock checker application.

Key Points
- The application executes shell commands that include user-supplied product ID and store ID.
- The raw output of the shell command is directly returned in the HTTP response.
- Goal: Exploit the vulnerability to execute the `whoami` command and identify the current user.

Lab: Blind OS Command Injection with Time Delays

Lab: Blind OS Command Injection with Output Redirection

Lab: Blind OS Command Injection with Out-of-Band Interaction

Lab: Blind OS Command Injection with Out-of-Band Data Exfiltration