Command Injection

Talha Eroglu, Okan Yildiz

Abstract

OS Command Injection is a type of cyber attack that involves injecting malicious code into a legitimate system command. This allows the attacker to gain unauthorized access to sensitive information or to manipulate system functions. The attack is possible when an application does not properly validate user input, allowing the attacker to insert arbitrary commands that are executed by the operating system.

OS Command Injection attack can include data loss, system compromise, and loss of trust in the affected organization. In some cases, the attack can even lead to financial losses or legal repercussions.

This article discusses Command Injection attacks, what vulnerable applications are and how to exploit the vulnerability, and finally, how to prevent attacks by writing safe codes.

Contents

1. What is Command Injection ?

Command injection is a security vulnerability that allows an attacker to execute arbitrary operating system (OS) commands on a server that is hosting an application. Shell injection or OS command injection are other names for command injection.

Before deeply diving into 'Command Injection', we can clarify the distinction between Command Injection and 'Code Execution' (Code injection) Any attack involving the injection of code that an application interprets or executes is called "code execution." The misuse of untrusted data inputs is exploited in this kind of attack. The absence of adequate input/output data validation makes it possible. Code execution attacks have the critical limitation of being bound to the application or system they target. For example, if we ran a successful code execution attack in the test environment, the permissions granted to Python on the host machine would limit us. Let's talk about command injection, which is the subject of our article, Executing commands in a system shell or other areas of the environment is a standard definition of command injection. Without injecting malicious code, the attacker increases a vulnerable application's default functionality by passing commands to the system shell.

Insufficient input validation leads to OS Command Injection attacks; however, they are only possible if the application code includes operating system calls that have user input combined with the invocation.

This vulnerability can occur in every computer program and on any platform. Any language that executes a command through the system shell is susceptible to injection, which can be orchestrated on Windows and Unix systems. For instance, you can find command injection vulnerabilities in router-embedded software, online applications and APIs, server-side scripts, mobile apps, and even the core operating system software.

A command injection can compromise the infrastructure of that program, its data, the entire system, associated servers, and other devices.

2. Possible consequences of command injection

- Network traffic from the target system can be redirected to another system under the attacker's control, allowing them to intercept and see private information.
- Security measures like firewalls and intrusion detection/prevention systems (IDS/IPS) can be avoided using command injection.
- The target system may be completely compromised if an attacker can run any code on it.
- Using command injection, an attacker can change direction and target different systems connected to the same network as the vulnerable machine.
- Attackers can alter data on the target system, which might result in lost or inaccurate data.
- By injecting malicious commands that use up all the resources available, an attacker can cause a Denial of Service (DoS) to the target system.

3. How to exploit example vulnerabilities ?

As seen in Figure 1, a web application runs on a virtual machine, accessed from the localhost:8000 port. Functionalities in the application were implemented using different libraries and methods. These buttons call OS commands on the host server and provides output to the user.

First, we will discuss using suitable payloads for vulnerability detection in various implementations. Payloads vary by host operating system, but in this article, we will only cover Linux and unix-like operating systems. However, since the basic concept is the same, you can access different payloads over the internet. Then, after ensuring the vulnerability exists for each sample, we will see how we can practically test payloads using Burp Suite. Finally, we will discuss Pseudo-terminal, Reverse TCP Shells and summarize the exploitation part.

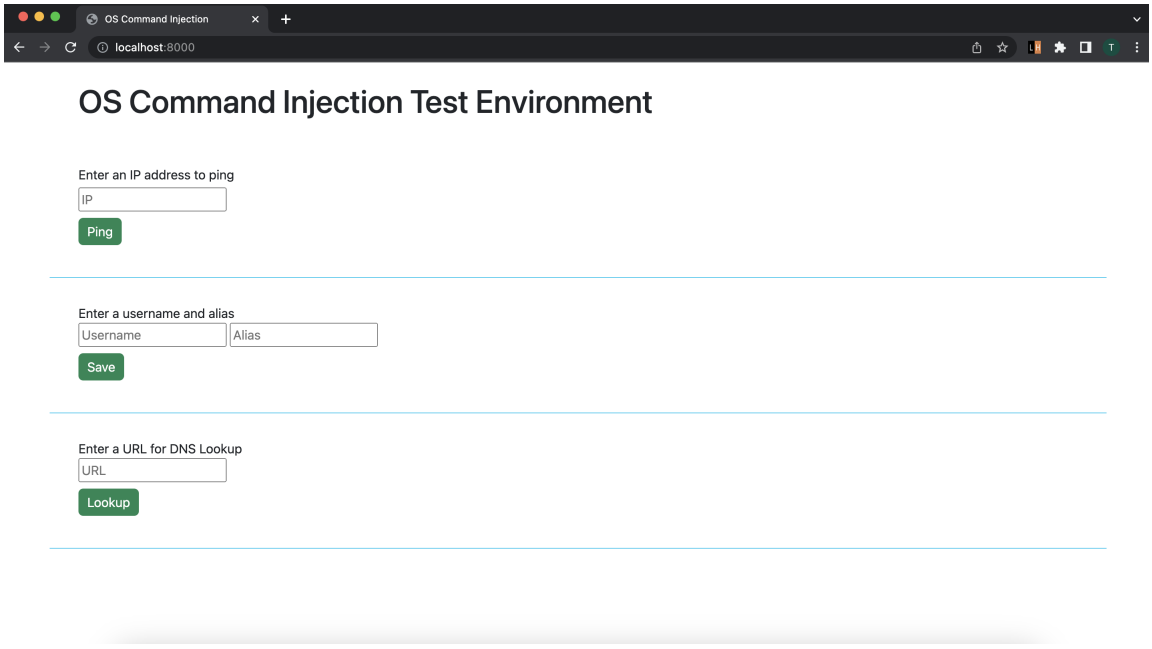

Figure 1: Web application developed in Django

3.1. Detecting vulnerability in first form

When we enter an IP address using the first button for its intended purpose, we encounter an expected output, as in Figure 2.

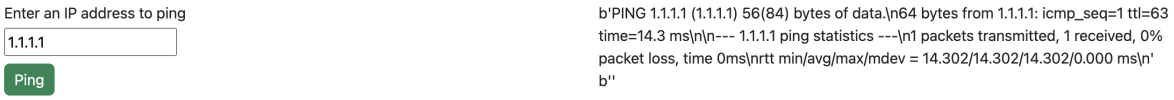

Figure 2: Submitting IP with Ping button

Then we write '; whoami' to the search query to check if we can run extra OS Commands, suspecting that user-supplied input is not sanitized. In the red framed output section in Figure 3, we see a new field consisting of a 'dev.' This is because we bypassed the IP field with ';' and executed the 'whoami' instruction in the host machine.

Figure 3: IP

3.2. Detecting vulnerability in alternative implementation of first form

Now we have another implementation of ping functionality in our web application. When we try to use '; unixcommand ' we confront with error message, 'Invalid IP address form'.In Figure 4.

We may be dealing with some input sanitizing here. Therefore, we need to

Figure 4: Error message when trying to inject commands

bypass this control by changing the inputs accepted by the system. So, first, we try the valid input form; after changing the inputs step by step, we see that the commands added to the end of the proper IP addresses are not stuck in this control. We successfully printed out working directory with 'pwd' command. In Figure 5.

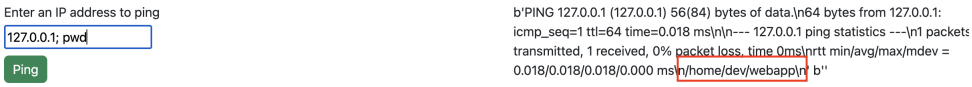

Figure 5: Succesfully bypassing input check

3.3. Detecting vulnerability in second form

Figure 6: Save form

First, we can understand the save form and what it does in the background. With two inputs, it gives the output as "Your alias saved successfully." Nothing changes if we try to give OS commands to the Username field (Figure 7).

Figure 7: Consistent output

But when we use the Alias field for the commands, we also get 'test' in addition to the success message. We tried to read the passwords in the etc/host/ directory, and although we did not have a decent output, the change in the output gave us a clue that something was wrong here. (Figure 8)

Figure 8: Changed output

Based on the previous tip, a string manipulation might exist. When we try to execute our command between semi-colons, we see that we can inject the command in Figure 9. We can also get working directory and search for other user inputs. Figure 10

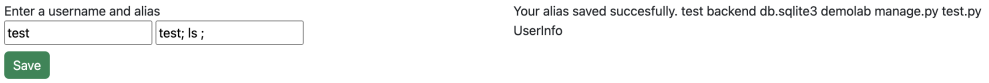

Figure 9: Successfully injecting ls command

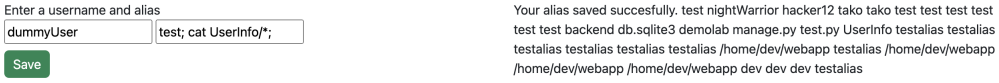

Figure 10: Accessing saved user information with cat command

3.4. Detecting vulnerability in last form

We can also execute OS commands by manipulating the URL area in the same way as in other examples.

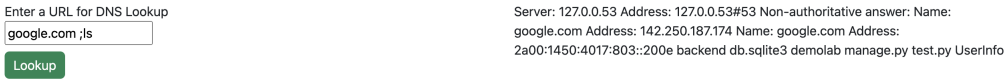

Figure 11: Executing OS commands on DNS Lookup form

3.5. Burp Suite

We will use the Ping form while showing the Burp Suite usage step by step.

- Open web page and ensure under the Proxy tab, Intercept is on
- Give an IP address as input in our web page and submit.
- Burp Suite will intercept our request, right click on the Intercepted Raw request and Send To Intruder.

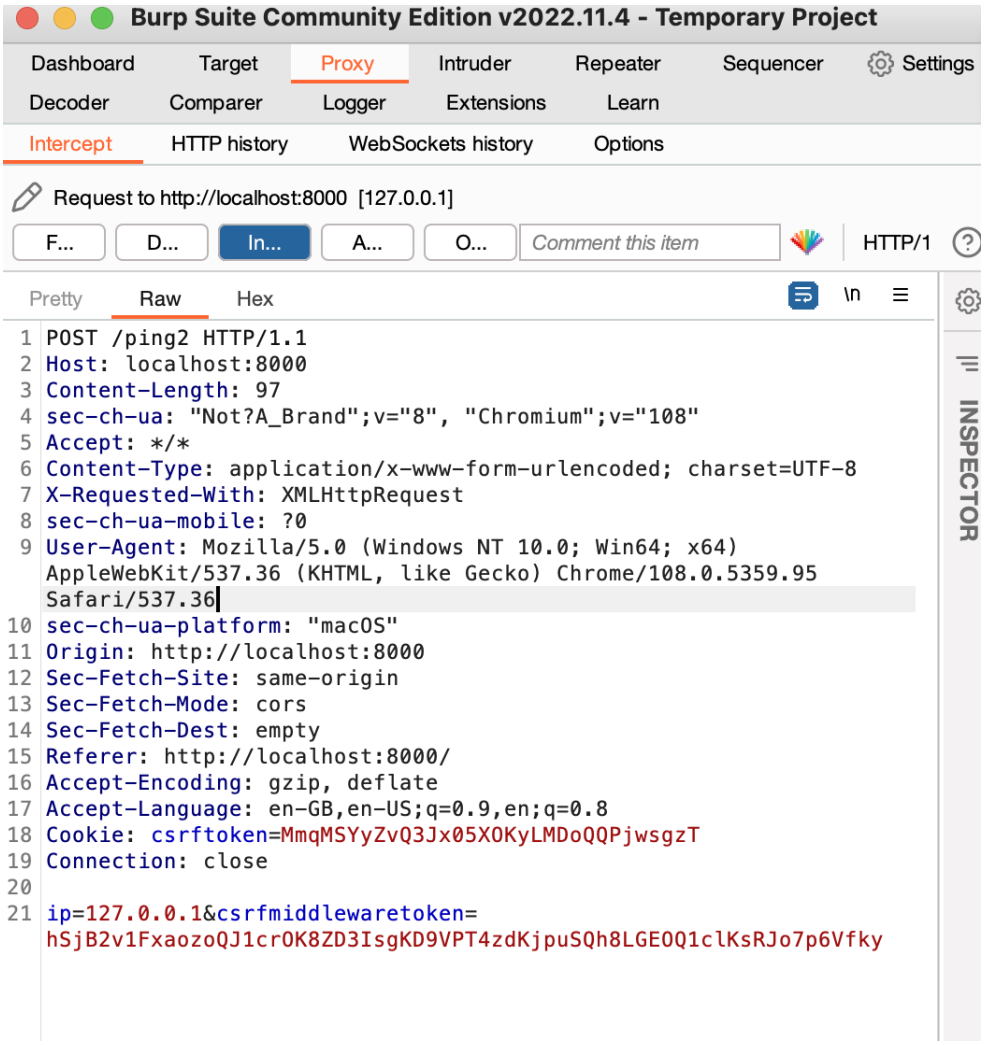

Figure 12: Intercepted request

- Clear unrelated payload positions with buttons on the right such that we will only have our IP field as payload position.

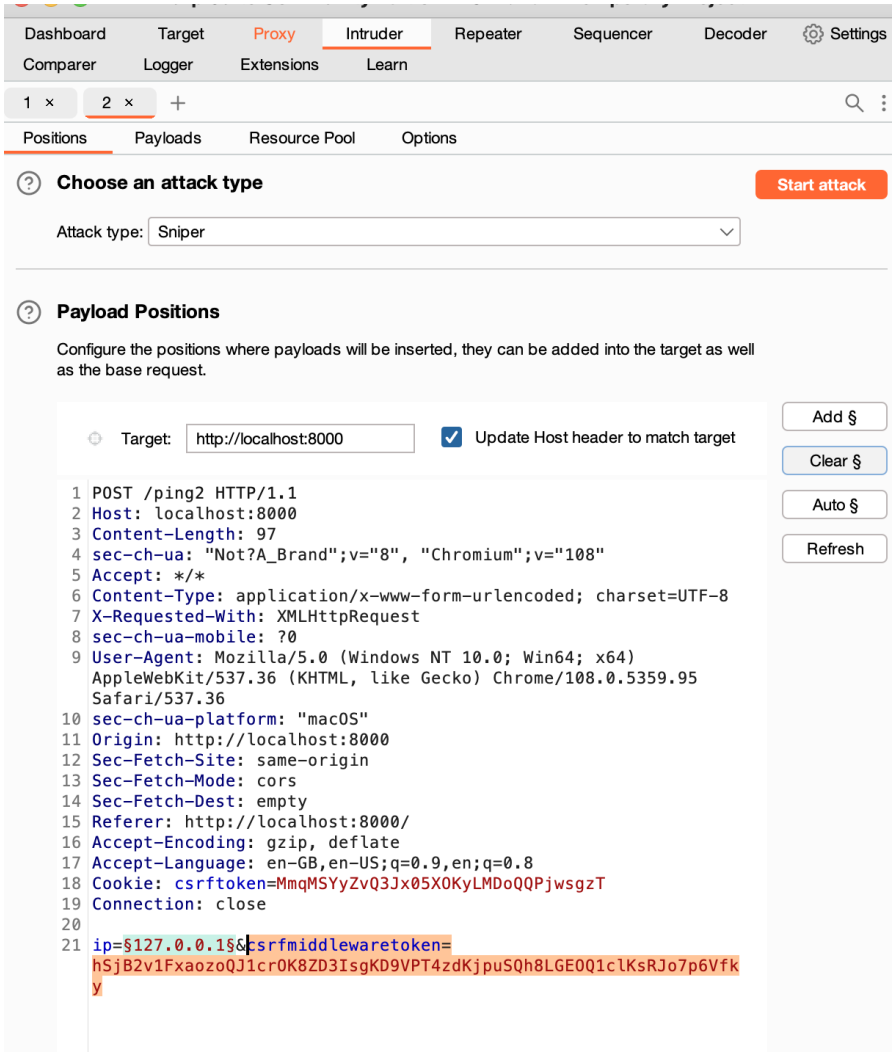

Figure 13: Clearing Payload Positions

- Go to Payloads, under Payload Options, load your payload and start the attack.

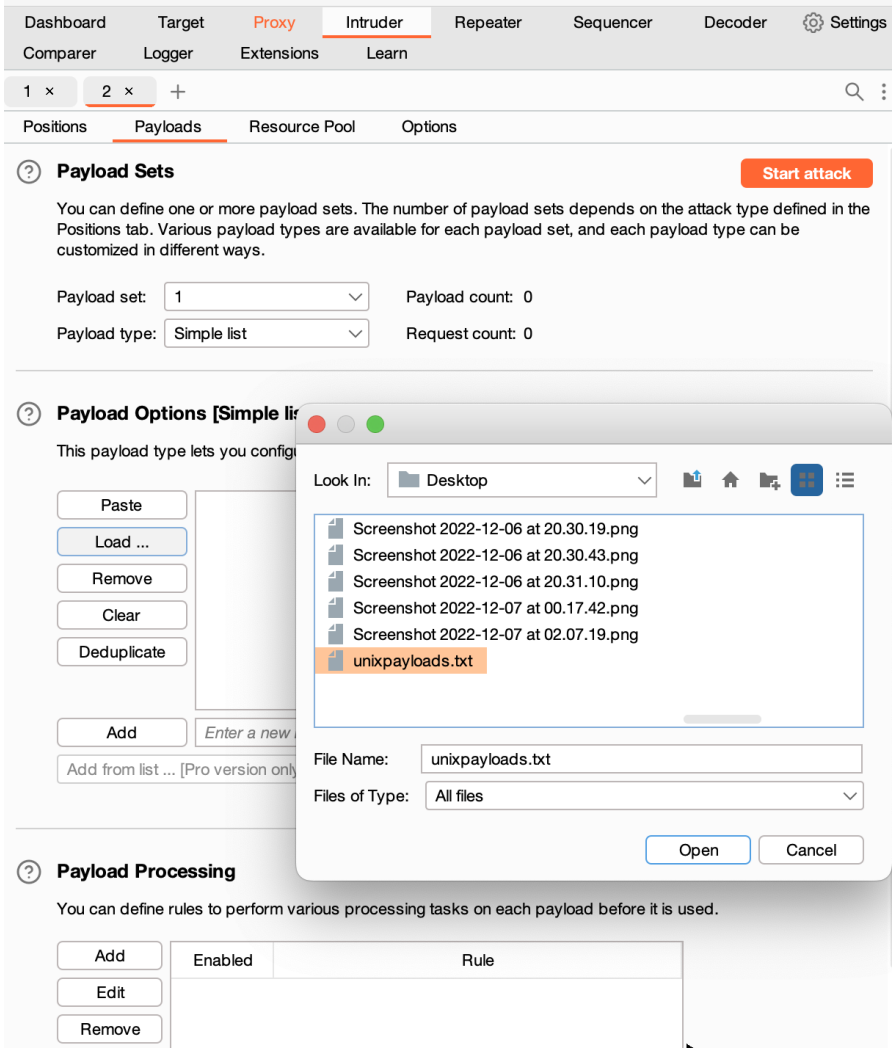

Figure 14: Loading payload file

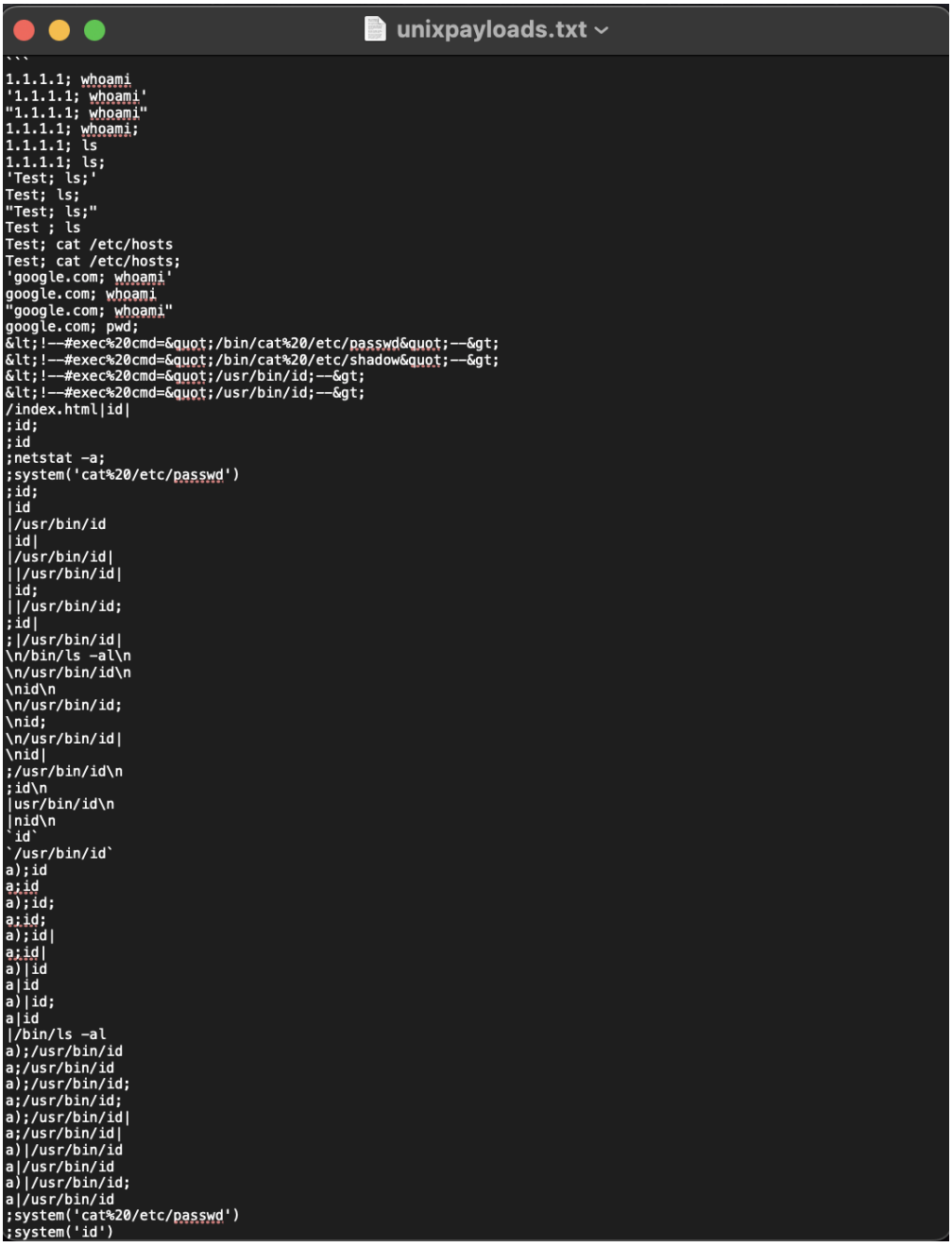

Figure 15: Preview of the payload file.

- As seen in the Figure 16, it can be determined which command is working, and you can examine the responses.

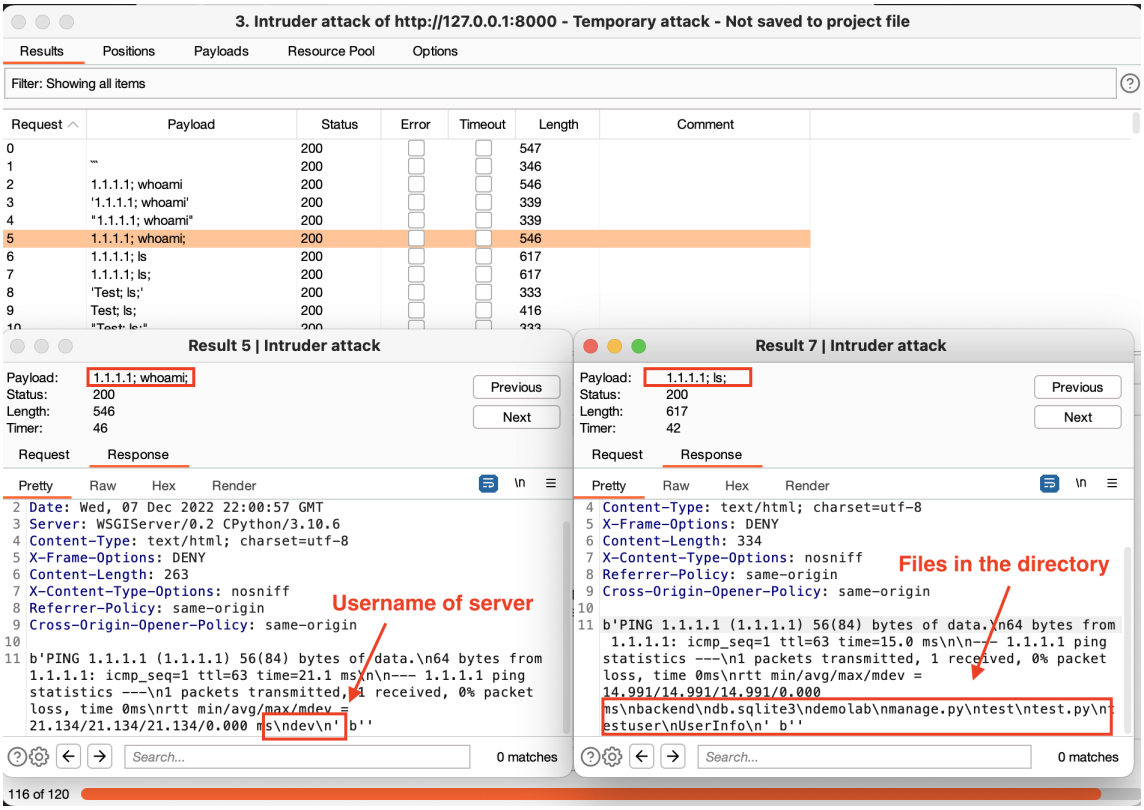

Figure 16: Attack requests and responses

3.6. One step further, Reverse TCP Shell

Using Commix (Command Injection Exploiter), we can open a pseudo-shell and then configure a reverse TCP shell on the target machine with the raw intercept file we got from Burp Suite. A reverse TCP shell is a type of shell in which the target machine opens a connection back to the attacking machine. This allows the attacker to control the target machine and access its files, execute commands, and more. It is called "reverse" because it is the opposite of the more common "forward" shell, in which the attacker connects to the target machine from their own. It provides the attacker with a wide range of capabilities, including the ability to run arbitrary code, collect system information, and escalate privileges.

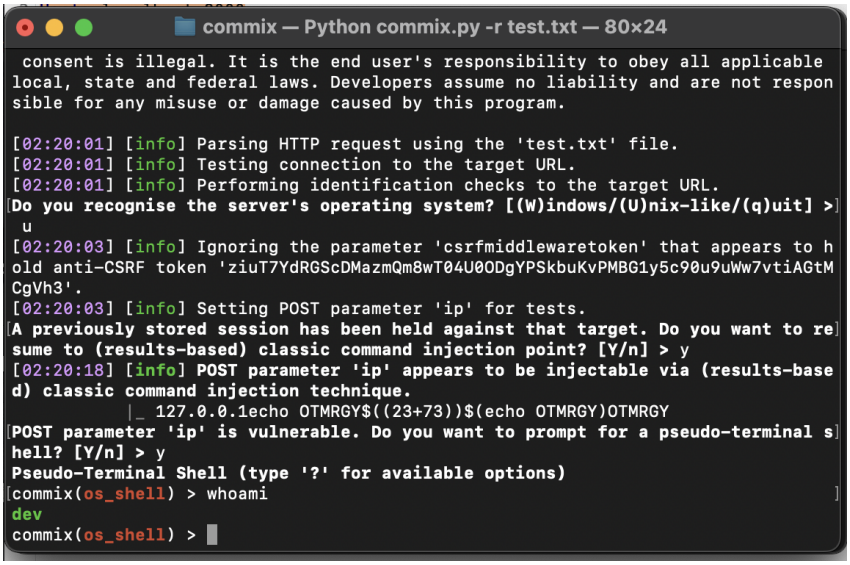

Figure 17: Pseudo-Terminal in our test environment

4. Vulnerable code examples

In this part, we will discuss vulnerable code examples and codes we exploited in the previous section. Like other injection attacks, unsanitized user input makes command injection possible. And this is irrespective of the programming language used. Code examples will be written in Python.
The figures below contain the vulnerable codes we exploited in the previous section.

```python
def ping(request):
    out = ''
    if request.POST:
        ipAddress = request.POST['ip']
        cmd = subprocess.run(f'ping -c 1 {ipAddress}', stdout=subprocess.PIPE, stderr=subprocess.PIPE, shell=True)
        out = f'{cmd.stdout} {cmd.stderr}'
    return HttpResponse(out)
```

Figure 18: First Ping

```python
def ping2(request):
    out = ''
    if request.POST:
        ipAddress = request.POST['ip']
        if not re.match("\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}", ipAddress):
            return HttpResponse("Invalid IP address form")


        cmd = subprocess.run(f'ping -c 1 {ipAddress}', stdout=subprocess.PIPE, stderr=subprocess.PIPE, shell=True)
        out = f'{cmd.stdout} {cmd.stderr}'
    return HttpResponse(out)
```

Figure 19: Second Ping

```
def lookup(request):
    out = ''
    if request.POST:
        hostname = request.POST['domain']
        cmd = 'nslookup ' + hostname
        out = subprocess.check_output(cmd, shell=True)
    return HttpResponse(out)
```

Figure 20: DNS Lookup

```
def nick(request):
    if request.POST:
        alias = request.POST['alias']
        username = request.POST['username']
        out = os.popen(f'echo {alias} >> {os.getcwd()}/UserInfo/{username}.txt').read()
    return HttpResponse(f'Your alias saved succesfully.\n{out}')
```

Figure 21: Nickname save

5. How to prevent command injection ?

In this part, we will discuss how to prevent command injection, discuss general techniques, and finally share the fixed version of the code we exploit.

- Validate and sanitize user input: This includes evaluating all user input to ensure it is in the desired format and free of potentially harmful characters or commands. This can assist in avoiding dangerous code injection into your applications by attackers.
- Avoid using OS Commands: Applications should not include codes that utilize operating system commands fed with user-controllable data. There are nearly always safer alternatives for using server-level instructions that attackers can't manipulate into doing anything other than what they were programmed to do.
- Use an up-to-date, secure web application framework: Modern web application frameworks have built-in protection against frequent injection

attacks. To stop these attacks, ensure you are utilizing an up-to-date and secure framework.

- Use WAF: By filtering incoming traffic and preventing requests that contain harmful code or commands, a web application firewall (WAF) can help to safeguard your applications. This may add another level of defense against command injection attacks.

- Lastly, informing your users about the dangers of injection attacks and the necessity of only entering reliable information into your applications is crucial. This can lessen the chance of attackers taking advantage of unsuspecting users.

To avoid command injection, Use an internal API whenever possible (if one exists) rather than an OS command. Avoid passing user-controlled input wherever possible, and if you have to, use an array with a sequence of program arguments rather than a single string. To do this in Python, we must set the subprocess's shell flag to false. (shell=False'). Because shell=True propagates existing shell settings and variables, using it is risky. Executing OS commands with shell=True makes it considerably more straightforward for a malicious actor to execute commands since variables, glob patterns, and other unique shell features in the command string are processed before the command is launched. On the other hand, when using particular shell features, such as word splitting or argument expansion, shell=True can be useful. If such a feature is necessary, use the various modules available to you instead, such as os.path.expandvars() for parameter expansion or shlex for word splitting. More work is required, but other issues are avoided.

In our code, for the ping function, first, we disabled the shell flag for the subprocess.run command. After we added a function to use the ipaddress library, Our IP address is not actually being verified by the ipaddress module. It attempts to generate a Python IP address object using the supplied string. We must develop our IP validation function to apply this reasoning to an IP.

After implementing the validate_ip function, we add a tested, secure regex to ensure only if the input is a valid IP address.

```python
def validate_ip(address):
    try:
        socket.inet_aton(address)
    except:
        False
    return True

def ping2(request):
    out = ''
    if request.POST:
        ipAddress = request.POST['ip']
        if not(validate_ip(ipAddress) and re.match("^((25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\\.){3}(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)$", ipAddress)):
            return HttpResponse("Invalid IP address form")

        cmd = subprocess.run(['ping', '-c', '1', ipAddress], stdout=subprocess.PIPE, stderr=subprocess.PIPE, shell=False)
        out = f'{cmd.stdout} {cmd.stderr}'
    return HttpResponse(out)
```

Figure 22: More secure ping function

In DNS lookup function, we added nslookup library and avoided usage of OS commands.

```python
from nslookup import Nslookup
def lookup(request):
    out = ''
    if request.POST:
        domain = request.POST['domain']
        if not re.match("^((?!-))(xn--)?[a-z0-9][a-z0-9-_]{0,61}[a-z0-9]{0,1}\.(xn--)?([a-z0-9\-]{1,61}|[a-z0-9-]{1,30}\.[a-z]{2,})$", domain)
            return HttpResponse('Not a valid domain.')
        dns_query = Nslookup()
        ips_record = dns_query.dns_lookup(domain)
    return HttpResponse(f'{ips_record.response_full} \n{ips_record.answer}')
```

Figure 23: More secure DNS Lookup function

Also in nick function, we used write() and open() instead of OS commands.

```
def nick(request):
    if request.POST:
        alias = request.POST['alias']
        username = request.POST['username']
        out =''
        try:
            f = open(username, "a")
            f.write(alias)
            f.close
            out =  "Your alias saved succesfully"
        except:
            out = f'Error: {sys.exc_info()}'

    return HttpResponse(out)
```

Figure 24: More secure nick function

6. Attempt to exploit more secure code

Since we have provided more secure implementation, we can test using the Burp suite as we did in the 3rd section by combining open-source payload examples and payloads that we used previously. Figure 25

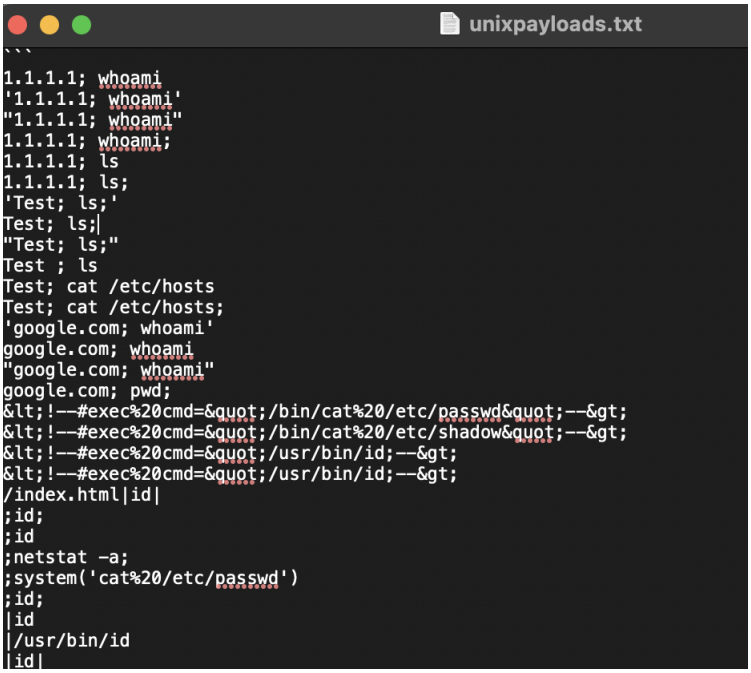

Figure 25: Payloads

First, we turn intercept on and send our request from the web page. After loading payloads, Figure 26, we start an attack on different forms in our web application.

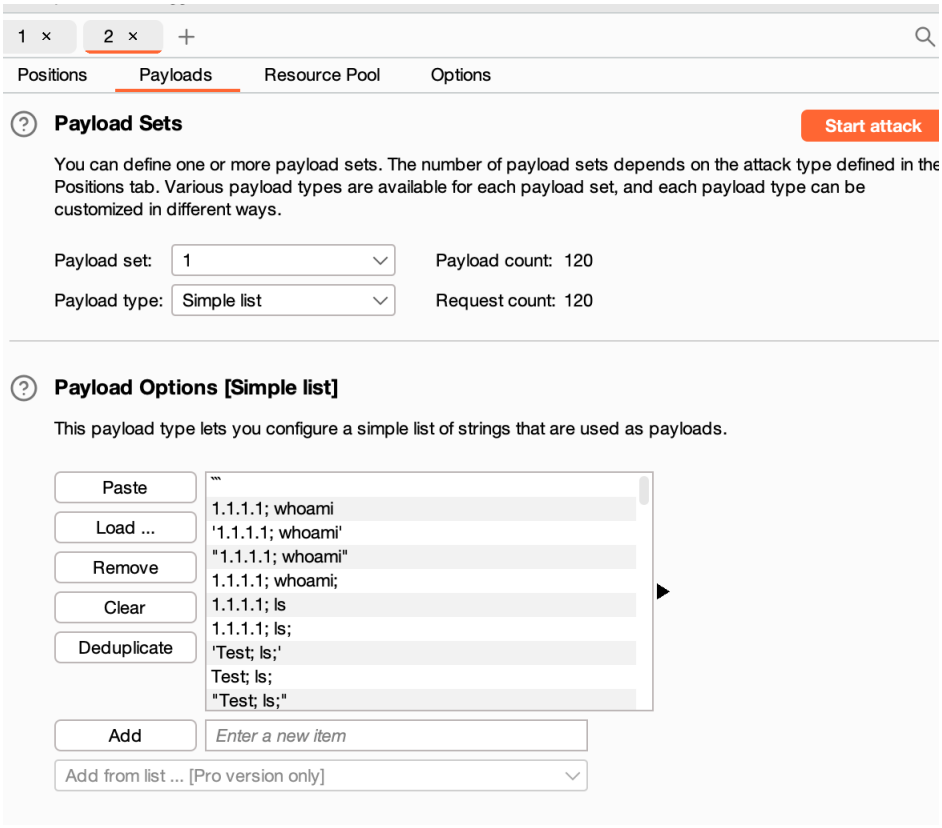

Figure 26: Starting attack with payloads

Finally, we can inspect the responses of different payloads and ensure that we did manage to prevent injecting commands to the host OS.

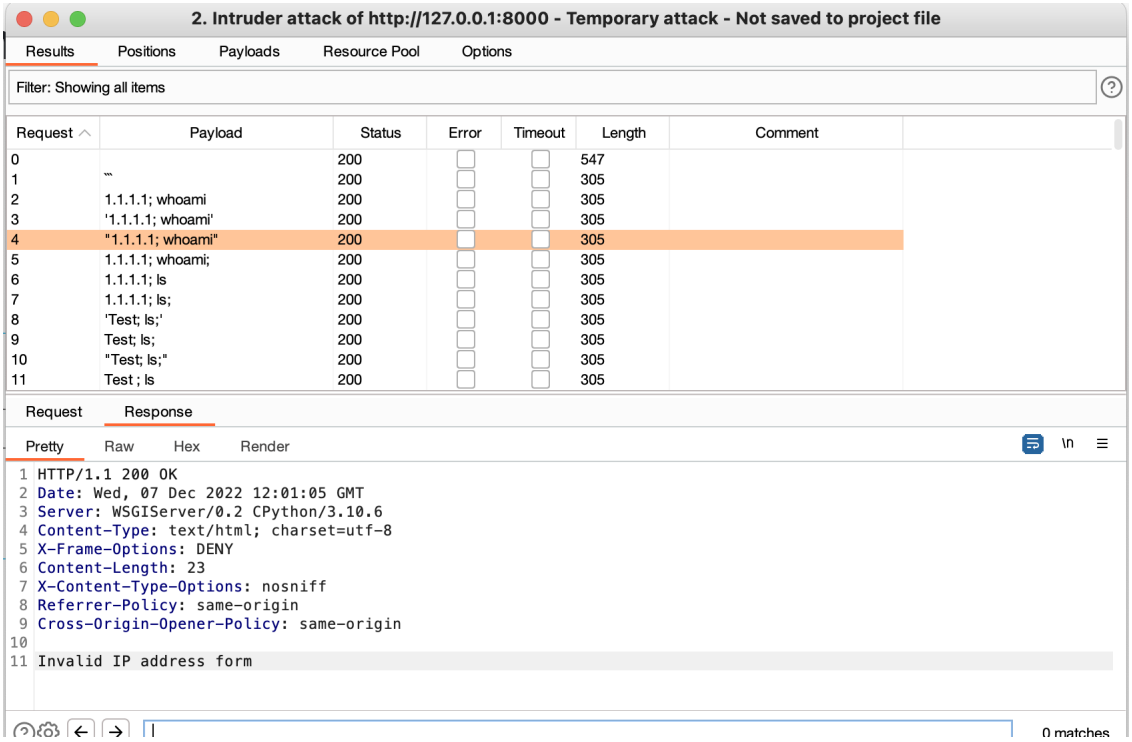

Figure 27: Inspecting responses

7. References

- https://semgrep.dev/docs/cheat-sheets/python-command-injection/
- https://portswigger.net/support/using-burp-to-test-for-os-command-injection-vulnerabil
- https://knowledge-base.secureflag.com/vulnerabilities/code_injection/os_command_injection_python.html
- https://thesecmaster.com/what-is-command-injection-vulnerability-and-how-to-prevent-it
- https://portswigger.net/web-security/os-command-injection
- https://github.com/commixproject/commix
- https://stackoverflow.com/questions/3172470/actual-meaning-of-shell-true-in-subprocess
- https://www.stackhawk.com/blog/command-injection-python/
- https://www.shiftleft.io/blog/find-command-injection-in-source-code/
- https://www.abstractapi.com/guides/validate-ip-address-python
- https://cheatsheetseries.owasp.org/cheatsheets/OS_Command_Injection_Defense_Cheat_Sheet.html
- https://github.com/payloadbox/command-injection-payload-list