Input Validation & Security Testing

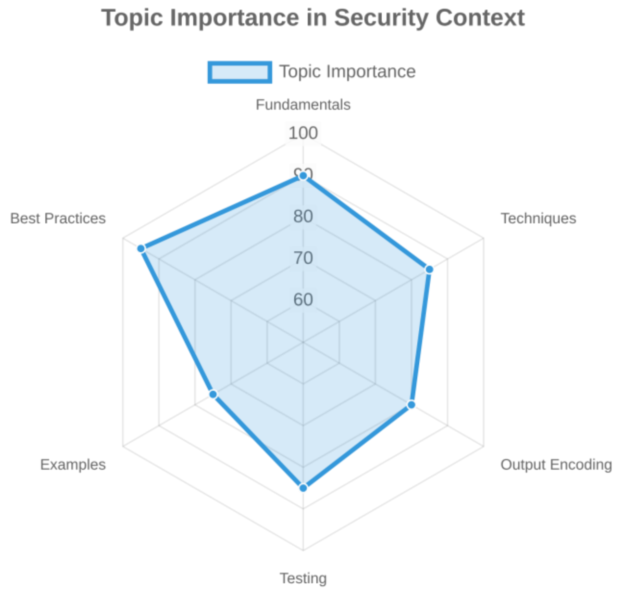
Building Secure Applications Through Proper Validation

Dr. Mohammed Tawfik

Secure Software Engineering

What We'll Cover Today

- **Input Validation Fundamentals** - Why it matters
- **Validation Techniques** - Practical approaches
- **Output Encoding** - Safe data display
- **Testing Methodologies** - How to test validation
- **Real-World Examples** - Common scenarios
- **Best Practices** - Industry standards

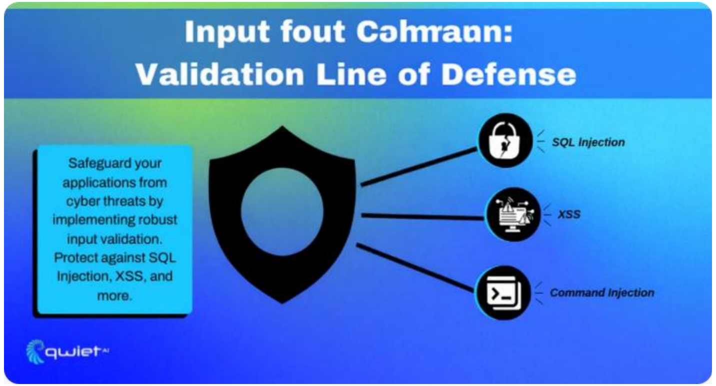


Important: This builds on concepts from previous chapters - we focus on practical implementation here!

Why Input Validation is Critical

The Reality

- User input is the **primary attack vector**
- 70% of security vulnerabilities stem from poor input handling
- Validation failures lead to data breaches, system compromise



Common Attack Scenarios

❌ **What Happens Without Validation:**

- Malicious users inject SQL commands
- Scripts execute in other users' browsers (XSS)
- System commands run on your server
- Sensitive data gets exposed or corrupted

Input Validation Fundamentals

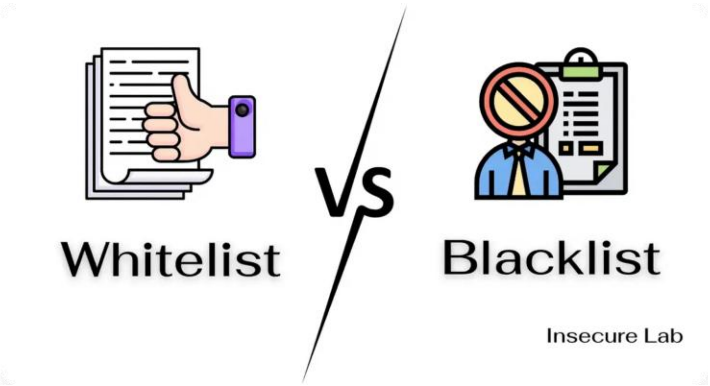
Core Principle

"Never Trust User Input - Always Validate"

Two Main Approaches

Whitelist (Allow)	Blacklist (Deny)
✅ Define what IS allowed ✅ More secure ✅ Recommended approach	❌ Define what is NOT allowed ❌ Easy to bypass ❌ Not recommended

Best Practice: Always use whitelist validation - only allow known good input!



Types of Input Validation

1 Syntax Validation

Check format, length, character set

Example: Email format, phone numbers

2 Semantic Validation

Check if data makes logical sense

Example: Birth date in the past, valid country codes

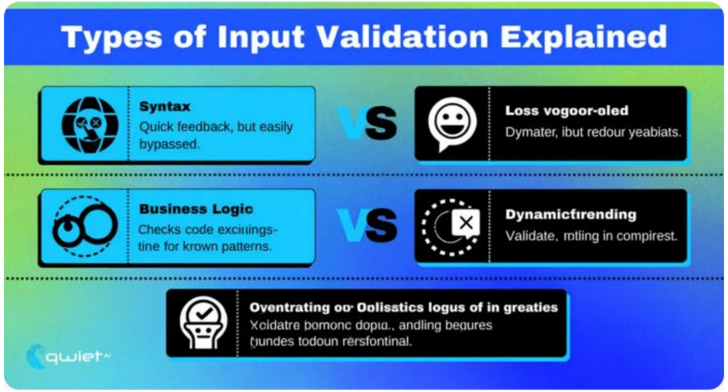
3 Business Logic Validation

Check against business rules

Example: Account balance, user permissions

Example: Age field validation

- **Syntax:** Must be numeric, 1-3 digits
- **Semantic:** Must be between 0-150
- **Business:** Must be 18+ for certain services



Where to Validate Input

Client-Side Validation

✅ **Purposes:**

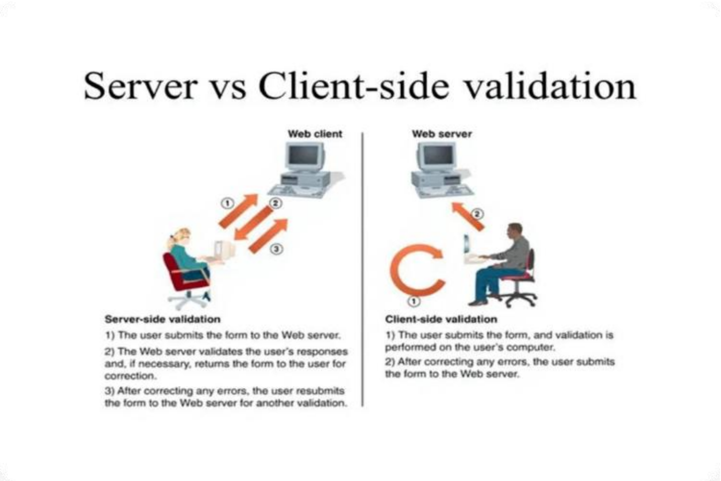
- Improve user experience
- Reduce server load
- Provide immediate feedback

⚠️ **Security Note:** Client-side validation can be easily bypassed - never rely on it for security!

Server-Side Validation

✅ **Critical for Security:**

- Cannot be bypassed by users
- Final line of defense
- Must validate ALL input sources



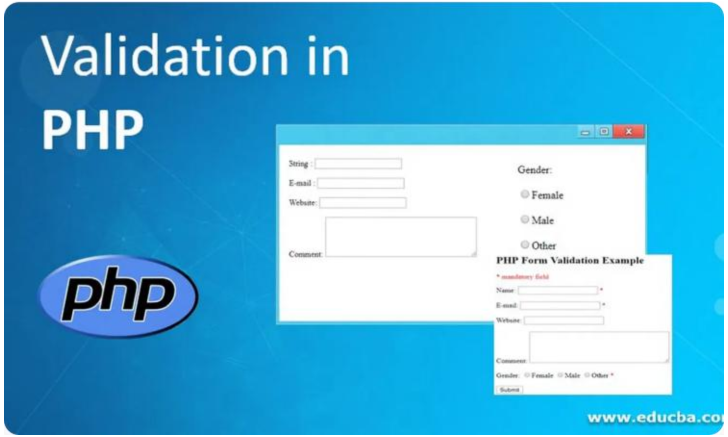
PHP Input Validation Examples

Basic String Validation

```
function validateUsername ($username) { // Remove whitespace $username = trim($username); // Check length (3-20 characters) if (strlen($username) < 3 || strlen($username) > 20) { return false; } // Check format (alphanumeric + underscore only) if (!preg_match('/^[a-zA-Z0-9_]+$/' , $username)) { return false; } return $username; // Return sanitized version }
```

Email Validation

```
function validateEmail ($email) { $email = trim($email); return filter_var ($email, FILTER_VALIDATE_EMAIL); }
```



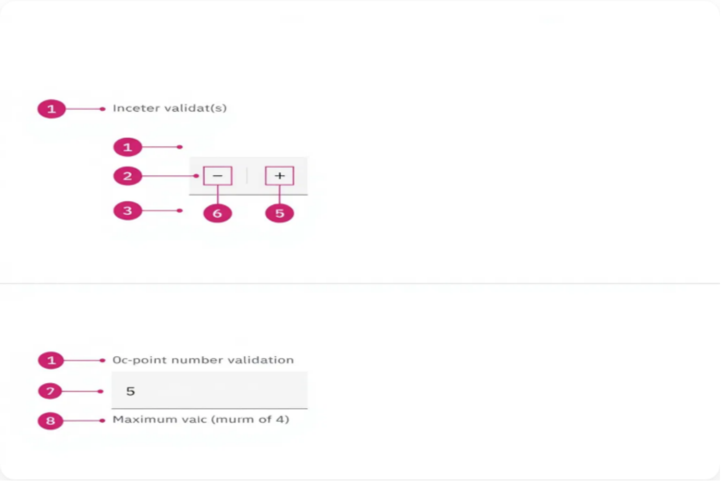
Numeric Input Validation

Integer Validation

```
function validateAge ($age) { // Check if numeric  if (!is_numeric ($age)) { return false;
} // Convert to integer  $age = ( int)$age; // Check range if ($age < 0 || $age > 150) {
return false; } return $age; }
```

Price/Money Validation

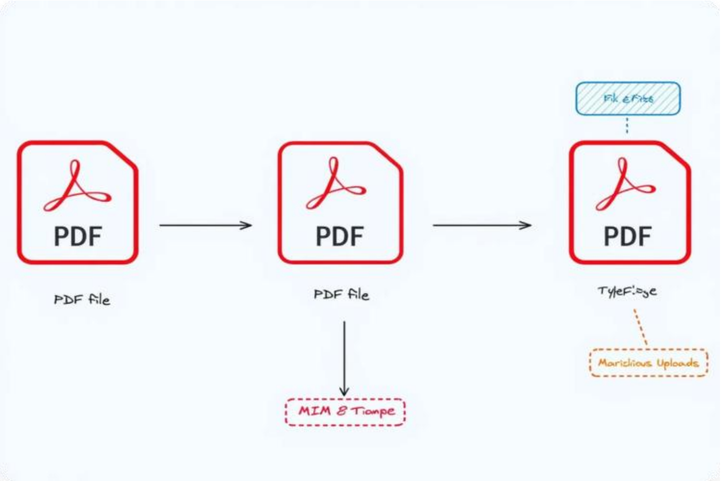
```
function validatePrice ($price) { // Remove currency symbols and whitespace    $price
= trim(str_replace (['$', ','], '', $price));// Check format (numbers with optional
decimal) if (!preg_match ('/^\d+(\.\d{1,2})?$/', $price)) { return false; } $price =
(float)$price; return ($price >= 0) ? $price :  false; }
```



File Upload Validation

Critical Security Checks

```
function validateFileUpload ($file) { $errors = []; // Check file size (max 5MB) if
($file['size'] > 5 * 1024 * 1024) { $errors[] = "File too large (max 5MB)" ; } // Check file
type $allowedTypes = ['image/jpeg', 'image/png', 'image/gif']; if
(!in_array ($file['type'], $allowedTypes )) { $errors[] = "Invalid file type" ; } // Check file
extension $allowedExtensions = ['jpg', 'jpeg', 'png', 'gif']; $extension =
strtolower (pathinfo ($file['name'], PATHINFO_EXTENSION)); if (!in_array ($extension,
$allowedExtensions )) { $errors[] = "Invalid file extension" ; } // Check for upload
errors if ($file['error'] !== UPLOAD_ERR_OK) { $errors[] = "Upload failed" ; } return
empty ($errors) ? true : $errors; }
```



SQL Injection Prevention

❌ Vulnerable Code

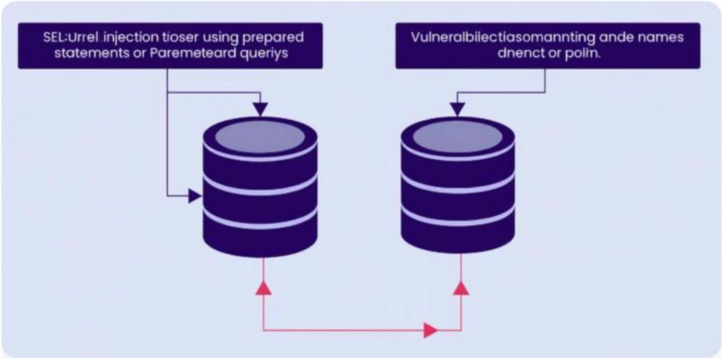
```
// NEVER DO THIS!  $query = "SELECT * FROM users WHERE username = '" .
$_POST['username'] . "'"; $result = mysqli_query($connection, $query);
```

Problem: User can inject SQL commands

✅ Secure Code - Prepared Statements

```
// ALWAYS DO THIS!  $query = "SELECT * FROM users WHERE username = ?";
$stmt = $pdo->prepare($query); $stmt->execute([$_POST['username']]); $result
= $stmt->fetchAll();
```

Solution: Parameters are properly escaped



Cross-Site Scripting (XSS) Prevention

❌ Vulnerable Output

```
// DANGEROUS!  echo "Hello " . $_POST['name'];
```

Attack: User enters `<script>alert('XSS')</script>`

✅ Safe Output Encoding

```
// SAFE! echo "Hello " . htmlspecialchars($_POST['name'], ENT_QUOTES, 'UTF-8');
```

Result: Scripts are displayed as text, not executed

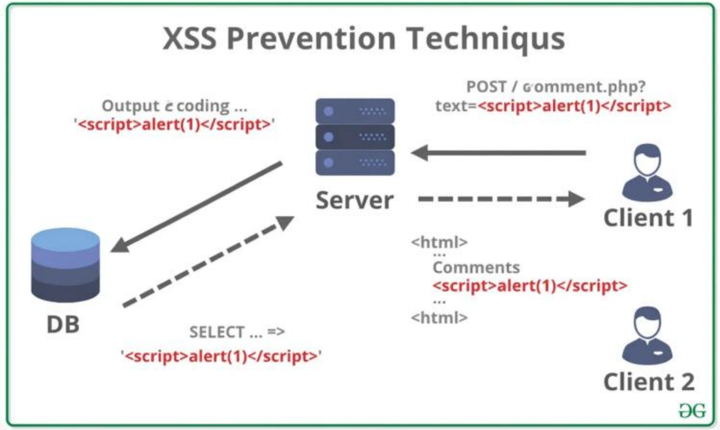
Context-Specific Encoding

HTML Context: htmlspecialchars()

JavaScript Context: json_encode()

URL Context: urlencode()

CSS Context: CSS escaping functions



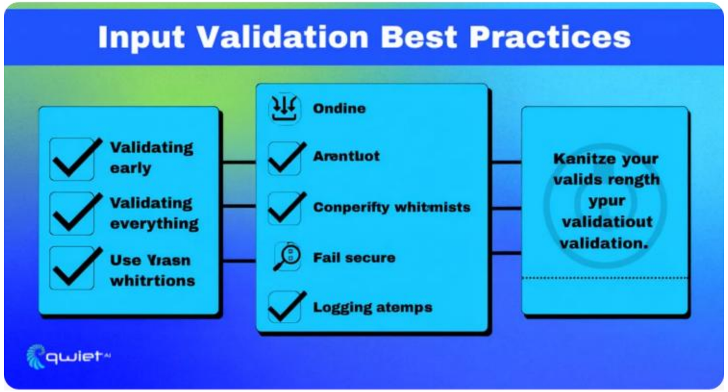
Input Validation Best Practices

The Golden Rules

1. **Validate Early:** Check input as soon as it arrives
2. **Validate Everything:** All user input, no exceptions
3. **Use Whitelist:** Define what's allowed, not what's forbidden
4. **Fail Securely:** Reject invalid input gracefully
5. **Log Attempts:** Record validation failures for monitoring

Common Mistakes to Avoid

- ❌ Relying only on client-side validation
- ❌ Using blacklist filtering
- ❌ Forgetting to validate file uploads
- ❌ Not encoding output properly
- ❌ Trusting "hidden" form fields



Security Testing Fundamentals

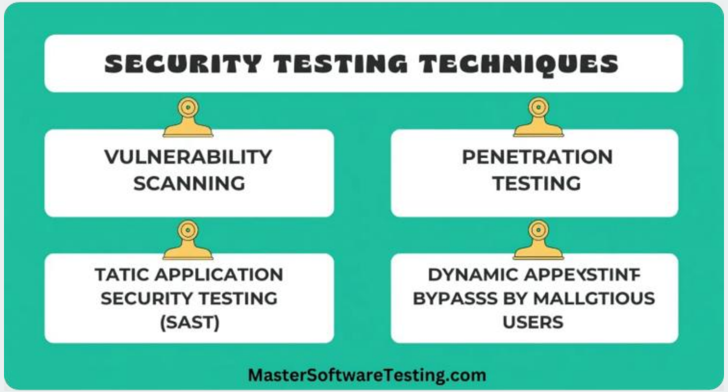
Why Test Validation?

- Ensure validation rules work correctly
- Find edge cases and bypasses
- Verify error handling
- Test boundary conditions

Types of Security Testing

Unit Testing → Integration Testing → Security Testing

🎯 **Goal: Prove that your validation cannot be bypassed by malicious users**



Manual Security Testing

Testing Approach

1. Understand the Application
2. Identify Attack Vectors
3. Test Boundary Cases
4. Attempt Bypasses
5. Document Results

Common Test Cases

- Text Fields: Special characters, long inputs, scripts
- Numeric Fields: Negative values, zero, very large numbers
- Email Fields: Invalid formats, special characters

Testing for SQL Injection

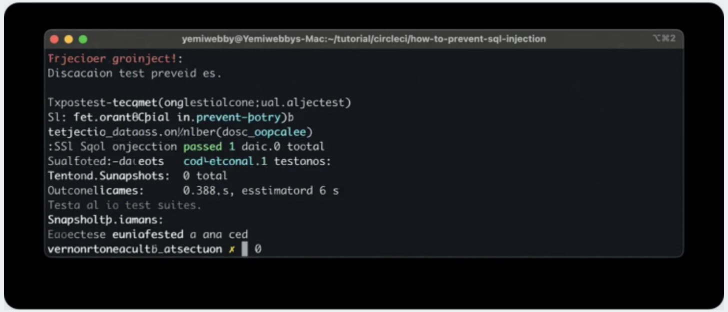
Basic Test Payloads

```
-- Authentication bypass:    ' OR '1'='1 admin'; --  -- Database exploration:    ' UNION SELECT table_name,column_name FROM information_schema.columns; --        -- Destructive test (never use in production!):    admin'; DROP TABLE users; --
```

What to Look For

- Error Messages: Database errors revealing structure
- Timing Changes: Queries taking longer than normal
- Unexpected Results: Getting data you shouldn't see
- Successful Login: Bypassing authentication

Expected Result: All attempts should fail with generic error messages



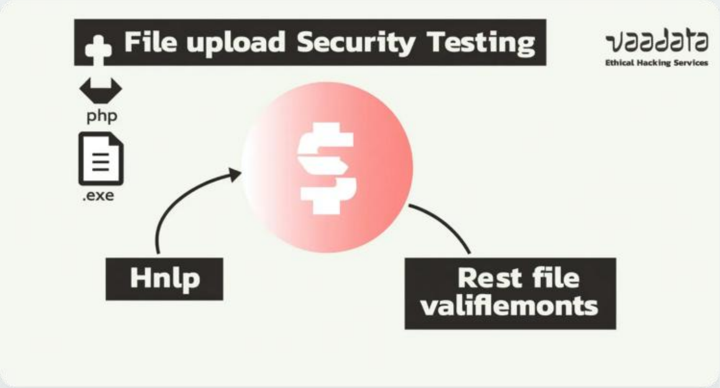
Testing for Cross-Site Scripting

Basic XSS Payloads

```
// Basic alert test:   <script>alert('XSS')</script>   // Event handler based:   <img src=x
onerror=alert('XSS')>   // JavaScript URL:   <a href="https://private-us-east-
1.manuscdn.com/sessionFile/CjFyq4LFPmsBcW1jC2FBFy/sandbox/slides_resource_i9zmxrlm
64ecdf1e-e2b-
prod_1755886754564_na1fn_L2hvbWUvdWJ1bnR1L25ld19wcmVzZW50YXRpb25faW1hZ2VzL3N
x-oss-
process=image/resize,w_1560,h_1560/format,webp&Expires=1798761600&Policy=eyJTdGF0Z
Pair-Id=K2HSFNDJXOU9YS&Signature=Wl-ttb6-7jvJLgHP4-
W8kroYbxMLnUKvrje8H~LtgJhoaw~cEj1jXb7weyNa6af2xXvcVQXNJAdS49Cx2cWi2RCJyXR1HOa
k~YLfzVOn2rdXIl1E69nxmUCGbqEfeSBcxJQcwyKL7SJY~tjKsWGUrD-
xhWC2f39J7aVSFp6XuOn8QzSIftlH16dntEarMZr0xLvu7QH8lCcYSLAJhfvoIkIh8myI9APIfae5SxJ
me</a>
```

Testing Process

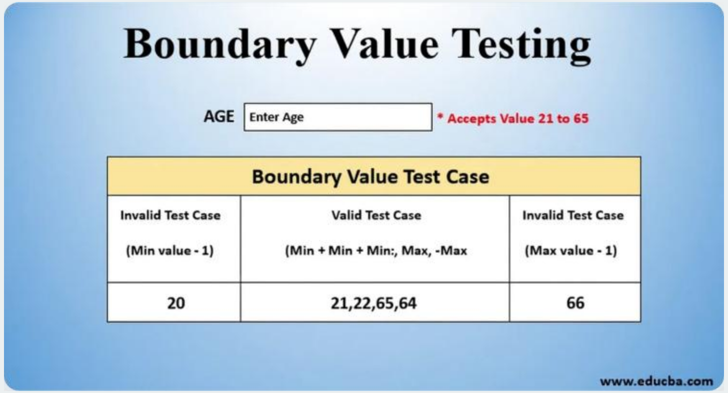
1. Submit XSS payload in input fields
2. Check how data is displayed
3. Look for unescaped output
4. Test different contexts (HTML, JavaScript, URL)



File Upload Security Testing

Test Cases

1. **File Type:** .php, .exe, .bat files (Should be rejected)
2. **File Size:** Extremely large files (Should be rejected)
3. **Malicious Content:** Files with embedded scripts (Should be scanned/rejected)
4. **Filename:** ../../../etc/passwd (Should be sanitized)
5. **MIME Type:** Mismatched content and extension (Should be validated)



Critical: Never execute uploaded files - always store them outside web root!

Boundary Value Testing

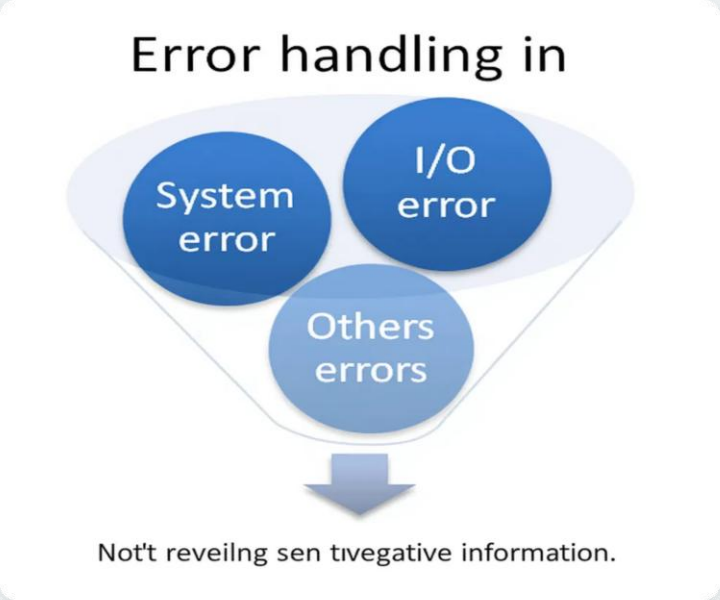
What are Boundaries?

Boundaries are the limits of valid input where validation rules change. Testing at these points is critical for finding vulnerabilities.

Minimum Values | Maximum Values | Edge Cases | Transition Points

Example: Age Field (18-65)

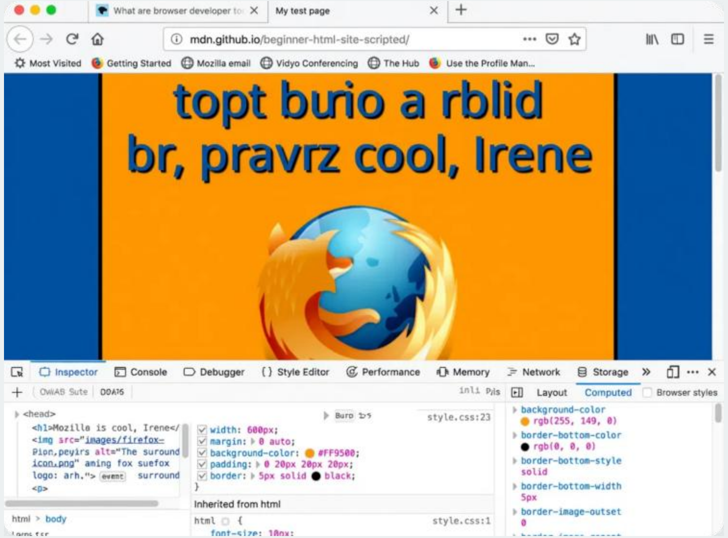
Test Value	Expected Result	Test Type
17	Reject	Min-1
18	Accept	Min
19	Accept	Min+1
64	Accept	Max-1
65	Accept	Max



Testing Error Handling

What to Test

- Error Messages: Are they generic and safe?
- Information Disclosure: Do errors reveal system details?
- Application State: Does it remain stable after errors?
- Logging: Are security events properly logged?



Good vs Bad Error Messages

Bad Error Message

Error: Failed to connect to MySQL server at 192.168.1.5:3306, user: dbadmin, password: ******

Good Error Message

Error: Unable to complete your request. Please try again later.

Security Rule: Never expose technical details to users in error messages!

Simple Automated Testing

Browser Developer Tools

Inspect Element

Examine and modify HTML/CSS in real-time

Console

Execute JavaScript and view errors

Network Tab

Monitor requests and responses

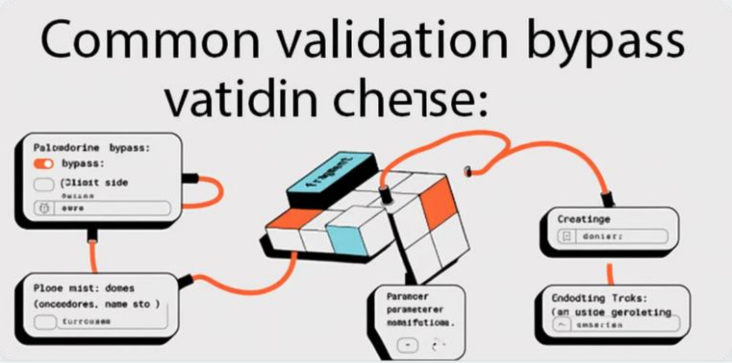
Proxy Tools

Burp Suite Community

Intercept and modify HTTP requests

OWASP ZAP

Free security testing suite



Documenting Security Tests

What to Record

- **Test Case:** Specific test performed
- **Input Used:** Test payload
- **Expected Result:** What should happen
- **Actual Result:** What happened
- **Status:** Pass/Fail
- **Risk Level:** Severity if exploited

Sample Test Record

Field	Value
Test Case	Username XSS Test
Input Used	<script>alert('XSS')</script>
Expected	Input rejected or sanitized
Actual	Script executed when displayed
Status	**Failed**
Risk Level	High

Common Validation Bypasses

Techniques Attackers Use

Client-Side Bypass

Disabling JavaScript or modifying HTML in browser

```
// Disable form validation
document.getElementById("form").noValidate = true;
```

Parameter Manipulation

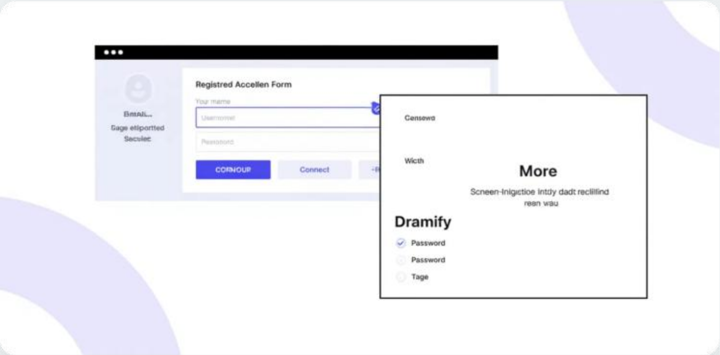
Modifying hidden form fields or request parameters

```
<input type="hidden" name="isAdmin" value="false">
// Changed to value="true"
```

Encoding Tricks

URL encoding, double encoding, HTML entity encoding

```
// Double-encoded script tag
%253Cscript%253Ealert(1)%253C/script%253E
```



Defense: Always validate on the server side and normalize input!

Security Testing Checklist

Comprehensive Validation Testing

XSS Testing

- Test all text inputs with script tags
- Test event handlers (onerror, onload)
- Test URL parameters and hidden fields

SQL Injection

- Test login forms with SQL syntax
- Test search fields with UNION queries
- Test numeric fields with SQL operators

File Uploads

- Test executable file extensions (.php, .exe)
- Test MIME type validation bypass
- Test path traversal in filenames

Numeric Validation

- Test boundary values (min/max)
- Test negative numbers where inappropriate
- Test floating point vs integer handling

Error Handling

- Check for technical details in errors
- Test application state after errors
- Verify proper error logging

Authentication

- Test password complexity requirements
- Test account lockout mechanisms
- Test session timeout handling

Security Testing Checklist

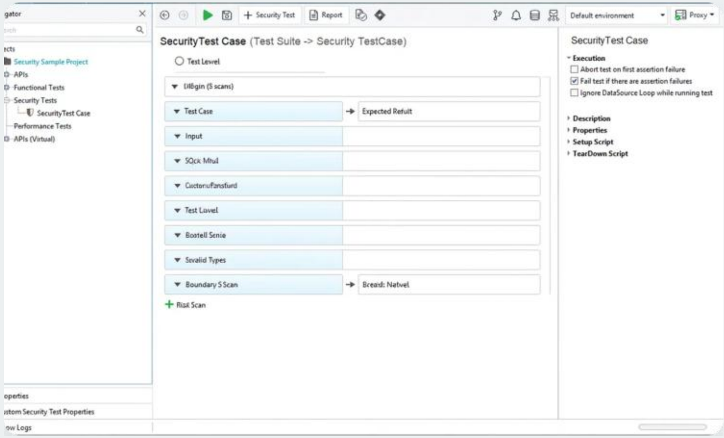
Real-World Example: User Registration

Testing a Registration Form

Username | Email | Password | Age

Profile Picture

Step-by-Step Testing



Username

- Try <script>alert('XSS')</script> **(Should be rejected)**
- Try very long username (100+ characters) **(Should be limited)**

Email

- Try ŌŎPMŌÑÖMÒÕ **(Should be rejected)**
- Try ÀÊŒNǾŐPËMÕÑǾPĀÃÎ Ĭ ĬĀÅÊ ŒNǾŐPËÀĚ ÑŔMÖ ŐÕÑBŃŎÖ **(Should be sanitized)**

Password

- Try ŒOŎǾP **(Should be rejected)**
- Try ... **(Should be rejected as common password)**

Scope & Objectives

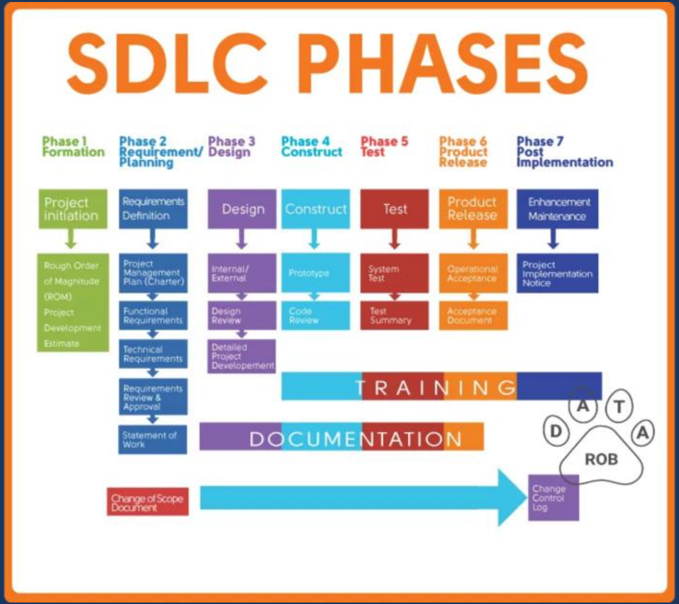
Users: Employees (view their own payslip for the current month)

Data: Names, employee IDs, pay rates, allowances, deductions, net pay (confidential/PII)

Out of scope: HR onboarding, tax filing, bank transfers, admin UI

Security goals:

- Confidentiality of payroll data
- Integrity of calculations
- Availability of the endpoint



Requirements & Analysis

1.1 Assets

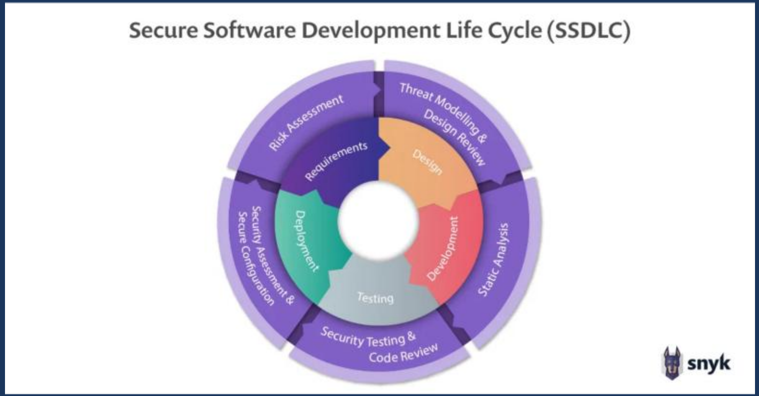
- A1: Payroll data (highly confidential)
- A2: Authentication/identity tokens
- A3: Secrets (DB creds, keys)

1.2 Threats (high level)

- Credential stuffing / ID guessing
- Injection (SQL/NoSQL)
- Broken auth/session
- Insecure storage/logs
- DoS / brute-force

1.3 Security Requirements

- SR1: OIDC / SSO via enterprise IdP
- SR2: Input validation
- SR3: Parameterized queries



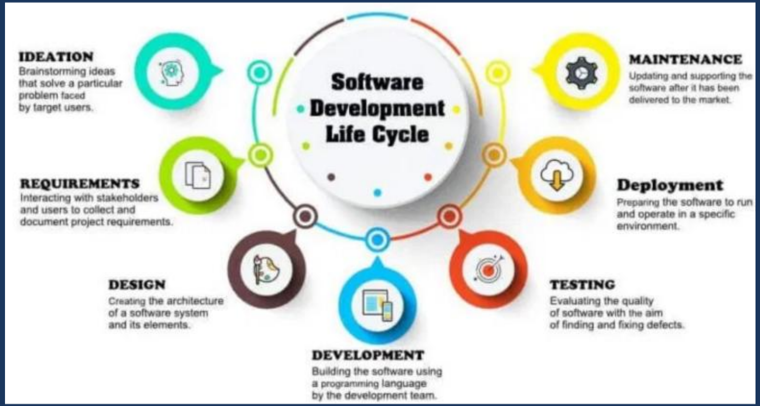
Design

2.1 Architecture

```
[Browser] —HTTPS—> [API Gateway] —> [Payroll API] —SQL—> [Payroll DB]
              |-> [Secrets: Vault]
              |-> [Identity: OIDC]
```

2.2 ER Model (minimal)

```
Employees(emp_id PK, name, email)
Payslips(emp_id FK, month, gross, deductions, net, PRIMARY KEY(emp_id,
month))
```



2.3 Key Design Controls

- Enforce **AuthZ**: WHERE emp_id = :subject_emp_id
- Strict **input validation** and **output encoding**
- **Least privilege** DB user
- **Fail-closed** responses with uniform messages

Threat Modeling (STRIDE snapshot)

Element	Threat	Example	Mitigation
Login/OIDC	Spoofing	Stolen tokens	Short token TTL, refresh rotation
API param	Tampering	SQLi via `month`	Param queries, allow-list regex
Logs	Repudiation	Access denied	Audit logs, clock sync
DB	Info disclosure	Mass read	Row-level authZ, least privilege
API	DoS	Flooding	WAF + rate limit + circuit breaker
Build	EoP	Malicious dep	SCA, pin versions, review PRs

Implementation (Tiny Code Sample)

Python/Flask Implementation

```
import re
from flask import Flask, request, jsonify
from db import get_payslip_for
ID_RE = re.compile(r"^EMP-\d{6}$")
MO_RE = re.compile(r"^\d{4}-\d{2}$")
app = Flask(__name__)

@app.get('/api/payslip')
def payslip():
    emp_id = request.headers.get('X-Subject-EmpId','')
    q_emp = request.args.get('emp','')
    month = request.args.get('month','')
    if not (ID_RE.fullmatch(emp_id) and
            ID_RE.fullmatch(q_emp) and
            MO_RE.fullmatch(month)):
        return jsonify({"message":
          "If authorized, your payslip will be shown."}), 200
    if q_emp != emp_id:  # enforce ownership
        return jsonify({"message":
          "If authorized, your payslip will be shown."}), 200
    row = get_payslip_for(emp_id, month)
    if not row:
        return jsonify({"message":
          "If authorized, your payslip will be shown."}), 200
    return jsonify({"emp_id": emp_id, "month": month, **row})
```

Verification & Testing – By Phase

Phase A: Requirements

- **Activities:** Security story writing; abuse/misuse cases
- **Tools:** OWASP ASVS L1–L2, Markdown ADRs

Phase B: Design

- **Activities:** Architecture/DFD, threat model (STRIDE)
- **Tools:** Microsoft Threat Modeling Tool, Draw.io

Phase C: Implementation

- **Activities:** Secure coding, parameterized queries
- **Tools:** GitHub Advanced Security, SonarQube

Phase D: Integration

- **Activities:** Wire API ↔ DB ↔ IdP
- **Tools:** OWASP ZAP baseline, k6/JMeter

PM Phase	PM Processes	Project Milestones	SD Processes	SD Phase
Initiation	• Define project parameters • Identify risks and quality standards • Develop Initial Project Plan	➢ **Project Charter** ➢ **Subsystem 1 Business Requirements** ➢ **Subsystem n Business Requirements** ➢ **Subsystem 1 Functional Specifications** ➢ **Subsystem n Functional Specifications** ➢ **Initial Project Plan**	• Identify Business Requirements • Define Process and Data Models • Produce Functional Specifications	Requirements
Planning	• Refine project parameters • Assess risks / define QA and QC procedures • Refine Project Plan	➢ **System Prototype** ➢ **Subsystem 1 Technical Specifications** ➢ **Subsystem n Technical Specifications** ➢ **Project Plan**	• Define technical architecture • Prototype system components • Produce Technical Specifications	Design
Execution	• Manage project parameters • Monitor and control risks and quality • Manage project execution	➢ **Subsystem 1 Test** ➢ **Subsystem n Test** ➢ **Subsystem 1 Acceptance** ➢ **Subsystem n Acceptance** ➢ **User Documentation**	• Build System Components • Conduct System Testing • Produce Technical Documentation	Construction
		➢ **Data Validation** ➢ **Subsystem 1 Deployment** ➢ **Subsystem n Deployment** ➢ **System Transition**	• Convert/initiate data • Perform System Acceptance • Deploy and transition system	Implementation
Closeout	• Perform project assessment • Identify lessons learned • Archive project information	➢ **Project Assessment**		

Phase E: Verification

- **Tools:** DAST scan (ZAP), manual tests

CI/CD Security Gates

1. **Pre-commit**

   Secret scan (Gitleaks), formatter, linter

↓

2. **PR checks**

   Unit tests, SAST (CodeQL), SCA (Dependabot), policy checks

↓

3. **Merge to main**

   Build image → container scan → ZAP baseline

↓

4. **Staging deploy**

   Integration tests → ZAP full → manual authZ tests

↓

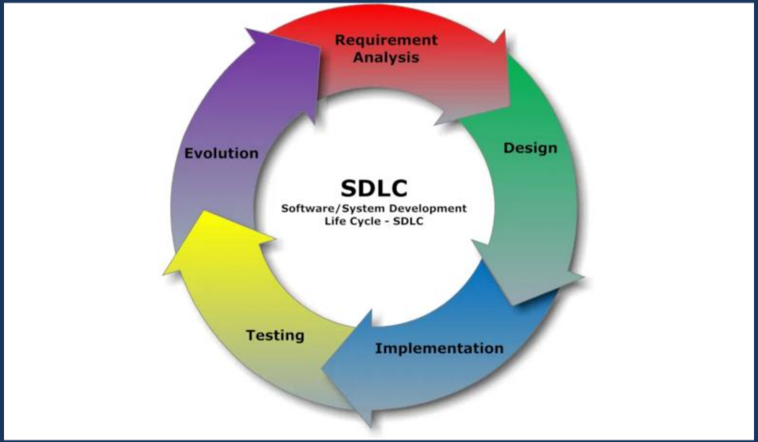
5. **Prod deploy**

   Change approval + smoke tests + header/TLS checks

Sample Test Cases & Common Issues

Sample Test Cases

- **AuthZ:** User EMP-123456 requests payslip for EMP-654321 → **Denied** (generic message)
- **Input validation:** month=2025-13 or emp=EMP-001 → generic response, no stack traces
- **SQLi:** month=2025-01' OR '1'='1 → no effect (param query), generic message
- **Rate limit:** 100 req/min from same IP → limited with 429



Common Issues & Fixes

- **Enumeration via error text** → **Return uniform messages; log details server-side only**
- **Missing authZ check** → **Always match emp_id from token to query; add central middleware**
- **Secrets in code** → **Move to Key Vault; rotate immediately**

Deliverables by Phase

Requirements
- Security user stories
- NFR list

Design
- DFD, ERD
- Threat Model (TMT file)
- ADRs

Implementation
- Minimal service + unit tests
- Secure code review checklist

Verification
- ZAP report
- CodeQL results
- SCA report

Release
- IaC manifests
- Hardening checklist

Operations
- Runbook; alert playbooks
- Monthly patch reports

Tools Summary

Microsoft Tools

- Threat Modeling Tool
- Azure DevOps Pipelines
- GitHub Advanced Security
- Defender for DevOps
- Key Vault
- Entra ID
- Monitor/Insights
- Sentinel

Common OSS/3rd-party Tools

- OWASP ZAP
- CodeQL
- Gitleaks
- Dependabot
- SonarQube
- k6/JMeter
- Terraform
- Trivy/Grype

Thank You!

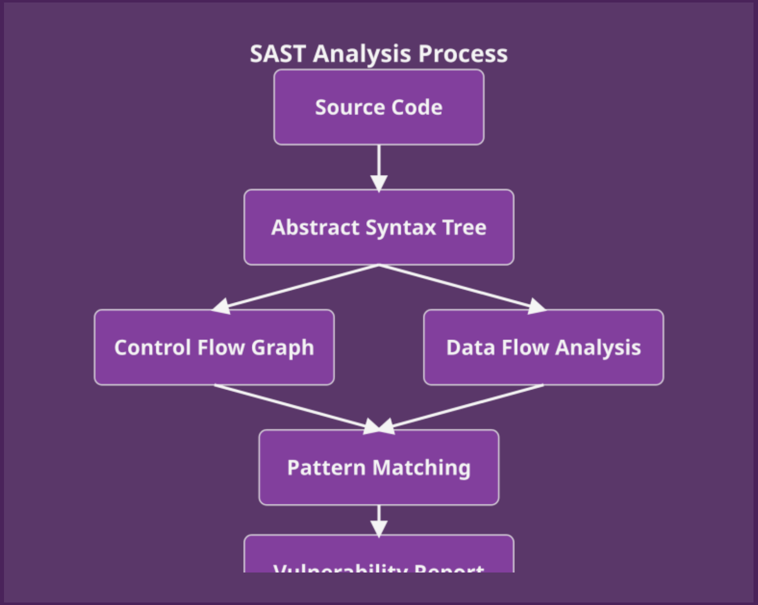
Chapter 6: Advanced Software Security Testing

Techniques and Tools in Practice

Dr. Mohammed Tawfik

SAST Beyond Basics: Practical Application

- **Strategic Deployment Points:** Integrating SAST at key stages: pre-commit hooks, pull requests, and CI/CD pipelines for maximum effectiveness
- **Analysis Engine Mechanics:** Understanding how SAST tools parse code into Abstract Syntax Trees (ASTs) and analyze control/data flow to identify vulnerabilities
- **Context-Aware Analysis:** How modern SAST tools consider application context, framework-specific patterns, and cross-component interactions
- **Incremental Scanning:** Optimizing SAST performance by analyzing only changed code while maintaining awareness of the broader codebase



Configuring and Tuning SAST Tools

- **Rule Configuration:** Tailoring security rules to match your application's technology stack, architecture, and business context
- **False Positive Reduction:** Implementing baseline scans, rule suppression, and custom exclusions to minimize noise and focus on actionable findings
- **Third-Party Code Management:** Strategies for handling dependencies, including exclusion from analysis, separate scanning, and dependency vulnerability tracking
- **Custom Rule Development:** Creating organization-specific rules to detect unique security patterns and enforce internal security standards

SonarQube Custom Rule Configuration

```java
// Example SonarQube custom rule in Java
@Rule(
    key = "AvoidUnsafeDeserialization",
    name = "Avoid unsafe deserialization",
    description = "Unsafe deserialization can lead to RCE",
    priority = Priority.CRITICAL,
    tags = {"security", "vulnerability"}
)
public class UnsafeDeserializationRule extends BaseTree

    private static final List<String> UNSAFE_CLASSES =
        Arrays.asList(
            "ObjectInputStream",
            "XMLDecoder",
            "XStream"
    );

    // Implementation to detect unsafe patterns
    @Override
        visitMethodInvocation( ) {
```

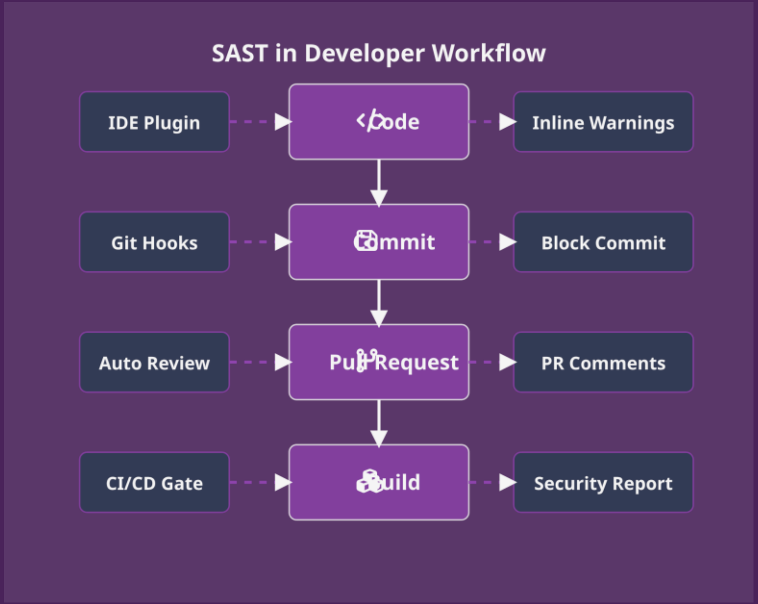
Interpreting and Prioritizing SAST Findings

- **Understanding SAST Reports:** Decoding key metrics like vulnerability density, severity distribution, and trend analysis to gauge security posture
- **Risk-Based Prioritization:** Moving beyond severity ratings to consider exploitability, business impact, and affected component criticality
- **Validation Techniques:** Methods to verify findings, including manual code review, proof-of-concept testing, and correlation with other security tools
- **Remediation Planning:** Organizing findings into actionable work items with clear ownership, timelines, and verification criteria

SAST Finding Prioritization Matrix

SAST Integration with Developer Workflows

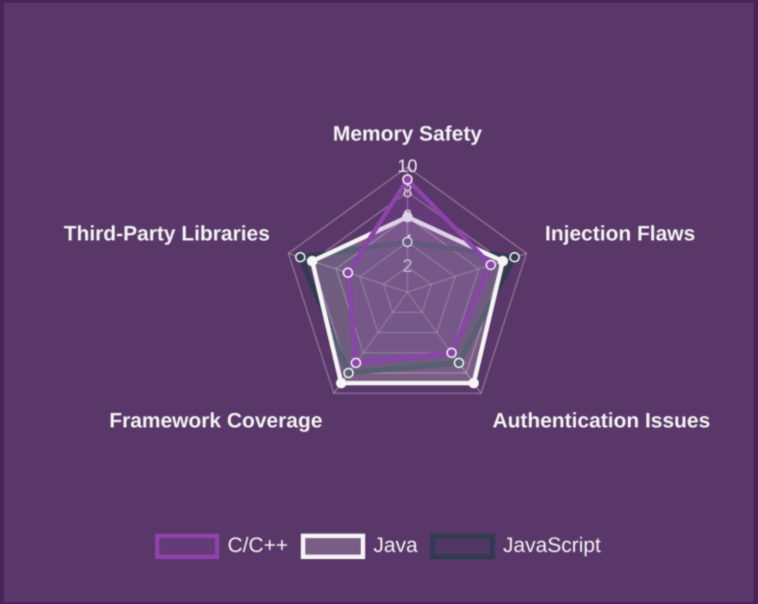
- **IDE Integration:** Real-time security feedback through IDE plugins (VS Code, IntelliJ, Eclipse) that highlight vulnerabilities as developers write code
- **Git Hooks and Pull Requests:** Pre-commit and pre-push hooks that prevent insecure code from entering the repository, with automated security reviews on PRs
- **CI/CD Pipeline Integration:** Automated SAST scans in build pipelines with configurable quality gates that can block deployments based on security findings
- **Developer-Friendly Notifications:** Contextual security alerts with remediation guidance delivered through familiar channels (Slack, MS Teams, email)



SAST for Different Programming Languages

- **Language-Specific Vulnerabilities:** Each language has unique security concerns—memory management in C/C++, deserialization in Java, injection in dynamic languages like PHP and JavaScript
- **Polyglot Environments:** Strategies for effective SAST in multi-language applications, including coordinated scanning and unified reporting across language boundaries
- **Tool Selection Criteria:** Evaluating SAST tools based on language support depth, framework awareness, and accuracy for your specific technology stack
- **Framework-Specific Analysis:** How SAST tools leverage knowledge of popular frameworks (Spring, Django, React) to provide more accurate and contextual security findings

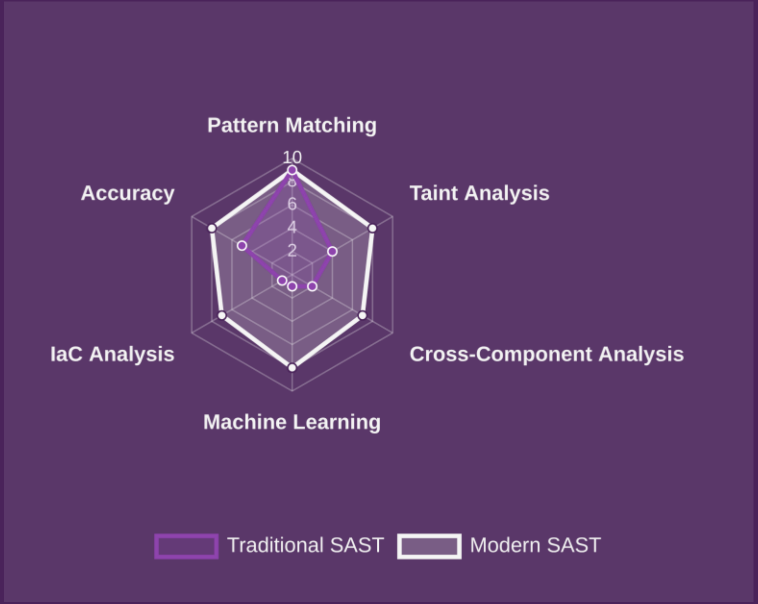
SAST Effectiveness by Language



Advanced SAST Techniques

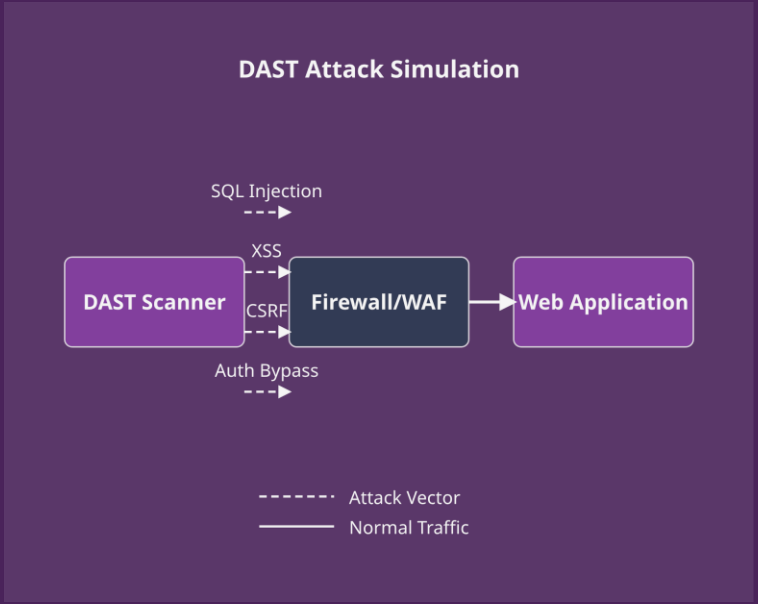
- **Taint Analysis:** Tracking untrusted data from sources (user inputs) to sinks (sensitive operations) to identify potential injection vulnerabilities
- **Machine Learning in SAST:** Using AI to improve detection accuracy, reduce false positives, and identify complex vulnerability patterns that rule-based systems might miss
- **Cross-Component Analysis:** Detecting vulnerabilities that span multiple files, modules, or services by analyzing data flow across component boundaries
- **Infrastructure as Code Analysis:** Extending SAST beyond application code to detect security issues in infrastructure definitions (Terraform, CloudFormation, Kubernetes manifests)

SAST Evolution and Capabilities



DAST in Practice: Simulating Real-World Attacks

- **Attacker's Perspective:** DAST tools operate as "black-box" testers, probing applications from the outside without knowledge of internal code, simulating real attackers
- **Strategic Deployment Points:** Pre-production testing, continuous monitoring in staging environments, and scheduled scans in production with proper safeguards
- **Discovery Techniques:** Combining automated crawling, manual exploration, and API specification analysis to maximize application coverage
- **Safe Attack Simulation:** Configuring non-destructive payloads, implementing proper test isolation, and using production-safe scanning profiles



Advanced DAST Scanning and Configuration

- **Complex Application Scanning:** Configuring DAST tools for modern web architectures including single-page applications (SPAs), JavaScript-heavy frontends, and multi-step workflows
- **Authentication Handling:** Setting up authenticated scans with session management, multi-factor authentication support, and maintaining session state throughout testing
- **Custom Attack Payloads:** Developing organization-specific attack vectors, fuzzing patterns, and payload libraries to test for unique vulnerabilities
- **Scan Policy Optimization:** Balancing scan depth, coverage, and performance through targeted policy configuration and exclusion rules

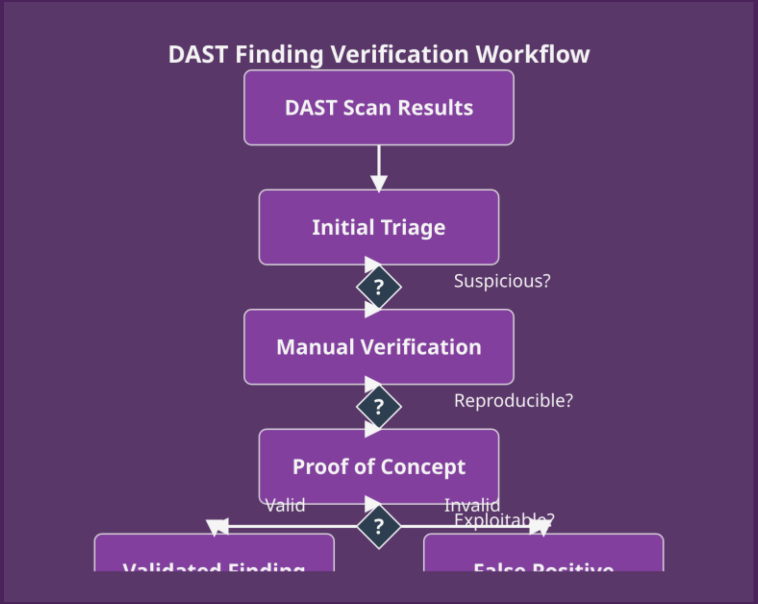
OWASP ZAP Authentication Configuration

```
# ZAP Authentication Script Example
function authenticate(helper, paramsValues, credentials) {
    // Get the login URL
    var loginUrl = paramsValues.get("Login URL");

    // Create the login request
    var request = helper.prepareMessage();
    request.setMethod("POST");
    request.setRequestBody(
        "username=" + encodeURIComponent(credentials.getParam(
        "&password=" + encodeURIComponent(credentials.getParam(
        "&csrf_token=" + extractCsrfToken()
    );

    // Submit login request
    helper.sendAndReceive(request);

    // Check if authentication succeeded
    var msg = request.getResponseHeader().toString();
                                ) > 0;
```

Analyzing and Exploiting DAST Findings

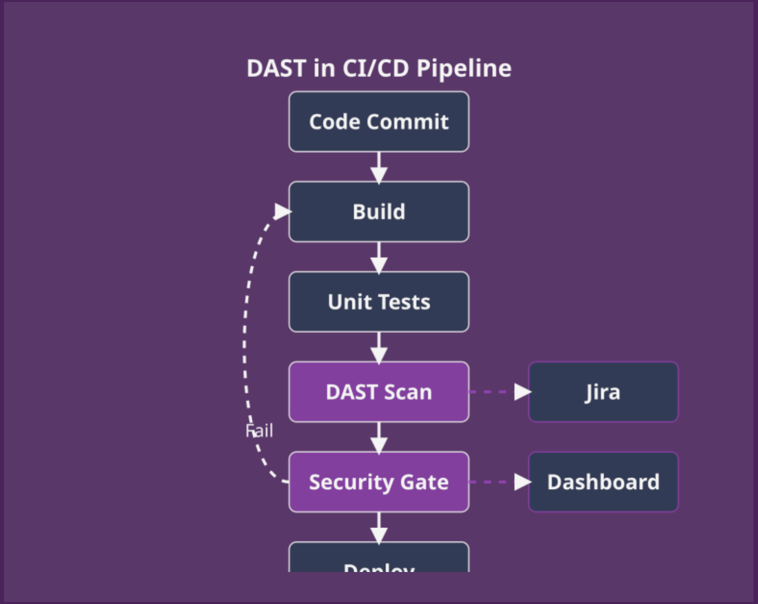
- **Interpreting DAST Reports:** Understanding scan results, including vulnerability details, attack vectors, evidence, and potential impact on the application
- **Manual Verification:** Techniques for validating DAST findings to eliminate false positives, including proof-of-concept testing and contextual analysis
- **Risk-Based Prioritization:** Evaluating DAST findings based on exploitability, business impact, affected data sensitivity, and remediation complexity
- **Chaining Vulnerabilities:** Identifying how multiple low-severity findings can be combined to create high-impact attack chains requiring immediate attention

DAST Finding Verification Process



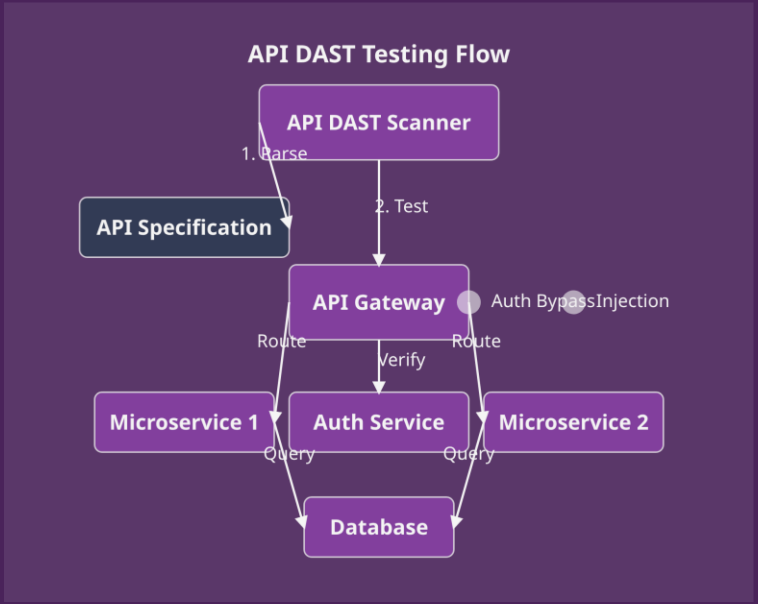
DAST Automation and Integration

- **CI/CD Pipeline Integration:** Automating DAST scans as part of deployment pipelines with configurable security gates that prevent vulnerable code from reaching production
- **Issue Tracking Integration:** Automatically creating tickets in systems like Jira, Azure DevOps, or GitHub Issues when vulnerabilities are discovered
- **Scheduled Scanning:** Implementing regular automated scans to continuously monitor applications for new vulnerabilities introduced by changes or emerging threats
- **Metrics and Reporting:** Generating trend analysis and security posture dashboards to track vulnerability remediation progress over time



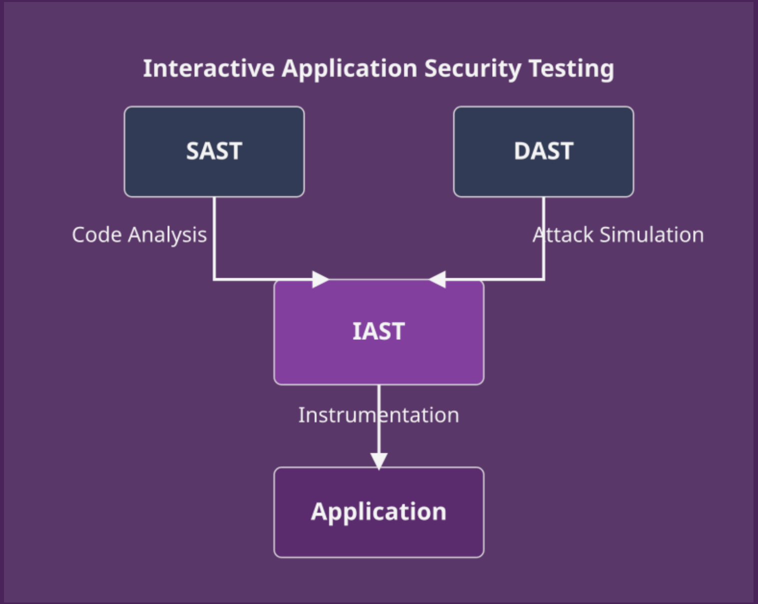
DAST for APIs and Modern Architectures

- **API-Specific Testing Challenges:** Addressing the unique security concerns of RESTful, GraphQL, and SOAP APIs, including parameter manipulation, authorization bypass, and resource exhaustion
- **Specification-Based Testing:** Leveraging OpenAPI/Swagger, RAML, or GraphQL schemas to generate comprehensive test cases that cover all endpoints and operations
- **Microservices Security:** Testing strategies for distributed architectures, including service-to-service communication, API gateways, and authentication boundaries
- **Serverless Function Testing:** Adapting DAST for cloud-native applications with ephemeral compute resources, focusing on input validation, environment variables, and third-party integrations



IAST: Bridging the Gap Between SAST and DAST

- **Hybrid Approach:** IAST combines the code-level insights of SAST with the runtime attack simulation of DAST, providing comprehensive security testing
- **Runtime Visibility:** Instruments application code to monitor execution flow, data movement, and security controls during normal testing activities
- **Key Advantages:** Higher accuracy with fewer false positives, precise vulnerability location, real-time feedback, and context-aware analysis
- **Limitations:** Language/framework dependencies, potential performance impact, and requires active application execution



Implementing IAST Agents and Monitoring

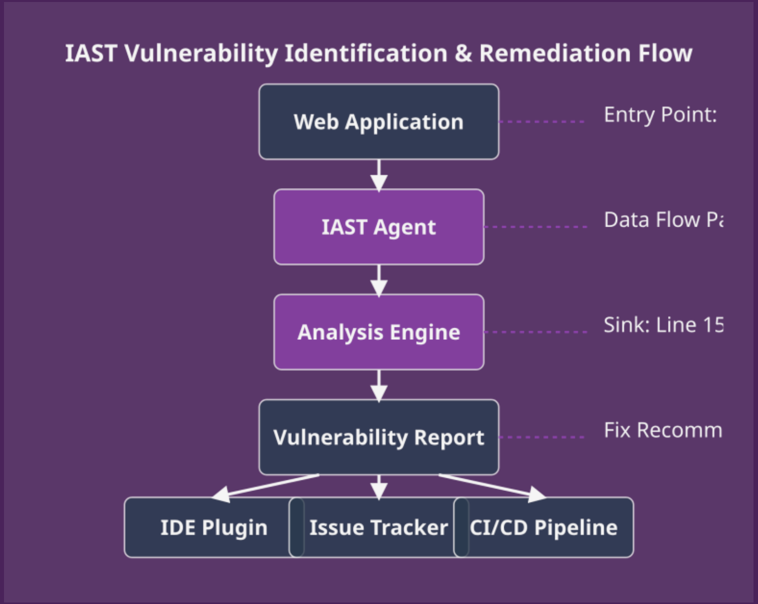
- **Agent Deployment Options:** JVM agents for Java, .NET CLR profilers, Node.js modules, Python instrumentation, and language-specific integration approaches
- **Configuration Strategies:** Tuning sensitivity levels, defining custom rules, specifying trusted sources/sinks, and setting up exclusions for third-party code
- **Environment Integration:** Deploying IAST in development, QA, and staging environments with appropriate monitoring dashboards and alert mechanisms
- **Performance Optimization:** Balancing security coverage with application performance through sampling rates, selective instrumentation, and monitoring overhead

Java IAST Agent Configuration Example

```
# Contrast Security Agent Configuration

MJÑŌP:
  logger È
    level : INFO

assess:
  ÑŌMNŌÑ ÈPǾÞÑ
  sampling È
    enable : true
    NMŒNŌŌÕÑ : 5
    ŊǾÑØÞÑŌNŘ : 10

rules È
  sql_injection :
    enabled : true
    ŒÑQŅǾPŘ ÈNǾÒPÒNMÕ
  xss È
    enabled : true
    : high
```

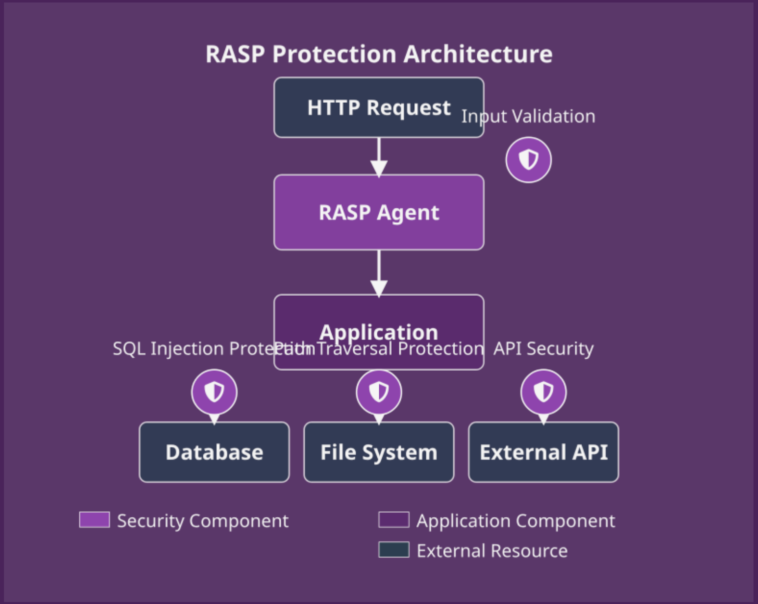
IAST for Vulnerability Identification and Remediation

- **Precise Vulnerability Location:** IAST pinpoints exact lines of vulnerable code, method calls, and data flow paths, eliminating the guesswork in remediation
- **Complete Attack Chains:** Traces the full vulnerability path from entry point to exploitation point, showing how data traverses through the application
- **Developer Integration:** Delivers findings directly to developers through IDE plugins, code review tools, and issue trackers with remediation guidance
- **Accelerated Fix Cycles:** Reduces mean-time-to-remediate (MTTR) by providing actionable context, proof-of-concept examples, and fix validation



Runtime Application Self-Protection (RASP) in Depth

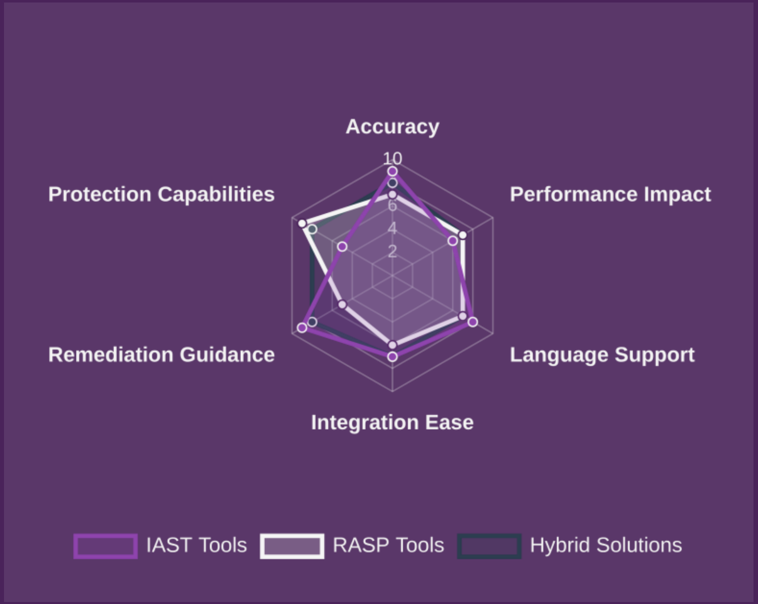
- **Proactive Defense Mechanism:** RASP instruments applications to detect and block attacks in real-time, providing protection even against zero-day vulnerabilities
- **Operational Modes:** Monitor-only for detection without blocking, protection mode for active intervention, and virtual patching to temporarily secure known vulnerabilities
- **Detection Techniques:** Context-aware analysis of HTTP requests, database queries, file operations, and system calls to identify malicious behavior patterns
- **Deployment Considerations:** Performance impact assessment, tuning protection levels, and integration with existing security monitoring and incident response processes



IAST/RASP Tools and Deployment Strategies

- **Leading Solutions:** Contrast Security (IAST/RASP), Synopsys Seeker (IAST), Checkmarx CxIAST, HCL AppScan, Signal Sciences (RASP), and Imperva Runtime Protection
- **Deployment Models:** Agent-based instrumentation, network-based monitoring, container security, and cloud service integration for modern application architectures
- **Performance Considerations:** Balancing security coverage with application performance through sampling, selective monitoring, and optimized instrumentation
- **Integration Strategy:** Phased rollout approach starting with non-critical applications, gradually expanding to business-critical systems with tuned protection levels

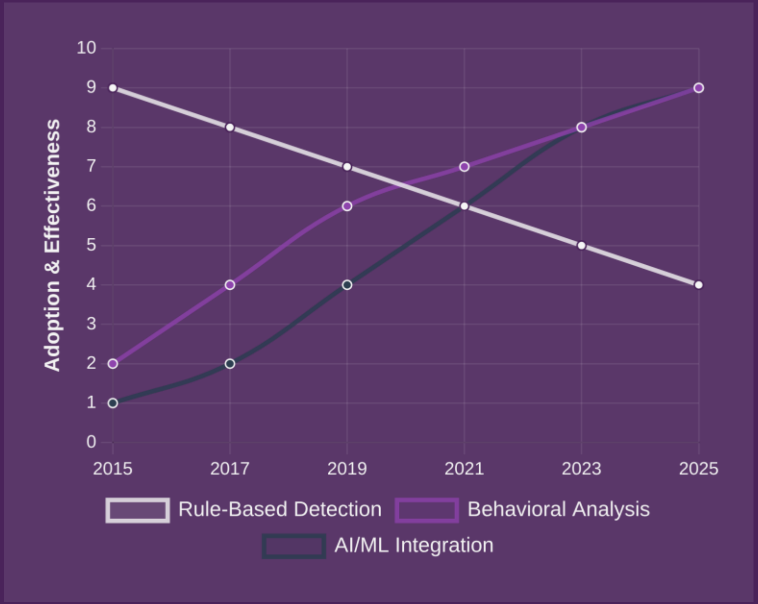
IAST/RASP Tool Comparison



Future of Runtime Security: AI and Behavioral Analysis

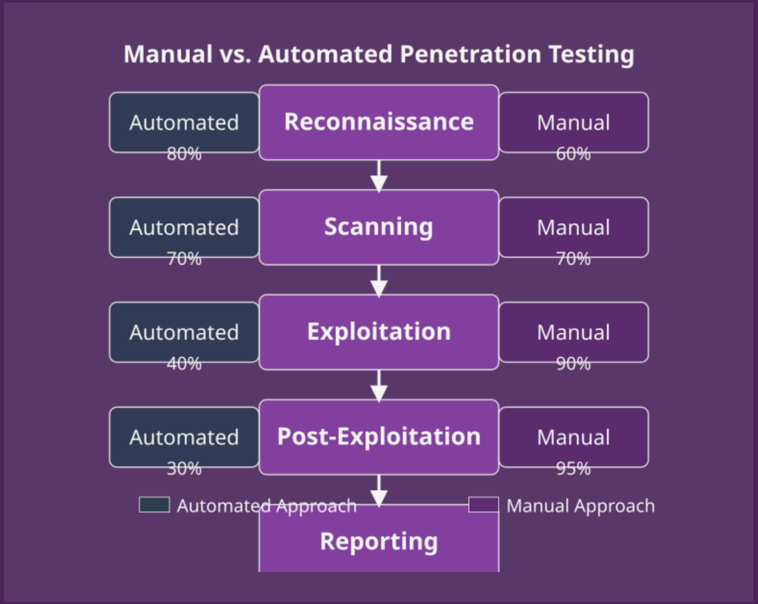
- **AI-Driven Threat Detection:** Machine learning models that establish application behavior baselines and identify anomalies without relying solely on predefined signatures
- **Behavioral Analytics:** Advanced pattern recognition that monitors user sessions, API calls, and data access patterns to detect sophisticated attacks and insider threats
- **Predictive Security:** Anticipating potential attack vectors based on application usage patterns, emerging threats, and vulnerability intelligence feeds
- **Cloud-Native Integration:** Runtime security solutions designed specifically for containerized applications, serverless functions, and service mesh architectures

Evolution of Runtime Security Capabilities



Penetration Testing: Beyond Automated Scans

- **Human Element in Testing:** Skilled penetration testers bring creativity, intuition, and adaptability that automated tools lack, enabling discovery of complex vulnerabilities
- **Hybrid Approach:** Combining automated scanning for breadth with manual testing for depth, leveraging tools for reconnaissance while applying human expertise for exploitation
- **Custom Exploit Development:** Creating tailored exploits for application-specific vulnerabilities that standard tools cannot detect, including business logic flaws
- **Attack Chaining:** Connecting multiple low-severity vulnerabilities to demonstrate high-impact attack paths that automated tools would report as isolated, low-risk issues



Practical Fuzzing for Vulnerability Discovery

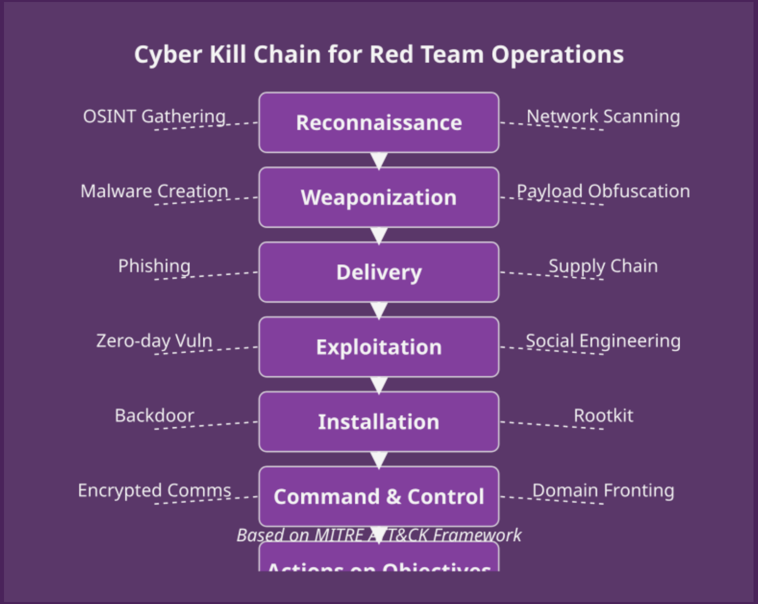
- **Fuzzing Techniques:** Mutation-based (modifying existing inputs), generation-based (creating inputs from scratch), and evolutionary fuzzing (using genetic algorithms)
- **Modern Fuzzing Tools:** AFL++, libFuzzer, Jazzer, Honggfuzz, and Mayhem for advanced coverage-guided fuzzing across different programming languages
- **Target Selection:** Prioritizing attack surface components that handle untrusted input, parse complex formats, or implement custom protocols
- **Measuring Effectiveness:** Tracking code coverage, unique crashes, execution paths, and time-to-first-bug metrics to optimize fuzzing campaigns

AFL++ Fuzzing Example

```
# Compile target with instrumentation
$ CC=afl-clang-fast CXX=afl-clang-fast++ \
  ./configure && make


[illegible]
[illegible]
$ cp testcases/* inputs/


# Start fuzzing campaign
[illegible]
  -m none -t 1000 \
  ./target @@


# Analyze crashes
[illegible]
    [illegible]
      -ex "bt" \
      -ex "quit" \
      [illegible]
  done
```

Red Teaming and Adversary Emulation

- **Adversary Mindset:** Adopting the perspective, techniques, and objectives of real-world attackers to identify security gaps that standard testing might miss
- **MITRE ATT&CK Framework:** Leveraging this knowledge base of adversary tactics and techniques to structure realistic attack scenarios and evaluate security controls
- **Attack Playbooks:** Developing detailed scenarios that simulate specific threat actors, including their tools, techniques, and procedures (TTPs)
- **Purple Teaming:** Collaborative exercises where red teams (attackers) and blue teams (defenders) work together to improve detection and response capabilities



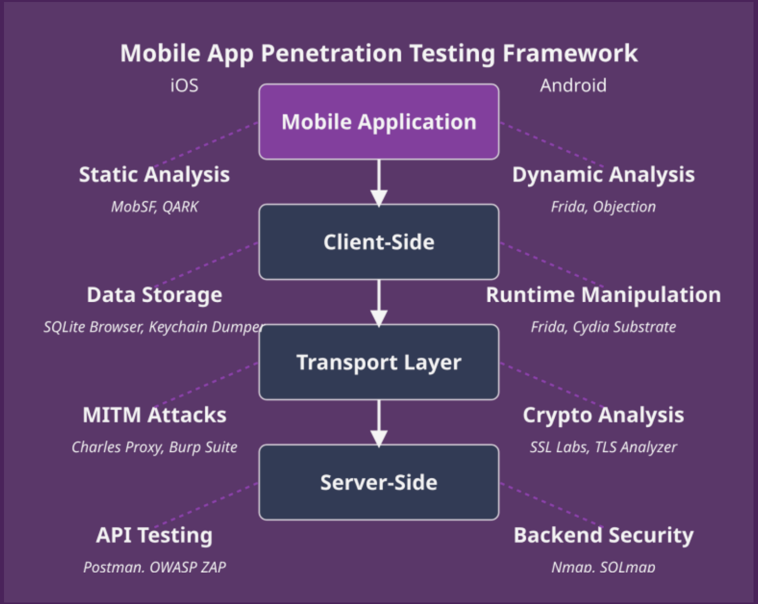
Cloud and Container Security Testing

- **Cloud-Specific Vulnerabilities:** Misconfigurations, excessive permissions, insecure APIs, and shared responsibility model gaps that create unique attack vectors
- **Container Security Challenges:** Image vulnerabilities, runtime protection, orchestration security, and escape vulnerabilities that can compromise host systems
- **Specialized Testing Tools:** Cloud Security Posture Management (CSPM), Infrastructure as Code (IaC) scanners, container image scanners, and runtime security monitoring
- **Security Testing Approach:** Combining automated compliance scanning, penetration testing, and chaos engineering to identify resilience gaps in cloud-native applications

Cloud & Container Security Testing Focus Areas

Mobile Application Penetration Testing

- **Platform-Specific Vulnerabilities:** iOS security model bypasses, Android intent hijacking, WebView exploits, and platform-specific permission models
- **Data Storage Security:** Testing for insecure local storage, keychain/keystore misuse, sensitive data in logs, and clipboard vulnerabilities
- **Communication Security:** Certificate pinning bypass, man-in-the-middle attacks, insecure API communications, and deep link vulnerabilities
- **Testing Tools & Techniques:** MobSF for static analysis, Frida for runtime manipulation, Charles Proxy for traffic interception, and app repackaging for code injection



Ethical Hacking and Responsible Disclosure

- **Ethical Framework:** Operating within legal boundaries, obtaining proper authorization, respecting privacy, minimizing damage, and maintaining confidentiality
- **Rules of Engagement:** Establishing clear scope, timeline, permitted techniques, communication channels, and escalation procedures before testing begins
- **Responsible Disclosure Process:** Privately reporting vulnerabilities to organizations, providing adequate time for remediation, and coordinating public disclosure
- **Bug Bounty Programs:** Structured programs that provide safe harbor for security researchers and incentivize the discovery and responsible reporting of vulnerabilities

Key Takeaways and Best Practices

- **Defense in Depth:** Implement multiple security testing techniques in combination rather than relying on a single approach to identify vulnerabilities
- **Continuous Security Testing:** Integrate security testing throughout the development lifecycle, not just as a final gate before production deployment
- **Automation + Human Expertise:** Balance automated security testing tools with manual testing and expert analysis for comprehensive coverage
- **Security as Education:** Use security testing results as learning opportunities to improve developer awareness and coding practices

Security Testing Effectiveness by Approach