Module 4: Software Vulnerabilities

CYBR 515: Software Security
Summer 2024
Mohammad Jamil Ahmad, Ph.D.

WestVirginiaUniversity.
JOHN CHAMBERS COLLEGE OF BUSINESS AND ECONOMICS

Table of Contents

- Injection attacks
- Broken authentication and session management
- Request forgery
- Language-specific defenses
- Example of enterprise web defenses
- Tools for vulnerability detection in source code repositories

Web Application Vulnerabilities

- **Definition:** Web application vulnerabilities are weaknesses in the security of web applications that can be exploited by attackers.
- **Consequences:** Data breaches, unauthorized access, code execution.
- **Focus Topics:**
  1. Injection attacks
  2. Broken authentication and session management
  3. Request forgery
  4. Language-specific defenses
  5. Enterprise web defenses

1. Injection Attacks Overview

- **Definition:** Malicious attempts to inject unauthorized code or commands.
- **Common Types:**
  A. SQL Injection (SQLi)
  B. Cross-Site Scripting (XSS)
  C. Command Injection
  D. Lightweight Directory Access Protocol (LDAP) Injection
  E. XPath Injection
- **Real-world Example:** ResumeLooters stole data from 2 million job seekers using SQLi and XSS.

1.A. SQL Injection (SQLi)

- Definition: Attackers inject malicious SQL statements into input fields.
- Impact: Unauthorized access, data disclosure, data manipulation.
- Example: Injecting ' OR '1'='1' -- into a login field to bypass authentication.
- **Detection Tools:**
  - SQLMap
  - Burp Suite
  - Bandit

SQLi example

Vulnerable code

```python
import sqlite3

# Vulnerable code
def vulnerable_login(username, password):
    conn = sqlite3.connect('example.db')
    cursor = conn.cursor()
    query = f"SELECT * FROM users WHERE username = '{username}' AND password = '{password}'"
    cursor.execute(query)
    return cursor.fetchone()

# Simulating an attack
username = "admin"
password = "' OR '1'='1"
print(vulnerable_login(username, password))
```

Fix: Use parameterized queries

```python
def secure_login(username, password):
    conn = sqlite3.connect('example.db')
    cursor = conn.cursor()
    query = "SELECT * FROM users WHERE username = ? AND password = ?"
    cursor.execute(query, (username, password))
    return cursor.fetchone()
```

1.B. Cross-Site Scripting (XSS)

- Definition: Malicious scripts (JavaScript) injected into web pages.
- Impact: Data theft, session hijacking, website defacement.
- Example: Injecting <script>alert('XSS');</script> into a comment field.
- **Detection Tools:**
  - OWASP ZAP
  - XSS Me

1.C. Command Injection

- Definition: Malicious commands injected into input fields executed by the system.
- Impact: Unauthorized access, remote code execution.
- Example: Injecting ; ls into a file input field.
- **Detection Tools:**
  - Burp Suite
  - Veracode

1.D. LDAP Injection

- Definition: Malicious LDAP statements injected to manipulate authentication processes.
- Impact: Unauthorized access, data disclosure.
- Example: Injecting `*)(uid=*))(|(uid=*` into an LDAP search filter to bypass authentication.
- **Detection Tools:**
  - OWASP ZAP
  - LDAPSoft

1.E. XPath Injection

- Definition: Malicious XPath queries injected to manipulate XML data retrieval.
- Impact: Unauthorized access, data disclosure.
- Example: Injecting ') or '1'='1 into an XPath query to bypass authentication.
- **Detection Tools:**
  - Burp Suite
  - OWASP ZAP

Mitigating the risks of injection attacks

1. Parameterized statements
2. Input validation and sanitization
3. Least privilege principle:
4. Web Application Firewalls (WAFs)
5. Regular security audits
6. Security headers

1. Parameterized statements

- Parameterized statements (or prepared statements) are a way of structuring SQL queries so that input data is treated as a parameter rather than part of the SQL command itself. This helps prevent SQL injection attacks by separating the code from the data.

2. Input Validation and Sanitization

- Input validation ensures that user input is in the expected format and meets specific criteria before processing. Sanitization cleans user input to remove or encode any potentially harmful characters.
- How to Implement:
- In Python, use libraries such as *validators* or custom validation functions.

```python
from markupsafe import escape
import re

def is_valid_username(username):
    return re.match("^[a-zA-Z0-9_]*$", username) is not None

def process_user_input(input_data):
    if is_valid_username(input_data):
        sanitized_input = escape(input_data)
        return sanitized_input
    else:
        raise ValueError("Invalid input")
```

3. Least Privilege Principle

- The least privilege principle restricts access rights for users and applications to only what is necessary to perform their functions, reducing the risk of unauthorized access or modifications.
- **How to Implement:**
  - Configure database user roles with minimal permissions.

```
CREATE ROLE readonly;
GRANT CONNECT ON DATABASE mydb TO readonly;
GRANT USAGE ON SCHEMA public TO readonly;
GRANT SELECT ON ALL TABLES IN SCHEMA public TO readonly;
ALTER DEFAULT PRIVILEGES IN SCHEMA public GRANT SELECT ON TABLES TO readonly;
```

4. Web Application Firewalls (WAFs)

- A Web Application Firewall (WAF) is a security system that monitors, filters, and blocks HTTP traffic to and from a web application, protecting it from various attacks, including SQL injection and XSS.
- **How to Implement:**
  - Use a WAF service like AWS WAF or Cloudflare WAF.
- **Example:** Setting up AWS WAF:
  1. Create a Web ACL in AWS WAF.
  2. Add rules to the Web ACL to block or allow traffic based on conditions like IP addresses, query strings, and request size.
  3. Associate the Web ACL with an AWS resource such as an Amazon CloudFront distribution or an Application Load Balancer.

5. Regular Security Audits

- Regular security audits involve systematically reviewing and assessing the security of an application or system to identify and address vulnerabilities.
- **How to Implement:**
  - Use automated tools like OWASP ZAP or manual penetration testing.

6. Security Headers

- Security headers are a key part of securing web applications. They instruct web browsers on how to handle content, mitigate various types of attacks, and enforce security policies.
- Security headers are HTTP response headers that help enhance the security of web applications by instructing browsers on how to behave when handling your site's content.

A. **Content Security Policy (CSP)**:
B. **X-Content-Type-Options**:
C. **X-Frame-Options**
D. **X-XSS-Protection**
E. **HTTP Strict Transport Security (HSTS)**
F. **Referrer-Policy**:
G. **Cross-Origin Resource Sharing (CORS).**
H. **Feature Policy**
I. **HTTP Public Key Pinning (HPKP)**
J. **Incident response plan**

6.A. Content Security Policy (CSP)

- Content Security Policy (CSP) allows you to define a whitelist of trusted content sources such as scripts, stylesheets, images, and fonts. This helps prevent XSS attacks by restricting the sources from which content can be loaded.

```python
from flask import Flask, make_response

app = Flask(__name__)

@app.after_request
def add_security_headers(response):
    response.headers['Content-Security-Policy'
    return response

@app.route('/')
def index():
    return "<h1>Content Security Policy Exampl

if __name__ == '__main__':
    app.run()
```

6.B. X-Content-Type-Options

- The X-Content-Type-Options header set to nosniff instructs browsers not to override the detected MIME type of a resource, helping prevent MIME-sniffing attacks.

```
@app.after_request
def add_security_headers(response):
    response.headers['X-Content-Type-Options'] = "nosniff"
    return response
```

6.C. X-Frame-Options

- The X-Frame-Options header can be set to DENY or SAMEORIGIN to prevent clickjacking attacks by specifying whether a browser should be allowed to render a page in a frame or iframe.

```
@app.after_request
def add_security_headers(response):
    response.headers['X-Frame-Options'] = "DENY"
    return response
```

6.D. X-XSS-Protection

- The X-XSS-Protection header enables the browser's built-in XSS protection mechanisms, which can help detect and mitigate XSS attacks.

```
@app.after_request
def add_security_headers(response):
    response.headers['X-XSS-Protection'] = "1; mode=block"
    return response
```

6.E. HTTP Strict Transport Security (HSTS)

- HTTP Strict Transport Security (HSTS) instructs browsers to only access a website over HTTPS, preventing man-in-the-middle attacks and protocol downgrade attacks.

```
:
@app.after_request
def add_security_headers(response):
    response.headers['Strict-Transport-Security'] = "max-age=31536000; includeSubDomains"
    return response
```

6.F. Referrer-Policy

- The Referrer-Policy header controls how much referrer information is included in requests, helping to prevent information leakage.

```
@app.after_request
def add_security_headers(response):
    response.headers['Referrer-Policy'] = "strict-origin-when-cross-origin"
    return response
```

6.G. Cross-Origin Resource Sharing (CORS)

- Cross-Origin Resource Sharing (CORS) headers control which domains can access resources on your server, mitigating unauthorized cross-origin req uests such as CSRF.

```python
from flask_cors import CORS

app = Flask(__name__)
CORS(app, resources={r"/*": {"origins": "*"}})

@app.route('/')
def index():
    return "<h1>CORS Example</h1>"

if __name__ == '__main__':
    app.run()
```

6.H. Feature Policy

- Feature Policy allows you to selectively enable, disable, or restrict browser features and APIs, mitigating various attacks and vulnerabilities.

```python
@app.after_request
def add_security_headers(response):
    response.headers['Feature-Policy'] = "geolocation 'none'; microphone 'none'"
    return response
```

Example Flask Application with All Security Headers

```
from flask import Flask, make_response
from flask_cors import CORS
from markupsafe import escape

app = Flask(__name__)
CORS(app, resources={r"/*": {"origins": "*"}})

@app.after_request
def add_security_headers(response):
    response.headers['Content-Security-Policy'] = "default-src 'self'"
    response.headers['X-Content-Type-Options'] = "nosniff"
    response.headers['X-Frame-Options'] = "DENY"
    response.headers['X-XSS-Protection'] = "1; mode=block"
    response.headers['Strict-Transport-Security'] = "max-age=31536000; includeSubDomains"
    response.headers['Referrer-Policy'] = "strict-origin-when-cross-origin"
    response.headers['Feature-Policy'] = "geolocation 'none'; microphone 'none'"
    return response

@app.route('/')
def index():
    return "<h1>Security Headers Example</h1>"

if __name__ == '__main__':
    app.run()
```

How to Detect and Verify Security Headers

- **Using Tools:**
  - **Security Headers.com**: A free online tool to test the HTTP response headers of your website.
  - **OWASP ZAP**: An open-source web application security scanner to verify if security headers are correctly implemented.
- **Testing with SecurityHeaders.com:**
  1. Deploy your Flask application.
  2. Visit SecurityHeaders.com.
  3. Enter your web application's URL and click "Scan".
  4. Review the results to ensure all headers are correctly implemented.

2. Broken authentication and session management

- **Definition:** Security vulnerabilities that lead to unauthorized access and identity theft.
- **Impact:** Can result in major security breaches, such as the Equifax data breach of 2017, exposing sensitive information of 147 million consumers.
- **Example Incident:**
  - **Equifax Data Breach (2017):** A well-known example of broken authentication leading to a massive data breach.

What is Broken Authentication?

- **Definition:** Occurs when attackers exploit flaws in the authentication process to gain unauthorized access to user accounts.
- **Common Issues:**
  1. **Weak password policies:**
     1. Lack of complexity requirements.
     2. No enforcement of strong password policies.
  2. **Credential stuffing:**
     1. Attackers use leaked username-password pairs from one service to access another service.
  3. **Insecure session management:**
     1. Poorly managed user sessions, leading to session fixation or hijacking.

How to Fix Broken Authentication

- **Mitigation Measures:**
  - Enforce strong password policies.
  - Use Multi-Factor Authentication (MFA).
  - Protect against credential stuffing by monitoring and blocking suspicious login attempts.

Session Management

- **Definition:** Secure creation, maintenance, and termination of user sessions after authentication.
- **Common Issues:**
  1. **Session fixation:** Attackers set or hijack a user's session ID.
  2. **Session timeout:** Lack of proper session timeout policies.
  3. **Session invalidation:** Inability to properly invalidate or destroy sessions.

How to Fix Session Management

- **Mitigation Measures:**
  - Use secure session management practices, including encryption and secure cookie attributes.
  - Implement session timeout to log out inactive users.
  - Generate random and unique session identifiers.
  - Invalidate or regenerate session IDs upon login or privilege changes.

Token-Based Authentication

- **Definition:** Uses tokens such as JSON Web Tokens (JWTs) for enhanced security.

Regular Security Audits and Compliance

- **Importance:** Regular audits and penetration testing to identify and address vulnerabilities.
- **Compliance:** Ensure adherence to security standards such as PCI DSS.

Education and Training

- **Importance:** Raise awareness among developers about secure authentication and session management practices.
- **Regular Training:** Keep developers updated on the latest security threats and best practices.

3. Request Forgery

- **Introduction to Cross-Site Request Forgery (CSRF)**
- **Definition:** A web security vulnerability where an attacker tricks a user's browser into performing an unwanted action on a trusted site where the user is authenticated.
- **Impact:** Can lead to unauthorized actions such as changing account settings or making purchases.
- **Example Incident:**
  - **Samy Worm (2008):** A CSRF attack on the WordPress blogging application created by Samy Kamkar.

CSRF Attack Scenario

1. **Setup:** An attacker tricks a user into loading a page with a malicious request.
2. **Execution:** The request is sent to a target site where the user is authenticated.
3. **Session Exploitation:** The browser includes the user's session cookie, making the request appear legitimate.
4. **Unintended Action:** The target site processes the malicious request, leading to unintended actions.

Example of Vulnerable Code (CSRF)

- **Vulnerability:** This example lacks protection against CSRF, allowing unauthorized requests.
- **Detection Tools:**

  OWASP ZAP: Can be used to scan for CSRF vulnerabilities.

```python
# Vulnerable Code: No CSRF Protection
from flask import Flask, request, render_template_string

app = Flask(__name__)

@app.route('/transfer', methods=['POST'])
def transfer():
    amount = request.form['amount']
    recipient = request.form['recipient']
    # Process the transfer
    return f"Transferred {amount} to {recipient}"

if __name__ == '__main__':
    app.run()
```

How to Fix CSRF

- Fixing CSRF with Anti-CSRF Tokens:
  Mitigation Measures:
  - Use anti-CSRF tokens in forms and requests.
  - Set SameSite attribute on cookies.
  - Check the Referer header on incoming requests.
  - Include custom headers in requests.

```python
# Fix: Using CSRF Tokens
from flask import Flask, request, render_template_string, session
from flask_wtf.csrf import CSRFProtect, generate_csrf

app = Flask(__name__)
app.secret_key = 'supersecretkey'
csrf = CSRFProtect(app)

@app.route('/form')
def form():
    csrf_token = generate_csrf()
    return render_template_string('''
        <form method="POST" action="/transfer">
            <input type="hidden" name="csrf_token" value="{{ csrf_token }}">
            Amount: <input type="text" name="amount">
            Recipient: <input type="text" name="recipient">
            <input type="submit" value="Transfer">
        </form>
    ''', csrf_token=csrf_token)

@app.route('/transfer', methods=['POST'])
def transfer():
    amount = request.form['amount']
    recipient = request.form['recipient']
    # Process the transfer
```

Preventive Measures Against CSRF

1. **Anti-CSRF Tokens:**
   - Unique tokens included in forms and requests to validate legitimacy.
2. **SameSite Cookie Attribute:**
   - Controls when cookies are sent with cross-site requests.
3. **Referer Header Checking:**
   - Ensures requests originate from the same domain.
4. **Custom Headers:**
   - Used in requests to validate their legitimacy.
5. **Double-Submit Cookies:**
   - Token stored in both cookie and request parameter for verification.
6. **Content-Type Validation:**
   - Ensures correct Content-Type header in requests.

Implementing Best Practices

- **Secure Coding Practices:** Avoid introducing vulnerabilities that CSRF attacks could exploit.
- **Session Management:** Implement secure session management practices, including session timeout and secure session token generation.
- **Security Audits:** Regularly conduct security audits and penetration testing.
- **Education and Training:** Provide training for developers on secure coding practices and CSRF prevention.

4. Language-Specific Defenses

- **Definition:** Security measures tailored to the programming language used in web applications.
- **Importance:** Protects against common web attacks like SQL injection, XSS, and CSRF by leveraging language-specific tools and practices.
  1. **PHP Defenses**
  2. **Java Defenses**
  3. **Python Defenses**
  4. **Node.js (JavaScript)**
  5. **Ruby (Ruby on Rails)**

PHP Defenses: Filter Input Data

- Filter Input Data: Clean up what users type into the website to remove harmful stuff.
- **Example:**
  - Before: Accepting any input without checking.
  - After: Using filter_input() to clean the input.

```php
$username = filter_input(INPUT_POST, 'username', FILTER_SANITIZE_STRING);
```

- Use functions l

```php
$username = filter_input(INPUT_POST, 'username', FILTER_SANITIZE_STRING);
$password = mysqli_real_escape_string($conn, $_POST['password']);
```

`mysqli_real_escape_string()` to sanitize user input.

PHP Defenses: Prepared Statements

- Use a safe way to handle database queries to prevent attacks.
- **Example:**

```
$stmt = $conn->prepare("SELECT * FROM users WHERE username = ?");
$stmt->bind_param("s", $username);
$stmt->execute();
```

PHP Defenses: Cross-Site Scripting (XSS) Prevention

- Prevent harmful scripts from running on your website.
- **Example:**

```
echo htmlspecialchars($user_input, ENT_QUOTES, 'UTF-8');
```

PHP Defenses: Session Security

- Keep user sessions secure by using strong session handling practices.
- **Example:**

```
session_start();session_regenerate_id(true);
htmlspecialchars($user_input, ENT_QUOTES, 'UTF-8');
```

PHP Defenses: Content Security Policy (CSP)

- Control which resources (like scripts and images) can be loaded to prevent attacks.
- **Example:**

```
header("Content-Security-Policy: default-src 'self'");
```

Java Defenses: Input Validation

- Check and clean user input to prevent harmful data.

- Example:

```
String username = request.getParameter("username");
if (username != null && !username.matches("[A-Za-z0-9_]+"))
{
    throw new IllegalArgumentException("Invalid username");
}
```

Java Defenses: Hibernate ORM

- Use a safe method for database interactions.
- Example:

```
Query query = session.createQuery("from User where username =
:username");
query.setParameter("username", username);
```

Java Defenses: Anti-CSRF Tokens

- Use unique codes to ensure requests are legitimate.
- Example:

```
@Controller
public class FormController {
      @GetMapping("/form")
      public String showForm(Model model) {
              model.addAttribute("csrfToken", generateCsrfToken());
               return "form";
   }}
```

Java Defenses: Security Headers

- Use headers to enhance security.
- Example:

```
@RestController
public class SecurityController {
@GetMapping("/secure")
public ResponseEntity<String> secure() {
HttpHeaders headers = new HttpHeaders();
headers.add("Strict-Transport-Security", "max-age=31536000;
includeSubDomains");
return new ResponseEntity<>("Secure Content", headers, HttpStatus.OK);
} }
```

Rest of examples are shown in the book

PMD for static analysis

- **PMD:** An open-source static analysis tool that scans Java and other languages for potential bugs, code smells, and security vulnerabilities. It provides customizable rulesets and integrates with various IDEs and build tools.
- Manual is posted on eCampus.

PMD key features

1. **Detects Common Code Issues:**
    1. Finds bugs, dead code, suboptimal code practices, duplicate code, and possible performance issues.
    2. Identifies potential security vulnerabilities.
2. **Multi-Language Support:**
    1. Primarily supports Java but also supports JavaScript, XML, XSL, PLSQL, Apache Velocity, and others.
3. **Customizable Rulesets:**
    1. Users can create custom rulesets tailored to their specific needs.
    2. PMD includes a large number of predefined rules for various coding standards and best practices.

PMD key features

4. **Integrations:**
    4. Integrates with various IDEs (e.g., Eclipse, IntelliJ IDEA, NetBeans).
    5. Supports build tools like Maven, Gradle, and Ant.
    6. Can be used in CI/CD pipelines to ensure code quality during the development process.
5. **Detailed Reports:**
    1. Generates detailed reports in various formats (HTML, XML, CSV, text) highlighting identified issues and providing suggestions for remediation.

OWASP Dependency-Check

- An open-source tool that helps identify project dependencies with known vulnerabilities.
- It is part of the OWASP (Open Web Application Security Project) suite of security tools. Dependency-Check is particularly useful for ensuring that the libraries and frameworks your project relies on do not introduce security risks.

OWASP Dependency-Check key features

1. **Vulnerability Identification:**
    1. Detects vulnerabilities in third-party libraries and frameworks used in your project.
    2. Utilizes databases like the NVD, NPM Audit, Retire.js, and OSS Index to identify known vulnerabilities.
2. **Multi-Language Support:**
    1. Supports a wide range of programming languages and package managers, including Java (Maven, Gradle), .NET (NuGet), JavaScript (npm, Yarn), Python (Pip), Ruby (Bundler), PHP (Composer), and more.

OWASP Dependency-Check key features

3. **Integration:**
    3. Easily integrates with CI/CD pipelines and build tools like Jenkins, Maven, Gradle, and others.
    4. Provides plugins and CLI tools for different development environments.
4. **Detailed Reports:**
    3. Generates detailed reports highlighting the vulnerabilities, their severity, and recommendations for remediation.
    4. Supports various report formats, including HTML, XML, JSON, and CSV.

Bandit

- **Bandit** is an open-source security linter for Python code. Developed by the OpenStack Security Project, Bandit analyzes Python programs to find common security issues. It inspects the abstract syntax tree (AST) of each Python file to identify potential vulnerabilities and coding mistakes.
- **Key Features of Bandit:**

Banidt Features

- Security Checks: Bandit includes a wide range of security checks to identify common issues such as:

- Hardcoded passwords
- Use of weak cryptographic methods
- Potential command injection vulnerabilities
- Insecure use of functions like eval
- Insecure handling of user input

Banidt Features

- Customizability: Users can configure Bandit to include or exclude specific tests, define severity levels, and customize output formats.
- Integration: Bandit can be easily integrated into CI/CD pipelines to automatically scan code during development and deployment processes.
- Detailed Reports: Generates detailed reports highlighting the severity and location of identified issues, along with suggestions for remediation

Snyk

- **Snyk** is a developer-first security platform that helps organizations find and fix vulnerabilities in their code, dependencies, container images, and infrastructure as code (IaC).
- Snyk integrates seamlessly into the development workflow, enabling developers to secure their applications from the start and continuously throughout the software development lifecycle

Snyk features

1. **Vulnerability Scanning:**
    1. Scans for vulnerabilities in open-source dependencies, container images, and IaC configurations.
    2. Provides detailed information about identified vulnerabilities, including severity, exploitability, and potential impact.
2. **Automated Fixes:**
    1. Suggests and automates the application of patches and upgrades to fix identified vulnerabilities.
    2. Provides pull requests with fixes directly to your version control system (VCS).
3. **Developer-Friendly Integration:**
    1. Integrates with popular development tools, IDEs, CI/CD pipelines, and VCS platforms such as GitHub, GitLab, Bitbucket, and Azure Repos.
    2. Supports multiple programming languages and package managers, including JavaScript (npm, Yarn), Python (pip), Java (Maven, Gradle), Ruby (Bundler), and more.

Snyk Featuyers

4. **License Compliance:**
    4. Checks for license compliance issues in open-source dependencies.
    5. Helps ensure that all dependencies meet organizational licensing policies.
5. **Infrastructure as Code (IaC) Security:**
    4. Scans IaC configurations (like Terraform and CloudFormation) for security issues.
    5. Provides best practices for securing infrastructure configurations.
6. **Container Security:**
    4. Scans container images for vulnerabilities and provides actionable insights to secure containerized applications.
    5. Integrates with container registries and CI/CD pipelines to ensure secure image deployment.

Comparison of tools

Criteria	PMD	OWASP Dependency-Check	Bandit	Snyk
Primary Function	Static code analysis	Vulnerability scanning in dependencies	Security linter for Python code	Comprehensive security platform
Supported Languages	Java, JavaScript, XML, XSL, PLSQL, Apache Velocity, and more	Java, .NET, JavaScript, Python, Ruby, PHP, and more	Python	Java, JavaScript, Python, Ruby, .NET, PHP, Go, and more
Platform Availability	Windows, macOS, Linux	Windows, macOS, Linux	Windows, macOS, Linux	Windows, macOS, Linux
Integration	IDEs (Eclipse, IntelliJ), build tools (Maven, Gradle, Ant)	Build tools (Maven, Gradle), CI/CD tools (Jenkins)	CI/CD pipelines	IDEs, VCS (GitHub, GitLab, Bitbucket), CI/CD tools (Jenkins, GitHub Actions)
Detection Capabilities	Bugs, dead code, code smells, security vulnerabilities	Known vulnerabilities in dependencies	Security issues like hardcoded passwords, weak cryptography, etc.	Vulnerabilities in code, dependencies, containers, and IaC
Automated Fixes	No	No	No	Yes
Custom Rulesets	Yes	No	Yes	No (but offers remediation advice)
Detailed Reports	Yes (HTML, XML, CSV, text)	Yes (HTML, XML, JSON, CSV)	Yes (text, JSON)	Yes (HTML, JSON, interactive dashboard)
License Compliance	No	No	No	Yes
Community Support	Strong community and active development	Strong community and active development	Strong community and active development	Strong community and active development