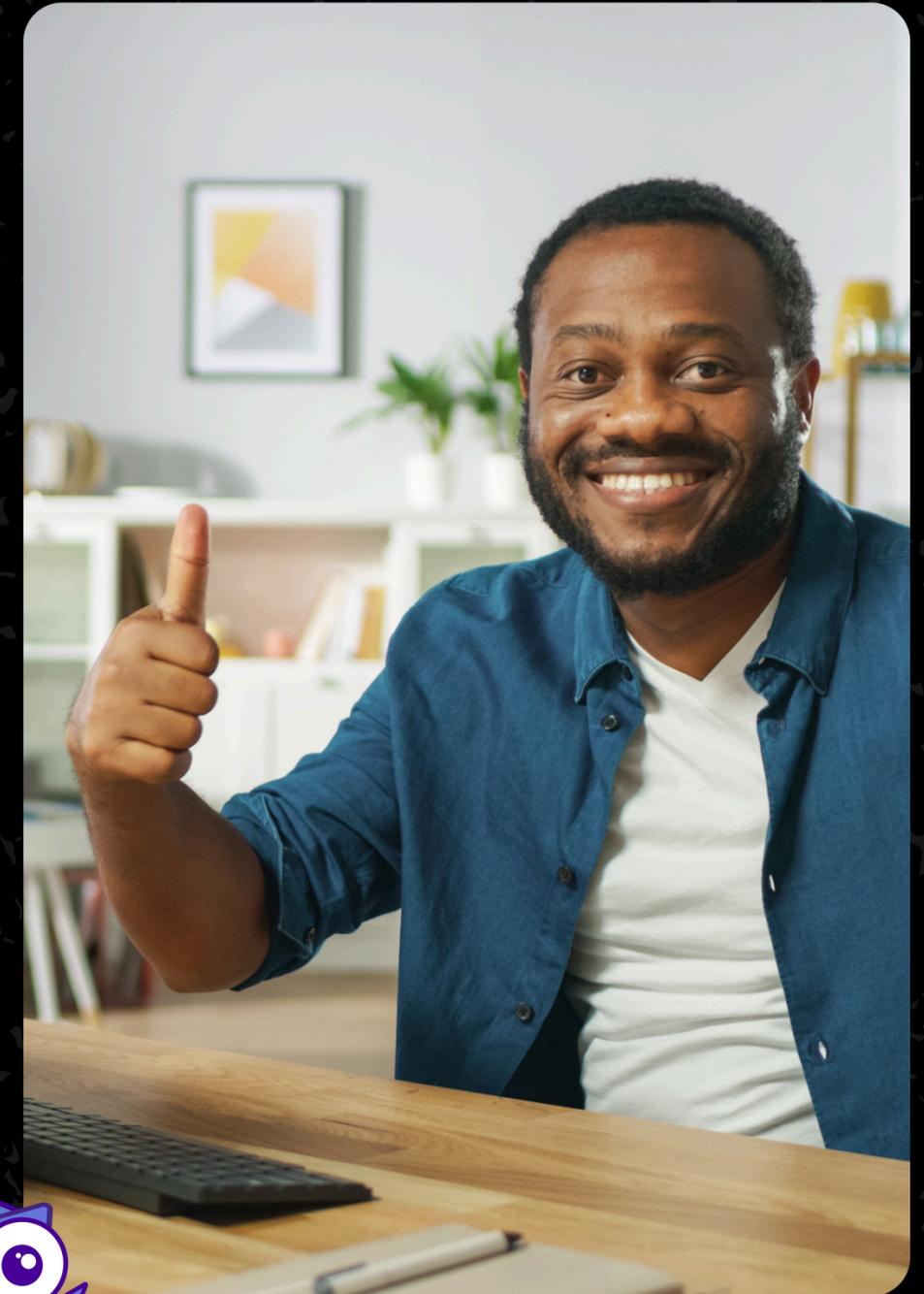


INTEGRATING SECURITY INTO THE DEVOPS WORKFLOW

Module Overview:

This training module will help you understand how to identify and manage vulnerabilities throughout the software development lifecycle. By incorporating security practices seamlessly into your development workflow, you'll be able to mitigate risks without compromising on productivity or user experience. The concept of integrating security directly into DevOps (often referred to as DevSecOps) will be woven throughout the module, highlighting the need for continuous security practices.



92%

of companies experienced a breach last year due to vulnerabilities of applications developed in-house.



CONTINUOUS VULNERABILITY DETECTION



Objectives:

Understand the OWASP Top 10

Understand how DevSecOps practices ensure continuous monitoring of vulnerabilities to secure the evolving application environment.

OWASP TOP 10

The OWASP Top 10 is a standard awareness document for developers and web application security. It represents a broad consensus about the most critical security risks to web applications.

- Broken Access Control
- Cryptographic Failures
- Injection
- Insecure Design
- Security Misconfiguration
- Vulnerable and Outdated Components
- Identification and Authentication Failures
- Software and Data Integrity Failures
- Security Logging and Monitoring Failures
- Server-Side Request Forgery

BROKEN ACCESS CONTROL

Access control enforces policy such that users cannot act outside of their intended permissions. Failures typically lead to unauthorized information disclosure, modification, or destruction of all data or performing a business function outside the user's limits.

Access control is only effective in trusted server-side code or server-less API, where the attacker cannot modify the access control check or metadata.

Example Attack Scenario

Scenario #1: The application uses unverified data in a SQL call that is accessing account information:

```
pstmt.setString(1, request.getParameter("acct"));  
ResultSet results = pstmt.executeQuery( );
```

An attacker simply modifies the browser's 'acct' parameter to send whatever account number they want. If not correctly verified, the attacker can access any user's account.

```
https://example.com/app/accountInfo?acct=notmyacct
```

Learn more about Broken Access Control [here](#)



CRYPTOGRAPHIC FAILURES

The first thing is to determine the protection needs of data in transit and at rest. For example, passwords, credit card numbers, health records, personal information, and business secrets require extra protection, mainly if that data falls under privacy laws, e.g., EU's General Data Protection Regulation (GDPR), or regulations, e.g., financial data protection such as PCI Data Security Standard (PCI DSS).

Example Attack Scenarios

Scenario #1: An application encrypts credit card numbers in a database using automatic database encryption. However, this data is automatically decrypted when retrieved, allowing a SQL injection flaw to retrieve credit card numbers in clear text.

Scenario #2: A site doesn't use or enforce TLS for all pages or supports weak encryption. An attacker monitors network traffic (e.g., at an insecure wireless network), downgrades connections from HTTPS to HTTP, intercepts requests, and steals the user's session cookie. The attacker then replays this cookie and hijacks the user's (authenticated) session, accessing or modifying the user's private data. Instead of the above they could alter all transported data, e.g., the recipient of a money transfer.

Scenario #3: The password database uses unsalted or simple hashes to store everyone's passwords. A file upload flaw allows an attacker to retrieve the password database. All the unsalted hashes can be exposed with a rainbow table of pre-calculated hashes. Hashes generated by simple or fast hash functions may be cracked by GPUs, even if they were salted.

[Learn more about Cryptographic Failures here](#)



INJECTION

An application is vulnerable to attack when:

- User-supplied data is not validated, filtered, or sanitized by the application.
- Dynamic queries or non-parameterized calls without context-aware escaping are used directly in the interpreter.
- Hostile data is used within object-relational mapping (ORM) search parameters to extract additional, sensitive records.
- Hostile data is directly used or concatenated. The SQL or command contains the structure and malicious data in dynamic queries, commands, or stored procedures.

Some of the more common injections are SQL, NoSQL, OS command, Object Relational Mapping (ORM), LDAP, and Expression Language (EL) or Object Graph Navigation Library (OGNL) injection.

Preventing injection requires keeping data separate from commands and queries

Example Attack Scenario

Scenario #1: An application uses untrusted data in the construction of the following vulnerable SQL call:

```
String query = "SELECT \* FROM accounts WHERE custID='" + request.getParameter("id") + "'";
```

[Learn more about Injection here](#)



INSECURE DESIGN

Insecure design is a broad category representing different weaknesses, expressed as “missing or ineffective control design.” Insecure design is not the source for all other Top 10 risk categories. There is a difference between insecure design and insecure implementation. We differentiate between design flaws and implementation defects for a reason, they have different root causes and remediation. A secure design can still have implementation defects leading to vulnerabilities that may be exploited. An insecure design cannot be fixed by a perfect implementation as by definition, needed security controls were never created to defend against specific attacks. One of the factors that contribute to insecure design is the lack of business risk profiling inherent in the software or system being developed, and thus the failure to determine what level of security design is required.

Example Attack Scenario

Scenario #1: A credential recovery workflow might include “questions and answers,” which is prohibited by NIST 800-63b, the OWASP ASVS, and the OWASP Top 10. Questions and answers cannot be trusted as evidence of identity as more than one person can know the answers, which is why they are prohibited. Such code should be removed and replaced with a more secure design.



SECURITY MISCONFIGURATION

The application might be vulnerable if the application is:

- Missing appropriate security hardening across any part of the application stack or improperly configured permissions on cloud services.
- Unnecessary features are enabled or installed (e.g., unnecessary ports, services, pages, accounts, or privileges).
- Default accounts and their passwords are still enabled and unchanged.
- Error handling reveals stack traces or other overly informative error messages to users.
- For upgraded systems, the latest security features are disabled or not configured securely.
- The security settings in the application servers, application frameworks (e.g., Struts, Spring, ASP.NET), libraries, databases, etc., are not set to secure values.
- The server does not send security headers or directives, or they are not set to secure values.
- The software is out of date or vulnerable (see [A06:2021-Vulnerable and Outdated Components](#)).

Example Attack Scenario

Scenario #1: The application server comes with sample applications not removed from the production server. These sample applications have known security flaws attackers use to compromise the server. Suppose one of these applications is the admin console, and default accounts weren't changed. In that case, the attacker logs in with default passwords and takes over.

[Learn more about Security Misconfiguration here](#)



VULNERABLE AND OUTDATED COMPONENTS

You are likely vulnerable:

- If you do not know the versions of all components you use (both client-side and server-side). This includes components you directly use as well as nested dependencies.
- If the software is vulnerable, unsupported, or out of date. This includes the OS, web/application server, database management system (DBMS), applications, APIs and all components, runtime environments, and libraries.
- If you do not scan for vulnerabilities regularly and subscribe to security bulletins related to the components you use.
- If you do not fix or upgrade the underlying platform, frameworks, and dependencies in a risk-based, timely fashion. This commonly happens in environments when patching is a monthly or quarterly task under change control, leaving organizations open to days or months of unnecessary exposure to fixed vulnerabilities.
- If software developers do not test the compatibility of updated, upgraded, or patched libraries.
- If you do not secure the components' configurations (see [A05:2021-Security Misconfiguration](#)).

Every organization must ensure an ongoing plan for monitoring, triaging, and applying updates or configuration changes for the lifetime of the application or portfolio.

Example Attack Scenario

Scenario #1: Components typically run with the same privileges as the application itself, so flaws in any component can result in serious impact. Such flaws can be accidental (e.g., coding error) or intentional (e.g., a backdoor in a component). Some example exploitable component vulnerabilities discovered are:

- CVE-2017-5638, a Struts 2 remote code execution vulnerability that enables the execution of arbitrary code on the server, has been blamed for significant breaches.
- While the internet of things (IoT) is frequently difficult or impossible to patch, the importance of patching them can be great (e.g., biomedical devices).

[Learn more about Vulnerable and Outdated Components here](#)



IDENTIFICATION AND AUTHENTICATION FAILURES

Confirmation of the user's identity, authentication, and session management is critical to protect against authentication-related attacks. There may be authentication weaknesses if the application:

- Permits automated attacks such as credential stuffing, where the attacker has a list of valid usernames and passwords.
- Permits brute force or other automated attacks.
- Permits default, weak, or well-known passwords, such as "Password1" or "admin/admin".
- Uses weak or ineffective credential recovery and forgot-password processes, such as "knowledge-based answers," which cannot be made safe.
- Uses plain text, encrypted, or weakly hashed passwords data stores (see [A02:2021-Cryptographic Failures](#)).
- Has missing or ineffective multi-factor authentication.
- Exposes session identifier in the URL.
- Reuse session identifier after successful login.
- Does not correctly invalidate Session IDs. User sessions or authentication tokens (mainly single sign-on (SSO) tokens) aren't properly invalidated during logout or a period of inactivity.

Example Attack Scenario

Scenario #1: Credential stuffing, the use of lists of known passwords, is a common attack.

Suppose an application does not implement automated threat or credential stuffing protection. In that case, the application can be used as a password oracle to determine if the credentials are valid.



SOFTWARE AND DATA INTEGRITY FAILURES

Software and data integrity failures relate to code and infrastructure that does not protect against integrity violations. An example of this is where an application relies upon plugins, libraries, or modules from untrusted sources, repositories, and content delivery networks (CDNs). An insecure CI/CD pipeline can introduce the potential for unauthorized access, malicious code, or system compromise. Lastly, many applications now include auto-update functionality, where updates are downloaded without sufficient integrity verification and applied to the previously trusted application. Attackers could potentially upload their own updates to be distributed and run on all installations. Another example is where objects or data are encoded or serialized into a structure that an attacker can see and modify is vulnerable to insecure deserialization.

Example Attack Scenario

Scenario #1 Update without signing: Many home routers, set-top boxes, device firmware, and others do not verify updates via signed firmware. Unsigned firmware is a growing target for attackers and is expected to only get worse. This is a major concern as many times there is no mechanism to remediate other than to fix in a future version and wait for previous versions to age out.

[Learn more about Software and Data Integrity Failures here](#)



SECURITY LOGGING AND MONITORING FAILURES

This category is to help detect, escalate, and respond to active breaches. Without logging and monitoring, breaches cannot be detected. Insufficient logging, detection, monitoring, and active response occurs any time:

- Auditable events, such as logins, failed logins, and high-value transactions, are not logged.
- Warnings and errors generate no, inadequate, or unclear log messages.
- Logs of applications and APIs are not monitored for suspicious activity.
- Logs are only stored locally.
- Appropriate alerting thresholds and response escalation processes are not in place or effective.
- Penetration testing and scans by dynamic application security testing (DAST) tools (such as OWASP ZAP) do not trigger alerts.
- The application cannot detect, escalate, or alert for active attacks in real-time or near real-time.

Example Attack Scenario

Scenario #1: A children's health plan provider's website operator couldn't detect a breach due to a lack of monitoring and logging. An external party informed the health plan provider that an attacker had accessed and modified thousands of sensitive health records of more than 3.5 million children. A post-incident review found that the website developers had not addressed significant vulnerabilities. As there was no logging or monitoring of the system, the data breach could have been in progress since 2013, a period of more than seven years.

[Learn more about Security Logging and Monitoring Failures here](#)



SERVER-SIDE REQUEST FORGERY

SSRF flaws occur whenever a web application is fetching a remote resource without validating the user-supplied URL. It allows an attacker to coerce the application to send a crafted request to an unexpected destination, even when protected by a firewall, VPN, or another type of network access control list (ACL).

As modern web applications provide end-users with convenient features, fetching a URL becomes a common scenario. As a result, the incidence of SSRF is increasing. Also, the severity of SSRF is becoming higher due to cloud services and the complexity of architectures.

Example Attack Scenario

Attackers can use SSRF to attack systems protected behind web application firewalls, firewalls, or network ACLs, using scenarios such as:

Scenario #1: Port scan internal servers – If the network architecture is unsegmented, attackers can map out internal networks and determine if ports are open or closed on internal servers from connection results or elapsed time to connect or reject SSRF payload connections.

Scenario #2: Sensitive data exposure – Attackers can access local files or internal services to gain sensitive information such as `file:///etc/passwd` and `http://localhost:28017/`.

[Learn more about Server-Side Request Forgery here](#)



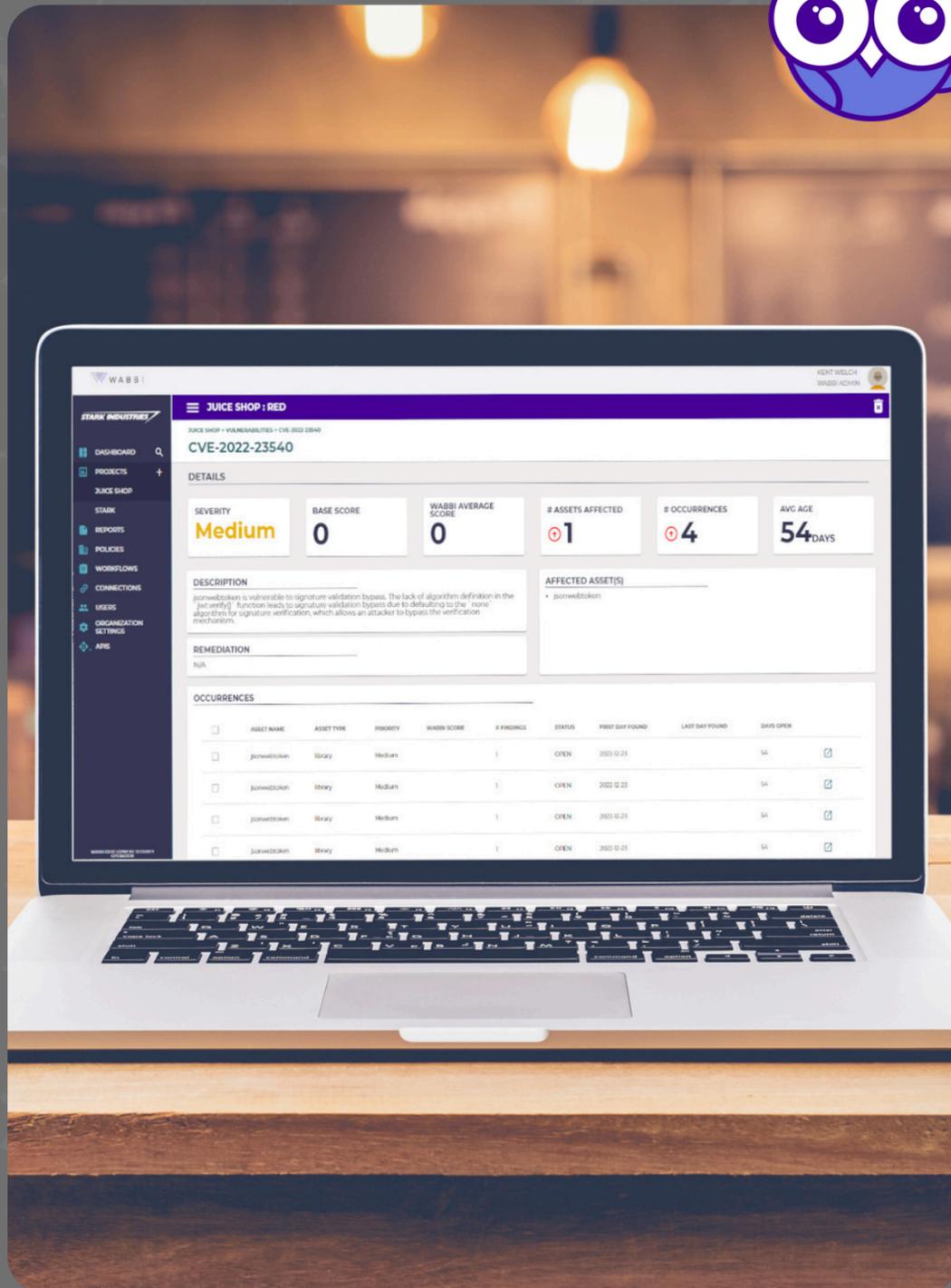


Why Continuous Monitoring is Essential

In DevOps workflows, the pace of development means that vulnerabilities can be introduced at any time, from initial code commits to deployment. Therefore, automated, continuous vulnerability scanning and testing are critical components of a secure DevOps pipeline.

DevOps Integration of Security Tools:

- **Automated Scanning in CI/CD Pipelines:** Integrate tools like SAST and DAST into your Continuous Integration/Continuous Deployment (CI/CD) pipelines to scan code and applications automatically with each commit and deployment.
- **Infrastructure-as-Code (IaC):** Ensure secure configurations through automated tests and compliance checks, preventing security misconfigurations at the infrastructure level.



KEY TOOLS

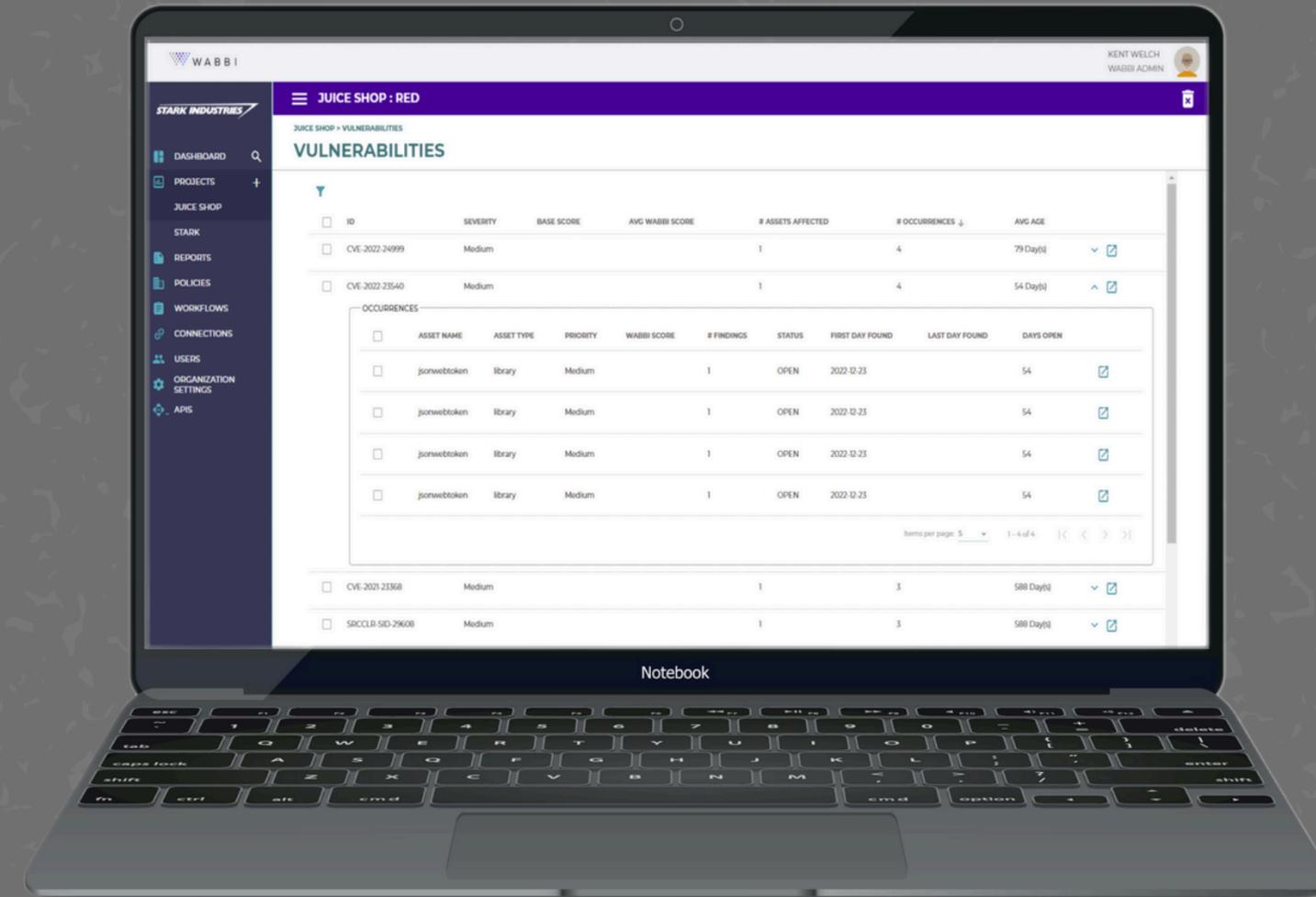


Vulnerability scanning during builds.

Automatically review and provide real-time feedback on code quality and vulnerabilities.

BEST PRACTICE TIP

Ensure that vulnerability scanning is part of every pull request and deployment stage to catch issues early without interrupting development flow.



RISK-BASED VULNERABILITY MANAGEMENT



Objectives:

Comprehend the importance of prioritizing vulnerabilities based on their severity, application risk, and business impact.

Apply DevSecOps principles to balance security with the speed of development by adopting risk-based vulnerability management strategies.

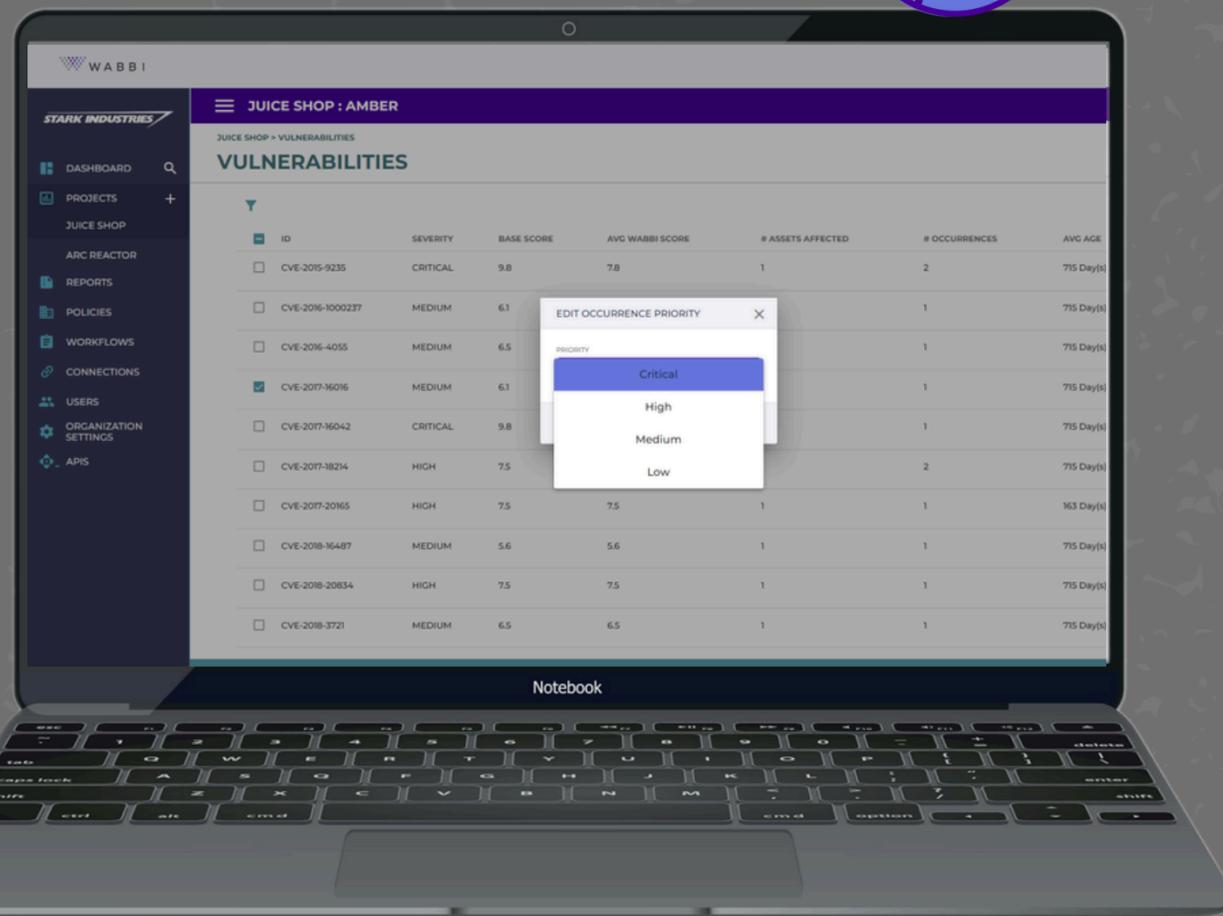


Prioritizing Vulnerabilities

Not all vulnerabilities pose the same level of risk. In a high-speed DevOps environment, it's essential to prioritize the vulnerabilities that matter most, based on their severity and potential impact on your systems and users.

Risk Profiling in DevOps:

- Assign risk scores to vulnerabilities based on the application's function and the potential business impact.
- Continuously assess vulnerabilities as part of your development pipeline, adjusting security measures as risks evolve.



How DevOps Teams Can Mitigate Risks:

- **Proactive Management:** Automate the identification of vulnerabilities early in the development lifecycle and ensure proper patching mechanisms are in place.
- **Collaboration Between Teams:** Encourage collaboration between security and development teams to balance security needs with functionality.

Tips for Triaging Vulnerabilities:

- **Prioritize on Business Impact:** Code analysis and prioritization based on business impact
- **Exploitability:** Tools like like EPSS can help you understand the likelihood a vulnerability will be exploited



ACTIVITY



Organize an Escalation of Privilege game with your team. Get the cards [here](#)!

Pick one of the OWASP top 10 and review some of your recent code with it in mind. What could you have done differently?



CONCLUSION

Security is not a standalone process, but one that should be integrated into the entire DevOps lifecycle. By embedding continuous vulnerability detection and management into every phase of development, organizations can maintain a secure application environment without compromising speed or agility. From using automated scanning tools to fostering collaboration between development and security teams, DevSecOps practices ensure vulnerabilities are detected and addressed quickly, leading to stronger applications and a more resilient security posture.



RESOURCES



[Check out the OWASP Top 10](#)



[What is DevSecOps](#)



[Understanding Vulnerability Management](#)

[Check out the World's Most Insecure App!](#)

[Learn about the OWASP DevSecOps Framework](#)



[Understand how DevSecOps Impacts different roles](#)



[Cut through the DevSecOps Noise!](#)

[Get to Know CWE](#)