Secure Coding. Practical steps to defend your web apps.

Copyright SANS Institute
Author Retains Full Rights

This paper is from the SANS Software Security site. Reposting is not permitted without express written permission.

Interested in learning more?
Check out the list of upcoming events offering "Defending Web Applications Security Essentials (DEV522)"
at http://software-security.sans.orghttp://software-security.sans.org/events/
Adapting AppSec to a DevOps World

Author: Rebecca Deck, sdsecurityacct@hotmail.com

Abstract

DevOps software development presents a fundamental challenge to traditional software security practices. Multi-day static and dynamic analysis run by a small pool of security experts is not a tenable model when the business demands multiple software releases per day. Modern system administration and quality assurance roles have adapted by using automation to empower developers to elevate code safely and as often as possible. By operating within the DevOps culture and tooling, security experts can educate developers and instrument systems in much the same way as other stakeholders in the development process. Proper abuse case development, metrics, unit, and integration testing can minimize risk while still enabling the rapid software development that businesses demand.
1. Introduction

Since its inception in 2009, the DevOps software development methodology has steadily gained in adoption, and most organizations either have implemented DevOps at some level or plan to soon (Statistica, 2017). The very concept of DevOps focuses on using automation to deliver software to production environments frequently and in a safe way. This continuous delivery is at odds with many traditional security controls like penetration testing, static, and dynamic analysis that take hours to weeks to execute. These tools often require security experts to run and further expertise to interpret results. Security teams in DevOps organizations can modify their processes to integrate with existing DevOps toolkits to ensure that software delivery is both quick and secure.

When organizations have adopted the DevOps philosophy, security teams are wise to find ways to adapt the organization. Several techniques that work well in Waterfall or Agile methodologies often meet with resistance in DevOps environments. Traditional secure software development approaches involve placing gates in the software development lifecycle (SDLC) and mandating that systems must meet security bars before elevating to production (Conklin & Shoemaker, 2014). Security teams are allotted sufficient time to perform any tests, and project managers often provide time for developers to remediate any findings before release. Attempts to insert manual gates will inhibit fundamental DevOps beliefs and may be seen by the business as efforts to stymie productivity (Kim, Humble, Debois, & Willis, 2016).

By embracing the ways of DevOps, security teams can not only maintain similar levels of efficacy but improve software and system security as well. Keys to this approach are proper abuse case development and using a DevOps toolkit to automate security testing without interrupting productivity. Once software developers have created abuse cases, they can then create unit and integration tests in the same way that they treat normal use cases. DevOps teams may choose from a large number of tools to perform unit an integration testing. This paper uses the Mocha testing framework to demonstrate a method for automated abuse case testing. Although organizations may utilize a different
Adapting AppSec to a DevOps World

Rebecca Deck, sdsecurityacct@hotmail.com

1.1. Tenants of DevOps

The DevOps methodology takes the lean and continuous improvement models originally developed for manufacturing and applies them to technology. Some of the original inspirations for DevOps include Lean, the Theory of Constraints, the Toyota Production System, and several other models (Kim, Behr, & Spafford, 2013). Although the tools for delivering software are very different from those used to deliver automobiles, the process remains the same. The ultimate goal is to develop a product that is useful to the consumer while minimizing rework and process delivery bottlenecks (Kim et al., 2016). By viewing the software delivery pipeline in the same light as a manufacturing process, organizations can then apply the same mechanisms to their process. While a complete review of DevOps is beyond the scope of this paper, a basic understanding of DevOps concepts is necessary to understand the role of information security teams.

The “three ways” characterize the DevOps movement. The first way involves understanding and optimizing the flow of work within an organization. The second way emphasizes the importance of feedback after issues occur. The third way requires that the organization continually improves and that resilient systems are developed through experimentation (Kim et al., 2016). As a security team, it is critical to support these three goals at all times while ensuring that all teams are developing secure work products (Kim, 2017).

DevOps organizations continuously seek to minimize “lead times.” A lead time is the amount of time between when a request creation and when the customer has access to the finished product. Short lead times are especially critical in software development where new features and systems quickly become obsolete. Excessive lead time is also indicative of unnecessary work or inefficient processes. Security teams can contribute to this goal by automating testing activities to the greatest degree possible. Additionally,
using security requirements to block production deployments should be used only for extremely serious risks (Kim et al., 2016).

Both traditional software security and DevOps agree that organizations should strive to discover vulnerabilities as early as possible. Prompt detection of issues drastically reduces the cost of fixing defects. As IBM reported, it costs roughly 100 times more to fix software issues in production than a defect caught at design time (Dawson, Burrell, Rahim, & Brewster, 2010). While both methods have the same goal, they each approach the problem differently. DevOps as a methodology organizes work into small groups (Kim et al., 2016), known as “batch sizes,” that are deployed to production as quickly as possible. This speed allows not only security issues to be discovered soon after coding is complete, but also whether features are useful to end users. The prompt delivery of new features is in contrast to projects in a waterfall environment where a complete system deployment must be meticulously planned, designed, developed, and executed as a single unit (Conklin & Shoemaker, 2014). Projects utilizing the waterfall methodology must rely on checks at various project gates to detect issues.

The third way of DevOps emphasizes continuous improvement of daily work. DevOps suggests that time is allocated to specifically work on optimizing routine tasks to make them more effective and more efficient (Kim et al., 2016). Workers are encouraged to take calculated risks to improve processes and system resiliency. While attempting to optimize processes is common to many organizations, the drive for improving resiliency is foreign to many IT organizations. Netflix went so far as to develop a tool called Chaos Monkey to randomly terminate systems in their Amazon Web Service (AWS) instance. Google maintains a team to simulate both small and extreme disasters, such as the loss of entire data centers or sections of the United States (Bort, 2016).

1.2. Security and DevOps

Software security teams have built processes around a series of tools and manual processes designed to be run by experts. These manual efforts often put security teams in direct opposition to the DevOps culture. The main point of contention is DevOps drive to minimize lead times by reducing work in progress (WIP). Software security processes
tend to rely heavily on vulnerability scanners, static and dynamic tools, and manual testing. This prioritization of tools has been effective as security teams can conduct these activities in a siloed environment with little interaction with development, quality assurance (QA), and operations teams. When tests are necessary, the internal customers in development, QA and operations can simply request a test from a security team, ticketing system, or third-party consultants. This requisition process is sufficient for older development methodologies with infrequent releases. Allocating two weeks for security testing and additional remediation time may be a reasonable choice if software is deployed only once per year. In DevOps where developers push code to production ten times in a day, it is unrealistic to expect such a large time window for testing. Even expecting that developers will coordinate each release with the security team is not feasible.

Software security teams should look to operations and QA teams for guidance on handling the blistering pace of DevOps software deployment. These teams have been effectively operating without becoming the system bottleneck since DevOps’ inception. Manual processes such as system provisioning, security scanning, and quality assurance testing quickly become roadblocks if these activities are not fully automated and can be initiated by developers. QA and operations teams use scripts and deployment tools to ensure that developers can safely push software to production. Some security tools now contain application programming interfaces (APIs) so that DevOps deployment tools can initiate scans. APIs are now common in tools like OWASP ZAP, Burp, and many static analysis tools (Vries, 2015). Even though the tools may initiate automatically, they may still take an unacceptable amount of time to scan the entire piece of software thoroughly. In these instances, scans can use fewer signatures or occur out-of-band with the deployment pipeline, so they do not block the push to production (Bird, 2016). If security testing does find vulnerabilities, then these can be addressed in upcoming releases. With multiple deployments per day, developers can potentially push security fixes to production within hours of vulnerability discovery.

The goals of any application security program should be to reduce the total number of vulnerabilities and to identify vulnerabilities as early in the SDLC as possible.

Rebecca Deck, sdsecurityacct@hotmail.com
Many SAST tools integrate with software engineer’s development environments. This integration can provide instant feedback to the developer, allow for quick remediation before developers commit code, and reduce the chances that a build will fail for a security issue. Early and accurate identification continues to be the goal of all software security measures. This goal limits the benefit of several detective measures, such as penetration testing and DAST, since they are time-consuming and require a fully functional system (Bird, Johnson, & Kim, 2015). Other security activities that can run during planning, initial coding, and in-band on build processes then become more important.

The lack of security personnel relative to software developers also presents a challenge when trying to ensure software security. It is reasonable for a development shop to have a 100 to 1 ratio of software developers to application security personnel (Kim et al., 2016). To offset this disparity in staffing, continuous integration tools should run automated security tests and tools (Bird, 2016). Even with the continuous integration of automated tools, such as SAST and DAST testing, software development teams have to increase their share of the security burden (Kim et al., 2016). With proper training, developers can perform security checks during code reviews, assist with threat models, and write automated abuse case tests. This process eases the burden on security personnel and allows them to focus on oversight and more specialized testing.

The Microsoft approach to Agile development advocates the creation of “security champions.” While all developers should train in secure development practices, a “security champion” receives additional training (Sullivan, 2009). This might include regular updates about security-related topics, enhanced abuse case development training, and additional training about security testing.

Traditional vulnerability management can be greatly simplified by hardening system images and automating their deployment. To minimize production deployment issues, DevOps uses the same standard system images for all environments from development through production. This standardization presents an opportunity for security teams to ensure that the operating system and any services are patched and hardened.

Rebecca Deck, sdsecurityacct@hotmail.com
2. Software Security Testing in DevOps

2.1. DevOps Testing Overview

In DevOps, unit and integration testing software tests new software features. Since abuse cases are handled identically to use cases, the same tools can test abuse cases as well. Unit testing frameworks can quickly execute large numbers of tests against application functions (McGraw, 2014). After code has passed the associated unit tests, developers can check it into the source control environment (Kim et al., 2016). Continuous integration / continuous deployment (CI/CD) environments can then build the software and run integration tests. These tests ensure that the functioning system responds expectedly. The remainder of this paper explores how unit and integration tests can be used to ensure that applications avoid vulnerabilities that normal static and dynamic testing tools have difficulty detecting. This paper covers neither specific methods for system hardening, nor the use of static or dynamic tools.

2.2. Abuse Cases

According to the 2017 AppSec survey from SANS, the most popular security controls are not necessarily the most compatible with a DevOps environment, nor are they the most effective. Of the top 11 most common security controls used in organizations, four are completely manual processes. Three of the remaining controls can only operate on a completed application. Only SAST, container security scanning, and third-party component analysis can run in an automated fashion without completed applications. With these two tools, only SAST detects security issues in custom user code (Bird, 2017).

The focus on the controls noted in the 2017 AppSec survey misses another effective security control – abuse cases (Bird, 2017). During normal software development, use cases are created to tell developers how the software should behave (McGraw, 2006). Abuse cases describe how the software should behave under adverse conditions, such as when a user sends invalid input to the application or attempts to access another user’s private data. By including abuse cases in the software development...
process, security is built into the software, and normal testing can detect the presence of vulnerabilities (Shackleford, 2017).

One of the barriers to including abuse cases as part of the software development lifecycle (SDLC) is creating the abuse cases. The common criteria, misuse cases, attack trees, and threat modeling are all methods to generate ideas for abuse cases (Ansari & Pandey, 2017). Of these methods, threat modeling is the only one to see adoption and development in recent years. Tools, such as the Microsoft Threat Modeling Tool and OWASP Threat Dragon, allow teams to generate large numbers of abuse cases. Threat modeling tools work by allowing a user to draw architecture diagrams of all components and communication paths involved in a system. The threat modeling tool then applies a template that analyzes the diagram and generates applicable threats. Teams can then prioritize these threats using the DREAD method (Shostack, 2014). These threats are then translated directly into abuse cases for the software and added to the project backlog. Developers can now address the potential vulnerabilities in the same way as normal bugs.

2.3. Creation of Abuse Cases

As developers add software use cases to a project, the developers must also create corresponding abuse cases. While security teams can aid in this process, it is unlikely that a security team will have sufficient resources to create all the abuse cases that an application needs. Instead, it is critical that security teams focus their efforts on nurturing development teams’ security skills. The Microsoft method of establishing “Security Champions. Security team members can then check raw numbers of abuse cases that developers have created to ensure that the process is functioning as intended.

Security teams can assist in abuse case construction by producing what the Microsoft Threat Modeling Tool calls “templates.” After a project team creates a system diagram in the modeling tool, the template can automatically generate a list of applicable threats for the application. If the security team has added sufficient relevant threats into the template, then this is an effective means of communicating abuse cases to the development team without manual intervention by security team members. Tools such as the Microsoft Threat Modeling Tool, Mozilla SeaSponge, or OWASP’s Threat Dragon
tool can facilitate this task. However, at the time of writing, only the Microsoft Threat Modeling Tool supports threat templates. Both SeaSponge and Threat Dragon have threat templates on their development roadmap. Therefore these tools may become viable enterprise threat modeling tools in the future.

When the development team creates a use case, this threat model can show whether or not corresponding abuse cases are warranted. Figure 1 shows an architecture diagram for the OWASP Juice Shop intentionally vulnerable web application drawn using the Microsoft Threat Modeling tool.

![Threat model design for the OWASP Juice Shop application.](image)

A full discussion of the vulnerabilities in the Juice Shop application is beyond the scope of this paper. However, this paper will use a subset of the generated vulnerabilities to illustrate the techniques of developing and testing abuse cases. The architecture in Figure 1 generates several threats, shown in Figure 2.
Each threat may generate several abuse cases. Some of the generic threats, such as “Spoofing source user in web application” and “SQL Injection,” may still require refinement to apply in the context of the application. More specific threats, such as “access another user’s shopping cart” or “SQL Injection on login page” can be useful intermediate steps to generating a full abuse case.

With the “Uncontrolled Redirection” threat as an example, several use and abuse cases are necessary to exercise the functionality. An example of a use case to test the redirection function may be “Redirection is permitted to a whitelisted site.” A use case would then exist for each site that is on the whitelist. In contrast, some possible abuse cases might be “Redirection to a non-whitelisted site is forbidden,” “Redirection outside the whitelist with whitelisted URL as a parameter,” or “Redirection outside the whitelist with whitelisted URL separated by a null byte.” This process allows our mostly automated threat model to guide the development of abuse cases.

2.4. Abuse Case Unit Testing

Unit tests are those that run on individual software functions as opposed to fully functional and assembled applications. This factor carries both advantages and disadvantages over integration testing, which tests more fully formed applications. First, unit testing is far faster than integration testing. Thousands of unit tests can run in mere seconds without creating additional systems or generating network traffic. Second, unit tests are simple to write. Each test will only check a specific condition of a single function to determine if the software functions correctly or not. Likewise, abuse cases unit tests will ensure that software operates in a secure manner given specific malicious input to a function.

Rebecca Deck, sdsecurityacct@hotmail.com
Given the threat of “Uncontrolled Redirection,” several unit tests are necessary to ensure that the software functions correctly given both safe and malicious input. The Juice Shop application defines a whitelist of acceptable sites for redirection in the insecurity.js file as shown in Figure 3.

```javascript
const redirectWhitelist = [
    'https://github.com/bkimminich/juice-shop',
    'https://blockchain.info/address/1AbKfgvw9psQ41Nbl8kufDQTezwG8DRZm',
    'https://gratipay.com/juice-shop',
    'http://flattr.com/thing/3856930/bkimminichjuice-shop-on-GitHub',
    'http://shop.spreadshirt.com/juiceshop',
    'http://shop.spreadshirt.de/juiceshop',
    'https://www.stickermule.com/user/1070702817/stickers',
    'https://explorer.dash.org/address/Xr556RzuwX6hg5EGpkybbv5RanJoZN17kW'
]
exports.redirectWhitelist = redirectWhitelist
exports.isRedirectAllowed = url => {
    let allowed = false
    redirectWhitelist.forEach(allowedUrl => {
        allowed = allowed || url.indexOf(allowedUrl) > -1
    })
    return allowed
}
```

![Figure 3. Redirection whitelist in insecurity.js](image)

This whitelist contains several entries, and each one will have an associated use case to ensure that the isRedirectAllowed function properly redirects to the whitelisted site. Using Mocha to write an automated test may result in code similar to the snippet shown in Figure 4.

```javascript
describe("Redirection Use Case", function() {
    it("Redirection inside whitelist is permitted", function(done) {
        expect(insecurity.isRedirectAllowed('https://github.com/bkimminich/juice-shop')).to.be.true
        done()
    });
});
```

![Figure 4. Unit test to ensure redirection to whitelisted sites is allowed.](image)

Testing the redirection abuse cases outlined earlier in this paper requires three abuse cases. Figure 5 shows these three abuse case unit tests.

Rebecca Deck, sdsecurityacct@hotmail.com
describe("Redirection Abuse Case", function() {
  it("Redirection outside whitelist is forbidden", function(done) {
    expect(insecurity.isRedirectAllowed('https://www.evilsite.local')).to.be.false
done()
  });
  it("Redirection outside whitelist with whitelisted URL as a parameter is forbidden" , function(done) {
    expect(insecurity.isRedirectAllowed('https://evilsite.local/?https://github.com/bkimminich/juice-shop')).to.be.false
done()
  });
  it("Redirection outside whitelist with whitelisted URL separated by null byte is forbidden", function(done) {
    expect(insecurity.isRedirectAllowed('https://evilsite.local\0https://github.com/bkimminich/juice-shop')).to.be.false
done()
  });
});

**Figure 5. Security unit tests to test redirection outside a whitelist.**

An additional concern listed in Figure 2 is a weak JSON Web Token (JWT) secret. According to OWASP, JWT supports the use of a “secret” to sign the token. This signature ensures that an attacker cannot tamper with the JWT and have the server accept the modified token. Attackers can perform an offline brute force against this token to attempt to recover the secret (Manico, Righetto, & Ionescu, 2017). As a protection against this brute force, secrets must be treated like passwords – they must be long, complex, and rotated periodically. A possible abuse case might be to “Brute force weak JWT Secrets.” However, this abuse case may run for an inordinate amount of time given the number of possible weak JWT secrets. An alternate approach may be “Resist JWT Brute Force - Secret Strength > 100” and use a NodeJS library to test password strength. This approach can quickly verify length and complexity in a short period. Figure 6 shows using the passwd-strength NodeJS module to ensure the JWT secret strength is at least 100.

Rebecca Deck, sdsecurityacct@hotmail.com
Adapting AppSec to a DevOps World

Rebecca Deck, sdsecurityacct@hotmail.com

Figure 6. Testing JWT secret strength using the passwd-strength library.

Using Mocha, the thirteen use and abuse cases listed can execute in 35 milliseconds, as Figure 7 depicts. It is possible to run many hundreds or even thousands of such abuse cases in only a few seconds. This amount of time is short enough to run in a CI/CD environment to ensure basic security tests pass before promoting software to a production environment. Additionally, the abuse case tests are simple enough to write that developers can test software as soon as they write it. Prompt testing allows developers to implement bug fixes while they still remember exactly where they made changes in the code; minimizing the time necessary to correct issues (Kim et al., 2016).
The unit tests reveal a total of ten passing tests and three failures. First, the test “Resist JWT Brute Force – Secret Strength > 100” has failed with the message “expected 0 to be at least 100.” This message indicates that the test observed the value of zero for the password strength and a passing score would be greater than 100. The test to ensure the redirection function rejects redirection outside the whitelist passes. However, two abuse cases that include redirection outside the whitelist, but include whitelisted text in the redirection URL, fail. Both tests assert that the test “expected true to be false.” The observed value in both cases is true – the function permits redirection – and the function should have returned false.
2.5. Abuse Case Integration Testing

Integration testing differs from unit testing in that it tests multiple components working together instead of individual units of code (Bird, 2016). Integration tests will be slower and take place further in the SDLC than unit tests, but more accurately test the running application’s attack surface. Multiple developers, or at least developers working over a relatively long period, often work on production-ready software. Coordination and time cause developers to make assumptions about how other components work, perhaps assuming that sanitization, encoding, or error trapping may occur elsewhere in the application. Unit testing cannot uncover such issues, and so integration tests are designed to uncover vulnerabilities arising from these assumptions.

The Mocha and Chai modules support integration testing, not just unit testing. With the addition of the SuperAgent module, it is possible to run fully authenticated integration tests against a live web application. Figure 8 demonstrates two abuse cases – authenticated access to another user’s basket and unauthenticated access to a user’s basket.
Authentication Code

```javascript
var expect = require("chai").expect;
var request = require('superagent');
const agent = request.agent();

var username = "admin@juice-sh.op";
var userpass = "admin123";
var authToken = ''; 
var url = "http://127.0.0.1:3000";

describe("Authenticated Basket Access Abuse Cases", function() {
  beforeEach(function(done) {
    agent
      .post(url + '/rest/user/login')
      .set('Content-Type', 'application/json;charset=utf-8')
      .send({email: username, password: userpass})
      .end(function(error, response){
        expect(response.status).to.equal(200);
        authToken = response.body['authentication']['token']
        done();
      });
  });

  it("Users cannot access another user's basket", function(done) {
    agent.set('Authorization', 'Bearer ' + authToken)
    agent.set('Cookie', 'token=' + authToken)
    agent.get(url+'/rest/basket/2', function(error, response, body) {
      expect(response.status).to.equal(403);
      done();
    });
  });
});
```

Authenticated Test

```javascript

describe("Unauthenticated Basket Access Abuse Cases", function() {
  it("Unauthenticated users cannot access baskets", function(done) {
    request.get(url+'/rest/basket/1', function(error, response, body) {
      expect(response.status).to.equal(401);
      done();
    });
  });
});
```

Unauthenticated Test

Figure 8. Authenticated and unauthenticated integration tests for basket access.

The “beforeEach” block of code performs requests to the target web server to establish a session as the admin@juice-sh.op user. The Juice Shop application returns a token in its response to a login request. The application then places this token in both the authorization HTTP header and the token cookie. The authenticated test portion of the Mocha test, shown in Figure 8, sets these values and performs a request for a basket which does not belong to the admin@juice-sh.op user. Finally, the script checks the HTTP response code of the server’s response to ensure that it is a 403 (forbidden) code.

Rebecca Deck, sdsecurityacct@hotmail.com
The unauthenticated portion of the script in Figure 8 is similar to the authenticated request, but it takes no action with the token. Additionally, this test case does not appear in the same “describe” block as the authenticated abuse test. Therefore, the “beforeEach” code segment will not run for the unauthenticated tests, which saves time. The only other difference in this test case is that it expects a 401 (authentication failed or not provided) server response code instead of a 403 since this request does not contain an authentication token.

These two abuse case tests produce the output in Figure 9. The unauthenticated basket abuse case passes, indicating that authentication is required to access the basket page. The authenticated basket abuse case fails with the assertion that the test “expected 200 to equal 403.” This text indicates that the test expects the cross-account basket access to return a 403 response code, but observed a 200 (OK) response code.

```
Authenticated Basket Access Abuse Cases
  1) Users cannot access another user's basket

Unauthenticated Basket Access Abuse Cases
  Unauthenticated users cannot access baskets

1 passing (93ms)
1 failing

1) Authenticated Basket Access Abuse Cases
   Users cannot access another user's basket:

   Uncaught AssertionError: expected 200 to equal 403
   + expected - actual

   -200
   +403

   at node_modules/abuseBasketSpec.js:33:36
   at Request.callback (node_modules/superagent/lib/node/index.js:706:12)
   at parser (node_modules/superagent/lib/node/index.js:906:18)
   at IncomingMessage.responseText (node_modules/superagent/lib/node/parsers/json.js:19:7)
   at endReadableNT (_stream_readable.js:1062:12)
   at process._tickCallback (internal/process/next_tick.js:152:19)
```

Figure 9. Shopping basket abuse case test results.
3. Abuse Case Testing Phases in DevOps

3.1. Phase 1 – Threat Modeling

The most efficient method for developing abuse cases is threat modeling (Yuan, Nuakoh, Williams, & Yu, 2015). This approach has several benefits over other abuse case creation methods. Security teams can pre-create templates to guide less experienced developers as they generate potential threats. Threat modeling has the added benefit of ensuring that architecture and data flow diagrams are created for the applications. The project team can meet this checkpoint before writing any application code. This early involvement helps ensure the project team considers security throughout the SDLC.

3.2. Phase 2 – Abuse Case Development

It is best to develop specific abuse cases along with the application’s use cases. The threat model template, when properly refined, will provide generic guidance on threats. An example use case, such as “Users should be able to view and modify their profile page,” can be combined with a threat of “users may attempt to access another user’s records.” This combination might yield abuse cases of “An unauthenticated user should not be able to access the profile page” and “an authenticated user should not be able to access another user’s profile page.” The abuse cases can then be included in the product backlog to ensure that developers code the application appropriately to avoid vulnerabilities.

3.3. Phase 3 – Write Code and Tests

During the software development process, developers should write automated test suites to confirm that the application performs correctly for both use and abuse cases. Depending upon the development process, developers may write the test suite before the application code. Developers can test the code before checking it into the source repository.

Rebecca Deck, sdsecurityacct@hotmail.com
3.4. Phase 4 – Execute Unit and Integration Tests

Before checking code into the source repository, developers should ensure that the application passes all unit tests and any integration tests that can run in the developer’s local environment. No code should be checked in until all tests execute successfully. When building software, all tests will re-run from within the CI/CD build environment to ensure developers did not neglect to run tests and that integrating multiple pieces of code does not cause vulnerabilities.

3.5. Phase 5 – Monitoring

The monitoring phase of abuse cases is chiefly concerned with ensuring developers are creating appropriate abuse cases. When appropriately configured, CI/CD build environments will monitor whether or not the application passes the tests – if the tests do not pass, the build will fail. Security teams can better spend their time ensuring that developers create an appropriate volume and type of abuse cases. Tracking this information may require labeling abuse cases with metadata, such as STRIDE classification, to validate that developers did not overlook any threat categories. Code coverage testing tools also allow security teams to verify how much application code security tests actually check. If security unit and integration tests for abuse cases are stored separately from non-security tests, then modules like Istanbul can determine which segments of code abuse case tests do not cover.

3.6. Phase 6 – Continuous Improvement

Continuous improvement mandates an effort to provide abuse cases that detect new classes of vulnerabilities or vulnerabilities that manual methods detected. An example may be if a penetration testing team discovers that account passwords are set to a predictable value if an account uses OAuth. The development staff should create a corresponding abuse case and automated test, but security teams must also address the cause of the missed abuse case. This treatment may include updating threat models to ensure they document OAuth connections correctly and updating threat templates to specify predictable user passwords as a threat to consider. With these updates, project

Rebecca Deck, sdsecurityacct@hotmail.com
teams should be able to detect the same issue in other existing applications and future applications.

4. Summary

DevOps heavily emphasizes automation during the development process. Current application security processes are extremely limited where automation is involved – often only providing a minimal API. Even with the ability to automatically begin a scan from a CI/CD tool, these tools may take an unacceptable amount of time to execute. Fully embracing DevOps requires more than jamming existing tools into a CI/CD pipeline and declaring that a security team has completed their job. DevOps teams already have tools and processes in place to develop and test software that functions correctly. Abuse cases are a simple solution to use these existing processes to improve security. Using threat models with security teams managing threat templates and developers writing abuse cases and automated test scripts is an effective way to achieve security integration. Processes that focus on forcing security tools to work in a DevOps environment are more likely to require a large security team to run tools and interpret results. Abuse cases require no such investment in an army of security specialists. With proper execution, abuse cases can detect vulnerabilities that standard security tools cannot while using only the tools with which developers are already familiar.

Rebecca Deck, sdsecurityacct@hotmail.com
Adapting AppSec to a DevOps World

References


Rebecca Deck, sdsecurityacct@hotmail.com


Rebecca Deck, sdsecurityacct@hotmail.com
## Upcoming SANS App Sec Training

<table>
<thead>
<tr>
<th>Event</th>
<th>Location</th>
<th>Dates</th>
<th>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>SANS 2020</td>
<td>Orlando, FL</td>
<td>Apr 03, 2020 - Apr 10, 2020</td>
<td>CyberCon</td>
</tr>
<tr>
<td>SANS Copenhagen August 2020</td>
<td>Copenhagen, Denmark</td>
<td>Aug 24, 2020 - Aug 29, 2020</td>
<td>Live Event</td>
</tr>
<tr>
<td>SANS Network Security 2020</td>
<td>Las Vegas, NV</td>
<td>Sep 20, 2020 - Sep 27, 2020</td>
<td>Live Event</td>
</tr>
<tr>
<td>SANS OnDemand</td>
<td>Online</td>
<td>Anytime</td>
<td>Self Paced</td>
</tr>
<tr>
<td>SANS SelfStudy</td>
<td>Books &amp; MP3s Only</td>
<td>Anytime</td>
<td>Self Paced</td>
</tr>
</tbody>
</table>