DevSecOps
Let's Write Security Unit Tests

February 20th, 2022

gcloud auth print-identity-token

PRINCIPAL SECURITY ENGINEER
PUMA SECURITY

Security assessments:
Cloud, DevSecOps, source code, web apps, mobile apps

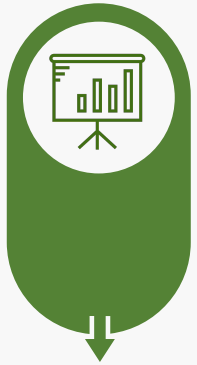
Coder:
Cloud automation, static analysis engine, security tools

SENIOR INSTRUCTOR
SANS INSTITUTE

Author and instructor

SEC540: Cloud Security & DevSecOps Automation

SEC510: Public Cloud Security: AWS, Azure, and GCP



EDUCATION AND TRAINING

Iowa State M.S. Information Assurance, B.S. Computer Engineering

AWS, CISSP, GSSP, GWAPT

CONTACT

Email:
ejohnson@pumasecurity.io

Twitter: @emjohn20

LinkedIn:
www.linkedin.com/in/eric-m-johnson

AGENDA

DevSecOps
Let's Write Security Unit Tests

1. DevSecOps Security Controls
2. Unit Testing 101
3. Security Unit Testing
4. Evil User Stories
5. Continuous Integration Testing

1 CLOUD & DEVSECOPS SECURITY CONTROLS

PUMA SECURITY

CLOUD & DEVOPS SECURITY CONTROLS

PRE-COMMIT

- Threat Modeling
- IDE Security Plugins
- Pre-Commit Hooks
- Peer Code Reviews

COMMIT (CI)

- Static Code Analysis
- Security Unit Tests
- Container Security
- Dependency Management

ACCEPTANCE

- Infrastructure as Code
- Cloud Infrastructure
- Dynamic Security Tests
- Security Acceptance Tests

PRODUCTION

- Security Smoke Tests
- Secrets Management
- Security Configuration
- Server Hardening

OPERATIONS

- Blameless Postmortems
- Continuous Monitoring
- Penetration Testing
- Threat Intelligence

PUMA SECURITY

2 PRE-COMMIT: UNIT TESTING 101

UNIT TESTING

Mature systems typically contain thousands of unit tests written by software engineers:

- Easy to write, easy to change
- Fast execution in code editors and CI pipelines
- Foundation of test-driven development (TDD)
- Code coverage measures % of code execution as tests run
- Rarely utilized security control in DevSecOps

UNIT TESTING TOOLS

Unit testing frameworks for various development platforms:

- JUnit (Java)
  - https://junit.org
- XUnit (C#, F#, VB)
  - https://xunit.github.io/
- Mocha (NodeJS)
  - https://mochajs.org/
- RSpec (Ruby)
  - http://rspec.info/
- PyUnit (Python)
  - https://wiki.python.org/moin/PyUnit

UNIT TEST EXAMPLE | XUNIT

xUnit example testing the happy path login:

```
[Fact]
public async Task Authenticate_Succeed_GivenPasswordIsValid()
{
  // Act
  var response = await  Client.SendAuthenticationPostRequestAsync(
    SeedData.Member1Email, SeedData.User1Password);

  // Assert
  response.StatusCode.Should().Be(HttpStatusCode.OK);

  var authenticationResponse =
    await response.Content.ReadFromJsonAsync<AuthenticateResponse>();
  authenticationResponse.JwtToken.Should().NotBeNull();
}
```

UNIT TEST EXAMPLE | XUNIT EXECUTION

Running xUnit test cases against .NET Core:

```
$ cd ./test/Coyote.Tests
$ dotnet test
```

Example xUnit output results:

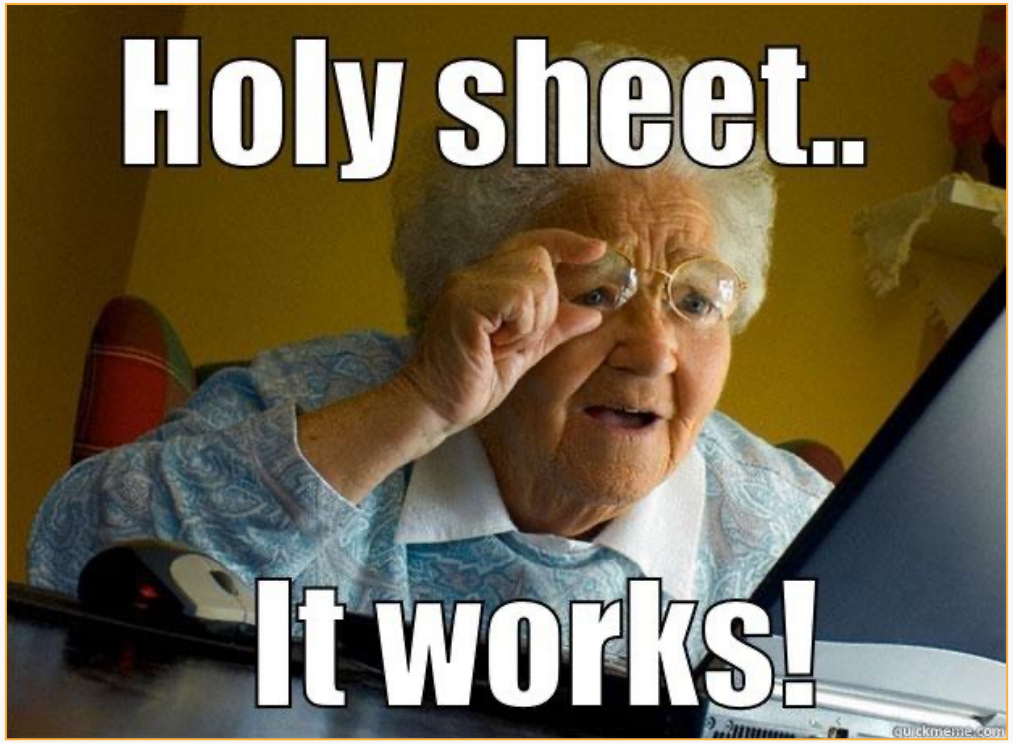
```
Microsoft (R) Test Execution Command Line Tool Version 17.0.0
Copyright (c) Microsoft Corporation. All rights reserved.

Starting test execution, please wait...
A total of 1 test files matched the specified pattern.
Passed! - Failed: 0, Passed: 11, Skipped: 0, Total: 11, Duration: 466 ms
```

3 PRE-COMMIT: SECURITY UNIT TESTING

SECURITY UNIT TESTING ROADBLOCKS

Security unit testing is often missing from DevSecOps pipelines because it requires collaboration between Development and Security:

- Software engineers often stay on the "happy path"
- InfoSec teams have the "attacker mindset"
- Working together can provide a valuable custom security control for applications



SECURITY UNIT TESTING RESOURCES

OWASP Application Security Verification Standards

- https://owasp.org/www-project-application-security-verification-standard/

OWASP User Security Stories

- https://github.com/OWASP/user-security-stories

Puma Security: Puma Prey Tests

- https://github.com/pumasecurity/puma-prey

SPUTR

- https://github.com/sethlaw/sputr
- https://www.blackhat.com/asia-17/briefings.html#domo-arigato-mr.-roboto-security-robots-a-la-unit-testing

BUILDING A SECURITY UNIT TESTING SUITE

InfoSec Pro Tip - Work with software engineers to build security-focused unit test suites:

- Create abuse cases and evil user stories
- Burn the security stories into code
- Run security test cases to enforce security requirements
- Require negative test cases that should never pass
- Focus on high risk code and business logic flaws first

HIGH RISK CODE CANDIDATES

High risk code responsible for any of the following functionality are logical candidates:

- Authentication
- Password handling
- Access control
- Output encoding
- Input validation
- Security Unit Tests
- Data entitlement checks
- User management
- Handling confidential data
- Cryptography
- Infrastructure code
- Declarative pipeline definitions

4 EVIL USER STORIES

ASVS 2.1| PASSWORD SECURITY

As a user, I shall not be able to create an account with a weak password.

ASVS 2.1.1: Verify that user set passwords are at least 12 characters in length (after multiple spaces are combined).

ASVS 2.1.7: Verify that passwords submitted during account registration, login, and password change are checked against a set of breached passwords

ASVS 2.1.1 & 2.1.7 | SECURITY UNIT TEST

Creating a new user account with a weak, known breached password:

```
[Theory]
[InlineData(7, "Tinkerbell", HttpStatusCode.BadRequest)]
public async Task ASVS_2_1_7(int index, string password, HttpStatusCode
                                                          statusCode)
{
  // Act
  …
  CreateUserRequest.Password = password;
  var response = await Client.CreateUser(CreateUserRequest);

  // Assert
  response.StatusCode.Should().Be(statusCode);
}
```

ASVS 3.5 | TOKEN SESSION MANAGEMENT

As an attacker, I shall not be able to tamper with authentication tokens.

ASVS 3.5.2: Verify the application uses session tokens rather than static API secrets and keys, except with legacy implementations.

ASVS 3.5.3: Verify that stateless session tokens use digital signatures, encryption, and other countermeasures to protect against tampering, enveloping, replay, null cipher, and key substitution attacks.

ASVS 2.5.3 | SECURTY UNIT TEST

Request user profile data without a JWT signature:

```
[Theory]
[InlineData(SeedData.Member2Email, SeedData.User2Password,
            HttpStatusCode.Unauthorized)]
public async Task ASVS_3_5_3(string username, string password, HttpStatusCode
                                                                statusCode) {
  // Act
  var token = await Client.GetJwtAsync(username, password);

  //Strip signature
  var tamperedToken = $"{token.Split(".")[0]}.{token.Split(".")[1]}.";
  Client.DefaultRequestHeaders.Add("Authorization", $"Bearer {tamperedToken}");
  var response = await Client.GetUserByMemberId(Convert.ToInt32(memberId));

  // Assert
  response.StatusCode.Should().Be(statusCode);
}
```

ASVS 4.2 | OPERATION LEVEL ACCESS CONTROL

As an attacker, I shall not be able to access a profile that belongs to another user.

ASVS 4.2.1: Verify that sensitive data and APIs are protected against Insecure Direct Object Reference (IDOR) attacks targeting creation, reading, updating and deletion of records, such as creating or updating someone else's record, viewing everyone's records, or deleting all records.

ASVS 4.2.1 | SECURITY UNIT TEST

Member level user reading the admin user's profile:

```
[Theory]
[InlineData(SeedData.Member2Email, SeedData.User2Password, 2, HttpStatusCode.OK)]
[InlineData(SeedData.Member2Email, SeedData.User2Password, 1,
                                                HttpStatusCode.Unauthorized)]
public async Task ASVS_4_2_1(string username, string password, int memberId,
                    HttpStatusCode statusCode)
{
  // Act
  var token = await Client.GetJwtAsync(username, password);
  Client.DefaultRequestHeaders.Add("Authorization", $"Bearer {token}");
  var response = await Client.GetUserByMemberId(memberId);

  // Assert
  response.StatusCode.Should().Be(statusCode);
}
```

PUMA SECURITY

ASVS 5.2 | SANITIZATION & SANDBOXING

As an attacker, I shall not be able to inject malicious data into the application.

ASVS 5.2.2: Verify that unstructured data is sanitized to enforce safety measures such as allowed characters and length.

ASVS 5.2.6: Verify that the application protects against SSRF attacks, by validating or sanitizing untrusted data or HTTP file metadata, such as filenames and URL input fields, and uses allow lists of protocols, domains, paths and ports.

ASVS 5.2 | SECURITY UNIT TEST

Storing malicious SSRF input data in the database name field:

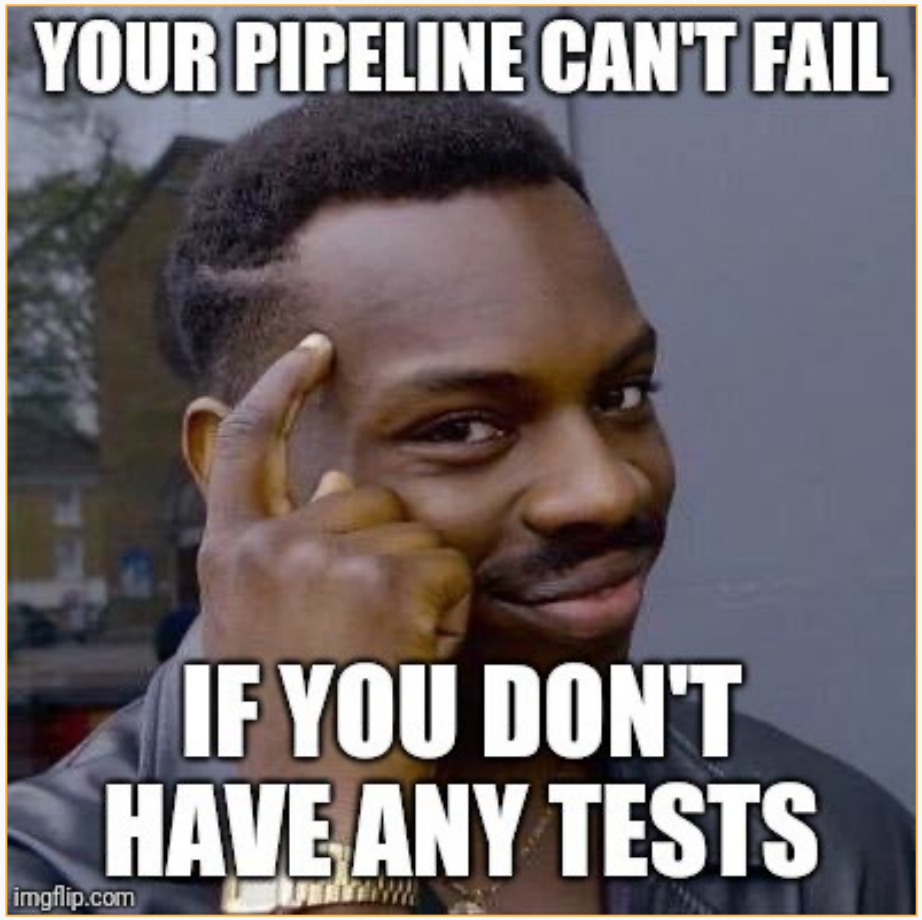
```
[InlineData(SeedData.Member1Email, SeedData.User1Password,
            "http://169.254.169.254", HttpStatusCode.BadRequest)]
[InlineData(SeedData.Member1Email, SeedData.User1Password,
            "http://metadata.google.internal", HttpStatusCode.BadRequest)]
public async Task ASVS_5_2(string username, string password, string name,
HttpStatusCode statusCode)
{
  …
  //Set payload and test
  CreateAnimalRequest.Name = name;
  var response = await Client.PostAnimal(CreateAnimalRequest);

  // Assert
  response.StatusCode.Should().Be(statusCode);
}
```

5 CONTINUOUS INTEGRATION TESTING

CONTINUOUS INTEGRATION TESTING

Prerequisite for automating build, test, and deployment:

- Commit triggers an automated pipeline
- Executes security unit tests
- Provides fast feedback to engineers (and security)
- Enforces continuous security compliance



.NET TEST | GENERATING TEST RESULT DATA

Exporting unit tests results to a test results file:

```
$ cd ./test/Coyote.Tests
$ dotnet test -logger "trx;LogFileName=coyote.trx"
```

Example .NET test unit test output results:

```
<ResultSummary outcome="Failed">
<Counters total="10" executed="10" passed="9" failed="1" error="0"
   timeout="0" aborted="0" inconclusive="0" passedButRunAborted="0"
   notRunnable="0" notExecuted="0" disconnected="0" warning="0" completed="0"
   inProgress="0" pending="0" />
</ResultSummary>
```

GH ACTION | UNIT TEST CONFIGURATION

GH Action workflow executing and parsing unit test results:

```
jobs:
  build:
  name: Puma Prey
  runs-on: ubuntu-18.04
  steps:
  …
  - name: Test
    shell: bash
    run: |
      cd ./test/Coyote.Tests
      dotnet test --logger "trx;LogFileName=coyote.trx"
  - name: Publish Test Results
    uses: dorny/test-reporter@v1
    with:
      name: "Coyote Test Results"
      path: "test/Coyote.Tests/TestResults/*.trx"
      reporter: "dotnet-trx"
```

GH ACTION | UNIT TEST RESULTS

Viewing GH Action workflows and unit test results:

All workflows

Showing runs from all workflows

Filter workflow runs

19 workflow runs

Puma Prey
Puma Prey #10: Manually run by ejohn20

Puma Prey
Puma Prey #9: Manually run by ejohn20

Coyote Test Results ❌

tests 15 passed, 13 failed

❌ test/Coyote.Tests/TestResults/coyote.trx

28 tests were completed in **13s** with **15** passed, **13** failed and **0** skipped.

Test suite	Passed	Failed
Coyote.Tests.Feature.AnimalControllerTests	1✓	
Coyote.Tests.Feature.HealthControllerTests	2✓	
Coyote.Tests.Feature.SafariControllerTests	2✓	
Coyote.Tests.Feature.TokenControllerTests	3✓	
Coyote.Tests.Feature.UserControllerTests	3✓	
Coyote.Tests.Security.AnimalControllerTests		9❌
Coyote.Tests.Security.TokenControllerTests	1✓	1❌
Coyote.Tests.Security.UserControllerTests	3✓	3❌

SECURITY UNIT TESTING SUMMARY

Keys To Success:

- Security teams often ignore unit testing and opt for long running, unreliable scanners to find low hanging fruit
- Work with software engineers to create abuse cases and unit & integration tests based on ASVS requirements
- Integrate security unit tests in Continuous Integration pipelines & monitor failing test cases
- Failing security tests should halt the build and require immediate attention before releasing changes

THANK YOU FOR ATTENDING!

Eric Johnson

Principal Security Engineer, Puma Security
Senior Instructor, SANS Institute

www.linkedin.com/in/eric-m-johnson

@emjohn20

ejohnson@pumasecurity.io