Security Impacts of Sub-Optimal DevSecOps Implementations in a Highly Regulated Environment

Jose Andre Morales, Thomas P. Scanlon, Aaron Volkmann, Joseph Yankel, Hasan Yasar

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213

This material is based upon work funded and supported by the Department of Defense under Contract No. FA8702-15-D-0002 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center.

DM20-0555

Introduction

Study Overview

- Large, well-funded, highly-regulated program
- Transitioning from traditional waterfall approach to DevSecOps and Agile
- Already completed several iterations of development using the waterfall approach with a heavy focus on milestones
- Researchers were engaged as program advisors for a two-year period during the transition to DevSecOps and Agile
- Study focused on relationship between a prime and sub contractor
- Researchers onsite with the sub and visits to the prime

DevSecOps Security Concerns

- Need to secure the development pipeline / software factory
- Need to build security into the software produced in the software factory

Examples of Security Threats

- Unauthorized access to data and source code
- Unintended privilege escalation of authorized users
- Data modification
- False alerts and updates
- Suppression of valid alerts
- Malicious dependency insertion
- Hijacking of tools such as build servers

Available Software Security Tools

- Static Application Security Testing (SAST)
- Origin Analysis / Software Composition Analysis (SCA)
- Dynamic Application Security Testing (DAST)
- Interactive Application Security Testing (IAST)
- Test Coverage Analyzers
- Correlation Tools
- Database Security Scanning

5 Key Findings

Incomplete Development Environments

Observation

Inability of the prime to provide needed development environments significantly impacted work performance

- **Test environments were not ready at the times they were needed**
- **Test environments lacked resources to complete necessary tests**
- **Frequent outages occurred in the test environments**
- **Automated testing tools were not configured properly, rendering them largely ineffective**

Incomplete Development Environments

Security Impact

Developers had limited testing time and little access to (functioning) automated testing tools

What Could Have Been Done?

- **Dedicate resources to architect DSO pipeline upfront**
- **Follow known reference architectures**
- **Establish an MVP for the pipeline before any coding begins**

Lack of Environment Parity

Observation

Environment parity was lacking between development and integration / testing platforms

- **Program had no requirement that development testing & integration testing be performed in equivalent environments**
- **Hot fixes were made to environments and not propagated to other environments nor checked-in to prevent reoccurrence**
- **Permissions were not consistent across environments**

Lack of Environment Parity

Security Impact

Parity differences between application environments fosters security weaknesses and inconsistent test results

What Could Have Been Done?

- **Adhered to Infrastructure-as-Code**
- **Adopted use of containers**
- **Thoroughly tested modifications to development environments**

Testing Did Not Reflect Production

Observation

An iterative testing process during development did not include production-like environments

- **Significant differences in network topology, system resources, and storage devices in test vs. prod**
- **User groups and permissions in test regions did not accurately reflect production**
- **End-to-end testing of system components was rarely done**

Testing Did Not Reflect Production

Security Impact

Lack of production-like testing will lead to hot (manual) fixes in production and missed security test cases

What Could Have Been Done?

- **Implement use of containers**
- **Adopt some model based systems engineering (MBSE) principles**
- **Dedicate resources to proper test tooling and construction or realistic test cases**

Contract Objectives > Development Goals

Observation

The sub focused too much on exit criteria to qualify for completion of development and testing, thereby increasing technical debt

- **Small, achievable tasks were favored**
- **Large, complex tasks were ignored or delayed**
- **Testing was done to pass established thresholds, not necessarily to ensure functionality and security**

Contract Objectives > Development Goals

Security Impact

Security tests were ignored as they weren't relevant to functional thresholds; some failed functional tests were ignored as long as threshold met

What Could Have Been Done?

- **Acceptance criteria should have been more rigidly defined**
- **Definition of Done and Definition of Ready should have been established**
- **Consider use of independent test team**

Security Not A Priority

Observation

Secure coding and system security were not prioritized

- **Secure coding and secure development practices were not a requirement**
- **Acceptance criteria solely focused on functionality, not security**
- **Security testing tools were not utilized**

Security Not A Priority

Security Impact

To truly make security a fundamental, integral aspect of software development, it needs to be engineered in

What Could Have Been Done?

- **Follow secure coding guidelines**
- **Utilize automated testing tools**
- **Harden all devices & have a defined patching process**
- **Scan for known vulnerabilities**
- **…**

DEVELOP & FOSTER A SECURITY MINDSET!!

Summary

Summary

- The transition to DevSecOps and Agile left many security gaps
- Tools and technology were thrown at problems with little attention to people and processes
- Short-term achievements were preferred to sound engineering discipline
- Security was not a priority
- Security concerns were not built into contracts, work planning, or software factory design