PARASOFT®

WHITEPAPER

How to Approach **DISA-ASD-STIG Compliance** for Software Development

Defense Information Systems Agency (DISA), Application Security and Development (ASD), and Security Technical Implementation Guides (STIG) is a set of guidelines for securing desktop and enterprise applications used by the Department of Defense. The guidelines cover in-house application development and the evaluation of third-party applications. They don't, however, cover commercial off-the-shelf software.

Achieving compliance to the DISA-ASD-STIG guidelines requires proof, usually in the form of documentation, that satisfies auditors. This paper discusses Parasoft's recommended approach to achieving compliance in an efficient, less risky, and cost-effective manner. To achieve this, a three-level approach is required:

Application scanning with static analysis tools to ensure vulnerabilities are detected and remediated in the application. The DISA-ASD-STIG has specific guidelines on what classes of vulnerabilities to detect and remediate.

Application testing for security with functional and penetration testing tools to verify and validate DISA-ASD-STIG requirements.

Shift-left compliance with preventative processes, which eliminates poor coding practices that lead to vulnerabilities. This wider swath of detection includes application scanning and the application of industry coding standards such as SEI CERT C/C++. It also includes guidelines like the removal of "code smells", which are poor practices known to be the root cause of software vulnerabilities.

This 1-2-3 punch is the key to achieving compliance by verification and documentation with the goal of maturing the process beyond detection into prevention of security vulnerabilities.

WHAT DOES COMPLIANCE WITH DISA-ASD-STIG MEAN?

Compliance to the guidelines is evaluated against product and process documentation as well as observing and verifying functionality:

DISA Category Code Guidelines	
CAT I	Any vulnerability, the exploitation of which will **directly and immediately** result in loss of Confidentiality, Availability, or Integrity.
CAT II	Any vulnerability, the exploitation of which **has a potential** to result in loss of Confidentiality, Availability, or Integrity.
CAT III	Any vulnerability, the existence of which **degrades measures** to protect against loss of Confidentiality, Availability, or Integrity.

Figure 1: DISA-ASD-STIG vulnerability categories

2.1.2 Functionality

When reviewing an application, aspects of application functionality must be evaluated to ensure the appropriate controls exist to protect the application and the application data. Items to consider include the type of data processed by the application such as classified, unclassified, and publicly releasable or Personally Identifiable Information (PII) data. The application's network connections, network access controls, data entry/egress points, application authentication mechanisms, application access controls, and application auditing mechanisms. These items will vary based upon application architecture, design, and data protection requirements.

– ASD STIG Overview, V4R9

In other words, the STIG requires "proof" of secure design and implementation through documentation, verification, and validation of all aspects of the software development lifecycle, including deployment and operation. These guidelines apply throughout the lifetime of the product including configuration, maintenance, and end-of-life.

DISA-ASD-STIG requires the use of application code scanners (Overview, Section 4.1) "…an automated tool that analyzes application source code for security flaws, malicious code, and back doors." In more common terminology this is static application security testing (SAST) implemented

through static code analysis and "should be utilized whenever possible. Particularly in the development environment where code that has been identified as requiring remediation can be addressed prior to release."

The DISA-ASD-STIG also requires the use of active vulnerability testing (like penetration testing tools) to test executable software. These tools are required during development and deployment to support vulnerability assessments.

HOW TO LOOK AT A STIG REQUIREMENT

All STIG requirements are stored as XML and not in human-readable form. A Java-based viewer is supplied in order to view the requirements. The viewer is shown below with each key area of the UI labeled. The requirements are searchable and filtered in the UI, and each requirement lists associated vulnerabilities. The details of each vulnerability contain a description describing each along with how to confirm the vulnerability doesn't exist in the software. There is a useful video describing how to use the tool.

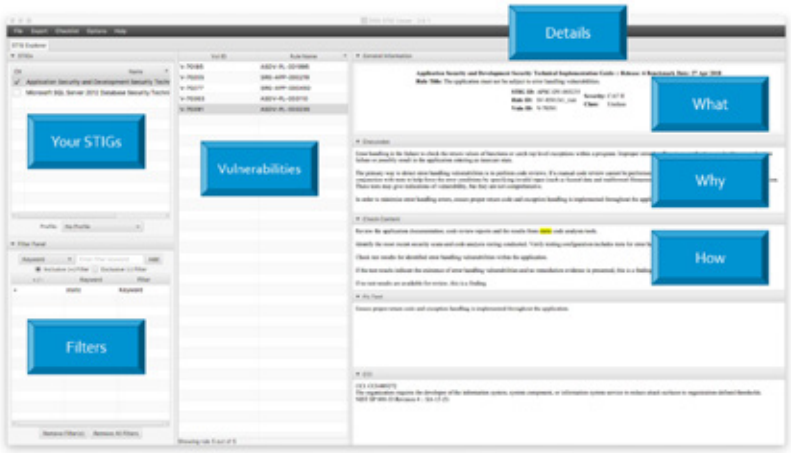


Figure 2: Java-based STIG viewer

The viewer is mentioned here because it's a critical resource needed by software organizations as the go-to source for STIG requirements and hopefully more used and useful than paper documentation.

As stated, vulnerabilities listed in the DISA-ASD-STIG are categorized by severity with most being in Cat II, which means the vulnerability has the potential to cause a loss of integrity, availability, and confidentiality. There are relatively fewer CAT I vulnerabilities, which are the most critical.

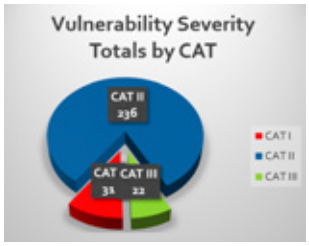


Figure 3: Distribution of vulnerability categories

THE ROLE OF STATIC ANALYSIS TOOLS IN DISA-ASD-STIG

The previous version (v3.x) of the DISA-ASD-STIG required the use of static code analysis along with specific static analysis guidelines to check against. However, this is not the case with the current version.

The latest revision uses the term "application scanning", which amounts to static code analysis and related technologies such as software composition analysis. In addition to the general requirement for vulnerability assessment via static code analysis, there are requirements for:

- OWASP Top 10 (V-69513)
- Overflows (V-70277)
- Race conditions (V-70185)
- Error handling (V-70391)

Although this looks like a small set of vulnerabilities, the reality is this translates into many related software weaknesses. For example, the OWASP Top 10 translates to 53 CWEs, each of which have multiple related weaknesses. Although this is the set of vulnerabilities specific for compliance, it's prudent to consider a wider swath of vulnerabilities to detect.

IMPROVING SECURITY WITH STATIC ANALYSIS AND OWASP TOP 10

The Open Web Application Security Project (OWASP), as the name implies, is an organization that is committed to improving the security of web applications. Their OWASP Top 10 project provides a list of the most common and high-impact web application security vulnerabilities.

The latest version of OWASP Top 10 is directly correlated to specific CWE IDs. It's now much easier to implement as a coding standard while still using it for penetration testing and DAST tools.

OWASP Top 10 — 2017
A1:2017 — **Injection**
A2:2017 — **Broken Authentication**
A3:2017 — **Sensitive Data Exposure**
A4:2017 — **XML External Entities (XXE)**
A5:2017 — **Broken Access Control**
A6:2017 — **Security Misconfiguration**
A7:2017 — **Cross-Site Scripting (XSS)**
A8:2017 — **Insecure Deserialization**
A9:2017 — **Using Components with Known Vulnerabilities**
A10:2017 — **Insufficient Logging & Monitoring**

Figure 4:
OWASP Top 10

Compliance to the OWASP Top 10 centers around making reasonable efforts to avoid the most common and critical security issues facing web applications today. While it's possible to use static analysis tools to detect most of the issues, some are not statically analyzable. A9, for example, is related to Software Composition Analysis (SCA).

SCA is another term for analyzing the software supply chain. For example, when using open source in a project, it's important to make sure any known vulnerabilities in the code (such as CVEs) are fixed in the version being used. This is now commonly being used as a security requirement in medical, safety-critical, and government projects. Parasoft supports this by integrating the OWASP dependency checker with Parasoft's static analysis output into a single report that provides full Top 10 coverage.

Parasoft static analysis has out-of-the-box support for OWASP Top 10 through pre-configured settings and specific web dashboard reports for C/C++. Java and C#/.NET. OWASP reporting in Parasoft tools provides a fully auditable compliance framework for projects. These reports are integrated into a standards-specific dashboard like the one in Figure 5 for DISA-ASD-STIG.

Figure 5:
Parasoft DISA-ASD-STIG dashboard

The reports for OWASP compliance in the Parasoft DISA-ASD-STIG dashboard use the same risk rating methodology provided by OWASP. This scoring provides quick prioritization of reported violations to help developers focus on the most important security vulnerabilities first. Their methodology takes into account:

- How difficult it is for someone to exploit found weaknesses.
- How common the problem is.
- How easy it is for an attacker to find the weaknesses.
- What happens if the weaknesses were exploited.

This provides a solid basis to prioritize the issues that are most important to your organization and software.

Parasoft tools allow the level of enforcement to match the goals of your project. It's up to the project team to decide, based on risk assessment, which of the violations that appear in the code they are most concerned about.

Compliance reports are available on demand. Compliance criteria is flexible and specific to the team's project and codebase. Developers can craft policies based on severity, risk, impact, age of code, importance of components, and so on, and easily use them to guide development and show efforts to an auditor.

Although many vulnerabilities can be found via application code scanning, there's still a requirement for a dynamic audit of the software—application scanning. Security penetration testing tools are one category of these tools but there is also a place for verifying correct behavior through manual and automated functional testing. In addition, it's important that teams expand their focus beyond the guidelines in the DISA-ASD-STIG to include preventative guidelines like those included in secure coding standards.

OWASP Compliance Report

Filter: Quartz - OWASP **Target Build:** Quartz-2019-03-07 **Compliance Profile:** OWASP Top 10 2017 - .NET **Analysis Tool:** Parasoft dotTEST 10.4.2 **Revision Date:** 2019-02-27

Project Compliance: Not Compliant

Weakness Detection Plan | Deviation Report (Total: 8) | Build Audit Report

Exploitability: All | Prevalence: Widespread: 3 | Detectability: All | Impact: All | Compliance: All

Weakness	Exploitability	Prevalence	Detectability	Impact	Compliance	# of Violations	# of Deviations	
							In-Code Suppressions	DTP Suppressions
OWASP-A3	Average: 2	Widespread: 3	Average: 2	Severe: 3	Not Compliant	9	0	0
OWASP-A6	Easy: 3	Widespread: 3	Easy: 3	Moderate: 2	Not Compliant	323	0	8
OWASP-A7	Easy: 3	Widespread: 3	Easy: 3	Moderate: 2	Not Compliant	75	0	0
OWASP-A9	Average: 2	Widespread: 3	Average: 2	Moderate: 2	Compliant	0	0	0
OWASP-A10	Average: 2	Widespread: 3	Difficult: 1	Moderate: 2	Compliant	0	0	0

5 items

Figure 6: Parasoft OWASP compliance report

DISA-ASD-STIG VALIDATION METHODS

The DISA-ASD-STIG outlines ways to verify compliance with requirements, which include application code scanning (already discussed), application scanning, manual review, and functional security testing.

Functional testing is verifying with automated tools or manual testing that the vulnerability is not present in the software. In other words, "do something, check something" (i.e. check if the action worked properly and was logged if necessary).

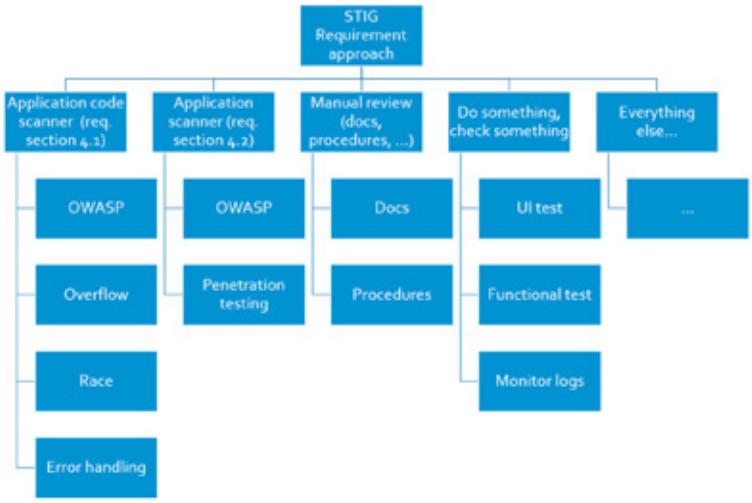


Figure 7:
STIG verification requirements

The functional verification of these STIGs looks daunting, but it's the familiar workflow that testers are already doing in their functional tests for software. For example, the UI login workflow illustrated in Figure 8 is familiar to most testers.

The illustration shows the flow of a user trying to log in to an application with the wrong password more than three times in a row. Testers confirm that the system locks the account, and that the attempt is logged in the appropriate log file. In other words, look at your policy, try the action, confirm it was handled correctly, and verify it was logged.

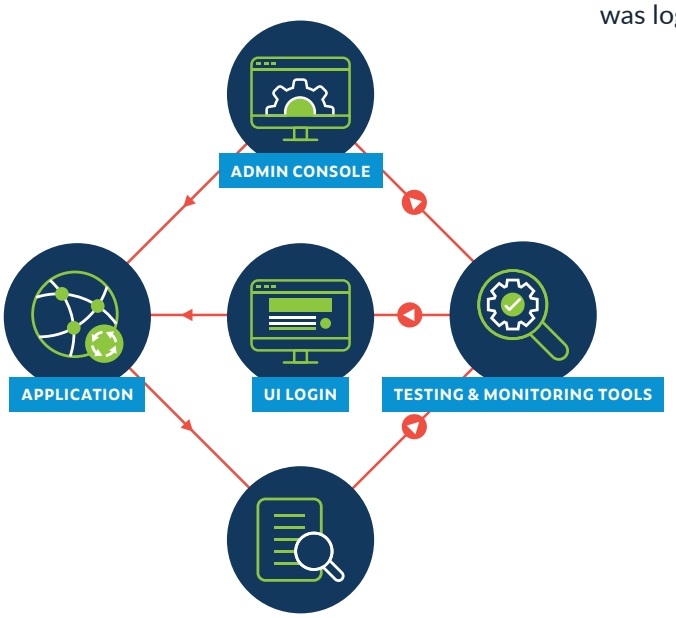


Figure 8:
Functional verification as "do something, check something."

API TESTING AND MESSAGING STANDARDS

A key area of test automation that benefits DISA-ASD-STIG testing is verifying API requirements and standards specified in the STIG. API tests are highly automatable with tools such as Parasoft SOAtest. For example, consider the DISA-ASD-STIG vulnerability ID V-69279, rule ID SV-83901r1, which states: "Messages protected with WS_Security must use timestamps with creation and expiration times."

One way to test for this vulnerability is by creating a test in SOAtest to examine SOAP messages and verify their time stamps, as illustrated below.

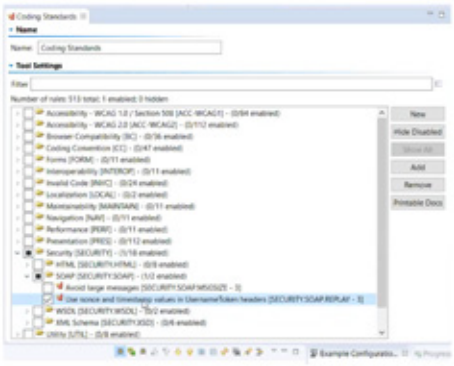


Figure 9: Parasoft SOAtest example

This test detects the lack of time stamps in the SOAP message headers and displays errors:

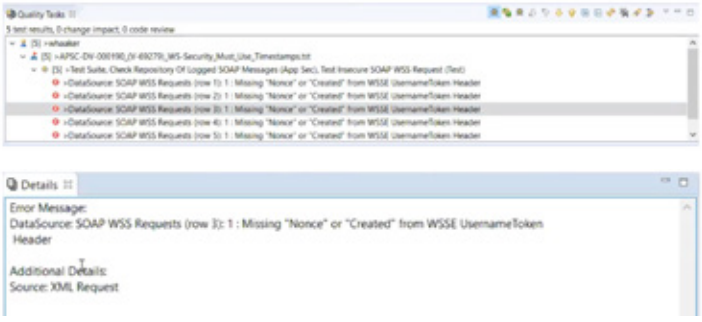


Figure 10: Parasoft SOAtest error message when vulnerability detected

It's possible to examine large streams of data for this vulnerability which would be tedious to do manually. This fits the workflow of "do something (automated), check something (automated.)" The preceding approach is a late lifecycle test where the software is nearly complete, and system and UI testing are taking place.

It's also possible to test this rule in development. The vulnerability gets removed earlier in the lifecycle when it's easier and cheaper to fix.

In the development phase, real clients are unavailable. This must be simulated by the tools and missing services can be virtualized if dependencies need to be met. Looking at the above example, we can create a client that sends SOAP requests to a secure server with the required WS-Security SOAP header.

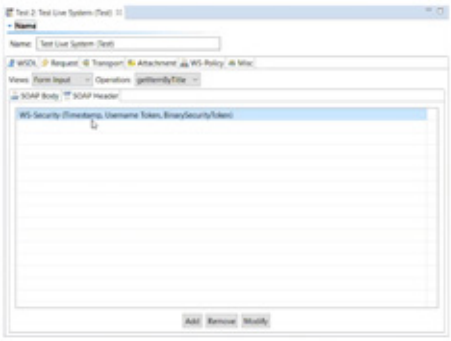


Figure 11: SOAtest example that simulates SOAP messages

When running this test, we discover that the SOAP request succeeds despite it being insecure due to missing time stamps. We're alerted by the tool. The expected behavior is for the service to send a SOAP fault message. If this doesn't happen, we get an error and we know the service is vulnerable.

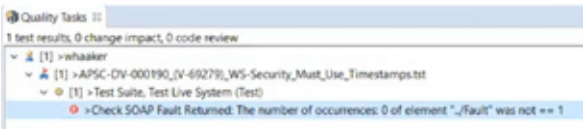


Figure 12: SOAtest error message when missing required information in SOAP header

This type of testing can be used for each API in the system; validating the security by checking for the listed vulnerabilities in the DISA-ASD-STIG. The benefit of this automated approach is the ability to clone and modify existing tests to suit other services, and the repeatability of the tests. Regression tests of this sort are completely automatable.

WHAT TO LOOK FOR IN SOFTWARE DEVELOPMENT AUTOMATION TOOLS FOR DISA-ASD-STIG COMPLIANCE

KEY FEATURES OF STATIC ANALYSIS TOOLS

Not all static analysis tools are created equally and there is more to each tool than just the analysis engine. The quality and depth of the analysis matter, and so does the storage and analysis of the results.

Integrations with other development tools such as IDEs and CI/CD pipeline tools are also important. Here are some of the key features that improve quality and security, and also speed up the adoption of the tools into the software development workflow:

- **Support to run in your IDE:** Static analysis works best when it catches coding violations, bugs and security vulnerabilities as the code is written. It's also critical that developers get access to the results of the current full-project build analysis and results.
- **Support to run on build servers and CI systems:** Running static analysis at the project level is also important since the scope of analysis includes all or most of the source code. Complex analysis such as data flow analysis works best in this mode. It's also important that the analysis integrates with an existing continuous integration and deployment toolchain and workflow.

  Additionally, the tools must keep track of each analysis on a per-file and per-build basis. Along with running in the IDE, you can implement a "trust but verify" policy. It's effective and doesn't impact your workflow negatively, unlike the way some strictly gated CI security implementations do.
- **Centralized configuration control:** Central control of testing and analysis configuration is critical for deploying the standards to all developers on the team. It's also the best way to tweak settings and deploy them consistently to the entire team. You can't rely on everyone having the correct standards. Using a centralized system enforces the same standards always.

» **Centralized reporting, audit, and analytics:** One of the crucial aspects of static analysis tools is the reporting and analytics capabilities. Projects create a large amount of data in terms of warnings and are multiplied build by build. How this data is managed is key to the successful adoption and return on investment for static analysis tools.

Dashboards, reports, and conformance tuned for each coding standard and security guideline are critical. Analytics that leverage risk models and help prioritize dramatically reduce the mountain of violations that information "straight out of the tool" (SOOT) can produce.

» **Full range of checkers:** A comprehensive set of checkers is important in order to support various use cases for static analysis. The set should include:

- Checkers that detect errors and vulnerabilities
- Checkers for prevention
- Checkers for so-called "bad smells" (code that doesn't look right on the first inspection and requires a closer look)

Supporting complex advanced checkers using data flow analysis is important and helps detect hard to find bugs. These bugs help you shift security left by testing earlier. To really harden your code, however, you also need to have checkers that prevent problems in the first place. For example, a checker that enforces input validation rather than just trying to find all possible ways to taint data. It's equally important to have comprehensive preventative checkers such as industry-standard security and safety guidelines like CERT, MISRA, OWASP, and CWE.

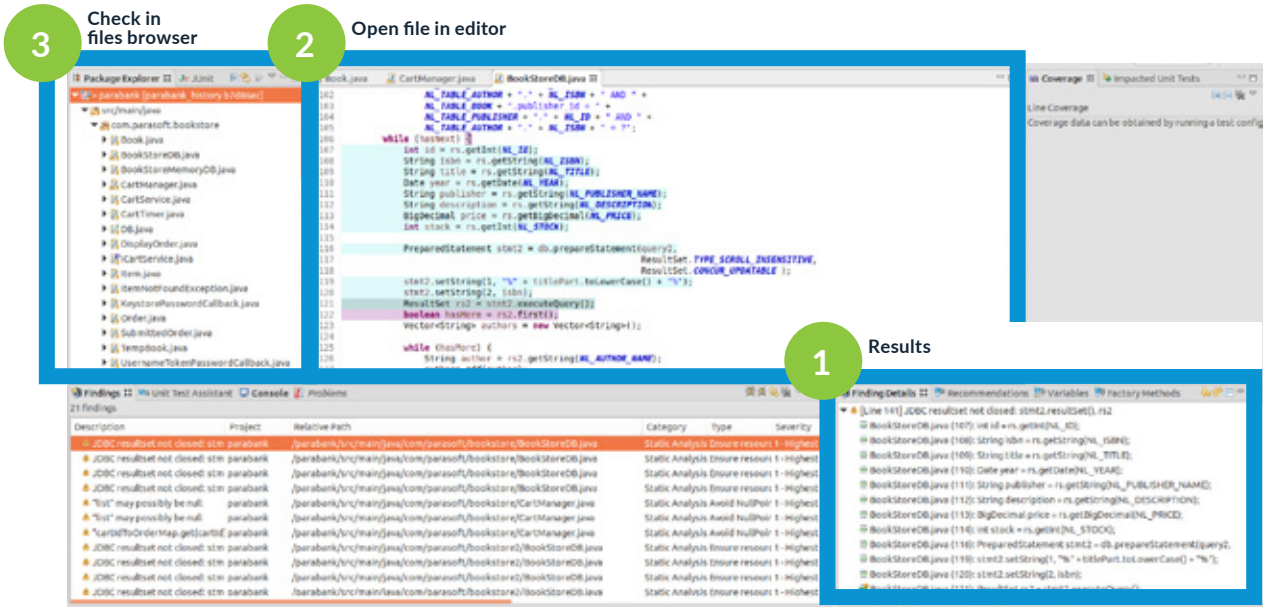


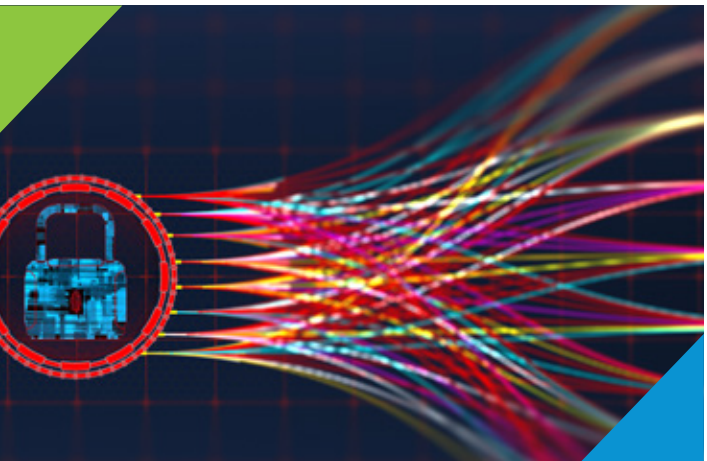
Figure 13: Parasoft Jtest integration with Eclipse

KEY FEATURES OF FUNCTIONAL TEST AUTOMATION TOOLS

Broad and simple creation: codeless and AI-powered. Your API testing tool should not require you to have any experience writing code. A codeless testing tool with an intuitive user interface will empower a large body of testers (at a variety of experience levels) to use the tool productively. API testing can be overlooked when developers push it to QA, and QA focuses on manual testing. Having an API testing tool that is visual and scriptless will enable testers to adopt this critical testing practice without having to spend lots of time on training and enablement.

Your API testing solution also needs to work with authentication, encryption, and access control. Many of your services will be deployed via an encrypted protocol such as SSL, as well as having a security policy such as OAuth, Basic auth, Kerberos, payload encryption, SAML, Signatures, etc. Additionally, you will need to validate that your security is working properly, so your API testing tool should have a mechanism to ensure that the standards are implemented properly and work flawlessly.

Test flow logic. Your API testing tool should have a mechanism for controlling test flow based on conditions. Not all test scenarios will execute in a linear fashion, so you may need to make automatic decisions at runtime that will affect how your test executes. An example of this might be ensuring that a response contains a specific element prior to moving to the next test step. Additionally, you may want to pause execution and poll a web service for a while to ensure a process has taken place. Your API testing solution must have the ability to analyze responses for key criteria and then use that information to control the rest of the test execution.



Test data management. Testers can spend a lot of time gathering adequate test data. Your API testing tool should support you in this activity by providing workflows for connecting to various data sources as well as generating test data itself. Your solution should have the ability to understand the types of data you require for given scenarios and build on that test data with additional use cases so that your test cases can be as flexible as possible.

Event monitoring. To enable end-to-end testing, your API testing solution must be able to monitor events as they flow through your system. You can validate inputs and expected outputs and understand how transactions transform as they move through your application. With multi-step validation by plugging into your application internals via JMS messaging, database monitoring, and so on, your solution will be able to provide greater levels of test coverage.

Support a large gamut of communication protocols. Modern software applications are comprised of a series of subsystems with many different message formats and protocols in place. You need functional test automation tools that go beyond supporting common HTTP based REST/JSON and SOAP/XML interfaces to cover everything from traditional interfaces (such as EDI, MQ, JMS, and SQL/Databases) to modern microservices and IoT protocols (such as Kafka, MQTT, AMQP) and beyond.

Integration into CI/CD pipelines and build systems. CI/CD is a critical component of DevOps/DevSecOps and accelerating delivery. To get the feedback you need to modernize the development process, integrate into the build process through scripted command line execution and into the CI/CD pipeline with native plugins.

A PRAGMATIC APPROACH TO DISA-ASD-STIG COMPLIANCE

The reality of software development for DoD, which requires DISA-ASD-STIG, is that you must test for all rules and vulnerabilities. It can be a daunting task, but automation is possible to lift some of the burden.

Parasoft's recommendation on how to approach complying with DISA-ASD-STIG is to leverage automation where it makes the most sense and use pre-emptive techniques to prevent vulnerabilities. It's more expensive and time-consuming to detect and fix vulnerabilities when software is almost complete versus during development. For this reason, Parasoft's approach is to "shift left" the vulnerability assessment, detection, and remediation earlier in the lifecycle.

FOR DEVELOPERS

Developers are less concerned with the larger scope of DISA-ASD-STIG requirements. However, there are critical steps they can take to make their life easier and reduce the backend workload during audits. A preventive, shift-left approach that makes use of automation is the key for developers.

Shift-left testing with code analysis. Using static analysis right from the start of development prevents vulnerabilities from making their way into the software in the first place. It's also a good way to assess the quality and security of legacy or third-party source code.

Turn on "code smells" and preventative standards checkers to harden the code. Beyond direct detection of bugs and vulnerabilities, it's important to prevent poor coding styles that can end up being a problem later.

Use static analysis for OWASP Top 10, overflow, race, and error handling. DISA-ASD-STIG specifically requires scanning for certain types of vulnerabilities. These should be done with an advanced static analysis tool that collates and analyzes results for later reporting and audits.

Use dynamic analysis and testing as needed for an audit. Software developers should leverage available tools as they can while the project progresses. As soon as code is testable, dynamic analysis and penetration tests should be started. Where a strong, secure coding practice has been established, use these tests to primarily validate the software is secure, rather than to find security issues.

FOR TESTERS

Testers in this environment are responsible for functional tests and for testing the STIG rules that require "do something, check something" validation. In some cases, test automation can help, while in others manual validation is required.

Build automated regression tests for STIG rules as is practical using functional API testing and service virtualization as necessary.

Manually test STIG rules that can't be automated.

SUMMARY

The DISA-ASD-STIG presents a fairly daunting set of requirements for securing software for DoD applications. There are various methods of demonstrating compliance with the rules outlined in the STIG—usually through audits of documentation, and reports and manual effort to use an application and check its logs. There are opportunities for automation in key areas outlined in the STIG such as application code and application scanning. Some of these are achieved through static analysis. Others through functional testing with a "do something, check something" approach to compliance.

A pragmatic approach that emphasizes preventative techniques that remove vulnerabilities early in the project lifecycle is recommended. Parasoft's static analysis provides early detection of vulnerabilities, and enforces code style and quality to prevent poor security practices as early as possible. Automating other STIG rules to the fullest with functional testing tools reduces the tedious manual testing to the max.

TAKE THE NEXT STEP

Detect vulnerabilities early and build quality into your software process from the beginning.

Talk to one of our experts to get started today.

LEARN MORE

How to Select and Implement the Right Secure Coding Standard

How to Choose a Modern Static Analysis Tool

ABOUT PARASOFT

Parasoft helps organizations continuously deliver quality software with its market-proven, integrated suite of automated software testing tools. Supporting the embedded, enterprise, and IoT markets, Parasoft's technologies reduce the time, effort, and cost of delivering secure, reliable, and compliant software by integrating everything from deep code analysis and unit testing to web UI and API testing, plus service virtualization and complete code coverage, into the delivery pipeline. Bringing all this together, Parasoft's award winning reporting and analytics dashboard delivers a centralized view of quality enabling organizations to deliver with confidence and succeed in today's most strategic ecosystems and development initiatives — cybersecure, safety-critical, agile, DevOps, and continuous testing.