Guidelines outlining security measures that shall be applied to the federal government's use of critical software

Critical software such as services supporting, sources of truth, provisioners, authenticators, audit services and any system processing or with the potential to access critical data requires that its operators have a clear understanding of how those services can be operated in a secure manner. That understanding must extend beyond a simple configuration guide or set of attributes stored within a configuration management system to include an evidence-based model with transparency into how the software was developed, tested and intended to be operated.

With modern software created from digital supply chains, for critical software it's insufficient to trust that a supplier uses secure development practices or that the supplier is using a variety of baseline testing tools such as static analysis, software composition analysis, dynamic analysis prior to each release. Running a testing tool neither represents that the tool is capable of identifying critical issues nor that the team is capable of redressing any findings returned from the tool. Further, there is no reason to assume that the supply chain or supplier hasn't been compromised in a way as to invalidate any self-attestation of compliance with any security standard.

As a result, a trust but verify model is required wherein suppliers have an obligation to describe the testing techniques they performed, provide the results of those tests with an indication of which findings were remediated prior to delivery of the software. This test transparency should be required during the procurement phase, and also with each update, patch, hotfix or major release. Having transparency throughout the lifecycle of the software helps ensure that compliance with security targets during procurement doesn't degrade as the software enters a sustainment phase and potentially a legacy mode.

Upon delivery of an application, the testing performed by the supplier needs to be verified. If the application framework or language is one without a distinction between "source" and "compiled", such as Node.js applications, then the verification process should include tests the supplier would be expected to perform on their source code prior to delivery, such as static analysis (SAST). For applications delivered in a compiled or binary form, such as a Microsoft Windows DLL, mobile application, or Linux shared object, verification testing should focus on confirmation of the security findings reported by the supplier. Some key test techniques valuable during this phase include:

- *Binary analysis*. Binary analysis is a form of software composition analysis designed to identify the development practices used by a development team. It commonly reports on open source components used within a binary, but also identifies aspects of how the application was compiled and how it operates. Some common elements that binary analysis can determine include compiler settings used for features like Address Space Location Randomization, dependencies on third party or external URLs, and embedded security tokens.
- *Interactive analysis (IAST) driven by DAST or API Fuzzing in a staging environment*. Interactive analysis allows the test tooling to inspect application operation using APIs provided by the language framework, but the IAST agent requires interaction by a user or test with the application to identify any issues. As such, IAST is limited by the test coverage, but by combining IAST with DAST or API Fuzzing, DAST can interact with exposed functions to expand IAST coverage. Additionally, the nature of DAST creates traffic flows that can identify implementation errors such as prototype pollution. By using

the combination of IAST and DAST in a staging environment, the impact of perimeter defenses is removed and IAST can be used to confirm the results of SAST performed by the supplier.

- *IAST driven by grey box penetration testing in a production environment*. As with IAST combined with DAST, IAST combined with grey box penetration testing seeks to confirm supplier security practices, but in this context the goal is to identify the effectiveness of perimeter and network access controls in mitigating unresolved security issues in the application. With this test configuration, the penetration team is provided with the results of prior testing and supplier test assertions allowing them to validate that the system meets security targets.
- *IAST driven log validation*. Since IAST operates based in interaction with an application under test, the application will naturally generate log output based on those interactions, be they positive or negative tests. This log information can then be used to validate the log format and flow information provided by the supplier and potentially define operational triggers related to log flows.

Each of these techniques are based on the premise that testing performed in isolation where the results of the testing effort aren't shared with teams operating software presumes the software development team's practices are implicitly trusted. As has been demonstrated continuously over recent years, organizational size is at best a minimal deterrent to a successful attack. Transparency of testing is one key element in changing that reality, but more importantly transparency allows teams to assess whether software developed and tested to one set of security targets can be deployed in an environment requiring a different set of security targets.

Evidence of source code testing should be provided with each release, patch or update to an application, where that evidence could be in the form of test results, third-party test attestations, or from a third-party repository of associated abstract syntax trees. Any source code testing results should include attestations about how exploitable weaknesses and vulnerabilities were addressed or mitigated such that those remediation and mitigation efforts can be validated prior to deployment.