SYNOPSYS®

Initial list of secure software development lifecycle standards, best practices, and other guidelines acceptable for the development of software for purchase by the federal government

As important as the security standards applied to source code might be, another large threat to the delivery of a properly secured application are weaknesses and assumptions within software development workflows. In effect, attestation of secure code practices and conveyance of test results are irrelevant if the development environment itself could be compromised. Mitigating this threat requires the development environment itself to be 'tamper evident' or minimally 'tamper resilient'.

Within a typical software development environment there are a limited number of services that are long running: source repositories, build management, and artifact repositories. For all other services, an ephemeral implementation both ensures reproducible outcomes while limiting the potential for compromise. In all instances, activity must be stored in immutable logs with activity monitoring in place. Contextually, only a limited set of developers will be authorized to perform changes to code within a given application at any point in time and the composition of development teams will also vary over time. The principle of least privilege should be applied to the software development lifecycle.

Applying development context and ephemeral execution to a standard development environment would yield:

- Source code access is provided by a long running source repository. Users are authenticated using MFA and are only authorized to specific codebases as appropriate to their current work assignment and authorization is revoked once their work assignment no longer includes access to that codebase. Repositories should be configured to append changes and disable any features that allow past code commits to be overwritten or effectively "rewrite history";
- The source repository is configured to validate that developer-executed testing has successfully completed prior to permitting any code commit. The results of that testing are then associated with the code commit;
- The source repository is configured to trigger the build management service following a successful code commit. The build management service is configured to provision build worker nodes in cloud infrastructure such as Kubernetes. Each build worker node is a container possessing the specific build tools required for the application language and framework, and the container image is digitally signed. The build management service doesn't directly build any code but instantiates an instance of a build worker node that builds the code to create any artifacts, stores them in an artifact repository, and automatically terminates the worker node;
- Upon a successful build, the build management service triggers security worker nodes in the same manner as build worker nodes. A security worker node is a container image possessing the tooling required to test the artifact using a specific technique and, as with build worker nodes, it is digitally signed. The results of the tests are stored in the artifact repository with the artifact being tested;
- Write access to the artifact repository is restricted to build and security worker nodes and read access to artifacts that have yet to be tested is also restricted. Only once acceptance criteria have been met are artifacts and testing results made readable by any accounts other than the build system.

While this workflow captures code development practices within a team that is exclusively using proprietary code and libraries, usage of open source software introduces critical changes due to how third-party component management is defined by a given language and development framework. Third-party component management is facilitated by the concept of a package manager that ensures that the correct version of third-party components are always used during the software build process.

In recent months there have been a series of targeted attacks on various package managers, all of which aim to pollute development environments with malicious components. Mitigating this class of attack is package manager specific, and package managers are programming language and application framework specific. While, for many package managers, disabling default behaviors and using fully qualified component names is an effective mitigation, that can not be assumed to be true for all package managers. Failure to properly configure package managers can lead to code of unexpected provenance being used in the creation of an application.

The outlines presented in this brief are one example of how a principle of least privilege can be applied to a software development lifecycle. Services should be active only for the minimum time required to perform their task, and access to those services should be limited to only those requiring access as part of an ongoing task. Monolithic services such as complex build pipelines performing multiple task groups, such as code assembly and all security testing, should be decomposed in ways that ensure artifacts that don't meet release criteria can't be deployed in production.