AdaCore

Executive Summary

AdaCore recommends that all security software:

1. **be developed preferably using principles of security by design, with particular emphasis on the systematic use of mathematical proofs of correctness;**
2. **be subjected to systematic static analysis early in the development cycle, with particular emphasis on those parts of the software for which mathematical proofs were not established;**
3. **be subjected to dynamic analysis with 100% statement coverage.**

Detailed Position Statement

This paper aims at providing AdaCore's input on the following area, that NIST is considering as part of the implementation of Executive Order 14028 of May, 12 2021:

> *Minimum requirements for testing software source code including defining types of manual or automated testing (such as code review tools, static and dynamic analysis, software composition tools, and penetration testing), their recommended uses, best practices, and setting realistic expectations for security benefits.*

More specifically, it will focus on three domains where AdaCore has specific expertise: security-by-design, static and dynamic analysis.

Security-by-Design

For the security-critical sections of the software, **programming languages that ensure security-by-design** should be favored over manual coding with languages such as C and C++, whose lack of strong typing, error-prone syntax and memory unsafety are a major cause of security vulnerabilities. By "programming languages" we mean traditional textual languages as well as modelling languages.

AdaCore recommends programming languages for which a **mathematical proof** can be computed that the software will behave as it is intended to. Examples of such formal technologies include the SPARK, Frama-C and Polyspace Prover technologies. SPARK, if used at the silver level or above, can prove the absence of runtime errors, a valuable property as buffer overflows are one of the most common vulnerabilities in software. Security-by-design also includes domain-specific languages that can be used to describe formally the expected behavior of the software and then automatically generate a provable implementation. Examples of this approach can be found with the Microsoft Research EverParse or AdaCore's RecordFlux technologies, which can be used to formally specify a communication protocol and generate secure implementation of the same.

These technologies are necessary for a complete long-term solution to some of the software-related security issues that need addressing, yet the market still needs to mature in order to adopt them pervasively. The main challenges associated with these approaches are (i) the difficulty to find engineering talent qualified with formal languages and provers or able to use these domain-specific languages; and (ii) the necessity to use security-by-design concepts from the beginning of the software development lifecycle: legacy code cannot be secured *ex post*, only specific security-critical components within legacy code can be rewritten from scratch using these technologies.

Static Analysis

Static analysis includes all the other technologies that can be used to assess the security of software without having to execute it. As opposed to security-by-design, it doesn't require the use of a specific language to identify and eliminate the programming constructs that are likely to cause specific weaknesses such as those listed in the CWE database.

The use of static analysis is already widespread in the US aerospace and defense industry: static analysis is easy to use and broadly applicable to both new and legacy software of any level of criticality. The static analysis market is mature and includes a large number of static analysis tools, including CodePeer and Polyspace for Ada, Synopsys Coverity for C, and many others.

The main challenge of static analyzers lies with the ability to practically analyze their messages, in particular when the rate of false positives is high. Several **best practices** can help to address this challenge:

- defining a coding standard for the application;
- doing static analysis early in the development lifecycle, ideally as a development activity;
- doing static analysis on limited portions of the code at a time;
- having developers who know best the code analyze the results.

From an organizational standpoint, these best practices can be summed up as integrating static analysis in a peer review process, or in a continuous integration pipeline. On the contrary, having static analysis performed at the end of the software development lifecycle, typically by test and validation engineers on the whole system, is far less effective.

No matter how static analysis is performed, it is important to point out that performing static analysis on the code will not remove all weaknesses, but only ensure that well identified ones are systematically detected. To this extent, **static analysis provides weaker guarantees than security-by-design**.

Dynamic Analysis

Dynamic analysis includes any technology that requires the execution of the software to assess its security. Dynamic analysis can be used either at the implementation stage (e.g. unit testing) or at the integration stage (e.g. system testing) and even at runtime (e.g. using runtime checks). Fuzz testing and model-based testing are two of the most commonly used dynamic analysis technologies. An important aspect, regardless of the method used, is to ensure that a sufficient coverage of the code under test is achieved. Our position is that **less than 100% statement coverage should be considered inadequate.** Software that is particularly critical may be subject to more stringent requirements: for instance, DO-178C requires DAL A software to achieve 100% MC/DC coverage.

Fuzz testing involves generating test vectors automatically from a known seed to identify vectors that can cause unwanted behaviors, such as crashes. Fuzz testing can be performed at the system level or at the unit level.

Fuzz testing is complementary to static analysis in that the mutation mechanism can find complex vulnerabilities that have not been formally identified before and involve interactions between different subsystems, as proven by the many high profile vulnerabilities found thanks to fuzzing technologies: OSS-fuzz has for instance found 25,000 bugs in various projects, while syzkaller claims 2854 bugs found in the Linux kernel, with 2046 fixed. Examples of dynamic analysis products include ForAllSecure's Mayhem, Synopsys Defensics, or AdaCore's GNATFuzz.

Model-based testing involves generating test vectors from a formal model, to thoroughly cover the actual execution space. A typical example can be found with the RecordFuzz technology that AdaCore is developing, where a formal description of a binary communication protocol - which can also be used to generate a provable, secure implementation with the RecordFlux tool - is used to generate test vectors for existing implementations of the same protocol. This type of dynamic analysis can be used as the entry point for further **penetration testing.**

From our perspective, the market for system fuzzing is technologically mature, and adoption in the industry is bound to increase. On the other hand, technologies to allow developers to use **dynamic analysis earlier in the development lifecycle are still lacking widespread use** and may be considered as a best practice to improve security of software where formal methods are not an option.

Contact Information

150 W. 30th Street, 16th floor
New York, NY 10001
USA
Tel. +1 212 620 7300
www.adacore.com

Romain Berrendonner
Security Solution Architect
berrendo@adacore.com