Positive Assurance:

an approach to making the assurance case for software

Larry Maccherone
CyLab - Carnegie Mellon
December 2007

Abstract

Tools and techniques are emerging that allow us to directly evaluate software artifacts to gain assurance that they are free of exploitable vulnerabilities. This is complementary to the current capability to assess the process used to build the software and the ability to assess the specification and design of security-relevant features (encryption, authentication, etc.). The current ad-hoc use of these direct evaluation tools and techniques results in critical gaps in coverage and inefficiencies. In this paper, we explore an approach to using these tools in a more structured way that aligns with the desire to document an "assurance case" for a particular piece of software.

The approach starts with having the user specify a list of narrowly defined attributes to assure. This list will generally be a subset of standard dictionaries like Mitre's Common Weakness Enumeration (CWE) (Mitre) or CERT's Secure Coding Standards (CERT) but may also include attributes for a particular purpose/product/organization. The user can decide upon the initial list based upon risk and then evolve the list over the life of the product. The user will then choose a regime of tools and techniques to "positively assure" each of the attributes enumerated in the chosen set. The output of using those tools and techniques is all gathered into a measurement information system so it is available as evidence for the assurance case.

This paper (1) defines "positive assurance", (2) gives more detail on this proposed approach, (3) discusses some of the issues that we expect to arise, and (4) explores the use of this approach with an example from the CWE and CERT's Secure Coding Standards.

Problems with current approach

We have fairly mature means and metrics for evaluating the processes that we use to create software. We also have long-standing capability to evaluate the specification and design of the security features in software products (encryption, authentication, etc.). However, these capabilities provide insufficient evidence that the software as-implemented is free from exploitable vulnerabilities.

Technical means are now emerging to support the direct evaluation of software artifacts for security-relevant attributes. Particularly in the last few years, progress has been made in providing efficient means to directly evaluate software artifacts using static source/binary analysis, dynamic analysis, model checking, testing and test generation technology, inspection techniques and tools, etc. By focusing on particular software attributes, these technologies are beginning to overcome the barriers of scale and adoptability, and are now being employed in a variety of ways by developers as well as purchasers and evaluators of software. Our studies through the DoD sponsored Code Assessment Methodology (CAMP) project have shown that the various tools in the market cover a wide range of software attributes related to security, and that there is complementarily coverage from individual tools. The approach to tool usage has an ad hoc character which leads to several categories of problems:

- **Decision making** - The output of these tools/activities is generally not aggregated into decision aids that can support assessments tuned to particular operational needs.
- **"Lampost" coverage** - A woman happens upon a man on his hands and knees under a lamppost outside a tunnel. What is he doing? Looking for his keys. Is that where he lost them? No, he lost them inside the tunnel. Why isn't he looking there? It's dark in there; it's light under the lamppost. Similarly, ad-hoc tool-driven efforts have a tendency to focus on what the tools are good at detecting and may ignore critical vulnerabilities. The evaluation of a product should be driven by first identifying what attributes are most important in light of the product's intended use, environment and implementation technology and then choosing the most cost effective way to assure them.
- **Duplication** - Many of the tools/activities are targeted at similar issues but this duplication is not easily recognized until tools are deployed. Even when it is known that two tools/activities target the same issue, without good measures, it's hard to know which has the best coverage (false positives, false negatives) and the best benefit/cost function.

Emerging trends and technology that offer hope for a solution

We see several ongoing trends that offer hope that the solutions proposed here are feasible.

Trend: Analysis technology adoption

Formal methods for proving the correctness of code has been around for many decades. However, recently have we seen significant adoption of analysis technology in the form of usable tools:

- Microsoft's FxCop, SAL and SLAM
- Commercial static analysis tools from companies like Fortify, Coverity, GrammaTech, KLOCWork and others
- Open source analysis tools like FindBugs and PMD

It appears that the problems of scale and adoptability that plagued earlier analysis technology have been largely solved and more tools like the ones above will emerge.

Trend: CWE and CERT's Secure Coding Standards

The duplication and coverage problems identified above cannot be solved without the ability to cross-reference the output of these tools and assurance activities against a common dictionary. Up until now, this has been a significant factor preventing the development of a useful measurement system for direct evaluation of products for security and software assurance. Recently, Mitre has gathered just such a dictionary in the form of their Common Weakness Enumeration (CWE). Mitre describes the CWE as, “a formal list of software weakness types created to... serve as a standard measuring stick for software security tools targeting these weaknesses [and] provide a common baseline standard for weakness identification, mitigation, and prevention efforts.”

Additionally, there are other efforts emerging that can satisfy this cross-referencing need. The CERT Secure Coding Standards is an example that complements CWE for particular needs related to C and C++ programming.

At this moment, the CERT work is more precisely defined but the CWE covers a wider array of issues and programming languages. However, the CWE is making progress at increasing the precision in the way items are defined and is improving the organizational structure of the enumeration. Likewise, the CERT is discussing plans to expand their Secure Coding Standards to cover the Java programming language. There is significant communication and even acknowledged sharing between these efforts so they are likely to improve together.

Trend: Test-driven development adoption

Test-driven development (TDD) advocates that automated test should be written while the production code is being written (if not before) and should become a maintained part of the software product. These tests will certainly make up some portion of the assurance case. Additionally, the rising adoption rate of TDD is indicative of an overall shift in the mindset of developers and teams. They now value the benefits of having a continuous check on their own work enough to invest in the creation of these substantial artifacts. Furthermore, many TDD techniques require changes in the actual designs and coding idioms used in the production code. Recent developer acceptance of "design for testability" is promising because other assurance techniques and technologies are made easier/possible if the developer is willing to engage in "design for assurability".

Trend: Software processes that are considered agile, but still manage with measurements and are considered "high-maturity"

It's a common misconception that agile software development methods do not use measurements and don't follow a defined process. Many high-maturity processes that have been defined to satisfy the requirements of process criteria like the SEI's Capability Maturity Model Integrated (CMMI), have been heavy weight. Agile methods emerged, in part, as a response to this tendency to focus on the process and measurement while forgetting about the ultimate goal of producing reliable software that meets clients' needs.

However, in order to achieve this goal, agile methods don't shun the use of measurements or clearly-defined, robust processes. The process design just takes into account the intricate balance of developer motivation, team dynamics and the realities of changing needs. The measurements are as simple as possible while still satisfying the overall need. There are direct parallels between agile method's "points" and estimation metrics that have been around for a while. Similarly, metrics like velocity and burn-down have direct parallels in traditional process definitions.

Moreover, we are now starting to see organizations with more traditional high-maturity processes incorporate agile methods into their overall approach. These organizations get productivity gains and higher adoption rates while maintaining their high-maturity level.

Positive assurance

The ad-hoc use of analysis tools along with testing and inspection should result in higher quality software with fewer vulnerabilities. This is enough motivation for these "bug finding" tools and techniques to see significant adoption. However, when you are "finding bugs", how do you know you are done? When you've found a hundred? A thousand? A million?... The point is that if you think of what you are doing as finding bugs, you never know when you are done.

On the other hand, if you enumerate and prioritize narrowly defined attributes of concern, and you then devise mechanisms that will give you ongoing evidence that the code is free of all instances of that narrowly defined attribute of concern, then you can at least say something "positive" about the code, e.g. that in all likelihood the code is free of this particular issue.

Positive assurance is not the same as proof that the code is free from all types of vulnerabilities; only that you can show evidence that it is free of the ones in your chosen set. Hopefully, that set will cover the most common issues first and will evolve as priorities change. Nor is it absolute proof that the code is free from even those in your chosen set. We stop short of choosing a legalistic definition like "preponderance of evidence" or "beyond a reasonable doubt", but we expect the community to evolve standards such as these.

Envisioned system

1. Specify a list of narrowly defined attributes to assure

The proposed approach starts with having the user specify a list of narrowly defined attributes to assure. This list will generally be a subset of standard dictionaries like Mitre's Common Weakness Enumeration (CWE) (Mitre) or CERT's Secure Coding Standards (CERT) but may also include attributes for a particular purpose/product/organization. The user can decide upon the initial list based upon risk and then evolve the list over the life of the product. Lower priority items might not be added until later in the development cycle.

2. Assure with chosen tools and techniques

The user will then choose a regime of tools and techniques to "positively assure" each of the attributes enumerated in the chosen set. The use of such tools will require:

a. Design for assurability - design patterns that isolate a specific aspect of concern or that, if followed, prevent the occurrence of some potential issues

b. Rules - a mix of generic and custom rules that will determine that the implementation follows the design patterns and that the code is free of identified issues

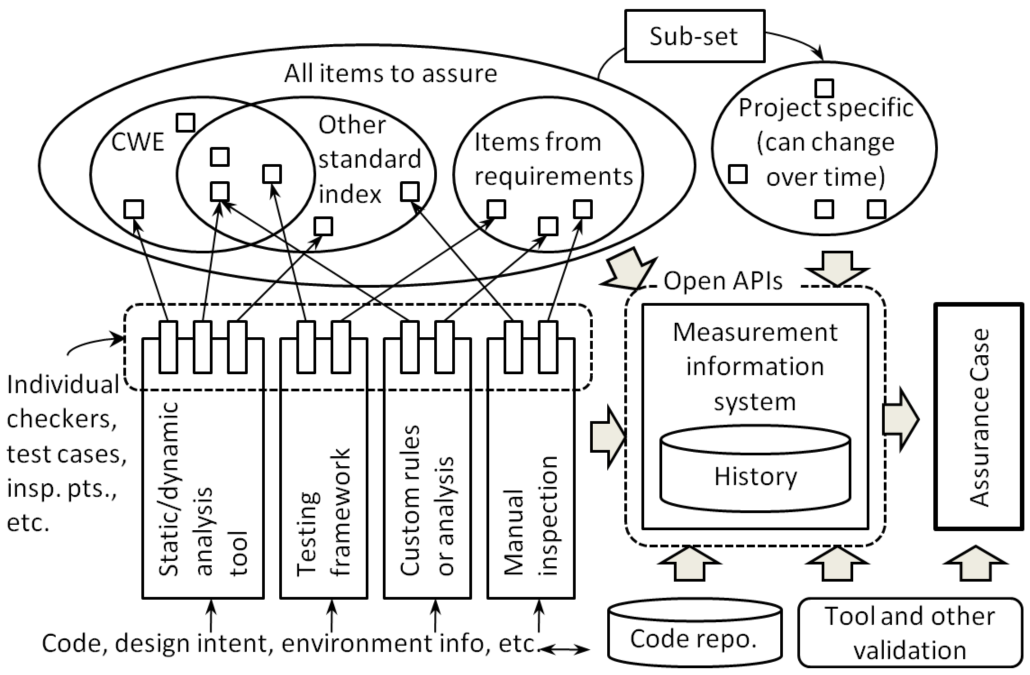
c. Design intent - embedding/attaching of all design intent necessary for the rules to operate, possibly in the form of additional annotations to the code

d. Sound analysis engines - evidence of the soundness of a particular analysis engine, given an implementation and design intent, to determine that the rules are followed

3. The output of these tools and techniques are aggregated in a measurement information system and tracked over time

The output of using those tools and techniques is all gathered into a measurement information system so that it is available as evidence for the assurance case. The data from this measurement information system must necessarily be an integral part of the overall process definition which will include traditional measures like effort, size and defects. The traceability of assurance measures to traditional productivity and quality inputs will allow the cost/benefit ratio of particular assurance activities to be calculated.

Measurement information system requirements


Sub-set
All items to assure
CWE
Other standard index
Items from requirements
Project specific (can change over time)
Open APIs
Measurement information system
History
Assurance Case
Individual checkers, test cases, insp. pts., etc.
Static/dynamic analysis tool
Testing framework
Custom rules or analysis
Manual inspection
Code, design intent, environment info, etc.
Code repo.
Tool and other validation

As indicated by the diagram above, the envisioned measurement system would:

- Import data through defined APIs from a variety of sources:
    1. popular analysis tools (e.g. Fortify's broad spectrum of security-related analyses, the widely-used open source Findbugs tool, and SureLogic's deep Java analyses),
    2. test-case output triggered by build tools like Ant or Maven and tools such as JUnit, and
    3. baseline measures gathered directly from the code in source code repositories like Subversion.
- Allow for the easy input of manual data from sources such as manual inspections
- Enable the specification of sets of security attributes or rules as input to the measurement functions and analysis
- Maintain the cross references between the attributes in the chosen set(s) against standard dictionaries like the CWE and CERT's Secure Coding Standards
- Generate derived measures
- Include visualizations that enable indicator and information product usage
- Maintain a history of measurement and analysis results traceable to the development history of the product as well as the individual attribute checkers
- Include facilities to enable the sharing of historical information like the false positive/false negative rates of individual attribute checkers beyond an individual team to the entire organization and possibly the broader community

It is expected that organizations will derive value from using the system to manage ongoing assurance activities as well as to collect data for cost/benefit analysis and other decision making analogous to the longitudinal data collection that supports cost estimation, release-date prediction, and the like. When organizations are willing to share data more broadly, then the model can yield value sooner and with greater precision.

Example from CERT's Secure Coding Standards

The CERT's Secure Coding Standard for C includes roughly ninety rules and about as many recommendations for writing secure code in the C programming language. The rules are all specified in a standard format which includes a discussion of the issue along with non-compliant and compliant code examples. Due to funding from the CAMP project and partially motivated by the subject of this paper, we have added a section for mitigation strategies to some of the rules. The goal of this effort was to start to identify and classify the assurance techniques that would be needed for this approach and get a feel for the cost and feasibility.

For instance, the rule, FIO02-A. Canonicalize file names originating from untrusted sources (Seacord & Christopher) includes such a section on mitigation strategies. It follows a common pattern that we see in many of the rules. Some manual inspection is called for to determine that the actual canonicalization is coded correctly. Static analysis can be used to confirm that all code paths of interest utilize this canonicalization routine. The concept of type states makes this sort of analysis clear (Strom & Yemini, 1986). In practice, extra design intent would be injected into the code to identify when the data changes state (Bierhoff, 2006).

We found that all of the rules that we looked at could be positively assured with some combination of static analysis, dynamic analysis and minimal manual inspection.

Future work

More work is needed to identify the effort involved in using this approach and identifying the gaps that will need to be filled before it can reach widespread adoption.

Robert Seacord and his team at the CERT are in the process of defining a set of custom rules using the Fortify commercial tools as well as the ROSE compiler-to-compiler framework (Quinlin, 2000) (Quinlin, ROSE Open Compiler Infrastructure for Parallel Source Code and Binary Security Analysis, 2007) from Lawrence Livermore National Labs. The goal is to field test a set of automatically assured rules that provides close to complete coverage of the Secure Coding Standards for C or C++. That work should give us a better idea on how much effort is involved in accomplishing that aspect of the approach.

We also have plans to pilot a measurement information system similar to the one described in this paper. The pilot will make use of tools that automate the gathering of process data. These tools include the Hackystat tools out of the University of Hawai (Johnson, Hongbing, Augustin, & Miglani).

Acknowledgements

This work was funded in part by the United States Department of Defense. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the sponsors.

The author would like to acknowledge fruitful discussions with William Scherlis, Robert Seacord, Chris Telfer, Ciera Christopher, and Jonathan Aldrich.

Bibliography

Bierhoff, K. (2006). Iterator specification with typestates. *SAVCBS '06: Proceedings of the 2006 conference on Specification and verification of component-based systems* (pp. 79-82). Portland, Oregon: ACM.

CERT. (n.d.). *CERT Secure Coding Standards*. Retrieved December 1, 2007, from http://www.securecoding.cert.org

Johnson, P., Hongbing, K., Augustin, J., & Miglani, J. (n.d.). *Hackystat*. Retrieved December 1, 2007, from CSDL: http://csdl.ics.hawaii.edu/Research/Hackystat/

Mitre. (n.d.). *CWE - Common Weakness Enumeration*. Retrieved December 1, 2007, from Mitre: http://cwe.mitre.org

Quinlin, D. (2007, September 18). *ROSE Open Compiler Infrastructure for Parallel Source Code and Binary Security Analysis.* CyLab, Pittsburgh, PA.

Quinlin, D. (2000, June). ROSE: Compiler support for object-oriented frameworks. *Parallel Processing Letters (PPL) , 10* (2/3), pp. 215-226.

Seacord, R., & Christopher, C. (n.d.). *FIO02-A. Canonicalize file names originating from untrusted sources.* Retrieved December 1, 2007, from CERT's Secure Coding Standard for C: https://www.securecoding.cert.org/confluence/display/seccode/FIO02-A.+Canonicalize+file+names+originating+from+untrusted+sources

Strom, R. E., & Yemini, S. (1986). Typestate: A programming language concept for enhancing software reliability. *IEEE Trans. Softw. Eng. , 12* (1), 157-171.