

NIST Special Publication 500-320

**Report of the Workshop on
Software Measures and Metrics to
Reduce Security Vulnerabilities
(SwMM-RSV)**

Paul E. Black
Elizabeth Fong

This publication is available free of charge from:
<https://doi.org/10.6028/NIST.SP.500-320>

NIST
National Institute of
Standards and Technology
U.S. Department of Commerce

NIST Special Publication 500-320

**Report of the Workshop on
Software Measures and Metrics to
Reduce Security Vulnerabilities
(SwMM-RSV)**

Paul E. Black
Elizabeth Fong
*Software and Systems Division
Information Technology Laboratory*

This publication is available free of charge from:
<https://doi.org/10.6028/NIST.SP.500-320>

November 2016



U.S. Department of Commerce
Penny Pritzker, Secretary

National Institute of Standards and Technology
Willie May, Under Secretary of Commerce for Standards and Technology and Director

Certain commercial entities, equipment, or materials may be identified in this document in order to describe an experimental procedure or concept adequately. Such identification is not intended to imply recommendation or endorsement by the National Institute of Standards and Technology, nor is it intended to imply that the entities, materials, or equipment are necessarily the best available for the purpose.

National Institute of Standards and Technology Special Publication 500-320
Natl. Inst. Stand. Technol. Spec. Publ. 500-320, 84 pages (November 2016)
CODEN: NSPUE2

This publication is available free of charge from:
<https://doi.org/10.6028/NIST.SP.500-320>

Abstract

The National Institute of Standards and Technology (NIST) workshop on Software Measures and Metrics to Reduce Security Vulnerabilities (SwMM-RSV) was held on 12 July 2016. The goal of this workshop is to gather ideas on how the Federal Government can identify, improve, package, deliver, or boost the use of software measures and metrics to significantly reduce vulnerabilities.

This report contains observations and recommendations from the workshop participants. This report also includes position statements submitted to the workshop, presentations at the workshop, and related material. Ideas from the workshop will be included in the Dramatically Reducing Software Vulnerabilities report, requested of NIST by the White House Office of Science and Technology Policy in Spring 2016.

Keywords:

Measurement; metrics; software assurance; security vulnerabilities; reduce security vulnerabilities.

Disclaimer:

This report includes position statements and presentation slides by authors who submitted their material to the workshop. The views expressed by the authors therein do not necessarily reflect those of the sponsors of this workshop.

Certain commercial entities, equipment, or materials may be identified in this document in order to describe an experimental procedure or concept adequately. Such identification is not intended to imply recommendation or endorsement by the National Institute of Standards and Technology (NIST), nor is it intended to imply that the entities, materials, or equipment are necessarily the best available for the purpose.

Acknowledgement:

The authors thank Peggy Himes and Rose Linares of NIST for their valuable help with the workshop and formatting this manuscript. Thanks to Simson Garfinkel for significant contributions, which improved this report. Thanks also to the many workshop participants who contributed, discussed, refined, and wrote up ideas.

Contents

1. Overview	1
1.1 Mechanics and Organization.....	2
1.2 Agenda and Schedule.....	2
1.3 Other Ideas from Breakout Session	3
1.3.1 Consider Vulnerabilities in All Parts of the Software Life Cycle	4
1.3.2 Government Contracting and Procurement, Liability, and Insurance	4
1.3.3 Education	5
1.3.4 Research Projects for Security, Quality, and Few Vulnerabilities.....	5
1.3.5 Government Funded Efforts	6
1.3.6 Third Party Review of Software	6
2. Observations and Recommendations	7
2.1 Better Code	7
2.2 More Useful Tool Outputs	7
2.3 Security Metrics	8
2.4 Additional Directions.....	8
2.5 References	8
3. Position Statements and Presentations.....	9

1. Overview

The 2016 Federal Cybersecurity Research and Development Strategic Plan [1] seeks to fundamentally alter the dynamics of security in the computer realm, reversing adversaries' asymmetrical advantages. The plan calls for “sustainably secure systems development and operation.” To achieve this, the plan describes a mid-term (3-7 years) goal of “the design and implementation of software, firmware, and hardware that are highly resistant to malicious cyber activities...” and reduce the number of vulnerabilities in software by orders of magnitude.

The Software Quality Group at the U.S. National Institute of Standards and Technology (NIST) felt that measures of software can play an important role in such dramatic reductions. Industry requires evidence to indicate how vulnerable a piece of software is, know what techniques are most effective in developing software with far fewer vulnerabilities, determine the best places to deploy defensive measures, or take any of a number of other actions. This evidence comes from measuring, in the broadest sense, or assessing properties of software. If there were comprehensive metrics, it would be straight-forward to determine which software development technologies or methodologies lead to sustainably secure systems.

Accordingly, in Spring 2016 we decided to organize a workshop to gather ideas on how the Federal Government can identify, improve, package, deliver, or boost the use of software measures and metrics to significantly reduce vulnerabilities. We called for short position statements, one to three paragraphs long, to begin to gather ideas. We asked for position statements, rather than papers, to decrease the work required of submitters. In the call for statements, we suggested the following subjects:

- Existing measures of software that can make a difference in three to seven years,
- Means of validating software measures or confirming their efficacy (meta-measurements),
- Quantities (properties) in software that can be measured,
- Standards (in both *étalon* and *norme* senses) needed for software measurement,
- Cost vs. benefit of software measurements,
- Surmountable barriers to adoption of measures and metrics,
- Areas or conditions of applicability (or non-applicability) of measures,
- Software measurement procedures (esp. automated ones), or
- Sources of variability or uncertainty in software metrics or measures.

Twenty positions statements were submitted. A program committee evaluated the submissions for relevance to the workshop theme and potential interest. The committee invited workshop presentations based on 10 statements.

Ideas from this workshop and other efforts will be included in the report on Dramatically Reducing Software Vulnerabilities, requested of NIST by the White House Office of Science and Technology Policy in Spring 2016.

The workshop was open to all, subject to the rules of entry to the NIST campus, and there was no cost to attend.

1.1 Mechanics and Organization

The workshop was co-chaired by Paul E. Black and Elizabeth Fong, National Institute of Standards and Technology and by Thomas D. Hurt, Office of the Deputy Assistant Secretary of Defense for Systems Engineering - Joint Federated Assurance Center (JFAC) lead.

The program committee consisted of Paul E. Black, David Flater, Elizabeth Fong, D. Richard Kuhn, and W. Timothy Polk.

The web site is <https://samate.nist.gov/SwMM-RSV2016.html>. In late April and early May, co-chairs sent the call for statements to mailing lists and many individuals. Here is the timeline:

- 22 May: deadline to submit statements
- 8 June: invitations to present sent
- 27 June: deadline for non-citizens to register
- 5 July: deadline for US citizens to register
- 12 July: workshop
- 31 July: deadline for submission of revised statement or presentation

1.2 Agenda and Schedule

The workshop was held at the National Institute of Standards and Technology, Gaithersburg, Maryland, on 12 July 2016. Over 90 people from Federal Government, software assurance tool makers, service providers, and universities attended. The program consisted of nine presentations invited on the basis of position statements and one breakout session. For the breakout, attendees were randomly assigned to one of six breakout groups. All the groups had the same charge: come up with the best ideas to use metrics to dramatically reduce software vulnerabilities in three to five years. The agenda was as follows:

8:30 am – 9:00 am	Registration
9:00 am – 9:10 am	Introduction, Safety, Schedule, Charge Paul E. Black, NIST
9:10 am – 9:15 am	Federal Cybersecurity Research and Development Strategic Plan Greg Shannon, White House Office of Science and Technology Policy
9:15 am – 9:30 am	Opening Remarks William F. Guthrie, Chief, Statistical Engineering Division, NIST
9:30 am – 10:00 am	Measuring Software Analyzability Andrew Walenstein, BlackBerry
10:00 am – 10:30am	Dealing with Code That is Opaque to Static Analysis James Kupsch, University of Wisconsin-Madison

10:30 am – 10:50 am	Break
10:50 am – 11:10 am	Composing Processes for Secure Development Using Process Control Measures William Nichols, Software Engineering Institute (SEI)
11:10 am – 11:30 am	Measure Early and Measure Often – SWAMP Miron Livny, Morgridge Institute for Research
11:30 am – 1:00 pm	Lunch
1:00 pm – 1:20 pm	CISQ Measures of Secure, Resilient Software Dr. Bill Curtis, Executive Director, Consortium for IT Software Quality (CISQ)
1:20 pm – 1:40 pm	Mostly Sunny with a Chance of Cyber-Doom David Flater, NIST
1:40 pm – 2:00 pm	Dynamically Proving That Security Issues Exist Dr. Andrew V. Jones, Vector Software, Inc.
2:00 pm – 2:20 pm	Breakouts
2:20 pm – 2:50 pm	Break
2:50 pm – 3:20 pm	Breakout Reports (6 reports at 5 minutes each)
3:20 pm – 3:40 pm	Toward Evidence-Based Low Defect Software Production James Kirby, Jr., US Naval Research Laboratory
3:40 pm – 4:00 pm	Using Malware Analysis to Reduce Design Weaknesses Carol Woody, Ph.D., Software Engineering Institute
4:00 pm – 4:20 pm	Summary – Our Next Step Paul E. Black, NIST

Although UL was invited to present based on their position statement, there was no presentation because of illness.

1.3 Non-Measurement Ideas from the Breakout Session

The discussions in the breakout groups were lively. Most of the groups continued their discussions to the end of the break time. Someone from each breakout group took five minutes to report their recommendations to the whole workshop when it reconvened. The workshop was focused on metrics and measures of software as a product and what could be done in a moderate time frame. Although charged to discuss ideas related to the workshop theme, every group included ideas related to software quality, assurance, software development, and cybersecurity in general. This section lists many of those ideas that are outside the scope of the workshop, and thus are not in Section 2. Some came from more than one group.

When we use phrases like “workshop participants” or “some who attended,” we usually mean a group of a dozen or so. In no case were all participants polled and a consensus, or even plurality, determined. Ideas were often brought up by one person, discussed and elaborated by others, then written or reported by yet others. Hence it is difficult to attribute ideas to particular people in most cases. We thank all those who participated in the workshop and made contributions, large and small, to the ideas noted in this report.

1.3.1 Consider Vulnerabilities in All Parts of the Software Life Cycle

Some who attended the workshop thought that security must be designed into the system from the beginning. It must be a part of the requirements of the system and the architecture of the software. Security often touches and influences many pieces of the system, from low-level details such as how data is stored to high-level details such as a global state recording the user's role or whether the user has been authenticated. When security is added later on, it is typically expensive to develop and test, difficult to use, inadequate, or all three [2].

Another caveat is that security cannot just be designed in, then forgotten. Security should be an integrated part of the entire software development lifecycle. Analysts, coders, testers, integrators, and operators all have vital roles into operating a secure system. Security cannot be relegated to a quality hurdle that the development process needs to surmount then forget. If the software development has a separate group of experts who have been thoroughly trained in cybersecurity and in low-vulnerability software, then developing less vulnerable software should be a partnership between the development team and the experts.

1.3.2 Government Contracting and Procurement, Liability, and Insurance

Many workshop participants felt that the Federal Government could lead a significant improvement in software quality by requiring software quality during contracting and procurement and by changing general expectations.

Participants felt that model contract language can include incentives for software to adhere to higher coding and assurance standards or punitive measures for egregious violations of those standards. The defense community [3], the financial sector, the automotive sector and the medical sector have published sample procurement language for cybersecurity and secure software. The focus on the lowest bidder must include provisions for "fitness for purpose" that factor in considerations for secure software. Only products that fulfill technical acceptance requirements should be considered. Software suppliers who have sloppy cyber hygiene should be identified in contract bidding. All software, especially third-party open source software (OSS), should be evaluated to substantiate that it does not have malware or known or new vulnerabilities, as much as feasible. Software should have a "bill of materials" such that those using it could respond to a new threat made public about some component or library in the software.

Participants generally agreed that new exploits will be discovered after software is put into use, hence the need for a bill of materials. They felt that companies developing software should be contractually liable for vulnerabilities discovered after delivery. Such liability clauses might be modeled after those used in the video game industry in the 2000s. Participants did not believe that there should be legal liability at this time. On the other hand, the language of liability clauses needs to be strict enough to, as one participant wrote, "hold companies accountable for sloppy and easily-avoidable errors, flaws, and mistakes."

One complicating factor is that liability includes a concept of responsible party. Responsibility may be hard to determine in the case of “open source” or freely available software.

The Financial Services Sector Coordinating Council (FSSCC) for Critical Infrastructure Protection and Homeland Security produced a 26-page document entitled *Purchasers’ Guide to Cyber Insurance Products* defining what cyber insurance is, explaining why organizations need it, describing how it can be procured, and giving other helpful information.

Many software assurance tool agreements include “DeWitt clauses” that prohibit the user from publishing any evaluations or comparisons with other tools. Participants felt that such restrictions slow the development of better techniques and make it difficult for users to determine which tool or tools are most beneficial for them. Contract language could specify an allowance of published evaluations, for example as suggested by Klass and Burger in “Vendor Truth Serum”.

1.3.3 Education

Many software developers are not taught the basic principles, practices, and importance of cybersecurity or provided with resources. It was the participants’ judgement that educating a large number of programmers in basic cybersecurity practices will significantly reduce vulnerabilities. The Federal Government could fund broad funding of on-line or self-study courses and work with companies to promote widely-available resources.

In addition, software developers should learn how and when to use powerful and sophisticated tools, which are now available. Participants opined that developers need to understand that they shouldn’t just turn off red flags raised by tools. As above, many institutions of higher education or training organizations can offer free training, once courses are developed.

Some workshop attendees noted that educating just front-line software developers is not enough. Managers and executives must also be educated in the risk management implications of software vulnerabilities and the importance of investing in cybersecurity and low vulnerability software.

1.3.4 Research Projects for Security, Quality, and Few Vulnerabilities

One participant suggested a major project to provide a single forum where researchers could share samples of code, share findings, collaborate on research, and publish results without intellectual property restrictions. A large, open repository of source code would allow researchers to conduct a wide range of data-driven research. Such research could lead to improved programming practices, ways to spot poor quality or malicious code, and new and improved software security metrics and measures. This must be independent of vendors and model and encourage scientifically valid research. The concept would be similar to the Human Genome Project (HGP), but for software instead of genomes. A critical difference is the intellectual property of software.

Participants felt that there needs to be increased scientifically valid research about the strengths and limitations of software assurance tools. Researchers and users could share their findings through a forum such as suggested above. There might even be a list of verified tools.

Another aspect of such a project is to collect incidences of malware, dead code, and other “code smells” so that they are available to researchers. These could augment Common Vulnerabilities and Exposures (CVE). Along the same lines, participants felt that there should be a repository of computer system breaches, like those mandated by the State of California. Such a repository is analogous to those maintained by the Food and Drug Administration (FDA) for medical device problems and the Federal Aviation Administration (FAA) for aircraft incidents.

Attendees noted that today every vulnerability is addressed with its own special, tactical response. Some suggested that there needs to be a substantial research agenda to develop a science of the appearance, detection, behavior, and useful responses of software vulnerabilities instead of treating each vulnerability as a unique problem with its own measure or metric. Given this knowledge, more generalized capabilities to counter classes of vulnerabilities could be developed. Over time a theory of software vulnerability could be developed that provides a larger context for the problem and systemic measures and metrics for detecting and countering classes of vulnerabilities.

1.3.5 Government Funded Efforts

In the participants’ estimation, the Government could fund the research and publication of business cases for secure software, including the cost of security breaches. Such studies or cases would bolster education mentioned in Section 1.3.3.

Workshop attendees suggested that the Federal Government could test or certify the software in widely-used or important modules, libraries, or packages. To partner with the private sector, the Government could fund such testing, perhaps as part of procurement.

Software quality could be improved by following up the Baldrige Cybersecurity Initiative. This may help encourage software companies, according to some participants.

1.3.6 Third Party Review of Software

Some participants felt that software and the software industry should be treated as other industries, like automotive, tobacco, and food. The Government mandated seatbelt use. Could it encourage the software development industry to adopt well-known techniques and practices that industry is reluctant to adopt, because of the belief that such efforts would make them non-competitive? Based on well-established science, the Government could issue directives, create cybersecurity and quality standards, and even mandate compliance. The FDA could enforce standards for medical devices, and the Federal Trade Commission (FTC) could have a role in software apps, since it deals with deceptive advertising [4, 5].

Some participants hoped that the Government would continue to nurture the efforts to add requirements for better security and for lower numbers of vulnerabilities to Federal Information Security Management Act (FISMA) documents, (e.g., NIST Special Publication (SP) SP 800-538, SP 800-64, SP 800-53 and SP 800-53a), and to Department of Defense standards and guidelines.

Participants judged that software could benefit from the programs and criteria of widely-accepted non-governmental organizations. Some possibilities are UL’s Cybersecurity Assurance Program (CAP), Consortium for IT Software Quality (CISQ) Code Quality Standards, and Core Infrastructure Initiative (CII) Best Practices badge.

2. Observations and Recommendations

The focus of the workshop was measures and metrics of software as a product. This section details general observations and suggestions of workshop participants. Some participants cautioned that software quality and security metrics may be the wrong emphasis to reduce software vulnerabilities, that such metrics may fade in emphasis as other software metrics have, for example cohesion and McCabe Cyclomatic Complexity.

2.1 Better Code

Participants praised two workshop presentations: Andrew Walenstein’s “Measuring Software Analyzability” and James Kupsch’s “Dealing with Code that is Opaque to Static Analysis.” Both stressed that code should be amenable to automatic analysis. Both presented approaches to define what it means that code is readily analyzed, why analyzability contributes to reduced vulnerabilities, and how analyzability could be measured and increased.

Some participants noted that there are subsets of programming languages that are designed to be analyzable, such as SPARK, or to be less error-prone, like Less Hatton’s SaferC. Participants generally favored using better languages, for example, functional languages such as F# or ML. However, there was no particular suggestion of *the* language, or languages, of the future.

Attendees also pointed out that while code-based metrics are important, we can expect complementary results from metrics related to the other aspects of the software. Some aspects are the software architecture and design erosion metrics, linguistic aspects of the code, developers’ background, and metrics related to the software requirements.

2.2 More Useful Tool Outputs

There are many powerful and useful software assurance tools available today. No single tool meets all needs. Accordingly, users should use several tools. This is difficult because tools have different output formats and use different terms and classes.

Participants emphasized that tool outputs should be standardized. That is, the more there is common nomenclature, presentation, and detail, the more feasible it is for users to combine tool results with other software assurance information and to choose a combination of tools that is most beneficial for them.

As explained in Section 1.3, participants felt the need for scientifically valid research about tool strengths and limitations, mechanisms to allow publication of third party evaluation of tools, a common forum to share insights about tools, and perhaps even a list of verified or certified tools.

2.3 Security Metrics

Participants didn't note any particular security or vulnerability metrics or measures. However, many participants felt that security or vulnerability measurement (or testing or checking) must be included in *all* phases of software development, as explained in more detail in Section 1.3.1. Except for atypical approaches like Clean Room, this measurement cannot be left as a gate at the end of the production cycle.

2.4 Additional Directions

Some workshop participants were of the opinion that there is a significant need for metrics and measures of binaries or executables. With today's optimizing compilers and with the dependence on many libraries delivered in binary, solely examining source code leaves many avenues for appearance of subtle vulnerabilities.

In the estimation of some attendees, model-based engineering opens the way to writing "source code" at a higher conceptual level and, more importantly, to formal proofs that certain properties are maintained.

2.5 References

- [1] Federal Cybersecurity Research and Development Strategic Plan. Available at https://www.whitehouse.gov/sites/whitehouse.gov/files/documents/2016_Federal_Cybersecurity_Research_and_Development_Strategic_Plan.pdf
- [2] James P. Anderson, "Computer Security Technology Planning Study," October 1972. Available at <http://seclab.cs.ucdavis.edu/projects/history/papers/ande72a.pdf>
- [3] "Suggested Language to Incorporate Software Assurance Department of Defense Contracts," Department of Defense (DoD) Software Assurance (SwA) Community of Practice (CoP) Contract Language Working Group, John R. Marien, chair, and Robert A. Martin, co-chair, February 2016. Available at <http://www.acq.osd.mil/se/docs/2016-02-26-SwA-WorkingPapers.pdf> Accessed 6 September 2016.
- [4] "Mobile Health App Developers: FTC Best Practices," April 2016. Available at <http://www.ftc.gov/tips-advice/business-center/guidance/mobile-health-app-developers-ftc-best-practices>
- [5] Keith Barritt, "3 Lessons: FDA/FTC Enforcement Against Mobile Medical Apps," January 2016. Available at <http://www.meddeviceonline.com/doc/lessons-fda-ftc-enforcement-against-mobile-medical-apps-0001>

3. Position Statements and Presentations

The program committee invited some of those who submitted position statements to make presentations at the workshop. This section allows those who were invited to publish their position in the manner that they choose. In some cases, this is just the position statement. In other cases, it is an extended version of the position statement. In yet other cases it is the possibly-edited presentation given at the workshop or some combination of all of them.

Please note that the following do not necessarily represent the opinion or result of the National Institute of Standards and Technology (NIST). The appearance here does not imply that NIST endorses any of these ideas or products.

The order here is the original order of presentations planned for the workshop.

3.1 Federal Cybersecurity Research and Development Strategic Plan, Greg Shannon, White House Office of Science and Technology Policy.

3.2 Opening Remarks, William F. Guthrie, Chief, Statistical Engineering Division, NIST.

3.3 Measuring Software Analyzability, Andrew Walenstein, BlackBerry.

3.4 Dealing with Code that is Opaque to Static Analysis, James Kupsch, University of Wisconsin-Madison.

3.5 Ken Modeste, UL.

3.6 Composing processes for secure development using process control measures, William Nichols, Software Engineering Institute.

3.7 CISQ Measures of Secure, Resilient Software, Dr. Bill Curtis, Executive Director, Consortium for IT Software Quality (CISQ).

3.8 Mostly Sunny with a Chance of Cyber, David Flater, NIST.

3.9 Dynamical Proving That Security Issues Exist, Dr. Andrew V. Jones, Vector Software.

3.10 Toward Evidence-Based Low Defect Software Production, James Kirby Jr., US Naval Research Laboratory.

3.11 Using Malware Analysis to Reduce Design Weaknesses, Carol Woody, Ph.D., Software Engineering Institute.

3.12 Measure Early and Measure Often – SWAMP, Miron Livny.

3.1 Federal cybersecurity Research and Development Strategic Plan, Greg Shannon, White House Office of science and Technology Policy



Fundamental R&D Challenge in Cybersecurity

Make cybersecurity less onerous
while providing more-effective defenses

Evidence of Efficacy and Efficiency



Federal Cybersecurity R&D Goals

- Near-term, S&T for **effective and efficient risk management**
- Mid-term S&T for **sustainably secure systems development and operation**
- Long-term S&T for **effective and efficient defensive deterrence**



R&D Objectives for Defensive Elements

- Twenty-one objectives to measure progress
- Objectives are not comprehensive
- Two examples
 - Deter, near-term: Establish quantifiable **metrics of adversary level of effort** needed to overcome specific cybersecurity defenses
 - Protect, mid-term: Create tools for static and dynamic analysis that **reduce vulnerabilities by a factor of 10**



Commission on Enhancing National Cybersecurity

Make detailed recommendations to strengthen cybersecurity in both the public and private sectors

- Develop recommendations regarding: (iii) further investments in research and development initiatives that can enhance cybersecurity



Information Available On-line

- **Federal Cybersecurity Research and Development Strategic Plan**
https://www.whitehouse.gov/sites/whitehouse.gov/files/documents/2016_Federal_Cybersecurity_Research_and_Development_Strategic_Plan.pdf
- **National Challenges and Goals for Cybersecurity Science and Technology**
<https://www.whitehouse.gov/blog/2016/02/08/national-challenges-and-goals-cybersecurity-science-and-technology>
- **Cybersecurity National Action Plan**
<https://www.whitehouse.gov/the-press-office/2016/02/09/fact-sheet-cybersecurity-national-action-plan>
- **Commission on Enhancing National Cybersecurity**
<https://www.whitehouse.gov/the-press-office/2016/02/09/executive-order-commission-enhancing-national-cybersecurity>



3.2 Opening Remarks, William F. Guthrie, Chief, statistical Engineering division, NIST



Opening Remarks:
NIST Workshop on Software
Measures and Metrics to
Reduce Security
Vulnerabilities

Will Guthrie, Chief
Statistical Engineering
Division



NIST's Mission & NMI Role

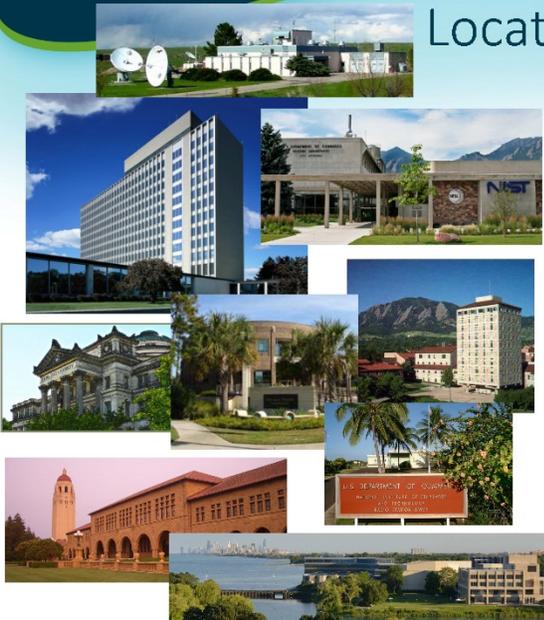
- NIST's mission is to promote U.S. innovation and industrial competitiveness by advancing measurement science, standards, and technology in ways that enhance economic security and improve our quality of life.
- NIST is the national metrology institute (NMI) for the United States. As an NMI, NIST
 - Maintains primary measurement standards for the seven base units in the SI system of units and for derived units
 - Offers calibration services and measurement standards to support international trade
 - Develops new measurement technologies

1

ITL INFORMATION TECHNOLOGY LABORATORY

NIST National Institute of Standards and Technology U.S. Department of Commerce

Locations



- Primary sites
 - Gaithersburg, MD
 - Boulder, CO
- Joint Research Institutes/Centers
 - DC/Boulder areas
 - Charleston, SC
 - Palo Alto, CA
 - Ames, IA
 - Chicago, IL
- WWV and WWVH
 - Fort Collins, CO
 - Kauai, HI

2

ITL INFORMATION TECHNOLOGY LABORATORY

NIST National Institute of Standards and Technology U.S. Department of Commerce

NIST Documentary and Physical Standards



FIPS PUB 202
FEDERAL INFORMATION PROCESSING STANDARDS PUBLICATION
SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions

CATEGORY: COMPUTER SECURITY SUBCATEGORY: CRYPTOGRAPHY

Information Technology Laboratory
National Institute of Standards and Technology
Gaithersburg, MD 20899-8900

This publication is available free of charge from:
<http://dx.doi.org/10.6028/NIST.FIPS.202>

August 2015



U.S. Department of Commerce
Five Point Society
National Institute of Standards and Technology
10101, Under Secretary of Commerce for Standards and Technology and Director

3

The screenshot shows the NIST Information Technology Portal. At the top left is the IITL logo (Information Technology Laboratory) and at the top right is the NIST logo (National Institute of Standards and Technology, U.S. Department of Commerce). The main heading is "IT Standards and Research". Below this is a navigation bar with links for "NIST Time", "NIST Home", "About NIST", "Contact Us", and "A-Z Site Index", along with a search box. The page content includes an "Information Technology Portal - Overview" section with a paragraph about advancing IT state-of-the-art in cyber security and biometrics. Below this are several links for "Cybersecurity Framework", "Cloud Computing", "Computer Security Resource Center", "Information Technology Laboratory", "National Cybersecurity Center of Excellence (NCCoE)", "Smart Grid", and "National Strategy for Trusted Identities in Cyberspace (NSTIC)". To the right, there is a "Select Language" dropdown, a "SHARE" button, and a "Powered by Google Translate" notice. Below that is a featured image of a credit card with a lock icon and a headline: "New NIST Security Standard Can Protect Credit Cards, Health Information". Underneath is a "News And Events" section with two items: "NSCI Seminar: New Technologies for Improved Computer Performance" and "NSCI Seminar: An Overview of High Performance Computing and Benchmark Changes for the Future".

4

The screenshot shows the "Programs and Projects" section of the NIST Information Technology Portal. At the top left is the IITL logo and at the top right is the NIST logo. The main heading is "IT Standards and Research". Below this is a navigation bar with links for "Software Testing Metrics" and "Telecommunications/Wireless". The "Programs and Projects" section lists several projects: "Measurement Science for Complex Information Systems" (aiming to develop and evaluate a coherent set of methods to understand behavior in complex information systems), "Lightweight Cryptography Project" (investigating the need for lightweight cryptographic algorithms), "Video Analytics" (Multimodal Information Group's video analytics program), "Interdisciplinary Projects" (spanning across multiple research areas within the group), "Advanced Video and Signal Based Surveillance" (Second Multiple Camera Single Person Tracking Challenge Evaluation), and "Speaker and Language Recognition Projects" (Speaker and Language Recognition program). To the right is a "Video Playlist" section featuring a video titled "Launch of the National Strategy" with a thumbnail showing a man speaking at a podium with "uschamber.com" on it. Below the video is a "Related Links" section with a link to "Budget in Brief FY 2013 - National Strategy for Trusted Identities in Cyberspace". At the bottom is a "Contact" section with "General Information" (301-975-NIST (6478), inquiries@nist.gov) and the address: "100 Bureau Drive, Stop 1070, Gaithersburg, MD 20899-1070".

5



Challenges in Measuring Software

- Physical quantities
 - regulated by physical laws and environments
 - measurements follow “nice” probability distributions (e.g. Gaussian, Poisson)
- Software
 - largely produced in human imagination, with some mathematical limits
 - measurements not purely deterministic or random
 - probability distributions may not be “nice” (e.g. multimodal, extremely skewed)

6



Need for Reliable Software

- Despite challenges, software has a critical impact on the economy and government
 - electronic commerce
 - identification of business trends/intelligence
 - delivery of health care
 - tracking disease spread
 - management of transportation systems
 - criminal investigations
 - product design
 - projecting climate trends

7

Today's Workshop

NIST Workshop on Software Measures and Metrics to Reduce Security Vulnerabilities

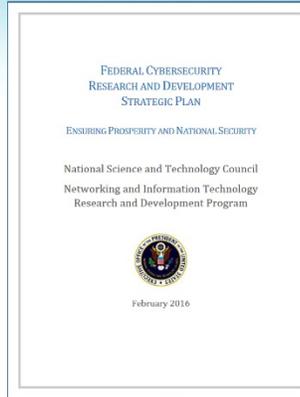
Purpose:

The Federal Cybersecurity Research and Development Strategic Plan seeks to fundamentally alter the dynamics of security, reversing adversaries' asymmetrical advantages. Achieving this reversal is the mid-term goal of the plan, which calls for "sustainably secure systems development and operation." Part of the mid-term (3-7 years) goal is "the design and implementation of software, firmware, and hardware that are highly resistant to malicious cyber activities ..." and reduce the number of vulnerabilities in software by orders of magnitude. Measures of software play an important role.

Industry requires evidence to tell how vulnerable a piece of software is, what techniques are most effective in developing software with far fewer vulnerabilities, determine the best places to deploy countermeasures, or take any of a number of other actions. This evidence comes from measuring. In the broadest sense, or assessing properties of software. With useful metrics, it is straight-forward to determine which software development technologies or methodologies lead to sustainably secure systems.

The goal of this workshop is to gather ideas on how the Federal Government can best use taxpayer money to identify, improve, package, deliver, or boost the use of software measures and metrics to significantly reduce vulnerabilities. We call for position statements from one to three paragraph long. Position statements may be on any subject like the following:

- existing measures of software that can make a difference in three to seven years,
- means of validating software measures or confirming their efficacy (meta-measurements),
- quantities (properties) in software that can be measured,
- standards (in both etalon and norme senses) needed for software measurement,
- cost vs. benefit of software measurements,



- surmountable barriers to adoption of measures and metrics,
- areas or conditions of applicability (or non-applicability) of measures,
- software measurement procedures (esp. automated ones), or
- sources of variability or uncertainty in software metrics or measures.

The output of this workshop and other efforts is a plan for how best the Federal Government can employ taxpayer money to significantly curtail software vulnerabilities in the mid-term.

Measuring Software Analyzability

Andrew Walenstein

Director, Security Research & Development
Center for High Assurance Computing Excellence
BlackBerry

The views and opinions expressed in this position statement are those of the author and do not necessarily reflect the official policy or position of BlackBerry.

The 2016 Federal Cybersecurity Research and Development Strategic Plan never directly defines “sustainability” of secure systems development, but it is easy to read it as meaning “cost-effective.” If so, the critical measure of any advance to secure systems development is the benefit it brings in terms of “bang for buck”. Given the limited scalability and high costs of humans, surely this means automation must be central to any comprehensive effort to advance sustainability of secure systems development. How can we otherwise expect to dramatically drive improvements to the underlying economics? How can we otherwise expect to scale? And if we’re concerns more specifically on *measuring software systems* so as to direct progress towards sustainable security, then it implies that software analysis automation must be a key ingredient.

Indeed, the same federal Strategic Plan alludes to such automation and, in the case of formal methods, it says “the applicability of these techniques is currently limited to modest programs with tens of thousands of lines of code. Improvements in efficacy and efficiency may make it possible to apply formal methods to systems of practical complexity.” That message is clear enough: our automation cannot scale to our *code*, so the *automation* must be improved. However, we can also profitably view it the other way around: given the *automated capabilities* we currently have, the *code* must be improved. This is not a new proposition—to pick just one example, Gerard Holtzmann’s “Power of 10: Rules for Developing Safety Critical Code” lists rules for software construction aimed at making its analysis (by human or computer) easier. But how can we measure “software security-analyzability”?

There are some theoretical and empirical techniques available to draw upon from the software obfuscation literature. Upon a little reflection it probably makes perfect sense why we should find it in that literature. At BlackBerry we are just starting to explore a related notion and measure of modularity supporting bounded model checking for security properties. For all the community’s efforts on measuring our analysis tools—NIST’s own SAMATE workshops are a fantastic example—there seem to be less emphasis on building measures of software analyzability. If we are aiming for sustainability through automation, though, how heavily should we be betting that we can *scale software analysis* more quickly than we can *improve development techniques* to yield more-analyzable software? And yet it’s not even a race; rather, it seems prudent to try to get improvements in analysis and in software to meet in the middle. So as a community let us make sure we adequately explore measures for “software security-analyzability”.



NIST WORKSHOP ON SOFTWARE MEASURES AND METRICS TO REDUCE SECURITY VULNERABILITIES

MEASURING SOFTWARE ANALYZABILITY

ANDREW WALENSTEIN

CENTER FOR HIGH ASSURANCE COMPUTER EXCELLENCE

July 12, 2016
Gaithersburg, MD

The views and opinions expressed in this presentation are those of the author and do not necessarily reflect the official policy or position of BlackBerry.



POSITION

**WE NEED TO BETTER MEASURE THE
ANALYZABILITY OF SOFTWARE**

BECAUSE

WE NEED TO MEASURE THE SECURITY OF SOFTWARE BETTER.



MOTIVATION AT BLACKBERRY

- Center for High Assurance Computing Excellence
 - Security assurance research (collaborative)
 - Have been exploring CBMC (with Oxford University)
 - CBMC = bounded model checker
 - Turns checks into Boolean satisfiability problem
 - Read code → generate SAT formula → search for solution
- Can be applied to find vulns due to integer overflow

CBMC
Download

 BLACKBERRY

2

MODEL CHECKING FOR INTEGER OVERFLOWS

```
char* stagefrt( char* buffer, unsigned int count, unsigned int size)
{
    unsigned int i;
    unsigned int alloc_size = size * count;

    char* copy = malloc( alloc_size );

    for( i=0 ; i<count ; ++i )
        strncpy( copy + i*size, buffer + i*size, size );
    return copy;
}
```

 BLACKBERRY

3

CHECKABLE – ANALYZABLE

```
void calls() {  
    char buffer[1024];  
    unsigned int over = UINT_MAX/2 + 1;  
  
    stagefrt( buffer, 2, 2 ); ← Verifies successfully using model checker  
    stagefrt( buffer, 2, over ); ← Finds overflow  
}
```



4

STILL ANALYZABLE

[incl.h](#)

```
#define lt void  
#define lk strcpy  
#define li const  
#define lc char  
#define ld unsigned  
#define la int  
#define ll stagefrt  
#define lm malloc  
#define lu for  
#define lq return  
#define lo main  
#define lp UINT_MAX
```

[main.c](#)

```
#include "incl.h"  
  
lc*ll(lc*lf,ld la lg,ld la le  
) {ld la lb;ld la  
lj=le*lg;lc*lh=lm(lj);lu(lb=0;l  
b<lg;++lb)lk(lh+lb*le  
,lf+lb*le,le);lq lh;}la lo(){lc  
lf[1024];ll(lf,2,lp/2+1);}
```



5

```
void stagefrt2( char* buffer, unsigned int count, unsigned int size) {
    unsigned int i;
    unsigned int alloc_size;
    if ( count < size && ( size > 12 || count < 32 ) ) {
        if ( size > 32 ) {
            if ( count < 3 )
                alloc_size = count * size; }
        else {
            alloc_size = size;
            count = 1; }}
    else if ( size > 1024 || (count < 42 && size > 2 ) ) {
        alloc_size = size;
        count = 1; }
    else {
        alloc_size = size;
        count = 1; }

    char* copy = malloc( alloc_size );
    for ( i=0 ; i<count ; ++i )
        strncpy( copy + i*size, buffer + i*size, size );
}
```



ACTUAL PROBLEMS FOR BMC

```
void calls() {
    extern unsigned int unstacked;
    char buffer[1024];
    unsigned int over = UINT_MAX/2 + 1;

    stagefrt( buffer, 2, unstacked ); ← FP?

    stagefrt( buffer, 2, encrypt(msg,pw)==res ? 2 : over );
}
↑ Hard...
```



MEASURES/METRICS APPROACH

“We can’t hope to raise the cybersecurity bar if we don’t know how to measure its height”

David Kleidermacher, CSO BlackBerry

Theory / approach

- Measuring drives improvement and investment – objective function
- What kind of improvement do we expect?



8

GOALS: HEIGHT OF THE BAR

Economically

- “sustainably secure systems development and operation” – economic viability question, *not* feasibility

Fantastically

- “reduce the number of vulnerabilities in software by **orders of magnitude**”

Urgently:

- A **3-7 year** goal



9

GOALS → AUTOMATION

Automation is the key

- We want *sustainability*
 - *How can costly humans be the answer?*
- We seek *orders of magnitude* improvement
 - *How can we do this without mobilizing orders of magnitude better automation?*

Security assurance automation

- Assurance = level of confidence that software functions as intended and is free from vulnerabilities (Mitre)
- Focus: checking security properties – the root of all **confidence**



10

TOOL LIMITATIONS

On formal methods:

“the applicability of these techniques is currently limited to modest programs with tens-of-thousands of lines of code. Improvements in efficacy and efficiency may make it possible to apply formal methods to systems of practical complexity”

[2016](#) Federal Cybersecurity R&D Strategic Plan



On static analysis coverage:

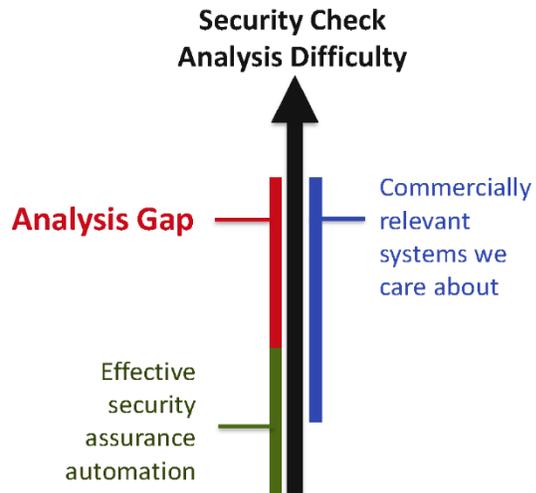
“Static tools only see code they can follow, which is why modern frameworks are so difficult for them. Libraries and third-party components are too big to analyze statically, which results in numerous ‘lost sources’ and ‘lost sinks’ – toolspeak for “we have no idea what happened inside this library.” Static tools also silently quit analyzing when things get too complicated.”

[Jeff Williams](#): Why It’s Insane to Trust Static Analysis

11

ANALYSIS GAP

- Gap between what we can automatically check and what we need to
- We need to reduce that gap
- Common focus: height of green -- need better checking tools
- But what about asking if we can make software more analyzable?



BLACKBERRY

12

MAKING SOFTWARE ANALYZABLE

Gerard Holzman proposed 10 coding guidelines for safety critical code.

When it really counts, ..., it may be worth going the extra mile and living within stricter limits.. In return, we should be able to demonstrate more convincingly that critical software will work as intended

Example Rule

Restrict all code to very simple control flow constructs – do not use goto statements, ... and direct or indirect recursion.

Rationale

Simpler control flow translates into stronger capabilities for verification...Without recursion, ..., we are guaranteed to have an acyclic function call graph, which can be exploited by code analyzers, and can directly help to prove that all executions that should be bounded are in fact bounded.

Gerard J. Holzman – NASA/JPL – The Power of 10: [Rules for Developing Safety Critical Code](#)

BLACKBERRY

13

APPROACH 1: HEURISTIC METRICS

Recipe

1. Identify properties of code that make it hard to analyze by “typical” analyzers
2. Define metrics that relate to those code properties

Example

- Holzmann: *do not use goto statements, ..., and direct or indirect recursion*
- Measure: based on observing gotos, direct and indirect recursion

Tradeoffs

- Easy to generate; might be pretty tool-independent
- No theory to guide and assess – we don’t want under- **or** over-restrictive

 BLACKBERRY

14

APPROACH 2: EMPIRICALLY MEASURE

Recipe

1. For a given tool, identify ways in which analysis is weakened by code
2. Modify analysis tools to provide measures of analyzability

Example

- Tool can find data exfiltration tracking taint through some pointers, but not all
- Modify tool to output measures relating to its success in following the taint

Tradeoffs

- Should be possible for many (all?) tools
- How usable / actionable are the reports?

 BLACKBERRY

15

APPROACH 3: NEW SOFTWARE METRICS?

- Problem: code not modularized well for the purposes of checking using CBMC
 - Difficult to set up a small checking “environment” or calling context
 - End up writing complex “drivers” and “stubs” and even modify the code
 - Notorious problem in model checking community
- Essentially a modularity problem – the wrong modularity?
 - The code might be considered nicely modular in terms of “ordinary” modularity metrics
 - But from the point of analyzing the code with CBMC, it was a tangled mess
 - *Is it possible to define new modularity metrics that ease CBMC-analyzability?*
- Not yet sure – ongoing research



16

APPROACH 4: ADAPT OBFUSCATION THEORY?

Theory

- Obfuscation = transformations that make analyzers break
- Making software more analyzable = deobfuscation
- Approach: define metrics using available theories of obfuscation potency

Example

- Giacobazzi and Dalla Preda use Abstract Interpretation to define obfuscation in terms of transformations that make analyzers *incomplete*
 - Yields a theoretical model for defining potency and comparing potency
 - Can we use this approach to define metrics on code?

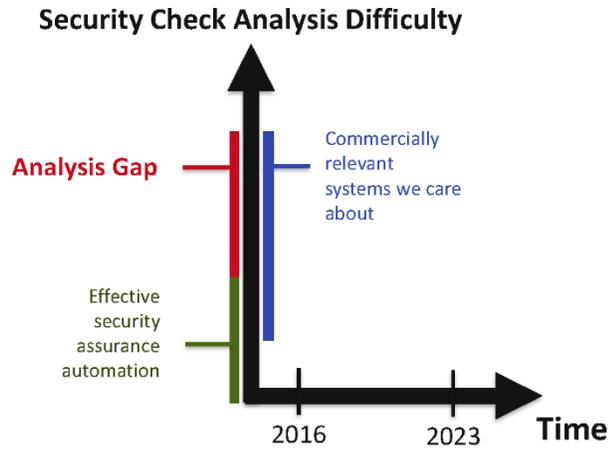


17

ANALYSIS GAP AND THE FUTURE

- What does the future hold for automation of security analysis?
- Where should we place our bets for making **orders of magnitude** improvement?

Imaging charting the green and blue peaks into the future....

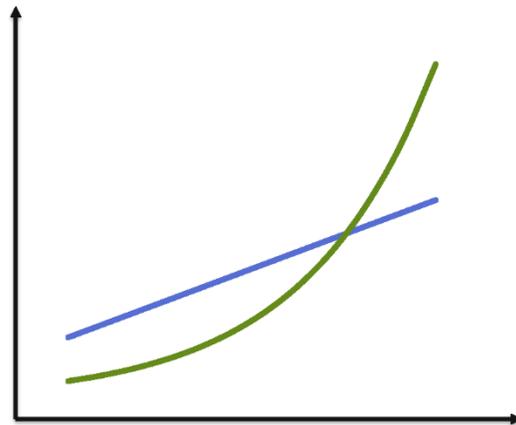


 BLACKBERRY

18

AUTOMATION REVOLUTION (CLOSED UNIVERSE)

- Fantastic improvement in automated security assurance (*Henry Gordon Rice is astounded*)
- Catch up to and surpass current needs
- Effective elimination of classes of vulnerabilities

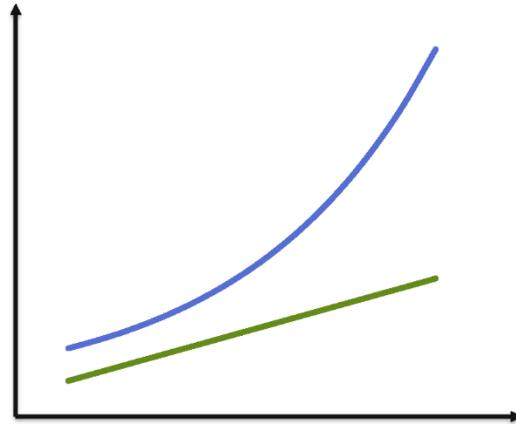


 BLACKBERRY

19

REGRESSION (OPEN UNIVERSE)

- Analysis loses ground
- Not promising for the future

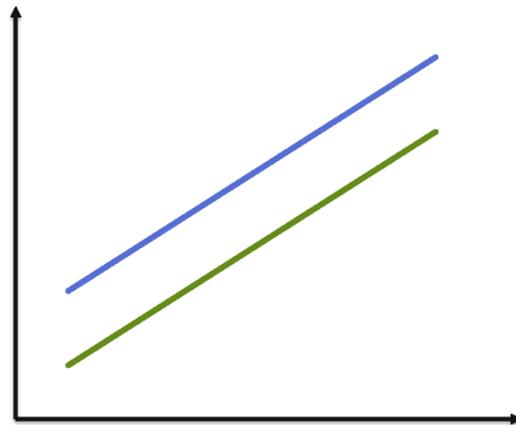


 BLACKBERRY

20

STASIS (FLAT UNIVERSE)

- Analyzability of software rises at same rate as our tool abilities
- Huge improvement possible for some systems
- But gap is same = same lack of assurance...have we succeeded on our goals?



 BLACKBERRY

21

BETS?

Sustainable orders of magnitude increase in security in 5-7 years

- *Where is your bet how it will most likely come to pass?*

A. Non-automation

- Humans, processes, standards, ...

B. Improvements in automation

- Improved & cheaper formal methods, program analysis, test generation...

C. Improvements in software analyzability

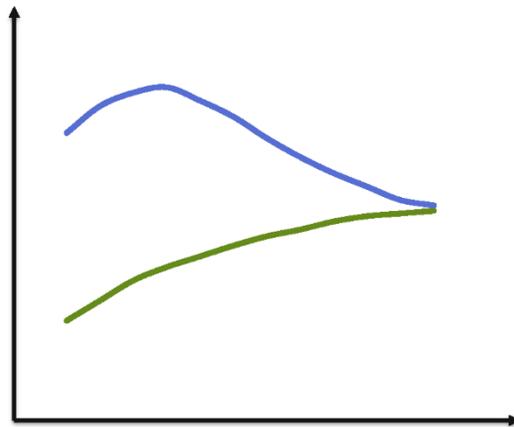
- Processes and tools that generate more analyzable code



23

RENDEZVOUS MODEL

- Automation improves slowly
- Analyzability is measured and slowly improves analyzability of code



22

POSSIBLE STEPS FORWARD?

1. Defining new measures, metrics
 - We can start defining as best we can and measure their utility.
2. Modifying tools to support analyzability improvements
 - Reporting loss of completeness/precision – and highlight problem code features?
 - Automated de-obfuscators?
3. Language / framework design
 - Can we make analyzability a key design feature?
4. Process change
 - Analyzability as a quality?
 - Analyzability gap as a type of maintenance debt?



3.4 Dealing with Code that is Opaque to Static Analysis

Barton P. Miller†‡ James A. Kupsch†‡ Vamshi Basupalli†‡ Elisa Heymann†*

†Computer Sciences Department, University of Wisconsin

‡DHS Software Assurance Marketplace (SWAMP)

*Autonomous University of Barcelona

Critical to producing secure software is the need to assess that software for weaknesses (and ultimately, vulnerabilities). A key part of such an assessment is the use of static analysis tools for scanning that software for weaknesses. Especially for legacy languages such as C and C++, *these tools are often confounded by constructs that are too complex to analyze, leaving significant blind spots in programs written in these languages.* Notable recent exploits, such as Heartbleed and the glibc DNS vulnerabilities, are examples where arcane (but not rare) coding practices prevented even the best of existing tools from finding the weaknesses that allowed the exploits. Current tools do not (and many cannot) distinguish between not finding a weakness because it is not present and not finding it because it is too difficult to find. The legacy programming languages will be with us for a long time, because many of our current major operating systems, systems software, server platforms, and even applications are written in these languages.

While there are many commonly used code metrics, our experiments have shown that these metrics do not strongly correlate with characteristics of a program that make it opaque to static analysis. Common metrics measure deal with shallow syntactic features including simple counts such as number of lines of code, comments, methods or fields; complexity measures of functions or methods computed from the simple counts such as cyclomatic complexity or Halstead complexity; and measures of relationships of properties of a function or class to others computed from simple counts such as cohesion or coupling. These measures do not capture the necessary characteristics of a program that would indicate which parts of a program can be reasonably analyzed.

We outline a research program to produce a new class of metrics, called *semantic opaqueness metrics*, that identify the parts of a program for which a static analysis tool cannot come to a firm (or perhaps even sound) conclusion that the code is demonstrably safe or unsafe. These metrics would be based on a deeper semantic analysis of the program, using state-of-the-art control and dataflow techniques. Such metrics would give guidance to the programmer so that they can transform the opaque parts of the code, simplify the code structure, such that static analysis tools can make definitive statements about the code. These metrics also will allow a more accurate scoring of programs according to their resistance to analysis (and therefore likely to be hiding critical weaknesses). A rapid path forward, allowing substantial progress in the 3-7 year timeframe is to base this work on a powerful open source compiler framework such as clang/LLVM or gcc. The broader impact of such an approach is that we can evolve our legacy code base, and new software written in legacy languages, such that we have effective means to assess this code.

Dealing with Code That Is Opaque to Static Analysis

Barton P. Miller^{†‡}, James A. Kupsch^{†‡},
Elisa Heymann^{†*}, Vamshi Basupalli^{†‡}

NIST Workshop on Software Measures and Metrics
to Reduce Security Vulnerabilities

Gaithersburg, MD

July 12, 2016

[†]Computer Sciences Department, University of Wisconsin
[‡]DHS Software Assurance Marketplace (SWAMP)
^{*}Autonomous University of Barcelona



Recent Experience

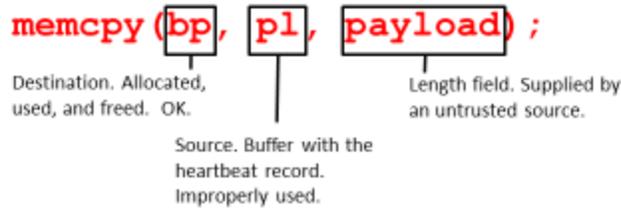
- Reviewed high profile vulnerabilities
 - **Heartbleed** (CVE-2014-010)
 - **glibc DNS resolver** (CVE-2015-7547)
- Obtained vulnerable source code
- Ran static code analysis tools on each
- Tools **failed to find the bugs**
- Bug was **opaque** to the tools



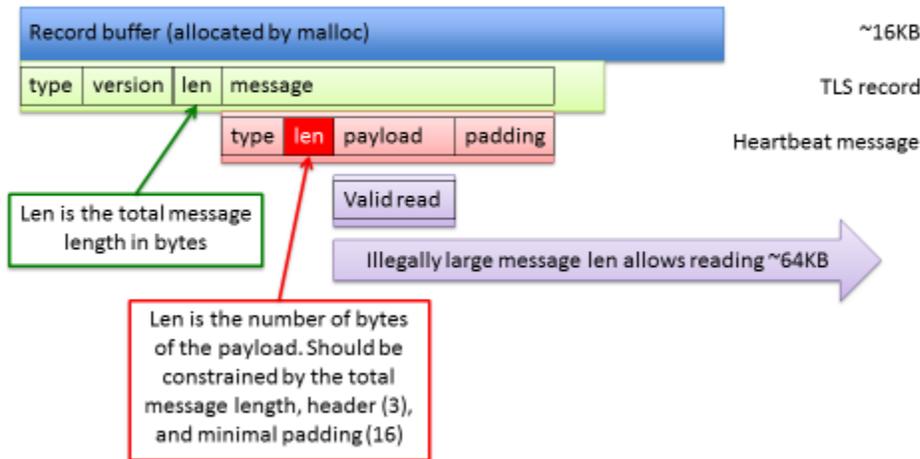
Heartbleed

At it's heart (sorry), it's just a buffer overflow...

- Failure of the OpenSSL library to validate the heartbeat packet length field (as compared to the size of the actual message).
- Heartbeat packets are contained within TLS packets.
- The heartbeat protocol is supposed to echo back the data sent in the request where the amount is given by the payload length.
- Since the length field is not checked, **memcpy** can read up to 64KB of memory.



TLS Heartbeat Protocol



Heartbleed



Added length check to remediate:

```
if (1+2+payload+16 > s->s3->rrec.length)
    return 0 // silently discard
```

And none of the current tools could find the problem...why?



Here's the offending code, slightly redacted

```
2556 unsigned char *p = &s->s3->rrec.data[0], *pl;

2563 n2s(p, payload);
2564 pl = p;
2565
2566 if (s->msg_callback)
2567     s->msg_callback(0, s->version, TLS1_RT_HEARTBEAT,
2568                   &s->s3->rrec.data[0], s->s3->rrec.length,
2569                   s, s->msg_callback_arg);
2570
2571 if (hbtype == TLS1_HB_REQUEST) {
2573     unsigned char *buffer, *bp;
2574     int r;

2580     buffer = OPENSSL_malloc(1+2+payload+padding);
2581     bp = buffer;

2584     *bp++ = TLS1_HB_RESPONSE;
2585     s2n(payload, bp);
2586     memcpy(bp, pl, payload);
```



Here's the offending code, slightly redacted

```
2556 unsigned char *p = &s->s3->rrec.data[0], *pl;

2563 n2s(p, payload);
2564 pl = p;
2565
2566 if (s->msg_callback)
2567     s->msg_callback(0, s->version, TLS1_RT_HEARTBEAT,
2568                   &s->s3->rrec.data[0], s->s3->rrec.length,
2569                   s, s->msg_callback_arg);
2570
2571 if (hbtype == TLS1_HB_REQUEST) {
2573     unsigned char *buffer, *bp;
2574     int r;

2580     buffer = OPENSSL_malloc(1+2+payload+padding);
2581     bp = buffer;

2584     *bp++ = TLS1_HB_RESPONSE;
2585     s2n(payload, bp);
2586     memcpy(bp, pl, payload);
```

1. Find the heartbeat packet in the (untrusted) user request



Here's the offending code, slightly redacted

```
2556 unsigned char *p = &s->s3->rrec.data[0], *pl;

2563 n2s(p, payload);
2564 pl = p;
2565
2566 if (s->msg_callback)
2567     s->msg_callback(0, s->version, TLS1_RT_HEARTBEAT,
2568                   &s->s3->rrec.data[0], s->s3->rrec.length,
2569                   s, s->msg_callback_arg);
2570
2571 if (hbtype == TLS1_HB_REQUEST) {
2573     unsigned char *buffer, *bp;
2574     int r;

2580     buffer = OPENSSL_malloc(1+2+payload+padding);
2581     bp = buffer;

2584     *bp++ = TLS1_HB_RESPONSE;
2585     s2n(payload, bp);
2586     memcpy(bp, pl, payload);
```

2. Extract user-stated payload length of the heartbeat packet



Here's the offending code, slightly redacted

```
2556 unsigned char *p = &s->s3->rrec.data[0], *pl;

2563 n2s(p, payload);
2564 pl = p;
2565
2566 if (s->msg_callback)
2567     s->msg_callback(0, s->version, TLS1_RT_HEARTBEAT,
2568                    &s->s3->rrec.data[0], s->s3->rrec.length,
2569                    s, s->msg_callback_arg);
2570
2571 if (hbtype == TLS1_HB_REQUEST) {
2573     unsigned char *buffer, *bp;
2574     int r;

2580     buffer = OPENSSL_malloc(1+2+payload+padding);
2581     bp = buffer;

2584     *bp++ = TLS1_HB_RESPONSE;
2585     s2n(payload, bp);
2586     memcpy(bp, pl, payload);
```

3. `pl` is an *alias* to the heartbeat payload start address.



Here's the offending code, slightly redacted

```
2556 unsigned char *p = &s->s3->rrec.data[0], *pl;

2563 n2s(p, payload);
2564 pl = p;
2565
2566 if (s->msg_callback)
2567     s->msg_callback(0, s->version, TLS1_RT_HEARTBEAT,
2568                    &s->s3->rrec.data[0], s->s3->rrec.length,
2569                    s, s->msg_callback_arg);
2570
2571 if (hbtype == TLS1_HB_REQUEST) {
2573     unsigned char *buffer, *bp;
2574     int r;

2580     buffer = OPENSSL_malloc(1+2+payload+padding);
2581     bp = buffer;

2584     *bp++ = TLS1_HB_RESPONSE;
2585     s2n(payload, bp);
2586     memcpy(bp, pl, payload);
```

4. Length of TLS packet that contains heartbeat packet



Here's the offending code, slightly redacted

```

2556 unsigned char *p = &s->s3->rrec.data[0], *pl;

2563 n2s(p, payload);
2564 pl = p;
2565
2566 if (s->msg_callback)
2567     s->msg_callback(0, s->version, TLS1_RT_HEARTBEAT,
2568                   &s->s3->rrec.data[0], s->s3->rrec.length,
2569                   s, s->msg_callback_arg);
2570
2571 if (hbtype == TLS1_HB_REQUEST) {
2573     unsigned char *buffer, *bp;
2574     int r;

2580     buffer = OPENSSL_malloc(1+2+payload+padding);
2581     bp = buffer;

2584     *bp++ = TLS1_HB_RESPONSE;
2585     s2n(payload, bp);
2586     memcpy(bp, pl, payload);

```

5. payload length **should be** ≤ TLS record length-19



Here's the offending code, slightly redacted

```

2556 unsigned char *p = &s->s3->rrec.data[0], *pl;

2563 n2s(p, payload);
2564 pl = p;
2565
2566 if (s->msg_callback)
2567     s->msg_callback(0, s->version, TLS1_RT_HEARTBEAT,
2568                   &s->s3->rrec.data[0], s->s3->rrec.length,
2569                   s, s->msg_callback_arg);
2570
2571 if (hbtype == TLS1_HB_REQUEST) {
2573     unsigned char *buffer, *bp;
2574     int r;

2580     buffer = OPENSSL_malloc(1+2+payload+padding);
2581     bp = buffer;

2584     *bp++ = TLS1_HB_RESPONSE;
2585     s2n(payload, bp);
2586     memcpy(bp, pl, payload);

```

6. allocate enough memory for echo packet (according user payload)



Here's the offending code, slightly redacted

```
2556 unsigned char *p = &s->s3->rrec.data[0], *pl;

2563 n2s(p, payload);
2564 pl = p;
2565
2566 if (s->msg_callback)
2567     s->msg_callback(0, s->version, TLS1_RT_HEARTBEAT,
2568                   &s->s3->rrec.data[0], s->s3->rrec.length,
2569                   s, s->msg_callback_arg);
2570
2571 if (hbtype == TLS1_HB_REQUEST) {
2573     unsigned char *buffer, *bp;
2574     int r;

2580     buffer = OPENSSL_malloc(1+2+payload+padding);
2581     bp = buffer;

2584     *bp++ = TLS1_HB_RESPONSE;
2585     s2n(payload, bp);
2586     memcpy(bp, pl, payload);
```

7. Copy heartbeat data based on the length they claimed. Can also grab other nearby data.



Here's the offending code, slightly redacted

```
2556 unsigned char *p = &s->s3->rrec.data[0], *pl;

2563 n2s(p, payload);
2564 pl = p;
2565
2566 if (s->msg_callback)
2567     s->msg_callback(0, s->version, TLS1_RT_HEARTBEAT,
2568                   &s->s3->rrec.data[0], s->s3->rrec.length,
2569                   s, s->msg_callback_arg);
2570
2571 if (hbtype == TLS1_HB_REQUEST) {
2573     unsigned char *buffer, *bp;
2574     int r;

2580     buffer = OPENSSL_malloc(1+2+payload+padding);
2581     bp = buffer;

2584     *bp++ = TLS1_HB_RESPONSE;
2585     s2n(payload, bp);
2586     memcpy(bp, pl, payload);
```

Need to actually know that **payload** length is not trusted (tainted) data.



Heartbleed

Conceptually, this is just an exercise in taint analysis. We need to following the original enclosing TLS packet from a socket, marking it as tainted.

Before disclosure:

- No tools we tried found the bug
- No tools we know of found the bug

Coverity “fixed” their tool by noting that extracting the integer payload length from a network byte-order uses a **byte-swap** instruction on a little endian machine, and such a swap instruction is rare enough that this is a sign that the data comes from the network.

GrammaTech could do the taint analysis starting at socket buffers, but didn’t do it because it was too slow in practice. When they turned it on *for the right section of code*, it found the problem.



Difficulties for SCA Tools

- Legacy languages inherent features
 - Raw memory access
 - Lack of type safety
 - Manual resource management
 - Pointers and pointer arithmetic
- Code complexity
 - Indirection
 - Large program state
 - Complex control flow



Why SCA Tool Fail to Report

- Not deducing accurate set of values or properties (tainted, initialized, not null, ...) for variables
- Not deducing correlation between variables
- Using heuristics to determine likely values or properties
- Uncertain results not reported to reduce false positives
- Confidence score may point to opaque code, if there is a report
- For non-reports, no way to convey confidence



17



Dynamic Analysis Tools

- We do not know of any dynamic analysis tools that found these vulnerabilities
- Difficulties:
 - Generating correct bad input sequence
 - Input data space is large
 - Input data sequence is complex



18



Goal: Less Opaque Code for SCA

- Two approaches
 - New code:
 - Use modern languages to prevent some defects
 - D, Rust, modern C++
 - Use (more) analyzable subset of language
 - MISRA
 - Checked C
 - C++ Core Guidelines, GSL (guideline support library), and SCA
 - Legacy code (and to a lesser extent new code):
 - Identify parts that are opaque
 - **Current metrics do not identify opaque code**



19

Common Metrics

- Metric Types
 - Simple counts:
 - Lexical elements: lines of code, comments, ...
 - Syntactic elements: parameters, types, operators, ...
 - Per function, file, or code base
 - Calculated metrics:
 - Examples: Cyclomatic, Halsted
 - Per function
 - Relationships between functions, classes, ...
 - Examples: Coupling, Cohesion, Connascence
 - Per pair of functions, classes, ...
- In our experience, these **metrics did not correlate with weaknesses or static analyzability**
- Focus: cost to develop, maintain, test, enhance, ...



20

Proposal: Opaqueness Metric

- Develop tools that identify program complexity in terms of opaqueness to analyzability by SCA tools
 - Semantic complexity of code that reaches a tool's ability to report due to reaching limits of the analysis algorithm's
 - Decidability
 - Implementation
 - Score regions of the source code with an opaqueness score
 - Also include rationale for poorly scoring regions
- Provide prescriptive advice to transform the code to be less opaque to SCA (more easily and correctly analyzable)



21



SCA Tool Providers Path Forward

- Best semantic code analysis is in commercial tools
- SCA tools already have much of the information
 - Know where assumptions are made
 - Location of assumptions are accurate
 - Should be accurate for users of the tool
- Limitations
 - Inherently not in their interest, reporting limitations is bad for marketing
 - Specific to the types of problems the tool finds and the power of the tool



22



Broader Path Forward: Develop Tool

- Start with existing open source analysis framework
 - Clang Static Analyzer
 - Gcc
- Fund open source tool based on framework to score the source code based on its opaqueness to static analysis
- Develop prescriptive guidance on transforming source to make code less opaque



Questions



3.5 UL's Position Statement:

UL believes that reducing software vulnerabilities will take considerable time, and requires a paradigm shift in the way how organizations develop software and firmware products, and how it can be measured. There are several ways to address software vulnerabilities. Reducing software vulnerabilities requires at a minimum 3 measures. The following summarizes the intent UL is proposing to both its customers and the industries involved:

- 1) Develop a scientific methodology to assess software and provide metrics tied to how software vulnerabilities can be identified and measured and means to address them. This methodology must be in stages. It must look at some of the fundamental software and environmental weaknesses that contribute to software weaknesses becoming vulnerabilities and providing a path for industry to address. To engage this in a methodical manner, it must be done in stages with some of the foundational problems to be addressed in the first phase, working with industry to adopt, and then supplementing those problems with newer more complex problems in future phases. Industry acceptance via adoption and not by mandate is key to resolve.
- 2) Provide an independent third party means to assess industry's ability to meet the requirements defined in phase 1. These should be driven by the procurement means of the government and not by mandate. The procurement means can provide the incentive to drive industry adoption.
- 3) Provide additional requirements to support industry vendors in learning how to build security into software to reduce the vulnerabilities that can arise. Helping industry build better software reduces the case for vulnerabilities.

UL has developed several foundational standards under the UL 2900 series under the UL cybersecurity assurance program that can support the initiatives above. This method can promote industry to develop better software and encourage purchasers of software to require better software

Ken Modeste
Connected Technologies
UL LLC
Commercial & Industrial(C&I)
333 Pfingsten Road, Northbrook, IL 60062-2096 U.S.A
Phone: 847-664-2659
Cell: 847-682-9703
Ken.modeste@ul.com

3.6 William R. Nichols, Software Engineering Institute

William R. Nichols, Software Engineering Institute

A recent update to a major mobile OS included security patches to address violations of least privilege, buffer overflows and multiple cases of memory corruption, all of which were common security problems as long ago as the 1990s. It's striking that after all these years, common implementation defects remain a major source of software security vulnerabilities, including such well known examples as OpenSSL Heartbleed and Apple "goto fail." Research literature shows that software defect rates contribute to these vulnerabilities, providing evidence that 1% to 3% of all released defects in the Windows and Linux operating systems were potentially exploitable. Based on this, we at the SEI believed that 1) quality attributes such as security are undermined by defects, 2) defective software cannot be secure, and 3) we can estimate the number of vulnerabilities if we know the overall defect level. To test this, we examined a small number of industry software products that have very low levels of defects. In these products, we found proportionally lower levels of vulnerabilities or safety critical issues. What the products had in common was a robust measurement framework supporting early and effective defect removal; this framework allowed the system developers to manage the quality processes during implementation. We therefore concluded that improving the quality in implementation is something that can be and is being done right now to reduce security vulnerabilities.

While the current quality improvements are a step in the right direction, they're only one part of making secure software. The software development lifecycle can incorporate many tools that can statically analyze source code, statically and dynamically analyze executables, examine for code coverage, scan web services, implement numerous testing techniques, and so forth. We also have assurance cases, architecture analysis and design language, architectural tradeoff methods, and design and code inspections. But how do these processes, practices, and tools contribute to creating an affordable, secure development process that can be implemented successfully?

Today, the real world costs and benefits of these efforts are largely a matter of expert opinion, which is helpful but insufficient. To be successful, the development process must be both effective and efficient, based on validated measures, so that we can determine not only how to compose the process but also how successful the process is. Since no tool or technique is perfect, real systems will be composed of numerous techniques for requirements, analysis, design, and implementation. We need to understand both the benefits and costs of these techniques as they fit into a comprehensive and coherent framework that includes measures of the product and the costs to produce it. To achieve the next level of security, our processes must meet measures of both effectiveness and cost; studies are needed to establish economic benchmark data for these measures, based on real world development.

3.7 Assessing the Cybersecurity of Federal Source Code with CISQ Measures

Bill Curtis, Executive Director, Consortium for IT Software Quality (CISQ)

Abstract:

Recent breaches compromising the confidentiality of Federal records accentuate the need for structural analysis of cybersecurity weaknesses in the source code of Federal systems. Advances in static analysis technology enable detection of a wide range of source code weaknesses that can be exploited to gain unauthorized entry. The Consortium for IT Software Quality (CISQ) is chartered by its sponsors to create standards for automating measures of software size and quality. CISQ standards have recently been approved by the Object Management Group for Automated Function Points, Reliability, Security, Performance Efficiency, and Maintainability. The latter four quality measures are based on definitions of these characteristics in ISO 25010, and provide source code level measures that supplement the largely behavioral measures in ISO 25023.

CISQ's Security measure is calculated from assessing 22 of the Top 25 weaknesses in the Common Weakness Enumeration repository (i.e., CWE/SANS Institute Top 25, OWASP Top 10) that can be detected through static analysis. These weaknesses include well-known culprits such as SQL injection, buffer overflows, and cross-site scripting. This measure provides an accurate indicator of the likelihood an attacker can find an exploitable weakness in a Federal application. Both the Software Engineering Institute and CAST have recently found that weaknesses causing reliability problems can in some cases be exploited for unauthorized entry, indicated that security is bound to other aspects of software quality. Since poor quality code is also insecure code, the overall structural integrity of Federal source code should be assessed to insure cybersecurity. Recent analysis results from government systems will be presented.

The continuing stream of breaches exploiting SQL injection, a weakness known since the late 1990s, indicate that both commercial and government IT departments are not doing enough to reduce the vulnerability of their applications. Based on recent embarrassing breaches, Federal IT needs a major undertaking similar to the Y2K endeavor to rid Federal source code of the most easily exploited weaknesses. Federal executives need to 1) assess the cybersecurity risk of agency systems using the CISQ standards along with other measures, 2) enforce remedial actions based on measurement results, and 3) develop policies to strengthen the cybersecurity of software in Federal agencies and in industries they regulate.

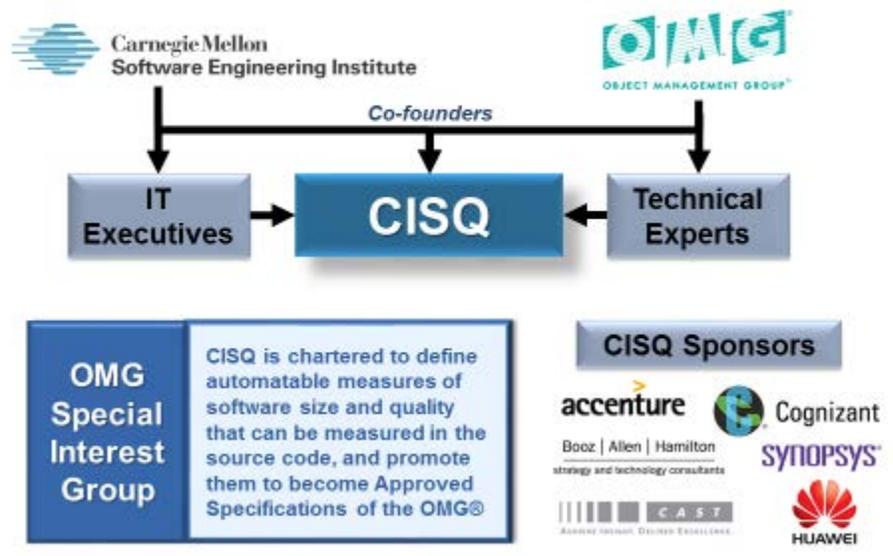


CISQ Measures of Secure, Resilient Software

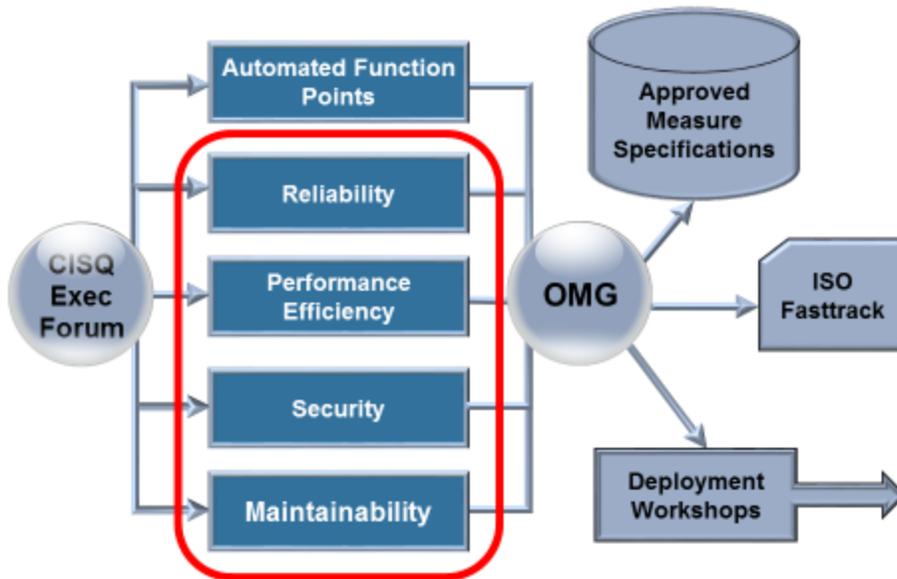
OMG Standards for Software Measurement

Dr. Bill Curtis
Executive Director, CISQ

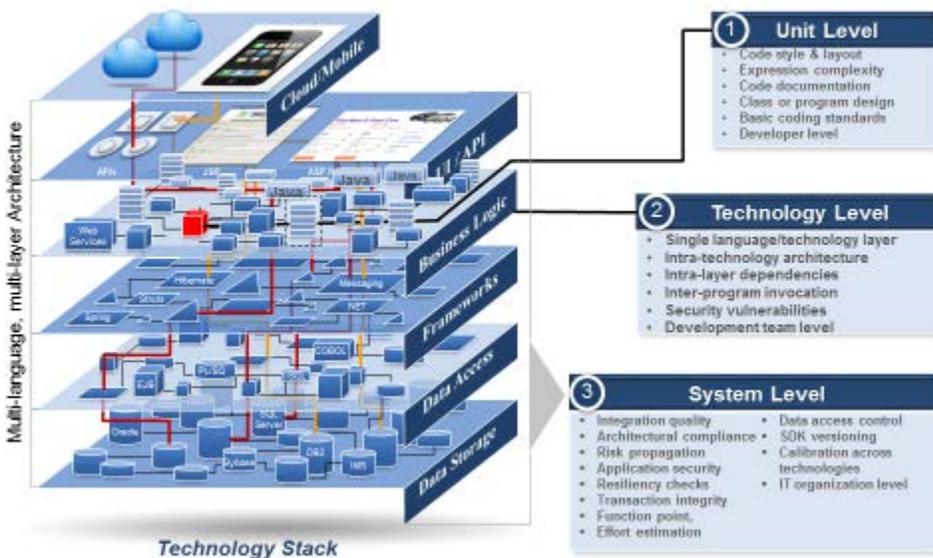
CISQ What is CISQ?



CISQ Measurement Standards



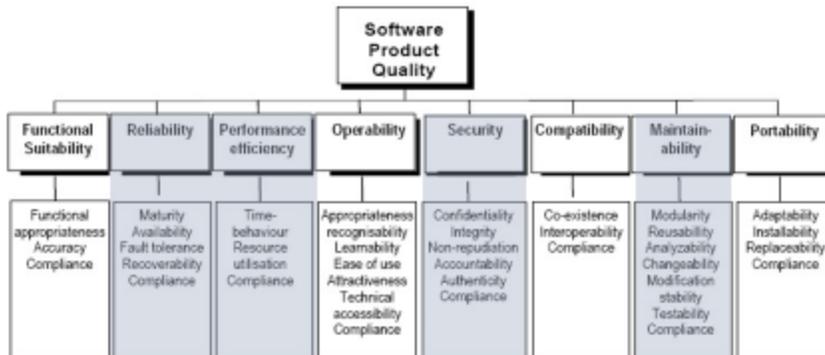
CISQ Measuring the Technology Stack



This publication is available free of charge from: <https://doi.org/10.6028/NIST.SP.500-320>

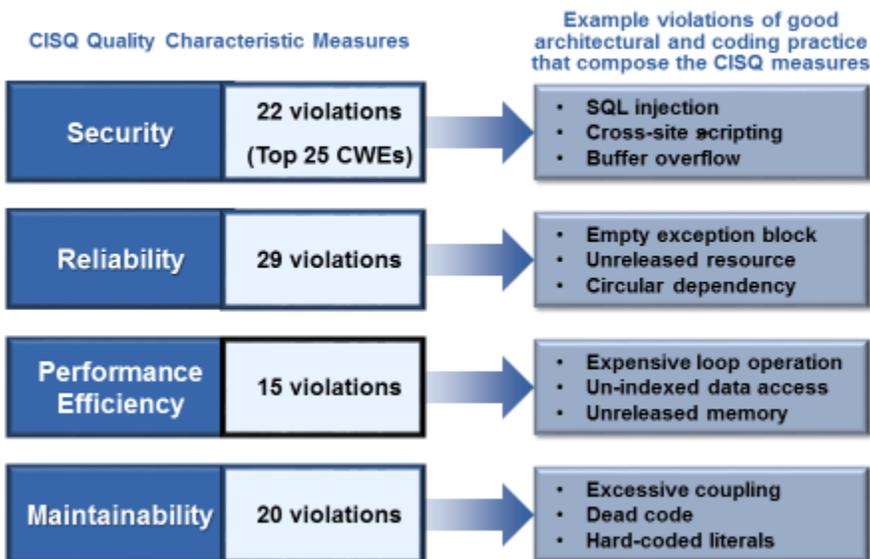
CISQ Relating CISQ Measures to ISO

- ISO 25000 series replaces ISO/IEC 9126 (Parts 1-4)
- ISO 25010 defines quality characteristics and sub-characteristics
- CISQ conforms to ISO 25010 quality characteristic definitions
- ISO 25023 defines measures, but not at the source code level
- CISQ supplements ISO 25023 with source code level measures



CISQ defined automatable measures for quality characteristics highlighted in blue

CISQ CISQ Measures Violations



CISQ Security Measure — Top 22 CWEs

- **CWE-22** Path Traversal Improper Input Neutralization
- **CWE-78** OS Command Injection Improper Input Neutralization
- **CWE-79** Cross-site Scripting Improper Input Neutralization
- **CWE-89** SQL Injection Improper Input Neutralization
- **CWE-120** Buffer Copy without Checking Size of Input
- **CWE-129** Array Index Improper Input Neutralization
- **CWE-134** Format String Improper Input Neutralization
- **CWE-252** Unchecked Return Parameter of Control Element Accessing Resource
- **CWE-327** Broken or Risky Cryptographic Algorithm Usage
- **CWE-396** Declaration of Catch for Generic Exception
- **CWE-397** Declaration of Throws for Generic Exception
- **CWE-434** File Upload Improper Input Neutralization
- **CWE-456** Storable and Member Data Element Missing Initialization
- **CWE-606** Unchecked Input for Loop Condition
- **CWE-667** Shared Resource Improper Locking
- **CWE-672** Expired or Released Resource Usage
- **CWE-681** Numeric Types Incorrect Conversion
- **CWE-706** Name or Reference Resolution Improper Input Neutralization
- **CWE-772** Missing Release of Resource after Effective Lifetime
- **CWE-789** Uncontrolled Memory Allocation
- **CWE-798** Hard-Coded Credentials Usage for Remote Authentication
- **CWE-835** Loop with Unreachable Exit Condition ('Infinite Loop')



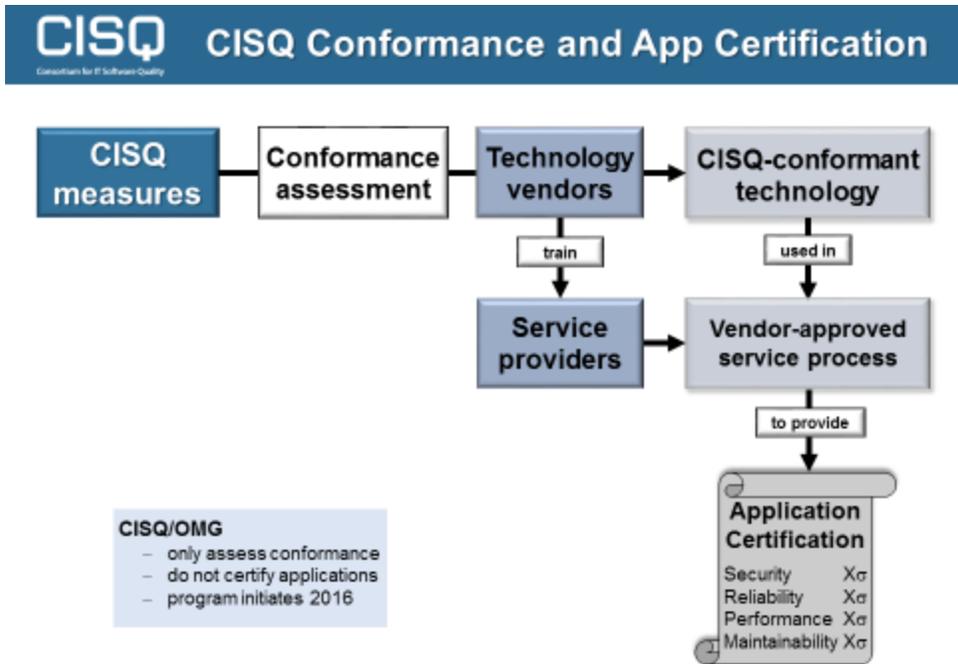
Robert Martin
MITRE



cwe.mitre.org

CISQ Issue → Quality Rule → Measure Element

Issue	Quality Rule	Quality Measure Element
CWE-79: Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')	Rule 1: Use a vetted library or framework that does not allow this weakness to occur or provides constructs that make this weakness easier to avoid, such as Microsoft's Anti-XSS library, the OWASP ESAPI Encoding module, and Apache Wicket.	Measure 1: # of instances where output is not using library for neutralization
CWE-89: Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')	Rule 2: Use a vetted library or framework that does not allow SQL injection to occur or provides constructs that make this SQL injection easier to avoid or use persistence layers such as Hibernate or Enterprise Java Beans.	Measure 2: # of instances where data is included in SQL statements that is not passed through the neutralization routines.



CISQ www.it-cisq.org
 Consultation for IT Software Quality

- Membership is free
- Measurement standards
- White papers, blogs
- Structural quality resources

- Automated FPs** <http://www.omg.org/spec/AFP/>
- Security** <http://www.omg.org/spec/ASCSM/>
- Reliability** <http://www.omg.org/spec/ASCRM/>
- Performance** <http://www.omg.org/spec/ASCPem/>
- Maintainability** <http://www.omg.org/spec/ASCMM/>

3.8 Mostly Sunny with a Chance of cyber

Mostly sunny with a chance of cyber¹

David Flater, NIST, 2016-05-09

Counting known vulnerabilities and correlating different factors with the vulnerability track records of software products after the fact is obviously feasible. The harder challenge is to produce “evidence to tell how vulnerable a piece of software is” with respect to vulnerabilities and attack vectors that are currently unknown. This means forecasting the severity and the rate at which currently unknown vulnerabilities will be discovered or exploited in the future, given a candidate system and its environment.

Meteorologists can observe the present state of a weather system and assume that the future state must evolve from it through the application of known physics. Small features that are below the resolution of the radar are correspondingly limited in their impact, so the uncertainty can be bounded. But for computer system vulnerabilities, there are no analogous limits. High-impact exploits of tiny, obscure quirks that were not on anyone’s “radar” appear with regularity. Although the resolution of that “radar” is continuously improved, the complexity of systems is increasing faster, so the relevant details are inexorably receding into the background.

Under these conditions, our best available predictors of future vulnerabilities in systems that were responsibly designed and implemented may be nothing more than metrics of size, complexity, and transparency. Unexciting as it may be, there is rationality to this approach. To develop a market for smaller, simpler, more verifiable systems would not be too modest a goal for a large government effort to attempt.

¹ Disclaimer: This statement reflects only the views of the author on the topics discussed, and does not necessarily reflect the official position that NIST may have about those topics.

Mostly sunny with a chance of cyber



David Flater
dflater@nist.gov
2016-07-06

1. This presentation reflects only the views of the author on the topics discussed, and does not necessarily reflect the official position that NIST may have about those topics.
2. Identification of commercial products and entities is not intended to imply recommendation or endorsement by NIST, nor is it intended to imply that the products or entities are necessarily the best available for the purpose.

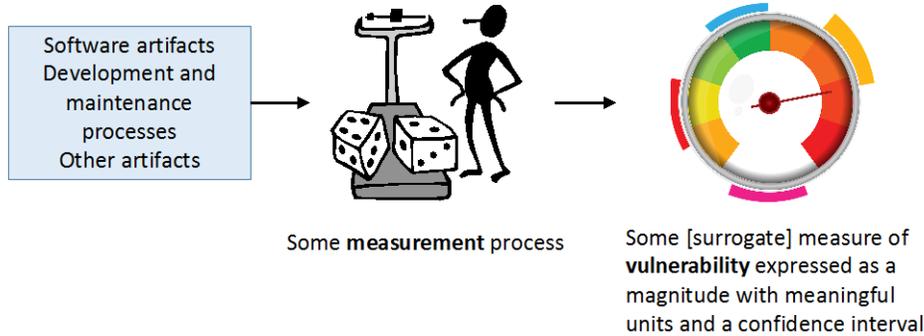
I added these notes after the workshop to include important points that don't appear in the text of the slides.

Thesis

- The nature of the challenge is not measurement, but prediction
- Conditions are unfavorable for making a rational prediction
- Measuring what *is* measurable and applying empiricism will move us forward
- Measuring *cost* reveals a complication

NIST Workshop on Software Measures and Metrics to Reduce Security Vulnerabilities

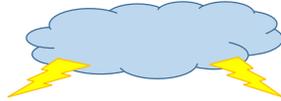
Challenge: produce “evidence to tell how vulnerable a piece of software is”



The metrology perspective is that measurement is about quantities. A quantity like 5 kg has meaning because it is defined as 5 times a standard reference, the unit. In most cases it would be nonsense to say that Software A is 5 times as vulnerable as Software B. Vulnerability is a quality, not a quantity. At best we may measure some quantity that helps us to characterize it better.

Measurement vs. forecasting

- Past: correlate different factors with the vulnerability track records of software products
- Present: count known vulnerabilities
 - Abuse of scale: count=2 does not mean twice as vulnerable as count=1; any count > 0 means *go fix your stuff*



- Future: forecast the severity and the rate at which currently unknown vulnerabilities will be discovered or exploited
 - No longer determining facts based on observations
 - Not causal: in theory, today's CVE *could be* the last
 - Prediction models can be better or worse

The count of known vulnerabilities is unsuitable as a surrogate measure of vulnerability. The future question is the most interesting one.

Prediction models

Attribute	Weather emergencies	Cyber emergencies
Preconditions	Known	Unknown, random
Conditions	Take time to evolve	Already in place
Set of variables	Fixed	Ever-expanding
Unseen details	Not important	Critically important
Guidance	Unguided	Precision-guided
Uncertainty	Frequentist	Epistemic
Degrees of control	Prepare, mitigate	<i>Preventable, in principle?</i>



In every respect but one (controllability), cyber emergencies are less predictable than weather emergencies. I will focus on the different impact of unseen details.

We can obtain an adequate prediction of impending weather emergencies even though the radar misses many small details. The butterfly effects do not matter as long as we can see the hurricane on its way with ample time to react. But for cyber emergencies it is exactly the opposite; it is the unseen details that are most likely to create an emergency with no warning at all.

Unseen details = blindside attack vectors

Where do they come from? Everywhere.

- **Electrical engineers**
 - Memory integrity quietly declined, enabling rowhammer.js
- **Implementation quirk, documented but overlooked**
 - Intel implemented an x86_64 instruction in a slightly different way than AMD had, enabling VM escape and escalate to hypervisor (XSA-7)
- **Unforeseen consequence of new feature**
 - Memory deduplication became a thing, enabling a much bigger side channel than was anticipated (Bosman et al. 2016)
- **Forgot about that legacy feature**
 - Everyone forgot about APIC register relocation or failed to see its usefulness, enabling another escalation to SMM (Domas 2015)
- **Accidentally introduced fault**
 - A random CPU erratum was discovered, enabling a remote exploit that looks like harmless code (Kaspersky & Chang 2008)

The threat model is of finite size. The unknown universe of potential attacks may be infinitely large. At least it is larger than our imagination, as we are consistently caught by surprise.

The idea that fully addressing the top 10 or top 25 attack vectors would cause there to be fewer successful attacks is an untested hypothesis. Past experience suggests that there is a large reserve of attack vectors that do not appear in the threat model. Perhaps attackers will simply move farther down the list and never run out of attacks.

Different perspectives, different metrics: the security industry sees progress in increasing the complexity of attacks, but the target sees no progress unless the frequency of attacks actually goes down.

The future will not be mitigated

- An assurance case is a fixed, closed-form expression up against an *evolving, open world*
- The unseen attack surface is vast and growing
- No opt-out



Even if you had complete visibility into the system as it stands, there is the problem of future-proofing the assurance case. We are forced to upgrade in order to close the barn door on known vulnerabilities. Each upgrade comes with an expanded attack surface, which leads directly to new vulnerabilities.

Risk models vs. unknown unknowns

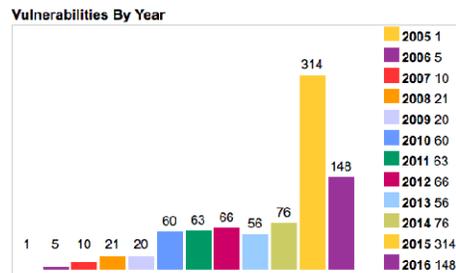
- "Risks"
 - Valid to estimate based on historical data
- "Structural uncertainties"
 - Follow from events that are rare or nonexistent in the historical record
 - Frequentist reasoning breaks down
- "*Unknowables*"
 - *Follow from inconceivable events*
 - *Bayesian reasoning breaks down*

Kees van der Heijden. Scenarios: The Art of Strategic Conversation.
John Wiley & Sons, 2nd edition, 2005.

A risk model cannot do justice to unknown unknowns. We cannot possibly estimate the probability of something that, by definition, we know absolutely nothing about. Such a number is nothing but an arbitrarily chosen safety margin.

Growth models

- No evidence that security grows / vulnerability decreases over time (?)
- "Trivial forecast has **some** predictive accuracy" (Timm Grams, "Reliability Growth Models Criticized")
- Applicable to the frequency of vulnerability discovery



Security may grow over time in tightly-controlled systems, but the more typical treadmill of vulnerabilities and mitigations suggests that it does not grow over time in general. (Taking the target's perspective that the difficulty of exploits is irrelevant if they just keep on happening.)

What is measurable?

- Known quantities
 - Track record of fixed vulnerabilities
 - Known unfixed vulnerabilities
 - Measurable hardness of certain kinds of defenses
- Hypothesized indicators of unknown vulnerabilities
 - Measures of diligence
 - Test/analysis coverage
 - Hardening measures
 - Size & complexity
 - Area of attack surface
 - "Code smells" (operationalized)
 - Transparency (including amenability to analysis of whatever kind)

Inventing a metric is only the beginning. Hypotheses must be tested. Measurements must be validated.

On measuring cost, and the problem that this reveals

- "Price of nonconformance" (Philip Crosby) or Cost Of Poor Quality (ASQ)
 - Post-release patching is much less costly than an auto recall
 - The consequential costs of vulnerabilities in COTS software are almost entirely paid by *consumers*, not *producers*
- "Quality is free"—not true
- "You can't afford *not* to test / build security in"—also not true
- Broken economy
- Consequence: there may be no security to 'measure'

This argument is not valid for products whose primary customer is the government, for regulated industries, or for long-lifecycle software. It applies only to the mass market.

We are familiar with studies showing that the cost of correcting defects is less if they are detected and corrected earlier in the process. But as long as the market tolerates faulty software, the producer's cost can be lowered further by just never correcting the defects. A lot of software is being produced as a consumable (or as part of a consumable) rather than a durable good. Maintenance is minimized, and after a date certain the product is simply abandoned and the next product is rolled out.

Within the mass market, the cost of poor security may even go negative: a more secure product may be too difficult to configure, resulting in a competitive disadvantage. Even if the cost of building security in is reduced to marginal as the strategic plan envisions, the business case may remain broken.

This economic problem may overwhelm and obviate the measurement problem.

Conclusions

- There is value in correlating different factors with the vulnerability track records of software products after the fact
 - Hypothesized indicators
 - Programming languages
 - Development techniques
 - Quality processes
 - Formal methods....
 - Engineering wasn't invented; it evolved
 - Do what [apparently] works, but verify and track progress
 - Goal: reliable predictors, best practices
- However, there also needs to be a business case
 - Redistributing risk may be necessary to "significantly curtail software vulnerabilities" in the COTS market



Empiricism is a useful strategy when we are overwhelmed by unknowns, but it must be used with great caution. Correlation is not causation. A good fit to past data does not ensure a good prediction. Hypotheses must be tested. Measurements must be validated. Apply science.

*"Measure what is measurable,
and stop yer lyin' about the rest"*

(Misquoting Galileo)

Software Metrology

David Flater
dflater@nist.gov

Not addressed: we also need software to be sufficiently functional running at least privilege that tricking users into granting excess permissions to trojans will no longer work.

3.9 Dynamical Proving That Security Issues Exist

Dynamical Proving That Security Issues Exist

Andrew V. Jones
Vector Software, Inc.
London, UK
andrew.jones@vectorcast.com

While static analyzers have given great benefit in processing and automatically checking large swathes of code, they still suffer from a high false-positive rate that leaves security engineers with a “needle in the haystack” when identifying the genuine vulnerabilities.

We put forward that, from a security perspective, approaches based on the “synthesis” of executions leading to specific software issues (i.e., the automatic construction of a dynamic test exploiting a given vulnerability), will become a unique and powerful weapon in the security mitigation arsenal.

The key benefit here is that these approaches can generate an “executable witness” through the code that demonstrates to a security practitioner exactly how a vulnerability can be exercised. Being executable is also crucial for the ever-growing market of IoT (internet of things) devices and CPS (cyber-physical systems), which are commonly targeted to embedded systems: there are real issues that may only be exploitable when run on the physical device. Furthermore, with IoT, we are seeing network-connected devices developed not using the traditional “web stack” (e.g., PHP, Apache, etc.), where security mitigations are commonly focused. This makes it hard to apply existing, “webstyle” penetration tools to such embedded devices, given their development is typically done in C or C++.

Such a “dynamic analysis” approach has two major benefits:

1. it can be used to verify that a given vulnerability has been “corrected” (in comparison to static analysis where a “warning” being removed could be a false-negative); and
2. for any issues that are spurious or are mitigated at the system-level, the automatically constructed and executable tests can form the documentation.

It is clear however that such dynamic approaches have a trade-off: zero false-positives (tool correctness allowing) at the expense of false-negatives. That is, while static analysis is “noisy” but (should) highlight all issues, a dynamic approach can only highlight what can be identified through unit/API-level testing, and through what the tool can create a test for (c.f., the halting problem). This is clearly a limitation, but the categories of issue detectable are still non-trivial; e.g., buffer overflows and NULL pointer issues are easily identifiable.

It is also possible to utilize such a “security-focused dynamic testing approach” to generate security metrics that can aid the practitioner in understanding and assessing the security of a given system. For example, the defect density (i.e., the ratio of defects per line of code) can be easily calculated given a set of identified vulnerabilities. Other relevant metrics include the “exploit depth”, that is, the length in the call chain from software entry to the location of vulnerability – a longer chain relates to more “missed opportunities” to mitigate against it. Similarly, the “attack surface” can also be enumerated by considering the source of any data related to the vulnerability (e.g., malicious data returned from a stub would be more serious than a function argument set directly via the test-tool). It is also possible to correlate the number of vulnerabilities detected with the “cyclomatic complexity” of a function. It follows that a function with a high complexity and a high number of vulnerabilities could be a weak point in the software that warrants further investigation.

We have demonstrated a proof-of-concept of the proposed scheme with a Tier 1 automotive supplier in the US, and found numerous security issues that could lead to a denial-of-service attack. These were issues at the software “product” level, where out-of-context re-use is of greater concern, when compared to their constrained usage within a single project. The approach was also integrated into the security review process of open source projects for a worldwide German industrial manufacturer – the outcomes highlighted potential API (mis-)uses that could lead to exploits “in the field”.

As always with security testing, we do not claim that dynamically executed tests are a golden panacea for software correctness; simply that they are another useful tool in attaining and preserving the overall (cyber)security of a given system.

Dynamically Proving That Security Issues Exist

Andrew V. Jones, Vector Software, Inc.
NIST SwMM-RSV, July 2016



© 2016 Vector Software, Inc. All Rights Reserved.

1/19

Focus of this talk

Chess & McGraw'04¹:

*Good **static checkers** can help spot and eradicate common security bugs*

Therefore (for the purposes of this talk!):

- If we find an instance of a CWE, it is a vulnerability!
- If it crashes the software, it can be a security issue
 - SIGSEGV ⇒ DoS!

¹see: <https://www.computer.org/csdl/mags/sp/2004/06/j6076.pdf>



© 2016 Vector Software, Inc. All Rights Reserved.

2/19

Focus of this talk

Chess & McGraw'04¹:

*Good **dynamic analysers** can help spot and eradicate common security bugs*

Therefore (for the purposes of this talk!):

- If we find an instance of a CWE, it is a vulnerability!
- If it crashes the software, it can be a security issue
 - SIGSEGV ⇒ DoS!

¹see: <https://www.computer.org/csdl/mags/sp/2004/06/j6076.pdf>



A tale of two customers...

Customer A

- We have some testing of open source projects
- Can you find any issues?
- Display issues

Customer B

- VectorCAST performed automated test-case generation
- Can you find any issues?
- Fuzzing of test-cases
- Display issues



What did they want?

The view from the trenches

- **Binary** – do we have any issues? Yes or no!
- **Count** – how many?
- **Identification** – what and where are they?



© 2016 Vector Software, Inc. All Rights Reserved.

4/19

So how did it work?

Crash-test generation

- Take a test that allocates a pointer
- Remove the `alloc`
- Run it
- Does it crash?
- If yes: **potential weakness!**



© 2016 Vector Software, Inc. All Rights Reserved.

5/19

Caveat emptor

- This is white-box unit testing – not black-box “Dynamic Application Security Testing” (DAST)!
- We can only find defects in what we can deduce a test for
 - Not trying to solve the halting problem – things can slip through our net
- Aiming for soundness (if we say it is a bug, it is a bug); no chance of completeness
 - We can't catch every bug because some are infeasible to generate unit tests for automatically



© 2016 Vector Software, Inc. All Rights Reserved.

6/19

Example

Example from LIGHTTPD (v1.4.20; v1.4.18 in SATE'08)

```
1 int buffer_copy_string_buffer(buffer *b, const buffer *src) {  
2     if (!src) return -1;  
3  
4     if (src->used == 0) {  
5         b->used = 0;  
6         return 0;  
7     }  
8     return buffer_copy_string_len(b, src->ptr, src->used - 1);  
9 }
```

- **Not** detected: CPPCHECK, Facebook's INFER, UNO
- Possible error: LINT, CODEHAWK
- SIGSEGV: VectorCAST!



© 2016 Vector Software, Inc. All Rights Reserved.

7/19

Results from SARD

- Took the null pointer issues from the [Software Assurance Reference Dataset](#)² (“vulnerable” C test-suite)
- Found 6 out of 7 issues
- We didn’t find (null_deref_local_flow-bad.c):

```
1  /* SNIP */
2
3  char k = 'a';
4  char* p = (char*)NULL;
5
6  switch (k)
7  {
8      case 0:
9          k = *p;          /* FLAW */
10
11  /* SNIP */
```

²see: <https://samate.nist.gov/SARD/>



Benefits

Static analysis might not detect it

- False-positives are high – is it a real error?
- False-negatives exist – maybe they didn’t show it?

Dynamic execution

- We don’t claim to detect everything
 - “happy” to have false-negatives
- If we *do* find something, it is definitely an issue!
- You can fix the issue, and re-generate and re-execute that test: if the error goes away, that issue is fixed!
 - With static analysis, you might have just hidden the error under a false-negative!



Vulnerabilities of interest

Automatic identification for CWE-398 (“indication of poor code quality”)

- Anything with “hard” errors
 - Use of a `null` pointer (CWE-476)
 - Buffer {under,over}flow (stack corruption) (CWE-124)
 - Divide by zero (CWE-369)
- VectorCAST supports stubbing ⇒ detection of
 - **Mismatched calls** – `malloc/free`, `fopen/fclose`, `pthread_mutex_lock/pthread_mutex_unlock` (CWE-401/404/413/415/590)
 - **Bad arguments** – `memcpy` (CWE-120/130)
 - **Unchecked return** – `malloc` (CWE-252)



© 2016 Vector Software, Inc. All Rights Reserved.

10/19

What are we aiming for?

- Source of tests (pick one!)
 - Take existing tests + code coverage data
 - Symbolic execution data for test-case generation
- Source of defects (pick one!)
 - Static analysis data (from \$YOUR_FAVOURITE_SA_ENGINE)
 - Symbolic execution data for vulnerabilities
- Generate
 - Fuzz’d tests or tests to cover vulnerabilities
- Execute tests
 - Detect vulnerabilities



© 2016 Vector Software, Inc. All Rights Reserved.

11/19

What are we aiming for?

- Source of tests (pick one!)
 - Take existing tests + code coverage data
 - Symbolic execution data for test-case generation
- Source of defects (pick one!)
 - Static analysis data (from \$YOUR_FAVOURITE_SA_ENGINE)
 - Symbolic execution data for vulnerabilities
- Generate
 - Fuzz'd tests or tests to cover vulnerabilities
- Execute tests
 - Detect vulnerabilities



© 2016 Vector Software, Inc. All Rights Reserved.

11/19

Metrics

I thought this was a talk on metrics?!

“actionable intelligence”



© 2016 Vector Software, Inc. All Rights Reserved.

12/19

Towards “application security”

Process³

1. Identify portfolio
2. Assess vulnerabilities
3. Manage risk

Some of the issues we find you might consider are “non-issues” or are mitigated against as part of your software architecture

- That’s great...
- ...be wary about software re-use across projects!

³see: https://www.rsaconference.com/writable/presentations/file_upload/asec-w25.pdf



Easy metrics

An approach to ascertaining quickly Chess’s “Morningstar for Software Security”⁴

- ☆☆☆ – “absence of obvious reliability issues”

The easy ones

- Defect density
 - Defects/SLoC
- Lines free from obvious issues (via code coverage)
 - Confidence of “defect freedom” (but not guaranteed!)
- Ratio of security tests free of defects
 - Higher ratio ⇒ more secure

⁴see: <http://www.securitymetrics.org/attachments/Metricon-2-Lee-Chess-Enterprise-Metrics.ppt>



More involved metrics

- Exploit depth (from how many levels can we trigger it?)
 - Akin to a linear “attack graph”
 - More steps ⇒ high critically
- Criticality (e.g., things that crash vs. things that don't)
 - Assess the risk using CWRAF/CWSS
 - SIGSEGV >> missing **free**
- Correlation between function complexity and number of defects
 - High complexity and number of defects ⇒ higher risk
- Percentage breakdown of metrics by type/grouping
- Attack surface⁵ (e.g., defect via params vs. return from stub)
 - Clearly serious if it is via a stub of **recv!**

⁵see: <http://www.cs.cmu.edu/~pratyus/tse10.pdf>



Sample metrics for null pointer defects

Metric	Project		
	LIGHTTPD	ZLIB	LIBXML2
Version	1.4.20	1.2.8	2.9.4
# files	89	16	84
SLoC ⁶	36,605	6,726	184,179
Unique # issues	709	113	2,926
Defect density (defects/line)	1/52	1/60	1/63
Avg. # of tests per defect	11	7	12
Tests hitting defects	69%	28%	40%
Funct's with defects	44%	44%	29%
Funct's with vg ≥ 20 and defects ⁷	51%	55%	66%

⁶measured with cLOC

⁷Jones'08: “[complexity] levels greater than 20 are considered hazardous”



Future metrics

- Number of vulnerabilities that are already “guarded” (e.g., if a pointer passes through some pointer test but still crashes)
 - Similar to disregarding issues if they are guarded by “intrusion protection systems”⁸
- Build a correlation to predict the vulnerability of a package⁹:
 - Extract a characteristic of the software for version n
 - Extract a vulnerability metric from the software for version n
 - Use characteristics of $n + 1$ to predict vulnerabilities in $n + 1$

⁸see: <http://www.securitymetrics.org/attachments/Metricon-1-Epstein-Software.ppt>

⁹see: <http://www.securitymetrics.org/attachments/Metricon-5-Massacci-Firefox-Vulnerabilities.pdf>



Take-home

Mainly: no “one size fits all” solution – use multiple tools!

- Dynamic execution can find certain vulnerabilities more definitively
- Need to always consider DP-E ratio (damage potential vs. effort)
- A number of metrics
 - Not necessarily specific to dynamic execution – also relevant to the output of a static analyser
- **Future work:** how can metrics be used to *predict* vulnerability



Fin.

Questions?

andrew.jones@vectorcast.com



© 2016 Vector Software, Inc. All Rights Reserved.

19/19

Toward Evidence-Based Low-Defect Software Production

James Kirby Jr.
Naval Research Laboratory
james.kirby@nrl.navy.mil

Introduction

It's hard to overstate the big bet that Defense, government, and industry have placed on computer software, this critical building material of the early 21st Century. All rely on software to deliver innovation in products and processes. Software provides as much as 90% of recent military aircraft functionality. Many previously mechanical, radio frequency, and chemistry-based products (e.g., automobiles, telephones, cameras) are now implemented by complex software driving computers embedded with sensors and actuators. [12] Much National Critical Infrastructure identified by Federal policy [15] is software-intensive, e.g., Communications, Financial Services, Healthcare and Public Health, Information Technology, Transportation Systems. Numerous important Federal initiatives are also software-intensive, e.g., Health IT, the National Strategic Computing Initiative (NSCI), Smart Cities and Connected Communities.

There is broad-based dependence on software throughout the economy. Table 1 “which lists U.S. industry sectors employing more than 50,000 software [developers], illustrates the heterogeneous dependence on software production of an advanced, modern economy.” [13] The “software industry,” which includes software publishers and internet services, employs only about half of the more than 175,000 software developers employed in the Information sector and less than 10% of the more than one million developers employed by all industry identified in Table 1. The software trade group, The App Association, found that only 10% of software developers are employed in Silicon Valley; the vast majority is widely spread across all 50 U.S. states. [11]

Table 1 Sectors employing more than 50,000 software [developers]

Industry Sectors	Developers (thousands)
Manufacturing	147.9
Wholesale Trade	59.5
Information	175.2
Finance & Insurance	99.2
Professional, Scientific, Technical Services	530.3
Management of Companies & Enterprises	54.9
Total	1,067

Because software is a critical building material, the source of that software is likewise critical. This paper uses the term *software production* to refer to three types of activities responsible for producing software:

- *Software development*, as widely understood.
- *Software sustainment*, which evolves software throughout operational life as needs, understanding, technology, and infrastructure inevitably evolve.
- *Software assurance*, which develops confidence that evolving software continues to exhibit critical properties. [13]

The economics of meeting time-to-deliver and cost objectives favor the production of defective, vulnerable software. This paper considers the following lines from security researcher Crispin Cowan to reflect the common wisdom of software development today:

"Perfect" (bug-free) software is impractically expensive and slow to produce, and so the vast bulk of consumer and enterprise software products are shipped when they are "good enough" but far from bug-free. As a consequence, there has been a constant struggle to keep attackers from exploiting these chronically inevitable bugs. [5]

Another way of expressing this common wisdom is to say that developers and users are unable to develop and sustain in a timely and affordable manner software exhibiting low defect rates. Many software defects are software vulnerabilities, making them a significant obstacle to the success of cybersecurity efforts. Too, software defects are a considerable drag on the economy. Users avoiding and mitigating software defects may waste as much as 1% of GDP. [10]

Alternative software production technologies

Enabling developers and users to develop and sustain low-defect software in a timely and affordable manner, which is critical to the US, requires improved software production technology guided by a deeper, evidence-based understanding. The third strategic theme of the NSCI, "Improve HPC [High Performance Computing] application developer productivity," describes improved software production technology for HPC: "new approaches to building and programming HPC systems that make it possible to express programs at more abstract levels and then automatically map them onto specific machines." [6] While there is much interest in reducing software vulnerabilities, industry is more likely to adopt technology that reduces time and effort to produce software with reduced vulnerabilities. Some promising alternative technologies to the prevalent labor-intensive hand-coding paradigm include:

- *Software product line engineering* [4][14] constructs software products as instances of a family of similar products in an effort to rapidly produce and evolve high-quality software. Developers and users resolve decisions to identify a desired family member. The resolved decisions reflect differences in customer needs and engineering tradeoffs.
- *Model-driven development* [8][9] generates software from models that specify software behavior. Developers and users create the models.
- *Synthesis formal methods* [1][7] generate correct-by-construction implementations of domain-specific high-level descriptions of desired behavior created by developers and users.
- *Program transformations*. [3] Developers and users guide selection of transformations of formal designs to produce correct-by-construction code.

Improving software production technology and knowing it

To have confidence that efforts to improve software production technology are succeeding, it is imperative to have evidence-based means to guide and evaluate them. The Goal/Question/Metric (GQM) paradigm [2] is an evidence-based approach to achieving an understanding that can provide guidance to fast, affordable production of low-defect software. Such an approach can help select improved alternative software production technologies and further refine them. GQM defines measurement in a top-down fashion based on goals. Identified goals are refined into a set of quantifiable questions, which imply metrics that guide data collection. Collected data provides the evidence-based view.

Goals that could lead to evidence-based, low-defect software production and some corresponding quantifiable questions include:

- *Reduce defect rate* of developed and sustained software.
 - What is the software defect rate? Where are defects inserted? Removed?
- *Reduce time* to develop, sustain, and assure software.
 - How much time is required to develop, sustain, and assure software? Is time wasted?
- *Reduce effort* to develop, sustain, and assure software.
 - How much effort is required to develop, sustain, and assure software? Is effort wasted? Is effort duplicated?
- *Widespread insertion* of improved software production technology.
 - What technology are developers and users using?
 - What national investment is required to insert improved technology?
 - What software tools are available to support improved technology? Is there a healthy market to sustain them?

Summary

Software and its production are critical to Defense, government, and the economy. The economics of meeting time-to-deliver and cost objectives favor the production of defective, vulnerable software, which undermines cybersecurity and imposes considerable drag on the economy. Industry is more likely to adopt technology that reduces time and effort to produce software with reduced vulnerabilities. Enabling developers and users to develop and sustain low-defect software in a timely and affordable manner, which is critical to the US, requires improved software production technology. An evidence-based approach to evaluating software production technologies can facilitate their successful selection and refinement.

Acknowledgements

The paper has benefited from the author's conversations with David Weiss and Sol Greenspan.

References

- [1] Alur et al. "Syntax-guided synthesis." *Dependable Software Systems Engineering* 40 (2015): 1-25.

- [2] Basili. "Applying the Goal/Question/Metric paradigm in the experience factory." *Software Quality Assurance and Measurement: A Worldwide Perspective* (1993): 21-44.
- [3] Baxter and Mehlich. "Reverse engineering is reverse forward engineering." *Reverse Engineering, 1997. Proceedings of the Fourth Working Conference on*. IEEE, 1997.
- [4] Campbell. "Renewing the product line vision." *Software Product Line Conference, 2008. SPLC'08. 12th International*. IEEE, 2008.
- [5] C. Cowan. "Reflections on Decades of Defending Imperfect Software," NSF WATCH Talk, 17 July 2014.
- [6] "FACT SHEET: National Strategic Computing Initiative." 29 July 2015.
- [7] Kant. "Synthesis of Mathematical-Modeling Software." *IEEE Software*, May 1993
- [8] Kirby. "Model-Driven Agile Development of Reactive Multi-Agent Systems." *Proc. 30th Annual Intl. Computer Software and Applications Conf. (COMPSAC 2006)*.
- [9] Kirby. "Specifying software behavior for requirements and design." *Journal of Systemics, Cybernetics and Informatics*. Oct. 2013.
- [10] National Institute of Standards & Technology. 2002. *The Economic Impacts of Inadequate Infrastructure for Software Testing*. Planning Report 02-3.
- [11] <http://www.nextgov.com/cio-briefing/wired-workplace/2016/07/90-software-developers-work-outside-silicon-valley/129852/>
- [12] K. Sullivan, private communication.
- [13] Weiss, Kirby, and Lutz. "Moving Toward Evidence-Based Software Production." *Perspectives on the Future of Software Engineering*. Springer Berlin Heidelberg, 2013. 275-298.
- [14] Weiss and Lai. "Software product line engineering: a family based software engineering process." (1999).
- [15] The White House, *Presidential Policy Directive—Critical Infrastructure Security and Resilience*, 12 February 2013.

3.11 Position Statement for July 12, 2016 Workshop

Carol Woody, Ph.D.

USING MALWARE ANALYSIS TO REDUCE DESIGN WEAKNESSES

Establishing that software has reduced security vulnerabilities needs to include consideration of the implemented software as it functions within a specific system context. Lots of things can be counted, but useful security metrics must show that the software functions as intended and only as intended. Every acquisition and development activity is driven by requirements. If it is not in the requirements (and requirements drive the contract which defines the acquisition) then it does not get done.

The first challenge is to establish that the requirements define the appropriate security behavior and design addresses these security concerns. The second challenge is to establish that the completed product, as built, fully satisfies the specifications. Measures, therefore, must address requirements, design, construction, and test.

Requirements must include appropriate consideration of the threats the software must be able to handle in the operational context. Malware attacks are growing alarmingly but structured mechanisms to include data from known malware attacks into requirements and architecture processes are nonexistent. SEI research has shown that when designs ignore these types of attacks, important security controls are omitted. Even those rare projects today that do some form of threat modeling fail to systematically consider prior successful exploits. Evidence indicates that projects with detailed data about successful prior attacks are more likely to appropriately create critical mitigations. This data exists but is not formulated in any structure that system and software engineers can understand and help stakeholders incorporate into requirements.

Copyright 2016 Carnegie Mellon University
DM-0003850

3.12 Measure Early and Measure Often – SWAMP, Miron Livny

Measure early and Measure often – the Continuous Assurance framework of the Software Assurance Market Place (SWAMP)

Miron Livny

Director of the SWAMP Project

Professor of Computer Science

Morgridge Institute of Research

Like any other complex system, early and frequent measurements of the properties of software artifacts throughout the design, implementation, and deployment phases significantly increase the impact of the obtained metrics on the quality of the observed software. The dynamic and diverse nature of software components, the metrics needed to assess the properties of software and the measurement technologies used to derive these metrics require a dependable and extensible framework that effectively automates the frequent derivation of the desired metrics.

The freely available Open Source framework of the SWAMP has been developed by a joint effort of four academic institutions and funded by the S&T division of the DHS to provide a suite of easy to use, easy to deploy and easy to integrate continuous assurance capabilities. The framework brings together state-of-the-art automation with the power of distributed High Throughput Computing technologies to manage the assessment of multiple software packages with multiple tools in a secure environment that enables controlled sharing across project boundaries. By offering a diverse collection of software packages and easy access to the GitHub repository, the open SWAMP facility supports developers of measurement tools who want to continuously evaluate their tools as they evolve through the design and implementation phases.

Input and feedback from actual users and potential users has been invaluable to the effort to design and implement the capabilities of the SWAMP. The workshop will provide us with a unique opportunity to present the current capabilities of the SWAMP and exchange ideas, challenges, requirements and solution with individuals who are committed to advance software security through a comprehensive approach to the measurement and assessment of software artifacts and the processes of designing, developing, operating and maintain complex software stacks.