Hunting for Aardvarks: Can Software Security be Measured?

Martin Gilje Jaatun

Department of Software Engineering, Safety and Security
SINTEF ICT
NO-7465 Trondheim, Norway
martin.g.jaatun@sintef.no
http://www.sintef.no/ses

Abstract. When you are in charge of building software from the ground up, software security can be encouraged through the use of secure software development methodologies. However, how can you measure the security of a given piece of software that you didn't write yourself? In other words, when looking at two executables, what does "a is more secure than b" mean? This paper examines some approaches to measuring software security, and reccommends that more organisations should employ the Building Security In Maturity Model (BSIMM).

1 Introduction

When discussing secure software engineering, the main argument for employing a secure software development lifecycle is that it makes the software "more secure" - but exactly what that means isn't entirely clear.

One way of measuring security could be to count the number of security bugs/flaws/attacks against a given software product over time, which is generally the service offered by the Common Vulnerability and Exposures (CVE) [1] and the National Vulnerability Database (NVD) [2]. You could argue that this gives an after-the-fact comparison between different products, with the products with the least number of vulnerabilities claiming the "most secure" title. However, these kind of statistics can easily degenerate into the "damn lies" category [3], since they do not take into account the following factors:

- What is the distribution of the software product?
- What is the attacker's incentive for breaking the product?

For instance, an obscure piece of software could easily go for decades without making it into any vulnerability databases even if it were riddled with security flaws, whereas one reason for the high number of discovered security flaws in Microsoft products can be that due to its large user base, Microsoft remains the target of choice among the hacker population. Also, to quote Fred Brooks: "More users find more bugs." [4].

The remainder of this paper is organized as follows: In section 2 we[1] present some other approaches toward measuring software security. In section 3 we discuss how different development methodologies might affect software security, and in section 4 we briefly consider what role testing might play in measuring software security. In section 5 we outline the Building Security In Maturity Model (BSIMM), and argue why it could be a good approach to software security metrics. We discuss our contribution in section 6, and offer conclusions in section 7.

2 Background

We will in the following present some previous work on measuring software security, all of which has been presented at the MetriCon series of workshops.

2.1 A Retrospective

Ozment and Schechter [5] studied the rate of discovered security flaws in OpenBSD over a span of 7 years and 6 months. Unsurprisingly, they found that the rate of new bugs discovered in unchanged code dropped toward the end of the period (i.e., the number of latent security flaws are presumably constant, and as time goes by, more and more of the flaws will be found). However, new code is continually added to the code base, and this means that also new vulnerabilities are introduced - Ozment and Schechter found that comparatively fewer vulnerabilities were introduced through added code, but attributed this to the fact that the new code represented a comparatively small proportion of the total code base (39%). In fact, that 39% of code had 38% of the total security flaws, which is within the statistical margin of error.

This contrasts with Brooks' contention that "Sooner or later the fixing ceases to gain any new ground. Each forward step is matched by a backward one. Although in principle usable forever, the system has worn out as a base for progress." [4] It is possible that different rules apply to operating systems than application programs – or that we still haven't reached the "trough of bugginess" in OpenBSD.

2.2 The MetriCon Approach to Security Metrics

The MetriCon workshop was held for the first time in August 2006, in conjunction with the USENIX Security Symposium. Since this workshop doesn't publish regular proceedings, details are a bit hard to come by for those that did not attend, but luckily the workshop publishes a digest[2] of the presentations and ensuing discussions, reported by Dan Geer [6–8] and Daniel Conway [9].

MetriCon covers a wide swath of what can be called security metrics, but is a reasonable place to look for contribtutions to measuring software security. A

[1] The reader is free to interpret this as the "royal we".
[2] At least for the first four events.

discussion at the first MetriCon [6] touched upon *code complexity* as a measure of security. This is an approximate measure, at best, since complex code is more difficult to analyze (and may thus help to hide security flaws); but correctly written complex code will not in itself be less secure than simple code. However, if proof of secure code is needed, complexity is likely to be your downfall – it is no coincidence that the highest security evaluation levels typically are awarded to very simple systems.

3 Comparing Software Development Methodologies

Traditional approaches to developing secure software have been oriented toward a waterfall development style and "Big Requirements Up Front" – see e.g. the documentation requirements of a Common Criteria evaluation [10].

However, the jury is still out in the matter of the security of code produced using e.g. agile methods vs. waterfall. There are proponents who claim that XP works just dandy in safety-critical environments (and, presumably, by extension with great security), while other examples demonstrate that an agile mindset purely focused on "let's get this thing working, and let's worry about security later" does not present the best starting point for achieving secure code.

Eberlein and Leite [11] state that the main reason agile methods result in poor security requirements is that the agile methods do not provide verification (are we building the product right), only validation (are we building the right product). Beznosov [12] thinks XP can be good enough, while Wäyrynen et al. [13] claim that the solution to achieving security in an XP development is simply to add a security engineer to the team.

Beznosov and Kruchten [14] compare typical security assurance methods with generic agile methods, and identify a large number of the former that are at odds with the latter (in their words: mis-match). Unsurprisingly, this indicates that it is not possible to apply current security assurance methods to an agile software development project. The authors offer no definite solution to this problem, but indicate two possible courses of action:

1. Develop new agile-friendly security assurance methods. The authors concede that this will be very difficult to achieve, but are not able to offer any insights yet on what exactly such methods could be.
2. Apply the existing security assurance methods at least twice in every agile development project; once early in the lifecycle, and once towards the end.

Siponen et al. [15] believe that all will be well if think about security in every phase. While Poppendieck [16] argues that agile methods (specifically: XP) are just as suitable as traditional development methods for developing safety-critical applications. Kongsli [17] opines that agile methods provide an opportunity for early intervention in connection with securing deployment, and argues for collective ownership of security challenges. However, a security specialist is still required as part of the team.

4 Testing for Security

There exist various static analysis tools that can analyze source code and point out unfortunate content, but just like signature-based antivirus products, these tools can only tell you about a set of pre-defined errors [18].

The ultimate challenge is to be presented with an executable and trying to figure out "how secure is this?". Jensen [19] discusses several approaches to evaluate an executable for unwanted side-effects, but this only covers software with hostile intent, not software that is poorly written.

Fuzzing [20] is a testing technique based on providing random input to software programs, and observing the results. This is an automated version of what used to be referred to as the "kindergarten test"; typing random gibberish on the keyboard. Unfortunately, while it may be possible to enumerate all intended combinations of input to a program, it is not possible to do exhaustive fuzz testing – even if you leave the fuzzer running for weeks, it will still not have exhausted all possible combinations. Thus, fuzzing is not a suitable candidate for a software security metric – if you find flaws, you know the software has flaws; if you don't find flaws, you know ... that you didn't *find* any flaws – but there may be flaws hiding around the next corner.

5 BSIMM and vBSIMM

The Building Security In Maturity Model (BSIMM) [21] and its simpler "younger brother"[3] BSIMM for Vendors (vBSIMM) were introduced by McGraw as an attempt to bypass the problem of measuring software security; arguing that if you cannot measure the security of a given piece of software, you can try to measure second-order effects, i.e. count various practices that companies that are producing good software security are doing.

5.1 The BSIMM Software Security Framework

BSIMM defines a Software Security Framework (SSF) divided into four domains each covering three practices (see Table 1). Each practice in turn covers a number of activities grouped in three levels (see below).

- The **Governance** domain includes practices *Strategy and Metrics*, *Compliance and Policy*, and *Training*.
- The **Intelligence** domain includes practices *Attack Models*, *Security Features and Design*, and *Standards and Requirements*.
- The **SSDL Touchpoints** domain refers to McGraw's approach to a Secure Software development Lifecycle [23], and includes practices *Architecture Analysis*, *Code Review*, and *Security Testing*. There are more touchpoints

[3] In a way the opposite of Sherlock Holmes' smarter older brother Mycroft - "When I say, therefore, that Mycroft has better powers of observation than I, you may take it that I am speaking the exact and literal truth." [22]

listed in McGraw's book, but these three have been identified by McGraw as the most important.
– The **Deployment** domain includes practices *Penetration Testing*, *Software Environment*, and *Configuration Management and Vulnerability Management*.

Governance	**Intelligence**	**SSDL Touchpoints**	**Deployment**
Strategy and Metrics	Attack Models	Architecture Analysis	Penetration Testing
Compliance and Policy	Security Features and Design	Code Review	Software Environment
Training	Standards and Requirements	Security Testing	Configuration Management and Vulnerability Management

Table 1. The BSIMM Software Security Framework

5.2 Maturity is one of the M-s in BSIMM

Each BSIMM practice contains a number of *activities* grouped in three maturity levels. Each maturity level is given a textual description, but it's not entirely clear if *all* the activities in a lower level need to be in place to progress to the next level – it may be assumed that the BSIMM "auditors" employ some discretion here when collecting the interview data.

To take a random example, we can look at the *Security Testing* (ST) practice within the **SSDL Touchpoints** domain. ST level 1 is labeled "Enhance QA beyond functional perspective", and comprises the activities:

– ST1.1: Ensure QA supports edge/boundary value condition testing.
– ST1.2: Share security results with QA.
– ST1.3: Allow declarative security/security features to drive tests.

ST level 2 is labeled "Integrate the attacker perspective into test plans", and currently has only two activities:

– ST2.1: Integrate black box security tools into the QA process (including protocol fuzzing).
– ST2.3: Begin to build/apply adversarial security tests (abuse cases).

The third ST level is labeled "Deliver risk-based security testing", and has four activities:

– ST3.1: Include security tests in QA automation.

- ST3.2: Perform fuzz testing customized to application APIs.
- ST3.3: Drive tests with risk analysis results.
- ST3.3: Leverage coverage analysis.

The BSIMM authors reccommend that if using BSIMM as a cookbook, an organization should not try to jump to the third level all at once, but rather implement the first-level activities first, and then move on only once the first level is truly embedded. This is partly because some higher-level activities build on the lower-level ones (e.g., ST2.1 and ST1.1), but also because the higher-level activities typically are more difficult and require more resources.

5.3 BSIMM in Practice

The BSIMM documentation is freely available under a Creative Commons license, and in theory there is nothing to stop anyone from using it to compare new organizations to the ones already covered. However, it is clear that the raw data used in creating the BSIMM reports is kept confidential, and BSIMM is no interview cookbook – it is safe to assume that participants are not asked directly "do you use attack models?", but exactly how the BSIMM team goes about cross-examining their victims is not general knowledge, and is thus difficult to reproduce.

Using the BSIMM as a research tool may therefore be more challenging than using it as a self-assessment tool, and the latter is certainly more in line with the creators' intentions.

6 Discussion

It is unlikely that we'll see any "fire and forget" solution for software security in the near future, but we may aspire to a situation of "forget and get fired", i.e. where software security becomes an explicit part of development managers area of responsibility.

Recently, we have seen in job postings for generic software developers that "knowledge of software security" has been listed as a desired skill – this may be a hint that the software security community's preaching has reached beyond the choir.

If you want a job done right, you have to do it yourself – but if you can't do it yourself, you need other evidence. It seems that for lack of anything better, the BSIMM approach of enumerating which of the "right" things a software company is doing is currently the best approach to achieve good software security. It is true that past successes cannot guarantee future happiness; but on the other hand, a company that has demonstrated that it cares enough to identify good software security practices is more likely to follow these in the future than a company that does not appear to be aware of such practices in the first place.

7 Conclusion and Further Work

There is currently no good metric which can easily decide which one of two executable is better from a software security point of view. It seems that currently, the best we can do is is to measure second-order effects to identify which software companies are trying hardest. If we are concerned about software security, those are the companies we should be buying our software from.

More empirical work is needed on comparing software produced by different methodologies, e.g. agile vs. waterfall. Intuitively, the former may seem less formal and thus less security-conscious, but an interesting starting point may be to compare the number of secure software engineering practices employed in the different organizations. Retrospective studies may also compare the track record of various methodologies over time, but the main challenge here may be to identify software that is sufficiently similar in distribution and scope to make the comparison meaningful.

Acknowledgment

The title of this paper is inspired by an InformIT article by Gary McGraw and John Stevens [18]. Thanks to Jostein Jensen for fruitful discussions on software security for the rest of us.

References

1. CVE: Common Vulnerabilities and Exposures (CVE). http://cve.mitre.org/
2. NVD: National Vulnerability Database Home. http://nvd.nist.gov
3. Clemens, S.L.: Notes on 'innocents abroad': Paragraph 20 (2010) ('There are three kinds of lies: lies, damned lies, and statistics.' - Attributed to Disraeli) – http://marktwainproject.org.
4. Brooks, F.P.: "The Mythical Man-Month". Addison-Wesley (1995)
5. Ozment, A., Schechter, S.E.: Milk or wine: does software security improve with age? In: Proceedings of the 15th conference on USENIX Security Symposium - Volume 15. USENIX-SS'06, Berkeley, CA, USA, USENIX Association (2006)
6. Geer, D.: MetriCon 1.0 Digest (2006) http://www.securitymetrics.org/content/-Wiki.jsp?page=Metricon1.0.
7. Geer, D.: MetriCon 2.0 Digest (2007) http://www.securitymetrics.org/content/-Wiki.jsp?page=Metricon2.0.
8. Geer, D.: MetriCon 4.0 Digest (2009) http://www.securitymetrics.org/content/-Wiki.jsp?page=Metricon4.0.
9. Conway, D.: MetriCon 3.0 Digest (2008) http://www.securitymetrics.org/content/Wiki.jsp?page=Metricon3.0.
10. ISO/IEC 15408-1: Evaluation criteria for it security part 1: Introduction and general model (2005)
11. Eberlein, A., do Prado Leite, J.C.S.: Agile requirements definition: A view from requirements engineering. In: PROCEEDINGS OF THE INTERNATIONAL WORKSHOP ON TIME-CONSTRAINED REQUIREMENTS ENGINEERING (TCRE'02). (2002)

12. Beznosov, K.: eXtreme Security Engineering: On Employing XP Practices to Achieve "Good Enough Security" without Defining It. In: Proceedings of the First ACM Workshop on Business Driven Security Engineering (BizSec). (2003)
13. J. Wäyrynen and M. Boden and G. Bostrøm: Security engineering and eXtreme programming: An impossible marriage? In: Extreme Programming and Agile Methods - Xp/ Agile Universe 2004, Proceedings. Volume 3134 of Lecture Notes in Computer Science., Springer-Verlag Berlin (2004) 117–128
14. Beznosov, K., Kruchten, P.: Towards agile security assurance. In: Proceedings of New Security Paradigms Workshop, Nova Scotia, Canada (2004)
15. Siponen, M., Baskerville, R., Kuivalainen, T.: Integrating security into agile development methods. In: Proceedings of Hawaii International Conference on System Sciences. (2005)
16. Poppendieck, M., Morsicato, R.: XP in a Safety-Critical Environment. Cutter IT Journal **15** (2002) 12–16
17. Kongsli, V.: Towards agile security in web applications. In: Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications. OOPSLA '06, New York, NY, USA, ACM (2006) 805–808
18. McGraw, G., Steven, J.: Software [In]security: Comparing Apples, Oranges, and Aardvarks (or, All Static Analysis Tools Are Not Created Equal) (2011)
19. Jensen, J.: A Novel Testbed for Detection of Malicious Software Functionality. In: Proceedings of Third International Conference on Availability, Security, and Reliability (ARES 2008). (2008) 292–301
20. Miller, B., Fredriksen, L., So, B.: An empirical study of the reliability of unix utilities. Communications of the ACM **33**(12) (1990)
21. McGraw, G., Chess, B., Migues, S.: Building Security In Maturity Model (BSIMM 3) (2011)
22. Doyle, A.C.: Memoirs of Sherlock Holmes. http://www.gutenberg.org/files/834/834-h/834-h.htm.
23. McGraw, G.: Software Security: Building Security In. Addison-Wesley (2006)