Open Source Software Projects Needing Security Investments

David A. Wheeler

Samir Khakimov

2015-03-17



INSTITUTE FOR DEFENSE ANALYSES

4850 Mark Center Drive  
ALEXANDRIA, VIRGINIA 22311-1882

Executive Summary

The Heartbleed vulnerability in the open source software (OSS) program OpenSSL was a serious vulnerability with widespread impact. It highlighted that some OSS is widely used and depended on, and that vulnerabilities in them can have serious ramifications, and yet some OSS have not received the level of security analysis appropriate to its importance. Some OSS projects have many participants, perform in-depth security analyses, and produce software that is widely considered to have high quality and strong security. However, some other OSS projects have small teams that have limited time to do the tasks necessary for strong security.

The Linux Foundation (LF) Core Infrastructure Initiative (CII) is trying to identify OSS projects that need special focus/help for security, so that it can best identify OSS projects needing investment. Similarly, the DHS HOST program’s goal is to “help facilitate the continued adoption of open technology solutions (including OSS) … to improve system security…” They have asked us to identify and collect metrics to help identify OSS projects that may especially need investment for security.

We have focused on automatically gathering metrics, especially those that suggest less active projects. We also provided a human estimate of the program’s exposure to attack, and developed a scoring system to heuristically combine these metrics. These heuristics identified especially plausible candidates. We then took those candidates, examined them further, and then identified a subset that we believe are especially concerning. For our initial set of projects to examine we took the set of packages installed by Debian base, and added a set of packages that we or others identified as potentially concerning; we could easily add more projects to consider in the future.

This is *not* a formal IDA document (IDA Paper or IDA document), nor has it been formally reviewed. This is an informal deliverable that provides information captured as part of our process to help identify open source software (OSS) projects that may need investment for security. It captures a brief literature search of ways to try to measure this, then describes results we have captured so far.

This document is released under the Creative Commons Attribution 4.0 International (CC BY 4.0) license. The supporting software (in Python) for capturing data is released under the MIT license. Some supporting data was sourced from the Black Duck Open HUB (formerly Ohloh), a free online community resource for discovering, evaluating, tracking and comparing open source code and projects.

Contents

[1. Introduction 1-1](#_Toc414364602)

[2. Some Past and Current Efforts to Identify Relevant OSS Project Metrics 2-1](#_Toc414364603)

[A. OSS metrics data sources 2-1](#_Toc414364604)

[1. Black Duck Open Hub (formerly Ohloh) 2-2](#_Toc414364605)

[2. OSS Repository statistics (GitHub, SourceForge, git, etc.) 2-3](#_Toc414364606)

[3. Linux distribution repositories 2-3](#_Toc414364607)

[4. FLOSSmole 2-4](#_Toc414364608)

[5. FLOSSMetrics project 2-4](#_Toc414364609)

[6. FLOSS Community Metrics Meeting 2-4](#_Toc414364610)

[7. Rodriguez survey of software data repositories 2-6](#_Toc414364611)

[8. PROMISE 2-7](#_Toc414364612)

[B. Methods for evaluating OSS projects 2-7](#_Toc414364613)

[1. Stol and Babar 2-7](#_Toc414364614)

[2. QualiPSo OpenSource Maturity Model (OMM) 2-8](#_Toc414364615)

[3. QSOS 2-9](#_Toc414364616)

[4. SQO-OSS / Spinellis et al 2-9](#_Toc414364617)

[5. Ghapanchi’s taxonomy for measuring OSS project success 2-11](#_Toc414364618)

[6. Wheeler OSS Evaluation model 2-13](#_Toc414364619)

[7. Doomed to FAIL index 2-14](#_Toc414364620)

[8. Internet Success 2-15](#_Toc414364621)

[C. Specific potentially-useful security metrics 2-17](#_Toc414364622)

[1. In-depth static analysis security tools (e.g., Coverity Scan) 2-17](#_Toc414364623)

[2. Lexically-scanning static analysis security tools (e.g., flawfinder and RATS) 2-18](#_Toc414364624)

[3. Wikipedia article on OSS Security 2-19](#_Toc414364625)

[4. Common Vulnerabilities and Exposures (CVE) count 2-19](#_Toc414364626)

[5. Schryen and Kadura 2-20](#_Toc414364627)

[6. Wheeler “Look at the Numbers” 2-21](#_Toc414364628)

[7. Presence of security test suite 2-23](#_Toc414364629)

[8. Presence/absence of past security reviews 2-24](#_Toc414364630)

[D. Specific potentially-useful general metrics 2-24](#_Toc414364631)

[1. Software quality metric measurement tools 2-24](#_Toc414364632)

[2. Compiler warning flags and style checkers 2-25](#_Toc414364633)

[3. Senior Defect Prediction measures: McCabe (Cyclomatic complexity) and Halstead 2-25](#_Toc414364634)

[4. Test coverage 2-25](#_Toc414364635)

[5. Source lines of code (SLOC) 2-26](#_Toc414364636)

[6. Linke survey 2-26](#_Toc414364637)

[7. Estimating Commit Sizes Efficiently (Hoffmann and Riehle) 2-27](#_Toc414364638)

[8. Gao: Choosing software metrics 2-27](#_Toc414364639)

[9. Menzies: Assessing Predictors of software defects 2-28](#_Toc414364640)

[10. Wang – How many software metrics for defect prediction 2-28](#_Toc414364641)

[11. Punitha – Software defect prediction 2-28](#_Toc414364642)

[12. COQUALMO 2-29](#_Toc414364643)

[13. DoD/Industry 1994 Survey 2-29](#_Toc414364644)

[E. Attack surface measurement 2-30](#_Toc414364645)

[F. Kenn White metrics set 2-31](#_Toc414364646)

[G. IDA ideas 2-31](#_Toc414364647)

[1. Exposure to attack 2-31](#_Toc414364648)

[2. Other work 2-32](#_Toc414364649)

[H. London January 2015 Meeting 2-33](#_Toc414364650)

[I. Additional areas (“TODO”) 2-35](#_Toc414364651)

[J. Comments on metrics 2-35](#_Toc414364652)

[3. Important OSS Projects 3-1](#_Toc414364653)

[A. Common in distributions 3-1](#_Toc414364654)

[B. Linux Foundation original list 3-2](#_Toc414364655)

[C. Kenn White list 3-3](#_Toc414364656)

[D. Google Application Security Patch Reward Program 3-4](#_Toc414364657)

[E. Recent problem reports 3-5](#_Toc414364658)

[F. Augmented list of programs 3-5](#_Toc414364659)

[4. Selected approach and early results 4-1](#_Toc414364660)

[A. Overall approach 4-1](#_Toc414364661)

[B. Caveats 4-1](#_Toc414364662)

[C. First stage 4-2](#_Toc414364663)

[D. Second stage 4-4](#_Toc414364664)

[5. Current process 5-1](#_Toc414364665)

[A. Current simplified approach 5-1](#_Toc414364666)

[B. Risk index 5-2](#_Toc414364667)

[6. Current process results 6-4](#_Toc414364668)

[A. Riskiest OSS programs (straight from scores) 6-4](#_Toc414364669)

[B. Riskiest OSS programs (estimated by us) 6-5](#_Toc414364670)

[C. Potential improvements to other sites to improve data 6-10](#_Toc414364671)

[7. Conclusions 7-1](#_Toc414364672)

[Table 1. Riskiest OSS Programs (straight from scores) 6-4](#_Toc414364673)

[Table 2. Riskiest OSS programs (human estimate informed by risk measures) 6-6](#_Toc414364674)

# Introduction

The Heartbleed vulnerability in the open source software (OSS) program OpenSSL was a serious vulnerability with widespread impact. Yet there are many ways that Heartbleed could have been detected before it was deployed [Wheeler2014h]. The Heartbleed vulnerability highlighted that some OSS is widely used and depended on, and that vulnerabilities in them can have serious ramifications, and yet they have not received the level of security analysis appropriate to its importance. Some OSS projects have many participants, perform in-depth security analyses, and produce software that is widely considered to have high quality and strong security. However, some other OSS projects have small teams that have limited time to do the tasks necessary for strong security (e.g., the OpenSSL project before Heartbleed).

Two groups have supported this work: the Linux Foundation and the Homeland Open Security Technology (HOST) program.

The Linux Foundation (LF) Core Infrastructure Initiative (CII) is trying to identify OSS projects that need special focus/help for security, so that it can invest in the most important projects that most need help. To achieve this, the LF CII wants to identify metrics for determining what OSS projects that most need security audits & related help, and then use that information to help them wisely select projects to invest in. The LF CII makes final decisions of what it will invest in, but has asked for help in identify appropriate metrics and their values.

The HOST program’s goal, as stated in the statement of work between IDA and the Georgia Tech Research Institute (GTRI), is to “help facilitate the continued adoption of open technology solutions (including OSS) within federal, state, and municipal public sector IT environments in order to improve system security…” The program has asked IDA to “provide continued subject matter expertise on in-depth research, studies, and analysis on the research domain” for HOST. The HOST project is in turn funded by the Department of Homeland Security (DHS). This is in support of securing the nation, e.g., executive order 13636 states that, “It is the policy of the United States to enhance the security and resilience of the Nation’s critical infrastructure and to maintain a cyber environment that encourages efficiency, innovation, and economic prosperity while promoting safety, security, business confidentiality, privacy, and civil liberties” [Obama2013]. Thus, identifying OSS projects that need help is also in the interest of the HOST project.

Other organizations might be interested in this work as well. For example:

* The “Snowdrift coop” (at <https://snowdrift.coop/>) was established in late 2014 to create “a matching patronage system funding freely-licensed works” and in the future might be interested in funding work to improve the security of OSS projects.
* The “European Parliament has approved funding for several projects related to Free Software and privacy. In the EU budget for 2015, which the European Parliament adopted on December 17, the Parliamentarians have allocated up to one million Euro for a project to audit Free Software programs in use at the Commission and the Parliament in order to identify and fix security vulnerabilities.” <https://fsfe.org/news/2014/news-20141219-01.en.html>
* The “Google Application Security Patch Reward Program” rewards proactive security improvements to select open-source projects. See <https://www.google.com/about/appsecurity/patch-rewards/>

The goal of this work was to perform a quick reaction study to gather data to help make reasonable decisions in a very short time. Further work might identify significantly improved measures and additional projects to be examined, but it was judged to be better to do a quick study with limited time (and document it) than to simply guess, or to delay for a long time for a more comprehensive study.

This document is not a formal IDA document or paper; it is a result of a brief and rapid analysis under a relatively short deadline. However, it uses the formal IDA format, to make it easy to transform it into a formal IDA document should that be desired in the future.

This document primarily uses the term “open source software” (OSS) for software that can be studied, used for any purpose, modified, and redistributed (modified or not). Other terms for such software including “Free software” (note the capital letter) and “Free/libre/open source software” (FLOSS). See the Open Source Definition (from the Open Source Initiative) and the Free Software Definition (from the Free Software Foundation) for details. In some cases the users of these different terms emphasize different motivations and purposes, but since we are simply focused on the software *resulting* from these efforts (instead of the motivations for development), we will ignore those distinctions in this paper. We will use “proprietary software” and “closed software” as antonyms for OSS. Note that “in almost all cases, OSS meets the definition of ‘commercial computer software’ under U.S. law [DoD2009] and that many OSS programs are co-developed and supported by commercial companies.

Per agreement by both GTRI and the Linux Foundation, this document is released under the Creative Commons Attribution 4.0 International (CC BY 4.0) license; the supporting software (in Python) for capturing data is released under the MIT license. Thus, they are both “Free Cultural Works” as defined by freedomdefined.org.

Section 2 lists past work identifying relevant ways to measure OSS projects. Section 3 is a list of especially-promising metrics, based on section 2, for measuring OSS projects’ need for security investment. Section 4 identifies important OSS projects that are widely used, yet might need investment. We are developing software to capture and combine this data into a separate spreadsheet for developing the recommendations, based on this document.

# Some Past and Current Efforts to Identify Relevant OSS Project Metrics

This section describes a brief survey (literature search) of some past efforts to identify relevant metrics of OSS projects. Our goal was to help identify metrics that might help identify projects needing investment.

Measuring the security of software is a notoriously difficult and essentially unsolved problem. Ideally we would identify metrics that directly determine if some OSS project is producing secure software or not. However, since perfect metrics are not available, we are instead interested in metrics that provide some evidence that a project’s product is more or less likely to be secure. Some product measures, for example, may suggest that the software has fewer security defects, or at least fewer defects in general. Other metrics examine the OSS project (including its processes) and may suggest that an OSS project is in trouble, e.g., it is relatively inactive, has few active contributors, or that much development was done long ago (when fewer developers knew how to develop secure software). For example, it is often noted that before Heartbleed, OpenSSL had relatively few developers and that many bug reports languished without response for long periods of time. These indicators may suggest that a project may need investment to make its software adequately secure.

Sources include surveys of OSS, existing evaluation processes for evaluating OSS, surveys of quality or security metrics (e.g., [Shaikh2009]) and organizations that track OSS metrics.

There was not time to do a complete survey, but we believe it is better to do a brief survey (and document it) than ignore the large set of materials available. These materials are probably not equally useful or credible; the goal was simply to survey various options to reduce the risk of overlooking especially useful sources of information. Some odd or improbable approaches might suggest a new and useful approach.

## OSS metrics data sources

It is much easier to get data from organizations that measure and curate it than to try to extract it for each program separately. There is also the hope that such organizations will try to select useful measures. Black Duck Open Hub (formerly Ohloh), in particular, provides relatively current data for many programs in an easily-obtained form.

### Black Duck Open Hub (formerly Ohloh)

Black Duck Open Hub, formerly Ohloh, maintains an active set of metrics data for a variety of OSS projects at <https://www.openhub.net> along with a nice UI for viewing them.

Looking at a sample project, such as Firefox, helps give a sense of what it records. The entry for Firefox (<https://www.openhub.net/p/firefox>) reports:

“In a Nutshell, Mozilla Firefox...

* has had 223,200 commits made by 3,187 contributors representing 12,554,058 lines of code
* is mostly written in C++ with a low number of source code comments [as a percentage compared to other programs in the same programming language]
* has a well established, mature codebase maintained by a very large development team with increasing Y-O-Y commits
* took an estimated 3,920 years of effort (COCOMO model) starting with its first commit in April, 2002, and ending with its most recent commit 26 days ago.”

It reports, for both 30 day and 12 month periods, the number of commits, the number of contributors (including a separate number for new contributors), and for the 12-month period it also reports the change from the previous 12-month period. It also includes user ratings.

It also provides “quick reference” information (such as the organization name), and some of that can also indicate the health of a project:

* Link(s) for Homepage, Documentation, Download, Forums, Issue Trackers, Code: Where present, these are signs of an active project
* Licenses: If these licenses are OSI and FSF approved, and especially if they are common, these are good signs. Unusual licenses can inhibit contribution.

This is a well-maintained site that with programmatic interfaces that make it easy to get the data they collect. The programmatic interfaces in some cases only have general statements (e.g, “mature codebase” or “very large development team”) instead of specific numbers, but these general statements can still be valuable.

A (gratis) key must be acquired for programmatic queries, and the website states that queries are limited to 1000 queries/day for each key (though this might not be enforced). We cached results to avoid creating a nuisance. However, they also imposed other conditions, so we arranged for a special exception with Black Duck for use in this project.

### OSS Repository statistics (GitHub, SourceForge, git, etc.)

Many OSS projects are hosted on a relatively few number of hosting sites that can also report a variety of statistics. If the repository directly provides that data, then the data is especially easy to get for such projects.

GitHub provides a variety of statistics and charts, particularly ones focused on project activity. Selecting “pulse” on a project’s project site reports for a selected time period the number of (direct) authors, commits, files, and number of additions and deletions (counted by lines). You can also select “graphs” to see a variety of graphs. The issue tracker can report number of open and closed issues. Programs can get information programmatically; see <https://developer.github.com/v3/repos/statistics/>

SourceForge switched to the OSS Allura software for repository management. “Tickets/View Stats” reports a variety of statistics, including number of tickets (total, open, closed), number of new tickets over various periods (7 days, 14 days, 30 days), number of comments on tickets, number of new comments on tickets over given period. They also support a REST API for obtaining this information for programs (most data is returned in JSON format); more information is at <https://sourceforge.net/p/forge/documentation/Allura%20API/>

Distributed version control software, including git, include a significant amount of metadata about commits because the project history is downloaded. Tools such as gitstats (<http://gitstats.sourceforge.net/>) can be used to quickly analyze this data and report additional information. Gitstats, for example, will report:

* General statistics: total files, lines, commits, authors.
* Activity: commits by hour of day, day of week, hour of week, month of year, year and month, and year.
* Authors: list of authors (name, commits (%), first commit date, last commit date, age), author of month, author of year.
* Files: file count by date, extensions
* Lines: Lines of Code by date

### Linux distribution repositories

Most Linux distributions (such as Debian, Ubuntu, Fedora, and Red Hat Enterprise Linux) use package managers to install (and uninstall) packages. These packages include metadata with important information, such as the software name, dependencies, and URL of the originating project. What’s more, if a package is installed in a distribution’s base or a widely-used group/task, it is likely to be widely used. Some distributions (such as Fedora) work to split up projects so that if they reuse software from another project, they are kept separate (so that security updates will properly update everything); this can help reveal important projects that might otherwise be hidden inside larger projects.

### FLOSSmole

FLOSSmole at <http://flossmole.org/> performs “collaborative collection and analysis of free/libre/open source project data” (per its front page). It is related to FLOSShub, a “portal for free/libre and open source software (FLOSS) research resources and discussion.”

It appears semi-active, with some datasets dated 2014, and various reports also dated 2014. However, some data is only available up through 2013. One challenge for them is that repositories are increasingly providing this information directly.

### FLOSSMetrics project

FLOSSMetrics stands for “Free/Libre Open Source Software Metrics” and is at <http://www.flossmetrics.org/>.

The main objective of FLOSSMETRICS is, per its website, “to construct, publish and analyse a large scale database with information and metrics about libre software development coming from several thousands of software projects, using existing methodologies, and tools already developed.”

It records various data for a variety of projects, e.g.:

* how many bugs are reported
* the average time it takes to fix a bug in a project’s lifetime

There is a “final report” in 2010 for this European project, and no obvious activity since then. Its database of projects at <http://melquiades.flossmetrics.org/projects> seems to have little activity since 2010.

Thus, this is likely to be no longer active. Active similar projects include FLOSSmole and Black Duck Open Hub (formerly Ohloh).

### FLOSS Community Metrics Meeting

The “FLOSS Community Metrics” meeting is a conference of those interested in collecting and analyzing OSS metrics, sponsored by Bitergia. Its website is at <http://flosscommunitymetrics.org/> ; they had a conference in July 2014 and another is expected in 2015.

The 2014 had several presentations that were interesting for measuring OSS quality; clicking on “slides” links provides the slides themselves.

Roberto Galoppini presented “You’re not entitled to your opinion about open source software!”, which proposed the following simple-to-collect metrics:

* Code Maturity [<1 year, 1-3 years, > 3 years]. Based on QSOS Age.
* Code stability (unstable, stable but old, stable and maintained)
* Project popularity (unknown, small but growing, well known)
* Case study availability
* Books availability
* Community management style
* Team size [1-5 members, 5-10 members, > 10 members] – can be found by analyzing commits
* Commercial support
* Training
* Documentation
* QA Process [n/a, existing but not supported by tools, supported by tools]
* QA tools [n/a, existing but not much used, very active use of tools]
* Bugs reactivity [poor, formalized but not reactive, formalized and reactive]
* Source [to be compiled, binaries available, virtual appliance available]
* Red Hat/Solaris/Windows
* Amount of comments [none, poorly commented, well commented]
* Computer language used [more than 3 languages used, 1 language primarily, 1 unique language]
* Code modularity [not modular, modular, available tools to create extensions]
* License
* Modifiability [no way to propose modification, tools to access and modify code available but the process is not well defined, tools and procedures to propose modifications available.]
* Roadmap [n/a, no detailed roadmap available, detailed roadmap available]
* Sponsor

James Faulkner (Liferay community manager) in “metrics are fun, but which ones really matter?” lists various metrics, including these “more interesting” ones:

* Time of bug report to fix
* Time from forum question to answer
* Number of ignored contributions
* Time from contribution to insertion in codebase

He also lists “basic” ones such as number of contributions, number of commits/lines, number of authors, number of bug reports, number of forum posts/answers, number of downloads, number of ignored messages, number of open tickets / code reviews.

There is reference to vizGrimoire, an OSS toolset and framework to analyze and visualize data about software development, available at <http://vizgrimoire.bitergia.org/> promoted by Bitergia.

### Rodriguez survey of software data repositories

Rodriguez et al’s “On Software Engineering Repositories and their Open Problems” describes various sources of data about software [Rodriguez2012]. They identified the following set:

* FLOSSMole: http://flossmole.org/
* FLOSSMetrics: http://flossmetrics.org/
* PROMISE (PRedictOr Models In Software Engineering): http://promisedata.org/
* Qualitas Corpus (QC): http://qualitascorpus.com/
* Sourcerer Project: http://sourcerer.ics.uci.edu/
* Ultimate Debian Database (UDD): http://udd.debian.org/
* Bug Prediction Dataset (BPD): http://bug.inf.usi.ch/
* International Software Benchmarking Standards Group (ISBSG): http://www.isbsg.org/
* Eclipse Bug Data (EBD) http://www.st.cs.uni-saarland.de/softevo/bug-data/eclipse/Software-artifact
* Infrastructure Repository (SIR): http://sir.unl.edu/
* Ohloh : http://www.ohloh.net/
* SourceForge Research Data Archive (SRDA): http://zerlot.cse.nd.edu/
* Helix Data Set: http://www.ict.swin.edu.au/research/projects/helix/
* Tukutuku: http://www.metriq.biz/tukutuku/

### PROMISE

PROMISE standards for “PRedictOr Models In Software Engineering”; its main website is <http://promisedata.org/> but its dataset is at <https://code.google.com/p/promisedata/>. This is a collection of data “real world software engineering projects… whatever data is available.” This is useful for tasks such as research into predictive metrics. However, it is a collection of *available* real-world data, not primarily *current* real-world data, and by itself it does not identify metrics that are necessarily relevant (just what it could provide).

The list of data specific to defects is at: <https://code.google.com/p/promisedata/w/list?q=label:Defect>

The PROMISE 2014 (“PROMISE ’14”) conference included various presentations, including a keynote by Audris Mockus (available <http://mockus.org/papers/promise1.pdf>) about the problems of acquiring data and prediction.

## Methods for evaluating OSS projects

There are a number of complete processes specifically for evaluating OSS (as software, a project, or both). These are typically evaluating OSS for a particular purpose, not their security per se, but they may have useful approaches to mine.

Wikipedia includes a comparision of a few of them <http://en.wikipedia.org/w/index.php?title=Open-source_software_assessment_methodologies&oldid=579922098>.

A key attribute is whether or not they support comparision. OSMM Capgemini and QSOS, for example, do support comparison.

The Open Business Readiness Rating (OpenBRR) project, announced in 2005, did not create a community. Its website <http://www.openbrr.org/> claims to be preparing an update; until that occurs, it is not considered further.

### Stol and Babar

There are so many processes for evaluating OSS that Stol and Babar have published a framework comparing them. <http://staff.lero.ie/stol/files/2011/12/OSS2010.pdf>. One complication is that there are several methods all named “Open Source Maturity Model”.

### QualiPSo OpenSource Maturity Model (OMM)

The QualiPSo OpenSource Maturity Model (OMM) a methodology for assessing Free/Libre Open Source Software (FLOSS) and more specifically the FLOSS development process. This methodology was released in 2008 and is released under the Creative Commons license.

A summary is at <http://en.wikipedia.org/wiki/OpenSource_Maturity_Model>.

It defines three maturity levels: basic, intermediate, and advanced.

Basic level requirements:

* PDOC – Product Documentation
* STD – Use of Established and Widespread Standards
* QTP – Quality of Test Plan
* LCS – Licenses
* ENV – Technical Environment
* DFCT – Number of Commits and Bug Reports
* MST – Maintainability and Stability
* CM – Configuration Management
* PP1 – Project Planning Part 1
* REQM – Requirements Management
* RDMP1 – Availability and Use of a (product) roadmap

Intermediate level requirements:

* RDMP2 – Availability and Use of a (product) roadmap
* STK – Relationship between Stakeholders
* PP2 – Project Planning Part 2
* PMC – Project Monitoring and Control
* TST1 – Test Part 1
* DSN1 – Design Part 1
* PPQA – Process and Product Quality Assurance

Advanced level requirements:

* PI – Product Integration
* RSKM – Risk Management
* TST2 – Test Part 2
* DSN2 – Design 2
* RASM – Results of third party assessment
* REP – Reputation
* CONT – Contribution to FLOSS Product from SW Companies

Unfortunately, I have had trouble accessing <http://www.qualipso.org/> for more information.

### QSOS

Qualification and Selection of Opensource Software (QSOS) dates from 2004; QSOS 2.0 was released June 2013. It is a “free project aiming to mutualize and capitalize technological watch on open source components and projects”; its main page is <http://www.qsos.org>. The method is distributed under the GNU Free Documentation License. Its goal is to provide a “formal process to evaluate, compare and select open source solutions.”

QSOS is a Drakkr project (<http://www.drakkr.org>); others include FLOSC (Free/Libre Open Source Complexity), a project providing method and tools to evaluate intrinsic complexity of open source components. Some information is available on GitHub <https://github.com/drakkr/drakkr>.

Unfortunately the site <http://master.o3s.qsos.org/> has been down so it is not obvious what information is currently available.

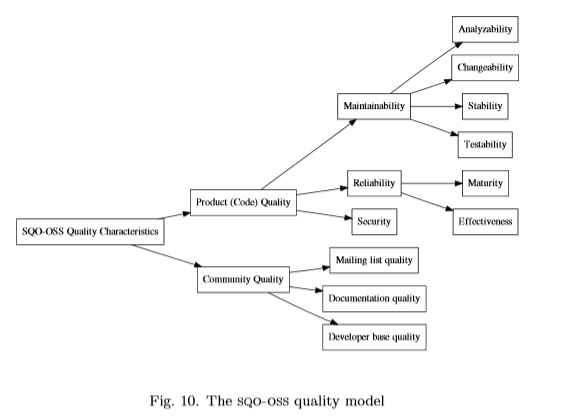
### SQO-OSS / Spinellis et al

“Evaluating the Quality of Open Source Software” [Spinellis2009] presents “motivating examples, tools, and techniques that can be used to evaluate the quality of open source… software”. It includes a “technical and research overview of [Software Quality Observatory for Open Source Software (SQO-OSS)], a cooperative research effort aiming to establish a software quality observatory for open source software.”

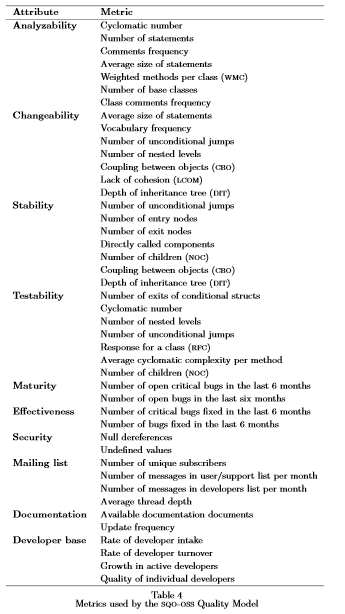
They noted several metrics:

* Use of various scanning tools including PMD (Java scanner), FindBugs (for Java), Checkstyle (Java style checker), Sonar, ESX (for C++), and Scan by Coverity.
* Use of metric suites such as Ohloh and Sourcekibitzer (the latter is for Java).
* Adherence to claimed coding style as a proxy for quality. They formatted FreeBSD code using indent and computed the number of lines that changed.
* Mean developer engagement (MDE), the average percentage of active developers who work on a project each week. Developers who stay inactive are eventually considered no longer part of the total.
* Cross-language metric tool, which collects metrics such as number of public attributes, number of children, etc.
* Metric for developer contributions. This adds measures for not just lines of code, but also for bug closing, documentation files, updating a wiki page, etc. Some more difficult to measure, e.g., “participate in a flamewar”.

It notes the SQO-OSS quality model:



And uses the following metrics to estimate them:



They rank various projects by comparing values to ideal values, e.g., the ideal candidate for the Excellent Analyzability quality attribute should have a McCabe Cyclomatic number equal to 4, Average function’s number of statements equal to 10, Comments frequency equal to 0.5 and average “size of statements” equal to 2.

### Ghapanchi’s taxonomy for measuring OSS project success

Ghapanchi et al’s “A taxonomy for measuring the success of open source software projects” [Ghapanchi2011] used a literature survey focused on measuring OSS project success. After identifying 154 publications in their initial set, they narrowed it down to 45 publication and categorized them into meaningful clusters.

They identified six broad areas that can lead to success:

1. Product quality. Different researchers have proposed many different measures for product quality (leading to product success). They report that [Crowston2003] is among the most cited; this is a content analysis of an online focus group that reported 7 main themes for OSS success: user, product, process, developers, use, recognition, and influence.
2. Project performance: These combine efficiency and effectiveness.
3. Project effectiveness: These attempt to measure “getting the right things done.”
4. Project efficiency: These determine the extent to which a project uses its resources to generate outcomes, typically using Data Envelopment Analysis (DEA). The goal is to determine Output/Input. Examples of input/output pairs are {number of developers, bug submitters}/{KiB per download, number of downloads, project rank}; {number of downloads, number of years} / {product size in bytes, number of code lines} ; {product size (bytes), development status} / {number of developers, product age}.
5. Project activity: This is “frequently regarded as one of the pillars of OSS project success.” Examples include the how frequently defects are fixed, new releases of the software posted, or support requests answered, often over some period of time.
6. User interest: The ability of an OSS project to attract community members to adopt the software (its popularity).

They note that “success” can be measure for both the product and the project, so they then map these areas to these kinds of success. User interest affects both product success and project success. They map product quality to primarily product success, while project activity, project efficiency, and project effectiveness map to project success. Project effectiveness and project efficiency themselves affect project performance.

Based on this survey, they provide a “practical list of OSS success metrics.”

|  |  |
| --- | --- |
| Aspect | Useful measures according to [Ghapanchi2011] |
| User interest | Traffic on the project Web site, downloads of the code, number of developers who have joined the project team, and the number of people who have registered in project mailing list to receive announcements such as new release regarding a project |
| Project activity | The number of software releases, number of patches, number of source code lines, number of code commits |
| Project effectiveness | The percentage of task completion (bug fix, feature request, and support request), number of developers the project has attracted, number of work weeks spent on the project |
| Project efficiency | Using a DEA model with one or some input indicators (e.g., number of developers, number of bug submitters, number of years, product size (bytes), development status) and one or some output indicators (e.g. kilobytes per download, number of download, project rank, product size in bytes, number of code lines) |
| Product quality | Code quality, documentation quality, understandability, consistency, maintainability, program efficiency, testability, completeness, conciseness, usability, portability, functionality, reliability, structuredness, meeting the requirements, ease of use, user friendliness |

[Ghapanchi2011]’s “practical list of OSS success measures”

They make an interesting observation that the kinds of data available for OSS are typically different than for proprietary software. “Traditional [closed source] software development success models frequently focus on success indicators such as system quality, use, user satisfaction and organizational impacts [that are] more related to the ‘use environment’ of the software, while studies on OSS success tend to look more at the ‘development environment’… [in traditional models] ‘development environment is not publicly available but the ‘use environment is less difficult to study, while in OSS the ‘development environment’ is publicly visible but the ‘use environment’ is hard to study or even to identify.”

One challenge for us is that we are primarily interested in projects that are *successful* in terms of widespread adoption and satisfaction of functional requirements, yet have serious *vulnerabilities*. These measures do not necessarily directly relate to our question, but can perhaps help suggest projects that are not adequately active.

### Wheeler OSS Evaluation model

David A. Wheeler (also author of this paper) has previously described a general process for evaluating open source software in “How to Evaluate Open Source Software / Free Software (OSS/FS) Programs.” [Wheeler2011e].

This process is based on four steps: Identify candidates, Read existing reviews, Compare the leading programs' basic attributes to your needs, and then Analyze the top candidates in more depth. This set of Identify, Read Reviews, Compare, and Analyze can be abbreviated as "IRCA". Important attributes to consider include functionality, cost, market share, support, maintenance, reliability, performance, scaleability, useability, security, flexibility/customizability, interoperability, and legal/license issues.

Its section on security mentions some metrics that might be useful:

* Coverity scan results, including the rung achieved, number of defects, and defect density.
* Fortify scan results (similar)
* Common criteria evaluation. These typically evaluate entire systems (e.g., entire operating systems), instead of focusing on specific projects that support a particular portion of an operating system, and thus do not provide the kinds of measures desired for this task.
* Reports of (many) vulnerabilities that are “unforgiveable” (MITRE identifies criteria for identifying vulnerabilities that are especially easy to find, and thus “unforgiveable”) [Christey2007].
* It is *known* that external organizations have or are reviewing the software, e.g., OpenBSD. However, some organizations (like OpenBSD) only fix *their* version; that version may not be the version all other systems use.

It also notes that experts can be hired to determine if the developers follow good security practices. Examples include:

* it minimizes privileges (e.g., only small portions of the program have special privilege, or the program only has special privileges at certain times)
* it strives for simplicity (simpler designs are often more secure)
* it carefully checks inputs
* source code scanning tools such as RATS and Flawfinder report few problems.

Its section on reliability notes metrics that may be useful are:

* Self-reported status (e.g., “mature”)
* Presence of an automated (regression) test suite

### Doomed to FAIL index

Tom “spot” Callaway, Fedora Engineering Manager at Red Hat, posted a blog post titled “How to tell if a FLOSS project is doomed to FAIL (or at least, held back...)” in 2009. The book “The Open Source Way” includes a chapter with an updated version of this index and is available online [Callaway]. This index is intended to be a quick measure of how well a FLOSS project follows common practices, particularly those that impede packaging or co-development by others. It measures “FAIL” points, so *low* scores are better; 0 is perfect, 5-25 is “You're probably doing okay, but you could be better,” and beyond 25 is an indicator of serious problems.

The measures are grouped into categories: size, source (version) control, building from source, bundling, libraries, system install, code oddities, communication, releases, history, licensing, and documentation. Examples of causes for fail points are:

* Source Control: There is no publicly available source control (e.g. cvs, svn, bzr, git) [ +10 points of FAIL ]
* Building from source: There is no documentation on how to build from source [ +20 points of FAIL ]
* Communication: Your project does not have a mailing list [ +10 points of FAIL ], or your project does not have a website [ +50 points of FAIL]
* Licensing: Your code does not have per-file licensing [ +10 points of FAIL ]

Obviously a high score does not always doom a project to fail, nor does a low score guarantee success. However, like any metric, it can provide a simple metric to point out potential issues in an OSS project. It is intentionally designed to produce a numerical score, making it relatively easy to report.

### Internet Success

The book *Internet Success* by Schweik and English reports a detailed quantitative analysis to determine “what factors lead some OSS commons to success and others to abandonment?” [Schweik2012]

Schweik and English examined over 100,000 projects on SourceForge, using data from SourceForge and developer surveys, using quantitative analysis instead of guesswork. They use a very simple project lifecycle model — projects begin in initiation, and once the project has made its first software release, it switches to growth. They also categorized projects as success, abandonment, or indeterminate. Combining these produces 6 categories of project: success initiation (SI); abandonment initiation (AI); success growth (SG); abandonment growth (AG); indeterminant initiation (II); and indeterminant growth (IG). Their operational definition of success initiation (SI) is oversimplified but easy to understand: an SI project has at least one release. Their operational definition for a success growth (SG) project is very generous: at least 3 releases, at least 6 months between releases, and has more than 10 downloads.

One of the key results is that during initiation (before first release), the following are the most important issues, in order of importance, for success in an OSS project according to this quantitative data:

1. “Put in the hours. Work hard toward creating your first release.” The details in chapter 11 tell the story: If the leader put in more than 1.5 hours per week (on average), the project was successful 73% of the time; if the leader did not, the project was abandoned 65% of the time. They are not saying that leaders should only put in 2 hours a week; instead, the point is that the leader must consistently put in time for the project to get to its first release.
2. “Practice leadership by administering your project well, and thinking through and articulating your vision as well as goals for the project. Demonstrate your leadership through hard work…”
3. “Establish a high-quality Web site to showcase and promote your project.”
4. “Create good documentation for your (potential) user and developer community.”
5. “Advertise and market your project, and communicate your plans and goals with the hope of getting help from others.”
6. “Realize that successful projects are found in both GPL-based and non-GPL-compatible situations.”
7. “Consider, at the project’s outset, creating software that has the potential to be useful to a substantial number of users.” Remarkably, the minimum number of users is surprisingly small; they estimate that successful growth stage projects typically have at least 200 users. In general, the more potential users, the better.

Some items that people have claimed are important, such as keeping complexity low, were not really supported as important. In fact, successful projects tended to have a little more complexity. We suspect both successful and abandoned projects often strive to reduce complexity — so it not really something that distinguishes them — and that sometimes a project that focuses on user needs has to have more complexity than one that does not, simply because user needs can sometimes require some complexity.

Similarly, they had guidance for growth projects, in order of importance, and these may suggest some metrics:

1. “Your goal should be to create a virtuous circle where others help to improve the software, thereby attracting more users and other developers, which in turn leads to more improvements in the software…” Do this the same way it is done in initiation: spending time, maintain goals and plans, communicate the plans, and maintain a high-quality project web site.” The user community should actively interacting with the development team.

(Wheeler notes that possible related metrics include: Actively-maintained website (e.g., date of last page change on website), messages/month (e.g., email, bug tracker, etc.), number of commits/month, number of committers, etc.)

1. “Advertize and market your project.” In particular, successful growth projects are frequently projects that have added at least one new developer in the growth stage.

(Wheeler notes that possible related metrics include number of developers that have been added (post initial release or within a year).)

1. Have some small tasks available for contributors with limited time.

(Wheeler notes that a possible metric is a posted list of small tasks for new/limited contributors.)

1. Welcome competition. The authors were surprised, but noted that “competition seems to favor success”. Personally, I do not find this surprising at all. Competition often encourages others to do better; we have an entire economic system based on that premise.
2. Consider accepting offers of financing or paid developers (they can greatly increase success rates). This one, in particular, should surprise no one — if you want to increase success, pay someone to do it.
3. “Keep institutions (rules and project governance) as lean and informal as possible, but do not be afraid to move toward more formalization if it appears necessary.”

The book has more detailed lists of metrics.

They also have some hints of how potential OSS users (consumers) can choose OSS that is more likely to endure. Successful OSS projects have characteristics like more than 1000 downloads, users participating in bug tracker and email lists, goals/plans listed, a development team that responds quickly to questions, a good web site, good user documentation, and good developer documentation. A larger development team is a good sign, too.

## Specific potentially-useful security metrics

Many metrics have been proposed for evaluating software that are more security-focused. Some are focused on security.

### In-depth static analysis security tools (e.g., Coverity Scan)

Some tools are specifically designed to look for potential security vulnerabilities and report them. Their sheer counts, perhaps limited to most severe and/or computed as densities, might give an indication of the security (or lack thereof) of software.

Coverity sells a proprietary tool that looks for security vulnerabilities. Coverity Scan, at <https://scan.coverity.com/>, is “a service by which Coverity provides the results of analysis on open source coding projects to open source code developers that have registered their products with Coverity Scan.” It supports C, C++, Java, and C#. An OSS project developer must specifically register their project to participate; results are then sent to the project developers.

The Coverity Scan project was initially launched under a contract with the Department of Homeland Security (DHS) to harden open source software which provides critical infrastructure for the Internet. Coverity Scan began in collaboration with Stanford University on March 6, 2006. During the first year of operation, over 6,000 software defects were fixed across 50 C and C++ projects by open source developers using the analysis results from the Coverity Scan service. DHS support ended in 2009, but the service has continued.

A list of projects covered by Coverity scan is at <https://scan.coverity.com/projects>; over 3200 participate. Even though the exact results are not posted publicly, the fact that *a project is on the list* maintained by Coverity is public, and that may by itself indicate that a project is interested in detecting and fixing vulnerabilities. A few projects have achieved “rung 2” which is a higher achievement.

A similar argument could apply to other tool makers who make tools that perform in-depth static analysis of software and provide scans of OSS projects. For example, HP/Fortify will provide static analysis tools for use in examining open source software, in partnership with Sonatype; details are here: <https://www.hpfod.com/open-source-review-project>.

There are some OSS tools that look for vulnerabilities, e.g., splint (for C only), that might perform such a role. (Note, however, that splint has not been maintained recently.)

Note that these tools use heuristics to determine what is a vulnerability, thus, different tools will report different values.

### Lexically-scanning static analysis security tools (e.g., flawfinder and RATS)

A variant is to use lexically-scanning tools to report constructs (“hits”) in software that are of special concern. Again, counts or densities could be reported. OSS tools such as flawfinder and RATS can do this. (Note: David A. Wheeler is the author of flawfinder.)

IDA has previously done in-house work measuring hit density, where hits are reports from flawfinder or another lexical tool, and the density is found by dividing by physical source lines of code. These tools simply report riskier constructs, not really vulnerabilities, but the theory is that if developers often use riskier constructs, they are more likely to produce insecure results. A comparison of sendmail and postfix of years ago suggests this might be a useful measure.

### Wikipedia article on OSS Security

Wikipedia’s article on “Open-Source Software security” has various comments about OSS security, including references to metrics and models. The following is from the page <http://en.wikipedia.org/w/index.php?title=Open-source_software_security&oldid=627231105> (which is a permanent link).

Metrics they mention include:

* Number of days between vulnerabilities. “It is argued that a system is most vulnerable after a potential vulnerability is discovered, but before a patch is created. By measuring the number of days between the vulnerability [being found] and when the vulnerability is fixed, a basis can be determined on the security of the system. There are a few caveats to such an approach: not every vulnerability is equally bad, and fixing a lot of bugs quickly might not be better than only finding a few and taking a little bit longer to fix them, taking into account the operating system, or the effectiveness of the fix.”
* Morningstar model. “By comparing a large variety of open source and closed source projects a star system could be used to analyze the security of the project similar to how Morningstar, Inc. rates mutual funds. With a large enough data set, statistics could be used to measure the overall effectiveness of one group over the other. An example of such as system is as follows:[7]
  + 1 Star: Many security vulnerabilities.
  + 2 Stars: Reliability issues.
  + 3 Stars: Follows best security practices.
  + 4 Stars: Documented secure development process.
  + 5 Stars: Passed independent security review.”
* Coverity (see discussion on Coverity)

### Common Vulnerabilities and Exposures (CVE) count

MITRE maintains Common Vulnerabilities and Exposures (CVE), a dictionary of publicly known information security vulnerabilities and exposures. MITRE and the NIST National Vulnerability Database (NVD) at <https://nvd.nist.gov/> maintain information about publicly-known vulnerabilities in released software, including OSS. Not all publicly-known vulnerabilities are assigned CVEs, but this is nevertheless a widely-used starting point for information about vulnerabilities.

Some obvious metrics suggest themselves:

* Number of CVEs assigned to each particular OSS project, perhaps over some fixed period (say 3 years).
* Number of CVEs with high severity (this is a subset of the whole). The NVD reports CVSS scores for each CVE.
* “Density” version (e.g., by dividing by thousands of lines of code). Since larger projects will tend to have more vulnerabilities, a “density” version can compensate for this.
* Average (or median) number of days between CVE reports.

These have a number of well-understood drawbacks. What we want to know is the number of vulnerabilities *remaining*, while CVEs report on the number *found*. The current CVE count might be low because no one is looking, or high because substantial effort has been spent to find and report vulnerabilities after it has been released. Also, if projects undergo substantial changes the CVE counts from older versions may or may not be relevant. Still, if a project has a large number of CVEs, it *might* indicate that the project has not been sufficiently active in countering vulnerabilities.

### Schryen and Kadura

Guido Schryen and Rouven Kadura in 2009 wrote “Open source vs. closed source software: towards measuring security” [Schryen2009]. In the process they provided summaries of previous work to measure security of OSS in section 3, “review: quantitative models” and provide a metric for measuring software (in their case, to measure responsiveness to vulnerability reports).

The spend time discussing various models based on security breaches (vulnerability reports). There are various time-based models, though they also note that these become inappropriate when the total effort spent on detecting vulnerabilities is not linear in time. They also review efforts based on software reliability models using exponential equations. Sadly, Rescorla examined empirical analysis of vulnerabilities of both open and closed source operating systems and found no strong statistical evidence that the “G-O” model (a specific form of this approach) approximated the number of detected vulnerabilities over time.

They also note problems in obvious metrics. In section 4 they note that a single vulnerability that is easy to discover, easy to exploit, and causes severe damage is far worse than ten vulnerabilities that are extremely hard to discover, can only rarely be exploited, and does not cause significant harm. They note various problems with CVSS, and argue that it would be better to separately categorize and measure a few severity classes (such as high, medium, and low).

Thus, they argue that it is “less reasonable to measure the number of intensity of patches, because this provides no information on the number of covered vulnerabilities or on the ages of covered vulnerabilities. [Instead] compute (statistical data on) the reaction time between detection and elimination of a vulnerability, weighted by the level of severity of the vulnerability. It might also seem reasonable to record how many of the detected vulnerabilities are unpatched.”

They propose a “patch index” metric. This is measured at some time tn, producing a metric termed PI(tn), as:

Where i is the index of an event that a vulnerability is announced or patched, ti is the corresponding point in time, pvti is the (possibly severity-weighted) number of detected and patched vulnerabilities in the time window [0;ti], and uvti is the corresponding (possibly severity-weighted) number of unpatched vulnerabilities. By this measure, PI=0 means that “for all announced vulnerabilities, a patch is already provided at the day of the announcement. In contrast, PI=1 would imply that none of the announced vulnerabilities has been patched.” They show graphs of this metric over time to determine trends.

Fundamentally this metric measures response to vulnerability reports, not the number or severity of vulnerabilities. Their paper shows curves over time of both Microsoft Office and OpenOffice that emphasize this. In their study, Microsoft Office had about 7 times more public CVE vulnerability reports than OpenOffice, but the leveled-off patch index is somewhat similar. They note that “probably more vulnerabilities in OpenOffice than in MS Office might have [existed], detected, potentially discussed in forums, and finally removed, before they could become a CVE vulnerability.” On average MS Office had 27% of all announced vulnerabilities unpatched compared to OpenOffice.org’s 18%, while vulnerabilities were patched more rapidly in MS office (median 67.5 days, mean 87 days) than in OpenOffice.org (median 85 days, mean 87.4 days). For more information, see their paper.

Note that [Schryen2011] is also by Guido Schryen.

### Wheeler “Look at the Numbers”

David A. Wheeler (the author) has, for many years, collected quantitative metrics about OSS in the paper “Why Open Source Software / Free Software (OSS/FS, FLOSS, or FOSS)? Look at the Numbers!” [Wheeler2014n]. The referenced papers are also available as a Google spreadsheet. It lists a number of metrics involving security or quality that people have examined.

The security-related measures that might be relevant for our purposes included:

* [Schrye2011] examines data such as the CVEs in the National Vulnerability Database and uses:
  + Mean time between vulnerability disclosures
  + Medians/standard deviations of the severity values of published vulnerabilities as measured by CVSS
  + Percent/number of unpatched vulnerabilities after 4 weeks, and their severity. The author found 17.6% (30.4%) of the published open (closed) source software vulnerabilities (in terms of the median) are still unpatched, though this is likely a significant overstatement since if the authors could not find evidence of patching, the vulnerabilities were counted as unpatched.
* Counts of CVEs or counts of critically-important CVEs (e.g., a meta-analysis by Bugtraq) – especially if compared to software with similar functionality.
* Percentage of CVEs that are critically important (e.g., by Nicholas Petreley’s paper “Security Report: Windows vs Linux”). Note that this primarily measures the effectiveness of countermeasures, but countermeasures matter.
* Vulnerability response time: Average/median time to respond to vulnerability report and produce a fixed version (e.g., Security Portal and Brian Krebs note this).

Reliability-related metrics that might be relevant include:

* Failure rates as measured by fuzz testing (note that there are varying approaches to fuzz testing)
* Reliability or average time up under stress testing.
* Total time off-line over a given year (this is hard to apply to a large set of small projects, however).
* Number or density of defects found by static analysis tools. Density of defects is the number of defects divided by the source lines of code (SLOC). Most tools will miss defects (false negatives), and falsely report issues as defects (false positives), but this is still an indicator of relative quality, since if developers are consistently using dangerous approaches they are more likely to eventually make a security-relevant mistake. If the tool focuses on security vulnerabilities it should be listed in the security-specific metrics. These include reports by Coverity, Reasoning, and others.
* “Maintainability index” – [Samoladas2004], published in the Communications of the ACM, examined almost 6 million lines of code using this metric. This metric was at one point chosen by SEI as the “most suitable tool for measuring the maintainability of systems with high-quality requirements”. Its purpose is to measure the “size” (by various measures) of individual modules, under the theory that if modules are too big it is hard to manage. However, this value is very much a heuristic built on other metrics; it is calculated as 171–5.2ln(avgV) – 0.23avgV(g) – 16.2ln(avgLOC) + 50sin(sqrt(2.4avgPerCM)), where avgV is “average Halstead Volume per module” (a size measure based on the number distinct operators and operands), avgV(g) is average cyclomatic complexity (a measure of structural complexity), avgLOC is average physical lines of code (excluding blank and comment lines), and avgPerCM is percentage of lines of comments with respect to the lines of code. See the paper for more. Note that the maintainability index is also mentioned by [Spinellis2009].
* Harvard Business School’s “Exploring the Structure of Complex Software Designs: An Empirical Study of Open Source and Proprietary Code” by Alan MacCormack, John Rusnak, and Carliss Baldwin (Working Paper Number 05-016) reported using the following metrics:
  + Change cost. This measures the percentage of elements affected, on average, when a change is made to one element in the system. A smaller value is better, since as this value gets larger, it’s becomes increasingly likely that a change made will impact a larger number of other components and have unintended consequences. E.g., a value of 17.35% means that if a given file is changed, on average, 17.35% of other files in the system will be changed, suggesting that they depend (directly or indirectly) on that file.
  + “Coordination cost,” an estimated cost of communicating information between agents developing each cluster. However, this measure is strongly dependent on the size of the system, so this is not useful for comparing projects of different sizes (and thus is not further considered here).

### Presence of security test suite

Heartbleed [Wheeler2014h], Apple’s “goto fail” [Wheeler2014g], and many other vulnerabilities could have been detected ahead-of-time through simple negative testing. That is, identify what should *not* be permitted, and include such tests in the regression test suite. E.G., for every field of a message, include a test with an invalid value (e.g., too high/too low for integers, incorrect length in headers, and so on). Such a test suite should be developed rigorously to cover each field or data type. The presence of such a test suite should increase the confidence that the software resists attack; its absence should raise questions.

### Presence/absence of past security reviews

Security reviews are no guarantee of finding all vulnerabilities. Still, if there is no evidence of a past in-depth security review (or *any* security review), that overall increases the risk of security vulnerabilities. Lack of security testing is also an issue.

It is also important to note the limits of any particular test or review. For example, FIPS 140-2 checks “cryptographic modules” – in particular to ensure that the cryptographic algorithms produce correct outputs given inputs – but it explicitly does *not* perform any tests or examinations of cryptographic protocols such as SSL [Wheeler2014h].

## Specific potentially-useful general metrics

Here are other metrics that are related (e.g., they attempt to predict where defects are especially likely). The theory is that if software has many defects, or other problems, it is more likely to have security problems. These are imperfect predictors, since the only way to know if something is a defect (or a security defect) is to first compare the software to a specification of what it *should* do. However, if some code is unusually complex, it may be more likely to contain a defect. The evidence that security and quality are interrelated is somewhat sparse. However, it is intuitively sensible, and [Woody2014] provides some evidence for it.

This is an absolutely huge field, and not as related to our problem, so this subsection in particular is an even smaller part of the field.

### Software quality metric measurement tools

Many tools, both OSS and proprietary, can measure various static attributes that may suggest something is more or less likely to contain defects. There are many metrics (such cyclomatic complexity as described below), and many ways to combine various lower-level metrics into higher-level metrics that might have some meaning.

Examples of proprietary software include CAST Software’s (<http://www.castsoftware.com/>), Semantic Design’s, and McCabe IQ (<http://www.mccabe.com/iq_developers.htm>). OSS tools include CCCC and Eclipse Metrics plugin.

One challenge is differing results from different tools. Lincke et al’s “Comparing Software Metrics Tools” <http://arisa.se/files/LLL-08.pdf> compares a few tools, and shows that different tools often report different values for the “same” measurement. This might seem odd since some of these metrics have definitions published in academic journals, but in practice many of these definitions have ambiguities that result in different values.

### Compiler warning flags and style checkers

An alternative approach is to use compiler flags and style checkers to maximally complain about potential issues. Both clang and gcc support many warning flags, for example. These can, again, be divided by KSLOC to give density figures.

### Senior Defect Prediction measures: McCabe (Cyclomatic complexity) and Halstead

The McCabe (cyclomatric complexity) and Halstead measures were defined in the 1970s for predicting defects in functions/methods (at the time called “modules”) through static analysis of code. They are widely used as static measures, in part because they are well-known.

McCabe argued that code with complicated pathways are more error-prone. His metrics focus on this. Especially known is Cyclomatic complexity, which measures the number of “linearly independent paths”; many consider a number more than 10 concerning. These *do* measure, in a sense, how much effort is needed to do full coverage in branch testing [McCabe1976].

Halstead argued that code that is hard to read is more likely to be fault prone. He estimated reading complexity by counting concepts, such as the number of unique operators [Halstead1977].

Fenton and Pfleeger have noted a number of problems with static measures, since they are clearly not complete measures [Fenton1997]. Of course, a measure does not need to be complete, just useful.

A summary of early predictive metrics is at: <https://code.google.com/p/promisedata/wiki/ShortTutorialOnDefectPrediction>. A short discussion of McCabe metrics is at <https://code.google.com/p/promisedata/wiki/McCabe> and a few notes on Halstead are at <https://code.google.com/p/promisedata/wiki/Halstead>.

### Test coverage

There are various ways to measure the quality of the tests (the “test coverage”) of a regression test suite, and many tools (including gcov/gcc) that can measure them. Especially common measures are statement coverage (the percent of statements executed by a test suite) and branch coverage (the percent of branches, both true and false, executed by a test suite).

### Source lines of code (SLOC)

Many organizations measure lines of source code (SLOC). By itself SLOC says nothing about security, but SLOC is highly correlated to development effort, and it is also correlated to any review effort (modulo other factors such as complexity). One complication is that SLOC can be measured different ways: Text lines, physical SLOC (which skip blank and comment lines), and logical sloc (which measure logical statements).

David A. Wheeler’s “sloccount” can automatically measure physical SLOC for many languages. One list of tools is available at <http://www.locmetrics.com/alternatives.html>.

### Linke survey

Lincke et al’s “Comparing Software Metrics Tools” <http://arisa.se/files/LLL-08.pdf> looked at many different tools for static metrics of (Java) source code that focused on classes. They identified 17 object-oriented metrics which (i) they could rather securely assign to the same concept, (ii) are known and defined in literature, and (iii) work on class level. They selected 9 such metrics which most software metrics tools they identified would report:

1. CBO (Coupling Between Object classes) is the number of classes to which a class is coupled.
2. DIT (Depth of Inheritance Tree) is the maximum inheritance path from the class to the root class [5].
3. LCOM-CK (Lack of Cohesion of Methods) (as originally proposed by Chidamber & Kemerer) describes the lack of cohesion among the methods of a class.
4. LCOM-HS (Lack of Cohesion of Methods) (as proposed by Henderson-Sellers) describes the lack of cohesion among the methods of a class.
5. LOC (Lines Of Code) counts the lines of code of a class.
6. NOC (Number Of Children) is the number of immediate subclasses subordinated to a class in the class hierarchy.
7. NOM (Number Of Methods) is the methods in a class.
8. RFC (Response For a Class) is the set of methods that can potentially be executed in response to a message received by an object of the class.
9. WMC (Weighted Methods per Class) (using Cyclomatic Complexity as method weight) is the sum of weights for the methods of a class.

Of course, a metric might be widely reported because it is easy to measure, not because it is useful. Still, widely-measured metrics might be useful.

### Estimating Commit Sizes Efficiently (Hoffmann and Riehle)

A common variable for measuring work contributed is the “commit size,” i.e., the number of lines added, removed, and changes. However, post-facto this can only be estimated; typically the only information that is available (especially in metrics repositories) is the number of lines added and removed for each change, and obvious ways (such as adding them up) lead to errors.

Philipp Hofmann and Dirk Riehle devised an improved method to estimate commit size based on the data actually available (the number of added and removed lines) using a linear regression model:

function real diff\_size(int a, int r)

if (0.01269 × a + 0.01540 × r > 2.9965)

return 0.9497 × a + 0.9744 × r – 2.9965

else

return 0.9370 × a + 0.9590 × r

end

end

More information is available in [Hofman2009].

### Gao: Choosing software metrics

“Choosing software metrics for defect prediction: an investigation” [Gao2011] examined how to identify a relatively small set of metrics that nevertheless supported defect prediction (separating moduels into “fault-prone” and “not-fault-prone”) [Gao2011]. “The results demonstrate that while some feature ranking techniques performed similarly, the automatic hybrid search algorithm performed the best among the feature subset selection methods. Moreover, performances of the defect prediction models either improved or remained unchanged when over 85% of the software metrics were eliminated.”

The most frequently selected attributes were: “number of distinct include files (FILINCUQ), number of different designers making changes (UNQDES), deployment percentage of the module (USAGE), base 2 logarithm of the number of independent paths (LGPATH), total span of branches of conditional arcs (CNDSPNSM), number of problems fixed that were found by designers in the prior release (DESFIX), and number of problems fixed that were found by customers in the prior release (CUSTFIX).”

### Menzies: Assessing Predictors of software defects

“Assessing Predictors of Software Defects” by Tim Menzies et al of 2004 (<http://menzies.us/pdf/04psm.pdf>) found that “When learning defect detectors from static code measures, NaiveBayes learners are better than entrophy-based decision-tree learners. Also, accuracy is not a useful way to assess those detectors. Further, those learners need no more than 200-300 examples to learn adequate detectors, especially when the data has been heavily stratified…”. This doesn’t seem as directly relevant for our purposes.

### Wang – How many software metrics for defect prediction

“How Many Software Metrics Should be Selected for Defect Prediction?” by Wang et al examine various learning algorithms over various metrics [Wang 2011].

Overall, they found that the best classification model for their dataset (using Eclipse data) was built with only three features selected by the AUC ranker using LR learner. Unfortunately, they do not appear to reveal *which* features, nor have I found enough data to allow simple reuse of the generated model, which makes this work tantalizing but difficult to use or repeat.

### Punitha – Software defect prediction

Punitha and Chitra’s “Software defect prediction using software metrics - A survey” surveyed materials to “help developers identify defects based on existing software metrics using data mining techniques and thereby improve software quality which ultimately leads to reducing the software development cost in the development and maintenance phase. This research focuses in identifying defective modules and hence the scope of software that needs to be examined for defects can be prioritized.” [Punitha2013]

In their approach, they built an inference system to predict modules most likely to be defective using supervised learning (inferring a function from labeled training data). Specifically they applied “SVM, a supervised training algorithm for classification of data into two sets, buggy and non-buggy. Then various rules are generated inferred from the support vectors. The final set of the rules is chosen from the given set of rules using genetic algorithm optimization. The experiments were performed on Eclipse bug data…”

Oddly, the paper describes figures of merit to show their effectiveness, and seems to suggest that the effectiveness is measured in the paper, but the actual measurement values for software defect prediction do not seem to be in the paper.

In any case, this paper notes the potential use of supervised learning to help determine vulnerable software or modules. Focusing modules may have real advantages; there is more data to work with, and focusing on problematic portions of a program (instead of the entire program) may increase the likelihood of real improvements. However, these are not focused on measuring security, and there is always risk that a module that doesn’t *seem* like a vulnerable or defective module is implementing a severe vulnerability in a straightforward way.

### COQUALMO

COQUALMO (COnstructive QUALity Model) is “an estimation model that can be used for predicting number of residual defects/KSLOC (Thousands of Source Lines of Code) or defects/FP (Function Point) in a software product.” Information, and a spreadsheet that implements the model, is available at <http://csse.usc.edu/csse/research/COQUALMO/>.

It was developed in part by Barry Boehm and is similar to the COCOMO effort estimation model. In particular, it requires information on the “defect removal” processes. It is not clear that this model would be particularly helpful for our situation, where we must examine a large number of different projects (instead of a single project).

### DoD/Industry 1994 Survey

David A. Wheeler co-authored a 1994 survey of software metrics in the Department of Defense (DoD) and industry [Springsteen1994]. This is ancient history, but it clearly shows that interest in software metrics has been around a long time. This survey was extremely broad, covering metrics for a variety of purposes not specifically relevant in our case. It made various points, e.g., that “collecting data is different [in general] and it is important to have a simple, goal-directed metrics program” [Springsteen1994, 2.6.1].

Common metrics used in the DoD that are relevant here are:

* Source lines of code (SLOC) for measuring size (with some variation on definition)
* Defect status metrics, e.g., the number and age of unresolved issues.

In industry the most common calculated metric in use was error density. Other metrics relevant to our purposes included:

1. customer severity days (severity of customer problem multiplied by days open, summed by severity level)
2. problems per user-month
3. mean time to defect after release
4. defect containment effectiveness (number of defects removed after internal review but before release, divided by the (number of defects removed after internal review but before release + number of defects remaining in release))

## Attack surface measurement

A different approach is to try to measure how easy it is to attack a program or system. If software has code that looks like a vulnerability, but it cannot be exploited, it doesn’t really matter.

“Measuring Relative Attack Surfaces” by Michael Howard, Jon Pincus, and Jeannette M. Wing introduced a metric “for determining whether one version of a system is more secure than another with respect to a fixed set of dimensions. Rather than count bugs at the code level or count vulnerability reports at the system level, we count a system’s attack opportunities. We use this count as an indication of the system’s “attackability,” likelihood that it will be successfully attacked.” [Howard2003]

“Measuring a System’s Attack Surface” by Pratyusa Manadhata and Jeannette M. Wing [Manadhata2004] develops a process for measuring and comparing *systems* (not individual programs)’s attack surface. They note that “Today we commonly use two measurements to determine the security of a system: at the code level, we count the number of bugs found (or fixed from one version to the next), and at the system level, we count the number of times a system version is mentioned in… Common Vulnerabilities and Exposures (CVEs) [31] etc. We argue [that] both measurements, while useful, are less than satisfactory… The system actions externally visible to the system’s users together with the system resources accessed or modified by each action constitute the system’s attack surface. Intuitively, the more actions available to a user or the more resources accessible through these actions, the more exposed the attack surface. The more exposed the attack surface, the more likely the system could be successfully attacked, and hence the more insecure it is. We can reduce the attack surface to decrease the likelihood of attack and make a system more secure… certain system resources are more likely to be opportunities, i.e., targets or enablers, of attack than others… We identify the system resources that are opportunities of attack by a given set of properties associated with the resources, and categorize them into attack classes. These properties reflect the attackability of a type of resource, i.e., some types of resources are more likely to be attacked than other types.” They then provide a specific description of this approach that counts the number of open TCP/UDP sockets, open RPC endpoints, services running as root, etc. As written this is intended for evaluating entire Linux distributions (or similar), not for evaluating individual software packages. It might be possible to identify the packages that directly *cause* these increases in the attack surface, to help identify specific packages involved in the attack surface. This would not identify indirect attacks; dependency information might help, but might also pick up too many packages unlikely to be a vulnerability source.

The later [Manadhata2007] expands on this approach and focuses more directly on measuring specific applications (e.g., IMAP servers and FTP daemons). This paper works to formalize the notion of a system’s attack surface and proposes “a method to measure a system’s attack surface systematically.” However, it still requires domain knowledge and execution of the program; it’s not clear this would scale well to the large number of programs we are considering in our case.

OWASP has some information on attack surface at:

<https://www.owasp.org/index.php/Attack_Surface_Analysis_Cheat_Sheet>.

## Kenn White metrics set

Kenn White had earlier collected data about projects for the Linux Foundation, specifically for this work. One set collected data about projects with these columns:

* Project (name)
* Source (URL)
* Maintainers
* Commits
* Notes

In a separate list he identified the following metrics: Project (name) , Source Repo (URL), CoreDependents, Maintainers, PrimaryMaintainer, Affiliations, CommitsLast30, StatsInfo (where to get statistics), SlocApprox, IssuesOlder1YR (a count of the number of issues open for more than 1 year), OpenIssues (count of open issues), IssuesInfo (URL for information on issues).

## IDA ideas

### Exposure to attack

Clearly we want to prefer packages where vulnerabilities make a difference. An externally-accessible vulnerability is especially bad, though an internal one (particularly one that allows privilege escalation) is bad as well. Packages directly exposed to attack are obvious, but indirect attacks are also dangerous (e.g., a vulnerability in an image processing or decompression program might be remotely exploitable).

One remarkably simple approach is to simply note if a package has at least one CVE. A program that has at least one CVE assigned to it has, by definition, had at least some way to exploit it. Therefore, any program with at least one CVE should be considered as exposed to at least some level.

A simple measure of “external exposure” might be helpful. After all, a program or library that is externally-exposed is probably more important to evaluate than one that is not. It is harder to counter attackers who are authorized to run software on a system, so we propose starting by focusing on countering attack via external networks. Here is a simple proposed qualitative measure (as an example):

* 3: direct network exposure to attacker. Programs that connect to a network port or directly manage protocols, e.g., OpenSSH.
* 2: known exposure to external network attacker data because they are often involved in processing of it. This would include decompression algorithms and image processing algorithms. It would also include anything that took user data and transformed it (e.g., turning it into a query) before sending it to a database.
* 1: Known to pass through attacker data from external networks. This is software where attacker data may pass in and out, but little significant processing is supposed to be done with it. This would include database systems (e.g., MariaDB and the dbm implementations). This would also include any shell used as /bin/sh; POSIX and many programming languages include built-in calls that go through the shell. This was an issue exposed by Shellshock.
* 0: Little external exposure. Attacks might still subvert vulnerabilities in these programs, and cause great damage, but they are deemed less likely.

There’s also the risk that something could be exploited to cause privilege escalation. This especially includes any package suid/setgid program (especially for a privileged user like root), as well as privileged processes that users can communicate with (e.g., via local sockets or Dbus). There’s a risk that the programs these depend on could be used; perhaps package dependency information could be used to estimate this.

There is some literature on measuring attack surface. However, it’s not clear those would be easy to measure across a large number of different programs in different languages, so we have not pursued them.

### Other work

IDA has previously done in-house work measuring hit density, where hits are reports from flawfinder or another lexical tool, and the density is found by dividing by physical source lines of code. These tools simply report riskier constructs, not really vulnerabilities, but the theory is that if developers often use riskier constructs, they are more likely to produce insecure results. A comparison of sendmail and postfix of years ago suggests this might be a useful measure.

Programming language can also be a potential indicator. A vulnerability can be created in any language. That said, it is especially easy to write vulnerable code in C and C++ because some of their fundamental operations (e.g., array and pointer access) provide no protections (e.g., against buffer overwrites or overreads) – a problem exposed by Heartbleed. C and C++ do not provide automatic memory management; there are advantages to this, but it can increase code complexity and provides additional methods of attack (e.g., through double-frees). PHP is another potentially-concerning language – in part because some of its operators have surprising properties (e.g., due to surprising type conversions), but also because many developers with limited skills develop using PHP. Thus, while there are many good developers who use PHP, there are many developers who do not know how to develop secure software and yet use PHP, so much so that some people will specially examine programs written in PHP.

A large percentage of software developed many years ago (say, more than 10 years ago) can be a potential sign of trouble. Many years ago fewer developers knew how to develop secure software, and a project with that much unchanged may suggest that it is fairly inactive. Of course, software can be stable because it is well-written and its requirements have not changed. This kind of information can be retrieved from version control systems, but in some cases the data is not easily available. A plausible proxy could be the project start date; a project that started long ago, even if active, might harbor many vulnerabilities due to old bad practices. (For example, many vulnerabilities in X-Windows were found in 2014, in part due to such issues.)

Complexity density (cyclomatic complexity divided by KSLOC) is potentially promising.

## London January 2015 Meeting

In January 2015 there was a meeting in London to examine the metrics that would suggest project that needed investment. Many people contributed their ideas of what metrics might be appropriate. Here is a summary created by Wheeler (the official summary is to be released at the time of this writing).

First, metrics that might assess project health:

1. Easy:
   1. History of many vulnerabilities (e.g., as counted by CVEs)
   2. Few/1/no developers
   3. Poor bug response (long delay on average, large % ignored) – for now, merge bugs and feature requests when they cannot be differentiated.
   4. Number of commits over time
   5. Exploitability (e.g., direct or indirect exposure to remote network, potential for exploit use as local privilege gain)
   6. Language in use (e.g., C or C++)
2. Hard but could be easy:
   1. Dependency information – this can help indicate importance
   2. Differentiate between bugs and feature requests – create a standard way to see the difference across all projects.
   3. Has it been fuzz tested? Audited?
3. Hard:
   1. We don’t know how to measure security
   2. What can projects do to make it easier to capture metrics? (e.g., tweaks in GitHub, SourceForge, etc.)
   3. Huge impact if broken. This is like importance/widespreadness. E.G., “What % of the Internet breaks if this breaks?”
   4. Cross-project algorithmic similarity. E.g., if code is copied/pasted, how can we tell that there’s a common problem? “SourceDNA”.
   5. Scoring algorithms: How do you combine metrics especially well?
   6. It’d be possible to use learning algorithms & analyzing data to see what is effective, but that requires that we know the “truth values” (which we typically do not know)
   7. Stakeholder’s priorities
   8. Exposure (I think this is related to exploitability)

There was a perceived need to make this data available to the public. There was also concern by some on how journalists could distort these reports (e.g., it might be mischaracterized as “all OSS is insecure”).

There would be many potential responses, including modifying it, funding developers & auditors, refactoring it, and rewriting it (from scratch).

## Additional areas (“TODO”)

Time limited what we could look for. Some additional areas to consider are:

* Examining more sources on “how to release” OSS, i.e, best practices. Failure to apply best practices may indicate problems. The “FAIL” measure is an example of this approach.
* Looking at more search results, e.g, for “Best software defect prediction metrics” and “OSS security evaluation”
* Reviewing more OSS evaluation methods; Wikipedia has a long list at <http://en.wikipedia.org/wiki/Open-source_software_assessment_methodologies>

Consider Ross Anderson’s paper “Security in Open versus Closed Systems:

The Dance of Boltzmann, Coase and Moore” <http://www.cl.cam.ac.uk/~rja14/Papers/toulouse.pdf>

## Comments on metrics

Some metrics that are especially obvious to collect at first are:

* Project name. This is more complex than you might think. The same name may be used by multiple projects, and different projects can share a name. Project forks complicate this further. In many cases the project URL can be used to determine if it is the same or different program, as Open Hub and typical Linux packages include this information.
* Source lines of code (SLOC). Larger programs will take longer to audit, so this is important to help understand scale.

Measurements of the project seem very promising, e.g.:

* Size of development team (committers), e.g., the number of people who committed over the last 12-month period. 0 is especially bad, but 1 is also bad (this is vulnerable to the “hit by the bus” problem). In some cases version control systems don’t report the real facts (if all changes are allocated to the final committer, not to the submitter).
* Activity metrics. Number of commits or change counts, especially over the last 12 months. Ideally this is stable or increasing over the previous 12 months.
* Bug report responsiveness. E.G., average response time for a bug report, the number or percentage of bug reports that have never been resolved after 30 days, the percentage or number of bug reports resolved but only after 30 days. These should be for true bugs, not including wish lists/feature requests, though that may not always be possible. There may need to be a separate statistic for % of bugs closed as “WONTFIX”; it is easy to close all bugs quickly if you won’t fix any of them. We should probably only care about the statistic for the last 12 months (e.g., of all bugs opened starting this year). Wheeler expects that most bug tracking systems are covered by GitHub (label “bug”), SourceForge, savannah, or Bugzilla. If the system doesn’t separate features vs. bugs, we can only report on the combination. On some systems (e.g., GitHub) there’s a way to identify bugs, but it’s not always used; in that case, report what you can. It would be great if, in the long term, organizations made it easier to separate bugs from non-bugs.

Measurements suggesting riskiness (likelihood of security problems) may include:

* Dominant programming language. C is considered especially dangerous, with C++ also dangerous.
* Year of project start (or at least, if it is more than 10 years old). If it started long ago, may have lots of bad practices. Version control databases may not record the entire history of a project, but they can typically report if it’s more than 10 years old. Open Hub does capture “maturity” which hints at this.

CVE counts are related to security, but their relationship is complicated. After all, if a program undergoes intense audits and has fixed them, it may have many more CVE reports than a critical program that has been ignored. We intend to capture CVE counts, but primarily as a way to determine that a program is especially relevant to security; if it has at least two CVEs, it clearly is relevant to security. (One could be a fluke.) See the discussion earlier about CVEs.

There are also some “project smells” (some inspired by the FAIL metric) that suggest that there may be problems:

* Failure to have a (working) website). (This is “communication” in the “doomed to FAIL” index).
* Failure to have a mailing list or IRC mechanism. (This is also “communication” in the “doomed to FAIL” index).
* Failure to have a public source version control repository. If users can only download the final code, there is no opportunity to review the code *before* users are supposed to use it, and there is little opportunity to collaborate. (This is “source control” in the “doomed to FAIL” index) Git is common, but it could use subversion, CVS (even though that is old), mercurial, etc.
* Failure to have a public bug report tracking system. Without this, it’s more difficult to determine if the developers are being responsive to reports, and it is harder for users to determine if the problem has occurred. It is possible to use mailing lists and IRC for bug reporting, but if that is the *only* mechanism there is a risk that problems (including vulnerabilities) will be forgotten and slip through the cracks. Some active projects don’t have a public bug report system, of course, but it is still a potential concern. This could be using Bugzilla, GitHub’s issue tracker, SourceForge’s tracker, etc.

So we should add these (up this point) – Open Hub provides many, and we are adding CVE counts, Bug tracker responsiveness, and project smells. At that point, we should have enough to do examination and reporting.

Finding code repositories is an interesting problem. We can use Open Hub to find human-visible project info (e.g., <https://www.openhub.net/p/openssl>). In some cases it points to “Issue tracker” and “Code Locations”. If not, we can look at “browse code” to see if it refers to a publicly-accessible repository (and not just a copy of a final downloadable tarball). E.G., if it ends in “trunk/” it is almost certainly a subversion repository; if it starts with “git:” or ends with “.git” it is a git repo.

It would be possible to examine projects for more complex measurements, especially of the code itself. However, these may take more time to collect, so it might be appropriate to only get these measures once a subset of projects has been identified, if we want to get them at all:

* Percentage of comments (compared to norm for that language). Programs with few comments may be harder to review, and thus vulnerabilities may be easier to miss. This metric is not especially informative, however; a program with many comments may be strewn with vulnerabilities, and comments can often mislead. This data is automatically provided by Open Hub, so we may as well use it.
* Potential security defect density. Run Coverity scan, flawfinder, RATS, and/or some other tool and divide these by the KSLOC. Higher numbers suggest that the program often uses risky constructs, and thus is at higher risk for vulnerabilities.
* Warning density. Run warning flag/quality tools (e.g., clang warning flags) and divide by SLOC, again, to see if the program often uses risky constructs.
* Regression test coverage. E.G., statement coverage, or even better branch coverage. A program that has poor test coverage has many untested areas. This requires running the regression test suite; that is not an easy thing to do for a large set of programs.
* Complexity (McCabe). Complexity measures for functions/methods can be rolled up in various ways, e.g., the percentage of functions or methods with a complexity more than 10, or the average complexity density (for each function, divide by lines of code, report the average). Highly complex code is more likely to have vulnerabilities. There are some OSS tools that can collect this data for some languages.
* Vulnerability repair speed aka responsiveness (how quickly they respond to a vulnerability report, especially a public one). This may be determinable from the bug database, but this is often harder because bug databases don’t always link to vulnerability reports. The CVE database may help.

# Important OSS Projects

Another challenge is to identify OSS projects that might need investment. These potential projects need to have metrics collected about them, and then compared to determine which ones most need evaluation.

There are many challenges to identifying important OSS projects. Many OSS projects themselves depend on other projects, which may be copied into them; this means that some lower-level libraries are not necessarily obvious as being widely-used.

Some OSS projects are rarely used, while others are very widely used. In general, we want to prefer ones that are widely used in various configurations. E.G., OSS that are widely used in common Linux distributions, especially if they are in the minimal install or in common configurations (e.g., server installs), are of special interest.

Not all OSS projects are equally exposed to attack; some software, such as network programs and decompression libraries are obviously directly exposed to data from attackers, while others are less exposed (and thus it is harder to determine their importance for investment). Some lists (e.g., the Linux Foundation original list) take this into account; in addition, we have contemplated using a rough ‘exposure’ metric to capture this issue.

Here are some sources that can be used as a starting point for identifying these projects.

## Common in distributions

One approach would be to take a common Linux distribution (e.g., Debian) and find:

1. Packages always installed in the minimal (“Base”) install
2. Packages installed by a predefined group, e.g., “Web Server”
3. Packages commonly installed in common cases as a server OS
4. Packages commonly installed by a developer. E.G., GnuPG (used for signing by git and many other tools) is an infrastructure component widely used to secure development.

We plan to start with the first two with Debian (“Base” and “Web Server”), augmented with installation of ssh, as these are extremely common programs. Debian is a useful place to start, since it has a large number of packages that represent OSS widely.

It would be possible to cross-reference this among Red Hat-derived distributions. They sometimes choose different package names, but in those cases, the project URLs could be used to match the actual programs.

## Linux Foundation original list

The Linux Foundation CII FAQ listed the following as “critical open source software project” after an initial review for prioritized funding, according to:

<http://www.linuxfoundation.org/programs/core-infrastructure-initiative/faq#faq11>:

* Network Time Protocol (NTP). Note: There are several implementations of NTP.
* OpenSSH
* OpenSSL

David A. Wheeler privately asked Greg KH what list the Linux Foundation was thinking about pursuing. He said, “I don't think the list is written down like ‘worth doing’ it was just a ‘dump of what projects should be researched to see if they are worth funding and if they even make sense.’ Basically everything in the ‘lower’ layers of a Linux system. Here's a list that I found that we had come up with, no judgment call from me if they are even at all relevant:”

1. Compression libraries, including LZ0, LZ4, libgz. David A. Wheeler would add implementations of unzip; many formats are basically extensions of the zip format, so there are many routines that depend on them.
2. Pluggable Authentication Modules (pam). Everyone uses it, and it is critically important.
3. Web services libraries: libcurl, libxml, json-c, libpng
4. Voice: libzrtp
5. Apache-related: mod\_ssl, mod\_tls, mod\_auth\_\*, mod\_compress
6. Encryption libraries: LibreSSL, modssl, modtls, BouncyCastle, gpg, otr, axolotl
7. Static analyzers: Clang, Frama-C
8. Nginx
9. OpenVPN. It was noted that funding model may be similar to what was tried with OpenSSL, where consulting funds the company. It was also noted that openVPN needs to correctly use OpenSSL in order to be secure, and this may be more worthwhile.
10. OpenWRT. Imagination is working on OpenWRT, but may not be focused on security. It was noted this is a big effort.
11. BIOS projects: coreboot and TianoCore
12. RDMA - kernel drivers that accesses memory directly (infiniband)

## Kenn White list

Kenn White had developed a list of potential projects, including:

* libBFD (binutils; gdb)
* libcURL
* libIDN
* libXML
* libLZMA

Example dependents for Web Servers/Core WS include:

* Apache (2.x): mod\_php, mod\_python, mod\_ssl
* BusyBox
* Django (PycURL)
* FastCGI/lighttpd
* Grails
* Java (JSP, J2EE, XML services)
* Nginx
* PHP (5.x)
* Ruby Gems
* Ruby on Rails (Passenger)
* Tomcat (6/7.x)

Example dependents for Core OS Services are:

* b43 (wireless networking)
* cryptsetup-­‐luks (volume/disk encryption)
* device-­‐mapper
* dhclient (DHCP)
* dracut (bootup, kernel bootstrap)
* Internationalized characters/Unicode/Puny/DNS
* iproute
* iptables
* gnupg2
* lvm2
* openssh (server and clients)
* SE Linux (policycoreutils)
* Mail (postfix)
* RPM and Yum
* RSyslog

## Google Application Security Patch Reward Program

On October 9, 2013, Google announced a program to rewards proactive security improvements to select open-source projects. Details are available at: <https://www.google.com/about/appsecurity/patch-rewards/> As of 2014-12-08 this program limited potential awards to the following OSS projects, which Google appears to consider as important:

1. Open-source foundations of Chrome and Android: Chromium, Blink, Omaha, AOSP
2. Security-critical, commonly used components of the Linux kernel (including KVM)
3. High-profile web and mail servers: Apache httpd, lighttpd, nginx, Sendmail, Postfix, Exim, Dovecot
4. Other high-impact network services: OpenSSH, OpenVPN, BIND, ISC DHCP, University of Delaware NTPD
5. Core infrastructure data parsers: libjpeg, libjpeg-turbo, libpng, giflib, zlib, libxml2
6. Other essential libraries: OpenSSL, Mozilla NSS
7. The reference implementation of Certificate Transparency and its open-source dependencies
8. Toolchain security improvements for GCC, binutils, and llvm
9. Security-relevant bits of common package managers: yum, apt, pip, npm
10. Popular web frameworks: Angular, Closure, Dart, Django, Dojo Foundation, Ember, GWT, Go, Jinja (Werkzeug, Flask), jQuery, Knockout, Struts, Web2py, Wicket

## Recent problem reports

Some software with known recent problems (vulnerabilities or known lack of support) include:

* Network Time Protocol (NTP) services, since these are directly exposed to external exploitation <https://ics-cert.us-cert.gov/advisories/ICSA-14-353-01>.
* GNU Privacy Guard (GPG) has only 1 developer, yet is widely used to secure software (including signing of commits). <https://gnupg.org/blog/20141214-gnupg-and-g10.html> There are also known problems with GPG: They have dropped the ability to read old formats (requiring people to keep old versions with known problems) and don’t’ support smart cards <https://lwn.net/Articles/626660/>

## Augmented list of programs

As noted earlier, we started with the first two with Debian (“Base” and “Web Server”), augmented with installation of ssh, as these are extremely common programs.

We then looked for programs/projects that should be added. We did not want to include simple bindings for different programming languages; there are many bindings for a given library, and since the functionality is primarily in the base, we wanted to start by focusing on the base library. For our purposes, we use Debian package names as an index key. The categories (based on the above) we added are:

1. Encryption libraries/tools: These were found using “grep -E '([Ee]ncrypt|[Dd]ecrypt|cryptographic|\<TLS\>|\<SSL\>)' …” on the full Debian package list and then manually reviewing the results. Note that in addition LibreSSL would qualify. This identified 214 packages, many of which are rarely used or are implementations for specific languages. That makes it harder to determine what to include. Manual review of this list identified the following as potentially especially important:  
     
   coolkey, gnutls-bin (and other gnutls), libopencryptoki0, libpolarssl-runtime. Note that libssl1.0.0, openssl, and libgnutls26 were already on the list.  
     
   These others may also be important: libace-inet-ssl-6.0.3, aespipe, aolserver4-nsopenssl, libbeecrypt7, ccrypt, claws-mail-smime-plugin, courier-imap-ssl, courier-ssl
2. Compression libraries. The Debian packages in the base that have “compress” in their description are: bzip2, gzip, libbz2-1.0, liblzma5, xz-utils, zlib1g. (These are thus already covered in our starting set.) At least adding “zip” and “unzip” (from info-zip) seems wise.
3. Image processing libraries.
4. C/C++-based XML processing libraries.
5. Key network protocols: ntp (Network Time Protocol), bootp

We then added others as they were identified as important, so they could be analyzed. Our time was limited; adding more would be useful.

The story of PAM is complex. OpenPAM is used by many \*BSDs and Apple MacOS X; see <http://www.openpam.org/wiki/History> for details. This is not the same as Linux-PAM.

The library for BFD files is part of binutils; binutils itself has significant maintenance, but the BFD portion less so. This is difficult to tease apart because of the way it is packaged.

Kenn mentioned “otr”; based on discussions with Dan Kohn, we believe this the “off-the-record” (otr) message protocol, and probably more specifically the portable sample library and toolkit implementation available via the home page <https://otr.cypherpunks.ca/>.

# Selected approach and early results

## Overall approach

There was little time, and the goal was to quickly identify plausible candidates that will then be reviewed by humans. Thus, we focused on metric values that can be easily and quickly acquired. In particular, we used data from Black Duck Open Hub (formerly Ohloh) where possible, as well as from the Debian package repository, as these are easy to get quickly.

We then plan to extend this start with other information that can help us quickly filter out likely candidates. We are gathering this data via programs, so that the results can be re-run later as more data is desired or updated information becomes available.

We have been asked by the Linux Foundation to focus on projects that have relatively little activity or current development effort. For example, the content management systems (CMSs) WordPress, Joomla!, and Drupal are widely-used, and all have had many vulnerabilities identified (especially once all their plug-ins are included).[[1]](#footnote-1) However, these CMSs have a number of developers behind them, who already look for and attempt to counter vulnerabilities. They could do better, but at least there is typically progress in those areas; our concern for now is those projects which are relatively inactive and thus are unlikely to improve over time. Some CMS plug-ins are widely used yet are inadequately audited; we are not looking such plug-ins at this time, but they would be good candidates for the future.

## Caveats

We have developed software to automatically gather data, attempt to “clean it up” (e.g., to match names), and report it. However, this is subject to the messiness of real-world data.

Names in particular are difficult. The same name may be used by unrelated projects, forks of projects may have different names, and different sources may use different names for the same project. We have tried to correlate the data, but this is certain to be imperfect.

Open Hub is an important data source for our process, but in some cases it only provides summary data in its API (e.g., it reports number of commits are going up or down, but not the exact number). In many cases we can get this information from other sources, but we have not bothered to do so unless it seemed important.

The first CVE reports used keyword matching. This means, for example, that if a CVE report mentions product X, then the CVE will show up even if the CVE does not actually apply to product X. CVEs are often specific to environments. This was later changed to use the Debian project’s special mapping database, which greatly improved the results.

Since it is difficult to directly estimate the vulnerability of software, we primarily emphasized process measures that suggest inadequate maintenance, as described above.

Our primary purpose is to simply identify likely candidates. It is acceptable to identify a few candidates that turn out to not need significant effort, since the candidates will be reviewed by humans anyway. Thus, we tried to err on the side of identifying something if we were not sure. However, we could not take this to extremes; simply identifying all OSS projects would not be useful.

## First stage

On 2014-12-16 we submitted an early mock-up of OSS projects and data about them. This was partly created automatically, and partially by hand, to see if our overall approach would work. We determined that while there were difficulties with data “messiness,” the overall approach did work.

We first took the list of Debian packages in Base + Web-server + ssh. Specifically we used Debian 7.7.0 (Wheezy), first released 2014-10-18, stable amd64 (x86\_64). That has 368 binary packages (not including 3 “tasks” which aren’t really packages, but it \*does\* include 4 packages installed when running on virtualbox). Our theory is that if it’s in Debian base, it applies to most Linux distros (it certainly applies to Ubuntu, and Red Hat based systems also have many of the same software packages). They probably apply to many non-Linux situations as well. In some cases one source package generates multiple binary packages. We can expand this, of course, but I think it’s a plausible start.

We then cross-referenced that list to Open Hub metrics. There’s about a 70% match (about 70% of the time, we found that Open Hub has metrics data), which saves a lot of time. We had to hand-determine the mapping between Debian and OpenHub. In the longer term, matching on home page URLs might help.

OpenHub doesn’t provide all the data we’d \*like\* to have, e.g., the exposure to vulnerabilities or how well they process bug reports, but it should still help us identify projects that are probably not well-maintained.

We then tried out to see if we could use just those metrics to help find concerning projects. In this case, we filtered on projects that Open Hub reported were Small/Single Developer and Small/Stable Activity. Note that this automatically does NOT include the 30% of projects where we have no data, but we were basically looking to see if the process made some sense. We then manually looked through the list. Just doing that suggests some plausible candidates, such as (giving the Debian package names): PAM-related packages (libpam-modules, libpam-modules-bin, libpam-runtime, libpam0g), libexpat1, procmail, zlib1g, libsasl2-2, libsasl2-modules, gzip, libfuse2, libgpgme11, libkeyutils1, libpng12-0. Ones that met the criteria, and perhaps might justify more investigation, include: netcat-traditional, dmidecode, libgdbm3, libnfnetlink0, libsemanage1, libsemanage-common, locales, libpopt0.

As noted there are projects we don’t have any data on yet, and there are other important metrics that OpenHub doesn’t provide (bug tracker or repository URL, CVE data, bug processing data, whether or not there are setuid or accessible privileged programs, etc.). In addition, there are almost certainly projects that aren’t in that set of Debian package yet should be considered. This was all a quick rough-up, anyway; I’d like to make this relatively automatic. We’re looking at how to improve on all those points. But I think it all shows that we’re making progress & that this is a reasonable direction.

The heading names began with “Debian\_” if the data source was Debian, and “Openhub” if the data source was Open Hub (Ohloh).

The headings in this version were:

1. Debian\_Package: Debian name of binary package
2. Debian\_Source: Debian name of source package; one source package may generate multiple binary packages.
3. Debian\_Version
4. Debian\_Description: Debian’s one-line description.
5. Debian\_Homepage: Homepage for the entire project.
6. Debian\_Install: Debian installation package, e.g., Standard System Utilities.
7. OpenHub Query Name: This mapped the Debian package name to the name useful for querying. In some cases this is odd, e.g., Debian’s “apt-listchanges” maps to “8066”.
8. Openhub\_name: The user-visible name produced by Open Hub.
9. Openhub\_description
10. Openhub\_homepage\_url
11. Openhub\_download\_url: Location where the stable version (e.g., tarball) can be downloaded.
12. Openhub\_twelve\_month\_contributor\_count: Number of contributors within the last 12 months.
13. Openhub\_total\_contributor\_count: Total number of contributors according to Open Hub. Note that many old projects will under-report, since this information is captured from version control systems that may not have full histories.
14. Openhub\_total\_code\_lines: SLOC.
15. Openhub\_main\_language\_name: Primary programming language.
16. Openhub\_licences
17. Openhub\_codebase\_factoid: Length of time extant (as text), e.g. “Mature, well-established codebase” or “Well-established codebase” or “Short source control history”
18. Openhub\_activity\_factoid: Activity as measured by commits compared to previous year, e.g., “Increasing Y-O-Y development activity” or “Decreasing Y-O-Y development activity” or “No recent development activity”
19. Openhub\_comments\_factoid: Percentage of comments in source, e.g., “Very few source code comments” or “Few source code comments”
20. Openhub\_devteam\_factoid: Size of development team, e.g., “No recent development activity” or “Only a single active developer” or “Small development team”

## Second stage

We took our early code and modified it to more automatically generate the results (the first version was partly done by hand, to see if the overall approach made sense). Our automated code is implemented in Python, which extracts data from a variety of sources. Data is cached to reduce the impact on servers and speed updates of results.

We also spent time finding additional matches between the Debian and Open Hub lists; in many more cases we found more matches. We then worked to extract CVE data for each project, as this would help us identify what was relevant for security.

The second stage was used to create draft results shown in London in January 2015.



Figure. Analysis process, stage 2

The “projects\_to\_examine.csv” file lists the projects to examine, along with related information (e.g., the keys to use with CVE or OpenHub). The list of projects to consider began with all files in Debian base, and we then added as appropriate. This file is editable; we added comments as appropriate to explain why they were added.

We also created an early scoring mechanism to try to combine the metrics. This was a “risk” score, where a higher value suggests a higher risk. This draft combined score is:

* Project website: 1 point if there is no identified project website (identified by either the Debian database or OpenHub database). There may be a website not identified by the website, but this can be determined later.
* CVE vulnerability reports: 3 points if 4+ , 2 points for 2-3, 1 point for 1.
* main\_language\_name: 2 points for C or C++. Secure programs can be written in these languages, but it is especially easy to make vulnerabilities in them.
* twelve\_month\_contributor\_count: 2 points for 0 contributors, 1 point for 1-3 contributors.
* FactoidTeamSize: 2 points for “No recent development activity” or “Only a single active developer”; 1 point for “Small development team” or unknown.

It’s arguable that the last two items double-count the number of contributors.

This identified several projects that we agreed were concerning, e.g., xauth. Some of the projects identified this this scoring mechanism are less likely to be impact security, however. We believe that part of the problem is that the *exploitability* of a program was not captured at all; we believed adding this data would help distinguish relevant programs.

On January 6, 2015, we received approval from Black Duck to use the Open Hub data in a different way than was usually allowed. They simply wanted to be sure that we would not download and redistribute their database wholesale, and they also wanted to ensure that they received credit (which we are happy to provide). They requested the following attribution: “Data sourced from the Black Duck Open HUB, a free online community resource for discovering, evaluating, tracking and comparing open source code and projects.” In the event that the data is quoted in text, they asked us to identify the source of that data by referring to it at “Black Duck data” or “data from Black Duck.”

At the January 2015 London meeting we received helpful feedback. Many noted the need to stress exploitability or importance (consistent with our thinking). Florian Weimer identified a much better source for package-specific CVE information, as well as a popularity database that might be useful. At the same time, Sam Khakimov determined that the Python program would be easier to understand if it used a more OO-based approach (it had previously simply grown as necessary to gather data, without worrying about the temporary structures used to capture the data). I noted that the code should pull all into memory, or at least into quickly-searched data structures, so that recalculation of results could be quick. Quick recalculation would mean that we’d be willing to try out refinements, instead of simply settling for the first approach that produced data at all.

# Current process

The latest version of our analysis program is written in Python, looks at the file “projects\_to\_examine.csv” to determine what to examine, gathers data from a variety of sources, and merges it into a file called “results.csv” for review. Much of this code was written by Sam Khakimov at IDA. The code is released under the MIT license. The package names are the Debian binary package names; the list include the base Debian packages and others that have been identified as important. This is a small set of the over 37,500 Debian packages.

## Current simplified approach

The following figure shows how the program currently works:



Figure. Analysis process, current stage

Note that some of this data (OpenHub) is sourced from the Black Duck Open HUB, “a free online community resource for discovering, evaluating, tracking and comparing open source code and projects”; we gratefully acknowledge them. Other sources include the Debian package information (obtained from apt), popularity data maintained by Debian, and CVE data provided in yet another Debian location. This version responds to January 2015 feedback in London, e.g., adding popularity score and an improved data source for CVE counts.

Unfortunately many of our data sources are extremely noisy. It is often difficult to determine if there is a match between the Debian database and the OpenHub database (when projects refer to the same home page then there is a clear match, but in other cases it is more difficult). In some cases there is no OpenHub entry at all. Our software works to counter this noisiness, and we hand-reviewed results with higher risk indexes (to reduce the likelihood that the repot included irrelevant work).

## Risk index

This implements a revised risk index:

* Project website: 1 point if there is no identified project website (identified by either the Debian database or OpenHub database). This is given only 1 point because our data sources often fail to identify websites even when then exist.
* main\_language\_name: 2 points for C or C++. Secure programs can be written in these languages, but it is especially easy to make vulnerabilities in them. OpenHub data is used to identify the primary language in many cases; where it is not available, the Debian repository website is scraped to determine the language.
* CVE vulnerability reports: 3 points if 4+ , 2 points for 2-3, 1 point for 1. The CVE count is now direct from Debian, and thus more reliable. CVE counts are a double-edged sword. A number may be small because there are few existing problems, or because few reviewed it; a number may be large because there are many existing problems, or because the software has undergone extensive review. We are using CVEs primarily to help determine the exposure of the program to attack; if several CVEs exist, then it is clearly exposed to attack.
* twelve\_month\_contributor\_count: 5 points for 0 contributors, 4 points for 1-3 contributors, 2 points if the number is unknown (blank).
* Debian popularity count: 1 point if the ‘popularity’ score per Debian is more than tenth percentile of the packages being analyzed. The set of packages being considered is heavily skewed to packages in wide use, and the ‘popularity’ score is noisy anyway (e.g., a single install from Debian from a router supplier might result in millions of uses). Thus, this is primarily used to reduce the interest in projects that appear to be less used.
* Exposure values: 2 points if directly exposed to the network (as server or client), 1 point if it is often used to process data provided by a network, and 1 point if it could be used for local privilege escalation. These values were from expert estimation.
* Application data only: ***Subtract*** 3 points if the Debian database reports that it is “Application Data” or “Standalone Data” as this indicates the package isn’t really code, but is instead application data for code (e.g., “geoip-database”). We believe there is an error in the Debian dataset; “apache2.2-bin” (“binary”) is marked as application data, while “apache2.2-doc” is not, but this doesn’t really matter since binary package “apache2-utils” also maps to the source package “apache2” and so Apache is still considered. Note that “isc-dhcp-common” is marked as data, but “isc-dhcp-client” is not (so the source package isc-dhcp is still considered). Note that “ca-certificates” (a list of certificate authorities) is lowered by this rule.

Areas to potentially improve include:

* Improve/fill in missing data. In general we have messy and incomplete data sources, which required time to address. We already do this in some cases, e.g., to determine the primary implementation language.
* Gather and analyze bug report processing, e.g., how long (on average) does it take to respond to a bug report, and how many bug reports lay unresolved after some time (such as 90 days). This turns out to be harder data to gather, because many projects doe not separate bug reports from enhancement requests. The “isitmaintained.com” site can analyze GitHub data, but not other sites, and it requires that a project use one of the tags it knows about.
* Perform static analysis on source code to determine the likely number of latent vulnerabilities (e.g., using Coverity scan, RATS, or flawfinder); measures such as hit density could indicate more problematic software.
* There are many ways to combine data to find a combined score; this heuristic is our current attempt.

There are many other ways to improve things as described earlier in the literature search.

# Current process results

The file “results.csv” contains the entire set of packages we’ve measured, sorted by the heuristic risk index we created. We first present the list of packages that are considered riskiest, considering only the risk score. We then manually reviewed the packages that the risk score considered riskiest, and selected the ones we believe are *actually* riskier (since humans can estimate other factors such as the likelihood that a program defect will lead to a vulnerability).

## Riskiest OSS programs (straight from scores)

The OSS Debian binary packages with the large two risk values) are as follows:

Table 1. Riskiest OSS Programs (straight from scores)

|  |  |
| --- | --- |
| **Binary package name** | **Source package name (if different)** |
| ftp | netkit-ftp |
| netcat-traditional | netcat |
| tcpd | tcp-wrappers |
| whois |  |
| at |  |
| libwrap0 | tcp-wrappers |
| traceroute |  |
| xauth |  |
| bzip2 |  |
| hostname |  |
| libacl1 | acl |
| libaudit0 | audit |
| libbz2-1.0 | bzip2 |
| libept1.4.12 | libept |
| libreadline6 | readline6 |
| libtasn1-3 |  |
| linux-base |  |
| telnet | netkit-telnet |

## Riskiest OSS programs (estimated by us)

We then examined the top packages as determined by the score, and selected a subset we thought were especially risky. In this process, we were guided at what to look at via the score, but also took into account our knowledge of how the programs are used, and marked those that appeared most risky with “++”; where we were especially concerned, we marked them using “+++”. Here the “Debian source” is the name of the source package (which is the name of the overall project this comes from), while “project name” is the Debian binary package (a source package can generate many binary packages). We did analysis on a binary package level, since there can be distinctions between different binary packages, but the table below re-groups the information into source package names to make this easier to follow.

Projects that we find *especially* concerning, because they appear to be relatively unmaintained even after searching for more information, are the following (we marked these with “+++”):

* **bzip2**
* **gzip**
* **expat (libexpat1)**
* **zlib (zlib1g)**
* **libjpeg8**
* **libpng (libpng12-0)**
* **unzip**
* **mod-gnutls (libapache2-mod-gnutls)**

The “libapache2-mod-gnutls” package is especially concerning; this is a key glue mechanism between Apache and GnuTLS, but there seems to be little maintenance. The others are libraries for processing various formats (images, compression, and XML) that are the basis for many other functions, but do not seem to be well-maintained. These packages appear because our metrics focus on projects that have relatively little maintenance, and either are exposed to the network or are often used to directly process data provided by potential attackers.

The following table is sorted by risk index (where there is a difference):

Table 2. Riskiest OSS programs (human estimate informed by risk measures)

|  |  |  |
| --- | --- | --- |
| **debian\_source** | **project\_name** | **comment\_on\_priority** |
| xauth | xauth | X Authentication - might enable local privilege escalation. Home Page: http://www.x.org/ There are current efforts to replace this (e.g., Wayland, Mir). One nice thing... www.x.org/wiki/Development/Security/ ++ |
| bzip2 | bzip2 | Bzip2 is widely used for decompression of network-supplied data. No code repo, only released code source tarballs. This is the reference implementation, many alternate implementations shown on https://en.wikipedia.org/wiki/Bzip2. Last release in 2010, v1.0.6, in response to CVE-2010-0405. No forums found. At least one HTTP server supports bzip2 for compression. **+++** |
| audit | libaudit0 | Audit framework for Debian, many other components depend on it. Useful for detecting problems. Project infrastructure at Red Hat. SVN repo available. Most recent code change 2 months ago. Mailing lists and IRC exist. v2.4 release 10/2014, plans exist for v2.5 and 2.6. Couldn't find an issue tracker. ++ |
| bzip2 | libbz2-1.0 | Bzip2 is widely used for decompression of network-supplied data. ++ |
| libtasn1-3 | libtasn1-3 | Important because X.509 certificates (standard for certificates) use ASN.1. A Gnu project, git repo and mailing list available from home page. No issue tracker found. Search on 'security' in mailing list archives shows a few messages from Debian using security-related compiler flags. Debian security page shows a history of CVE's being fixed, most recent in 2014. OpenHub page at https://www.openhub.net/p/libtasn1 gives stats. Active effort of one developer, with others contributing as recently as 9 months ago. ++ |
| bind9 | bind9-host, dnsutils, libbind9-80, libdns88, libisc84, libisccc80, libisccfg82, liblwres80 | Bind9 is critical for security. It does have active development. Support is via mailing lists. Very active development, issue tracker intentionally not public...claimed to be "huge backlog" of issues. Public git repo. Active history of CVEs raised and fixed. ++ |
| exim4 | exim4, exim4-base, exim4-config, exim4-daemon-light | This is an MTA, thus has to process untrustworthy data. Mailing lists, bugzilla tracker, public git repo. Active history of CVE issues and fixes. Active with 16 contributors in last year. ++ |
| isc-dhcp | isc-dhcp-client | In wide use, accepts data from external sources. Includes dhclient. Very similar to BIND. Perhaps slightly less active, lower developer count, but steady commits and releases. Somewhat critical due to being core network functionality. ++ |
| gnutls26 | libgnutls26 | TLS implementation. Less used than OpenSSL, but is in use (it avoids OpenSSL licensing issues). Good "security activity" metrics, except suspiciously empty GitLab issue tracker, especially when Debian Security page shows many existing vulnerabilities. They may sweep the public issue tracker so as not to expose what they're trying to fix before it's ready. Notably still supporting SSL 3.0, which has known exploits. There is evidence they do stay on top of things, though. They have responded to questions regarding FREAK in their mailing lists, explaining gnutls is not vulnerable due to how it operates. ++ |
| gpgme1.0 | libgpgme11 | Vital for protecting email (see gpg) Git repo available. Issue tracker site uses self-signed cert; it's concerning that a user would have to compromise the HTTPS security model just to view and interact with the bug tracker. Openhub doesn't reflect their mailing lists in \*multiple languages\*. Appears to be actively maintained mostly by one dev, but notably just got donations supporting him for the next 2-3 years. ++ |
| openldap | libldap-2.4-2 | Wide use, vitally important. Very active. Public repo, issue tracker, lists, active development, closing CVEs, the works. This is vitally important, but they appear to have things in hand in terms of project activity. ++ |
| pam | libpam-modules, libpam0g, libpam-modules-bin | Critical central role for authentication. Real site is http://www.linux-pam.org/ (Debian lists as http://pam.sourceforge.net/ but that points to https://fedorahosted.org/linux-pam/ which points to this). It has a somewhat active mailing list, and an active issue tracker where issues are clearly opened and closed. Stable version is somewhat old (2013); recent changes in git though relatively small (https://git.fedorahosted.org/cgit/linux-pam.git/) ++ |
| openssl | libssl1.0.0 | Critical, in wide use. OpenSSL is already funded by CII. ++ |
| net-tools | net-tools | It's unclear if the Debian package and SourceForge project are still related. This warrants further examination. Upstream listed as https://developer.berlios.de/projects/net-tools/ but has moved to http://sourceforge.net/projects/net-tools/ Has mailing though rather inactive. Has public git repo, latest commit 2015-01-07 on 2015-03-12. These are tools for controlling the network and typically run as root; it's unclear how exposed they are to network attack (most, if not all, are probably not exposed). This lack of exposure probably lowers their priority. ++ |
| openssh | openssh-client, openssh-server | Vital for security. CII already investing in this. ++ |
| openssl | openssl | Critical, in wide use. OpenSSL is already funded by CII. ++ |
| rsyslog | rsyslog | Processes data from untrusted users. Important today; as systemd becomes more used, its use may lessen but still important for merging data sources. Project page http://www.rsyslog.com/ Official repo on GitHub, https://github.com/rsyslog This is an active multi-person project; in the one-month period February 12, 2015 March 12, 2015, excluding merges, 7 authors have pushed 27 commits to master and 40 commits to all branches. On master, 37 files have changed and there have been 677 additions and 216 deletions. ++ |
| wget | wget | Directly processes potentially-malicious data from the Internet, often used in scripts. GNU project. https://www.gnu.org/software/wget/ Actual development on GNU's Savannah using git and its bugtracker, http://savannah.gnu.org/projects/wget/ Actively developed, many patches. wget-1.16.3 released 2015-03-09 on downloads. Note that this is directly exposed to attack. It has some code documentation (e.g., http://wget.addictivecode.org/NavigatingTheSource), but it warns that it can be fairly hard to follow, contains a fair number of hacks, and functionality that was "tacked on" (which is not good for security). ++ |
| apr-util | libaprutil1-ldap, libaprutil1 | Important. The general Apache Portable Runtime (APR) appears to be actively maintained. The LDAP driver is maintained as part of APR, http://apr.apache.org/ The APR-util version 1.5.4 was released September 22, 2014, so it is actively maintained. They use the apache.org repo and subversion; apr-util has been merged into main trunk, making it a little harder to see LDAP specifics to determine if the LDAP portion is active. Mailing list dev@apr.apache.org seems active. ++ |
| coolkey | coolkey | Smart card drivers. Not used in many environments, but critically important for security in other environments. Activity unclear. Managed via Fedora packaging active here: https://apps.fedoraproject.org/packages/coolkey but that is not where the real code is maintained. Important for users of PKCS#11, but slightly specialized. http://pkgs.fedoraproject.org/gitweb/?p=coolkey.git http://pkg-coolkey.alioth.debian.org/ ++ |
| ntp | ntp | Directly connects to network and open to attack. HAS Home Page, NO issue tracker, Has Mailing list, public repo. CII investing. ++ |
| gnupg | gnupg, gpgv | Vital for protecting email (CII has already invested with a one-time investment) ++ |
| gzip | gzip | Widely-used compression/decompression. Vital, a vulnerability here could be very serious. http://www.gnu.org/software/gzip/ Maintained on Savannah (http://savannah.gnu.org/projects/gzip/) using git. Last formal release was years ago (June 2013). Multiple contributors, but not many. Current git contributions, but only a few a month (2015-02-08, 2015-01-02, 2014-11-10, 2014-10-10), and most seem to be small/trivial (document and syntax tweaks). **+++** |
| expat | libexpat1 | Parses potentially-dangerous data. This is an XML parser library written in C. Maintenance appears to have effectively halted after its 2012 release. Project at: http://www.libexpat.org/ (It was at, http://expat.sourceforge.net/ and its movement is not obvious, so some may be using an older version; this was follow-on from James Clark's original Expat). On 24 March 2012 Expat 2.1.0 released, it includes security vulnerability fixes for 5 CVEs; Previous release was 2007. "Bug reports" produces error page - tracking does not seem to work. No public mailing list. CVS browsing suggests little or no activity after 2012 (and little before that). Many systems build on top of this library. Note that this doesn't support XML 1.1. **+++** |
| freetype | libfreetype6 | Dangerous. Processes data from network, including fonts embedded in websites. Main site http://www.freetype.org/ Last release FreeType 2.5.5, 2014-12-30. Git repo. Very active git repo by multiple contributors. ++ |
| libgcrypt11 | libgcrypt11 | Crypto library, security-relevant. LGPL crypto library. Part of GnuPG (GNU Privacy Guard). http://ftp.gnupg.org/gcrypt/libgcrypt/. Already selected by CII. ++ |
| keyutils | libkeyutils1 | Manages authorization and encryption keys required to perform secure operations. Public Repo, NO bug tracker, NO mailing list. ++ |
| xz-utils | liblzma5, xz-utils | Widely-used compression/decompression. Vital, a vulnerability here could be very serious. Has Home page, public repo, Fairly active ML http://www.mail-archive.com/xz-devel@tukaani.org/, NO bug tracker ++ |
| p11-kit | libp11-kit0 | Manages security keys. Has Home page, Bug Tracker https://bugs.freedesktop.org/enter\_bug.cgi?product=p11-glue, ML http://lists.freedesktop.org/archives/p11-glue/ fairly active ++ |
| pcre3 | libpcre3 | Regexes are widely used for security-relevant input validation. Has Home page, Bug Tracker http://bugs.exim.org/buglist.cgi?product=PCRE NOT very active, public repo http://sourceforge.net/projects/pcre/files/ ++ |
| cyrus-sasl2 | libsasl2-2, libsasl2-modules | Authentication library. Has Home Page, HAS bug tracker https://bugzilla.cyrusimap.org/index.cgi, fairly active ML http://asg.andrew.cmu.edu/archive/index.php?mailbox=archive.cyrus-sasl ++ |
| libxml2 | libxml2 | Processes attacker-provided data. HAS Home Page, Has bug tracker https://bugzilla.gnome.org/buglist.cgi?product=libxml2 which is very active. ++ |
| shadow | login | Front door for local login. Has Home Page, part of shadow tool suite, Latest release May 2014, Public Repo, Somewhat active mailing list, NO issue tracker. ++ |
| tar | tar | Widely-used compression/decompression. Vital, a vulnerability here could be very serious. Has Home page, mailing list, Public Repo git://git.savannah.gnu.org/tar.git, uses "technical support" for issue tracking (could be confusing). Does have changes in git repo, but only a few a month and typically small. ++ |
| zlib | zlib1g | Widely-used compression/decompression. Vital, a vulnerability here could be very serious. HAS Home Page, has mailing list zlib@gzip.org, NO issue tracker, Latest release April 2013 (old). **+++** |
| apr | libapr1 | Important. The general Apache Portable Runtime (APR) appears to be actively maintained. However, it's not as clear that the LDAP library in it is as actively managed. HAS Home Page, HAS Mailing List, Issue tracker https://bz.apache.org/bugzilla/enter\_bug.cgi?product=APR \++ |
| libjpeg8 | libjpeg8 | Widely-used format that is decompressed and provided by attackers. HAS Home Page, Can download as tarball, NO issue tracker, latest version/activity Jan 2014, NO mailing list **+++** |
| libpng | libpng12-0 | Widely-used format that is decompressed and provided by attackers. HAS Home Page, Can Access Source code, Not much activity - just warnings about CVEs that were fixed. **+++** |
| libressl | libressl | Fork of OpenSSL. HAS Home Page, Can Download as tarball, VERY Active on Github https://github.com/libressl-portable/portable ++ |
| unzip | unzip | Widely-used format that is decompressed and provided by attackers. HAS Home Page, Can Download Binaries, NO issue tracker. **+++** |
| giflib | libgif4 | Widely-used format that is decompressed and provided by attackers. HAS Home Page, Active issue tracker, Mailing list, public repo. ++ |
| mod-gnutls | libapache2-mod-gnutls | This is in widespread use for SSL/TLS on webservers; it uses GnuTLS instead of OpenSSL. Old website http://www.outoforder.cc/projects/apache/mod\_gnutls/ points to newer site "https://mod.gnutls.org/?outoforder-ref". At newer site the mailing list has a few relevant postings/month, though the March 2015 postings raise concerns about dropped patches. The bug tracker is badly spammed and one seems to be fixing/addressing it. Last release 17-Feb-2014. This looks like it badly needs help. **+++** |
| postfix | postfix | "This is a widely-used mail transfer agent (MTA). Postfix was created by Wietse Venema, main site www.postfix.org. On February 8, 2015 Postfix released 3.0.0, so currently active. Developer mailing list postfix-devel has activity. (e.g., http://news.gmane.org/gmane.mail.postfix.devel/cutoff=2948). There is no public repo; Wietse (the lead developer) does not want to allow any external review before formal release, as noted in https://groups.google.com/forum/#!topic/mailing.postfix.users/6Kkel3J\_nv4 (The Debian package may be an obsolete fork; the Debian package has been assembled by LaMont Jones from sources available from http://www.postfix.org, and can be cloned from git via: git clone git://git.debian.org/~lamont/postfix.git. Changes on http://anonscm.debian.org/cgit/users/lamont/postfix.git/log/ in git stop in 2011.) ++" |
| cryptsetup | cryptsetup | This is used to set up disk encryption (at rest), e.g., Linux Unified Key Setup (LUKS). Home page http://code.google.com/p/cryptsetup , uses git. Tracker in active use. Latest version 1.6.6 released 2014-08-16. Development ongoing; 8 commits in February 2015. ++ |

## Potential improvements to other sites to improve data

Some changes to other sites would make tasks like this much easier:

1. Debian could include in their packages detailed automatically-extractable URLs that identify their upstream source (and if different, the repository they directly use). This information is often in the “copyright” information, but not necessarily in a specific format, and it is often stale (e.g., if the project moves, the Debian data often includes only the old URL).
2. Individual projects should have publicly-visible version control repositories, discussion mechanisms (e.g., mailing list), and issue trackers (security-relevant issues may be temporarily private). Projects can use GitHub, SourceForge, Savannah, and similar systems to easily meet these needs; if they choose to self-host, they should do so in a standardized way so that automated crawlers can gather data across multiple projects. Doing this makes it much easier for third parties to determine if a project is active. There are good reasons to do this anyway; potential users and developers will also want to see this, to give them confidence that the software is being actively maintained by multiple people.
3. Issue trackers should have a clear and standard mechanism for distinguishing bugs from enhancement requests. Bug response times are important indicators for project responsiveness, and mixing them up with enhancement requests makes the data less useful. It would be very helpful if sites like GitHub would build in, by default, some tags for this purpose when projects start.

# Conclusions

This work does not “turn the crank and produce the answers.” Fundamentally there’s a lot of information humans know that isn’t easily captured.

For example, “ftp” (the client) and “netcat-traditional” now come out as high risk; that makes sense, because they definitely interact with the network and thus create opportunities for remote attack. What’s more, the ones used by Debian don’t appear to be very active projects. Are they \*really\* that critical, though? A vulnerability in netcat, while undesirable, is unlikely to have the same impact as a vulnerability in openssl, openssh, or some other programs.

There are also some strategic questions: Mail Transfer Agents (MTAs) scored higher than many other programs in terms of risk, in part because they are directly exposed to attack from the network. Should security investments be made in MTAs, or would money be better spent elsewhere?

Thus, though we provide list of programs identified solely by our metric, we also provide a list of programs that were identified as higher risk, and then selected the most likely ones from them. That is the process we would recommend to others as well. That said, we tried to be guided by the data as much as possible, especially for identifying the larger set to then be examined further. We’ve provided our initial cut here, using this process. The more detailed “results.csv” file includes the complete list of packages and data that we’ve gathered about them.

References

[Callaway] Callaway, Tom. “How to tell if a FLOSS project is doomed to FAIL.” *The Open Source Way*. <https://www.theopensourceway.org/wiki/How_to_tell_if_a_FLOSS_project_is_doomed_to_FAIL>

[Christey2007] Christey, Steve. Unforgiveable Vulnerabilities. 2007-08-02. <http://cve.mitre.org/docs/docs-2007/unforgivable.pdf>

[Crowston2003] Crowston, K., H. Annabi, and J. Howison. “Defining open source software project success.” *ICIS 2003: Proceedings of the 24th Internatinoal Conference on Information Systems* (Seattle, WA). 2003.

[DoD2009] U.S. Department of Defense (DoD) Chief Information Officer (CIO). “Clarifying Guidance Regarding Open Source Software (OSS)”. 2009-10-19. <http://dodcio.defense.gov/Portals/0/Documents/FOSS/2009OSS.pdf>

[Fenton1997] Fenton, N.E. and S.L. Pfleeger. Software metrics: a Rigorous & Practical Approach. International Thompson Press. 1997.

[Gao2011] Gao, Kehan, Taghi M. Khoshgoftaar, Huanjing Wang, and Naeem Seliya. “Choosing software metrics for defect prediction: an investigation”. Software – Practice and Experience. Pp. 579-606. 2011. <http://deepblue.lib.umich.edu/bitstream/handle/2027.42/83475/1043_ftp.pdf>

[Ghapanchi2011] Ghapanchi, Amir Hossein , Aybuke Aurum, and Graham Low. “A taxonomy for measuring the success of open source software projects.” *First Monday*. 2011. <http://firstmonday.org/ojs/index.php/fm/article/view/3558/3033>

[Halstead1977] Halstead, M.H. Elements of Software Science. Elsevier. 1977.

[Hofman2009] Hofmann, Philipp and Dirk Riehle. “Estimating Commit Sizes Efficiently.” <http://dirkriehle.com/wp-content/uploads/2009/02/hofmann-riehle-oss-2009-final.pdf>

[Howard2003] Michael Howard, Jon Pincus, and Jeannette M. Wing. Measuring Relative Attack Surfaces. 2003. http://www.cs.cmu.edu/~wing/publications/Howard-Wing03.pdf

[Manadhata2004] Manadhata, Pratyusa, and Jeannette M. Wing. “Measuring a System’s Attack Surface”. January 2004. CMU-CS-04-102. <http://www.cs.cmu.edu/~wing/publications/tr04-102.pdf>

[Manadhata2007] Manadhata, Pratyusa K., Kymie M. C. Tan, Roy A. Maxion, Jeannette M. Wing. “An Approach to Measuring A System’s Attack Surface”. August 2007. CMU-CS-07-146. <http://www.cs.cmu.edu/~wing/publications/CMU-CS-07-146.pdf>

[McCabe1976] McCabe, T.J. “A Complexity Measure”. IEEE Transactions on Software Engineering. 1976-12. Volume 2, number 4. Pp. 308-320.

[Obama2013] Obama, Barack (in his role as President of the United States). Executive Order 13636—Improving Critical Infrastructure Cybersecurity. The Federal Register. Vol. 78, No. 33. 2011-02-19. <http://www.gpo.gov/fdsys/pkg/FR-2013-02-19/pdf/2013-03915.pdf>

[Punitha2013] Punitha, K. and S. Chitra. Software defect prediction using software metrics - A survey. International Conference on Information Communication and Embedded Systems (ICICES). 2013. 2013-02-21 and -22. http://ieeexplore.ieee.org/xpls/abs\_all.jsp?arnumber=6508369

[Rodriguez2012] Rodriguez, Daniel, Israel Herraiz, and Rachel Harrison. “On Software Engineering Repositories and their Open Problems” <http://promisedata.org/raise/2012/slides/RAISE12_Rguez.pdf>

[Samoladas2004] Samoladas, Ioannis, Ioannis Stamelos, Lefteris Angelis, and Apostolos Oikonomou. “Open Source Software Development Should Strive For Even Greater Code Maintainability.” *Communications of the ACM*. Volume 47 Issue 10, October 2004. Pages 83-87. <http://dl.acm.org/citation.cfm?id=1022598>

[Schryen2009] Schryen, Guido, and Rouven Kadura. “Open source vs. closed source software: towards measuring security.” SAC '09 Proceedings of the 2009 ACM symposium on Applied Computing. Pages 2016-2023. ACM New York, NY, USA. ISBN: 978-1-60558-166-8 doi>10.1145/1529282.1529731. http://dl.acm.org/citation.cfm?id=1529731

[Schryen2011] Schryen, Guido. “Is Open Source Security a Myth?” *Communications of the ACM*, Vol. 54 No. 5, Pages 130-140. 10.1145/1941487.1941516. May 2011. <http://cacm.acm.org/magazines/2011/5/107687-is-open-source-security-a-myth/fulltext>

[Schweik2012] Schweik, Charles M. and Robert C. English. *Internet Success*.

[Shaikh2009] Shaikh, Siraj A. and Antonio Cerone. Towards a metric for Open Source Software Quality. ECEASST 2009. Volume 20. <http://journal.ub.tu-berlin.de/eceasst/article/view/279/287>

[Spinellis2009] Spinellis, Diomidis, Georgios Gousios, Vassilios Karakoidas, Panagiotis Louridas, Paul J. Adams, Ioannis Samoladas, and Ioannis Stamelos. Evaluating the Quality of Open Source Software. Electronic Notes in Theoretical Computer Science 233 (2009) 5–28.

[Springsteen1994] Springsteen, Beth, Dennis W. Fife, John F. Kramer, Reginald N. Meeson, Judy Popelas, and David A. Wheeler. Survey of Software Metrics in the Department of Defense and Industry. April 1994. IDA Paper P-2996. Institute for Defense Analyses, Alexandria, VA. IDA Log No. HQ 94-045746.

[Wang2011] Wang, Huanjing, Taghi M. Khoshgoftaar, and Naeem Seliya. “How Many Software Metrics Should be Selected for Defect Prediction?” 2011. Association for the Advancement of Artificial Intelligence. <http://www.aaai.org/ocs/index.php/flairs/flairs11/paper/download/2558/2993>

[Wheeler2011e] Wheeler, David A. *How to Evaluate Open Source Software / Free Software (OSS/FS) Programs*. <http://www.dwheeler.com/oss_fs_eval.html>

[Wheeler2014g] Wheeler, David A. The Apple goto fail vulnerability: lessons learned. 2014-11-27. <http://www.dwheeler.com/essays/apple-goto-fail.html>

[Wheeler2014h] Wheeler, David A. *How to Prevent the next Heartbleed*. 2014-10-20. <http://www.dwheeler.com/essays/heartbleed.html>

[Wheeler2014n] Wheeler, David A. *Why Open Source Software / Free Software (OSS/FS, FLOSS, or FOSS)? Look at the Numbers!* <http://www.dwheeler.com/oss_fs_why.html>

[Woody2014] Woody, Carol, Robert Ellison, and William Nichols. Predicting Software Assurance Using Quality and Reliability Measures. December 2014. CMU/SEI-2014-TN-026. <http://resources.sei.cmu.edu/library/asset-view.cfm?assetid=428589>

Acronyms and Abbreviations

API Application Programmer Interface

BY Attribution (a Creative Commons license)

CC Creative Commons

CII Core Infrastructure Initiative

CVE Common Vulnerabilities and Exposures

CVSS Common Vulnerability Scoring System

DHS Department of Homeland Security

FLOSS Free/Libre/Open Source Software

FS Free Software

FSF Free Software Foundation

GCC GNU Compiler Collection

GNU GNU’s Not Unix

GPG GNU Privacy Guard

GPL (GNU) General Public License

GTRI Georgia Tech Research Institute

HOST Homeland Open Security Technology

IDA Institute for Defense Analyses

KSLOC Thousand Source Lines of Code

LF Linux Foundation

MIT Massachusetts Institute of Technology

MTA Mail Transfer Agent

NIST National Institute of Standards and Technology

NTP Network Time Protocol

NVD (NIST) National Vulnerability Database

OS Operating System

OSI Open Source Initiative

OSS Open Source Software

OWASP Open Web Application Security Project

PAM Pluggable Authentication Module

REST Representational State Transfer

RPC Remote Procedure Call

SLOC Source Lines of Code

SSL Secure Sockets Layer

TLS Transport Layer Security

Y-O-Y Year-Over-Year

1. Those who doubt this can just look at http://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=wordpress, http://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=joomla, and http://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=drupal. [↑](#footnote-ref-1)