INSTITUTE FOR DEFENSE ANALYSES

Core Infrastructure Initiative (CII) Open Source Software Census II Strategy

David A. Wheeler

Jason N. Dossett

Final

October 31, 2017

Approved for public release; distribution is unlimited.

IDA Document D-8777

INSTITUTE FOR DEFENSE ANALYSES
4850 Mark Center Drive
Alexandria, Virginia 22311-1882

About This Publication

This work was conducted by the Institute for Defense Analyses (IDA) under contract HQ0034-14-D-0001, Task LX-5-3968, "Linux Foundation Core Infrastructure Initiative," for Linux Foundation Core Infrastructure Initiative. The views, opinions, and findings should not be construed as representing the official position of either the Department of Defense or the sponsoring organization.

Acknowledgments

Clyde G. Roby

For more information:

David A. Wheeler, Project Leader
dwheeler@ida.org, 703-845-6662

Margaret E. Myers, Director, Information Technology and Systems Division
mmyers@ida.org, 703-578-2782

INSTITUTE FOR DEFENSE ANALYSES

IDA Document D-8777

Core Infrastructure Initiative (CII) Open Source Software Census II Strategy

David A. Wheeler

Jason N. Dossett

Executive Summary

This paper proposes a strategy for quantitatively identifying the most important open source software (OSS) projects among the millions managed by language-level package managers and, of those, identifying the ones most needing security-related investments.

Our previous paper [Wheeler2015] quantitatively identified some important OSS projects in need of security-related investments, but it focused on only a select set of OSS managed by system-level package managers. Here, we instead focus on ensuring that we can cover the much larger set of OSS projects that are managed by language-level package managers. This larger scale required a different and more automated approach.

We propose a strategy for estimating importance primarily through dependency analysis, that is, by identifying, for a project, what else depends on it (directly or indirectly). We propose estimating security risk by identifying a set of risk indicators, heuristically assigning a weight to each one, and then totaling those weights to determine their risk. We then propose to combine these scores by selecting the "top most important" projects, then sorting those most important projects by risk.

To determine the feasibility of this approach, we developed a prototype that did some basic dependency analysis, computed a simple risk indicator, and then reported the combination. This paper briefly describes the prototype and lessons learned by applying it. The analysis execution time is highly dependent upon implementation; by changing our implementation strategy, we transitioned from days to less than an hour. While dependency analysis can work, we found that there are subtleties in versioning that must be carefully handled to produce accurate results. We also found other issues, e.g., challenges in vendoring and in handling Python dependencies that might need to be addressed.

This paper then presents an updated approach to identify importance and security risk in a more rigorous quantitative way, based on the lessons learned from the prototype. We do not expect this approach to eliminate the need for human judgment, but rather it will provide rigorously supported data to help make good judgments.

As part of our process and data collection, we computed a lower-bound estimate of the number of significant OSS projects, so we report that as well. As a lower bound, there are at least 3.26 million significant OSS projects.

We conclude that this strategy is feasible, and we recommend that the Linux Foundation (LF) Core Infrastructure Initiative (CII) implement this strategy to help it quantitatively determine the most important projects to assist.

Contents

1. Introduction

The Heartbleed vulnerability in the open source software (OSS)[1] program OpenSSL was a serious vulnerability with widespread impact. Yet Heartbleed could have been detected in many ways before it was deployed [Wheeler2014h]. The Heartbleed vulnerability highlighted the fact that the vulnerabilities in some widely used and depended-upon OSS programs can have serious ramifications, and yet some OSS projects have not received the level of security analysis appropriate to their importance. Some OSS projects have many participants, perform in-depth security analyses, and produce software that is widely considered to be of high quality and to have strong security. However, other OSS projects do not.

The Linux Foundation (LF) Core Infrastructure Initiative (CII) was established to "fund open source projects that are in the critical path for core computing functions [and] are experiencing under-investment."[2] The LF CII will make final decisions on what it will invest in, but it wants to base those decisions on quantitative data. The LF CII asked us to try to quantitatively determine what OSS projects are "most important," and of those, which ones most need security-related investments. The LF CII can then choose what to invest in and how. For example, it might choose to fix, refurbish, or fund a replacement of that software. It might also fund related work such as process improvements, hardening efforts, or training (such as on testing or on developing secure software). In all these cases, however, the first step is to identify the projects that appear to most need investment.

Our previous work, *Open Source Software Projects Needing Security Investments* [Wheeler2015], analyzed a set of OSS projects to help identify especially plausible candidates for investment to improve security. This work was well received, and it supported various initial CII investment decisions. However, like all work, it had limitations. That work considered only software packaged by system-level package managers[3] and ignored language-level package managers. The previous work depended

[1] Open source software can be defined as "software for which the human-readable source code is available for use, study, reuse, modification, enhancement, and redistribution by the users of that software" [DoD2009]. For more information, see the Open Source Definition [OSI].

[2] Per the Core Infrastructure Initiative (CII) page at http://www.linuxfoundation.org/programs/core-infrastructure-initiative

[3] A package manager automates the process of installing and otherwise managing packages. A package is a unit of software that can be installed and managed by a package manager. There are different kinds of package managers. A system-level or operating-system-level package manager manages the packages

on humans to identify which packages might be important (and then ranked the subset). It also included human-level analysis (e.g., to report whether a package had external connections), which is helpful but is hard to scale up to a large number of projects.

In this paper, we focus on quantitatively identifying the most important OSS projects, and of those, the ones most needing security-related investments. However, unlike [Wheeler2015], here we focus on including the set of OSS project results managed by language-level package managers. This is a much larger number of projects than we considered before. Since many of these projects are low-level "invisible" components, and there are many more of them, we believe humans are less likely to be able to determine *a priori* the most important projects. In addition, the sheer scale requires a more automated approach. Thus, we focused on developing entirely automated mechanisms to estimate importance and security risk.

We propose a strategy for estimating importance primarily through dependency analysis, that is, by identifying, for a project, what else depends on it (directly or indirectly). We propose estimating security risk by identifying a set of risk indicators, heuristically assigning a weight to each one, and then totaling those weights. We then propose to combine these scores by selecting the "top most important" projects, then sorting those most important projects by their total risk indicator. However, there are legitimate questions about whether or not this strategy is feasible, so we implemented a prototype of part of the strategy.

This paper first presents some background in Chapter 2, particularly for issues beyond those already covered in our previous paper. Chapter 3 describes the prototype we developed that did some basic dependency analysis, computed a simple risk indicator, and then reported the combination. In Chapter 4 we describe the lessons learned through the prototype. Chapter 5 then describes an updated approach, building in part on the experience gained from the prototype. Chapter 6 presents a lower bound estimate of the number of significant OSS projects, since we had the data to compute this. This paper ends with our conclusions.

for an entire operating system instance; an example is Debian's Advanced Package Tool (apt). A language-level or application-level package manager manages the packages for a specific programming language, programming environment, or application; an example is the JavaScript's npm.

2. Background

Our first step was to review methods and approaches already available. Our previous work, *Open Source Software Projects Needing Security Investments* [Wheeler2015], provides a lengthy survey of approaches related to gathering quantitative data to estimate importance and security. Below are a few approaches that are worth restating or were not mentioned previously (e.g., because they did not exist then). These are grouped into the following categories: data sources (including metrics tools and general information on metrics), measuring importance, and measuring security risk.

A. Data Sources

The following are a few sources for data (including metrics tools and general information on metrics):

- GrimoireLab <http://grimoirelab.github.io/>. This is a toolsuite for acquiring OSS project metrics. In particular, its Perceval component performs data gathering from OSS projects. Key parts of this are written in Python 3. This toolsuite is OSS and was developed by Bitergia (co-founded by Jesús M. González-Barahona). It stores information in ElasticSearch.
- OpenHub <https://www.openhub.net/>. This was formerly named Ohloh and is managed by Black Duck. It provides a variety of metrics for OSS projects.
- BigQuery <https://cloud.google.com/bigquery/>. This is a data warehouse for large-scale data analytics. A number of relevant databases are already available via BigQuery, so performing data queries against it is extremely convenient. BigQuery has a pay-as-you-go pricing model for queries. Relevant databases include:
  - GitHub archive <https://cloud.google.com/bigquery/public-data/github>. This provides GitHub-related data.
  - GHTorrent <http://ghtorrent.org/>. This stores GitHub events.
- Community Health Analytics for OSS (CHAOSS) <https://wiki.linuxfoundation.org/chaoss/metrics>. This group is working to identify OSS-related metrics.
- Grafeas. This is an open-source application program interface (API) to audit and govern a software supply chain [Elliott2017].
- Libraries.io <https://libraries.io/>. Libraries.io has parsed data from a wide variety of language-level package managers, and makes this freely available in a consistent format at <https://libraries.io/data>.

- Apache Kibble <https://kibble.apache.org/>. Apache Kibble is a suite of tools for collecting, aggregating and visualizing activity in software projects.

B. Importance

Measuring the importance of software is not easy. In our previous work, humans identified the initial set of projects to consider, and this does not scale.

The Battery Open-Source Software Index (BOSS Index) [Thakker2017] attempts to identify important projects using a set of four metrics. However, their index focuses on 40 projects, which we believe is too small a starting point.

GitHub has “stars,” which users can select to show appreciation. A project with stars is probably important to at least a few people. However, there is little evidence that the number of stars strongly correlates to real importance, and “stars” only apply to projects on GitHub (not all OSS projects are on GitHub).

As we discuss in Chapter 3, we use dependency analysis to estimate importance.

C. Security

Measuring the security of software is a notoriously difficult and essentially unsolved problem. Automated analysis of code can sometimes detect vulnerabilities, but false positives and false negatives make interpreting their results difficult. Project activity is not necessarily an indicator of security; a relatively inactive project might be feature-complete, while an inactive project might ignore security issues. If a project has several past vulnerabilities, it might indicate that the software has serious security problems, or it might indicate highly secure software with a large number of external reviewers. Learning algorithms require trustworthy training data sets, which are not available. A good way to analyze software security is to attempt to penetrate the software, but this approach is notoriously variable since it depends on the skills and approaches taken by the attackers; it is also extremely expensive and time-consuming.

Our previous work, *Open Source Software Projects Needing Security Investments* [Wheeler2015], provides a lengthy survey of these issues, particularly of the various approaches for estimating the security of software. In our previous work, we developed a useful approach for estimating security risk by identifying a set of risk indicators, heuristically assigning a weight to each one, and then totaling those weights. This is a practical method of quickly identifying projects that appear to be especially risky.

In this paper we build on the previous work, but discard measures that require human analysis (since they do not scale) or were difficult to acquire at scale. In Chapter 3 we discuss in further detail the metrics we selected.

3. Prototype Analysis Approach

In this chapter we describe our prototype analysis approach, in particular, how we identified the importance and security risk of various OSS projects.

A. Importance Analysis

The first step in the analysis was to quantitatively identify the most important OSS projects.

One approach to finding "important" components of a larger system is to identify the most important missions (fundamental purposes) of that system and then drill down into the system's design to determine what components are required to meet those missions. One name for this kind of analysis is "criticality analysis." This can work well when developing specific systems; however, it is harder to apply to larger systems (like nations or humanity). A nation could attempt to do this (e.g., the U.S. Government's critical infrastructure sectors), but that would require significant resources and risks overlooking key components. In the future, it might be possible to merge the information based on missions, but this seems like a difficult place for us to start.

One of the most obvious ways to quantify importance is to ask, "How many people use this?" If a lot of people use some particular software, then it is important. For example, if that software was compromised, a large number of people would have their systems compromised. Unfortunately, it is hard to tell exactly how many people use some particular software. One could look at the number of times some particular software was downloaded. However, using download counts has two problems:

1. Download mirrors and OS-level package managers are prevalent, especially for OSS software, and these can produce severe undercounts.
2. In continuous integration (CI) systems, software may be downloaded every time a test is executed, producing severe overcounts.

In short, it is hard to get plausible counts for actual end-user downloads. However, we believe there is an effective way to estimate usage without depending primarily on download counts.

First, we can focus on packaged software. By definition, OSS projects produce software. A project typically has a repository for storing and managing the history of the software it produces. In addition, this software may be packaged. Since software is easier to install and use if it is packaged, it is extremely likely that important software will be packaged. Thus, we will focus on ranking the importance of packaged software, aka packages. Since our previous work focused on software packaged using system-level packagers, here we will focus instead on software packaged using language-level packagers (such as npm). This means that we must determine the relative importance of language-level packages. (We could later add packages managed by system-level package managers, as discussed in section 5.C.)

Second, we can use dependency analysis to analyze packaged software. An easily quantifiable measure is to look at how many OSS repositories and other packages depend on a given OSS package. This is the approach chosen in this analysis. If a package is depended on by a large number of repositories and other packages, it is much more likely to have a higher number of end users ultimately dependent on it, thus the package is important. However, various details must be addressed, as discussed below, to make these counts reasonable to use.

1. Obtaining a Package's Dependencies

a. Types of Dependencies

Before we begin describing the data set we used, it is first useful to distinguish between direct dependencies and transitive dependencies. Direct dependencies are those which, as the name implies, a project relies on directly ("A depends on B" is a direct dependency). Transitive dependencies, on the other hand, include dependencies that a project indirectly depends on. For example, if "A depends on B" and "B depends on C" then C is a transitive dependency of A.

We must trace the transitive dependencies everywhere we can in order to calculate the number of packages and repositories that depend on a given OSS package. This approach can help reveal "hidden" dependencies that may not otherwise be obvious.

Software can have different dependencies depending on its environment and kind of use. Dependencies required by end-users to run the software are called **runtime dependencies.** These can be the same or different from those required when testing, developing, or even compiling the software.

Runtime dependencies are not the only kind of dependency that matters for security. A savvy attacker could compromise other kinds of dependencies, such as those for a test suite or set of development packages, in order to inject malicious code into other packages. However, the runtime dependencies are directly included when the software is run, so runtime dependencies are subject to attacks that other kinds of dependencies are not. Accordingly, in our importance analysis, we recommend looking at two cases, one for which we trace only runtime dependencies and one for which we trace dependencies regardless of the kind of dependency.

b. Libraries.io Data Set

To calculate the number of packages that depend on a given OSS project, we use the publically available Libraries.io data set [Libraries.io]. This data set includes dependency information from two type of sources. First, it gives dependency information obtained from various language-level package managers (npm, Pypi, Maven, etc.). Second, it also includes dependency information obtained by crawling through source repositories on Github, GitLab, and Bitbucket. This data set is available under the Creative Commons Attribution-ShareAlike 4.0 International (CC BY-SA 4.0) license. We gratefully acknowledge Libraries.io for providing this preprocessed data set, which we used as a starting point for our analysis.

The data set is quite extensive, but unfortunately has a few shortcomings. The most significant issue is that only direct dependencies are given for the majority of packages. The Libraries.io data set does include some data on transitive dependencies. However, this data set includes only the transitive dependencies that are directly declared (e.g., in a Gemfile.lock file). Therefore, we could not simply use the raw dependency data as the full set of transitive dependencies, and instead had to calculate transitive dependencies.

For this reason, we developed a set of Python scripts that would determine transitive dependencies of packages. Another program was then used to count the number of transitive dependencies for each OSS package available.

Another thing we had to consider is that many of the packages that have dependency information coming from package managers also have dependency information coming from source repositories. To avoid double-counting dependencies, we prioritize dependency information coming from package managers. Fortunately this is easy with the Libraries.io data set since source repositories for the packaged software are documented. In our analysis, if a source repository has a corresponding package in a package manager where dependency information is given, we do not use any dependency information from the source repository.

2. Filtering Out Insignificant Repositories

One of the challenges of dealing with dependency information coming from source repositories such as GitHub is the presence of repositories that are insignificant. Some examples of insignificant projects are homework assignments, temporary experiments, or someone’s toy project that they quickly created and abandoned. Including these types of repositories in our importance analysis could skew results in favor of projects that might not actually be as widely used as the numbers would imply.

To avoid including insignificant repositories in our analysis, we came up with indicators that could help determine whether a repository was insignificant. We created two sets of indicators: indicators of significance and indicators of insignificance. If a repository satisfied any of the indicators of significance it was included in our analysis. We dropped projects with too many indicators of insignificance (in our case, more than three of six indicators).

a. Indicators of Significance

As the name implies, if a repository has any indicators of significance it is most likely a real, active project, and is automatically considered a significant project. Five indicators of significance were used:

1. The repository has a “star” count > 50.
2. The repository has a “watchers” count > 50.
3. The repository has a fork count > 20.
4. The repository has a contributor count > 10.
5. The repository is the source repository for any project in one of the package managers analyzed.

If a repository satisfies any of the indicators of significance, it is included in the dependency analysis as a significant project.

b. Indicators of Insignificance

It can be difficult to determine whether a repository is insignificant, so we took a measured approach. We used a total of six indicators of insignificance to help determine whether a source repository was insignificant. If any project satisfied **more than three** of these indicators of insignificance **and** did **not** satisfy any of the indicators of significance (above) it was considered insignificant, and thus not included in the dependency analysis. The indicators of insignificance are:

1. The repository has 0 forks, less than 3 watchers, and less than 3 stars.
2. The repository has less than 3 open issues.
3. The repository’s homepage URL is empty.
4. The repository is the fork of a significant repository (as described above).
5. The repository has not been updated in at least a year **and** the time between its creation and last update is less than 6 months. (This is typically true for repositories representing homework projects.)
6. The repository has no license and no license file.

B. Security Analysis

As with [Wheeler2015], we estimated security risk by identifying a set of risk indicators, heuristically assigning a weight to each one, and then totaling those weights. The higher the risk value, the more risky the project appears to be.

Here are the measures we used in our prototype:

1. **No project website**. If there is no identified project website, 1 point of risk is assigned. This is given only 1 point because our data sources often fail to identify websites even when they exist. We also used this metric in [Wheeler2015]. Future versions could in addition check that the claimed website is still responding (if the website is no longer there, that is naturally a risk).

2. **Unchanged for a long time**. This indicates a project that has not had a commit in a long time. If the last commit is >3 years ago, it receives 4 points; if >2 years ago, 2 points; if >1 year, 1 point, and >6 months or unknown, 1 point. A project might be unchanged if it is feature-complete. However, a project that has not been updated in a long time is likely to be moribund and its security is probably not being continuously reviewed. This was calculated using the "last_pushed_at" field from libraries.io. Note that these times are in reference to the download date of the input data set, not the date the analysis was performed.

3. **Twelve-month contributor count**. In [Wheeler2015] we counted risk based on the number of recent contributors (called twelve_month_contributor_count). Zero contributors, 5 points; one contributor, 4 points; two to three contributors, 2 points, unknown (blank) number of contributors, 1 point. If there are multiple contributors, even if it's minor, there is clearly some collaboration, and that can reduce risk. However, this is harder to measure than we had thought. BigQuery stores commits in an easily parsable way for repos that have been deemed "open source" according to their license. However, this is not very accurate. We used BigQuery's githubarchive data set and obtained author and repo data for two types of events: "PushEvent" (this is a commit to the repository of any kind); and "PullRequestEvent" with a payload that includes ""action":"open"" (this is the opening of any pull request on the repo). We think that we can justify any pull request being opened counting as a contribution even if it is not accepted. Projects that do not have a github repository will have an unknown number of 1-year contributors so they will automatically receive 1 point.

4. **No contributing instructions**. If the field contributing_name is null, empty, the letter "f" (representing false), or has no known repository, there is 1 point of risk. This occurs if the project has no CONTRIBUTING file or similar; this file is a common way to indicate to new developers how to contribute to the project.

Failure to provide this information suggests that it may be more difficult to contribute to the project.

5. **No license file**. If the "license" value is empty or null, it receives 1 point of risk. Legal issues can dissuade security review. We used the libraries.io license detector because it seemed to detect licenses in more circumstances. A project might have a valid OSS license without a license file, but lack of a clear file does suggest it may be harder to determine.
6. **Unmaintained**. If "status" is "Unmaintained" it receives 3 points. This is determined by looking for the word "Unmaintained" in the project short description, which is a relatively common convention for unmaintained projects. Some projects include "Deprecated" but we decided to ignore this, since projects can be deprecated for a long time yet still be actively maintained.

We did not use these measures from [Wheeler2015] and do not recommend using them in later versions either:

1. **Debian popularity count**: In [Wheeler2015], if the "popularity" score per Debian is more than the tenth percentile of the packages being analyzed, it received 1 point. We could use a large number of stars or downloads to indicate somewhat similar information. However, in [Wheeler2015] this was really a way to identify importance, not security risk. Since we measure importance differently, it is no longer needed and doesn't make sense to use.
2. **Exposure value**. In [Wheeler2015], we used humans to determine the exposure of the software to attack. If the software was directly exposed to the network (as a server or client), 2 points; if it was often used to process data provided by a network, 1 point; and if it could be used for local privilege escalation, it also received 1 point. This is human-provided, and had to be dropped to support the larger scale of this work.
3. **Application data only**. In [Wheeler2015], if the Debian database reports that it is "Application Data" or "Standalone Data," subtract 3 points. This was because this indicates that the package isn't really code but is instead application data for code (e.g., "geoip-database"). There was no obvious corollary in this case.

C. Combining Importance and Security

We need to combine these measures of importance and security. Multiplying them together would be absurd, since the units would make no sense. Fundamentally, both the importance (as counts) and security risk (as a risk score) are *relative* measures.

Instead, we selected the “top 200” packages of each package manager (and for all OSS packages), then sorted them by risk to show the riskiest most important projects. This way of combining values avoids multiplication by unknown units.

4. Lessons Learned Through Prototyping

The purpose of the prototype was to quickly learn things so that we could more effectively implement the full analysis. Of course, our prototype did not include everything the full analysis would include. For example, it omitted a number of metrics, so that we could implement the prototype quickly. See Chapter 5 for details on enhancements we would like to make in the full analysis.

In this chapter, we identify key lessons learned from the prototyping effort. These are how to efficiently implement dependency analysis, the importance of handling dependency analysis with versioning, the challenges of vendoring, and other miscellaneous issues.

A. Efficiently Implementing Dependency Analysis

We first needed to ensure that we could efficiently implement dependency analysis at scale. After determining that several alternatives were poor approaches, we quickly decided on a highly parallelized approach that produces results in a timely way.

Some computer science textbooks discuss how to implement dependency tracing by storing the data as two-dimensional arrays of Boolean values and using matrix operations to determine dependencies. However, we immediately realized that this would be a terrible implementation approach given the size of our data set. In Chapter 6 we show that there are at least 3.26 million significant OSS projects; a single square two-dimensional matrix of Boolean values would require 1.33 Terabytes (TB), which requires a lot of memory and parallelizes poorly. Using a two-dimensional matrix to represent this data is a poor fit anyway, because the dependencies are sparse on this scale.

Our first version of the prototype cached dependency results as they were determined, which effectively implements a sparse matrix. Sometimes caching can help performance, and sometimes it doesn't; since caching was easy to do, and we thought it might be beneficial, we decided to try it first. However, in this situation, caching created more problems than it solved. First, the cache had to be shared among multiple processes, and this led to extremely input/output (I/O) bound processes. Second, the process had to constantly check to prevent duplication, which was computationally expensive.

The final version of the prototype simply loads all the direct dependencies as a constant graph. This is still a kind of sparse matrix, but instead, we do not store any interim results in memory at all. Instead, for each package and repository, we independently recalculate all its transitive dependencies without caching. This does mean that some calculations are duplicated; however, it also makes the problem extremely parallelizable because the parallel processes don't need to communicate often with each other and the results can go immediately to files (not memory). This approach does require a little more memory at the beginning compared to the previous approach, since each parallel process needs a complete copy of the direct dependency tree. Unlike the previous approach, however, memory use did not increase over time. It would have been possible to share this tree, reducing the initial memory requirements, but that would have required much more programming effort and turned out to be unnecessary. This alternative was unsurprising. In a parallel system it's often (but not always) the case that recalculation of results, instead of caching them, produces faster results…but often the only way to determine the best approach is to prototype it.

This final approach allows us to produce results in a timely way. The first prototype approach, which used caching, took 5 full days to traverse only two-thirds of the packages on 15 processors to compute the transitive dependencies. The final prototype version takes 39 minutes, 6 seconds to compute the transitive dependencies using 15 processors. We could have made this slightly faster; the code was in Python, which we could have compiled using a different implementation (such as PyPy or Cython) or rewritten in a different language. However, the entire process is now I/O bound, so rewriting it would probably not significantly decrease the execution time, and in any case this execution time is acceptable.

B. Versioning

One of the assumptions we made when approaching the importance analysis is that packages would tend to keep the same dependencies over time and that any deviations from this would have a very small impact in our analysis. As such, when calculating a package's transitive dependencies in our prototype analysis, we included all direct dependencies of a given package without controlling for version information. Unfortunately, the assumption that version information would have little effect was wrong.

The shortcomings of not including version information in our transitive dependency analysis were particularly evident when looking at Rubygems packages. For example, half of the top 20 riskiest packages based on combining security metric scores and importance (see section 3.C) were only ranked highly because versioning was not accounted for. The gem "spruz" is a great example of this. In our importance analysis, spruz shows ~975,000 dependents. It is the 10th ranked package by our importance measure and has a high risk score of 9. The only reason spruz is ranked so highly is because previous versions of several very popular gems (json_pure for example) depended on it. Had we included version information in our tracing of transitive dependencies, “spruz” would not have been ranked nearly as high for importance. A similar problem happens with “needle.” The very popular net-ssh package at one time depended on needle. The net-ssh package no longer depends on needle, but this isn't reflected in our current process because we intentionally ignored versioning in the dependencies.

To better illustrate the problem that arises from not including version numbers, consider the simple set of dependency relations shown in Figure 1. The circles indicate specific versions of a package or repository; boxes are drawn around multiple versions of the same underlying package or repository. If we do not consider versions, then when we trace transitive dependencies, we would say that A transitively depends on C; this is clearly not the case. Q has the same problem as well – without considering versions, we would conclude that Q depends on F when it really does not. In reality, the dependency relations are much more complex, so inaccuracies from not including versioning get out of hand quite quickly. This is particularly important when considering security risk. As seen with Rubygems, many of the high-risk packages would no longer be considered important had we properly considered versions.

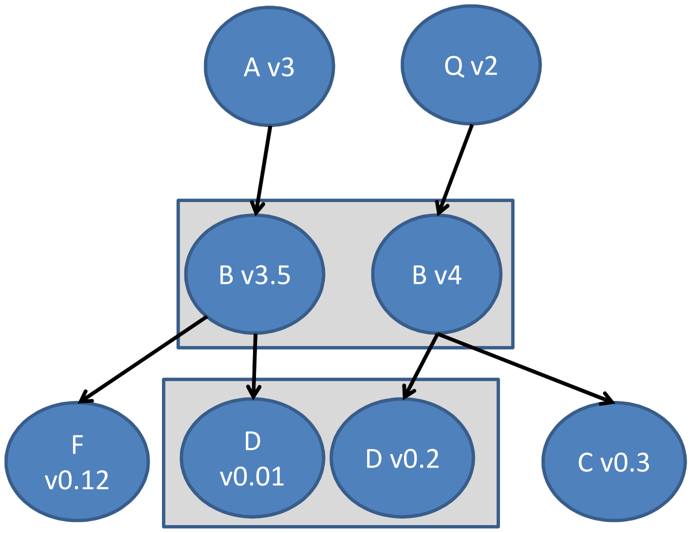

Figure 1. Notional dependencies demonstrating versioning problem

A potential solution is to change the algorithm as follows. First, for each component with dependencies, consider only the latest version. Second, trace through the dependencies, but *consider* the versions for each components being depended upon. Finally, for each package, count the number of dependents (packages or repositories) that depend on that package *ignoring* the version of the package that is depended upon.

If we apply this modified algorithm to the notional dependencies shown above, the following packages would have these dependents (sorted by number of dependents):

- D: B, Q, A;
- C; B, Q;
- B: Q, A;
- F: A.

Note that F is depended upon by A, because A depends on a version of B that in turn depends on F. By contrast, F is not depended upon by B or Q. Although a version of B does depend on F, the latest version of B does not depend on F, so B is not included in F's dependents. Q does depend on a version of B, but the version of B that Q depends on does not depend on F.

We believe this changed algorithm more accurately reflects dependencies.

C. Vendoring

Another issue that could cause potential miscounting of dependencies (and thus importance) is vendoring. Vendoring is the complete inclusion of a package by copying it inside another package. Rather than using a package manager to make sure dependencies are installed, some packages will just copy their dependencies in their codebases. While ignoring versioning can make some packages look more important than they are, not properly accounting for vendoring can make a package look less important since a vendored package won't show up in the dependency tracing.

A good example of vendoring causing issues in our prototype analysis was the vendoring of PyPI package *chardet* into the *requests* package. The *requests* package was very highly ranked in our importance calculations so one would expect *chardet* to, likewise, be highly ranked in importance. This was not the case since *chardet* was vendored into *requests*. While requests was ranked 3rd in importance for PyPI packages in our prototype run, *chardet* was ranked 266th. Note that this was true at the time the PyPI package manager data for the libraries.io data set was compiled. As of v2.16.0, *requests* no longer vendors its dependencies.

A complication is that in some sense ignoring vendored packages is correct. If package 1 is vendored inside package 2, then even if package 1 is updated, there is no reason a priori that package 2 will be updated.[4] Instead, when a package that is vendored is updated, the other copies are often *not* updated. However, the fact that the vendored software is *not* updated creates a heightened security risk.

[4] Vendoring is different from a "convenience copy." In a convenience copy, source code is copied into another package, but it is used only when the original package is unavailable. This situation was more common when Internet access was scarce, but is relatively rare today. The key is that users will typically use the updated software when there is merely a convenience copy, and that is simply not the case with a vendored package.

Several commercial organizations develop origin analysis tools (tools that identify software components in a larger system). We know that at least one of these tools (Synopsys' Protecode) can identify vendored software, and we suspect some others can too. If vendored software is to be detected, then partnering with one of those organizations, using one of their tools, or both could help resolve this.

D. Other issues

1. Python Dependency Parsing

During our prototype analysis, we also noticed a small bug in the Libraries.io dependency parser for PyPI. This parser, among other things, should look for files like: "req*.txt", "req*.pip", "requirements/*.txt" and "requirements/*.pip". Unfortunately, bugs in the implementation of the identification of files matching that pattern caused many unwanted files to be thought to have dependency information. This led to many "junk" dependencies that didn't really exist. Luckily, our importance and security analysis was unaffected because the names were inconsistently derived and thus were individually rare.

Tracing down this issue was made easier by the fact that the Libraries.io source code is open source software. Had that not been the case it would have been much harder to track down what was causing identification of so many strange dependencies. We have reported this issue to the libraries.io team by submitting it on the librariesio/bibliothecary repository on GitHub.[5]

2. Including Test and Build Dependencies

During our prototype analysis, we looked at the impact of including development and test dependencies along with the runtime dependencies. As touched on previously, an attacker could modify an important test framework to subvert a large number of packages that use it. If we don't include test and development dependencies in our dependency analysis, we will not be able to see packages that are important to testing and development, leaving us with a possible blind spot.

Unfortunately, including development and test dependencies in a second run through of our importance analysis didn't help identify important test and development packages. This was due to the cyclic dependencies that arise when including test and development dependencies. For example, once a test framework is included, there are typically a number of other packages that are depended upon to test the test framework

[5] See https://github.com/librariesio/bibliothecary/issues/404

itself. So many packages end up cycling back to one another that our importance measures become almost meaningless.

A better approach for identifying important test and build dependencies might be to include a project’s direct dependencies regardless of environment, but when tracing the transitive dependencies, traverse only subsequent runtime dependencies. We suspect that this would resolve these issues, if following test and build dependencies is considered important.

5. Updated Analysis Strategy

Based on past experience, here is our updated strategy for identifying important OSS with heightened security risks. The strategy is divided into three topics: importance analysis, security analysis, and other issues.

A. Importance Analysis

Experience with our prototype suggests that dependency analysis is a practical and defensible way to identify important OSS projects. The prototype also made it clear that it's important to deal with versioning, as described in section 4.B. See that section for our proposed approach to addressing versioning.

In our prototype we simply considered every OSS repository equally as our starting point for dependency analysis. Of course, not every OSS repository is equally valuable. That said, if their collective dependencies are representative of real use, this is still a reasonable approximation. If better data cannot be found, then we recommend starting there.

Ideally we would not weight OSS repositories equally. Instead, ones that are more widely used, or in some other way are "more important," should be weighted more heavily. As noted earlier, however, download counts and stars are not reliable measures. Download numbers are deflated because they are redistributed through caches and other systems, and they are inflated because they may be re-downloaded by continuous integration (CI) systems.

We have begun trying to get additional data on which OSS is actively being used, and by how many products or people, through a variety of organizations. If we can get a representative set of this data, we could then use that data as a weighting factor. This could be combined with counting every OSS repository as having at least one use to help address cases in which the additional data missed something important. That said, the analysis could proceed even without this improved additional set.

For a start we could continue to focus on "run-time" dependencies, since those are the components that execute at run-time. That said, test and build tools are important, so it would be useful to run a separate analysis to include them as well.

B. Security Analysis

As noted in [Wheeler2015], the current state of security metrics for software is quite limited. Instead of waiting for a perfect theoretical basis, we want to provide enough useful information to make reasonable decisions.

As always, an underlying problem with our security measurement approach is that it is based on heuristics – the selection of metrics, and their weights, is by expert estimate – instead of a more fundamental theoretical basis. This limitation is further discussed in [Wheeler2015]. Should better metrics be identified to us or become available, they could be used instead or in addition. For example, CHAOSS is working on improved metrics. The weights could be adjusted based on additional information and review. In addition, the weights could be scaled across a range, instead of simply being integer values.

Our current plan is to create an estimated "risk score" (where a higher value indicates higher risk), just as we did successfully in [Wheeler2015]. We would start with the risk metrics discussed in section 3.B (e.g., projects that are unchanged for a long time have a higher risk). Here are some additional risk measures we propose using:

1. **CVEs.** In [Wheeler2015] we examined the Common Vulnerabilities and Exposures (CVE) catalog. If there were >=4 CVEs, 3 points; 2–3 CVEs, 2 points; and 1 CVE, 1 point. As noted in [Wheeler2015], CVE counts are a double-edged sword. The number of reports may be low because there are few existing problems or because few reviewed it; the number may be high because there are many existing problems or because the software has undergone extensive review. In [Wheeler2015] we used CVEs primarily to help determine the exposure of the program to attack; if several CVEs exist, then it is clearly exposed to attack.

   However, it is a challenge to match CVEs to OSS projects. There are databases that simplify this for specific languages (e.g., Bundler-audit's data set for Ruby). The National Vulnerability Database (NVD) includes Common Platform Enumeration (CPE) identifiers, but the CPE information doesn't have enough information to reliably match the information to OSS projects (e.g., it lacks project URLs). Various organizations do work to develop this mapping, so we believe we can partner with one or more of them to get this data. In the longer term, it would be good to more clearly tie CVE data to OSS projects, e.g., by putting URLs in CPEs, so that organizations wouldn't need to recreate this data.

2. **Weakness Density.** We could use a static analysis tool, such as Synopsys Coverity, to measure the number of weaknesses, aka vulnerability findings or potential vulnerabilities. The weakness density can then be computed by taking the number of weaknesses and dividing that by the number of lines of code. If it is in the top 1% (for that package manager), 5 points; else if it is in the top 10%

(for that package manager), 3 points; else if it is in the top 50%, 1 point. While any particular weakness (finding) may not be a true vulnerability, if the density of weaknesses is unusually high for that language, that suggests that the developers were not actively taking steps to avoid risky constructs (from a security point of view). We expect software with an unusually large number of risky constructs to be more likely to have more vulnerabilities.

3. **Big.** Larger software systems are more likely to have defects, and some of those defects may be security vulnerabilities. In addition, it's difficult to understand larger programs. Thus, simply being larger increases the risk of security vulnerabilities. A trivial way to estimate this is to use the "size" parameter, which is the size of a repository in KB. If it is in the top 1% (for that package manager), 3 points; else if it is in the top 10% of size (for that package manager), 2 points; else if it is in the top 50%, 1 point. Directly using size has its problems because size includes test suites and documentation, and those are both indicators of *reduced* risk. A better way to measure size would be to measure non-test source lines of code (SLOC); we could use one of various SLOC counters to better estimate size.

4. **C/C++.** In [Wheeler2015], if the main programming language used was C or C++, 2 points of risk were assigned. Secure programs can be written in these languages, but it is especially easy to make vulnerabilities in them. Libraries.io does provide a "lang" field that can provide this. However, if we focus on only language-level package managers, this is unlikely to matter, because C/C++ programs are often installed using system-level package managers instead. Therefore, this should be implemented only if system-level packages are included.

5. **Contributors.** This considers the total number of contributors over all time. If there was only 1 contributor in the entire history of the project, 4 points; 2–3 contributors, 3 points; 4–10 contributors, 2 points; and an unknown number of contributors gets 1 point. This means that if a project does not have a repository then it will automatically get 1 point. Unfortunately this field in the libraries.io data set is unreliable. There are projects that have many more contributors but they show up as 0 in the data set. Thus, we do not use this for now.

6. **Average number of commits per month.** If there are a non-trivial number of commits per month, particularly in the last year, that suggests real activity. If the average number is in the lowest 10%, then assign 1 point of risk. Note that this measurement is correlated with some other measures, such as the number of contributors in the last 12 months.

7. **Pull (merge) requests ignored.** If more than 90% of all pull requests ever made are still open (neither closed nor merged), and there have been at least 10 pull requests, then assign 1 point of risk. The idea is that if pull (merge) requests are almost always ignored, that suggests that external suggestions or improvements are generally ignored. Note that a project may choose to reject many pull requests (perhaps they are poor quality); the issue here is ignoring them entirely. Note that GitHub calls these "pull requests" but other systems use other names such as "merge requests."
8. **Issues (or similar) ignored.** If issues are almost always ignored, that's not a good sign. If more than 90% of all issues ever opened are still open, and there have been at least 10 issues filed, assign 1 point of risk. Unclosed issues are not always bad, of course. However, a very high ratio of open issues to closed issues is a very good indicator of projects with a high level of risk. Our thanks to Jesús M. González-Barahona for noting this.
9. **No issues opened in the last year.** If no issues were opened in the last year, assign 1 point of risk. If nobody cares about opening tickets, either the project is really rock solid, or nobody cares. Our thanks to Jesús M. González-Barahona for noting this.
10. **Commercial support for development.** If no developer in the past year has contributed to the project primarily within that developer's business hours, assign 2 point. If most contributions in the past year are not within business hours, assign 1 point. Commercial support is hard to directly measure by automated means. However, if at least one developer who is a main contributor is routinely committing during business hours within their region[6] in that developer's time zone, that suggests direct funding of at least one developer. This is not perfect, since retirees and students might be included, but it's probably still a useful indicator.
11. **CII Best Practices badge.** A project that has earned at least the "passing" level best practices badge is actively performing a large number of activities that reduce risk, and should have 3 risk points removed. This can be easily determined using the badging project's REST interface.
12. **Lack of testing.** It should be possible in many cases to detect test code; if there's no evidence, give it 2 risk points. This can often be done by detecting various configuration files for continuous integration platforms and common test

[6] In many time zones, this is typically something like Monday–Friday 0700–1800. This obviously varies between countries, e.g., in Saudi Arabia, business hours are more likely to be something like Saturday–Wednesday 0800–1200 and 1500–1800 [ExpatFocus]. As part of this analysis, we would need to estimate business hours for each time zone.

framework indicators. A project that has a CII Best Practices badge (at least "passing") also has a testing process. One challenge with this indicator is that some tests may not be detected (particularly in cases where the test framework is maintained as a separate project, instead of being part of the project itself). More risk points might be appropriate, but we hesitate to do that because of the challenge of detecting tests.

Of course, some testing is better than others. It might be possible to measure the ratio of test code to regular code (a measure that Rails easily provides), or even test coverage (such as statement coverage or branch coverage). However, these test quality measures are more difficult to acquire, so they might be better saved for later refinements.

C. Other Issues

The approach for merging the importance and security data appears to work well. Again, we identify the "top" most important OSS (at least for each package manager), and then sort them by risk scores to find the "riskiest important software."

We have decided to focus on language-level package managers, since that was not covered in [Wheeler2015]. To cover all OSS packages, in the future we would also want to add system packages from some system package manager (such as Ubuntu, Debian, or Fedora). Loading dependency data from a system package manager is not difficult since that data is already available. The larger challenge is connecting all the dependency data because there often is no connection between the language-level package managers and the system-level package managers. We might be able to compensate for this by using data from sources such as Dockerfiles for a variety of widely used containers. That said, we think this would be a useful direction to build on.

One particular kind of software is implicitly important but not always identified as such: package managers themselves. Anyone using a language-level package manager obviously depends on it, but this dependency is not always captured by the package managers themselves. Others have noted the need for security in package managers, e.g., [Arizona2014] (focusing on system-level package managers) and [Athalye2014]. Package managers are clearly key to installation and maintenance, but they do not always receive the security attention they should for such a prominent role.

As noted in section 4.C, vendoring is challenging because it hides uses of software. Some tools are specifically designed to find these situations, and they might be useful in later versions to expand the accuracy of the analysis in the future.

6. Number of OSS Projects

We were able to quickly estimate the number of significant OSS projects, given the data sets we had already gathered to implement the prototype. In this chapter we present our results, and our justification for it.

We have determined that there are *at least* 3.26 million significant OSS projects. We believe that there are more OSS projects; this is merely a lower bound.

This estimate is a sum of two values:

1. GitHub and GitLab projects with an automatically identified license file, removing insignificant projects (e.g., homework projects) as described above and projects licensed under CC-BY-NC-ND (which is not an OSS license). This is 2,759,655 projects.
2. All projects packaged by language-level package managers that are not already covered by #1. This is 501,849 projects. If it's packaged, it's usually licensed as OSS and somebody cared enough to package it, so we didn't filter that further.

Of course, not all projects involve the same level of effort. The Linux kernel and the trivial JavaScript (npm) package left-pad both count as one project each. Nevertheless, this estimate is certainly adequate to show that there are a lot of OSS projects.

This estimate doesn't include many projects:

1. Some OSS project licenses are not easily detected (e.g., the license information may be embedded in a README file), so this estimate does omit some OSS projects. However, we must have some way to deal with a common problem: many projects on GitHub have no license at all. Unlicensed projects are not OSS because under most copyright laws worldwide express permission must be granted to allow copying (including redistribution) and modification. We have intentionally chosen a conservative approach – if it's not clear through our tools that the software is OSS, we presume it is not OSS.
2. Many projects manage their own repositories, or use repository hosting services other than GitHub or GitLab (such as SourceForge, Bitbucket, and Savannah). That said, GitHub is the most commonly used repository hosting service, so its data is a good place to start.

This estimate does not include some system-level packages. However, as of October 10, 2017, the number of system-level packages in Ubuntu's "artful" is 72,721 (72727-6),[7] and some of those packages are included in the earlier estimate. Even if all of those system-level packages were added, it would not significantly change the result.

[7] This is from the list of Ubuntu packages, https://packages.ubuntu.com/artful/allpackages?format=txt.gz

7. Conclusions

This paper provides evidence that it is feasible to try to identify the “most important” OSS packages, and from them, identify packages that most need investment for security. Our prototype efforts did help identify some challenges, but lessons learned from the prototype suggest how those challenges can be overcome.

Some OSS is very secure; some is not. The Heartbleed vulnerability in OpenSSL showed that even some widely used OSS needs more investment in its security. This kind of analysis can help to identify the most important OSS that most needs security-related investments. The LF CII could then choose what to invest in and how. The LF CII might choose to fix, refurbish, or fund a replacement; it might also fund work such as process improvements, training, and hardening efforts. But the first step is to do this wider analysis. Performing the full analysis, and investing in OSS projects identified using it, could prevent many serious security breaches that might otherwise occur.

Appendix A
Package Managers

Here is a list of package managers that manage the packages, the primary language or platform supported by each, and the number of packages managed by each package manager that something else depends on (per our data set). These are the package managers for which we have data from Libraries.io <https://libraries.io/data> and thus were analyzed by our prototype.

Table 1. Package managers

Package manager	Primary language or platform	# of used packages
npm	JavaScript	241,080
Maven	Java	41,722
Packagist	PHP	39,608
Rubygems	Ruby	32,402
NuGet	.NET (C#)	27,926
Pypi	Python	27,091
Bower	JavaScript (client side)	15,858
Go	Go	12,681
CPAN	Perl	10,599
CocoaPods	Objective-C, Swift	5,740
Cargo	Rust	3,239
CRAN	R	2,957
Clojars	Clojure	2,687
Hex	Erlang, Elixir	1,282
Pub	Dart	1,065
SwiftPM	Swift	637
Julia	Julia	538
Elm	Elm	353
Dub	D	323
Haxelib	Haxe	260
Meteor	JavaScript	149
Atom	Atom (editor)	76
Shards	Crystal	14

We omit Carthage because it is not really a platform (it’s a decentralized package manager for Cocoa).

For more information on many different package managers, including the full count of packages they manage over time, see <http://www.modulecounts.com/>.

Acronyms and Glossary

API: application program interface

BOSS: Battery Open-Source Software

CC-BY-SA-4.0: Creative Commons Attribution-ShareAlike 4.0 International; see https://creativecommons.org/licenses/by-sa/4.0/

CC-BY-NC-ND: Creative Commons Attribution-NonCommercial-NoDerivatives; see https://creativecommons.org/licenses/by-nc-nd/4.0/

CHAOSS: Community Health Analytics for OSS

CI: Continuous Integration

CII: Core Infrastructure Initiative

CVE: Common Vulnerabilities and Exposures, a catalog of known security threats.

I/O: Input/Output

LF: Linux Foundation

npm: (Historically) Node.js Package Manager. Today it is simply used as the name.

Open Source Software: "software for which the human-readable source code is available for use, study, reuse, modification, enhancement, and redistribution by the users of that software" [DoD2009]. For more information, see the Open Source Definition [OSI].

OSS: Open Source Software

package: A unit of software that can be installed and managed by a package manager.

package manager: Software that automates the process of installing and otherwise managing packages.

repository: A location for storing and managing the history of information (such as software).

REST: REpresentational State Transfer

SLOC: Source Lines of Code

TB: Terabyte (10^{12} bytes)

vendoring: The complete inclusion of a package by copying it inside another package.

Bibliography

[Athalye2014] Athalye, Anish, Rumen Hristov, Tran Nguyen, and Qui Nguyen. Package Manager Security. https://css.csail.mit.edu/6.858/2014/projects/aathalye-rhristov-viettran-qui.pdf

[Arizona2008] Arizona University. *Attacks on Package Managers*. https://www2.cs.arizona.edu/stork/packagemanagersecurity/attacks-on-package-managers.html

[DoD2009] Department of Defense (DoD). October 2009. *Clarifying Guidance Regarding Open Source Software (OSS)*. http://dodcio.defense.gov/Portals/0/Documents/OSSFAQ/2009OSS.pdf

[Elliott2017] Elliott, Stephen and Jianing Guo. October 12, 2017. "Introducing Grafeas: An open-source API to audit and govern your software supply chain." *Google Cloud Platform Blog*. https://cloudplatform.googleblog.com/2017/10/introducing-grafeas-open-source-api-.html

[ExpatFocus] ExpatFocus. "Saudi Arabia - Business Culture" Retrieved 2017-10-19. http://www.expatfocus.com/expatriate-saudi-arabia-business-culture

[Libraries.io] *Libraries.io Open Data*. Libraries.io. Licensed under the Creative Commons Attribution-ShareAlike 4.0 International (CC-BY-SA-4.0) license. https://libraries.io/data

[OSI] Open Source Initiative (OSI). The Open Source Definition (Annotated). Version 1.9. https://opensource.org/osd-annotated

[Thakker2017] Thakker, Dharmesh, Max Schireson, and Dan Nguyen-Huu. April 7, 2017. "Tracking the explosive growth of open-source software." *Tech Crunch*. https://techcrunch.com/2017/04/07/tracking-the-explosive-growth-of-open-source-software/

[Wheeler2015] Wheeler, David A. and Samir Khakimov. June 19, 2015. *Open Source Software Projects Needing Security Investments*. IDA Document D-5459.

REPORT DOCUMENTATION PAGE	*Form Approved OMB No. 0704-0188*

1. REPORT DATE (DD-MM-YY) 31-10-17	2. REPORT TYPE Final	3. DATES COVERED (From – To)

4. TITLE AND SUBTITLE Core Infrastructure Initiative (CII) Open Source Software Census II Strategy	5a. CONTRACT NUMBER HQ0034-14-D-0001
	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBERS
6. AUTHOR(S) David A. Wheeler, Jason N. Dossett	5d. PROJECT NUMBER LX-5-3968
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESSES Institute for Defense Analyses 4850 Mark Center Drive Alexandria, VA 22311-1882	8. PERFORMING ORGANIZATION REPORT NUMBER D-8777
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) Mike Dolan Linux Foundation Core Infrastructure Initiative 1 Letterman Drive, Building D, Suite D4700 San Francisco CA 94129	10. SPONSOR'S / MONITOR'S ACRONYM LF CII
	11. SPONSOR'S / MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION / AVAILABILITY STATEMENT

Approved for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES

Project Leader: David A. Wheeler

14. ABSTRACT

This paper proposes a strategy for quantitatively identifying the most important open source software (OSS) projects, and of those, the ones most needing security-related investments.
We propose a strategy for estimating importance primarily through dependency analysis, that is, by identifying for a project what else depends on it (directly or indirectly). We propose estimating security risk by identifying a set of risk indicators, heuristically assigning a weight to each one, and then totaling those weights. We then propose to combine these scores by selecting the "top most important" projects, then sorting those most important projects by their total risk indicator.
To determine the feasibility of this approach, we developed a prototype that did some basic dependency analysis, computed a simple risk indicator, and then reported the combination. This paper briefly describes the prototype and shortcomings we discovered by applying it. In particular, while dependency analysis can work, we found that there are subtleties in versioning that must be carefully handled to produce accurate results. We also found that there are special challenges in handling Python dependency data that must be addressed.
This paper then presents an approach to identify importance and security risk in a more rigorous quantitative approach. This is based on additions and modifications to the prototype approach, based on our discoveries and other information available. As part of our process and data collection we computed a lower-bound estimate of the number of significant OSS projects, so we report that as well.
We conclude that this strategy is feasible, and we recommend that the Linux Foundation Core Infrastructure Initiative (CII) implement this strategy to help it quantitatively determine the most important projects to assist.

15. SUBJECT TERMS

Open source software, OSS, Free software, security, software security, software assurance, importance, computer security, risk, Linux Foundation, Core Infrastructure Initiative, Census, dependency analysis

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON Mike Dolan
a. REPORT Unclassified	b. ABSTRACT Unclassified	c. THIS PAGE Unclassified	Unlimited	38	19b. TELEPHONE NUMBER (Include Area Code) 1-408-890-8600