Mining commit log messages to identify risky code
Greg Barish, Matthew Michelson, and Steven Minton
InferLink Corporation, El Segundo, CA, USA

Abstract - Software vulnerabilities and other risks continually emerge as new code is introduced or existing code is modified. The effect of these risks can be disastrous, not only for companies and organizations that provide this software, but also for those that use such software. However, the risks of new/modified code could be potentially mitigated if it were possible to reliably scan all code upon commit. In this paper, we describe a novel approach that leverages prior commit log messages as one means for training a system to automatically flag new commits. Our data-driven approach is designed to complement the hard-wired approaches that most static and dynamic code analysis tools use today. We demonstrate our approach in the context of two major existing projects: Apache Web Server (httpd) and Apache Tomcat, both popular web containers used by hundreds of thousands of organizations.

Keywords: machine learning, security, vulnerabilities, software engineering

1 Introduction

Over the past few years, the profile of security breaches, hacks, and the leveraging of software weaknesses has risen dramatically. It is not uncommon to find a story of such failures in the headlines of national or world news. From the Sony hack [4], to the Ukraine power grid disruption [8], to ransomware incidents [6], news of software security breaches has risen to the headlines-level. Many of these events are, at their core, due to weaknesses or vulnerabilities in software. Exploitation of these vulnerabilities is frequently at the root of, or least part of the chain of an attack.

The existence of vulnerabilities in software is simply a function of human error. For example, one common type of vulnerability is a buffer overflow, which can be introduced when a programmer either forgets or does not realize that there is no validation on input from users or external resources, and thus it is possible for an attacker to leverage lack of bounds checking on the input and thus gain control to unallocated areas of memory. Another type of vulnerability is use-after-free, which occurs when a programmer errantly uses memory that has already been released back to the operating system and is thus being used illegally in the code from that point forward (and, as such, can be exploited similarly to buffer overflow). Both of these errors are due to a lack of precaution by a programmer, who was either lazy, not knowledgeable of risks, or simply overlooked some concerns in the rush to release code.

Interestingly, some vulnerabilities are not due to direct actions by the programmer, but by indirect actions, such as the inclusion of open-source third-party libraries into a project. Leveraging of these libraries has become common practice, facilitated by open source aggregators like GitHub and BitBucket, and there are literally hundreds of thousands of these libraries that can be plugged into other projects. However, open-source third-party libraries can have their own vulnerabilities, and when they do, it can affect thousands of software projects built upon that library. Thus, it has become of interest to identify use of libraries that are at risk.

Ideally, all types of vulnerabilities could be detected by automated mechanisms before they are released to consumers of that software. Some of this is possible today, thanks to code analysis tools like FindBugs [1]. However, these tools are language-specific (i.e., not portable) and are not flexible, depending purely on a pre-defined set of hard-wired syntax or semantic rules.

While static and dynamic code analysis can be helpful tools, we can potentially do more to mitigate code risks. One such approach is to take a document-oriented perspective and look at the source code as one would a set of documents. Just as one could build a text classifier based on terms in a labeled set of "training" documents, so could one potentially apply the same approach to build a text classifier for risky code.

For example, use of a variable named "password" can be a key indicator that such code could be risky. In particular, an expression like:

\[ \text{String password} = \text{"secret";} \]

can be particularly alarming because it implies that the variable password is not only being assigned, but is being assigned to a hardcoded value, visible to anyone who looks at the source code. Static analyzers would typically not be applicable to this need because they do not focus on the semantics of the terms involved and because the number of rules required to handle such cases, and textual variations thereof (e.g., "adminPassword") would become unmanageable.

In contrast, one could envision a data-driven approach that leverages a set of labeled code to, for example, discover the set of terms that are correlated with security fixes. Knowledge of these terms and their prediction likelihoods could be used to evaluate unlabeled code. Thus, new code checked in could be automatically scanned to see if it contained risky indicators, such as a term like password.

The core intuition to this line of thinking is that there is likely to be signal in how humans refer to data and behavior in code. For example, people will tend to label variables as "password" or "buffer" if they are about passwords or buffers (both frequent culprits in vulnerabilities). Functions like encrypt() will likely be more risky than functions like capitalize(). Thus, human language pervades code and can provide signal that we can potentially detect automatically.

In this paper, we describe a document-oriented approach towards identifying risky code that is designed to complement...
(not replace) existing tools like static and dynamic analyzers. This rest of this paper is organized as follows. We first qualify risky code in more detail in Section 2 of this paper. We then discuss an approach to identifying risky code in Section 3, and then present initial findings applying our approach to two well known open source Web containers (Apache Server and Apache Tomcat) in Section 4. Finally, we compare our approach to related work and identify next steps in Sections 5 and 6.

2 Risky code

Let us first elaborate on the nature of risky code. We define risky code as new/modified code that is explicitly labeled with sensitive terms/phrases (e.g., "security risk") or bears sufficient similarity to past code labeled risky. Source code commit messages are convenient labels to explore because programmers tend to describe the nature of their change as part of committing the code. There are other possible sources, such as code comments, that could be included, but we focus initially on the commit log messages.

Explicitly labeling code as risky varies based on project and organization, but there are some obvious demarcations. One such explicit sign is the association of fix with a particular "CVE" (Common Vulnerability and Exposure) identifier. CVEs are the standard mechanism for labeling vulnerabilities. MITRE, along with the National Vulnerability Database, is in charge of assigning CVEs based on request (or delegation).

When a commit is marked as associated with a CVE, such as "fixed CVE-2014-0050", we can reasonably assume that the code checked in is relevant to the fix. For example, in the Apache Tomcat project, the commit shown in Figure 1 was the fix for this particular CVE.

When such commits are made, code is added, deleted, or changed. That code which has been deleted or changed is typically the offending code, and thus the provably risky (or flawed code). For example, in the CVE-2014-0050 case, the updated code includes the change shown in Figure 2.

The figure shows code changes related to how multi-part streams are handled (adding in exception catching for boundary conditions). The lines with a "+" have been added, the lines with a "-" have been deleted. From a document-oriented perspective, terms like MultipartStream, boundary, and input were terms that were associated with this change and can thus be flagged as indicators of risk. If we explore the larger set of code in Tomcat related to CVEs, we can potentially identify the common terms that indicate risk. For example, we may find that case-insensitive subsequences like SSL, password, username, login, SQL, etc. are common indicators of risk. In short, we can look at the challenge of risky code identification as one of traditional text classification, in which we are looking to build a classifier for commits that may contain subsequences associated with past risk.

Unfortunately, most security fixes are not conveniently associated with a CVE, and thus the set of code to analyze could be small (or non-existent) for certain projects. Many organizations, especially smaller third party projects, do not know about CVEs or do not file for them. Furthermore, many vulnerabilities are found and fixed internally in a project. These issues are usually part of the set of the other (non-security-related) bugs associated with the project. Nevertheless, such issues are no different than CVEs in terms of impact if exploited.

However, commit messages related to code changes to address non-CVE security issues can vary. Sometimes, a log message might be "XSS fixes" or "ensure bounds checking to prevent buffer overflow", and so forth. As mentioned earlier, internal standards, language, and even the type of project can influence what terms are used.

As we describe in Section 3, our approach allows the terms of interest to be configured as part of risky code detection, based on the needs of an organization.

3 Learning to identify risky commits

To determine if we can identify risk indicators based on security-related commit messages, we built a generic processing pipeline that automatically learns which source code terms are associated with such commits. We then applied that knowledge to classify previously unseen commits, in order to flag those commits as risky – even if they do not have a commit message that would obviously indicate this risk.

The pipeline is shown in Figure 3 and has the following high-level phases, which we discuss in detail below:

1. Gather all commits that are labeled with explicit security terms (e.g., "CVE", "security fix", etc.)

2. From these commits, aggregate source code that has been removed (i.e., we presume that there was

![Figure 1: Commit header for CVE-2014-0050 fix in Apache Tomcat](image-url)
something "wrong" about such code, since it was removed/changed).

3. Tokenize and normalize the resulting set of source code (e.g., deconstruct terms like "adminrPassword" into "admin" and "password", as well as sequences like "javax.net.ssl.SSLSocket" into "javax", "net", "ssl", and "socket")

4. Remove stopwords, including language operators and keywords (such as void, return, if/then, while, for, etc.)

5. Learn probabilistic model of terms/phrases that predict risk, based on the tokenized source code, so that other non-obvious commits can be flagged as potentially risky

Gathering all the commits for a given project is relatively straightforward. Most code today is source controlled using git [5] and we can easily search commit messages that contain terms/phrases of interest, such as "CVE" or "security fix". This returns a set of git commit hashes, which we can then use to pull "diffs" for those commits.

These diffs contain information about changes to source code and non-source code, such as documentation or graphical assets (e.g., images). The first step is to discard any aspect of the commit that is not source code. Within the set of remaining code, we can evaluate the diff in detail to identify which code has been added or removed. Added code can be seen as "good code" because it is part of the "fix". Removed code can be seen as potentially "bad code" because the fix involved removing, or at least modifying it. Thus, our next step is to extract the set of code that has been removed, via information from the diffs.

The third step is to tokenize and normalize the resulting suspicious code. Tokenization is based on whitespace, but also on camel case and language specific markers (e.g., the periods in "javax.net.ssl.SSLSocket", the parentheses that indicate function declaration, and so forth). An interesting decision is whether to include source code comments (e.g., "/* login to external system */"). For this work, we ignored such comments. However, there is potentially rich value in this data, as we revisit in the conclusion of this paper.

The resulting terms are then normalized by converting everything to lower case. We then remove stopwords, a process similar to traditional stopword removal in documents [3], with the exception that we can also use the language reserved word dictionary as an automatic first pass filter. More specifically, we ignore all language keywords and operators, since those are not assumed to have semantic hints ( unlike variable or function names).

The final step was to leverage the resulting code to learn models that predict risk. To do this, we learned Naïve Bayes (NB) classifiers [7] based on the resulting bag-of-words representation of the remaining code. NB classification was chosen because it is a common baseline strategy used for text classification that generally yields good results. Once learned, the NB classifiers allowed us to flag unlabeled commits as risky or non-risky based on code "terms".

4 Initial results

To evaluate the feasibility of our approach, we applied our framework to the corpus of code from two popular projects: Apache Tomcat and Apache Web Server (httpd). Both of these projects are used by hundreds of thousands of organizations. While both projects have commits related to CVEs, the vast majority of security fixes made are not labeled with a CVE.

More specifically, as the table below shows, both Apache Tomcat and Apache Web Server have relatively few commits with specific reference to a distinct CVE. In contrast, both

| Search: | Git-based project | Commit parser / code-deleted identification | Code cleaner (stopwords, stems, etc) | Bag of words | Naive Bayes model |
| "CVE" | "security" | "SSL", etc. | | |

Figure 3: Code analysis pipeline
contain many more references to "secur" (stem of "security" and "secure") and "SSL", which could be clear indicators of risk/sensitivity.

<table>
<thead>
<tr>
<th></th>
<th>Apache Tomcat</th>
<th>Apache Web Server</th>
</tr>
</thead>
<tbody>
<tr>
<td>cve</td>
<td>44</td>
<td>56</td>
</tr>
<tr>
<td>secur</td>
<td>183</td>
<td>171</td>
</tr>
<tr>
<td>ssl</td>
<td>541</td>
<td>1296</td>
</tr>
</tbody>
</table>

Given this reality, our goal was to see if we could learn to predict commit events for security related terms (like SSL or security/secure), terms that might be obvious flags of risk. More specifically, can we learn what code terms we correlated with those log messages? If we can, then we can potentially predict code risk even without the label, which would allow us to build a better commit filter. Thus, if one committed code with the message of "fix for bug 12", we can look at the code itself to estimated actual risk.

To explore this, we applied the code analysis pipeline described in Section 3 to both Apache projects. Again, we focused on the "secur" and "ssl" examples, since there was very little data purely on CVEs from both. In doing so, we tested our NB models using a standard 10-fold cross validation technique. Specifically, for each project, we collected the commits from positive ("risky") cases and an equal number of commits of negative ("non-risky") cases. The non-risky commits were chosen from a random set of commits that did not include any of the risky commits.

A summary of the results appears in the table below. The table show the effectiveness of classifiers at predicting log messages with same semantic intents (e.g., "security fix") in unlabeled code. In particular results below show that a simple NB classifier was able to predict risky code 79% of the time on Apache Tomcat (where "risky" = commit messages that contained the prefix "secur"), and 73% on Apache Web Server. Results for "ssl" were similar, but lower.

<table>
<thead>
<tr>
<th></th>
<th>Apache Tomcat</th>
<th>Apache Web Server</th>
</tr>
</thead>
<tbody>
<tr>
<td>secur</td>
<td>100</td>
<td>79%</td>
</tr>
<tr>
<td>ssl</td>
<td>200</td>
<td>71%</td>
</tr>
</tbody>
</table>

Our study is not comprehensive, but it does illustrate the feasibility of our approach, that it is possible to detect risky, unlabeled code based on learning from terms associated with explicitly labeled risky code. More generally, that we can predict certain properties of the code added/changed/removed based purely on log messages, which intuitively would make sense.

5 Related work

As mentioned earlier, there has been significant attention paid to static code analysis techniques as a way to detect code risks [2]. Static analysis techniques vary, from looking at lexical constructs in the code to evaluating program semantics. There are several open source tools for static code analysis for various languages, including FindBugs for Java, RIPS for PHP, and OWASP LAPSE+, and FlawFinder for C++. Commercial tools are produced by companies such as HP/Fortify, IBM, and Veracode.

In addition to static analysis, dynamic code analysis involves examining code in the runtime environment, including evaluating compiled code, or at least code compiled to an intermediate form. While there are less options for open source dynamic analysis tools, there are multiple commercial offerings.

One recently popular form of analysis, applicable to both static and dynamic paradigms, is taint analysis, which looks at variables which have been mutated and the sources of those mutations. Interestingly, some languages – such as Perl and Ruby – have taint analysis built in as a feature of the language.

Each of these analysis techniques leverage library of things to look for (e.g., syntactic or semantic rules/constructs). As such, they are typically applicable to specific languages and not usually portable. In contrast, our work here ignores the syntactic and semantic aspects of the code, simply focusing on the code as a document. Our approach is inherently portable because most of the labels used for functions and variables reflect human-level modeling/choices and this is largely the same no matter which language is used, though we note that our detection of language-specific function names or third-party libraries does not make our approach fully-portable.

We view our efforts in this work as complementing, not replacing static and dynamic code analysis, since the features being analyzed are independent. Ideally, our approach can be used in conjunction with other analyzers to produce a better overall analysis, increasing recall while reducing false positives.

6 Conclusion

In this paper, we have described the need to predict risky code as a way to help automatically mitigate potential vulnerabilities, before they are released. We have presented a text classification approach to identifying risky code. Our approach involves building a simple Naïve Bayes classifier per the terms associated with risky commits. Our approach is highly portable because it does not depend on understanding or compiling the language; rather, it focuses on the way programmers describe data and behavior. We have shown that preliminary testing on two popular web container projects, Apache Web Server and Apache Tomcat, yields encouraging results.

In future work, we aim to look at many more popular open source repositories to see how our results generalize over a larger set of software. We also intend to see if an aggregate bag-of-words approach is more powerful. For example, can we process security-related commits in Apache Tomcat and use that knowledge to better predict risky code in a similar project (without training on that project specifically)? Finally, we intend to explore how other predictive models (multinomial NB, n-gram, SVM, etc.) fare, in contrast to a basic Naïve Bayes approach.

7 References

conference on Object-oriented programming systems and applications companion (pp. 805-806). ACM.


