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 indicidents [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 mechnanisms 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:

```
String password = "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 contast, one could evision 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