Predicting Security Vulnerabilities in Software Components through Code and Meta-level Analysis

1 Introduction

Software vulnerabilities are security-related bugs software that may be exploited by threats. Vulnerabilities are known to cost millions or billions of dollars to the affected vendors and users (Telang and Wattal 2005). Numerous methods have been used in practice to detect and eliminate vulnerabilities from software. These include manual code auditing, automatic testing, and static analysis. Despite years of research on eliminating vulnerabilities however, the number of vulnerabilities is still on the rise (Anon.).

In this proposal, we take a different approach towards vulnerability mitigation. With several years of software development and maintenance, we believe there is enough data to guide prediction of vulnerabilities in a meaningful manner. Intuitively and at a high-level perspective, studying code that was found to be vulnerable in the past may help to predict currently unknown vulnerabilities. That is the code in a previously vulnerable site and that in another site with a similar but yet unknown vulnerability is similar in some sense. Code similarity for such purposes may be measured in the forms of the dependencies of the code on other components or pieces of code, or the system calls used, or other specific features such as checks of specific error numbers, and so on.

For this project, we will apply modern machine learning and data mining techniques on several security-focused features such as security advisories, alerts, and release notes, and also features from static analysis that we will develop. Specifically, for static analysis, we believe that control-flow and data-flow based analysis that reveals semantic-level artifacts of the code may be especially helpful. Furthermore, we will also use the version control history to obtain patterns relating different security advisories and alerts, results from static analysis, and possibly even coding styles. Previous work (Neuhaus et al. 2007; Neuhaus and Zimmermann 2009) has shown success at using component dependencies for predicting vulnerabilities. We will therefore also use dependencies amongst components as features in our system. While the security advisories and alerts and version control data are the meta-level information that hint at the pieces of code at which to focus, the code-level information obtained from dependency analysis and static analysis will provide us with a further understanding of code fragments that may be specially vulnerable.

Apart from the use of various features as described above, we will also use multiple machine learning and data mining techniques (including decision trees, neural networks, support vector machines, and association mining) and finally develop ensemble models based on

these individual techniques to derive the best results. We expect to obtain from our model actionable insights that will speed up vulnerability discovery by directing related effort on specific areas in the code.

2 Related Work

Attempts to quantify software reliability have been made since the 70s (Musa, Iannino, and Okumoto 1987). Many models have been developed that try to relate software faults to the age of the software, code complexity, programmer experience, organizational maturity, and so on (Takahashi and Kamayachi 1985; Engel and Last 2007). Security vulnerabilities are a special type of faults and may need special considerations for reliable estimation and prediction. It is more desirable to incorporate independent variables that have what are believed to be strong physical relationships to security vulnerability, and not variables that are more generic in nature.

Alzhami et al. (O. Alhazmi, Malaiya, and Ray 2005; O. H. Alhazmi, Malaiya, and Ray 2007) have developed regression models around the vulnerabilities of major software, such as different releases of Windows and Red Hat Linux. These models may be able to predict the rate at which vulnerabilities are discovered in general but cannot pin point the components that may be more vulnerable than others. Hence, these models provide little actionable information. As discussed earlier, previous research has empirically showed that dependencies amongst software components have good vulnerability-prediction capability (Neuhaus et al. 2007; Neuhaus and Zimmermann 2009). Because of previous success of dependency-based prediction, we will actively explore it further in this proposal as well.

3 Our Technique

As discuss briefly earlier, we rely on meta-information collected during the development and maintenance of the code and stored as security advisories and alerts databases, bug tracking systems, and version control systems to automatically identify code fragments that possessed vulnerabilities and how the vulnerabilities were patched. We next perform some static code analyses to identify code fragments that are semantically similar to already patched code fragments and yet themselves have not been patched. These code fragments are highly likely to possess vulnerabilities and may be checked further with more thorough static analysis, testing, or manual code auditing. In the rest of this section, our techniques are described in further detail. Figure 1 shows our proposed architecture.

3.1 Curating Meta-information

Our starting point will be all the security advisories and alerts collected from various public and internal databases. These may include CVE/NVD databases, and advisory databases from major vendors such as Cisco, Microsoft, Red Hat, and so on. Another important place to look for vulnerability data is the bug tracking or issue tracking database for the software.

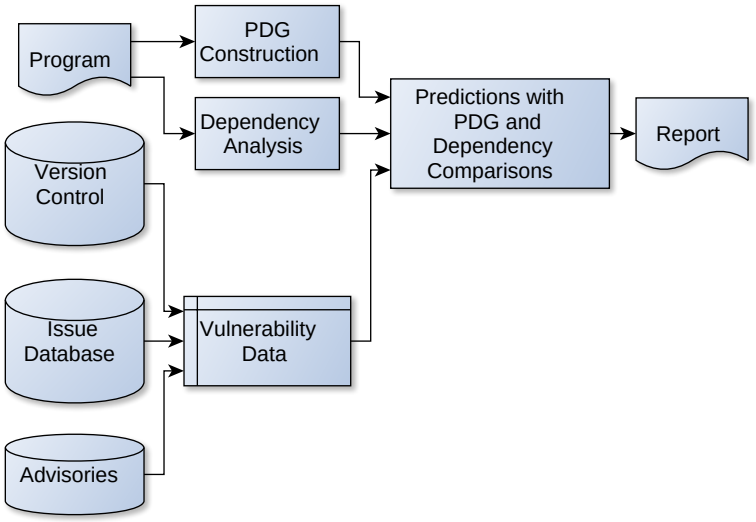

Figure 1: Overall architecture

Obviously, this information is most only if the specific bugs or issues related to vulnerabilities are labeled as such, i.e., they are marked as security-critical. Furthermore, we can also leverage information security-related internal information derived during software testing and auditing before release to enhance our information about possible vulnerabilities.

In the next step, we will parse information from version control systems. It is typical that developers leave bug IDs from the bug tracking database or CVE or other advisory IDs in the commit messages for the commits that patch the corresponding vulnerabilities. For open source software, CVE databases also sometimes provide the commit IDs from the commits that patch the corresponding vulnerabilities. Through these ways, we can link the vulnerabilities with the patches and thus pin-point the affected software components and the solutions that may be of possible use later.

3.2 Dependency Analysis

Software is usually designed in a modular manner and different components have inter-dependencies. Software components may be defined at multiple levels of granularity, including an entire program or library, files, classes (in languages like C++) and functions. Researchers have previously analyzed program-level and file-level dependencies and used them successfully to predict vulnerabilities (Neuhaus et al. 2007; Neuhaus and Zimmermann 2009). We will use similar approaches and also try finer granularities to derive predictions with low false positives. Although it is not clear why dependencies are a good predictor empirical results have shown them to be so. One may try to speculate why this is the case. Some components may be poorly maintained, and depending on them may introduce vulnerabilities. Moreover, some components may be difficult to use; such difficulty may increase the likeliness of improper use. Finally, dependencies, to some extent, define the problem a particular component is handling.

Some problems (such as authentication) may have greater security implications than others (such as a text editor).

3.3 Analyzing Program Dependence Graphs

We will also use program analysis techniques to determine semantic similarity between code fragments and thus identify possibly vulnerable code fragments. Our technique will be to compute program dependency graphs at the function level. Widely used in program analysis and compiler optimizations, a program dependence graph (PDG) is a directed graph with vertices representing the statements within the program and edges representing data flows and control flows.

In our technique, we will first generate the dataset of the PDGs for all the previously vulnerable and now-patched functions. Based on results from dependency analysis, we will select functions that may be similar to these previously vulnerable functions and then compare the PDGs of these functions with the PDGs of previously vulnerable functions. In our comparison, syntactic entities, such as variable names, etc., will not have nay affect. Furthermore, PDGs do not need to match completely. If there is a partial match, but the vulnerability is possibly present (which can be inferred by comparing with the PDG of the patched function), we will get a warning here. PDG computations are local (i.e., only atthe function-level) and hence not very expensive. Matching PDGs however may be expensive and as part of our research, we will use adapt existing graph comparison algorithms for our domain of research.

We are currently planning to support analysis of C and C++ code, using language parsing tools to generate symbolic representations of programs. Parsing C is relatively straightforward, and can be accomplished by a tool such as CIL. C++ is somewhat more difficult, due to its inherent semantics. Our currently planned approach is to hook into the LLVM parser using the Python library 'libclang', and then created graphs based on the parsed data structures. Given more time, a likely expansion of our work would include analysis of other languages, even those that are not in the C family.

4 Work Division

Chao Shi and Boyuan He will work on extracting meta information about security flaws and security advisories from various databases and repositories. Maciej Swiech and Josiah Matlack will work on creating language parsers and graph generators. A more clear division of labor is not possible to define at this time, due to the nascent nature of our current approach. More details on specific work distribution will be determined as our research progresses. The layout given here is meant as a very basic guide for the general approach to come.

5 Schedule

For the meta-information extraction part, We will be looking for some APIs to access the CVE/NVD databases first and if the API does not exist, we will write scripting language

for parsing the CVE/NVD website. The reason we start out on CVE/NVD is that we are most familiar with it and the database is well organized. As for other bug tracking databases, we can think of Metasploit at this stage. We will add more sources in the future. For the vendor-dependent advisaries, we will probably put that into lower priority.

Next we will focus on mining the version control system, we will first learn more about how svn is working and think of a way to parse the svn repository and then extract the related commit comments.

We expect that by the end of the quarter, we can have a library which will give a long list of known vulnerabilities and the comparison between the vulnerable code and the patched code. This would become a perfect training example set for the machine learning process.

6 References

Alhazmi, Omar H., Yashwant K. Malaiya, and Indrajit Ray. 2007. "Measuring, analyzing and predicting security vulnerabilities in software systems." *Computers & Security* 26: 219–228.

Alhazmi, Omar, Yashwant Malaiya, and Indrajit Ray. 2005. "Security vulnerabilities in software systems: A quantitative perspective." In *Data and Applications Security XIX*, 281–294. Springer.

Anon. "A Decade in Review, Transition on the Way." NSS Labs. https://www.nsslabs.com/blog/decade-review-transition-way.

Engel, Avner, and Mark Last. 2007. "Modeling software testing costs and risks using fuzzy logic paradigm." *Journal of Systems and Software* 80: 817–835.

Musa, John D., Anthony Iannino, and Kazuhira Okumoto. 1987. *Software reliability*. McGraw-Hill New York.

Neuhaus, Stephan, Thomas Zimmermann, Christian Holler, and Andreas Zeller. 2007. "Predicting vulnerable software components." In *Proceedings of the 14th ACM conference on Computer and communications security*, 529–540.

Neuhaus, Stephan, and Thomas Zimmermann. 2009. "The beauty and the beast: vulnerabilities in red hat's packages." In *Proceedings of the 2009 conference on USENIX Annual technical conference*, 30–30.

Takahashi, Muneo, and Yuji Kamayachi. 1985. "An empirical study of a model for program error prediction." In *Proceedings of the 8th international conference on Software engineering*, 330–336.

Telang, Rahul, and Sunil Wattal. 2005. "Impact of software vulnerability announcements on the market value of software vendors-An empirical investigation." *SSRN 677427*.