Extended Abstract: Predicting Security Vulnerabilities from Neglected Conditions

Introduction

The goal of this project is to review over the methods used to analyze how neglected conditions over time open up security vulnerabilities. One major example of neglected bugs include the Hot Potato Bug on Windows, which was first found in the early 2000s but neglected. Ten years later, it turned into a major security vulnerability in which any patch that aimed to fix it would break backwards compatibility on all versions of Windows.

In this project, I will be exploring current methodologies for bug prediction and security vulnerabilities. Current methods include analyzing software metrics and code history to characterize vulnerabilities and using classifiers to predict bugs. In this paper, I go over the related works, and go into detail on the works, and conclude with thoughts on the absence of work in overlapping the two fields.

Related Works

There has been work done on detecting neglected software conditions using a combination of rule-based techniques and data mining techniques [1]. Developers provide minimal constraints defining software violations they wish to find. Data mining techniques are used to discover rules. Programs are represented in graphs describing program dependence graphs. Using these techniques and representation, clues are mined to identify neglected conditions.

On the other hand of the spectrum, there have been machine learning methods to predict bugs [2]. One major problem with machine learning methods is the result of many false positives, which would alarm a typical developer and lead to distrust in the prediction model. [2] reduces the features by discarding features with the lowest gain ratio in order to identify the best subset of features that would lower the number of false positives.

In general, the [2] focus on software bugs and is oriented towards software engineering. [3] explores whether security vulnerabilities in Mozilla can be predicted using code complexity metrics. While complexity metrics can predict vulnerabilities at low false positive rates, it returns a high false negative rate. In a similar work, done by [4] on Windows Vista, the same method is used in order to predict security vulnerability. The metrics predict vulnerabilities with high precision and low recall values. In other works that use software engineering for prediction, components in a software are predicted as vulnerable based on the imports [8] and source code analysis is performed to study the evolution of detected vulnerabilities in hope of predicting new ones in the future [7].

Methodologies

[1] approaches neglected condition detection in a rule-based and data mining method. The goal is to find clues that would reveal neglected conditions. Programs are represented as program dependency graphs (PDGs) where the graph models data dependencies between statements and control dependencies. Two statements are data dependent if a variable defined from the

first statement going to the second statement is not redefined. Two statements are control dependent if the second statement is executed if the first statement is executed. By graphing the PDGs, rules can be mined, revealing neglected conditions. A rule is mined by examining popular patterns in the graphs, and then the rule is examined for violations. A conditional rule is said to contain a violation if it lacks control point, control dependence, data dependence with respect to some similar condition rule. A major drawback in this approach is speed; mining graphs similar to some n-edge graph is exponential.

[4] predicted security vulnerabilities on Windows Vista by using software engineering metrics and comparing the metrics against known vulnerabilities on Windows Vista. For each Windows Vista binary, the metrics were calculated and compared against the known vulnerabilities. Then, a classifier was trained to predict whether a given Vista binary had no vulnerabilities or one or more vulnerabilities with respect to some software engineering metric. The paper claims that alternative metrics should be explored to characterize vulnerabilities due to the high number of false positives in their results. [4] also built prediction models based on the number of dependencies in a binary. It has been shown with studies performed on Eclipse and Firefox that dependencies can predict vulnerabilities [5][6]. The dependency based SVM classified binaries perform better recall-wise than the classifiers based on software engineering metrics. [2] brings [4]'s work an extra step by studying a feature selection process for classifiers that predicts bugs.

Conclusion

Based on this literature review, we learn a few things. The methods in detecting vulnerabilities are quite rudimentary, resorting to arbitrary software feature collection and predictions via machine learning. The major drawback being that the predictions over one software are limited to that software only and not transferrable to other software. On the other hand, works in neglected conditions focus solely on identifying what is not there based on what is already there from other places in the code.

In short, the majority of works fall under one of two categories: 1) detect bugs or vulnerabilities or detected neglected condition. But there has been no work in an intersection over the two fields; or more specifically, does a neglected condition end up being a vulnerability?

Sources:

[1] Ray-Luang Change, Discovering Neglected Software Conditions by Mining Software Dependence Graphs, https://etd.ohiolink.edu/rws_etd/document/get/case1218722056/inline

[2] Shivaji et el., Reducing Features to Improve Bug Prediction, https://users.soe.ucsc.edu/~ejw/papers/shivaji-ase09-short.pdf

[3] Yonghee Shin and Laurie Williams, An Emprical Model to Predict Security Vulnerabilities using Code Complexity Metrics, http://www4.ncsu.edu/~yshin2/papers/esem08-shin.pdf

[4] T. Zimmermann, N. Nagappan, and L. Williams, Searching for a Needle in a Haystack: Predicting Security Vulnerabilities for Windows Vista, https://www.utdallas.edu/~kxh060100/cs6301fa15/zimmermann.pdf

[5] T. Zimmermann, N. Nagappan, "Predicting Defects with Program Dependencies", http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5316024&tag=1

[6] Adrian Schröter, Thomas Zimmermann, Andreas Zeller, "Predicting Component Failures at Design Time", http://dl.acm.org/citation.cfm?id=1159739

[7] Massimiliano Di Penta, Luigi Cerulo, Lerina Aversano, "The Evolution and Decay of Statically Detected Source Code Vulnerabilities", https://www.computer.org/csdl/proceedings/scam/2008/3353/00/3353a101.pdf

[8] Stephan Neuhaus, Thomas Zimmermann, Christian Holler, Andreas Zeller, "Predicting Vulnerable Software Components", http://dl.acm.org/citation.cfm?id=1315311