DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Impact factors for severity assessment of bugs discovered via compositional symbolic execution

Ricardo Nales

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Impact factors for severity assessment of bugs discovered via compositional symbolic execution

Einflussfaktoren zur Bewertung der Severity von Programmfehlern, welche mit Compositional Symbolic Execution entdeckt wurden

Author:	Ricardo Nales
Supervisor:	Saahil Ognawala
Advisor:	Profr. Dr. Alexander Pretschner
Submission Date:	Submission date

I confirm that this master's thesis in informatics is my own work and I have documented all sources and material used.

Munich, Submission date Ricardo Nales

Acknowledgments

To my supervisor, for always guiding me, and pointing me in the right direction. To my friends, for always helping, believing and encouraging me. And to my family, without them none of this would have been possible. My most heartfelt thanks to all of you.

Abstract

Software projects are big undertakements, usually developed by several teams. All of them will be supporting the same codebase, which will grop exponentially week by week, and surely gathering a good amount of bugs along the way. For this matter, testing has been widely adopted to find bugs and resolve them. But the question that arises when a bug is found more times than not is: how do we know which bug(s) we should prioritize? how can we measure the impact of a given bug? how much resources are required to solve this bug?

In this Master Thesis we set out to find if there was a correlation between the graph features of a program's callgraph, and its vulnerabilities found through a compositional symbolic execution, using MACKE. The graph features are to be generated by an internally developed tool, which in turn would allow us to automate the process of assessing the vulnerability.

We then follow to test our automatic assessment model with real professional feedback by using a GUI with which users can interact and see the results for themselves.

Contents

1 Motivation

In todays world, all software that has been developed by a team of programmers is bound to have errors, and the amount of them you might have in a project will only keep incrementing if they are not dealt with in a timely manner. Of all these errors that are found, they usually lead to vulnerabilities in our system, which can be exploted in several ways – ways which will be discussed at a later point in this thesis – and sometimes the time to fix one vulnerability completely outweights the benefit provided. That is why knowing the severity of a given vulnerability is usualy crucial to assess correctly the amount of resources we need to assign for its correction.

1.1 The severity of vulnerabilities

Since the inception of informatics, errors in code, be it semmantic or synactic, have existed. These errors, as mentioned previously, usually lead to security issues in our application, called software vulnerabilities.

Most errors arise from simple programming mistakes, and vary from one programing language to another, as well as the field where these programms are being put to use. Regardless of how a vulnerability comes to be, a solution needs to be found for it, which usually includes its development, testing, and a deployment.

But in projects where there are more vulnerabilities than resources to assign to them, or when fixing one vulnerability would yield a benefit that is simply not worth the investment? What if there was a way to include a numerical approximation that assesses its severity? We go into detail throughout this chapter, covering all of the reasons that lead us to undertake this project.

1.1.1 How are vulnerabilities found

A vulnerability can be found in a pleathora of ways: it can be a simple semicolon missing from the code, it could be a typo in one variable, these are now-a-days easy to find since there are IDEs (Integrated Development Environment); or it could be that someone overlooked sanitizing an input, that could lead to buffer-overflow errors during runtime. Whichever the source of a given vulnerability, the approach to find a it

is always the same: reproduce this error by supplying the inputs that got the software to an unstable state, or in most cases, crashed it.

1.1.2 The severity of a vulnerability

A vulnerability has the power to bring a whole system down. They can be a force to be reckoned with, when it has spawned inside a codebase that has built up to millions of lines of code. A fix for it could be trivial, if a developer understands the nature of the vulnerability, and/or has experience with the system, and/or he introduced the vulnerability himself.

But what happens in the normal occation when a vulnerability is found and there is no easy solution at hand? When resources are scarse and we need to prioritize which vulnerability to focus on next so that our system may continue working as inteded? When there are extremely huge time constraints and we can only budget to fix one at a time, because our workforce is limited?

1.1.3 Common Vulnerability Scoring System

The Research by the National Infrastructure Advisory Council (NIAC) in 2003/2004 led to the launch of CVSS version 1 in February 2005, with the goal of being "designed to provide open and universally standard severity ratings of software vulnerabilities". This initial draft had not been subject to peer review or review by other organizations. In April 2005, NIAC selected the Forum of Incident Response and Security Teams (FIRST) to become the custodian of CVSS for future development.

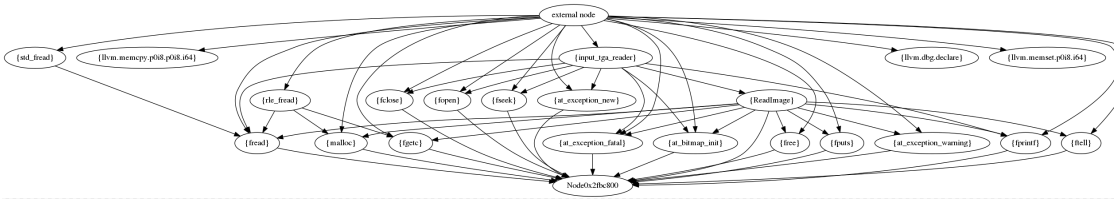
FIRST.Org, Inc. (FIRST), a US-based non-profit organization, whose mission is to help computer security incident response teams across the world[2] asked themselves these same questions back in 2005, when they decided to introduce CVSS, or Common Vulnerability Scoring System[2].

Throughout this thesis we introduce ways to find the severity of a specific vulnerability. Depending on several values, and using the aforementioned metric CVSS3 as our basis to rate them, we will give a vulnerability a numeric score, which could allow us to know what vulnerability should be given priority, and which one might be harmless.

1.2 Symbolic execution

Symbolic execution is a way of testing a program to determine what inputs allow the software to execute. Instead of using normal variables for testing, an interpreter follows the execution assuming symbolic values as the inputs instead of using concrete values – as per in normal execution – which can take any value for the given type of

Figure 1.1: Program callgraph



the input, thus covering the complete range of possible branches the program could take, therefore discovering vulnerabilities along its way.

1.2.1 Modular and Compositional Analysis with KLEE Engine

Throughout this thesis MACKE (Modular and Compositional Analysis with KLEE Engine)[3] will be used to find vulnerabilities, and callgraphs of programs to fully analize the source of the vulnerability, and the possible severity of it.

This tool makes use of symbolic execution techniques at the level of C functions, and then combines the results using static code information and inter-procedural path feasibility[3].

The use of this tool allows us to find an extensive amount of undocumented and documented bugs. The documented bugs found will be analyzed in the initial phases, since they have already been assessed manually with the CVSS3 framework.

1.2.2 Callgraphs

A callgraph[4] is the relationship between functions in a computer program, where several functions are present. It is, for the most part, user readable, and allows for an overall view of a program. It helps to pinpoint functions that might be unused, and if there are a few functions that are too intertwined. In general, a callgraph allows users to find if the intended purpose of the program has been met, and if the means to achieve the result was correct.

Call graphs can be dynamic or static. A dynamic call graph is a record of an execution of the program, for example as output by a profiler. Thus, a dynamic call graph can be exact, but only describes one run of the program. A static call graph is a call graph intended to represent every possible run of the program. The exact static call graph is an undecidable problem, so static call graph algorithms are generally overapproximations. That is, every call relationship that occurs is represented in the graph, and possibly also some call relationships that would never occur in actual runs of the program.

Call graphs can be defined to represent varying degrees of precision. A more precise call graph more precisely approximates the behavior of the real program, at the cost of taking longer to compute and more memory to store. The most precise call graph is fully context-sensitive, which means that for each procedure, the graph contains a separate node for each call stack that procedure can be activated with. A fully context-sensitive call graph is called a calling context tree. This can be computed dynamically easily, although it may take up a large amount of memory. Calling context trees are usually not computed statically, because it would take too long for a large program. The least precise call graph is context-insensitive, which means that there is only one node for each procedure.

1.2.3 Directed graph attributes

Knowing that we can extract the callgraph of a program, and that it is a set of vertices connected by edges, we know we will be dealing with a directed graph, and that its attributes can be mathematically analyzed.[1]

2 Related Work

Ask Saahil for guidance

3 Development

4 Results

List of Figures

Bibliography

[1] U. S. R. Bondy John Adrian; Murty. *Graph Theory with Applications*. 1976.

[2] I. (FIRST.Org. *Common Vulnerability Scoring System v3.0: Specification Document*.

[3] e. a. Ognawala S. *MACKE: Compositional analysis of low-level vulnerabilities with symbolic execution*. 2016.

[4] U. K. A. S. B. Sathe. *Data Flow Analysis: Theory and Practice*. 2009.