Automated Security Test Case Generation

Jeremy Gonzalo Arellano
Dept. of Computer Science and Engineering
University of Notre Dame
Notre Dame, IN, USA
jarella2@nd.edu

I. Introduction

In today's modern world, software is constantly being faced with the threat of security breaches. Often, these breaches can be severe, leaving data compromised and services possibly suspended in the process [1]. While we have developed ways to prevent such breaches from happening thanks to the many static and dynamic approaches to security analysis, there still lies the problem of diagnosing and exposing any security threats that may still be lying around after this analysis [4]. Therefor it is critical that some form of automated test case generation exist, where these generated tests will uncover potential vulnerabilities that often would be overlooked in normal analysis [4]. While current security testing methods are proficient at uncovering bugs/vulnerabilities, the amount of manual effort and time that these methods waste are not very efficient to large DevOps and agile environments [6]. Thus, the need for automated security test case generation is needed crucially in the world of technology today.

Software vulnerabilities stem from a plethora of sources, stemming from both low-level coding bugs that are generated from when the developer is creating the source code, to high-level design flaws that have to do with the architecture of a software piece [2][3]. While several tools and framework exist for identifying coding bugs and linting, detecting security bugs is another issue that requires different approaches and different scopes. In many cases, no security tool is perfect since there are many different vulnerabilities to consider and many more that emerge with evolving technology. Beyond just programming, a software's or system's architecture can also play a role in the security of itself, often discovering flows and potential security weaknesses based on design choices [5]. And beyond the architecture or a program/system, the evolution of a project exacerbates the challenge of maintaining a robust security posture throughout the development lifecycle [6].

This project aspires to bridge this gap by proposing an automated framework for security test case generation. Leveraging different algorithms, the framework aims to autonomously generate a suite of security test cases, prioritized on an integrated risk assessment system [4]. Thus, this approach not only augments the existing security testing process but also facilitates a more proactive stance towards software security, aiding in the early detection and remediation of security flaws [5].

Thus, throughout this research project, the aim is to answer the following questions:

1) How can we use evolutionary algorithms to automatically create security test cases?

2) How does ranking test cases based on risk affect the detection of seriuous vulnerabilities?

3) How does our autoamted framework compare to current security testing methods in effectiveness and adaptability? How does our method compare to manual testing

By analyzing these questions throughout the scope of this research project, the goal of creating and providing a new approach to automated solution for security test case generation. This project aims to contribute to the wider range of the software development ecosystem, providing a solution that will enhance the capability to uncover and resolve software vulnerabilities in everyday development.

II. Related Work

While the core of this project serves to improve the landscape of security tools and techniques, several efforts have already been made in the realm of automated test case generation. In the realm of automated security test cases, various approaches have been taken at mitigating the risks of security breaches. Some have been more focused on the architectural perspective of a software, while others have taken the approach of using established but continuously evolving algorithms for generating security test cases.

Marksteiner, Ramler, and Sochor (2019) approach the problem in a unique way through integrating threat models within their automated test case generation. While the study's main focus was surrounding the use of these test cases with IIoT applications, their methodology emphasizes the transition from threat identification to test case development [1]. From this transition of identification to test case development, the takeaway can be used to form an improved tool that ambitiously expands away from IIoT application into other developer applications such as web apps. Thus, this integration is important for ensuring comprehensive coverage of potential vulnerabilities within the early development of an application, and important to serve as a foundation when developing an automated framework for security tests [1].

Building upon architectural flaws, further research has determined a shift within security on software's security design flaws. The IEEE center for security Design's report (2014) especially speaks on this shift through examining the move from bug detection to identifying security design flaws in software applications [2]. The study analyzes and points to the conclusion that security analyst in general must move away from the practice of focusing on an applications bug, but rather the design flaws of an application in general. Thus, this perspective given by the report is critical in understanding the role that architecture plays in software security. Through their analysis of the top 10 security design flaws, their research and findings provide an important guideline to building an analysis algorithm that can target these design issues rather than just code-level bugs [2].

Other works such as McGraw's research provide a framework that build upon existing software security practices, establishing a set of best practices that should be practiced in the field [3]. Throughout the research "Software Security: Building Security In" (2006), McGraw emphasizes the necessity of considering security at every level of an applications development, not just at the end of its development [3]. By ensuring that security is not just looked at as a last stage item in a software's development, the focus is given to these practices to ensure that security is a fundamental aspect of software development. Thus, the research done will hopefully give a new light to different test case approaches by integrating security considerations in different stages of software development.

III. Methodology

In order to discover the answers to our proposed questions, a framework first needs to be established for creating the automated security test case generation. By creating a system that utilizes evolutionary algorithms and by using architectural analysis in this system, the goal is to tackle an answer to our first proposed question, following suite with its performance by answering the following questions. Through this process of building and correcting our framework, the objective is to pave a path to test case generation that will be trivial on building on previous research.

A. Framework Development

Our aim to developing our test case generation is to split the process to create a thorough framework. To integrate both architectural analysis and evolutionary algorithms into these test cases, we need a basis to work on and a starting point to see how these approaches can work together to produce anticipated results.

1) Data Collection: The first step would be to gather any data nessesary to evaluate our test cases on through looking at different database sources and software repositories. Databases such as vulnerability databases would provide useful in this search, as we can compare some of the most common secuirty flaws within this database when evaluating our test suite. Utilizing the CWE catalog as a primary source of software weaknesses, we can further build a more detailed approach to different secuirty breaches, and document how these secuirty breches could play a role in the developemnt of our framework. A possible parsing mechanism would be developed to extract and retreive relevant data from the list, extracting elements from the "Weakness" catigory of a CWE. By parsing these different XML files and excluding non-relevant information, we can further build a foundation for identifying vulnerabilities. And with different views on security breaches, the goal of studying these databases would hopefully bring a different perspective and testing points for our test cases.

Software Repositories would be another credible source of data when building our framework. Not only would this give real-world examples to develop and test our test-cases on, but it would also allow us to confirm common secuirty patterns and flaws that are found in this process. By gathering a diverse range of real-world software projects, we can test the effectivness of our framework and make adjustments as needed. While the selection of a repository might be complex, the choice will mostly be selected upon whether the repository can be tested and whether the codebase offers some software vulnerabilities.

2) Evolutionary Algoriothm Implementation: Once our database has been selected and has been solidly tested, our algorithmic approach can be developed. By drawing inspiration from genetic algorithms (GAs) and from parallel evolutionary methods, the hope is to form algorithms that prove effiicent when deploying test cases [4]. Thus, our objective when developing these secuirty algorithms is to optimize our defined test covereage criteria (such as branch or path coverage), and evolve our intial test cases into effective ones.

While this process of developing these algorithms plays out, some form of parallelization should be used to improve the speed and efficiency of our test generation process. Since parallelization offers some form speed improvements and effectiveness improvements when generating tests, it is important for us to develop this when creating/testing our algorithm [8]. Thus, our design should implement our evolutionary algorithm that will work on generating test cases concurrently, allowing for shorter generation times and more data for evaluation when testing the effectiveness and ability of our algorithm in the testing phase.

3) Architectural Analysis: As mentioned before, the generation and analysis of the architecure of a software is important for us to analyze in the presepective of our first research question. By conducting analysis on different architecural documents from software projects, we can isolate the important steps when looking at a code-base and recognizing design flaws/vulnerabilites. Through exploring model-based testing approaches and using different architecrtral analysis tools, the aim here is to identify and document any architectural vulnerabilities that might have been found in this analyis [9]. And through looking at these common patters, integrating them into our algorithm approach.

To integrate these test cases with our architectural analysis, the primary objective is to find common occurances from our data sources and integrate them into our algorithm findings. By mapping differen architecrual flaws into our secuirty evaluatios, we can map them to certain test case senarios and hopefully cover more vulnerabilites by considering this. Thus, with architectural analysis, the aim is to ensure that our test-suite covers both code-level bugs and architectural flaws along the way.

B. Evaluation Framework

The next step once we have established a framework for test-case generation is to develop a framework for testing the effectiveness of our algorithm. By rigorously testing the effectiveness and efficiency of our automated security test case framework, the algorithm will have some metrics that can be used to improve the structure and analysis.

1) Test Case Applications: By pinning our algorithm and test-case generation on real world software bases, we can test the performace metrics of our algorithm. Generating these test cases will also show whether the performace changes on different practical enviornments, and how widly the algorithm can be used. Pulling different open-source projects from GitHub will give a base to this research goal, allowing for the idenificiantion of flaws in the algorithm when looking at already idenitfied security flaws in the GitHub repository.

2) Risk Assesment: In answering our second proposed queestion, we must develop some type of model that will assess the risk accosiated with certain secuirty vulnerabilities into our framework [10]. This model will serve the puprose of categorizing and prioritizing test cases based on the sevarity of a secuirty vulnerability, integrating a type of priority queue when reporting these issues back as metrics. Through this, a prioritization will be given when executing the test cases, addressing the most critical vulnerabilities first and efficiently using resources on the faults that will most likely take the longest to assest and fix.

3) Performance Metrics: As mentioned before, we need a way to test our algorithm and approach to see if it any good when comparing it to previous tools. Not only would this provide the answer to our third research question, but it will serve to guide our algorithm to better metrics once flaws are identified and fixed. For one, coverage must be essentially covered for all code bases tested with our testing-suite. This would inlcude paths such as branch coverage, paths, and other things that are normally called in the program being analyzed. Once identified, these will be pinned against what is already known from that database, giving way for a percentage to see how much of known vulnerabilites our algorithm covered. Efficiency would also be another metric worth measuring here, as it would give us the time and resources that our algorithm uses in its run time. The focus here would be to optimize our performace to be less time consumming and less resource intensive as previously mentioned tools. Thus a timer and possibly resource monitor would serve useful to integrate into our program to evaluate its efficiency metrics. False Positive/Negative reporting would also serve vital to measure thoughout the process of this project, since it would be important to see if our algorithm is perfoming correctly. By keeping track of the rate of false positive and false negatives, we can create a metric when pinned against found vulnerabilites, making it easier to asset the effectivness of the testing suite. Thus, from these metrics, tweaks and refinements can be made to the algorithm and be reported later when discussing its performance when compared to other tools.

IV. Early Results

From what has been tackled and what is left on the project, there has been some advancements made in achieving an automated security test case generation.

- Data Collection: A couple of preliminary sets have been gathered as a bases of our framework. These include some CWEs that are common and would be easy to test in the future when pinning our algorithm into it.
    - EduLearn – Offers moderate to high complexity in its codebase with opportunities to test vulnerabilities like data, API, and user privacy vulnerabilities.
- Basic shell of our program has been developed; however, it is not defined yet since this is still being developed. As of now, the shell focuses just on generating simple test cases and running it on simple applications.
- Similarly, a framework and basic shell has been developed when assessing the risk of a security vulnerability.
- Explored a couple of architectural tools used to analyze code bases for vulnerabilities.

V. Next Steps

To finish out this project, there are a couple of things that need to be addressed before our testing suite is complete:

- Create a generalized extraction process from getting CWE and reporting their security vulnerabilities.
- Develop and perform architectural analysis on selected software projects.
- Refine our algorithm from the shell created to be more complex for other test cases. (on this note, parallelization also needs to be implemented onto this part)
- Integrate our risk assessment model into our algorithm.
- Run a couple of datapoints through our model to develop some metrics of this algorithms performance.

Rererences

[1]S.Markstiener, R.Ramler. Integrating Threat Modeling and Automated Test Case Generation into Industrialized Software Security Testing, 2019

[2] T. I. C. S. C. for Secure Design. Avoiding the top 10 security design flaws, 2014.

[3] G. McGraw. Software security: building security in, volume 1. Addison-Wesley Professional, 2006

[4] Fraser, G., & Arcuri, A. (2014). A large-scale evaluation of automated unit test generation using EvoSuite.

[5] B.Chess, & G.McGraw. Static analysis for security, (2004)

[6] B. Boehm & R. Turner. Management Challenges to Implement Agile Processes in Traditional Development Organization, 2005

[7] Zalewski, M. (2011). American fuzzy lop
[8] Weiwei Wang, Shumei Wu, Zheng Li, Ruilian Zhao, Parallel evolutionary test case generation for web applications, Information and Software Technology, Volume 155, 2023
[9] Mohd-Shafie, M.L., Kadir, W.M.N.W., Lichter, H. et al. Model-based test case generation and prioritization: a systematic literature review. Softw Syst Model 21, 717–753 (2022). https://doi.org/10.1007/s10270-021-00924-8
[10] Felderer, Michael & Haisjackl, Christian & Pekar, Viktor & Breu, Ruth. (2014). A Risk Assessment Framework for Software Testing. 10.1007/978-3-662-45231-8_21.