Automated software vulnerability discovery, exploitation, and patching

Henry Post, hp2376@nyu.edu, NYU-CS-GY-6813

Abstract

In this paper, I propose a model for integrating various techniques for discovering, testing, fuzzing, exploiting, and fixing software vulnerabilities. These capabilities can be combined into a unified model that leverages existing research on these topics.

In fact, many aforementioned techniques are already used in the private sector by paid and free SCA (software composition analysis), SAST (static appsec testing), DAST (dynamic appsec testing), IAST (integrated appsec testing), fuzzing, exploit mutation, and source-sink taint analysis tools.

Extended Abstract

The current nature of most open-source projects exist as a machine-readable, machine-parseable format. Builds are highly standardized and adhere to a common directory, syntax, and language format. Most open-source software is written in only a handful of languages: C, C++, C#, Python, Java, Go, Ruby, PHP.

Github, as a highly discoverable, open, and transparent code hosting solution, has enabled millions of developers to contribute application source code. Larger projects such as Apache Tomcat (webserver), Visual Studio Code (code editor), TensorFlow (ML library), React Native (JS Server-side framework), etc.

Because of the nature of open-source, any software bugs created are also immediately visible to the entire world. These bugs may go unnoticed for years, later to be discovered by a security researcher, black hat hacker, end user, or anyone else who uses the software.

With the advent of highly discover-able and machine-readable open source projects, as well as automated vulnerability scanning, I believe that exists an emerging field of automated vulnerability scanning of open-source, which could lead into automated exploitation of undiscovered open-source vulnerabilities.

We are already seeing software component (or BOM, Bill of Materials) scanning, such as Snyk (http://snyk.io/) and White Source (https://www.whitesourcesoftware.com/) show up. These detect components with already-discovered vulnerabilities (i.e. known CVEs), but not new vulnerabilities.

Additionally, software such as Fortify, WebInspect, SonarQube, are offered commercially or for free, to perform software composition analysis (SCA), Static AppSec Testing (SAST), or Dynamic AppSec Testing (DAST). These tools can allow you to discover new vulnerabilities in software without manual testing.

If one or more tools were to be fully integrated with a framework that allowed for code scanning, exploit generation, exploit mutation, exploit confirmation, and exploit execution, it would make

exploiting security vulnerabilities in open-source projects trivial compared to traditional human-powered methods.

To counter the risk of a wide-scale automated software exploitation tool from being used on many open-source projects, perhaps to great effect, a method must be devised for automatically detecting and patching open-source software vulnerabilities, before threat actors are able to auto-detect and auto-exploit software vulnerabilities.

In this paper I plan to survey the current publicly available research in this paper's domain (Automated software vulnerability discovery, exploitation, and patching), and perform analyses of OSS using freely available automated tools to see how easy it is to automatically discover, exploit, and perhaps even patch software vulnerabilities.

Hypothesis

In this paper, I propose a novel model of discovering, enumerating, exploiting, and patching software vulnerabilities, using taint analysis, abstract syntax tree parsing and exploit detection/confirmation using dynamic application security testing, and fuzzing with genetic algorithms.

These methods of analyzing, exploiting, and patching vulnerabilities alone are not new. However, incorporating all of these techniques into a single framework is.

I wish to also propose extensions to this model to react to external sources of information. To the current open-source ecosystem, publicly available vulnerability disclosure ecosystem, such as the NIST NVD, MITRE CVE, as well as harvesting proof-of-concept exploits from various databases, or event OSINT sources.

There already exists a large number of papers covering how to parse application source code into tree structures [8] and analyze the “taint” introduced by handling user input through an application’s data flow. Below is a simplified example (fig. 2) of how the flow of an SQL Injection vulnerability could be detected by a method of Taint Analysis. The code can be seen in (fig. 1).

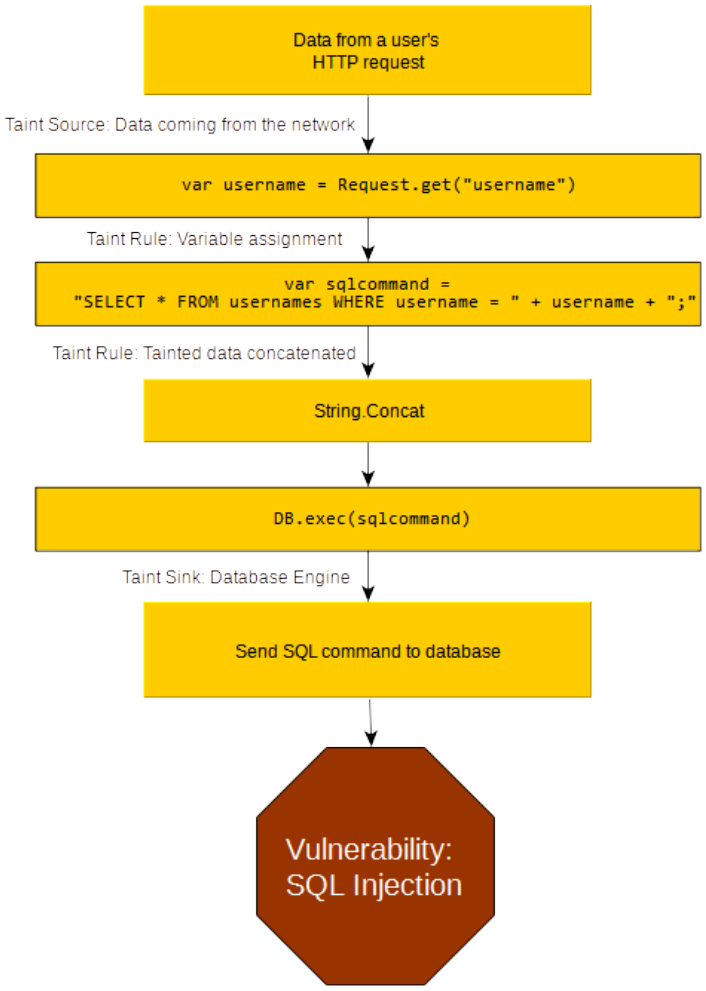


Figure 1: A simplified example of Taint Analysis on the flow of data through a web application.

```
var username = Request.get("username");

var sqlcommand = "SELECT * FROM
usernames WHERE username = " + username
+ ";"

DB.exec(sqlcommand)
```

Figure 2: Source code vulnerable to SQL Injection.

Figure 3: A tree of the parsed source code, generated from https://astexplorer.net/

In (fig 3.), you can see the “+” operator shown to contain both the string literal “SELECT * FROM …” as well as the second binary expression (username + “;”). This tree structure allows the movement of the data from “Request.get()” to be tracked all the way to “DB.exec()” by making rules that track the flow of data through variables, assignment, transformations, and function calls.

There are a number of useful properties that parsing source code into an abstract syntax tree has. It allows you to extrapolate vulnerabilities by taking advantage of the properties of graphs as a data structure [4],[6],[7].

If we combine this static graph analysis approach with a dynamic application scanning approach, such as sending HTTP requests through some sort of fuzzing engine (such as SQLMap), we can marry the code parsing benefits of static tree analysis with the exploit-confirmation benefits of dynamic application security testing.

Once the code has been transformed into a tree structure, taint can be tracked, and vulnerabilities can also be extrapolated by analyzing patterns of code organization [8] and mutating them in order to generate new code graphs that “look similar to” known-exploitable sub-trees that exist within our code.

Once we have an analysis engine that can analyze the code as a graph, and its properties and trends, we can add another module to automate the **patching** of security vulnerabilities by leveraging “taint cleanse” rules (fig. 4). To build on the SQL Injection example, we would be able to automatically generate a code fix by suggesting we modify the code graph to include a “taint cleanse” node of “Parameterize Query” that would prevent the tainted data from reaching the database in an unsafe way.

We would be able to generate fixes automatically because of the nature of our analysis engine – Because the code is already in a tree format, we can fix vulnerabilities by performing operations on the graph itself (node swaps, node replacements, node deletion), and transforming the tree back into application source code.

We can also leverage databases such as NIST NVD and MITRE CVE to create new sources of taint within our application, or even across multiple applications, because we will be able to associate vulnerable components with specific CVEs.

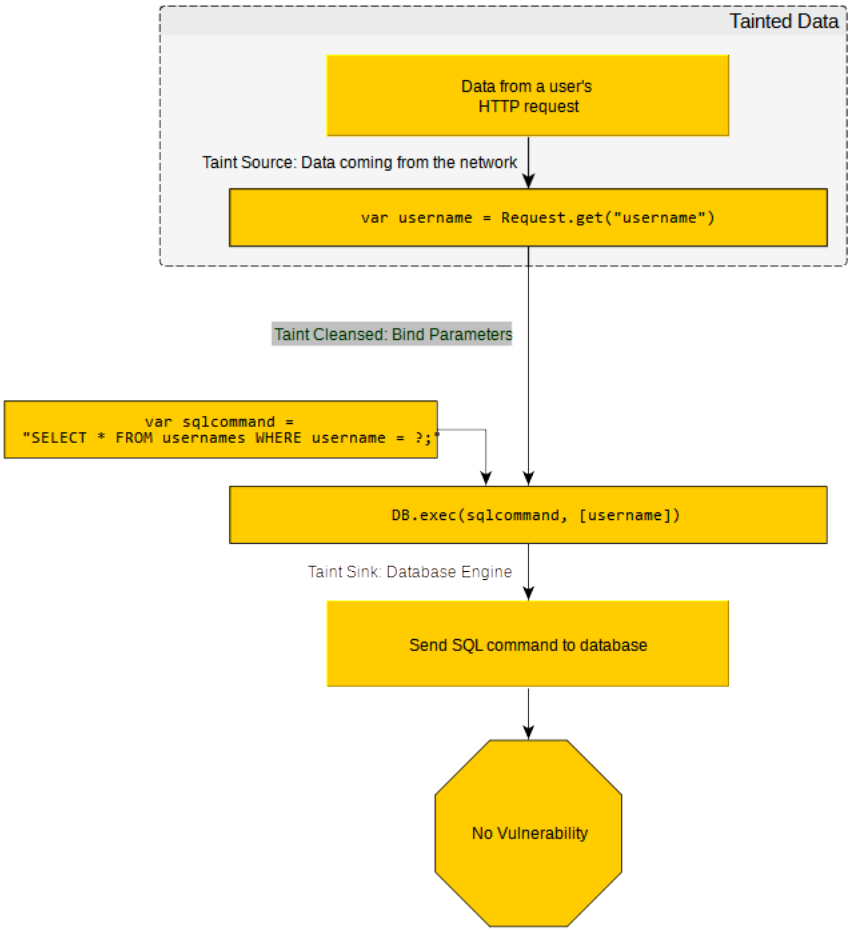


Figure 4: An example of a "taint cleanse" rule, Bind Parameters, preventing tainted data from reaching a Taint Sink, the database.

This model of parsing source code, analyzing vulnerabilities, extrapolating vulnerabilities, and confirming vulnerabilities through dynamic testing is also extendable through machine learning [3] and by using genetic algorithms [8],[7],[6] to mutate properties. By taking these techniques and applying them to our proposed model of exploit detection and confirmation, we can create a model that should be able to detect a large number of obvious vulnerabilities, but also more difficult-to-detect vulnerabilities due to the large dataset of not only general-purpose machine learning datasets, but code as data, and therefore code trees as data.

Metric

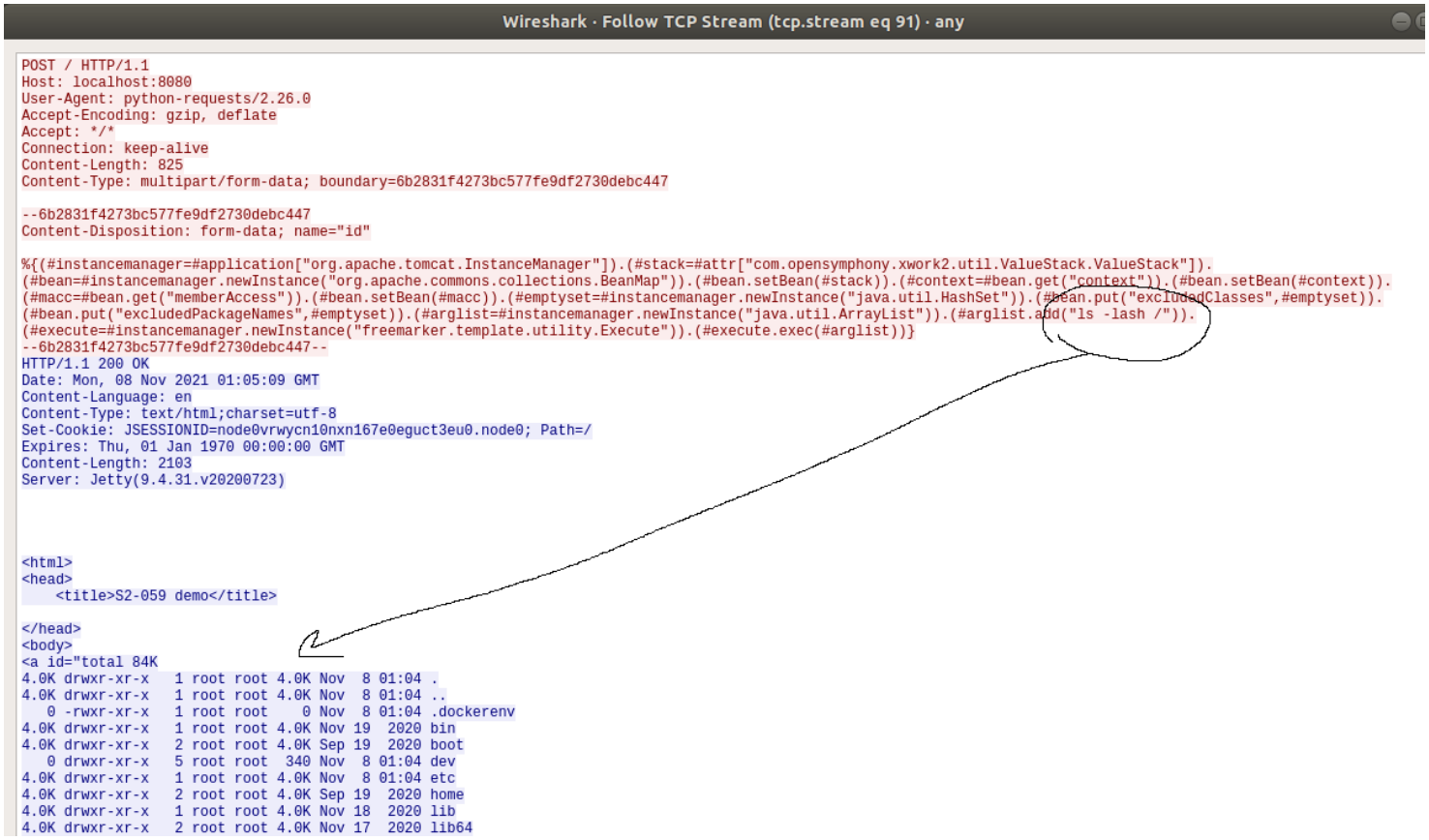
I do not have a single metric proposed to evaluate evidence, but I will examine a handful of techniques that are relevant to discovering risks that can be mitigated in software systems. This can be accomplished by adopting some or all of the methods of securing software that are described in this paper.

After writing this, I realize that this paper is slowly starting to sound like a very long advertisement for different DevOps/DevSecOps practices that many organizations are adopting. I am choosing to explicitly acknowledge this here to give context as to why this paper is

Many organizations with IT infrastructure, especially those that develop, maintain, patch, and release software, must manage risk generated as a result of having a software development lifecycle. Normal organizational processes such as patch management, using third-party components, and developing new software carry with them risks that could be known or unknown, and could be mitigated in some capacity, or not mitigated at all. This all depends on the procedures each organization puts in place to discover risk and mitigate it.

Lack of risk visibility on vulnerable third party components

The first source of risk I will discuss is the risk generated by a lack of visibility of vulnerable third party components. Take Apache Struts 2 (and 1), for example. This component is vulnerable to a quite nasty Remote Code Execution bug, CVE-2020-17530, that occurs when a user sends an HTTP request that exploits an overly-permissive call to a graph navigation language called “Object Graph Navigation Language”. Below is a wireshark packet capture of a proof-of-concept demo of this CVE – red is a HTTP request, and blue is an HTTP response. You can see something odd here – Linux commands executed over the web, from HTTP headers.



A very similar Struts 2 vulnerability is what enabled the Equifax Data Breach in 2017 [9], in which about **145 million** social security numbers, names, and birthdays were leaked, among other data. For reference, there are currently **332 million** US Citizens. Note that Equifax's situation is slightly different than our example, because they knew about the vulnerability, but chose not to patch the vulnerability.

Imagine that you are a medium sized (~5000 employee) organization that develops Java web applications, among other software. You have recently begun an effort to modernize your AppSec program, but lack metrics on what specific components your software developers are incorporating within deployed software.

You have no idea what kind of risk exists within your organization with regards to third-party vulnerabilities, like the nasty Struts 2 and Struts 1 CVEs. This lack of visibility is worse than just not mitigating the risks – You could be exposing very critical IT systems, entire business lines, and PII/PCI data because of unknown third party vulnerabilities. In this scenario, you would have no idea of the scope or potential impact of these third party vulnerabilities. You would not know when, if, or how these vulnerabilities could be exploited.

To solve this "lack of visibility" problem, you need to use a system that tracks third party component vulnerabilities and quantifies the risk that these components bring. This is what dependency scanning would provide. A full "bill of materials" that includes CVE (Common Vulnerability Enumeration) data, providing you the visibility to manage that risk.

Lack of risk visibility on software defects

Similar to not knowing risk exposure due to vulnerable third party components, not knowing risk exposure due to software defects can also lead to very negative business outcomes.

If your company, on average, per project, changes 1000 new lines of code every week, there is a high chance that some of those code changes will introduce bugs.

Just like with our third party vulnerability issue, if an organization does not know what codebases contain code defects, they cannot start managing the risk because they simply do not know where it is and what the impact of the risk is.

Much like the example involving 3rd party components, the solution to this is similar.

1. Perform manual code reviews, and/or
2. Use automated tools to track software defects.

This is what automated code scanning provides. It fixes the “lack of visibility” problem for in-house software.

Lack of effective mitigation strategy for vulnerable third party components or software defects

Once an organization has a sufficient monitoring and reporting, they must start devising a system to mitigate, fix, or otherwise handle the risk that **any vulnerability** would create.

This is up to the specific organization, and I will not be making recommendations or discussing specifics in this section, other than “some mitigation strategy must exist for all detected vulnerabilities”.

References

- [1] Z. Wang, Y. Zhang, Z. Tian, Q. Ruan, T. Liu, H. Wang, Z. Liu, J. Lin, B. Fang, and W. Shi, “Automated Vulnerability Discovery and Exploitation in the Internet of Things,” Sensors, vol. 19, no. 15, p. 3362, Jul. 2019 [Online]. Available: http://dx.doi.org/10.3390/s19153362
- [2] “Vulnerability scanning of IOT devices in Jordan using SHODAN,” IEEE Xplore. [Online]. Available: https://ieeexplore.ieee.org/document/8277814.
- [3] “The Coming Era of alphahacking?: A survey of automatic software vulnerability detection, exploitation and patching techniques,” *IEEE Xplore*. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/8411838.
- [4] “A machine learning-based approach for automated vulnerability remediation analysis,” *IEEE Xplore*. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/9162309.
- [5] “Robot hacking games,” *Center for Security and Emerging Technology*, 05-Oct-2021. [Online]. Available: https://cset.georgetown.edu/publication/robot-hacking-games/.
- [6] “Automated vulnerability analysis: Leveraging control flow for evolutionary input crafting,” IEEE Xplore. [Online]. Available: https://ieeexplore.ieee.org/document/4413013.
- [7] *Kameleonfuzz: Evolutionary fuzzing for black-box XSS detection*. (n.d.). Retrieved October 24, 2021, from https://www.researchgate.net/publication/259175145_KameleonFuzz_Evolutionary_Fuzzing_for_Black-Box_XSS_Detection.
- [8] F. Yamaguchi, A. Maier, H. Gascon and K. Rieck, "Automatic Inference of Search Patterns for Taint-Style Vulnerabilities," 2015 IEEE Symposium on Security and Privacy, 2015, pp. 797-812, doi: 10.1109/SP.2015.54.
- [9] Fred Bals, “Equifax, Apache Struts, and CVE-2017-5638 vulnerability” [Online]. Available: https://www.synopsys.com/blogs/software-security/equifax-apache-struts-vulnerability-cve-2017-5638/

Extra Resources

Tools

For a tools table, see “Henry Post - NYU-CS-GY-6813 - Automated software vulnerability discovery, exploitation, and patching - Tools List.fods”.

Vulnerable Software List

For a list of vulnerable software, see “Henry Post - NYU-CS-GY-6813 - Automated software vulnerability discovery, exploitation, and patching - Vulnerable Software List.fods”

Links

https://github.com/HenryFBP/NYU-CS-GY-6813-research-paper-work

https://github.com/HenryFBP/NYU-CS-GY-6813/tree/master/research-paper/papers/final

https://pentester.land/cheatsheets/2018/10/12/list-of-Intentionally-vulnerable-android-apps.html

https://owasp.org/www-project-vulnerable-web-applications-directory/

https://forums.hide01.ir/topic/vvmlist-a-well-curated-list-of-intentionally-vulnerable-machines/~51/