A review on automated software vulnerability discovery, exploitation, and patching

Sharath Katta Sridhar, *sk8671@nyu.edu*, *NYU-CS-GY-6813*

Introduction

The threat landscape for the domain of cybersecurity has diversified and changed drastically the past 5 years.

There has been an abundance of automation of cybersecurity-related objects and actions, such as creating, provisioning, and modifying cyber infrastructure, with tools like Terraform, Docker, and Kubernetes; for speeding up the development of applications, such as application testing; and new programming frameworks, such as Angular, React, and Go. There have also been new tools and techniques used to attack targets by hackers, like Metasploit, Radare, or afl++, to name a few.

It is now easier and faster for information systems to be created, modified, and defended, but also easier for attackers to probe, evaluate, and attack said systems.

I believe there is a growing need to create a unified model for automated software vulnerability discovery, exploitation, and patching.

In other words, there should then be an open-source implementation of said unified model (a great example being "Mayhem", that won DARPA's Cyber Grand Challenge in 2016) that can be widely used, to secure information systems against generic automated threats.

Abstract

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

In fact, many techniques mentioned in this paper 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.

These more advanced techniques surrounding data-mining existing codebases for software vulnerabilities are becoming necessary now. Here are some reasons for the necessity of applying more advanced defensive techniques within industries that use, create, and modify information systems:

1. High volume of rapid code changes
2. Automated, non-human threat actors
3. Constantly changing threat landscape due to various factors

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, a lot of 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.

In fact, the Cyber Grand Challenge, held by DARPA in 2016, achieved this. Teams were pitted against eachother to secure various software endpoints against sophisticated and nuanced attacks. This resulted in a series of Cyber Reasoning Systems interacting with eachother on enterprise hardware, generating useful data and conclusions about the future of Cyber Reasoning Systems and Cybersecurity automation as a whole.

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 explore various pieces of research focused on 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, they are not often combined into one tool or framework. I want to explore the idea about combining these individual flaw-finding and flaw-fixing techniques into one model.

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. 1) 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. 2).

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

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

DB.exec(sqlcommand)
```

Figure 1: Source code vulnerable to SQL Injection.

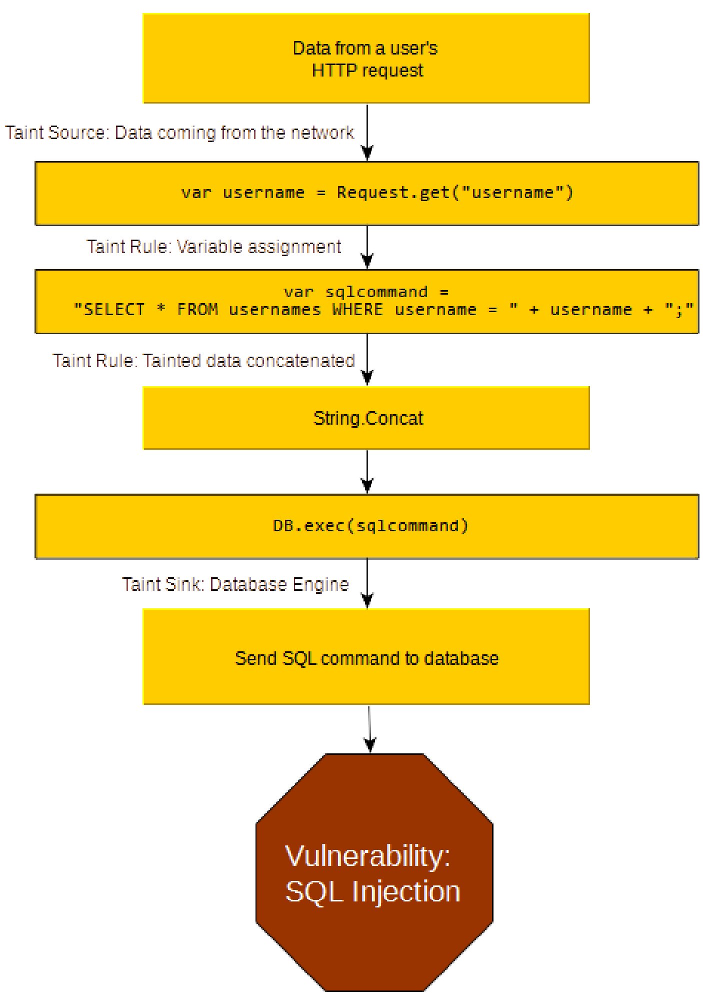


Figure 2: A simplified example of Taint Analysis on the flow of data through a web application.

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.

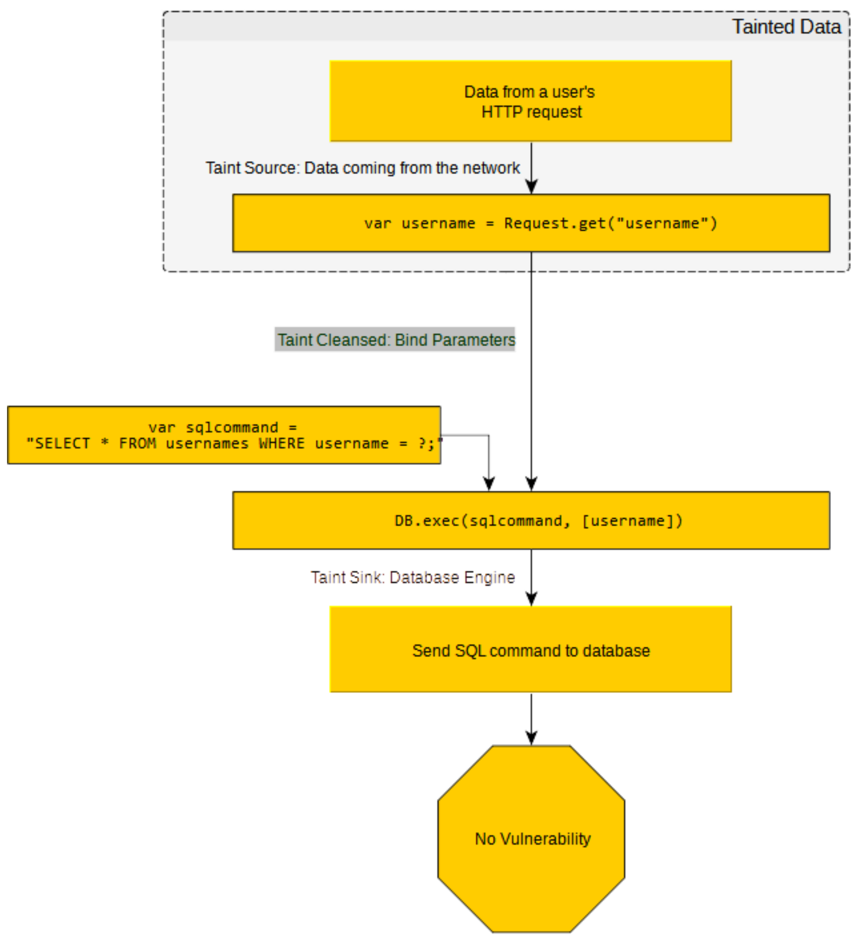


Figure 4: An example of a "taint cleanse" rule, Bind Parameters, preventing tainted data from reaching a Taint Sink, the database.

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

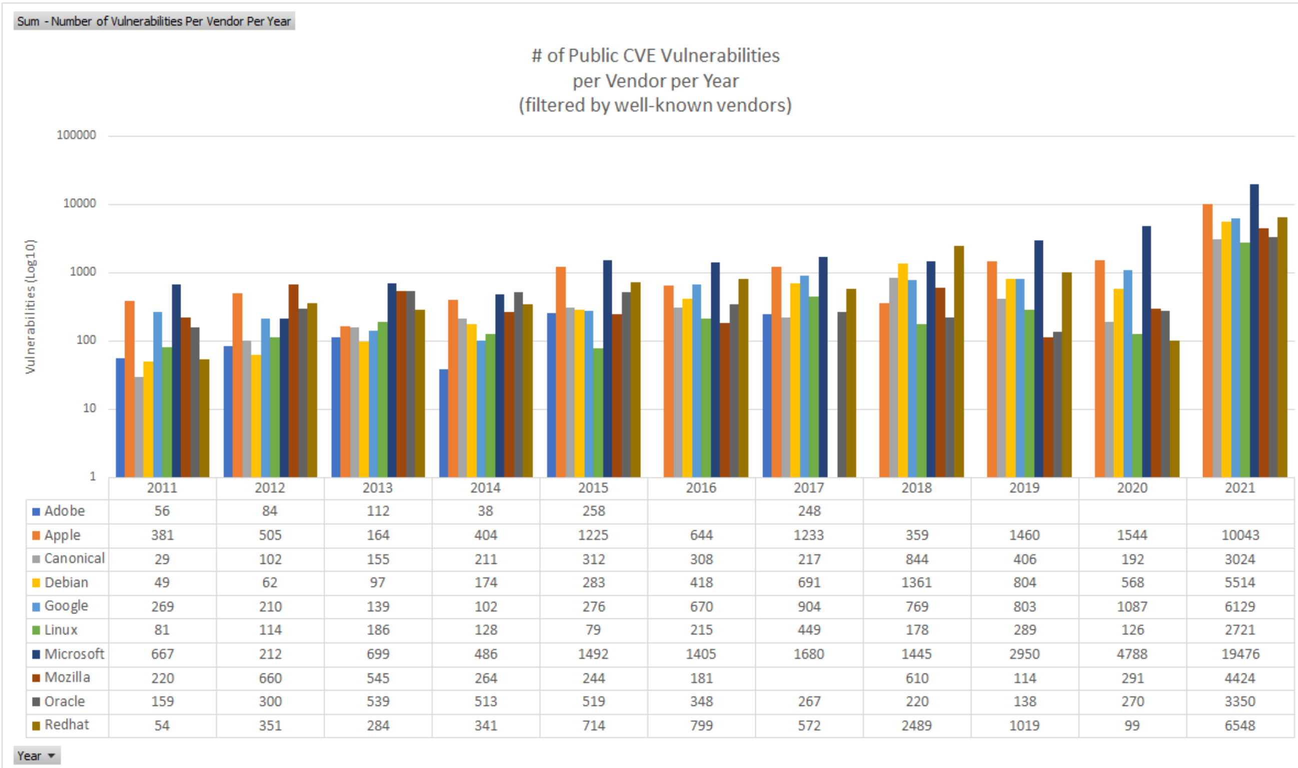
to associate vulnerable components with specific CVEs.
This model of parsing source code, analyzing vulnerabilities, extrapolating vulnerabilities, and confirming vulnerabilities through dynamic testing is also extendable through machine learning [3] and by using genetic algorithms [8],[7],[6] to mutate 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.

Evidence

See this file: Microsoft Excel Worksheet for raw data, or [this link].

This data comes from [http://cvedetails.com/], a site that collates data from NIST’s NVD XML Web API interface. Ideally, I could scrape this data directly from NIST NVD website, if I had more time to do so. I scraped it and aggregated it into an excel sheet, and filtered it using Pivot Tables to summarize data. I found some interesting things.

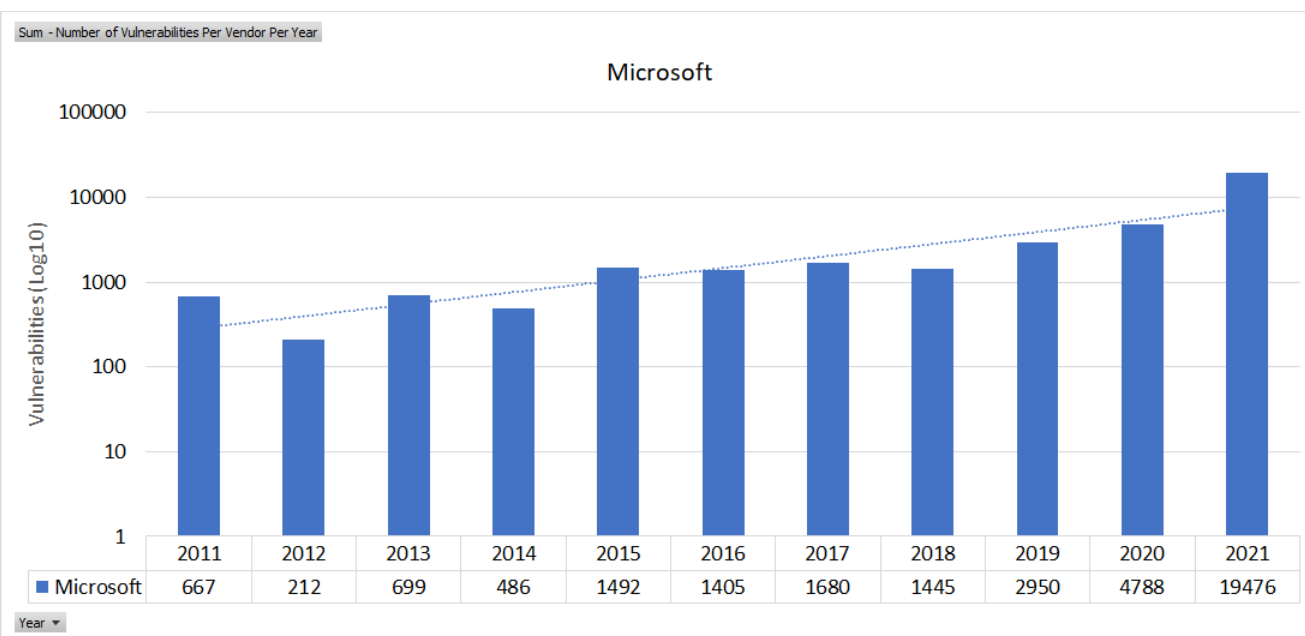
I chose to look at the Top 50 Vulnerable Applications, year-over-year, from 2011 to 2021. I made some simple metrics that suggest that the presence of vulnerabilities within vendors may be growing exponentially, perhaps due to more reporting, more scrutiny, and/or more development.



	2011	2012	2013	2014	2015	2016	2017	2018	2019	2020	2021
Adobe	56	84	112	38	258		248				
Apple	381	505	164	404	1225	644	1233	359	1460	1544	10043
Canonical	29	102	155	211	312	308	217	844	406	192	3024
Debian	49	62	97	174	283	418	691	1361	804	568	5514
Google	269	210	139	102	276	670	904	769	803	1087	6129
Linux	81	114	186	128	79	215	449	178	289	126	2721
Microsoft	667	212	699	486	1492	1405	1680	1445	2950	4788	19476
Mozilla	220	660	545	264	244	181		610	114	291	4424
Oracle	159	300	539	513	519	348	267	220	138	270	3350
Redhat	54	351	284	341	714	799	572	2489	1019	99	6548

Figure 4: A figure showing the exponential increase in CVEs per year per vendor, with well-known vendors such as Apple, Google, Microsoft shown. For example, Microsoft had 667 publicly reported CVEs in 2011. In 2021, they had 19,476.

From Figure 4, we can see from both the chart, and the table, that all vendors experienced an exponential increase in CVEs from 2011 to 2021. Just looking at Microsoft or Apple, like in Figure 5 and Figure 6, the trend looks strikingly similar – thousands from 2015-2020, tens of thousands in 2021.



	2011	2012	2013	2014	2015	2016	2017	2018	2019	2020	2021
Microsoft	667	212	699	486	1492	1405	1680	1445	2950	4788	19476

Figure 5: A figure showing the exponential increase of vulnerabilities in Microsoft products, year-over-year.

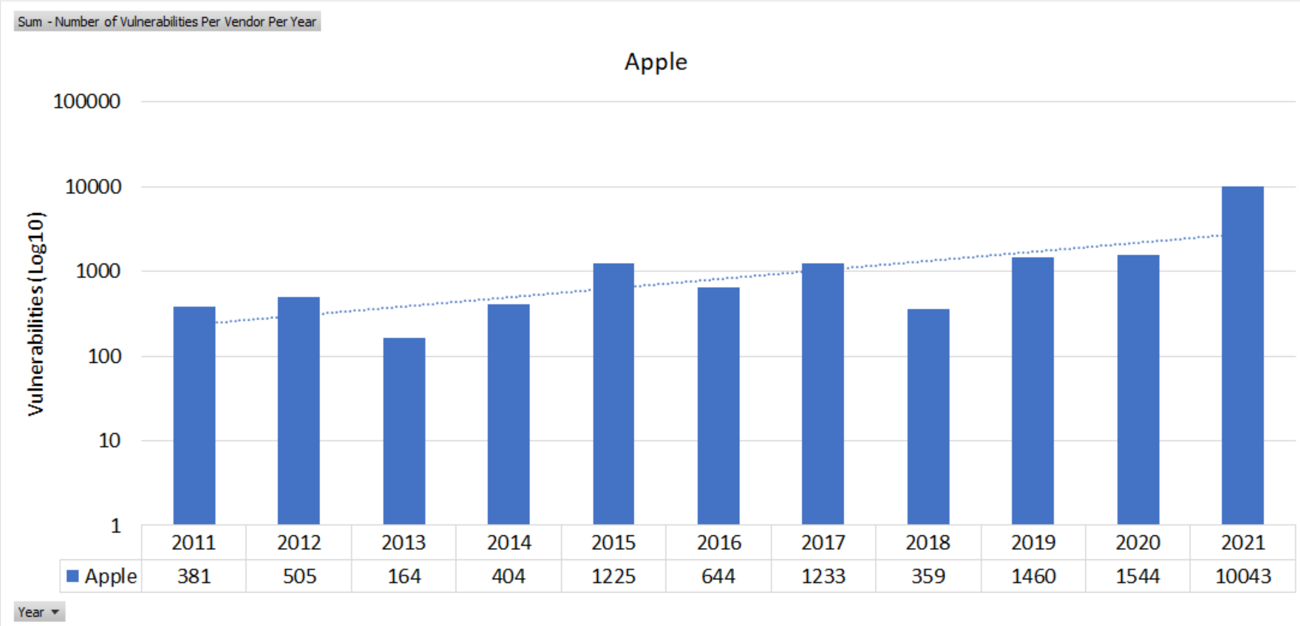


Figure 6: A figure showing the exponential increase of vulnerabilities in Apple products, year-over-year.

This data, at a glance, shows that the prevalence of CVEs is increasing exponentially. I think, this is the most important takeaway from this data.

A single CVE can take weeks or more to fix (worst case), depending on how complex the vulnerable code is. For any given company, manually patching these vulnerabilities may cost thousands of dollars per vulnerability. Let’s take Microsoft in 2021, with their 19,476 CVEs. Let’s assume it takes just 200 man-hours, which is 1 work week, 9am to 5pm, and $4000 (in wages, perhaps for multiple people) to fix 1 CVE.

This would equate to ~88 man-years of effort, and $77,904,000, to fix all 19,476 CVEs reported for all Microsoft products in 2021. Just from 1 work week and $4000 per software defect.

My point, from this quick “guesstimate”, is that it is far too labor-intensive to manage vulnerabilities at the current scale that they are being generated at, and automated systems are necessary to remediate and triage software vulnerabilities.

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 incomplete and oddly structured.

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.


```
POST / HTTP/1.1
Host: localhost:8080
User-Agent: python-requests/2.26.0
Accept-Encoding: gzip, deflate
Accept: */*
Connection: keep-alive
Content-Length: 825
Content-Type: multipart/form-data; boundary=6b2831f4273bc577fe9df2730debc447

--6b2831f4273bc577fe9df2730debc447
Content-Disposition: form-data; name="id"

%{(#instancemanager=#application["org.apache.tomcat.InstanceManager"]).(#stack=#attr["com.opensymphony.xwork2.util.ValueStack.ValueStack"]).
(#bean=#instancemanager.newInstance("org.apache.commons.collections.BeanMap")).(#bean.setBean(#stack)).(#context=#bean.get("context")).(#bean.setBean(#context)).
(#macc=#bean.get("memberAccess")).(#bean.setBean(#macc)).(#emptyset=#instancemanager.newInstance("java.util.HashSet")).(#bean.put("excludedClasses",#emptyset)).
(#bean.put("excludedPackageNames",#emptyset)).(#arglist=#instancemanager.newInstance("java.util.ArrayList")).(#arglist.add("ls -lash /")).
(#execute=#instancemanager.newInstance("freemarker.template.utility.Execute")).(#execute.exec(#arglist))}
--6b2831f4273bc577fe9df2730debc447--
HTTP/1.1 200 OK
Date: Mon, 08 Nov 2021 01:05:09 GMT
Content-Language: en
Content-Type: text/html;charset=utf-8
Set-Cookie: JSESSIONID=node0vrwycn10nxn167e0eguct3eu0.node0; Path=/
Expires: Thu, 01 Jan 1970 00:00:00 GMT
Content-Length: 2103
Server: Jetty(9.4.31.v20200723)



<html>
<head>
    <title>S2-059 demo</title>

</head>
<body>
<a id="total 84K
4.0K drwxr-xr-x   1 root root 4.0K Nov  8 01:04 .
4.0K drwxr-xr-x   1 root root 4.0K Nov  8 01:04 ..
   0 -rwxr-xr-x   1 root root    0 Nov  8 01:04 .dockerenv
4.0K drwxr-xr-x   1 root root 4.0K Nov 19  2020 bin
4.0K drwxr-xr-x   2 root root 4.0K Sep 19  2020 boot
   0 drwxr-xr-x   5 root root  340 Nov  8 01:04 dev
4.0K drwxr-xr-x   1 root root 4.0K Nov  8 01:04 etc
4.0K drwxr-xr-x   2 root root 4.0K Sep 19  2020 home
4.0K drwxr-xr-x   1 root root 4.0K Nov 18  2020 lib
4.0K drwxr-xr-x   2 root root 4.0K Nov 17  2020 lib64
```

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".

Related Research

- *[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

    - This source is useful to building out a generalized vulnerability discovery framework whose main strength is its use of a genetic algorithm to explore possible vulnerability sources in a time and computation-efficient manner.

      The idea of using a genetic algorithm in order to fuzz is a great enhancement on old-fashioned methods of fuzzing, and the paper details with a graph, that GA fuzzing is orders of magnitude faster than old fashioned fuzzing techniques.

      This is useful to this paper because the concept "exploitability requires discoverability". It also exposes useful ways of looking at exploit discovery such as "control flow penetration". This can be used to think about fuzzing vulnerabilities blindly, and why it is extremely inefficient – You are essentially attempting to reverse-engineer application logic, which entails crafting inputs that are able to take advantage of specific characteristics of control flow that may be flawed in some way.

- *[2] "Vulnerability scanning of IOT devices in Jordan using SHODAN," IEEE Xplore. [Online]. Available:* https://ieeexplore.ieee.org/document/8277814.

    - This paper is a survey of vulnerable/exposed IoT devices in Jordan. It surveys them and shows that there exist a high number of exposed and publicly available IoT devices, which are likely vulnerable to a number of known and possibly non-publicly-disclosed vulnerabilities.

      The conclusion we can draw from this paper is, that even using publicly available tools like Shodan, you can obtain a shockingly high coverage of vulnerable or possibly vulnerable devices – which is pertinent to this paper.

- *[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.

    - This paper covers the domain of "Autonomous Cyber Reasoning System[s]", which are systems that are able to automatically enumerate, analyze, exploit and/or patch vulnerabilities *in already-assembled binary files.* This paper also explores other concepts related to automated vulnerability detection, and discusses shortcomings of fuzzing tools (like AFL, as its fuzzing engine is syntax-blind, or Fortify SCA, as it is static and knows nothing of the runtime, variables, etc).

        - It mentions specifically that augmenting logic tree traversal, whether it be to augment code or to exploit it, would be well-served by some sort of guided ML approach.

- *[4] "A machine learning-based approach for automated vulnerability remediation analysis," IEEE Xplore. [Online]. Available:* https://ieeexplore.ieee.org/abstract/document/9162309.
    - This paper explores a system for deciding what to do given any specific security vulnerability and a patch: Do we patch? When? How many devices/servers are vulnerable to the vulnerabilities? Would the patch negatively affect devices?
        - Dealing with vulnerabilities en masse, and not as a problem that a single human or group of humans can solve, is a practical way to look at vulnerabilities given how common and widespread they can be, not to mention the benefit of having split-decision actions be taken at times (4AM?) that humans will likely not be present to respond.
- *[5] "Robot hacking games," Center for Security and Emerging Technology, 05-Oct-2021. [Online]. Available:* https://cset.georgetown.edu/publication/robot-hacking-games/.
    - This 2021 document outlines the actions taken by China in an attempt to emulate DARPA's Cyber Grand Challenge, in 2016.
        - The gist from this paper is that other nations and nation-states are likely interested, and probably taking part in the creation of Autonomous Cyber Reasoning Systems, and, although this is not stated in the paper, likely for malicious intent.
- *[6] "Automated vulnerability analysis: Leveraging control flow for evolutionary input crafting," IEEE Xplore. [Online]. Available:* https://ieeexplore.ieee.org/document/4413013.
    - This document explores the massive increases in efficiency you can achieve with non-naive fuzzing approaches, such as genetic algorithms and markov models to guide the mutation of inputs into software fuzzing approaches.
        - The takeaway for this document for me is, that these fuzzing techniques, specifically applying ML and statistical/graph theory-related concepts, is a great way to extend current fuzzing and parsing models related to security vulnerabilities.
- *[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.
    - KameleonFuzz is a newer fuzzer, that also uses GA (genetic algorithms) for its fuzzing. It has some nicer features than [6], like trying to infer the exact place the taint is processed within the black box that reduces queries required to find a vulnerability. (i.e. What type is the variable? Is the taint sink the DOM? The database?)
- *[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.*
    - This paper covers parsing code to tease out possible vulnerabilities from it – but not with Abstract Syntax Trees, but with "Code Property Graphs".

◦

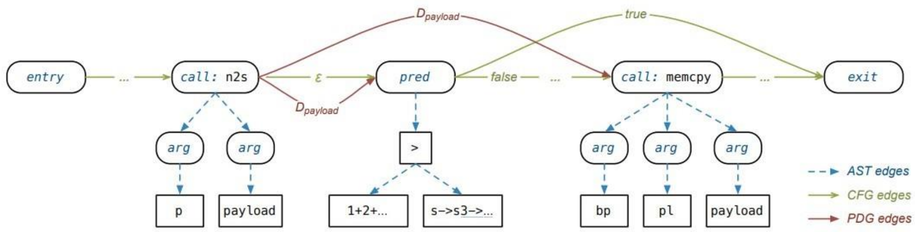


Fig. 2: An excerpt of the code property graph for the "Heartbleed" bug. Data-flow edges, control-flow edges and syntax edges are indicated by red, green, and blue color respectively.

- ◦ A CPG (Code Property Graph) is a representation of control flow and data flow, and makes it easy to query these trees (graphs) for useful vulnerability data.
- ◦ This paper concludes a >90% reduction in analysis efforts for the software they tested, OpenSSL, if you use their method versus manual review.

- *[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/

  - ◦ This paper covers how CVE-2017-5638 enabled attackers to (probably, 50-50 chance; ~143 million Americans affected) breach your social security number.
    - ▪ I included this paper to demonstrate to readers how important some concepts in this survey paper are – The Equifax breach didn't have to be automated. It could have easily been done by partially manual processes, by humans.
      - Hell, there are still Equifax XSS bugs that either aren't patched or took >3 years to patch – Check Twitter. I'm sure there are plenty of monolithic companies that simply don't have the resources, a mature cyber program, or simply don't care enough to fix some of these very critical bugs.
      - Can you imagine if some of the companies, or even our government, were subject to a highly automated, distributed, and sophisticated attack using some and/or all of the techniques discussed in this paper?

- *[10] T. Avgerinos et al., "The Mayhem Cyber Reasoning System," [Online]. Available: https://ieeexplore.ieee.org/document/8328972*