Research Statement

In the past year, we have seen a remarkable increase in exploitation of vulnerabilities in open-source libraries, e.g. Heart Bleed, Freak Attack, and POODLE etc. The popularity and widespread use of 3rd party components make these libraries an attractive target for hackers. Developers often have very little knowledge about the components used in their projects. At SourceClear, I have built a novel framework to identify and discover all the components used in a software project, analyze the vulnerabilities in those components and suggest automated remediation and patching. We also analyze the capabilities of libraries and generate security profiles that give more visibility into the functionality present in each library. In addition, we employ techniques based on data mining and machine learning to learn new vulnerabilities and find previously undisclosed issues in popular libraries and components.

SourceClear provides a complete safety net for developers using libraries and components. The new vulnerabilities and undisclosed issues that we discover from machine learning during the project are also published (https://srcclr.com/registry/explore) via a security intelligence feed.

Our framework is composed of 4 major parts (P1-P4). A major novel and distinct feature of our proposal is the emphasis on polyglot analysis. We have developed techniques that work with a variety of programming languages (including but not limited to Java, Ruby, C, C++ and JavaScript) and build systems (like Maven, Ruby Gems, NPM and Make files). The main focus of our framework is to automatically find components with known vulnerabilities in an application. This vulnerability type is one of the OWASP Top Ten [2] most common security issue. Also, due to the gap between the time to disclose and time to fix, all other security bugs eventually become issues of using components with known vulnerabilities. Thus by targeting and focussing on this particular vulnerability type we ensure that our approach is complimentary to any other analysis that is geared at finding specific security issues.

P1: Analysis of Library Dependencies in Software

We have developed an analysis that can find all the 3rd party libraries and components used in a software project. This is a challenging problem because to resolve transitive dependencies in a project we need to create an environment similar to the one used to build the project. We have found that it is essential to consider the transitive dependencies in order to build the most accurate inventory of the components used in an application. Existing tools like OWASP Dependency Checker [1] are based on source code analysis only and do not consider the knowledge of all the libraries in various package repositories, thus, they fail to locate transitive dependencies.

Finding the components used in an application is the first step towards vulnerability analysis. The second step is to use this information about components and match it against known vulnerabilities. We use high quality fingerprints based on build coordinates and byte-code hashes to match the vulnerable components and their versions accurately. Based on our initial experiments we have found that relying only on file hashes, CPEs or manifest information leads

to lot of false positives and false negatives (https://blog.srcclr.com/using-cpes-for-open-source-vulnerabilities-think-again/). The OWASP Dependency Checker [1] uses CPE information and thus gives large number of false positives.

P2: Capability Analysis of Libraries

Developers place a lot of trust in public repositories of 3rd party libraries. Often developers download and install libraries in their project without having any idea of what capabilities or features those libraries have. To address this issue we propose to build security profiles of libraries. The security profile captures all the capabilities the library has, i.e. a security profile can answer questions like Can the library access the network? Does it implement a cryptographic algorithm? Does it write to file system?

We have built a static capability analyzer that can construct the security profile of the component and tell if it is using capabilities like Network, I/O, and Database. To build a security profile we first define for each platform (Java, Ruby, Node.js) a capability API specification. The specification maps various method calls supported by the underlying platform to a particular capability. During the profiling process we can look for these method calls to determine if the component uses that particular capability. We are unaware of an existing system that can provide security profiles for various libraries used in an application.

P3: Exploitability Analysis of Vulnerabilities

Sometimes the vulnerability in the library may not be exploitable in the application since it doesn't use the vulnerable part of the library. Or, the library may not be vulnerable in all configurations. In all of these cases relying only on version information may lead to false positives, i.e. we may mark some projects as vulnerable even though they really cannot be exploited. To improve this, we built a vulnerable parts (methods) analysis of the library (https://blog.srcclr.com/vulnerable-methods-under-the-hood/). This vulnerable parts analysis includes a static analysis of the application and library (including call graph construction). To the best of our knowledge no other system or framework provides such detailed exploitability analysis of vulnerabilities.

The vulnerable parts analysis involves the following steps:

1. While analysing the vulnerability and the affected components we identify the methods calls and files that are responsible for the vulnerability.
2. This vulnerable methods and files information will be used to trigger an automated analysis of the component to discover all the other files in the component where that method is called (or that file is used).
3. All those method calls and files together form the vulnerable part of the component.
4. While scanning the source-code of a project if we find it using a vulnerable version of the library, we can trigger a more detailed analysis that compares and checks if the project is using the vulnerable part of the library.

5. This will improve the accuracy, as we will only report projects that are using vulnerable parts within the vulnerable version of the library.

As an example, consider the CVE-2014-3577 [3], it reports a vulnerability in a well-known and widely used Java library. According to the CVE information, the Apache HttpComponents HttpClient library before version 4.3.5 is vulnerable. Thus, if we only rely on this information and we find that the httpclient-4.3.4jar component is used in a software application we would mark that application as vulnerable. However, in this particular case the only vulnerable method in the library is the following one from the AbstractVerifier class - "org.apache.http.conn.ssl.AbstractVerifier.getCNs()". A more detailed analysis takes this into account and following the steps (1-5) detailed above, marks the application as vulnerable only if it makes a call to the vulnerable method.

P4: Discovery of New and Unknown Vulnerabilities

We collect a wide variety of data about vulnerabilities and libraries during this project, we can use pattern matching and machine learning based algorithms on this data to find new and undisclosed vulnerabilities. We have set up an Apache Spark Cluster to help in big data analysis. From a known vulnerability, we extract patterns based on the vulnerable methods and then scan all the components in our dataset to check if any of them use the same pattern. We are also working on machine learning based classifiers to predict vulnerable components. In our initial experiments, we have found that for some vulnerabilities, the impact due to undisclosed components to be as high as 20 times (i.e. we found 20x more issues than known publicly).

Limitations and Challenges:

Our framework is not a silver bullet for software security. Developers will still need to rely on other tools and be diligent in building their software. Since our perspective in this framework is based on, the developer of the application himself using the tool to improve the security of software, we do not consider things like malware, bad actors or active hacking attempts in this work. In addition, security above the application layer including front-end issues like XSS, SQL Injection etc. are better addressed using tools that apply static and dynamic analysis techniques on source-code.

References:

[1] OWASP Dependency Check,
https://www.owasp.org/index.php/OWASP_Dependency_Check

[2] Using Components with known Vulnerabilities,
https://www.owasp.org/index.php/Top_10_2013-A9-Using_Components_with_Known_Vulnerabilities

[3] CVE-2014-3577, http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-3577