SourceClear

The Busy Managers’ Guide To Open Source Security
Introduction

In light of the recent Equifax hack, there is a lot of talk about the security of open-source code and no shortage of advice from pundits about what companies should do. The need for a security strategy for open-source code is not a new concept and the recent data breach has highlighted its importance even more. This document is a guide to open-source security so your company doesn’t become the next Equifax.

It covers:

• Why the growth of open-source code has become a problem
• The security problems with open-source code itself
• The process that tech companies should have in place to protect themselves
• Important facts to consider when creating a security solution

Why the growth of open-source code has become a problem

In a recent survey, almost 80% of all companies, from enterprises to hot silicon valley startups, say they now rely on open-source.

As open-source has become mainstream, the amount of code available in the form of libraries has exploded. The frequency with which that code is being released is also increasing and open-source code is constantly being broken up into smaller and smaller chunks of reusable code in the form of libraries. We have done the math and estimate that if things continue at the same rate, there will be almost half a billion open-source libraries available to developers within a decade.
To further complicate matters, software developers rely on tools called package managers to determine which libraries and which versions of libraries are needed by the software they are creating. These package managers make many important decisions on behalf of the developer behind the scenes, in most cases without them ever realizing. For instance, for every one library that a developer chooses to use, package managers can add as many as ten more libraries to the final code, creating complex dependency trees. The result is that when an application built with open-source code is finally ready to be shipped, it’s usually made up of 90 percent of code that came from other people.
This mass of reusable code represents vast amounts of reusable vulnerabilities to hackers. Much like the conditions that allowed computer viruses to proliferate, these vulnerabilities have created a complex and extensive attack surface that is very attractive for the bad guys. Open-source is now being attacked en-mass. At SourceClear we see vulnerabilities discovered in around 75% of all projects scanned representing a major business risk for organizations who don’t currently have an open-source security strategy implemented.

The traditional approach to application security has been to invest in educating your own developers (training and policies) and in running security code analysis tools before your code is shipped. In the open-source world we live in today, this approach is no longer valid because 90 percent of the code simply bypasses those checks.

The security problems with open-source code

When using open-source code, you need to be able to answer four key questions:

- What open-source code am I using?
- Where did it come from and should I trust the authors?
- Does my code contain any vulnerabilities?
- Does the code do anything bad?

Security is a big and complex topic. For the sake of this guide, let’s focus on answering the two most important questions, “Does my code contain any vulnerabilities?” and “Does the code do anything bad?”
Type of Vulnerabilities

Open-source code vulnerabilities can be classified into seven types. We wrote about this in a blog post, The Seven Deadly Sins of Open-Source Libraries, in September 2017. The seven types are:

1. Disclosed Vulnerability - a vulnerability where information is available in public databases such as the National Vulnerability Database in the form of CVE’s. CVE is a claim based system and claims require secondary analysis, verification and data enrichment such as the vulnerable versions and the vulnerable methods.

2. Inherited Vulnerability - a new vulnerability that is the result of a library inheriting a library with another vulnerability via its dependency & call graph (both conditions needed). The typical Java library inherits four other libraries and the typical NPM module inherits nine other libraries, making inherited vulnerabilities quote common.

3. Embedded - a new vulnerability that is the result of inheriting a library with another vulnerability by embedding its code (usually as a result of cut-and-paste or adding a JAR file or XML parser in a parent library). Sites like Conjars make these type of vulnerabilities a growing problem.

4. Similar - a new vulnerability that is the same or similar to another known vulnerability but that is now found in a different library.

5. Reintroduced - the same vulnerability that has been fixed in a previous release of the library but that has been reintroduced in a later version of the same library. This is quite common when libraries maintain multiple versions.

6. Zero Days - new issues that have not yet been the subject of disclosure but known about by someone and likely being used by the bad guys in the wild.

7. Half Days - new issues that have not yet been the subject of disclosure but can be found in places like commit logs, change-logs and issue trackers if you know where to look. Half days are often obscured and sometimes hidden but more often than not hiding in plain-sight.
The Equifax hack was the result of not patching a disclosed vulnerability, in this case CVE-2017-5638, in the Apache Struts framework. Patching disclosed vulnerabilities is the most basic level of attention you could pay to the problem. Equifax should have done much better in this regard, but just paying attention to disclosed vulnerabilities won’t be sufficient to keep your business safe. Later in this paper we talk about why you should not rely on the CVE system and the National Vulnerability Database for open-source security vulnerabilities alone.

Malware

While attacks like those at Equifax exploiting vulnerabilities are grabbing the headlines today, we think malware in open-source will be making the headlines in the future. Our researchers have demonstrated at security conferences how one innocent developer using a single malicious open-source library could easily lead to the total compromise of a web application, like an online bank. Ransomware has been a very profitable business on desktops, and the bad guys are now turning to open-source used in business applications as the mule to deliver their evil payloads to much higher-value targets.

Malware activity usually starts on the dark web but signs are now starting to find its way into the public open-source ecosystem. Recently, 39 libraries written in JavaScript and hosted at npmjs.org were found to contain malware written by a developer in Russia and went undetected for two weeks. When these libraries were used by developers, the hackers would read environmental variables on developers’ desktops and central build servers, variables that are often used to store SSH keys and API keys to services like AWS and GitHub. On September 14th, 2017, another 20 Python packages were found to contain malware.
The reality is that developers are used to adding code without checking what it does, and sophisticated hackers have developed simple attack techniques that are subtle and effective. Inspired by phishing attacks, they create libraries that look like innocent libraries but contain malicious code. Some hackers have even built up a following using legitimate code only to modify it later with malign payloads that infect all of their followers when they upgrade to newer versions.

It maybe vulnerable open-source libraries we are talking about today, but the threat of malware is quickly approaching.

**The process and technology that companies should have in place to protect themselves**

When the blame game started after the Equifax event was made public, some tried to defend the company saying that finding vulnerable open-source is hard and patching harder still. This is untrue. If any company is not taking the following basic steps, they will likely be found guilty of negligence if hacked. All companies should have the following four steps in place:

- Create and enforce security policies
- Know what is being used and know what is vulnerable
- Update vulnerable libraries
- Prevent bad code from being used
Create a policy

When it comes to open-source usage within your organization, developers need to know what is acceptable and what is not. Creating and communicating a company policy that is easy to understand and comply with and easy for you to enforce. Here is a simple, real-world policy as an example from a progressive technology company.

You can use whatever open-source code you want as long as:

1. You have checked the code and verified that it does not contain malware or code generally considered to be dangerous or malicious.
2. The versions you are using are free of high and medium risk vulnerabilities as reported by SourceClear.
3. All production code found to be calling the vulnerable methods of high and medium risk vulnerable libraries must be patched and redeployed immediately.
4. All production code with high and medium risk vulnerable libraries but where the vulnerable methods are not being called must be patched and redeployed in the next significant release.

We have seen other companies that have detailed policies that talk about only using libraries that are actively maintained and about how developers should only use code where they know the authors. However, we have generally observed that these policies are largely ignored. Short, simple and enforceable works.
Know what open-source is being used and what is vulnerable

When you have a policy in place (or even before you do), the first step in getting ahead of open-source code security is knowing what is being used. We call this Visibility. You need to know what libraries and what versions of those libraries are being used by which projects. An old fashioned way to describe this was a “Bill of Materials,” but modern developers recoil at people describing software in manufacturing terms so we suggest you avoid using the term if possible.

You could conduct a company-wide audit of all projects, but the reality of code built with open-source is that any small change can result in a big change in the final system, including changes in the open-source being used. Because of this, point-in-time audits aren’t very helpful. In a modern DevOps and continuous delivery world, code is changing hundreds of times a day and it’s highly unlikely that your inventory will be accurate when you come to read it, yet alone when you actually need it. A one-off audit can be helpful to understand just how much open-source is being used.

The right approach is to scan all code when every change occurs, and maintain an “always up-to-date” inventory of what is and what has been used at any point in time. This is best accomplished by integrating a specialized static analysis scanner like SourceClear into your build environment (continuous integration server). Every time quality tests are run, open-source security tests are also run and the results recorded. It is important that you do this using real static analysis when the build happens so you can see the complete dependency graph. Simple solutions that attach to or are integrated into source code management systems, like GitHub, simply grep the build files and so only work for non-compiled scripting languages like Ruby on Rails and not for Java, used by Apache Struts.
When you know what libraries are being used by each project (and if the library is a direct or indirect dependency), as well as what versions of those libraries are being used, you can match that information to live vulnerability services like SourceClear and know immediately which libraries are considered vulnerable. For those that are and that match your policy, you can update them.

When scanning, it's important to understand that the majority of time when developers use a vulnerable library they don't actually use the vulnerable parts of the code and therefore their projects aren't actually vulnerable. In a study with our customers, 92 percent of Java projects that use vulnerable libraries don’t use them in a vulnerable way. This is important as developers are busy and often skeptical of security and so asking them to update vulnerable libraries where there isn't an actual vulnerability is like “the boy that cried wolf.” Specialized static analysis is the only way to determine if the vulnerable part of a vulnerable library is being used.

**Update vulnerable libraries**

Almost everyone today is using insecure open-source code. Once this reality is accepted, it becomes clear that collaboration between security teams and software developers is key to remediation. Our experience is that integrating open-source code security seamlessly into existing tools and processes, such as the Git workflow and ticketing systems like Jira, is absolutely critical.

When asking developers to update libraries, you will likely face resistance with typical responses that it’s hard and that updating may break the application. This is often called [Dependency Hell](#) in developers circles and is unfortunately a fact of life. Luckily, developers are smart and work on hard problems all day long and so when empowered with a little information can almost always work it out.
Updating direct dependencies, those specified by the developers of your code, is usually a matter of changing the version in your code and re-building the project. Security Automation Tools like SourceClear can help developers see if they are using parts of the library that have changed between versions, and pinpoint additional changes that may need to be made or at least point them in the right place to look.

Updating indirect dependencies (libraries required by other libraries) is hard. Developers usually have little visibility about what indirect libraries are used and little or no control on what versions they use. There are usually temporary workarounds to override vulnerable libraries by adding later versions as direct dependencies in your projects.

Most companies adopt a policy of requiring vulnerable libraries to be updated if the project is actually using the vulnerable parts of the vulnerable code. This maintains safety while reducing the amount of updates by as much as 90 percent.

**Prevent bad code from being used**

The old saying goes that prevention is better than a cure and it’s entirely possible to create a solution that goes a long way to preventing bad open-source from entering your software supply chain. While there are many options, such as only serving up binaries from a local repository that are known to be “clean,” our experience is that this is usually easily bypassed and rarely effective. Unless you plan to stop your developers from using the Internet, they will always find a way to bypass these controls in order to be more effective at their jobs.
What we have seen to be effective is for companies to create enforcement rules on their continuous integration servers that will fail the build if projects using the vulnerable parts of high and medium risk vulnerable libraries are found before the code is pushed into staging or production. Continuous integration servers act as a central checkpoint for teams and therefore provide an effective team-wide control. Companies often find that initially creating warnings goes a long way and that by creating a combination of warnings and errors based on severity is usually enough for developers to self police themselves. Integration into CI allows you to trust, but verify, developers.

**Important facts to consider when creating a solution**

This section describes critical facts to consider when creating or choosing a security automation solution for open-source code. Security can be complex and requires specialist technology from genuine experts in the field. Some developer tools companies are now positioning their technology as security tools and companies that were experts at managing open-source licensing are now playing security experts on the Internet. The following section will help you identify and evaluate what is right for you.

**Your open-source code scanner must perform real static analysis**

**Summary:** Only use a tool that does real static analysis.

The first generation of security code review tools like FlawFinder and RATS used grep to find potential security issues. Grep is a Unix utility to specify a search string to match on. The first generation of license checking tools also used these same techniques and many have carried it over and still use it in their security solutions today. Application security professionals knew that grep was not
sufficient and in the 2000’s we saw the birth of static analysis security testing tools (SAST) like Fortify and Coverity which have now become the defacto standard for security review of custom code. Static analysis is a technique where you build the project and perform a complete analysis of the programs source code or object code. In static analysis you traverse the codes call structure and data-flow using algorithms to look for patterns and signs that the software could be compromised.

Static analysis is also a critical technique when applied to the problem domain of open-source code security. In fact, this is the only way to determine all the code present and if code is actually being used in a vulnerable way.

Static Analysis allows you to create a list of all open-source being used. This structure of libraries specified by the developer and libraries in turn required by those libraries is called a Dependency Graph (DG). The DG is important because 75-90 percent of the open-source libraries are only pulled into a project when the build occurs. This means that if you don’t use static analysis and only scan the build files checked into the source-code, you miss the majority of the code and are only analyzing 25 percent of the code. Some scripting languages do produce additional files (generally called lock files) than can help, but these are not universally adopted and only applicable to some cases.

Even when including vulnerable libraries, 90 percent of the time projects aren’t using the compromised code in those libraries, and therefore the vulnerability is not real. The only way to determine if the potential vulnerability is real is to create a Call Graph (CG), which describes ways in which the program operates, and then to trace calls from your project down to the vulnerable methods in the Dependency Graph (DG). SourceClear uses control flow analysis on the CG and DG to perform this.

If you are not using specialized static analysis, you are not doing it right.
Be mindful where you scan and enforce your policy

Summary: Implement scanning in your Continuous Integration System.

As we mentioned at the start of this section, some developer tools companies are now positioning their technology as security tools. We think this is dangerous. There are many places in the lifecycle of an application from coding to deployment where you could in theory apply security testing and technical controls, but there are significant advantages and disadvantages in each scenario.

Developers Desktop / Code Editor - There has been a trend to “shift left,” meaning move security testing earlier in the development cycle and so Integrated Development Environments or IDE’s sound like an interesting option. Having developers find and fix security issues as they code is largely an urban myth that stemmed from a flawed study in the 80’s with less than a hundred participants. Even if it were cheaper to find and fix bugs early, scanning open-source code on a desktop is problematic. To start with, most code doesn’t build from source, meaning that you can usually only do simplistic analysis and not a full static analysis of the project. Secondly, most developers use a wide variety code editors and integrated development environments (IDE’s), which would be hard for any security tools vendor to keep up with. Certainly asking a developer to change tools to get security is a non-starter.

Binary Repository (Artifactory / Nexus) - there is a class of tools called binary repositories that sit on a company's network and serve up open-source code to local developers and build tools. These tools are intended to cut down on traffic and avoid hundreds of developers fetching the same library from the Internet over and over again. The two big vendors in the space are Sonatype with their Nexus product, and JFrog with their Artifactory product. Both now offer security options.
While serving up vulnerable code from a system you control is akin to having a local file server with malware and only hosting safe versions seems logical, the reality is that unless you have perfect network access control and developer endpoint controls, there are hundreds of ways developers can and will still download open-source from the Internet. These solutions just provide a warm and comfy feeling but little to no real protection. Even if they did and perfect network and endpoint controls were in place, you still can’t determine which projects contain bad open-source or if it is being used in a vulnerable manner.

Source Code Management (GitHub) - some security vendors have built plugins to source code management systems like GitHub where teams manage source code. Integrating with source code management sounds convenient, but it is not suitable for open-source security testing. In order to do a complete and accurate scan, you need to perform static analysis. This can only be done by actually building the code. Most code does not build from source without significant human wrangling, and source code management tools themselves do not provide build features. The only analysis that can be done on source code management systems is superficial, quick scans that look at the direct libraries. Superficial security testing is never a good option.

Continuous Integration - In the world of DevOps, all teams now run continuous integration servers that take code when it is checked into source code management and continuously build it and run a barrage of tests. Progressive teams are further automating this in a process called Continuous Delivery, creating workflows and rules that automate publishing code if it passes the quality criteria. Continuous integration is the only place where open-source security testing is effective.
First of all, it’s where code is built. This means you can perform a full static analysis on it and know with confidence all of the open-source being used and how it is being used. Secondly, it’s the place where developers check for other quality issues and is a familiar checkpoint that is constantly monitored. Finally, as a central checkpoint, Continuous Integration servers have built-in functionality to fail a build if conditions are not met, allowing you to prevent bad open-source from being shipped into production in the first place.

**Most vulnerabilities are not in public databases like the NVD**

**Summary : Don’t rely on the NVD and CVE’s**

The truth of the matter is that most vulnerabilities are simply never reported i.e. not disclosed and are not present in public vulnerability databases like the National Vulnerability Database. In fact, we estimate that of the total gene pool of open-source library vulnerabilities, it is likely made up of something like only 5 percent of disclosed issues in the form of CVE’s. Traditionally, all security advisories from vendors like Microsoft and Oracle have been assigned CVE’s so that the public can track them, but the CVE system is neither complete nor effective for open-source security issues.

In the open-source code world, when one of the millions of developers discover a vulnerability or one is reported to them by their users, they generally do one of three things: ignore it, file a bug to be fixed later or simply fix it right then and there on the spot. Few developers ever go through the laborious process of filing a report and waiting months for the the report to be published in the NVD before discussing the issue. Even established open-source communities like the Apache Software Foundation, who have an official policy of creating a CVE for all security issues, don’t get them for the vast majority of issues they find and fix. We know because we watch the code in their projects and expose where issues are being quietly found and fixed.
To highlight this, I just used SourceClear to scan the Apache Spark project, a very popular open-source big data system. There were 9 libraries with CVE’s and 22 libraries with vulnerabilities that are not in the CVE system, including a high risk remote code execution flaw.

The problem is more complex for the NVD and CVE system than apathy or lack of awareness from developers. The open-source ecosystem itself is alive with thousands of new versions of libraries being published everyday and the relationships between libraries is changing constantly. Even when there is a report that a specific version of a specific library is vulnerable, the NVD (or any public database publishing static data) isn’t able to build the inter-relationships to see what else is affected. Even if it did it simply wouldn’t be able to keep that up-to-date.

CVE is also a claims-based system where anyone can claim an issue and it’s up to someone else to dispute that claim. Our researchers perform secondary analysis on all CVE’s and know that many claims are incorrect. Let’s take CVE-2017-1000034 as an example.

Akka versions <=2.4.16 and 2.5-M1 are vulnerable to a java deserialization attack in its Remoting component resulting in remote code execution in the context of the ActorSystem.

The CVE description of the vulnerability simply says that the Akka framework is affected by a deserialization vulnerability. It provides no details about the underlying component that is affected or the attack vectors that can be used to exploit the issue. After analysis, we know that it is the akka-actor library that is affected and further analysis showed that the vulnerable library is used by 357 other libraries (357 other inherited vulnerabilities).
Collaboration between security and developers is critical

Summary: Integrate with developer’s ticketing systems to get things fixed.

Software development is a team sport and updating open-source libraries requires collaboration from both security and developers.

Security teams can analyze the potential business impact and risk of vulnerabilities, and developers are required to create the updates and push the code through tools and into production. Historically, security teams worked in parallel tools for developers and communicated in unfamiliar ways using Word documents.

Developers live in ticketing systems like Atlassian’s Jira and on chat systems like Slack. When asking a developer to take action it is critical that communication happens inside of these channels. Not only is this the familiar way to communicate, it allows developers to rank and prioritize their work relative to the other tasks they are inevitably working on. Tight integration into tools like Jira also allows security teams to know when issues have been closed and which ones are still open.
Conclusion

Open-source is growing exponentially, and with its unchecked growth comes massive security threats. This is not a problem that is projected to happen in the future, it’s a problem that is real and present in the world of open-source code today. The number of vulnerabilities will continue to grow with open-source code, and the problem will grow with it. We’ve seen how ransomware has affected businesses and we expect ransomware to be the next wave of issues to affect software built with open-source code.

C level executives are being fired, companies are losing billions of dollars in market cap and governments are talking about regulation in the hope of halting these data breaches.

Solutions exist and companies need to ensure they have a security process that uses technology to identify and mitigate these threats. Utilizing specialized static analysis while building software integrated in your continuous integration system is the only way to keep ransomware and open-source vulnerabilities out of your business applications.

Are you ready to see if your applications are secure? With our 30 day Trial you can get started today.