Vulnerability Detection in Open Source Software: The Cure and the Cause

Millar, S. (2017). *Vulnerability Detection in Open Source Software: The Cure and the Cause*. Queen's University Belfast.

Document Version:
Publisher's PDF, also known as Version of record

Queen's University Belfast - Research Portal:
Link to publication record in Queen's University Belfast Research Portal

Publisher rights
Copyright the author 2017.

General rights
Copyright for the publications made accessible via the Queen's University Belfast Research Portal is retained by the author(s) and / or other copyright owners and it is a condition of accessing these publications that users recognise and abide by the legal requirements associated with these rights.

Take down policy
The Research Portal is Queen's institutional repository that provides access to Queen's research output. Every effort has been made to ensure that content in the Research Portal does not infringe any person's rights, or applicable UK laws. If you discover content in the Research Portal that you believe breaches copyright or violates any law, please contact openaccess@qub.ac.uk.

Open Access
This research has been made openly available by Queen's academics and its Open Research team. We would love to hear how access to this research benefits you. – Share your feedback with us: http://go.qub.ac.uk/oa-feedback

Download date:06. Feb. 2024

Vulnerability Detection in Open Source Software: The Cure and the Cause

Stuart Millar, *PhD Cyber Security Researcher, 13616005, Centre for Data Science and Scalable Computing, CSIT, Queen's University of Belfast*
smillar09@qub.ac.uk

***Abstract*— According to Veracode, a Gartner-recognised leader in application security, 44% of applications contain critical vulnerabilities in an open source component [16]. Most companies do not have a reliable way of being notified when zero-day vulnerabilities[1] are found, or when patches are made available. This means that attack vectors in Open Source Software (OSS) exist longer than they should. This paper discusses the cause of OSS vulnerabilities, why they are a major issue, and how they may be mitigated. Conventional methods of detection are discussed along with novel approaches and research trends. A new conclusion is made that it may not be possible to replace expert human inspection of OSS although it can be effectively augmented with techniques such as machine learning, IDE plug-ins and repository linking to make OSS implementation and review less time intensive. Underpinning any technological advances should be better knowledge at the human level – development teams need trained, coached and improved so they can implement OSS more securely, know what vulnerabilities to look for and how to handle them. It is the use of this blended approach to detection which is key.**

***Index Terms*— open source software, cyber security, vulnerability detection, static analysis, dynamic analysis, software assurance, machine learning.**

I. Introduction

Open source software is that which is developed collaboratively in the public domain with a licence that grants rights to the user base which are usually reserved for copyright holders. A well-known open source licence is the GNU General Public Licence that allows free distribution under the condition that further developments are also free. In a globally connected software society, a sizeable amount of development work is effectively crowdsourced to an international community of OSS developers with little understanding of the security problems this creates [1]. 3rd party libraries increase development speed but there is a corresponding increase in risk also, with the Heartbleed bug in OpenSSL[2] being a prime example.

Research into vulnerability detection in OSS is crucial as more than half of the Fortune Global 500 companies use vulnerable OSS components, with vulnerable libraries also being repackaged in software. This OSS uptake shows no sign of reversing or slowing, with a Black Duck Software survey [19] indicating that 43% of respondents think OSS is superior to its commercial equivalent. Black Duck Software are a global provider of note with regard secure management of OSS code.

Another study carried out in 2012 from Aspect Security (a founding member of the Open Web Application Security Project, OWASP) and Sonatype found that more than 50% of the Fortune Global 500 companies have downloaded vulnerable OSS components, security libraries and web frameworks [2]. This study analysed 113 million Java framework and security library downloads by more than 60,000 commercial, government and non-profit organisations from the Central[3] Repository. The report found the vast majority of library flaws remain undiscovered, that the presence of a vulnerability (or an absence of one) is not a security indicator, and that typical Java applications are likely to include at least one vulnerable library.

Further, the same study shows most organisations do not have a strong process in place for ensuring the libraries they rely upon are up-to-date and free from vulnerabilities. The study stresses there are no shortcuts and they go as far as saying the only useful indicator of library security is a thorough review that finds minimal vulnerabilities – in other words, software assurance, or the measure of how safe the software is to use, needs to be generated internally. One might say this is surprising, as in many other product or service industries, this assurance – consider it some kind of warranty or seal of approval perhaps – is offered up by the supplier without hesitation to help build trust and sell to the customer.

At the crux of OSS vulnerability is that today's applications commonly use thirty or more libraries which in turn can comprise up to 80% of the code in any such application. These libraries have the same full privileges of the application that use them, letting them access data, write to files or send data to the internet. Anything the application can do, the library can do.

This paper was submitted on 26th March 2017 by Stuart Millar, PhD Cyber Security Researcher at the Queen's University of Belfast. Email: smillar09@qub.ac.uk

[1]A zero-day vulnerability is an undisclosed software vulnerability that hackers can exploit to adversely affect programs, data, additional computers or a network.

[2] Heartbleed results from improper input validation (due to a missing bounds check) in the implementation of the TLS heartbeat extension. The vulnerability is classified as a buffer over-read, a situation where more data can be read than should be allowed.

[3] Central is the software industries most widely used repository of OSS with more than 300,000 libraries.

Aspect Security estimate that custom built Java applications contain 5-10 vulnerabilities per 10,000 lines of code. A library has on average 10,000 to 200,000 lines of code, therefore the chances a library has never had a vulnerability are very slim, with it being more likely (if it has been classed as 'safe') that it has not been examined for vulnerabilities. Hence libraries with no vulnerabilities should not automatically be considered 'safe'. [2] states most vulnerabilities are undiscovered which, based on this reasoning, seems logical, and recommends that the only way to deal with the risk of unknown vulnerabilities is to have someone who understands security analyse the source code. Tool support provides hints but is not a replacement for experts because, as we will see when we study existing conventional methods of detection in Section II, the lack of context within libraries makes it virtually impossible for tools to conclusively identify vulnerabilities.

Pham et al. [4] take the same position as the Aspect/Sonatype study, agreeing that recurring vulnerabilities in software are due to reuse. This reuse includes the same code base with an identical or very similar code structure, method calls and variables. Interestingly these attributes form the basis of a proposed method of detecting unreported vulnerabilities in one system by consulting knowledge of reported vulnerabilities in other systems that reuse the same code. This is included as part of the discussion of new detection methods in Section III.

Linus' Law [8] is often quoted in relation to OSS, which is "given enough eyeballs, all bugs are shallow", meaning with a large enough number of developers looking at code, errors can be found. However, this is a questionable claim from a scientific viewpoint, and an empirical study of Linus' Law by Meneely and Williams [9] appeared to show more collaboration meant more vulnerabilities. They found that files with changes from nine or more developers were sixteen times more likely to have a vulnerability than files changed by fewer than nine developers. The inherent collaborative nature of OSS unavoidably creates vulnerabilities that require addressing.

A review of existing relevant literature regarding conventional and newly researched detection methods was carried out [1-7, 9-15]. Section II handles the former, Section III the latter and then conclusions are presented with ideas for future work.

II. Conventional Methods of OSS vulnerability detection

This is a relatively new area of research so there is not an abundance of publications on methods of vulnerability detection in OSS. However, those that have been written thus far describe three conventional methods – static analysis (a black box[4] technique), dynamic analysis (a white box[5] technique) and code reviews (again, a white box technique).

1. Static Analysis

Many static analysis techniques and tools scan source code and detect vulnerabilities in software after it has been written, which encourages late detection and produces a lot of false positives[6]. In the literature reviewed for this paper, Sampaio & Garcia [6] were the only researchers that explicitly referenced the cut and thrust of the software development process, saying that external static tools for secure programming don't fit into such a workflow, since they don't work with the IDE and are retrospective. Zhang et al. [3] concur that static analysis produces high levels of false positives, as do Grieco et al. [12] and Perl et al. [10]. Shahmehri et al. [5] point out it is hard to know both which vulnerabilities a static analysis tool deals with, and to get assurance a tool is up-to-date.

Goseva-Popstojanova and Pehinschi [7] specifically wrote about the capability of static code analysis to detect vulnerabilities, concluding that tools are not effective. They tested three widely used commercial tools and found 27% of C/C++ vulnerabilities and 11% of Java vulnerabilities in their dataset were missed by all three. In some cases, they were comparable to or worse than random guessing. They too make the point about tools being prone to false positives, and this consolidates the need to find other methods of detection rather than rely solely on static analysis. That is not to say static analysis is of little use, as some compliance regulations require inventories of OSS components so that risks can be addressed. Static tools, such as Veracode Software Composition Analysis (SCA) [16], can scan open source code and create an inventory, so when a new vulnerability is disclosed, it is known which applications use the vulnerable OSS. Another example is the OWASP Dependency-Check tool [18] that analyses code and creates reports on associated CVE entries.

2. Dynamic Analysis

Dynamic analysis can also be called runtime analysis. Fuzzing is used here, where inputs are changed using random values to detect unwanted behavior [5]. Hafiz and Fang [11] researched the nuances of how vulnerabilities were discovered by reporters, and how those same reporters shared their findings with the OSS community. They found running a fuzzer and debugging was the chosen method for developers exploring binary executables to find buffer overflows. Vulnerability reporters tend to make their own fuzzing tools, seeing it as part of the learning process and preferring this approach over more systematic exploration methods.

[12] notes the usefulness of fuzzing, and that it needs only basic knowledge to undertake, however they also say fuzzing does not allow the control of program execution, large campaigns are needed for results, and it is time consuming. [3] contends fuzzing doesn't scale, if dynamic symbolic

[4] Black box testing is a software testing method in which the internal structure/ design/ implementation of the item being tested is not known.

[5] White box testing is a method of testing software that tests internal structures or workings of an application.

[6] A test result which wrongly indicates that a particular condition or attribute is present, i.e. a false alarm

execution[7] is used, as it explores code paths simultaneously which could create large workloads.

3. Code Reviews

These involve manual inspection of the source code. Consequently, this method requires a lot of human effort, a view shared by Perl et al. [10]. Working on source code manually does without question however detect vulnerabilities [11], and recall that [2] argued code reviews, conducted by someone with appropriate security knowledge, is in fact the only way to properly deal with vulnerabilities.

III. New Methods of OSS vulnerability detection

We have established the issues with the conventional methods in the main are that static analysis produces too many false positives, dynamic analysis doesn't scale, and code reviews are very time consuming. Research into new methods tries to address these problems via some interesting and novel approaches.

1. Distributed demand-driven security testing

Proposed by Zhang et al. [3], this involves many clients using OSS, and one main testing server. For this paper, a hub and spoke layout has been used for illustration, as per Figure 1. When a new path in a program is about to be exercised by user input, it is sent to the testing hub for security testing. Symbolic execution is applied to the execution trace to check potential vulnerabilities on this new path, and if one is detected then a signature is generated and updated back to all the clients for protection. If a path exercised by an input will trigger any vulnerability that has already been detected, the execution is terminated. This allows testing to focus on paths being used and stops attackers exploiting unreported vulnerabilities at a client site.

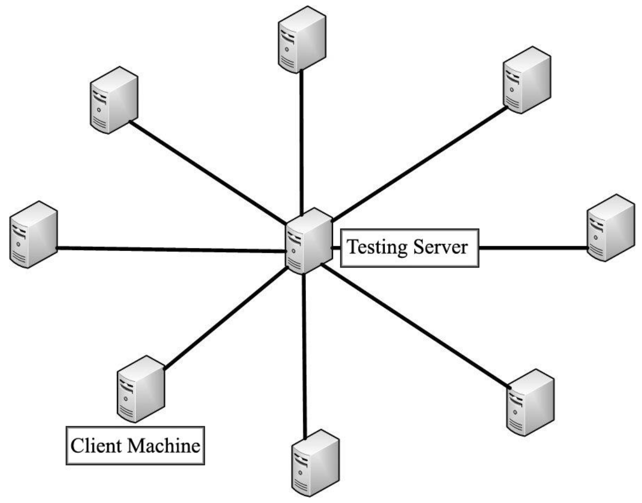


Figure 1 – Hub and spoke layout for distributed demand-driven security testing

[7] Symbolic execution uses symbolic values for variables instead of concrete values to execute all paths in a program.

However, questions remain over how to handle the large time and space overheads at the client sites, how sensitive data is transmitted and handled, and actual implementation details are scarce. That said, the principle of increasing test coverage of important paths as users exercise them is sound, and [3] offers a basic conclusion that machine learning can identify patterns of bugs at the testing server and use them to predict problematic code.

2. Use of Execution Complexity Metrics

Shin et al [13] examined complexity metrics collected during code execution, considering them potential indicators of vulnerable code locations. Table 1 describes these metrics. They measure the frequency of function calls and duration of execution functions. Firefox and Wireshark were studied using Callgrind[8] to gather the metrics and the results showed these execution complexity metrics may be better indicators of vulnerable code than the conventional static complexity metric, Lines of Code (LoC).

Name	**Definition**
NumCalls	The number of calls to the functions defined in a file.
InclusiveExeTime	Execution time for the set of functions, S, defined in a file including all the execution time spent by the functions called directly or indirectly by the functions in S.
ExclusiveExeTime	Execution time for the set of functions, S, defined in a file excluding the execution time spent by the functions called by the functions in S.

Table 1 – Execution complexity metrics defined in [13]

The initial results, shown in Table 2, indicate the percentage of vulnerable files in execution is higher than the percentage of vulnerable files in total, and hence execution complexity metrics could be good indicators of vulnerability. This can reduce the code inspection effort as prioritisation can take place based on the metrics.

Program	**% of vulnerable files**	**% of vulnerable files in executed files**
Firefox	3.8%	11%
Wireshark	7.8%	19%

Table 2 – Execution statistics from [13]

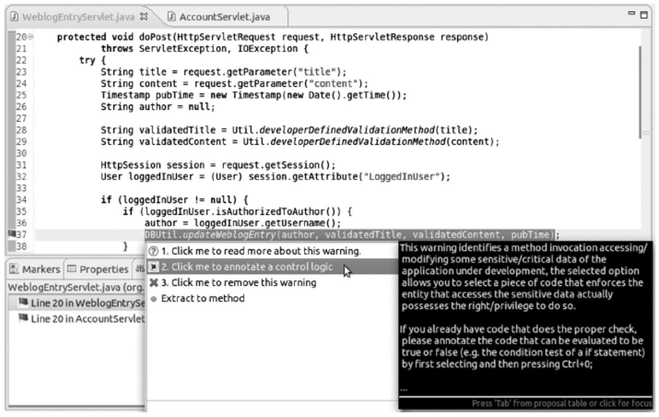
[8] Callgrind is a Valgrind tool for profiling programs. The collected data consists of the number of instructions executed on a run, their relationship to source lines, and call relationship among functions together with call counts.

4. Integrated Development Environment (IDE) Plugins for Early Detection

Sampaio & Garcia [6] attempted to detect vulnerabilities earlier in the development process by using an Eclipse Java plug-in, arguing developers should be aware of security vulnerabilities as they are coding. To reduce false positives, they proposed context-sensitive data flow analysis which uses a program's context of variables and methods when searching for vulnerabilities instead of pattern matching,

Zhu et al. [14] present interactive static analysis, also known as IDE static analysis. They too developed an Eclipse Java plug-in for detecting code patterns that gives a two-way interaction between the IDE and the developer. According to [14], their tool detected multiple zero day vulnerabilities.

Figure 2 shows a screenshot of this tool where the developer is instructed to annotate access control logic for a highlighted sensitive method call.



```
    protected void doPost(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {
        try {
            String title = request.getParameter("title");
            String content = request.getParameter("content");
            Timestamp pubTime = new Timestamp(new Date().getTime());
            String author = null;

            String validatedTitle = Util.developerDefinedValidationMethod(title);
            String validatedContent = Util.developerDefinedValidationMethod(content);

            HttpSession session = request.getSession();
            User loggedInUser = (User) session.getAttribute("LoggedInUser");

            if (loggedInUser != null) {
                if (loggedInUser.isAuthorizedToAuthor()) {
                    author = loggedInUser.getUsername();
                    DBUtil.updateWeblogEntry(author, validatedTitle, validatedContent, pubTime);
            }
```

Markers
Properties
WeblogEntryServlet.java (org.
Line 20 in WeblogEntrySe
Line 20 in AccountServlet
1. Click me to read more about this warning.
2. Click me to annotate a control logic
3. Click me to remove this warning
Extract to method

This warning identifies a method invocation accessing/ modifying some sensitive/critical data of the application under development, the selected option allows you to select a piece of code that enforces the entity that accesses the sensitive data actually possesses the right/privilege to do so.

If you already have code that does the proper check, please annotate the code that can be evaluated to be true or false (e.g. the condition test of a if statement) by first selecting and then pressing Ctrl+0;

...

Figure 2 – a screenshot from an IDE static analysis tool developed in [14]

5. Machine Learning

Most OSS code is managed using version control systems like Git or CVS, with vulnerable code inserted via commits from the developer to the main data repository. But most tools can't run on a small code snippet in an individual commit, and checking the whole project is time consuming. Perl et al. [10] implemented a type of machine learning algorithm[9] called a Support Vector Machine (SVM) that used metadata[10] from commits made to OSS repositories.

The SVM used features from the metadata such as the number of added, deleted or modified functions and how often a contributor had contributed to a given project before. Their results showed that false positives were reduced by over 99% compared to those generated by a static analysis tool - to be exact, their SVM driven tool generated 36 false positives compared to 5460 generated from the static analysis tool. The goal of their work was to reduce the chance of vulnerabilities getting from a vulnerable commit into the fully deployed software.

[12] also developed a machine learning tool to predict vulnerabilities for large scale software like operating systems. They took the popular Debian OS as an example, since it has 30,000 programs and 80,000 bug reports. Clearly, code flaws can be hard to find manually in a code base of that size, so the application of machine learning is of interest. Their classification results were not conclusive but nevertheless, as an initial study, they showed promise for large-scale vulnerability detection only using binary executables, an approach which does not appear to have been attempted elsewhere.

6. Further Knowledge Formalisation and Linking Repositories

Algahtani et al. [15] discussed formalising knowledge representation to determine transitive dependencies in software. The idea is the various vulnerability repositories that exist online like the NIST National Vulnerability Database, or the Common Weakness Enumeration (CWE) database can be linked and simultaneously used to find out if a project is indirectly dependent on vulnerable components.

IV. Conclusions & Future work

The global use of OSS presents such a huge number of attack vectors that discovering novel techniques of vulnerability detection is an essential area of research. Of the new methods mentioned in this paper, it is the opinion of the author that machine learning, early detection IDE plug-ins and linking repositories show much promise for future work. Machine learning lends itself well to feature-rich OSS which speeds up classification of vulnerable code and reduces the time burden on development teams. Early detection IDE plug-ins will help developers implementing OSS to grow and consolidate their secure coding knowledge. Linking repositories ensures better value from the separate, unconnected datastores of vulnerabilities as they presently exist.

Improvements in OSS vulnerability detection may be quicker to realise than one would think – English et al. [17] mention Pareto's law, where 80% of effects can be contributed to 20% of causes, and so identifying a small proportion of problematic OSS code then focusing testing efforts using a selection of detection methods could improve code quality and time-to-release, whilst reducing development and maintenance costs. The exact mix of techniques will vary from one OSS scenario to another but the conclusion this paper draws is that the very existence of a strategy that uses a blend of methods that augment each other is likely to be of significantly more benefit than using just one approach in isolation.

Acknowledgement

This paper was supported by the Centre for Secure Information Technologies (CSIT), at the Queen's University of Belfast.

[9] Machine learning is a type of artificial intelligence where computers use algorithms to learn iteratively, teaching themselves to recognise patterns.

[10] Metadata is a set of data that describes and gives information about other data.

References

[1] S. Koussa, “13 tools for checking the security risk of open-source dependencies”, TechBeacon, May 2016, https://techbeacon.com/13-tools-checking-security-risk-open-source-dependencies-0, accessed 20/3/17

[2] J. Williams, A. Dabirsiaghi, “The Unfortunate Reality of Insecure Libraries”, Aspect Security, March 2012, http://cdn2.hubspot.net/hub/315719/file-1988689661-pdf/download-files/The_Unfortunate_Reality_of_Insecure_Libraries.pdf?t=1490125724196, accessed 20/3/17

[3] D. Zhang et al, “A distributed framework for demand-driven software vulnerability detection”, The Journal of Systems and Software 87, pgs 60-73, Elsevier, 2014

[4] N. Pham et al, “Detection of recurring software vulnerabilities”, ASE ’10, September 20-24, 2010, Antwerp, Belgium

[5] N. Shahmehri et al., “An advanced approach for modeling and detecting software vulnerabilities”, Information and Software Technology 54, pgs 997-1013, Elsevier, 2012

[6] L. Sampaio, A. Garcia, “Exploring context-sensitive data flow analysis for early vulnerability detection”, The Journal of Systems and Software 113, pgs 337-361, Elsevier, 2016

[7] K. Goseva-Popstojanova, A. Perhinschi, “On the capability of static code analysis to detect security vulnerabilities”, Information and Software Technology 68, pgs 18-33, Elsevier, 2015

[8] Linus’ Law, https://en.wikipedia.org/wiki/Linus%27s_Law, accessed 20/3/17

[9] A. Meneely, L. Williams, “Secure Open Source Collaboration: An Empirical Study of Linus’ Law”, CCS ’09, November 9-13, 2009, Chicago, Illinois, USA.

[10] H. Perl et al., “VCCFinder: Finding Potential Vulnerabilities in Open-Source Projects to Assist Code Audits”, CCS ’15, October 12-16, 2015, Denver, Colorado, USA

[11] M. Hafiz, M. Fang, “Game of detections: how are security vulnerabilities discovered in the wild?”, Empir Software Eng (2016) 21:1920-1959, Springer Science+Business Media New York

[12] G. Grieco et al., “Toward Large-Scale Vulnerability Discover using Machine Learning”, CODASPY ’16, March 9-11, 2016, New Orleans, LA, USA.

[13] Y. Shin, L. Williams, “An Initial Study on the Use of Execution Complexity Metrics as Indicators of Software Vulnerabilities”, SESS ’11, May 22, 2011, Waikiki, Honolulu, HI, USA

[14] J. Zhu et al., “Supporting secure programming in web applications through interactive static analysis”, Journal of Advanced Research (2014) 5, pgs 449-462, Elsevier, 2014

[15] S. Alqahtani et al., “Tracing known security vulnerabilities in software repositories – A Semantic Web enabled modeling approach”, Science of Computer Programming 121 (2016) pgs 153-175, Elsevier, 2016

[16] Veracode Software Composition Analysis, https://www.veracode.com/products/software-composition-analysis, accessed 20/3/17

[17] M. English et al., “Fault Detection and Prediction in an Open-Source Software Project”, ACM 2009

[18] OWASP Dependency Check, https://www.owasp.org/index.php/OWASP_Dependency_Check, accessed 20/3/17

[19] Black Duck Software, “78% of Companies Run on Open Source Yet Lack Formal Policies”, https://www.blackducksoftware.com/de/node/1011, 16th April 2015, accessed 20/3/17