Collaborative Security Code-Review

Towards Aiding Developers Ensure Software-Security

Hala Assal[*], Jeff Wilson, Sonia Chiasson, and Robert Biddle
School of Computer Science
Carleton University
Ottawa, Canada

1. INTRODUCTION

Humans make mistakes, and software programmers are no exception. Software vulnerabilities are discovered everyday; close to 8,000 vulnerabilities were reported in 2014, and almost 2,500 were reported in the first four months of 2015 [9]. Microsoft Security Response Centre defines software vulnerabilities as *a security exposure that results from a product weakness that the product developer did not intend to introduce and should fix once it is discovered* [8].

Integrating security in the Software Development Lifecycle (SDLC) leads to better quality software than when security was considered as an additional task [11]. Major software companies are taking the initiative to integrate security in the SDLC, starting from the early stages of the development. For example, Microsoft has been following a security-oriented software development process since 2004. The Microsoft Security Development Lifecycle (SDL) introduces security early in the development process and throughout the different stages of the traditional SDLC [7]. Google, on the other hand, has an independent *Security Team* responsible for aiding security reviews during the design and implementation phases, as well as providing ongoing consultation on project-relevant security risks and their possible remedies.

Static analysis [12] is a method of software testing that can be performed throughout the different stages of the development to ensure software is free of vulnerabilities introduced to the code due to programming errors. Static analysis does not require the code to be executed, thus incomplete versions of the software can be tested. This allows testing software during early stages when errors are less expensive to fix [3, 1]. Static-code Analysis Tools (SATs) are tools that automatically analyze static-code to uncover vulnerabilities.

2. Static-code Analysis Tools (SATs)

Despite their benefits, SATs are not widely accepted by the Software Engineering community. Some reasons for the underuse of SATs are [5, 6]: *(i) The quality of tool output.* A SAT produces false positives when it mistakenly reports a vulnerability. With large projects, SATs produce a large volume of warnings that could reach thousands. Thus, with many false positives and up to thousands of potential vulnerabilities, it is sometimes inefficient for developers to use these tools. *(ii) Support for collaboration.* Developers are reluctant to use SATs because they do not adequately support collaboration between team members. Even though some tools allow developers to exchange vulnerability reports, they usually takes the developer out of the development environment, and thus out of context, discouraging developers from using this feature, and subsequently the tool altogether. *(iii) Tool customization.* Many tools require complicated steps to be customized, and yet they do not fulfill developers' needs. Developers believe the quality of tool output could be improved (e.g., showing less false positives) if they were able to customize it to look for vulnerabilities they care about the most. *(iv) Result understandability.* SAT fail to explain their reasons for triggering a warning; they usually do not clearly explain what the problem is, why it is considered a problem, and how it could be fixed. Without obvious reasoning about why a warning was issued, developers may not be able to develop trust for the analysis tool, thus lowering the likelihood of using it for vulnerability-detection. *(v) Actionable tool output.* Code suggestions and quick fixes are two features missing from most current SATs. Developers expect to be guided to fix a detected vulnerability. However, further research is needed in this regards, as developers might wrongfully accept a quick fix to a false positive, without thoroughly inspecting the code and the vulnerability report, potentially leading to more problems.

In this poster we aim to shed light on specific usability issues of tools used by developers to uncover security vulnerabilities in source code (e.g., SATs); a topic that has not been thoroughly studied in the usable security community. We also introduce a prototype to support what we call *collaborative security code-review (CSCR)*, where developers/testers collaborate to ensure the software under review is free from security vulnerabilities.

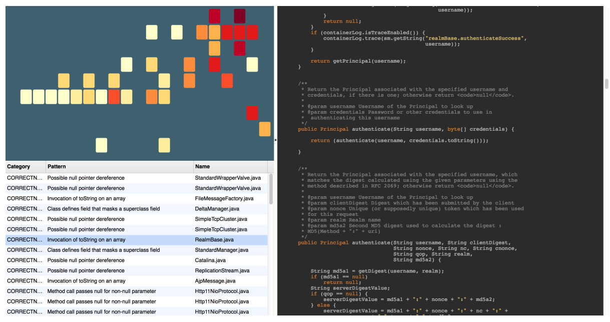
3. COGNITIVE WALKTHROUGH OF FINDBUGS

We began by surveying the different available tools that perform static-analysis for Java code [10], such as CodeSonar,[1] klocwork,[2] Coverity-SAVE,[3] and Findbugs.[4] For the purpose of our work, we chose to use Findbugs as it is an open source tool, is one of the tools used in Microsoft SDL, and has been one of the most used tools in similar research projects [5, 2, 13].

Next, we conducted a Cognitive Walkthrough of Findbugs v.2.0 with a group of six evaluators who are experts in the fields of security and usable security. The session lasted 90

[*]Corresponding author: HalaAssal@scs.carleton.ca

[1]`http://www.grammatech.com/codesonar`
[2]`http://www.klocwork.com`
[3]`http://www.coverity.com/products/coverity-save/`
[4]`http://findbugs.sourceforge.net`



Category	Pattern	Name
CORRECTN...	Possible null pointer dereference	StandardWrapperValve.java
CORRECTN...	Possible null pointer dereference	StandardWrapperValve.java
CORRECTN...	Invocation of toString on an array	FileMessageFactory.java
CORRECTN...	Class defines field that masks a superclass field	DeltaManager.java
CORRECTN...	Possible null pointer dereference	SimpleTcpCluster.java
CORRECTN...	Possible null pointer dereference	SimpleTcpCluster.java
CORRECTN...	Invocation of toString on an array	RealmBase.java
CORRECTN...	Class defines field that masks a superclass field	StandardManager.java
CORRECTN...	Possible null pointer dereference	Catalina.java
CORRECTN...	Possible null pointer dereference	ReplicationStream.java
CORRECTN...	Invocation of toString on an array	AjpMessage.java
CORRECTN...	Method call passes null for non-null parameter	Http11NioProtocol.java
CORRECTN...	Method call passes null for non-null parameter	Http11NioProtocol.java
CORRECTN...	Method call passes null for non-null parameter	Http11NioProtocol.java

```
                                               username));
            }
            return null;
        }
        if (containerLog.isTraceEnabled()) {
            containerLog.trace(sm.getString("realmBase.authenticateSuccess",
                                          username));
        }

        return getPrincipal(username);
    }


    /**
     * Return the Principal associated with the specified username and
     * credentials, if there is one; otherwise return <code>null</code>.
     *
     * @param username Username of the Principal to look up
     * @param credentials Password or other credentials to use in
     *  authenticating this username
     */
    public Principal authenticate(String username, byte[] credentials) {

        return (authenticate(username, credentials.toString()));

    }


    /**
     * Return the Principal associated with the specified username, which
     * matches the digest calculated using the given parameters using the
     * method described in RFC 2069; otherwise return <code>null</code>.
     *
     * @param username Username of the Principal to look up
     * @param clientDigest Digest which has been submitted by the client
     * @param nonce Unique (or supposedly unique) token which has been used
     * for this request
     * @param realm Realm name
     * @param md5a2 Second MD5 digest used to calculate the digest :
     * MD5(Method + ":" + uri)
     */
    public Principal authenticate(String username, String clientDigest,
                                  String nonce, String nc, String cnonce,
                                  String qop, String realm,
                                  String md5a2) {

        String md5a1 = getDigest(username, realm);
        if (md5a1 == null)
            return null;
        String serverDigestValue;
        if (qop == null) {
            serverDigestValue = md5a1 + ":" + nonce + ":" + md5a2;
        } else {
            serverDigestValue = md5a1 + ":" + nonce + ":" + nc + ":" +
```

Figure 1: A screenshot of the CSCR prototype

minutes and was voice recorded. Two of the evaluators also took notes during the session. The session started with running Findbugs analysis on the source code of Apache Tomcat v.6.0.41.[5] Next, the evaluators explored the tool's interface and its warnings of potential vulnerabilities. The evaluators then focused on some warnings and worked towards classifying them as false positives or true vulnerabilities. Finally, the evaluators discussed different features they would prefer to be available in SATs.

We concluded, in accordance with previous research, that Findbugs does not adequately support collaboration. For instance, reviewers cannot easily share their reviews. We also observed, on multiple occasions, that reviewers were trying to direct each other's attention to areas in the code by pointing to the code and drawing imaginary annotations on the screen. In addition, we noticed that reviewers were absorbed in classifying individual vulnerability warnings, rather than assessing the overall security of the software and focusing on the bigger picture.

4. CSCR PROTOTYPE

Figure 1 shows a screenshot of the CSCR prototype that was built using React[6] and NODE-RED[7] Javascript frameworks. The top left pane shows a visual representation of detected vulnerabilities. Cell(i, j), shows the number of detected vulnerabilities that belong to category i and have rank j.[8] Tapping on a cell populates the bottom left pane with a list of vulnerabilities belonging to the chosen category and rank. Tapping on one of the listed vulnerabilities, shows the source code where this vulnerability appears in the right pane. Through the CSCR prototype we aim to leverage the benefits of SATs, while overcoming their shortcomings. In particular we aim to:

Support collaboration in code reviews. The CSCR prototype uses large multitouch displays as an interface. Using large multitouch displays for collaborative work is a research area that, although in its infancy, shows promising results in supporting and promoting collaboration between team members. Moreover, the prototype implements collaborative gestures to leverage their intuitiveness, and test their potential to encourage collaboration and aid in effective communication and knowledge flow within the team.

Support analysis. Visual analysis environments effectively focus developers' attention to the most critical vulnerabilities in source code[4]. The CSCR prototype uses interactive information visualizations to allow developers to get an overview of the system's security, with the ability to filter vulnerability categories, zoom into source code files, and inspect details of vulnerabilities. With such visual analysis support, we hypothesize that developers will be able to prioritize vulnerabilities, focus on the most important ones, and discover hidden patterns. In addition, we aim to investigate the effectiveness of suggesting known vulnerability fixes.

Ensure seamless integration with the SDLC. The CSCR aims to support the natural workflow for developers while encouraging collaboration at key points. We will investigate different functionalities, such as enabling developers to annotate source code, save and reload these annotations, write vulnerability reviews for a single or multiple vulnerabilities, link relevant documentation to reviews, assign vulnerabilities to developers to fix, and auto-generate review documentation based on the aforementioned functionalities.

[5]`https://tomcat.apache.org/download-60.cgi`
[6]`https://facebook.github.io/react/`
[7]`http://nodered.org`
[8]The explanation of "category" and "rank" can be found on `http://findbugs.sourceforge.net/findbugs2.html`

5. REFERENCES

[1] P. Anderson. Measuring the value of static-analysis tool deployments. *Security Privacy, IEEE*, 10(3):40–47, May 2012.

[2] N. Ayewah and W. Pugh. The Google FindBugs Fixit. In *Int. Symp. on Software Testing and Analysis*, ISSTA, 2010. ACM.

[3] B. Chess and G. McGraw. Static analysis for security. *IEEE Security & Privacy*, 2(6):76–79, 2004.

[4] J. R. Goodall, H. Radwan, and L. Halseth. Visual analysis of code security. In *Int. Symp. on Visualization for Cyber Security*, VizSec, 2010. ACM.

[5] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge. Why don't software developers use static analysis tools to find bugs? In *Int. Conf. on Software Engineering (ICSE)*, 2013.

[6] C. Lewis, Z. Lin, C. Sadowski, X. Zhu, R. Ou, and E. Whitehead. Does bug prediction support human developers? Findings from a Google case study. In *Int. Conf. on Software Engineering (ICSE)*, 2013.

[7] Microsoft Corp. Microsoft Security Development Lifecycle (SDL) – process guidance. `https://msdn.microsoft.com/en-us/library/windows/desktop/84aed186-1d75-4366-8e61-8d258746bopq.aspx`, 2012. [Accessed May-2015].

[8] Microsoft Corp. Definition of a security vulnerability. `https://msdn.microsoft.com/en-us/library/cc751383.aspx`, 2015. [Accessed May-2015].

[9] National Vulnerabiliy Database. NVD statistics results. `https://web.nvd.nist.gov/view/vuln/statistics`. [Accessed May-2015].

[10] V. Okun, Delaitre, Aurelien, and P. E. Black. Report on the Static Analysis Tool Exposition (SATE) IV. *NIST Special Publication 500-297*. National Institute of Standards and Technology, 2013.

[11] A. Shostack and C. Wysopal. *Threat modeling: designing for security*. John Wiley & Sons, 2014.

[12] I. Sommerville. *Software Engineering*. Pearson Education, 9 edition, November 2011.

[13] H. Zhong and Z. Su. An empirical study on real bug fixes. In *Int. Conf. on Software Engineering (ICSE)*, 2015.