Vulnerability Recognition by Execution Trace Differentiation

Fabien Patrick Viertel, Oliver Karras, Kurt Schneider
Email: {fabien.viertel, oliver.karras, kurt.schneider} @inf.uni-hannover.de
Software Engineering Group
Leibniz Universität Hannover
30167 Hannover, Germany

Abstract

In context of security, one of the major problems for software development is the difficult and time-consuming task to find and fix known vulnerabilities through the vulnerability documentation resulting out of a penetration test. This documentation contains for example the location and description of found vulnerabilities. To be able to find and fix a vulnerability, developers have to check this documentation.

We developed a tool-based semi-automated analysis approach to locate and fix security issues by recorded execution traces. For identifying the affected source code snippets in the project code, we determine the difference between a regular and a malicious execution trace. This difference is an indicator for a potential vulnerability. As case study for this analysis we use vulnerabilities, which enable remote code execution. We implemented this approach in a software prototype named FOCUS+. This tool visualizes the traces and differences by several views such as a method call graph view. All views facilitate direct access to affected code snippets and point to the possible vulnerabilities. Thus, identified security gaps can immediately be fixed in FOCUS+.

Keywords

Code Exploit, Execution Trace, Vulnerability Analysis

1 Introduction

In software development, developers have to take care of requirements for a software system. In general, there are functional and non-functional requirements. As increasingly important part of the non-functional requirements, there is security. In context of remote code execution, a security incident could be defined as deviation of the regular behaviour of software. For detecting a security incident, the regular behaviour of a system is always a benchmark. Software execution traces describes the runtime behaviour of an individual program session. Nevertheless, in the Security Developement Life Cylce of Microsoft it is defined as a difficult and time-consuming task to fix security bugs [1]. For fixing a bug, the identification of the vulnerability location in the application source code is necessary.

A possible consequence of a vulnerability is the malicious use in form of remote code exploits. We define a remote code exploit as added program code, which is not part of the regular application source code via using a vulnerability of the application.

Security experts are responsible for identifying the occurrence of vulnerabilities and their location. Often security experts are hired for fulfilling a penetration test and usually they are not available for the whole project lifecycle. It is possible that they are not available anymore, when their detected vulnerabilities will be fixed [2]. Month to years after their detection, the insecure code will be fixed [3]. Therefore, their findings have to be well documented, such that developers can fix the existing vulnerabilities without directly speaking with a security expert.

In our previous work, we have developed a prototype called FOCUS using execution traces such as video screencasts enriched by audio comments for vulnerability documentation [4]. With FOCUS a security expert can record realised penetration tests.

This paper describes the enrichment of the tool FOCUS with mechanisms for supporting the analysis and fixing of found vulnerabilities.We have developed an approach for automatically identifying the location of vulnerabilities out of the code execution recordings of a penetration test. For this purpose, we focus on remote code execution detection. For supporting execution trace analysis, different views are added. To enable the reuse of recorded trace files in further applications, a new tracing mechanism and trace file format is used. The enhancement of our tool is named FOCUS+.

The remainder of this paper is structured as follows. In Section 2 related work of source code analysis is described and we emphasize the difference to our approach. The approach and further FOCUS enhancements will be described in section 3. Section 4 concludes the work of this paper.

2 Related Work

Ernst et al. [5] developed the Daikon system which can run other programs to detect program invariants. The Daikon system observes the computed values of a program and reports properties that were true over the observed executions. The found invariants are useful for program understanding. The results of dynamic detection of likely invariants can be used for multiple purposes such as test case generation, prediction of incompatibilities in component integration, repair of inconsistent data structures.

Maletic and Collard [6] presented an approach, so called meta-differencing, to conduct analysis of source code differences. They automatically extract syntactic information about source code changes by applying an underlying source code view. This information can be queried and searched to extract specifics of an change to identify new added methods and comments or preprocessor changes.

Buckner et al. [7] support program comprehension during incremental changes with a software tool called JRipples. This tool provides programmers with organizational support to make an incremental change process easier and more systematic. JRipples analyzes a program to keep track of inconsistencies. Based on these traces, developers obtain support in impact analysis and change propagation, which are the two most difficult activities of incremental change.

All of the named related work offers execution traces or source code analysis for different purposes. No of them considers the support for fixing and localizing vulnerabilities in source code.

3 Approach

First of all, in FOCUS+ the recording of execution traces for an application session will be done through the Kieker performance monitoring framework [8]. In comparison to other log file generation approaches, Kieker works with aspect weaving, which makes it possible to record traces without manipulating the application source code.

To analyse vulnerabilities, we compare the trace of a recorded penetration session to the base trace including the regular application behaviour. The trace of the penetration contains a recorded code exploit for the viewed application, performed by a security expert. For identifying vulnerable methods, we look for new and missing method executions in comparison of penetration and base trace. If a method call is missing or additionally added into the malicious trace, a potential vulnerable code snippet is found. The last method call which is in the regular execution trace as well as in the penetration trace will be considered to eliminate this vulnerability. After this method call the application sequence has changed. This is properly the method which enables the execution of remote code. The method will be highlighted to the developer. The parameters of method calls and other entries of variables are disregarded. For the analysis, only the existence as well as sequence of method call execution will be considered. All information that can be extracted out of the results for the differentiation of two execution traces will be summarized in the following.

Table 1: Methods of example application scope

Method	Description
getLogin	Gets the entered login credentials
checkLogin	Correctness check of login
login	Proceeds the login
skipCheck	Skips the login credentials check

- **Equal method execution** – Method execution is in both traces
- **New method execution** – Method execution is only in the trace of malicious use
- **Missing method execution** – Method execution is only in the trace of regular application behaviour
- **Called by method** – By which method the executed method is called

For example, we consider a login process of an application implementing the methods listed in Table 1. Regularly the system should get the entered login credentials, check their correctness, and proceed the login if they are correct. Otherwise, the login request should be denied. This behaviour is described via the method execution sequence of *getLogin*, *checkLogin* and *login*. We assume that there is a vulnerability inside of the *getLogin* method, which enables remote code execution. The check for the correct login credentials could be skipped such that the method *skipCheck* is executed instead of *checkLogin*. The *skipCheck* method is exploited code that is not part of the regular project source code. FOCUS+ detects this differentiation and displays the results to the developer. These results includes the previous mentioned information about the method executions.

For visualisation, there are two different views with similar functionality. At first, there is the execution graph view shown in Figure 1. The displayed call graph is the simplified representation of the method call graph view included in FOCUS+. It contains the graphical representation of two application sessions in one graph. The second view displayed in Table 2, is a side-by-side list view containing two lists. Each list represents an application session. Both of these views give an overview of the trace differentiation of the two application sessions. Each node in the graph view and every row in the list view symbolizes a method call, which is at least part of one trace file. For every method call, the views contain the information

Table 2: List view - differentiation of two trace files

State	Method	Method	State
Equals	getLogin	getLogin	Equals
Missing	checkLogin	skipCheck	Exploit
Equals	login	login	Equals

about the last correct method call and the state of method execution. The state differs in potential exploit code, missing method calls or equal occurrence in both traces. Furthermore, it is possible to directly access the source code of these method calls.

Comparing these views, they distinguish in depiction of their contents. The graph based view, overviews two application sessions in one common view. For large application sessions, the contained information could overwhelm the developer. The list view is a structured, time-ordered and scrollable list, which prevents the information overwhelming. Already for small application sessions, it is not possible to overwiew all method calls in one screen.

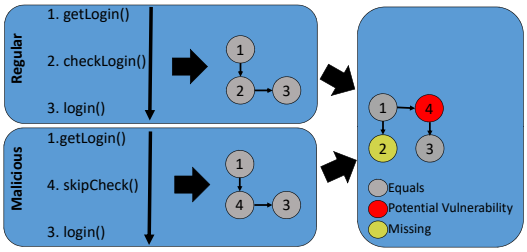

Figure 1: Graphview - differentiation of two trace files

So far, our vulnerability recognition approach is limited to a single system. If an application uses further external systems, which execute other program languages like a database system, remote code exploits for that language cannot be detected. This is reasoned to the instrumentation of the trace mechanism. To applying our approach, a deterministic code execution is required.

4 Conclusion

In this paper, we describe the enhancement of our tool-based documentation approach by supporting methods to analyse previously found vulnerabilities. This analysis works on execution traces of different application sessions. The traces will be recorded by our tool FOCUS+ as a by-product. If there is a difference between two application session traces of the same application, FOCUS+ will recognize that difference as potential vulnerability of the regarded application. For analysing vulnerabilities of a specific application scope, a record of the regular execution and a record of a penetration test containing remote code execution is necessary. Furthermore, it is possible to jump directly to the potential insecure source code snippets, to fix these vulnerabilities.

For future steps, we plan to create a user study to identify the advantages of FOCUS+ in comparison to regular vulnerability reports created by security experts. The focus will be on potential timesaving in finding the vulnerable code parts as well as the clarity of documentation for vulnerabilities.

References

[1] Michael Howard, Steve Lipner, The security development lifecycle: SDL: a process for developing demonstrably more secure software, Microsoft Press, page 13, 2006.

[2] Siv Hilde Houmb, Shareeful Islam, Eric Knauss, Jan Jürjens, and Kurt Schneider, Eliciting security requirements and tracing them to design: An integration of Common Criteria, heuristics, and UMLsec, Requirements Engineering, vol. 15, no. 1, page 63-93, 2009.

[3] P. Bhattacharya, L. Ulanova, I. Neamtiu, S. C.Koduru, Prototypes As Assets, An Empirical Analysis of Bug Reports and Bug Fixing in Open Source Android Apps, 2013 17th European Conference on Software Maintenance and Reengineering (CSMR), IEEE Computer Society, page 133-143, 2013.

[4] Kurt Schneider, Prototype as Assets, not Toys: Why and How to Extract Knowledge from Prototypes, Proceedings of the 18th International Conference on Software Engineering, IEEE Computer Society, page 522-531, 1996.

[5] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao: The Daikon system for dynamic detection of likely invariants, Science of Computer Programming, vol.69, no. 1, page 35-45, 2007.

[6] Jonathan I. Maletic and Michael L. Collard: Supporting source code difference analysis, 20th IEEE International Conference on Software Maintenance, IEEE, page 210-219, 2004.

[7] Jonathan Buckner, Joseph Buchta, Maksym Petrenko, and Vaclav Rajlich: JRipples: A tool for program comprehension during incremental change, 13th International Workshop on Program Comprehension (IWPC 2005), IEEE, page 149-152, 2005.

[8] Andre Van Hoorn, Jan Waller, Wilhelm Hasselbring, Kieker: A Framework for Application Performance Monitoring and Dynamic Software Analysis, Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering, page 247-248, 2006.