STATIC TAINT ANALYSIS FOR DETECTING SECURITY VULNERABILITIES

Jakub Říha
Bachelor Degree Programme (3), FIT BUT
E-mail: xrihaj03@stud.fit.vutbr.cz

Supervised by: Tomáš Vojnar
E-mail: vojnar@fit.vutbr.cz

Abstract: Static taint analysis uses data flow information to track potentially distrusted and tainted data values. This technique plays a vital part during security review processes aimed at mitigating security vulnerabilities of web-based applications.

This document is intended to describe theoretical foundations as well as the construction of such taint analyser based on the .NET Framework and the analysis services provided by the Roslyn compiler.

Keywords: taint analysis, security review, Roslyn compiler, .NET Framework

1 INTRODUCTION

Static code analysis examines properties of computer software based on its source code without actually executing it. This sort of analysis can be applied in many areas including compiler optimizations, bug hunting, and security reviews [1]. In the area of security, automated static analysis can be more effective in detecting security defects than manual code review which is a costly and error-prone process.

2 MOTIVATION

Taint code analysis can be seen as an important factor in detecting security vulnerabilities in modern web-based applications [2]. This sort of analysis can be successfully applied on data flow security risks based on invalid data injection. Examples of these are SQL injection and Cross-site scripting which are prevalent attacks on software systems and can pose significant threats [3].

A taint analyser can be viewed as a tool suitable at detecting such risks. The analyser checks all possible data flow paths in program executions marking those which can be potentially influenced by an outside input. If any of these tainted variables take part in command execution prior to their sanitization, the analyser warns the developer. The program can then be redesigned to sanitize the input and therefore suppress a possible future security breach.

3 IMPLEMENTATION

As noted above, taint analysis, a form of information-flow analysis, establishes whether values from untrusted methods and parameters may flow into security-sensitive operations [2]. This use of data flow information to determine possible paths which an attacker can misuse is called *taint propagation*.

The goal of this work is to construct such a static taint analyser that detects SQL injection vulnerabilities. A program that has an exploitable SQL injection vulnerability contains a data flow path from user supplied inputs to a method parsing SQL commands. The task of the analyser is to find such a vulnerable path. This section describes our implementation of the analyser.

3.1 GATHERING COMPILER INFORMATION

In order to perform analysis, the analyser uses syntactic and semantic information about the program code being analysed. In our case, this information is obtained through the APIs offered by the Roslyn compiler. Project Roslyn is a Microsoft developed set of APIs for exposing C# and Visual Basic .Net compilers as services [4].

In particular, the Roslyn compiler exposes abstract syntax trees (AST) as a representation of a program code structure. Moreover, the APIs contain a semantic service which allows one to gather semantic information about individual nodes of ASTs.

3.2 PROGRAM FLOW REPRESENTATION

Our implementation of the analyser works on a flow representation of program code called a *control flow graph*. It is a directed graph that represents paths that a program can traverse during its execution. The analyser uses an *adjacency-list* representation of the control flow graph which is suitable for a sparse graph representation [5]. Each control flow graph vertex is called a *basic block* and contains one or more *flow rules*.

The flow rules, determine how the program code handles concrete data flow values during a program execution. The rules are created from AST nodes which are relevant to taint analysis. These rules can be divided as follows [2]:

- **Source** rules define program locations where tainted data enter the system. This location may be a system boundary such as an HTTP data delivering method in case of web applications.
- **Sink** rules define program locations that should not receive tainted data (for example, a method that interprets SQL commands).
- **Pass-through** rules define program locations where tainted data are propagated further. This can be the case of variable assignments. The left side variable of an assignment expression is tainted by a right side tainted variable.
- **Cleanse** rules are similar to pass-through rules except they clean the taint mark of a variable. These are all input validation methods. For example, a method that escapes all control characters in HTML code.
- **Call** and **return** rules define program locations where methods are called/returned.

Control flow graphs of individual methods are created from ASTs which are provided by the Roslyn compiler API. The AST is transformed into a control flow graph by recursive deepening.

3.3 TRAVERSING CONTROL FLOW GRAPH

The task of a taint analyser is to traverse through the control flow graph visiting basic blocks and applying flow rules on the current data context. Our implementation of the analyser uses a *worklist algorithm* [6] which follows the order of data flow value changes. The worklist algorithm traverses control flow graphs in a reverse post-order fashion, that is, a node is visited before any of its successors has been visited.

3.4 TAINT DATA CONTEXT

The analyser keeps the context of tainted variables during the graph traversal. This context is formally represented as a set of tainted variables. Flow rules trigger set operations and thus modify the data context. This method of keeping the data context is called *bit vector data flow analysis* [6].

The flow equations for our taint analysis are as follows:

$$In_n = \begin{cases} BI & n \text{ is } Start \text{ block} \\ \bigcup_{p \in pred(n)} Out_p & \text{otherwise} \end{cases} \quad (1)$$

$$Out_n = f_n(In_n) \quad (2)$$

In the above, n is the current basic block being traversed and $pred(n)$ are predecessor blocks of n. BI is the information that is available at the $Start$ block of the control flow graph. In_n and Out_n are data flow variables whose values are being defined by the data flow equations. The flow function f_n is the transformation effected by the basic block n on data flow values.

3.5 INTERPROCEDURAL ANALYSIS

To increase precision of the analysis, our analyser performs the computation across program method boundaries. This is called an *interprocedural analysis* [6]. Our solution uses a value-based whole program analysis. This approach directly computes the data flow information and propagates the inherited data flow information from callers to callees and the synthesized data flow information from callees to callers.

4 CONCLUSION

This paper describes the construction of a static analyser based on the Roslyn compiler syntax and semantic services. Firstly, the analyser constructs control flow graphs of individual program methods. Then it uses the worklist algorithm to traverse the control flow graphs and applies flow rules on the data context.

REFERENCES

[1] Aho, Alfred V., Lam, Monica S., Sethi, R., Ullman, Jeffrey D. *Compilers: Principles, Techniques, and Tools*. 2nd ed. Boston: Addison Wesley, 2007. ISBN 03-214-8681-1.

[2] Chess, B., West, J. *Secure Programming with Static Analysis*. Upper Saddle River: Addison-Wesley, 2007. ISBN 978-0-321-42477-8

[3] Hunt, T. *OWASP Top 10 For .NET Developers*. Pluralsite, 2011

[4] Ng, K., Warren, M., Golde, P., Hejlsberg, A. *The Roslyn Project: Exposing the C# and VB Compiler's Code Analysis*, http://msdn.microsoft.com/en-us/vstudio/roslyn.aspx, 2014

[5] Cormen, Thomas H., Leiserson, Charles E., Rivest, Ronald L., Stein, C. *Introduction to algorithms*. Cambridge, Massachusetts: MIT Press, 2009. ISBN 02-625-3305-7.

[6] Khedker, Uday P., Sanyal, A., Karkare, B. *Data Flow Analysis: Theory and Practice*. Boca Raton: CRC Press, 2009. ISBN 978-0-8493-2880-0.