IDC HERZLIYA | Efi Arazi School of Computer Science

The Interdisciplinary Center, Herzlia
Efi Arazi School of Computer Science

SAF: Static Analysis Improved Fuzzing

M.Sc. Dissertation

Submitted in Partial Fulfillment of the Requirements
for the Degree of Master of Science (M.Sc.)
Research Track
in Computer Science

Submitted by Moti Cohen
Under the supervision of Dr. David Movshovitz

May 2014

Acknowledgements

I would like to thank a few people for their help during the writing of this dissertation. I would like to thank my supervisor Dr. David Movshovitz, who provided constant support, encouragement and endless useful advice. I also want to thank Prof. Anat Bremler-Bar for the help in the submission of this work. Lastly, I want to thank my family for supporting me during this long journey.

Abstract

Security vulnerabilities are one of the major threats these days to Applications and Web sites alike. A report from May 2013[37] estimates that 86% of all websites had at least one serious vulnerability during 2012. This serious problem presents an opportunity for researchers and companies to investigate new techniques for the mitigation of these security issues.

During the last couple decades there have been many publications of different techniques that try to deal with the rise of security issues in modern application. These techniques are usually divided into two categories: static and dynamic. Static techniques use Static Analysis methods to investigate code structure and find bugs. Dynamic methods try to analyze the runtime behavior of a piece of software and find bad behavior at runtime. In the last few years there has been a rise in papers combining both approaches.

Researches in recent years have made great improvements in the effort to solve the problem we described. FindBugs [12] devised a method relying on software development best practices. Pixy [14] provides a Static Analysis based dataflow detection in PHP code. WebSSARI [13] and The Griffin Project [19] provide a Static Analysis with runtime analysis and program protection. These techniques have shown success in finding real bugs. Given that, these techniques usually suffer from a high rate of false positive results.

In our work, we try to deal with the high false positive rates using a different approach. Our work relies on an old software testing technique called Fuzz Testing [20]. This technique tests an application with random or semi random input, trying to cause un-wanted behavior. We suggest using knowledge learned from Static Code analysis to the improvement of the Fuzzing process.

The advantage of our approach is the ability to improve the performance and focus of ordinary fuzzing. A simple fuzzer has different input generation categories, for different testing scenarios. In each scenario, the fuzzer can generate numerous inputs for the testing phase. The knowledge of how the tested software operates can

help reduce the number of tests needed to be run, or on the other hand increase the number of relevant tests. For the "learning" of software structure and behavior, we built the SAF (Static Analysis Improved Fuzzing) framework.

The SAF framework is designed to analyze Java written web applications. The purpose of our framework is to learn the program dataflow, specifically the flow of user input, and use this learned information to improve Fuzz Testing. The idea is to locate program sinks (like SQL command execution), and to find dataflow paths that lead user input (sources) to those sinks. This can provide us with a categorization of each user input parameter, where the category represents the possible attack vector that be used through that parameter. With this information we can improve the performance of Fuzz testers by reducing non-relevant tests, and increase relevant ones.

We have performed an experimental run of our framework on three open source projects, one educational and two blog applications. These applications perform DB communication, File system access etc. We have managed to locate data flows that locate to sensitive program sinks, and have potentially reduced the number of parameters needed to be checked by up to 90% (at a given attack category, SQL injection for example).

Table of Contents

1 Introduction

Security vulnerabilities are one of the major threats these days to Applications and Web sites alike. A report from May 2013[37] estimates that 86% of all websites had at least one serious vulnerability during 2012. The first step to deal with these application layer vulnerabilities is to detect the vulnerabilities that exist in the application before the application is being deployed and even at the development stage. This subject has gained a lot of interest both in the academy and the industry for techniques to discover security bugs (vulnerabilities) prior to application production deployment. One of the main techniques to discover application vulnerabilities is via Fuzz Testing.

Fuzz Testing [20] (or Fuzzing) is a method to evaluate the ability of a program to handle illegal input by automatically generating inputs and passing them to the program. This technique have shown success in general bug detection [20, 21], and for security vulnerability detection [9, 24].

Blackbox style fuzzing is the simpler form of fuzzing, and is basically generating inputs for a given program, without knowledge of what "goes inside" that program. This technique is rather simplistic, relatively easy to implement, and yet has shown success [20]. Given that, this technique has a major limitation. The size of the input space is enormous, and because of that, some (or probably most) of the possible inputs are not generated, because this will take a long time. To counter this problem, researchers turned to Whitebox style fuzzing.

Whitebox fuzzing techniques are much "smarter", because they know how the software works, and try to use that information to their advantage. The basic idea is to analyze the source code, and try to extract valuable information for the testing phase. These approaches [7, 8] are based on Taint Analysis and Symbolic Execution and generate tests that are tailored to a specific code. The goal in these methods is to evade input validation and to change code execution paths, ultimately increasing code coverage.

The downside of Whitebox techniques is their complexity, taking major time to execute and might even be impossible [18, 5]. These techniques also focus on code coverage, and do not take into account the application surroundings (like communication to SQL DB). In most web application, there is a number of peripheral resources (DB, File System, etc.) used by the application, and therefore current Whitebox methods don't fit these types of application testing.

In this work we aim at improving the results of Fuzz Testing techniques via source code Static Analysis. Our work utilizes White Box techniques in order to gain knowledge about application structure, and by so enhancing the performance of Fuzz Testing. The basis of our analysis is to improve the fuzzing of injection-type vulnerabilities (SQL-injection, XSS etc.) by finding relevant sinks in the program (for example, SQL command execution) and following the data flow **to** those sinks, until reaching a source (user input). After finding such data flows, fuzz testing can focus its attention to these flows.

The advantage of our approach is the ability to improve the performance and focus of ordinary fuzzing. A simple fuzzer has different input generation categories, for different testing scenarios. In each scenario, the fuzzer can generate numerous inputs for the testing phase. The knowledge of how the tested software operates can help reduce the number of tests needed to be run, or on the other hand increase the number of relevant tests.

The structure of this work is as follows: chapter 2 provides a general background on Static Analysis techniques and algorithms, with an emphasis on the methods we used. Chapter 3 describes the major prior work done in the area of security flaw detection. Chapter 4 presents the limitations of the presented techniques and lays out our solution proposal. Chapter 5 provides a detailed review of our solution. Chapter 6 reviews the experimental results, and chapter 7 summarizes the dissertation.

2 Static Code Analysis

Static Code Analysis techniques exist for a few decades now, and have evolved greatly. These techniques have shown success in finding general software bugs [6, 13], and specifically for finding security flaws [3, 19]. In the following sections we review the main algorithms and data structures that are used in the field of Static Code Analysis, and in this work as well.

2.1 Abstract Syntax Tree (AST)

This structure holds the syntactical structure of a program's code. When parsing program code, the generated structure is an AST. This is the basic data structure for all code analysis techniques, as well as compilation techniques.

Programming languages are usually described as Context Free Grammars, which represent the language's statement structure (grammar free). These grammars can be represented as a tree, where each node corresponds to an expression in the relevant language [11 (chapter five)].

2.2 Intermediate Representation (IR)

Compilers and standard Static Analysis techniques usually transform the analyzed code from the source language (Java, Bytecode, etc.) into an equivalent representation in some mid-level language. This new representation is used in order to simplify the analysis phases, because the IR is supposed to be simpler and probably more suitable for analysis.

2.3 Static Single Assignment (SSA) Form

SSA form is a property of program representations that states that a variable can be assigned a value only once anywhere in the program. This means that multiple assignments to the same variable create new versions of that variable and practically new variables. This representation is usually part of the IRs that are used, and has great importance in simplifying some analysis algorithms, like constant propagation [36].

2.4 Control Flow Graph (CFG)

A CFG represents the transfer of control in a given program. Usually, a CFG is built per method. The basic idea behind this data structure is to represent the possible execution paths a program can go through. For example, if we have an IF statement, then the predicate statement points to two blocks of code, the THEN statement and the ELSE statement.

This structure is very useful for program traversal, understanding program dependencies between statements, etc.

2.5 Call Graph

A Call Graph is a directed graph that holds the calling relationship between different methods in the program. Every method in the code is represented as a node, and an edge represents a call from method A to method B. In this work we talk about a Static Call Graph, which represents all calling relations between methods, as opposed to a dynamic call graph that represents calls that were performed in a single program execution.

Call Graphs do not only hold caller-callee relationships, but might also hold a chain of method calls leading to a specific method. This is called the context. For each method, a node is created for each possible call stack (chain of method calls) leading to that method. The calculation of this graph is undecidable, and therefore existing algorithms use approximations.

2.6 Dataflow Analysis

This is one of the core techniques in code analysis for all applications. The purpose of dataflow analysis is to find at a given point in a program the set of possible objects that can reach that point (a constant, an object defined in a prior instruction). For example, Constant Propagation deals with the flow of constants to different variables in different program locations. Given a specific variable and a specific program location, does that variable at the given location point to some constant value? And if so, to which value? The analysis itself can be performed on a per method basis (intraprocedural), or on the entire program code, taking into account method calls (interprocedural).

Dataflow analysis comes in many flavors, according to the desired application. The basic concepts in Dataflow are:

- Flow sensitivity – does the analysis take into consideration the control flow of the program? In most cases, the analysis is flow sensitive.

- Context sensitivity – in the case of interprocedural analysis, is it aware of the actual method caller (the context) and the chain of calls, or all call contexts the same.

2.7 Def-Use Graph

A classical property calculated in Static Analysis is reaching definitions. Each assignment or calculation instruction defines a new value (definition). This definition is used in future calculation, and the desired property to calculate is at a given program point, which definitions is "live", meaning not been overridden by another instruction. The solving of this problem yields the data structure Definition-Use graph. The idea is to be able to find for each value defined its future uses (forward traversal), or at a given use which definitions might reach that point (backward traversal). This graph is normally built per method, and is very useful for data flow analysis. We use this data structure frequently in our analyses.

2.8 Pointer and Alias Analysis

Pointer Analysis (or Points-to) comes to answer the question to which area in memory a pointer points to. Alias Analysis answers the question when two pointers point to the same area in memory. As mentioned in prior analyses, these also have different configurations and precision levels: flow-sensitivity, inter-procedural, context-awareness. These analyses are some of the most advanced, and also most complicated. Most algorithms that exist use some sort of approximation [17].

3 Prior Work

In the last couple of decades, there is a significant increase in research of software bug detection techniques and software protection. Bug detection techniques are used for general and security flaw detection. The main goal in this case is to locate security bugs prior to production deployment. Application code is analyzed and searched for different types of software bugs, from general coding errors (null pointer dereference) to security vulnerabilities (data leakage). Software Protection is meant to protect applications at runtime. In this case, the idea is to change the application behavior or environment in a way that will provide runtime protection.

Application manipulation is usually done by code instrumentation, where guarding code is inserted prior to sensitive instruction execution. Of course, there are some hybrid methods which incorporate the two types together. In this section we will provide a review of the major work done in the area, with an emphasis on security flaws.

3.1 Static Analysis for Security Flaw Detection

3.1.1 FindBugs

FindBugs [12] define error prone Code Patterns (based on real-life bugs), and implement pattern searching mechanisms. These patterns include many types of bug classes, for example Thread Synchronization and Performance. They also incorporate security related patterns, like SQL Injection potential. FindBugs implements standard code analysis mechanisms in order to identify these patterns.

The emphasis in the FindBugs paper is on best-practice code analysis, rather than on complicated analysis techniques. They implement a set of Bug Pattern Detectors, which look after very simple but unhealthy code structure. These detectors traverse the program code, making use of the Control Flow data analyzed by the framework, and some incorporate Intraprocedural Dataflow analysis.

In the case of SQL injection, the authors of FindBugs implemented a simple pattern detector. The best-practice is to use constant SQL commands, or commands created from constant parts. Their analysis searched for SQL commands that were created from the concatenation of different strings that some of them were created on the fly.

Their technique has a few advantages. The simple analysis allows them to scan major code bases without the need to worry about performance. One of the major drawbacks of static analysis is its potential running time. Their approach also allows the simple and easy writing of bug detectors.

There are two major pitfalls with the FindBugs approach. One, each unique pattern has to be implemented in the tool, and small nuances might cause the tool to not recognize the actual pattern. Second, the lack of global information about interprocedural interactions causes the tool to either provide false positives, or miss potential bugs.

If we take a look at the SQL injection detector described above, we can see how their assumptions can lead to false positive and negative results. In their analysis, they treat dynamic SQL queries as bad – this means also configuration read queries. This will lead to enormous amounts of false positives in most software projects. In addition, they don't take into account information about the function context, and functions called. This also leads to false alerts, and can also lead to false negatives.

3.1.2 The Griffin Project

The Griffin project [19] provides a comprehensive static and runtime analysis of J2EE programs, in effort to find security vulnerabilities. In his work, Benjamin Livshits implemented a sound static analysis framework and combined it with program instrumentation that provides runtime analysis techniques.

The static analysis phase in Livshits' work, is based on a source to sink lookup. The main concept behind this is to define program locations which are sensitive as sinks (SQL command creation, HTML page generation, etc.), and to define potential harmful inputs locations as the sources. After defining these sets of program location, the task is to find data flow transitions which cause data to propagate from a source to a sink.

The output of the previously described analysis is a set of data flow paths. These paths might contain sanitization (input validation), but the problem is that in some cases it cannot be definitely asserted that a path is sanitized. Therefore, this might result in a false positive alert (a path might be falsely identified as dangerous, though the input is actually validated).

In his work, Livshits provided some significant improvements to existing static analysis techniques, especially Pointer Analysis precision improvement and reflection analysis precision improvement.

In addition to providing a static analysis framework, which has shown useful for finding security bugs in existing applications, The Griffin project also provides a runtime analysis framework that complements some of the limitations that exists in static vulnerability detection (some were mentioned above).

The runtime mechanisms in the Griffin project are composed of two parts. First, a data flow tracking mechanism which follows the data propagation between source and sink at runtime, and therefore knows the actual values at execution (as opposed to static analysis). At a given sink in the program, the analysis framework can verify if the arrived data was sanitized. For cases where the data was not correctly sanitized, comes the second part of the framework. This part provides "recovery" mechanisms, which are calls to sanitation functions.

Sanitation is defined in Livshits' work as a set of methods that stop "taint propagation", meaning the output returned by these functions is considered to be valid. If a tainted data arrives at a sink, it means that data didn't go through sanitation (this precise analysis can only be done at runtime). In the static case, if there is a possible path by which data that is not sanitized arrives at a sink, that is considered a vulnerability, though it might be a false alarm because of static analysis approximations.

The Griffin project showed impressive results in the field of vulnerability detection. Despite these results, the work has its limitations. The Static Analysis has the inherent limitation of being intractable, and therefore is bound to be based on heuristics and approximations which lead to false positive/negative. On the other hand, the runtime analysis has an impact on performance, and can also introduce new bugs to the instrumented application. Livshits noted some of these limitations in his work.

3.1.3 Scripting Languages

In the field of web application scripting languages, there has been some advancement with the creation of Pixy [14] and WebSSARI [13], both projects currently analyzing PHP applications.

3.1.3.1 Pixy

Pixy [14] provide a static analysis framework for security vulnerability detection in PHP code. The Pixy framework is designed to locate Taint-Style (Injection) vulnerabilities. One of the major challenges and limitations of the PHP language is its un-typed nature.

Pixy is based on Dataflow analysis. Their implementation takes into consideration interprocedural relations, flow data, and context awareness. In addition to this powerful approach, they also implemented an alias analysis and constant propagation (also called literal analysis). These techniques improve the precision of their technique, basically reducing false positives.

Pixy is a powerful analysis framework, but has its own limitations. The authors implemented only method analysis, and do not support the object oriented structure which became more powerful in version 5[4]. Pixy also suffers from false positive results. In their limitations section [14, 15] they refer to some false positive results that arise from a few different PHP coding scenarios. The general false positive ratio described is 50% [15], but the benchmark is of medium size, and perhaps needs some expansion to evaluate the methods precision. On the other hand, they don't provide information about false-negative, and don't categorize their analysis as a sound one.

3.1.3.2 WebSSARI

WebSSARI [13] is similar to Livshits' work, by incorporating static and dynamic techniques. The tool can be divided into two phases. Phase one analyzes the code for potential dangerous information flows. The mitigation to these dangerous information flows is provided via phase two, which inserts guards that somewhat protect the code analyzed.

This approach has two major limitations, similar to Livsiths' work. The static analysis provides false positive alerts. This issue is more severe in this work due to the dynamic and un-typed nature of PHP. In order to provide a sound analysis, the authors had to make some approximations which provide many false positives. The second limitation is production influence. The instrumentation phase can insert two unwanted behaviors – performance deterioration and software bugs.

3.2 Dynamic Methods

Dynamic analysis methods can somewhat be considered as complementary to static analysis. These methods are based on data examination at execution time, rather than compilation (or offline) time. There are two major categories of Dynamic Analysis: Dynamic Taint Analysis and Fuzz Testing and Symbolic Execution. We will describe in brief these techniques and review some research results.

3.2.1 Dynamic Taint Analysis

Dynamic Taint analysis is incorporated into the source program, and examines the program behavior at runtime. The idea behind this technique is to observe the data propagation and manipulation of suspicious data (tainted data), for example user input or file content. These techniques were described above as part of WebSSARI[13] and Livshits'[19] work, and were researched extensively.

Dynamic Taint Analysis (and other runtime methods) try to protect at runtime from possible attacks. This approach has advantages, for it lowers the possibility of a successful attack on a given application. On the other hand, the limitation to these approaches is the change in code behavior. These techniques can cause a serious performance impact, and even introduce new bugs and vulnerabilities.

3.2.2 Fuzz Testing and Symbolic Execution

Fuzz Testing or Fuzzing is a mechanism of software testing via random or semi-random input generation. This technique was first introduced by [20], where random input strings were generated and passed to various UNIX utilities of different versions. Although the simplistic testing approach, this work have shown to be very effective in finding different kinds of bugs, in versatile UNIX tools. This work opened the door to automatic input generation techniques and automatic testing.

Modern Black-Box fuzzers provide different kinds of inputs, that is divided into several application input categories, in order to try and "bubble up" the relevant software bug [25, 27]. For example, a fuzzer will try to generate inputs that might cause an SQL injection bug appear, or a Cross Site Scripting relevant input. These different inputs have many nuances, so there are many inputs in each category.

JBroFuzz [25] is based on categorized inputs. Each input category is related to a specific applicative behavior. For example, there is a SQL injection category, which is related to DB communication, and there is a numeric category that is related to numeric computations. These different categories are intended to allow the tester to check different application capabilities, and it is up to the tester to choose the correct categories.

The major limitation with classical Black-box fuzzing (like JBroFuzz) is the need to specifically configure the tool for each parameter tested with the relevant categories. JBroFuzz, for example, has about 50 input categories, and running all categories for all input parameters is not relevant at best case and not feasible at worst. When correct input categories are selected for each parameter, it is possible to test more thoroughly each parameter, because we narrowed the search space.

Besides the Black-Box fuzzing approaches, there is an increase in White-Box style fuzzing research in the last few years. White-Box fuzzing tries to bring the knowledge of code analysis into the process of application fuzzing. The aim is to provide "smarter" and more precise input generation that will provide better code coverage.

In [8] a White-box fuzz testing framework was developed, called SAGE (Scalable, Automated, Guided Execution). SAGE receives a program to test, and performs an execution and analysis step over and over. The program is run on an input file for it. While the program is running, its behavior is observed. During execution trace logs are kept for the analysis phase. After the program finished running, and assuming no unusual events occurred, the trace logs are analyzed via Symbolic Execution which builds a constraint problem of input constraints. The last phase is to use a constraint solver to solve the generated problem, and the solution is the input for the program in the next iteration. The input generated by the constraint solver should cause the program to avoid some conditions or take branches which were not taken in the previous execution. The SAGE framework has shown to be effective in finding many bugs, including in some major commercial products.

4 Presented Analyses Limitations and Our Solution

After reviewing the existing approaches and algorithms in the field of security flaw detection, we will try to summarize in this chapter the limitations and problems that exist in these solutions. The limitations will be divided to two categories: Static and Dynamic methods.

4.1 Static Analysis Limitations: False Positive/Negative review

Static Analysis tools have one major limitation. Due to the fact that most questions that can be asked about code are undecidable, common algorithms perform approximations and heuristics. This leads to two possible outcomes, false positive result and false negative result. A false positive is a warning that is not really a bug. A false negative is a true bug that was not caught.

Most Static Analysis tools and frameworks (including the ones described here) usually provide sound analysis, meaning there shouldn't be false negatives but for the price of false positives.

In [1] a review of the experimental results of FindBugs was performed, and at least 30% percent of the warnings provided by the tool were irrelevant. Kupsch and Miller performed in [16] a thorough comparison of two of the best Static Analysis tools in the market and manual code review. They concluded that the existing tools are not strong enough to find all the serious bugs and moreover these tools provided an enormous amount of alerts that had "a serious impact on the effectiveness of the analysis".

4.2 Dynamic Methods Limitations

The different dynamic methods we presented in the previous chapter have a few serious drawbacks and limitations, as previously noted. First, let's discuss input generation. Fuzzing and other input generation techniques have the problem of large input space in which these application search for the correct "bug-bubbling" input. In the case of SAGE, these types of applications might suffer from a lack of "context" understanding, as can be shown in the following example:

```
protected void doGet(HttpServletRequest req, HttpServletResponse resp)
                       throws ServletException, IOException {
        Connection conn;
        conn = makeConnection();

        Statement statement = conn.createStatement();

        statement.executeQuery(req.getParameter("Query"));
}
```

SAGE looks at the code, trying to maximize code coverage with input generation. Any input in the Query parameter will cause this code to execute, which means SAGE will not try different inputs against the Query parameter. But this code is trivially faulted, because the query that is executed against the Database is provided by the user. This code will cause an exception or unintended behavior if the input provided will be an illegal SQL command or a legal but malicious SQL command.

The second type of dynamic analysis we review was Dynamic Taint Analysis. These techniques were based on runtime input inspection and different types of remedies during runtime. This approach has two possible problems: performance impact and code behavior changing. Since these methods insert new instructions to the source code, the newly instrumented code has to run slower than the original. For example, in [2] their instrumentation (which works on C programs) can cause more than 12% impact on runtime performance, and other frameworks are worse than that.

In the case of behavior changing, code instrumentation changes the actual program behavior (trying to fix bugs and protect the system), but can also introduce new bugs, to which the original system programs are not aware. This can be a real problem in production environments when trying to find out why a program is not working correctly.

4.3 Our Solution – The SAF Framework

In this chapter we have discussed the limitations of existing analysis methods and frameworks. Our work tries to enjoy both worlds (static and dynamic) and give a new solution to the presented problems. Our tool takes information learned by Static Analysis, and uses that information in order to perform Fuzz testing optimization.

We chose this hybrid approach for the following reasons. Although classical fuzzing has shown to be useful in finding many software bugs, it still has the limitation of going over all possible inputs, and this issue leads fuzzers to find some of the bugs, and most of the time the trivial ones. Static Analysis has shown to provide valuable information for different testing schemes, and therefore we believe this information can be useful to improve the searching algorithms of classical fuzzers.

Our tool takes as input a web application written in the Java language. The tool follows these steps:

a. Find program key points – This step goes over the code, and find interesting points which behave as data sinks or sources.
b. Before doing a full code analysis, we make a rough analysis, which should find for each web page which sink groups it possibly has (SQL, XSS, etc.).
c. The next step is to find source to sink data flows, which imply that program input (web request parameters) may arrive at the found sinks.

The steps described here provide for each page the list of possible sink categories, and for each parameter the relevant sinks we found it to arrive to. This information can later be used in order to optimize web application fuzzers, to focus the testing effort effectively by providing relevant input only to the relevant page parameters.

4.4 Analysis Challenges and Opportunities

The purpose of our analysis is to locate the "route" each application parameter goes through, and for what it is used. This question has a few challenges hidden inside:

- We need to find the actual path each sink derived data goes through. This means we not only need to find the existence of a possible harmful data flow, but to locate the actual parameter. Without knowing the actual parameter we will not be able to guide fuzz testers.
- We need to perform a two phase flow analysis. Sources are usually a call to some API method that reads a specific parameter from the web request. The name of the parameter is passed to that API call. After finding a source-sink path, we need to locate the actual parameter used in that source.

On the other hand, our analysis approach has its advantages. Since the purpose of our approach is to reduce the amount of testing performed by an automated tester (the fuzzer), we are a bit less concerned with false positives. As will be discussed in the results chapter, there are levels of configuration that can lead to higher accuracy and less false positive results (at the price of performance). This means, we can scale better since it is less severe to have some false positives.

A second advantage is the complexity of the performance. Eventually, we perform only data flow analysis on the analyzed program. Some of the approaches presented in chapter 3, have very computationally heavy parts (like constraint solving for input generation) which might cause the whole testing framework to not scale. The use of fuzzing with a more lightweight analysis might be more suited for large code bases.

5 The SAF Framework

In this chapter we introduce the SAF (Static Analysis Improved Fuzzing) framework. Our analysis framework is based on Static Code analysis algorithms together with our own heuristics which come to deal with the problems we encountered during the research. We will provide a review of the different stages of the analysis, with emphasis on our additions. Some of the algorithms and data structures we relied on are described with greater detail in the Static Analysis Appendix.

The input to our analysis is a compiled web application written in Java [22] (1.6 or prior). Meaning, we evaluate Java Bytecode, and therefore don't require the source code. The entire analysis framework is based on the Wala [31] framework, provided by IBM T.J. Watson Research Center.

5.1 Sink Locator

Our Analysis starts with locating the program points which represent sinks. Sinks are locations which act as a data flow ending point, because of their sensitivity. For example, a SQL query creation statement is a sink, because any string that will arrive to that point will be sent to the DB. For our analysis purposes, possible sinks are specific API method calls. We describe a sink as a three tuple$< c, m, i >$, where c a

Java class, *m* is a method inside that class and *i* is the instruction that describes a class to a sensitive API method, inside the method *m*.

5.2 Request Parameter Locating

The last issue we deal with is locating the actual user inputs – request parameters. When performing a call to a web page or a servlet, the user provides the name of the page and various parameters. These parameters are provided by key-value pairs through the URL. We wish to track the actual parameters that were propagated to the interesting sinks, rather than saying that some parameters arrived. This gives us better precision in focusing Fuzzers to the correct parameters.

A request parameter read in Java is a simple API call on the ServletRequest object, passing it the actual parameter name. We can therefore treat this location as another sink, searching for the parameter name origin. We perform a traversal on the passed name object, and currently can find the actual parameter name if that was defined in the code (as a constant). We use all the methods described in the previous sections, and the points-to analysis provides us the actual constant value.

5.3 Sink Elimination

This stage is a simple one, and its intention is to clear out irrelevant testing categories for a given page. Meaning, if a page *p* has no SQL statement executing inside its possible execution paths than we need not check for SQL injection vulnerabilities. This analysis is done by constructing the program Call Graph (With Wala), and afterwards traversing the Call Graph backwards (from callee to caller) to locate possible pages it is located in. More on the Wala Call Graph data structure in Appendix B.

5.4 SAF Dataflow Analysis

This is the most complex and important phase in our analysis process. After locating relevant sinks inside the analyzed application, we wish to find if these sinks were provided input from outside sources, specifically web request parameters. The analysis is done from sink to source (backwards analysis), where for each sink found we perform a backward traversal on program instruction to find the different assignments that led to the sink.

Our analysis uses a few important constructs and algorithms implemented in the Wala framework. These constructs are:

- Context-Sensitive Call Graph – we use the Call Graph data structure, and we use a version that is context sensitive. Meaning, for each method, every call stack that can lead to that method is saved inside the graph. We use a Call-String context, as explained in Appendix B. Currently we use a 1-level call string context, as a higher (more precise) call string has a major performance impact, and currently impractical.
- Def-Use Graph – we use the Definition-Use graph provided by Wala, which is constructed for each method. This allows us to perform a traversal inside a given method over definition chains.
- Pointer-Analysis – we use the Wala provided context-sensitive pointer analysis. This provides us in some cases (due to analysis limitations, like call-string context limitation) to which object a pointer is pointing to (actually where that object was created).

5.4.1 Dataflow Algorithm

Our analysis algorithm is a recursive one, and is composed of a few sub methods (algorithms). So we will provide the general flow, and describe in detail the inner parts later in this chapter. The general algorithm for sink-source path finding goes as follows:

```
(1)  foreach sink in sinks
(2)       BackwardTraverse(sink)



(3)  BackwardTraverse(Instruction instruction)
(4)       foreach use in GetUses(instruction)
(5)            defs <- FindDefinitions(instruction, use)
(6)                 foreach definition in defs
(7)                      BackwardTraverse(def)
```

The general algorithm is very simplistic, because the basic idea behind it is very simple. If, for example, we would have had a graph describing precisely all data transitions possible in a given program, we need only to traverse that graph. The actual difficulty arises from the graph edge building, meaning to find for each object where it might have been defined or assigned.

Get Instruction Uses (line 4 in the pseudo code)

If we look back to SSA form described in the Static Analysis background, we noted that each value is defined once, and that each assignment to the same method variable creates a new value. The function of getting instruction values used, is to simply locate the value numbers that are used as arguments to the given instruction. Let's look at a simple example:

```
(1)  Int a = 5;
(2)  Int b = 10;
(3)  a = a + b;
```

The code provided above creates three values in SSA Form (which is used in our analysis), but uses only two method variables. Instruction 1 creates V1 which is the Integer value of 5. Instruction 2 creates V2 which is the Integer value of 10. The last instruction creates V3 (although it is assigned to variable a), which has the value of 15. When analyzing instruction 3 we can see that it uses V1 and V2, and provided the right data structure, can go back to the defining instruction for these values. This is the actual method of locating instruction uses, and when using the Wala infrastructure it is simply calling a method [34].

Get Value Definitions (line 5 in the pseudo code)

After we have located for a given method the different values used as parameters, we wish to locate the points in the program where these values were defined. We use two main mechanisms during this analysis, Points-to analysis and code traversal.

The first step we try to perform in order to find to which value a program-variable points to is to run the points-to analysis provided via Wala. We use an interprocedural, context-sensitive configuration to get a more precise analysis. The answer this analysis gives us (if it can detect it) is a program instruction that defined the object we are referencing, or a few possible instructions. This could be a constant assignment, a call to get an input from the user, etc.

If the points-to analysis was in-capable of locating where the current variable was defined in, we perform a traversal over the program code, locating the possible program instructions that defined our used values. This is done on two manners. Inside a given method, we go over the definitions-use graph described earlier, finding which instruction defined our value. If the current used value is a parameter passed to the current method being analyzed, we locate via the Call Graph data structure all the possible methods that called our method, locate the interesting parameter in that call, and go on with analyzing that parameter.

After we have located the instruction or instructions that defined our interesting value, we continue analyzing that instruction's uses onward, until we reach a sink (request parameter get). At this point we continue code analysis with the same technique in order to locate which request parameter was read, and that is the real sink of interest.

There is a limitation to the code analysis described here. If the points-to analysis fails in telling us where an object we might be pointing at by a given variable was defined, we perform a method based code traversal. But if data was passed via an object field or static field, and not via method parameters, we will be missing this data transfer. For this issue we have performed some analysis heuristics that locates where an object field was possibly initialized, and these points are the next phase for our analysis. We will describe these heuristics in the next section.

5.4.2 Object Field Access Heuristics

The object field analysis comes to mitigate a major drawback in the analysis described in the previous section. Given that we are handling an object oriented language, most information is stored and transferred via object fields. Given this limitation, we came up with this simple but scalable heuristic.

Assuming we perform the general traversal algorithm described earlier. And assuming we have reached an object field read, and want to traverse to the origin of that field. In this case we perform a lookup of all possible assignments performed to that field that were originated from the same call stack that we might be in. The field access operation is located in some method. Let's call the set of execution paths (method calls) that lead to the actual method executing the read *p1, … , pn*. Each *pi* is a path of method calls, from the entrance method up to the method calling the read from the object field. All these paths end up in the same method, so we can look at it as a reversed tree. Let's look, for example, at an assignment to the field occurring at a different method. We can build for that instruction the same reversed tree of possible call stack, called *q1, … , qm*. We say that there is a possible relation between the read and the write if two call-stacks intersect, meaning there is a method that called both the assigning and the reading methods (even if not calling directly). We perform this analysis by doing a Breadth-First Search on the call graph, starting from the method executing the field assignment, and after locating all possible field writes.

The previously described algorithm gives us for a given read from a field the set of possible assignments to that field. These assignments are the next instructions we analyze in our traversal algorithm described above.

5.4.3 String Manipulations

During the building of our analysis framework, we come up with a serious limitation. We are not analyzing the behavior of the Java supplied framework elements. There are two types of manipulations: string manipulation objects – StringBuilder and StringBuffer, and string provided methods (substring, replace etc.). The purpose of these constructs is to handle string concatenation, replacement, trimming etc. Since we don't analyze the way these constructs work, we missed the data flow that occurs through the inner data structures of these objects.

First, we will show the improvements for StringBuilder and StringBuffer. Let's look at a short code example:

```
      StringBuilder builder = new StringBuilder();

      Builder.Append(inputString);
```

The code above describes the appending of an input string into a StringBuilder object. The string that was contained in the input string variable is now concatenated into the inner structures of StringBuilder, and therefore that structure is now "tainted" with user input. When that data is read, the input will be passed along.

We mitigated this problem, by treating the calls to StringBuilder methods as tainting. When we see that a value from a specific StringBuilder is reaching a sink, we try to find method calls on that StringBuilder that inserted input, and then traverse that input to see if it came from a user input. This way we avoid the analysis of the StringBuilder internals, but tackle the relevant behavior. The improvement shown here is the same in the StringBuffer's case.

In the case of string provided methods, we have provided special treatment to methods that provide a part of a string as an output. For example, the substring method acts on a string instance, and return a part of that string. If the original string is tainted, then so is the returned string. We have identified these locations and followed the data flow through them.

5.5 Algorithm Execution Example

We wish to provide an example of all the algorithm steps, in order to better explain how the different phases interact and how each part works. We will be analyzing a small code section from an educational application called WebGoat [28], which we are also analyzing in the experimental part.

The following code is taken from a simple servlet, which shows bad practices in the context of user authentication:

```
(1)   username = s.getParser().getRawParameter(USERNAME);

(2)   password = s.getParser().getRawParameter(PASSWORD);

(3)   String query = "SELECT * FROM user_system_data WHERE user_name
      = '" + username + "' AND " + "password = '" + password + "'";

(4)   Statement statement = connection.createStatement();

(5)   ResultSet results = statement.executeQuery(query);
```

The scenario shown here is very simple. We have an object defined in WebGoat that reads request parameters, which is called ParameterParser. The instruction *s.getParser()* retrieves that object. The first two statements read two parameters from the request. The third instruction builds a SQL query with the read data. The fourth statement creates a new SQL statement, and the fifth instruction executes the created query against the DB. The vulnerability in this case is fact that user input goes un-checked to the DB, and this flaw can easily be attacked by even an un-skilled attacker.

The following sections will go over all steps of the algorithm described earlier to show how all steps interact.

5.5.1 Sink Locating

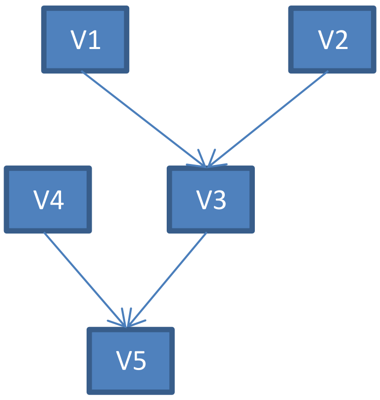
When we talked about the sinks we defined them as specific method calls. In our case, the sensitive call is line 5:

```
(5)   ResultSet results = statement.executeQuery(query);
```

The dataflow analysis phase will start from this point, and traverse the code backwards to locate all the interesting data locations.

5.5.2 Dataflow

The analysis in this simple example is based on the def-use graph we mentioned in the dataflow section. The def-use graph looks approximately like this (value numbers correspond to the instruction numbers):



The interesting part of the shown def-use graph is the flow from V1+V2 -> V3 -> V5. So, we are moving on the graph backwards. First, we analyze V5 and understand it uses V4 and V3. The next step is to analyze V4 and V3. V4 has no interesting uses in it, so we drop it. But, V3 has interesting uses, and we follow them. V1 and V2 were defined by a call to getRawParamter. The actual code inside reads a parameter from the web request (the parameter name is the one passed to getRawParameter). When we analyze the code for getRawParameter, we can conclude that there is an interesting path from a user defined value (parameter) to a SQL command.

5.5.3 Request Parameter Locating

After we concluded there is a dataflow between a parameter read and a SQL command execution, we need to find out the real parameter – the name of the request parameter. In the code example, this is the constants passed as a parameter to the getRawParameter method. It is very important to understand that there are actually two parameters being read, and to locate the two names. In this case it is rather straight forward since the parameter names are constants being used directly in the code.

5.5.4 String Manipulation

In this example we analyzed a string concatenation command – line (3). Theoretically, this can be performed by a special concatenation operator which creates a new string that holds the value of the strings provided to it. In the contemporary versions of Java compilers, these commands are compiled to the use of StringBuilder. This means that the simple command we showed is compiled to the following code:

```
(1)    StringBuilder builder = new StringBuilder( "SELECT * FROM
       user_system_data WHERE user_name = '");
(2)    builder.append(username);
(3)    builder.append("' AND " );
(4)    builder.append("password = '");
(5)    builder.append(password);
(6)    builder.append("'");
```

Since this is the real code we are analyzing, we need to take care of the usage of the StringBuilder object. What we do is to locate the operations that insert values into the string builder (all instructions in this example), and to traverse the parameters to these instruction. The more interesting ones are the parameters to instructions (2) and (5). This changes significantly the def-use analysis we described earlier. We need to locate the instructions that use the StringBuilder object (2-6), and check if these instructions are adding values to the builder object. Then we can continue as described earlier by searching the definitions of the values used in the append instructions.

5.6 SAF Limitations

The analysis framework we have built has a few limitations, and here we will list them and provide an explanation for each:

- Non string parameter values – the analysis framework implemented handles only string value propagation. This is a minor issue because most injection vulnerabilities are related to string value handling, where non string values (mostly numeric) are less risky.
- Constant parameter names – In order to locate the request parameters that arrive at a certain sink, we assume parameter names are constants in the code. There are other ways to map parameters, but we do not cover these in this paper.
- Lack of array typed object members handling – array fields were more complicated to analyze, and given the fact that this type of fields is not common we have decided to postpone this issue to future work.

- Limited reflection handling – our analysis handles reflection partially. Some reflection usages have shown to not work, mostly from framework limitations. There are possibilities to improve precision, but these options are not scalable to even small applications.

6 SAF Experimental Results

In order to test the capabilities, accuracy and performance of the SAF framework we have devised a set of test cases. The test set is built of two parts. First, a set of synthetic cases we written to check the accuracy of our framework. Second, a set of applications taken from the Stanford Securibench [30] project.

The synthetic test set is built of 23 tests that exhibit different data flow scenarios:

- Intra-procedural data flow
- Inter-procedural data flow, via parameters
- Data flow through object fields
- Flow through static object fields
- Different control flow constructs (If, switch statement, loops, exception handling)
- Object inheritance recognition

We ran the SAF framework on these tests and found all data flow routes. In the next two chapters we will describe the real life benchmark and discuss its results.

6.1 Stanford Securibench [30] benchmark

The Stanford Securibench is a collection of various real life applications, grouped together to provide a benchmark of programs for code analysis techniques. This benchmark was collected by the researchers in Stanford and used in Livshit's doctoral thesis.

We have performed our analysis on a portion of these applications, since we targeted Java web applications, specifically Servlet based.

6.1.1 OWASP WebGoat[28]

WebGoat is an educational web site being developed for a few years. WebGoat is a deliberately vulnerable application that was built to teach and show all kinds of web vulnerabilities and how to avoid them.

We have taken WebGoat's version 0.9 that comes with SecuriBench. This is a rather primordial version of WebGoat, but it serves our needs.

6.1.2 Blueblog

Blueblog is a blogging application, written in Java and based on the java servlet mechanism. It is a small application, designed to allow the creation and viewing of different blogs. Inside, the different blogs are stored on the server file system, in different folders for different blogs.

We decided to include this application in our analysis because it is a real world open source application, and its design and technologies are relevant for our analysis framework. We are analyzing the version taken from Securibench.

6.1.3 Personal Blog

This is another blogging application taken from the Securibench benchmark. Personal Blog is a simple blog application written in Java with various frameworks in use. We focus on this application because it has a different kind of injection vulnerability that resembles SQL injection.

6.2 Securibench Results

In this section we will review the analysis results on the benchmark application described in the previous section.

6.2.1 WebGoat

As described before, WebGoat is an educational application, with different vulnerabilities embedded inside its code. The WebGoat project is divided into lessons, where each lesson is a different servlet. A lessons comes to show a specific web vulnerability (like persistent cross site scripting), but might hold more vulnerabilities. The version of WebGoat we are analyzing is composed of 12 lessons. We have performed two analyses on the WebGoat code. The first tried to locate data flow from different user input to a SQL command. The second analysis tried to locate data flow to shell command execution statements. We will describe each shortly.

6.2.1.1 SQL Analysis

We have performed data flow analysis from web request parameters to SQL statements (not as parameters, but strings that were concatenated to SQL commands). The data flows we have been able to locate might be susceptible to SQL injection attacks, and therefore might put the application at risk.

Out of the 12 lessons being analyzed, 5 of them were not using SQL at all, and our code recognized this correctly. Meaning, There is no need to try and execute SQL injection attacks on these pages. In the pages where SQL commands are executed, we have managed to locate all the parameters that lead to a SQL command, but some of the parameters that were not being used in SQL commands were wrongly outputted (33% false positive). This issue was mitigated by using a more precise Call Graph that took into account a greater call stack (more on call graphs in Appendix B.). This tweak solved the problem, but caused the analysis to run longer and might not be usable in large code bases.

With regular analysis configuration we have come up with 0% false negative but 33% false positive. With greater analysis precision (Call Graph tweaks) we managed to get 0% false positive.

6.2.1.2 Command Execution

Inside WebGoat there is a lesson showing Parameter Injection. This vulnerability class relates to a security bug that allows an attacker to run specific programs, or change the behavior of programs being run by the vulnerable application through passed parameters. This means that a user supplied parameter somehow reaches a shell script being run. The WebGoat lesson related to this vulnerability has a bug allowing the user to run any program. In our analysis we have been able to find the parameter reaching this vulnerability, and showed that other lessons are not vulnerable to this – meaning 0% false positive and 0% false negative.

6.2.2 BlueBlog

As opposed to WebGoat, BlueBlog is a real application, and has a lot less vulnerabilities. Since there is no use of SQL, this application is not susceptible to SQL injection attacks, and as our analysis showed, not vulnerable to command execution. Given that, we have managed to find a potential vulnerability related to Path Traversal. This vulnerability type is described in Appendix A.

We managed to locate two locations in the BlueBlog code that access a File object (representing file system file), and the path to that file is constructed from two locations: first, from the URL (which can be changed by an attacker), and second, from a specific request parameter. In this case we also had no false positive or negative results.

6.2.3 Personal Blog

Personal Blog uses different frameworks, and specifically Hibernate [10]. Hibernate is a Java implementation of an ORM, which a framework for DB communication. Therefore, Personal Blog doesn't use native SQL queries, but rather lets Hibernate handle all the messy stuff. Hibernate is considered rather secure, but it also has an injection vulnerability if not used correctly. This is called HQL (Hiberante Query Language) injection [23]. This is the case with Personal Blog. We have analyzed the application code and found two paths that lead from a user provided input to an HQL query – which is possible injection vulnerability.

We have analyzed the entire Personal Blog source code, and located precisely the two request parameters that lead to an HQL query. In addition, we had no false positives or negatives. Out of the 29 request parameters read from user input we have located that only 2 were leading to HQL queries. This is a reduction of 93%.

6.2.4 Performance Analysis

We have performed the experimental phase on a standard laptop with the following specifications:

- Intel Core i7 2.2 Ghz, Quad core CPU
- 8GB RAM

- 64-bit Windows 7
- Java SE 1.6

These are the running time we achieved for the relevant benchmark applications:

Application	**# sinks**	**LOC**	**Time 1(s)**	**Time 2(s)**	**Time 3(s)**
WebGoat	37	4827	53	100	561
Personal blog	9	3423	9	18	48
BlueBlog	12	2556	7	13	39

Table 1: Performance results for the benchmark run

The results in Table 1 describe the relation between source code size and running time, as well the relation to the precision. As we discussed earlier, we have the ability to tweak the algorithms precision with a Call Graph parameter which states the depth of the call string being remembered by the data structure. Time k relates to a depth of k. We can see that the running time increases rapidly as we increase the precision, somewhere between polynomial to exponential factor.

The results show an increase in running time in relation to code size, close to linear. These running times are also affected by the number of sinks we have located, since each sink is analyzed separately.

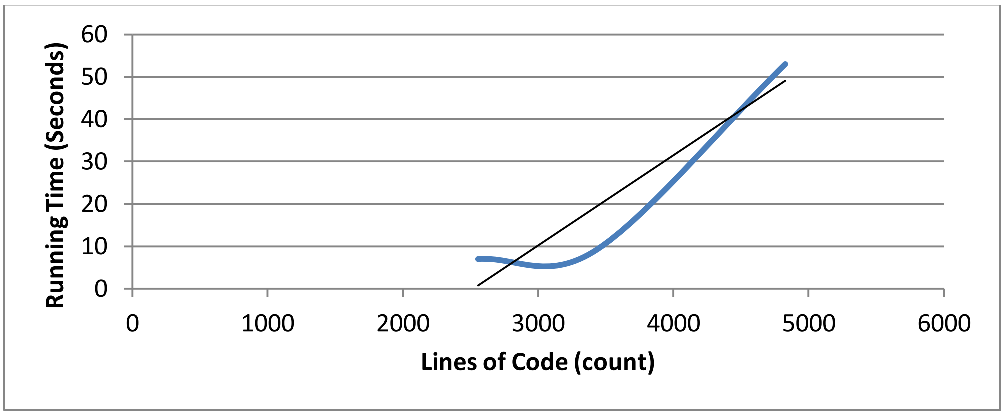


Chart 1: Relation between LOC to the running time of the least precise analysis

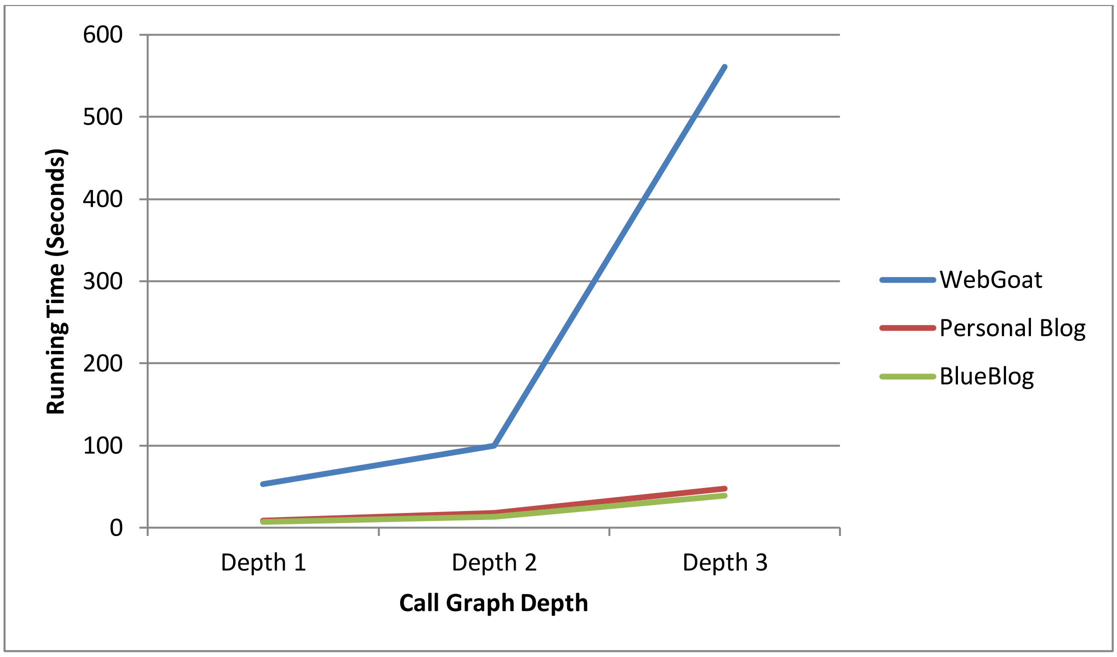


Chart 2: Relation between Call Graph depth To running time of each application

6.3 Fuzzer Integration

In this section we will provide a review of how to use the results we achieved in the experimental run, and discuss the possible improvement it can achieve. The experimental run included 3 applications, testing 4 types of application vulnerabilities. We showed that we can detect the "real" use of each parameter, specifically if it was used in different sensitive resource access. The ability to detect the actual data flow is the foundation to this section.

The following table describes the applications we tested, specifying the number of servlets, total number of parameters, and the number of parameters relevant to each category we checked in the experimental analysis (SQL injection, Command Injection, Path Traversal and HQL Injection).

Application	# Servlets	# Parameters	# SQL	# CMD	#File	#HQL
WebGoat	12	22	10	1	0	0
BlueBlog	1	6	0	0	2	0
Personal Blog	12	29	0	0	0	2

Table 2: Experimental run number summary

According to Table 2, we now have information that can help us improve the focus of a Fuzzer based test run. We can decide to avoid testing the parameters found non-related to the categories we checked, and put more focus on those that are relevant.

To calculate the percentage of tests that can be avoided, we propose a simple testing model. Assuming there is X tests a Fuzzer performs on a given parameter for a given category (SQL, HQL, etc.). Then the number of tests needed to be performed per application is:

$$\#tests_{app} = X(\#parameters)(\#categories)$$

For the tested applications, these are the numbers:

Application	# Servlets	# Parameters	# tests	# actual test	% reduction
WebGoat	12	22	88X	11X	87.5
BlueBlog	1	6	24X	2X	91.6
Personal Blog	12	29	116X	2X	98.2

Table 3: total expected tests

We also added the actual number of tests needed to be performed with our improvement (X tests per parameter found in each category). We can see the percentage of tests reduced is up to **98%.**

6.4 Results Summary

We have managed to show that by analyzing data flow in different web application, we can reduce the possible parameters that a fuzzer should check for a specific vulnerability class. In the case of SQL injection in WebGoat we got a 50% reduction in parameters, and in the case of Command injection we got a reduction of about 95%. In the case of BlueBlog we got a reduction of about 50% in the Path Traversal category. And, in the case of Personal Blog, we have reduced parameters leading to HQL queries in 93%. Furthermore, we can say that there is no need to run sql/command injection tests on the BlueBlog or Personal Blog code.

When analyzing the total results, and no per category, we showed the improvement is much more dramatic. In total we showed a reduction of up to **98%** in the number of tests needed. This is due to the ability to show that some testing categories are irrelevant in different parts of the applications we tested. This shows the true strength of our approach in reducing unneeded tests.

Nevertheless, we have found some limitations during the experimental phase. We have seen that our framework might suffer from false positives as well. This issue can be addressed by a more precise analysis, but at the cost of performance.

These results show that our analysis can be used to reduce the number of tests needed to be run in a standard fuzzing test, or increase the number of tests in the relevant areas provided by our tool.

7 Summary

The problem of web application security vulnerabilities analysis is a huge one. There are many nuances and complications to it. We have reviewed the major work done in the area and showed the complexities of the field. We have shown a method to improve the performance of existing testing mechanism, specifically Fuzz Testing, via the SAF framework. Our framework was built for Java web applications.

The SAF framework relies on widely used Dataflow Static Analysis algorithms with a few changes. The Static Analysis implemented in this work tries to locate the flow of user provided data to sensitive program locations, and to leverage this acquired knowledge for the improvement of Fuzz Testing. We have reduced the number of fuzz tests needed to be run by up to 90%.

In our opinion this work can be extended in various exiting and interesting directions:

- Create integration with one of the existing fuzzers, and influencing its tests with the analysis results shown here.
- Improve analysis precision by new heuristics, tackling the limitations mentioned above.
- Improve analysis performance and scalability

- Learn more info from control flow – control flow graph holds data about program execution flow. Certain input validations can be identified and used to craft better input in order to bypass these input validations. In the previous work section we have shown a few papers that use this information (via symbolic execution), and there is a possibility to integrate these works with ours.
- Handle newer technologies of Java web applications, and new types of threats

The area of application security is increasing every year, and presents numerous challenges for contemporary computer science researches. We hope this work can help others in their efforts to solve some of these new problems.

Appendix A. Web Application Security Vulnerabilities review

Web applications have multiple types of security vulnerabilities, and each vulnerability type has its own special characteristics. We will provide a short review and description of Injection vulnerabilities, for they are the most relevant to our work. The description for these vulnerabilities is taken from the OWASP Top 10 [32].

Injection related bugs provide an attacker the ability to run a command or a query against a backend service provider. For example, SQL injection flaw, provides the ability to execute some SQL command against a service provider, in this case a DB. These vulnerabilities are usually characterized by passing of unchecked user input as a part of a command to the backend service provider. We will review some major injection attack types.

SQL Injection [29]

SQL is a query language used in most modern Databases for the retrieval and update of data. Databases hold the organization's data, and are therefore a very interesting target for different attackers. A SQL injection is the creation and execution of a SQL command that was not intended to run against the data base.

Path Traversal [26]

Path Traversal is a vulnerability type that allows an attacker to follow the directory structure of the application server. This is done by injection path values (like slashes and dots) into a specific parameter, and these values are concatenated to a file path in the application code. This constructed file path is later accessed by the application server, and possibly allowing the attacker to gain information about the server's file system, or even retrieve the contents of these files.

Cross Site Scripting (XSS)

The target of this attack is the browser itself, and not a backend service provider. A mechanism that is vulnerable to XSS allows an attacker to execute a script (Javascript, etc.) inside a victim's browser, as if it was a part of a legitimate website (the vulnerable site). These vulnerabilities look much like other Injection types, perhaps the exception that there is nuance called Persistent XSS in which injected text is first stored in some storage, and presented to the user in a different page than the one it was created on.

Appendix B. Wala Data Structures and Algorithms Review [31]

Wala (T. J. Watson Libraries for Analysis) is a Static Analysis framework promoted by IBM research. Wala is written in Java and is used to analyze different Java/JavaScript applications. Wala provides the data structures and implements the algorithms reviewed in the Static Code Analysis section. We will focus on the different features and tweaks of the major algorithms we used.

Context Sensitive Call Graph [33]

As mentioned in the Prior Work chapter, a call graph represents the caller-callee relations between all the methods of the analyzed code. Wala provides an abstract representation of a Call Graph, with a few implementations behind it. We will discuss the implementation we used which a Context-Sensitive Call Graph, based on call-string context.

A Call Graph node is a method together with its calling context (considering our graph is context sensitive). Let's look at a simple example. Assuming we have to methods, main and foo, and main calls foo. In this case we will get two nodes in our graph and an edge between them, one for the main method (with null context, because no one calls it) and one for foo with the call in main as its calling context. We can complicate this example by saying that main calls foo twice. In this case, there will be two nodes for foo, one for each call from main.

What we have shown in the previous paragraph is how Call Graphs hold the calling context inside a program. This context can grow exponentially if we take into account chains of method calls (main calls foo that calls bar). The information about the context is stored in a "call string" which is a stream of method + instruction (in the relevant method) that represent the calling sequence. For example,

```
Main@25 ; foo@13 ; bar@32
```

This example shows a sequence of three calls, from main to foo, from foo to bar and from bar to the method in the Call Graph node. The numbers represent instruction numbers inside the relevant methods.

As expected, the greater the call string the more complex and time consuming the analysis is. We usually used a one method call string length, and this gave us reasonable results. As mentioned in the WebGoat analysis results, we changed this behavior, by increasing the depth to 4. This gave more precise results (reduced false positives completely), but increased significantly the running time.

Pointer Analysis [35]

Pointer Analysis tries to answer the question to which object a pointer points to. This issue relates heavily to the previous section, since the application's memory content is influenced by the previously run instructions (or the context).

We use a version of Pointer Analysis that uses context sensitive Call Graph, and removed all optimizations to get the most precise result. Given that, the precision of this analysis is influenced by the precision of the Call Graph (as discussed in the previous section.

Bibliography

[1] N. Ayewah, W. Pugh, J. D. Morgenthaler, J. Penix, Y. Zhou. Evaluating Static Analysis Defect Warnings On Production Software. PASTE '07 Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering

[2] W. Chang, B. Streiff, and C. Lin. Efficient and extensible security enforcement using dynamic data flow analysis. In Proceedings of the 15th ACM conference on Computer and communications security (CCS '08)

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

[4] Classes and Objects – Introduction, PHP Manual Language Reference http://www.php.net/manual/en/oop5.intro.php

[5] R. Dechter. Constraint Processing, Morgan Kaufman, 2003

[6] D.Englar et al. A few billion lines of code later: using static analysis to find bugs in the real world, Communications of the ACM, Volume 53 Issue 2, February 2010

[7] V. Ganesh, T. Leek, and M. Rinard. Taint-based directed whitebox fuzzing. In Proceedings of the 31st International Conference on Software Engineering (ICSE '09). IEEE Computer Society, Washington, DC, USA, 474-484. DOI=10.1109/ICSE.2009.5070546 http://dx.doi.org/10.1109/ICSE.2009.5070546

[8] P. Godefroid, M. Y. Levin, D. Molnar. Automated whitebox fuzz testing. 2008.

[9] P. Godefroid, M. Y. Levin, and D. Molnar. SAGE: Whitebox Fuzzing for Security Testing. communications of the acm, march 2012.

[10] Hibernate. http://hibernate.org/

[11] J. E. Hopcroft, R. Motwani, J. D. Ullman. Introduction to Automata Theory, Languages and Computation, Prentice Hall.

[12] D. Hovemeyer and W. Pugh. Finding bugs is easy. In Proceedings of

the Onward! Track of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications, October 2004

[13] Y. Huang et al. Securing Web Application Code by Static Analysis and Runtime Protection. Proceedings of the 13th international conference on World Wide Web

[14] N. Jovanovic , C. Kruegel , E. Kirda. Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities (Short Paper). 2006 IEEE SYMPOSIUM ON SECURITY AND PRIVACY

[15] N. Jovanovic , C. Kruegel , E. Kirda. Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities (Technical Report)

[16] J. A. Kupsch, B. P. Miller. Manual vs. Automated Vulnerability Assessment: A Case Study. First International Workshop on Managing Insider Security Threats (MIST), West Lafayette, IN, USA, June 2009.

[17] W. Landi, B. G. Ryder. Pointer-induced aliasing: a problem classification. POPL '91 Proceedings of the 18th ACM SIGPLAN-SIGACT symposium on Principles of programming languages.

[18] W Landi. Undecidability of static analysis. ACM Letters on Programming Languages and Systems (LOPLAS) Volume 1 Issue 4, Dec. 1992

[19] B. Livshits, Improving Software Security With Precise Static And Runtime Analysis, Ph.D. Thesis, 2006.

[20] B. P. Miller, L. Fredriksen, and B. So. An empirical study of the reliability of unix utilities. Communications of the ACM ,Volume 33 Issue 12, Dec. 1990

[21] B. P. Miller, D. Koski, C. Pheow, L. V. Maganty, R. Murthy, A. Natarajan and J. Steidl. Fuzz revisited: A re-examination of the reliability of UNIX utilities and services. 1995

[22] Oracle Java home page. http://www.oracle.com/technetwork/java/index.html

[23] ORM injection. https://www.owasp.org/index.php/Interpreter_Injection#ORM_Injection

[24] OWASP – Fuzzing. https://www.owasp.org/index.php/Fuzzing

[25] OWASP JBroFuzz – Payloads and Fuzzers. https://www.owasp.org/index.php/OWASP_JBroFuzz_Payloads_and_Fuzzers

[26] OWASP - Path Traversal. https://www.owasp.org/index.php/Path_Traversal.

[27] OWASP Testing Guide Appendix C: Fuzz Vectors https://www.owasp.org/index.php/OWASP_Testing_Guide_Appendix_C:_Fuzz_Vectors

[28] OWASP WebGoat. https://www.owasp.org/index.php/Category:OWASP_WebGoat_Project

[29] SQL injection, http://en.wikipedia.org/wiki/SQL_injection

[30] Stanford SecuriBench. http://suif.stanford.edu/~livshits/securibench/

[31] T.J. Watson Libraries For Analysis (WALA). http://wala.sourceforge.net/wiki/index.php/Main_Page

[32] The OWASP Top 10 – 2013, https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project

[33] Wala Call Graph User Guide. http://wala.sourceforge.net/wiki/index.php/UserGuide:CallGraph

[34] Wala Intermediate Representation User Guide. http://wala.sourceforge.net/wiki/index.php/UserGuide:IR

[35] Wala Pointer Analysis User Guide: http://wala.sourceforge.net/wiki/index.php/UserGuide:PointerAnalysis

[36] M. N. Wegman, F. K. Zadek. Constant propagation with conditional branches. ACM Transactions on Programming Languages and Systems, Volume 13 Issue 2, April 1991

[37] White Hat Website Security Statistics Report, https://www.whitehatsec.com/resources/stats.html

תקציר

בעיות אבטחה הינן אחד מהאיומים הגדולים בימינו על אפליקציות ואתרי אינטרנט. דו"ח שפורסם במאי 2013 [37] מעריך ש - 86% מכל אתרי האינטרנט חוו לפחות פרצת אבטחה משמעותית אחת בשנת 2012. בעיה קשה זו מהווה הזדמנות לחוקרים וחברות לחקור שיטות חדשות לטיפול בבעיות אבטחה אלו.

במהלך השנים האחרונות יצאו פרסומים רבים של טכניקות שונות ומגוונות להתמודדות עם העלייה בבעיות אבטחה באפליקציות מודרניות. טכניקות אלו מתחלקות בעיקר לשתי קטגוריות: סטטיות ודינמיות. שיטות סטטיות משתמשות בניתוח סטטי אשר חוקר את מבנה הקוד ומוצא באגים. שיטות דינמיות מנסות לנתח את התנהגות האפליקציה בזמן ריצה ולזהות חריגות או התנהגות שגויה.

בשנים האחרונות נעשו שיפורים גדולים ומשמעותיים במאמץ לפתור את הבעיה המוזכרת לעיל. ב [12] נוסחה שיטה אשר מבוססת על שיטות מומלצות בקרב מפתחים, על בסיס נסיון עבר (best practices). [14] מספק כלי שנקרא פיקסי, אשר מבוסס על ניתוח זרימת מידע בתוכניות PHP בשימוש בטכניקת ניתוח קוד סטטי. [13] ו[19] מימשו טכניקות ניתוח סטטי עם ניתוח דינמי והגנה דינמית על אפליקציות. טכניקות אלו הראו הצלחה רבה במציאת באגים אמיתיים. עם זאת, שיטות אלו סובלות ברוב המקרים מיחס גבוהה של תוצאות שוא.

בעבודתנו, אנו מנסים להתמודד עם סוגיית תוצאות השוא מכיוון שונה. עבודתנו נסמכת על שיטת בדיקות תוכנה קלאסית שנקראת Fuzz Testing [20]. שיטה זו בודקת אפליקציות בעזרת ייצור קלטים רנדומליים (או חצי רנדומליים) עבור האפליקציה הנבדקת, בנסיון לגרום להתנהגות לא רצויה. אנו מעוניינים בעבודתנו להיעזר במידע שנלמד על מבנה האפליקציה בעזרת ניתוח סטטי ולשפר את היכולות של תהליך הFuzzing.

היתרון של שיטתנו היא היכולת לשפר את הביצועים והמיקוד של שיטת הFuzzing. כלי Fuzzing פשוט מכיל מספר קטגוריות של ייצור קלטים, עבור מקרי בדיקה שונים. בכל מקרה, הכלי יכול לייצר אינספור קלטים עבור שלב הבדיקה. מידע על מבנה התוכנה הנבדקת יכול לעזור לצמצם את מספר הבדיקות שיש להריץ, או לחלופין להגדיל את הבדיקות הרלוונטיות. לטובת למידת מבנה התוכנות הנבדקות כתבנו את תשתית SAF.

תשתית SAF מיועדת לניתוח של אפליקציות אינטרנט הכתובות בג'אווה. מטרת התשתית היא ללמוד את זרימת המידע בתוכניות השונות, ספציפית זרימת קלט משתמש, ולהשתמש במידע זה לצורך שיפור כלי Fuzzing. הרעיון המרכזי הוא לאתר "בורות" בתוכנית (כמו הרצת שאילתא למול מבנה נתונים), אשר אליהם מידע זורם, ולמצוא מסלולי זרימת מידע המביאים

מידע מהמשתמש לאותן נקודות. ממצאים אלו יכולים לספק סיווג עבור כל פרמטר שהמשתמש יכול להזין, ולייצג איזה וקטור תקיפה ייתכן ואפשר לנצל דרך אותו פרמטר. בעזרת מידע זה ניתן לשפר את הביצועים והמיקוד של כלי Fuzzing.

במסגרת מחקרנו ביצענו בדיקה והרצנו את SAF על שלושה פרוייקטי קוד פתוח, אחד חינוכי ושתי אפליקציות בלוג. אפליקציות אלו מבצעות גישה למבני נתונים, גישה למערכת קבצים וכדומה. אנו הצלחנו למצוא מסלולי מידע לנקודות רגישות בתוכנית, ובאופן פוטנציאלי הצלחנו להקטין בעד כדי 90% את כמות הפרמטרים שיש לבדוק עבור קטגוריה ספציפית (שאילתות SQL למשל).

המרכז הבינתחומי בהרצליה
בית הספר אפי ארזי למדעי המחשב

שיפור יכולות פאזינג בעזרת ניתוח קוד סטטי

סיכום עבודת גמר המוגשת כמילוי חלק מהדרישות לקראת
תואר מוסמך במסלול מחקרי במדעי המחשב

על ידי מוטי כהן

העבודה בוצעה בהנחיית דוד מובשוביץ

מאי 2014