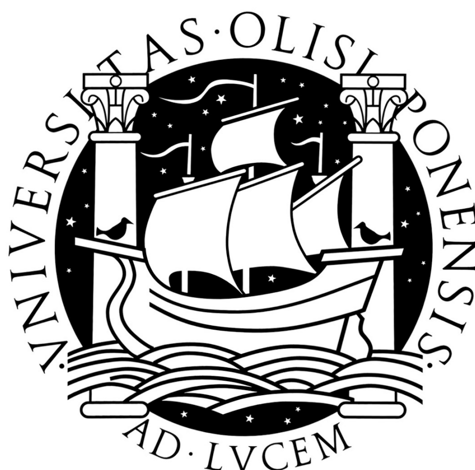
UNIVERSIDADE DE LISBOA
FACULDADE DE CIÊNCIAS
DEPARTAMENTO DE INFORMÁTICA



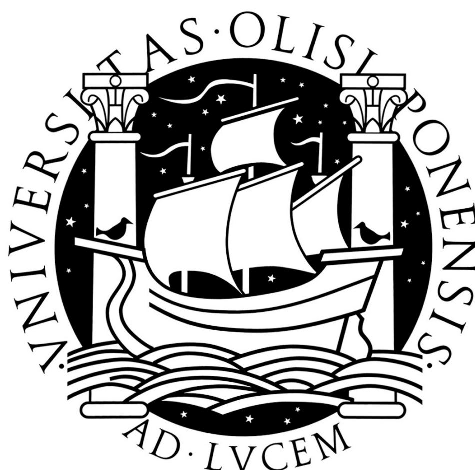
Realistic Vulnerability Injections in PHP Web Applications

Francisco José Marques Vieira

MESTRADO EM SEGURANÇA INFORMÁTICA

2011

UNIVERSIDADE DE LISBOA
FACULDADE DE CIÊNCIAS
DEPARTAMENTO DE INFORMÁTICA



Realistic Vulnerability Injections in PHP Web Applications

Francisco José Marques Vieira

Orientador
Nuno Ferreira Neves

MESTRADO EM SEGURANÇA INFORMÁTICA

2011

Resumo

A injecção de vulnerabilidades é uma área que recebeu relativamente pouca atenção da comunidade científica, provavelmente por o seu objectivo ser aparentemente contrário ao propósito de fazer as aplicações mais seguras. Esta pode no entanto ser usada em variadas áreas que a fazem merecedora de ser investigada, tais como a criação automática de exemplos educacionais de código vulnerável, testar mecanismos de segurança em profundidade, e a avaliação de detectores de vulnerabilidades e de equipas de segurança.

Esta tese propõe uma arquitectura para uma ferramenta de injecção de vulnerabilidades genérica que permite a inserção de vulnerabilidades num programa, e aproveita a vasta investigação existente sobre detecção de vulnerabilidades. A arquitectura é também extensível, suportando a adição de novas vulnerabilidades para serem injectadas.

Foi também implementado e avaliado um protótipo baseado nesta arquitectura por forma a perceber se a arquitectura era implementável. O protótipo consegue injectar a maior parte das vulnerabilidades da class taint-style, desde que em aplicações web desenvolvidas em PHP. Esta tese contém também um estudo sobre as vulnerabilidades presentes nas últimas versões de algumas aplicações PHP bem conhecidas, que permite perceber quais os tipos de vulnerabilidade mais comuns. Este estudo conclui que as vulnerabilidades que o protótipo permite já incluem a maioria das vulnerabilidades que aparecem habitualmente em aplicações PHP.

Finalmente, várias aplicações PHP foram usadas na avaliação. O protótipo foi usado para injectar diversas vulnerabilidades sobre estas aplicações, e após isso as injecções foram analisadas à mão para verificar se uma vulnerabilidade tinha sido criada ou não. Os resultados mostram que o protótipo consegue não só injectar uma grande quantidade de vulnerabilidades mas também que estas são atacáveis e realistas.

Palavras-chave: Vulnerabilidades, Injecção, PHP, Análise Estática

Abstract

Vulnerability injection is a field that has received relatively little attention by the research community, probably because its objective is apparently contrary to the purpose of making applications more secure. It can however be used in a variety of areas that make it worthy of research, such as the automatic creation of educational examples of vulnerable code, testing defense in depth mechanisms, and the evaluation of both vulnerability scanners and security teams.

This thesis proposes an architecture for a generic vulnerability injection tool that allows the insertion of vulnerabilities in a program, and leverages from the vast work available on vulnerability detection. The architecture is also extensible, supporting the addition of new vulnerabilities to inject.

A prototype implementing the architecture was developed and evaluated to analyze the feasibility of the architecture. The prototype targets PHP web applications, and is able to inject most taint-style type vulnerabilities. The thesis also contains a study on the vulnerabilities present in the latest versions of some well known PHP applications, providing a better understanding of which are the most common types of vulnerabilities. This study shows that the vulnerabilities that the prototype is able to inject already includes the majority of the vulnerabilities that appear in PHP web applications.

Finally, several PHP applications were used in the evaluation. These were subject to injections using the prototype, after which they were analyzed by hand to see whether a vulnerability was created or not. The results show that the prototype can not only inject a great amount of vulnerabilities but that they are actually attackable.

Keywords: Vulnerabilities, Injection, PHP, Static analysis

Acknowledgments

There are several people that I want to thank for their help and support. Without them this thesis would probably never have come to be. I want to thank...

My advisor, Professor Nuno Neves, for his guidance and patience throughout this whole adventure that was making this thesis and for his diligence when reviewing this thesis.

My company, Portugal Telecom, for supporting me financially throughout these 16 months

My boss, Eduardo Pinto, for his flexibility that allowed me to finish the thesis even after I started working.

Miguel Correia, which managed to constantly pressure me to finish the thesis while at the same time being lenient each time a deadline was missed.

My colleagues from the master, which made these 16 months so much easier to bear with their great sense of humor and companionship.

My friends, which have been neglected for too long and to whom I owe many hours of conversations and fun.

And last but not least, my family and my girlfriend Sofia, who have heard me complain more in 16 month than I ever had, and who miss me the most when I fail to be with them.

Lisboa, April 2011

Contents

1 Introduction **1**

1.1 Motivation 1

1.2 Design Decisions 2

1.2.1 Static Analysis vs Dynamic 3

1.2.1.1 Disadvantages 3

1.2.1.2 Advantages 3

1.2.2 Injection at Source Code vs Binary 3

1.2.2.1 Disadvantages 3

1.2.2.2 Advantages 4

1.2.3 PHP vs Other Server-Side Programming Language 4

1.2.3.1 Disadvantages 4

1.2.3.2 Advantages 5

1.3 Contributions 5

1.4 Document Structure 5

2 Related Work **7**

2.1 Vulnerability Injection 7

2.2 Fault Injection 8

2.3 Static Analysis 9

2.3.1 Front End 9

2.3.2 Middle end 10

2.3.3 Static Analysis on PHP 13

2.4 Vulnerability Detection Tools 13

2.4.1 RIPS 13

2.4.2 Pixy 14

3 **Vulnerabilities Distribution** **17**
3.1 Applications Analyzed 17
3.2 Analysis Methodology 18
3.3 Results 22

4 **Architecture** **25**
4.1 Goals 25
4.2 Architecture 26
4.2.1 Main module 27
4.2.2 Front-end 28
4.2.3 Middle-end 29
4.2.4 Vulnerability Location Seeker 29
4.2.5 Vulnerability Injector 31
4.2.6 Typical Work Flow 32
4.3 Location Plugins 33
4.3.1 Taint-Style Vulnerabilities 39
4.3.2 Cross-Site Scripting 45
4.3.3 SQL Injection 49

5 **Prototype Implementation & Evaluation** **53**
5.1 Prototype 53
5.2 Evaluation 56

6 **Conclusions and Future Work** **61**
6.1 Conclusion 61
6.2 Future Work 61

Bibliography **63**

List of Figures

3.1 Vulnerabilities Distribution 22
3.2 Vulnerabilities Per Application 24

4.1 Architecture for a generic vulnerability injection tool 27
4.2 Vulnerability Location Seeker Architecture 34

5.1 Pixy architecture plus modifications for Injection Tool 54
5.2 Vulnerabilities injected 57
5.3 Difference of the vulnerabilities detected by RIPS 58

List of Tables

3.1 Versions analyzed per application 19

5.1 Vulnerabilities detected by RIPS 58

Chapter 1

Introduction

1.1 Motivation

In a world growing every day more dependent on computer systems, we are slowly becoming aware of the need for security in applications to prevent attacks that could result in the loss of material, money or even human lives.

Web applications in particular are being used today as front ends to many security critical systems (e.g., home banking and e-commerce), but due to their high exposure, they are particularly susceptible to being heavily attacked. This means that they require special care to make them secure and resilient against these threats.

Security teams and vulnerability scanners are some of the approaches used to eliminate the flaws that weaken the applications, the vulnerabilities. However, without a systematic way of evaluating them, it is difficult to choose the right security tool for the job, or find out what knowledge the security team is lacking.

Evaluating vulnerability scanners might be done by comparing the results from each other, to see which tool finds more vulnerabilities with less false positives. This however only gives a result relative to the other vulnerability scanners, meaning that if all vulnerability scanners are bad the this method will not produce meaningul results. Evaluating a security team is even harder since most companies do not have the resources required to employ more than one security team, and even if they have, they probably would not want to have them doing the same work just for the sake of comparing the results.

The best way to evaluate both security teams and vulnerability scanners is to have them work over an application whose vulnerabilities are already known, in order to be able to check not only which types of vulnerabilities they find best but also which ones they failed to find at all. While this can be done by using some older version of an application, with some known vulnerabilities, finding a version that has just the right vulnerabilities for the evaluation might be very difficult. Therefore, a method is necessary to give to the evaluator the control over which vulnerabilities are inserted in the test application, to ensure that the evaluations are effective and efficient.

This thesis proposes a solution to this problem by proposing a vulnerability injection tool architecture. This tool is be able to inject vulnerabilities into an application that are both attackable and realistic, meaning that it is possible to use the injector to create a test application whose vulnerabilities are chosen by that evaluator.

A vulnerability injection tool can also be used to estimate the number of vulnerabilities that are left to correct in an application after it has already been tested [1]. Assume we have an application that not yet been tested for vulnerabilities and thus has an unknown amount N of vulnerabilities. If I extra vulnerabilities are injected in the application before it is tested, and after it is tested i vulnerabilities from the ones previously injected are found plus n vulnerabilities that were originally in the application, we can estimate N by solving the formula $N/n = I/i$. The assumption behind this reasoning is that the inserted vulnerabilities are a good representation of real vulnerabilities. This means that the injections should be based on an average distribution of the vulnerabilities in applications, or even better, on data more specific to the specific case, if available.

Another interesting use for a vulnerability injection tool would be to create educational examples of vulnerabilities. The most well known example of this is WebGoat [2]. There are some advantages to being able to create educational examples on the fly instead of relying on third party educational examples. First, it would be easier to create specific examples since all that is needed is a working application to inject vulnerabilities. Second, it would also be possible to create examples that are more suited to the person being evaluated, for example by only injecting vulnerabilities which the person has difficulty in finding, or by employing application similar to the ones the person works with, making the learning process more helpful.

One last example of how could a vulnerability injection tool be used for is to test security in depth. The trick here is to inject vulnerabilities in all parts but one of the security in depth mechanism, and them bombard the security in depth mechanism with attacks. If attacks go through, it proves that the component that was not injected with vulnerabilities is not good enough to guarantee a secure environment and therefore should be improved to work as a proper component of the security in depth mechanism.

A vulnerability injection tool is thus more powerful than one would think at first, and this thesis will show what is required for the creation of such a tool. Before that, however, we will discuss some of the design decisions that had to be taken in relation to some key characteristics of the vulnerability injection tool.

1.2 Design Decisions

Each decision is divided in two parts, with the first part explaining potential disadvantages that one choice would have when compared to the alternative, and the second part explaining its advantages and trying to help the reader understand why certain design decisions were taken.

1.2.1 Static Analysis vs Dynamic

1.2.1.1 Disadvantages

- Dynamic analysis might help to understand the logic of the application and its protocols, since the details of how the application works are hidden and what is analyzed is what the application does, not how it does it. This could be a useful complement to static analysis because it could help to understand which parts of the code are actually reachable from the outside, allowing for more realistic injections.
- Dynamic analysis is agnostic to the language of the application and the compiler, since it only looks at the output of the application and tries to infer information from it. Static analysis must depend on one of these at least.
- Static analysis requires access to the code, which might not be available.

1.2.1.2 Advantages

- Dynamic analysis might be very useful when trying to detect vulnerabilities, but when the goal is to inject vulnerabilities one also needs to modify the source code. Therefore, analyzing the code is not only a good idea but even required.
- Static analysis can be used in code that cannot be compiled, like when it is being developed.
- Since static analysis does not require an interaction with an actual running application, there is no risk of destroying production data.
- Static analysis allows the evaluation of every code path, including paths that would be difficult to find using only dynamic analysis.

1.2.2 Injection at Source Code vs Binary

1.2.2.1 Disadvantages

- Source code is just a language that the programmer uses to tell the computer what the application does, but the end result depends completely on the executable generated by the compiler. In order to inject realistic vulnerabilities one requires deep analysis of the code, but only analyzing the binary code can allow full understanding of what will run, without relying in the abstractions from the source code or the peculiarities of the compiler.
- Source code analysis is language dependent, which means that a different injection tool must be built for each language.
- There are times when only the binary code of an application is available making source code analysis impossible.

- All the abstractions in source code not only make it farther from the executable that runs in the machine, but also creates complexity with the introduction of concepts such as functions, objects, etc.. These concepts are absent in the binary code and force the injection tool to be more intelligent in order to understand and be able to analyze the source code.

1.2.2.2 Advantages

- Only by making the injections at the source code is it possible to create educational examples, plus all uses that require security teams to find the injected vulnerabilities, such as evaluating a team's efficiency or estimating the vulnerabilities left after an audit.

- Since the injections are done in the source code, they are independent of the machine where the code is run or the compiler that will be used.

- Injections in the source can be more realistic than vulnerabilities in the binary code, since the source code is where vulnerabilities are usually introduced. One more reason for this is the fact that languages might have mechanisms to avoid that some vulnerabilities which do not exist at the level of the binary code. Therefore, if vulnerabilities are inserted in the binary code, they could be unrealistic because they would never occur in real code.

- Doing the injection at the source level allows for both the vulnerable code and binary to be used, since it is possible to just compile the injected code. It is not possible to obtain realistic source code from binary code, so if injections were made in the binary code, it would not be possible to obtain the vulnerable source.

- Source Code injection can be developed for all languages, including interpreted languages.

- It is possible to inject vulnerabilities without compiling the code, which is not only more practical but might even be the only choice if the code is still being developed.

1.2.3 PHP vs Other Server-Side Programming Language

1.2.3.1 Disadvantages

- PHP is as dynamic as a language can get. While this might make PHP more powerful and interesting to developers, it makes its analysis much harder. Some features that have this effect are the possibility to include source files at run-time, which make it very difficult to understand the code that will actually run. Additionally, the fact that PHP is weakly typed requires type inference to be performed, and the way it allows for aliases to be created dynamically also requires alias analysis. These were only some simple examples, many more can be found in [3].

1.2.3.2 Advantages

- PHP is by far the most used server-side language in web applications [4]. In order to make this thesis as useful as possible, choosing a relevant language was a decisive factor.
- It is easy to learn. This was very useful because the author of the thesis did not know any server-side language and opting for PHP was a plus.

1.3 Contributions

The main contributions of this work are the following:

- A study on the vulnerabilities of seven well known PHP applications that were patched in the past three years. This study provides a better understanding of what are currently the most common vulnerabilities in web applications.
- An architecture for a generic vulnerability injection tool that allows the insertion of vulnerabilities in a program. This architecture leverages from the vast work available on vulnerability detection. It can also be extended through the use of plugins, making it possible to add new vulnerabilities to inject.
- An overview of the class of taint-style vulnerabilities (the most common type of vulnerabilities according to the previously mentioned study) and of its two most common instances, Cross-Site Scripting and SQL Injection. Examples of plugins for the architecture are also presented for the class and its instances.
- A prototype implementing the architecture. This prototype is evaluated by using it to inject vulnerabilities on four applications, verifying that the injected vulnerabilities are indeed attackable, and verifying that an open source vulnerability scanner cannot detect some of them.

1.4 Document Structure

Chapter 2 describes the related work in Vulnerability Injection, gives a background review on Fault Injection and Static Analysis, and presents two vulnerability detection tools that are used in the rest of the thesis.

A study on the distribution of vulnerabilities in web applications is presented in Chapter 3, showing what are the most common vulnerabilities that were found.

In Chapter 4, a generic architecture for a vulnerability injection tool is presented, and some examples are shown for parts of the architecture. This includes a description of the Cross-Site Scripting and SQL Injection vulnerabilities and their preventions.

Chapter 5 describes the implementation of a prototype that tries to follow most of the previously mentioned architecture, and the results of evaluating the prototype against four PHP scripts.

Finally, Chapter 6 concludes the thesis and provides some pointers for future work that might be done on this topic.

Chapter 2

Related Work

This chapter provides an overview on current research in Vulnerability Injection, and gives a background review of Fault Injection and Static Analysis that are helpful to understand the rest of this thesis. It finishes with the presentation of two vulnerability scanners that will be used for building the prototype from Chapter 5.

2.1 Vulnerability Injection

There is not much work on vulnerability injection in our current day. The reasons for this might have to do with the fact that it looks counterproductive to the end goal of making software more secure and also that it is as hard (if not harder) as the opposite problem of detecting vulnerabilities.

This might explain why only one team, Fonseca et. al., has focused on this exact problem. Fonseca et. al. build upon this concept in two different works: in [5] they present the concept and delve deeper into it in [6]. They implement the concept by building a vulnerability injection tool for PHP that injects SQL Injection vulnerabilities.

The tool receives a PHP source file as input and runs it through three components in sequence. First a dependency builder is executed to discover which code is included by the PHP include calls in the input file. This is required in order to find out exactly what code is run when the file is called.

Next it runs a variable analyzer responsible for listing the dependencies between the variables, i.e., for each variable it finds all variables whose value can have an influence on it. This is used to find out which variables can actually influence SQL query strings, which in turn allows discarding all other variables since they are not useful to inject the vulnerability.

Finally a vulnerability injector is run to actually insert the vulnerability in the source code. This last component uses the concept of Vulnerability Operators in order to decide where and how to inject the vulnerabilities. A Vulnerability Operator is composed of a Location Pattern, which defines rules about where can the vulnerability be injected, and a Vulnerability Code Change which defines the modification required to create the vulnerability.

While their work provides a good insight into how to build a realistic vulnerability injection tool, the fact that it tackles a single type of vulnerability makes it limited for practical use.

Besides this, the concept of Vulnerability Operator is very informal and not very easy to apply as there is a lack of separation of concerns of the Location Patterns and the Vulnerability Code Changes. This can be seen in the examples in [5] where the descriptions of each Vulnerability Code Change also contain some conditions for the injection that are very similar in nature to the Location Patterns rules.

2.2 Fault Injection

Fault injection has been used since the 1970s [7] to test the dependability of fault tolerant systems. By injecting faults into specific components of a system it was possible to verify whether the system could tolerate faults in those components, and it was also possible to learn the impact of the eventual failures.

The first approach for fault injection was to inject faults directly in the hardware. These were used mostly to test the tolerance of the hardware to occasional failures and to evaluate the dependability of systems where high time resolution is required.

With time, this technique was found to be quite limited and expensive and thus it was replaced by software implemented fault injection (SWIFI), which was nearly as powerful and much cheaper. Besides these advantages, with SWIFI it was possible to evaluate the dependability of specific applications, something that was not feasible with hardware injection. SWIFI can be divided into two subcategories, runtime injections and compile-time injections.

With runtime injections, faults are injected during the execution of the software, through the use of timer activated interrupts, traps which activate on certain events or even code injection. Runtime injections allow for fault injections to occur on-the-fly, making it possible to modify data that is being used by the system and allowing for decisions on the injection to be made based on the current state of the application. Most types of runtime injections also have the advantage of not requiring the source code to be available or in the case it is available it still has the advantage that it does not require recompiling the code for each injection.

Compile-time injections on the other hand are done by modifying the source or assembly code before the program is loaded and executed. They can emulate permanent faults while introducing very little perturbation to the execution of the application because the faults are injected before the code is run. Compile-time injections are also the best way to emulate realistic software faults (bugs) for reasons already stated in Section 1.2.2.2.

Vulnerability injections in the source code are thus a particular case of compile-time injections, where its additional requirements are that the software faults (bugs) introduced must make the application vulnerable to an attack that violates its security requirements, while not affecting the apparent behavior of the application under normal operating circumstances. This requires knowing, or at least guessing based on data gathered from other applications, what are both the security requirements and normal behavior of the target application, which makes vulnerability injection harder to implement correctly than compile-time injection.

2.3 Static Analysis

Static Analysis is a form of analysis of computer software where the code is analyzed without being executed. It can be used to gather a lot of information about the code, from defects or bugs, unreachable code and unused variables, whether it adheres to good programming practices, software metrics, and can even be used to formally prove whether the application has some given properties.

While it can also be applied on binary code, it is most commonly applied to source code. In fact, compilers use it to do their job and thus are a good model to verify what and how is static analysis done [8]. Compilers are usually composed of three main components: the front end where source code is parsed, its syntax and semantics is validated and is transformed into an intermediate representation ready for further analysis; the middle end where the code is subject to static analysis in order to be optimized; and the back end where the code will be finally translated into the output language. Static analysis tools typically have a similar work flow to the front and middle end except for the optimization part which is where tools diverge according to their purpose, and thus these two components are worth a deeper look at.

2.3.1 Front End

The front end is usually divided in three main phases.

The first phase is usually lexical analysis which converts the characters from the code into tokens. Each token has a type and value which is defined by regular expressions. The type of a token can be an integer, identifier, open_parenthesis, close_parenthesis, equal_symbol, semicolon, etc. Its value is the actual value of the token, like 123 for an integer or “abc” for a identifier. While all tokens have a value, for some types it might be empty, such as in open_parenthesis or semicolons. Also, since tokens are defined by regular expressions, types are limited to what can be described by them, so while it might be possible to categorize a token as a variable in PHP due to the $ that variables have at the beginning, in Java that classification must be done at a latter phase of the analysis. The output of this phase is thus a sequence of tokens that are latter subject to syntactic analysis.

Syntactic analysis or parsing is the next phase and transforms the tokens resulting from the lexical analysis into an internal representation according to the programming language’s context-free grammar [9]. Parsing determines thus if the sequence of tokens received, and thus the initial source code, can be derived and how from the start symbol of the context-free grammar. If it finds out that it cannot be derived then it means there is a syntax error in the source code and it can tell where the error is. If there are no syntax errors then a syntax tree is derived, which is just a tree data structure that represents the original code’s syntax. This might be done either by representing directly how the code is derived from the grammar by having the start symbol of the grammar as the root of the tree and the tokens as leaves, in which case it is called a concrete syntax tree, or instead all the extra information that is useful for parsing but useless for latter analysis is discarded, e.g., semicolons to mark the end of a sentence, resulting in an abstract syntax tree. Parsing can then catch many syntax errors such as unmatched parenthesis, malformed expressions, missing semicolons,

etc., but due to the fact that context-free grammars are limited in that they cannot remember the presence of constructs over arbitrary long inputs, there are still some errors that are usually seen violating the syntax of the language that syntactic analysis cannot catch.

In order to catch the last of the syntax errors another phase is needed, semantic analysis. It is responsible for using the syntax tree to perform a series of checks, such as checking whether types being used in functions and expressions match the ones expected, whether identifiers being used have been previously declared and whether it was done only once, whether reserved keywords are not used wrongly, whether the number of arguments when a function is called is correct, etc. It is also in this phase that the symbol table and the attribute grammar are created.

2.3.2 Middle end

The middle end is where most of the static analysis occurs and is thus the most interesting component of the compiler. Many different types of analysis can be realized and many artifacts can be produced by the middle end so only the ones relevant to this work will be discussed in the next paragraphs.

A very important artifact is the control flow graph (CFG). A CFG is a graph where each node represents a basic block, a sequence of instructions in the code that is always run in sequence, i.e., has only one jump target, at the beginning, and only one jump, at the end. Edges in a CFG represent possible jumps in the control, i.e., if there is an edge between basic blocks A and B, then there is a jump at the end of block A that leads to block B. There are also two special nodes in a CFG, the entry node which corresponds to the start of the program and the exit node that corresponds to the end of the program. CFGs are built through control flow analysis and are used by many other types of analysis, namely data-flow analysis.

A call graph is similar to a CFG in the sense that it also represents the control flow of the application, but while a CFG node represent a basic block, a call graph node represents a procedure and an edge represents a call between procedures. Call graphs present thus a higher level look at the control flow of the application which makes them easier than CFGs to understand by humans, allowing for manual analysis in search for anomalies. They are also used in static analysis, for example to search for unused procedures, or to detect recursion during the analysis to prevent it from falling into an infinite loop.

Data-flow analysis is a general type of analysis whose purpose is to determine at each point in the code all the possible states. This includes but is not restricted to variable's values; data-flow analysis can be applied to any piece of information that can be propagated throughout the code and changes depending with the point in the code being analyzed. It works at the level of basic blocks and uses the control flow graph to decide where the state should be propagated to. A data-flow analysis must always define a transfer function and a join operation. The transfer function is applied to the basic blocks and transforms the input state into an output state while the join operation gather all output states from the predecessors of a basic block and joins them into a single input state. Assuming a default state for the entry block, as long as the transfer function and join operation were well chosen, it is possible to propagate the state in order to find all possible

states at each basic block. Depending on the type of data-flow analysis being made, it might be necessary to make transfer functions and join operations that are only approximations of the real application's behavior in order to avoid incurring into state explosion or into an analysis that fails to terminate.

Taint analysis is one of the easiest to understand variants of data-flow analysis. Its purpose is to analyze the points in the code where variables are tainted or untainted. Being tainted might depend on the context but usually it means that the value is derived from user input whether directly or not. The transfer function will define which operations create tainted variables, such as calls that get user input, which operations transfer the taint flag, such as concatenation or assignments of tainted variables, and which clear the taint flag, such as sanitization functions or assignments of untainted variables. The join operation will define what to do when at a merge point a variable might have different taint values. Usually a conservative approach is taken here and if a variable might get to that point tainted through at least one path it is assumed to be tainted from there on. Section 4.3.1 describes taint-analysis in more detail.

Another important variant of data-flow analysis is reaching definitions. Reaching definitions is used to generate use-def and def-use chains. A use-def chain is a data structure that contains all definitions of a variable that can reach a given use of the same variable without intervening definitions. A def-use chain is the exact opposite data structure; it contains all uses of a variable that are reachable from a given definition. By creating use-def and def-use chains for every use and definition of each variable, it is possible to determine which definitions may reach a point in the code. This is used in many types of analysis, such as constant propagation, where variables that can only take a constant value are replaced by it, or to transform code into the static single assignment form, where each variable is assigned only once, which makes other analyses much easier. It can also be used as a basis for data dependence analysis whose purpose is to define for each variable which other variables its value might depend upon, which in turn can be used to perform taint analysis.

A last type of analysis which has great importance is alias analysis. Alias analysis tries to find out whether there are memory locations that can be accessed in more than one way, i.e., whether there are aliases of the pointer to that memory location. This happens in languages like C with pointers or PHP with variable aliasing. Alias analysis determines the alias state for variable pairs at each point in the code. The alias state can be non-alias in case the analysis can safely conclude that the two variables are not alias, must-alias if it can safely conclude that the two variables are alias, and may-alias if it cannot conclude anything. Alias analysis is a great help for other types of analysis such as constant propagation since the behavior of the application might change completely if aliases are taken into account, i.e., attribution of a value to a variable no longer affects only that variable but affects all its aliases also.

In fact, although these types of analysis have clear and separate goals, each one contributes with a little more knowledge about the way the program works, which in turn helps other analyses return more precise results, which in turn allow for a re-analysis with that new knowledge and so on. This is the reason that most static analysis tools prefer to do most analyses in parallel. Variants of data-flow analysis in particular are a good choice for this since the propagation is based on the control flow graph and thus is the same for every variant, all that is required is that all transfer functions and join operators are merged together, which results in a single but much more powerful analysis.

Some last concepts that should be mentioned are related to properties that different analysis can have. An analysis can be:

Flow-sensitive A flow-sensitive analysis takes the order of the program statements into account. A flow-insensitive analysis has typically a better performance but can only give answer of the type "something is true somewhere in the program" while a flow-sensitive analysis could say "something is true between lines 10 and 20". As an example, data-flow analysis is inherently flow-sensitive while alias analysis might not be.

Path-sensitive An analysis is path-sensitive if information is propagated differently depending on branches taken. For example, assume the following example:

Listing 2.1: Example of a program where path-sensitivity is required for correct analysis

```
if (x < 10) {
    flag = 1;
} else {
    flag = 0;
}
if (flag && x >= 10) {
    //do something
}
```

If the first `if` branch is taken, then after the condition we have `x < 10` and `flag = 1`, while if the `else` branch is taken we have `x >= 10` and `flag = 0`. This makes it obvious that the second `if` branch will never occur.

This is how a path-sensitive analysis works, but if the analysis was path-insensitive, at most the analysis would be able to say that after the first condition, `x < 10` or `x >= 10` and `flag = 1` or `flag = 0`, which would make it possible for the second condition to occur. The problem with path-sensitive analysis is that unless simplifications are made, the number of paths will grow exponentially with the number of branches, e.g., a program with 10 `if` conditions will have 2^{10} different paths, making path-sensitive analysis unusable.

Interprocedural An interprocedural analysis is one that performs dataflow analysis between functions as opposed to intraprocedural analysis where the analysis is only made within a single function. While intraprocedural analysis is much more limited since it only analyzes functions independently, it is also much easier to implement since interprocedural analysis requires constant propagation and side effect analysis between function calls, which can be more complicated if the language has runtime polymorphism and even worse if it allows for duck-typing. Besides, interprocedural analysis requires that most or all of the program is available for analysis, something that might not be true if third party libraries are used.

Context-sensitive Context-sensitive analysis is a type of interprocedural analysis in which the analysis of a function is made taking into account the context in which it is called. This is the opposite of what happens in a context-insensitive analysis, where a function is analyzed independently and its analysis is propagated equally to all calls of the function, being thus

oblivious to the context of the call. Again, while a context-sensitive analysis is able to deliver a much more precise result, a context-insensitive analysis has better performance since it only needs to analyze each function once.

2.3.3 Static Analysis on PHP

Static analysis on PHP is quite challenging due to the dynamic nature of PHP. Many works have tackled these challenges [10, 11, 12], but Biggar et al. [3] provides the most comprehensive explanation on both the behavior of PHP, the challenges it brings to static analysis, and how to solve them. Biggar starts with a description of the features of PHP that make it so difficult to analyze, such as its latent, weak and dynamic typing, the fact that it allows source includes and code evaluation at runtime, duck-typed objects, run-time aliasing, etc. A description of the language behavior in greater detail follows which shows how the PHP language implements its features behind the scenes. This knowledge of the mechanics of PHP is then used to create a model of the PHP language that is loyal to most of the language's behavior while keeping it simple and scalable. Finally, Biggar describes how to perform static analysis on PHP using this model. It does it by simulating a symbolic execution of the program one statement at a time while applying many of the analyses described in at last Section 2.3.2, such as type analysis, literal analysis, alias analysis and reaching definitions, which allow it to resolve branch statements and model types and references through the complete program. It applies the results of its analysis to the phc ahead-of-time compiler in order to boost its optimization process.

2.4 Vulnerability Detection Tools

While the purpose of vulnerability detection is the complete opposite of what this work tries to achieve, tools that try to detect vulnerabilities statically have many characteristics in common with a potential vulnerability injection tool, mainly related to the static analysis that it needs to perform and to the vulnerability modeling that is required both to detect vulnerabilities or to detect potential injection locations. It is thus interesting to analyze tools that perform vulnerability detection on PHP.

2.4.1 RIPS

RIPS [13] is written in PHP and thus requires setting up a local web server in order to use it. Once that step is done it can be controlled completely using a practical web interface that allows scanning files for vulnerabilities while customizing the verbosity level, the vulnerabilities to analyze, and even the code style in which results are presented.

It uses the PHP built-in tokenizer extension to split the code into tokens which are then parsed sequentially. Some tokens are attributed with special meaning, such as file inclusions, functions, classes, return keywords, variables, curly braces, exit or throw keywords and function calls, and thus, each time one is found while parsing, it performs some action with it such as including the

tokens from the included file, adding the variable to a list of declared variables, adding a variable as a dependency of another, or changing the variables context when finding curly braces.

RIPS models vulnerabilities using configuration files that define what are the potential vulnerable functions (PVFs), which parameters of these functions are vulnerable and what are the functions that can sanitize user input to be used in these parameters. It also defines which variables and functions might return user input.

In order to find the vulnerabilities, RIPS performs taint analysis by checking whether each function call is a PVF, and if it is, going to each vulnerable parameter and start going up their dependency tree until either user input is found which results in a vulnerability warning, or a sanitization function is found which stops the search for user input on that dependency branch. If none of these are found when all dependencies have been analyzed, RIPS assumes there is no vulnerability.

RIPS contains a large library of about 189 PVFs, for many types of vulnerabilities, such as Cross-Site Scripting, SQL Injection, Code Evaluation, File Inclusion, File Disclosure, File Manipulation, Command Execution, XPath Injection, LDAP Injection, Connection Handling and also vulnerabilities related to Flow Control. This makes RIPS very interesting as it can detect a large range of vulnerabilities. On the other hand, RIPS has several severe limitations that make it less interesting. The fact that it parses the tokens sequentially and only once is one of the most glaring limitations, since it assumes that every function or class being used has been defined previously in the code, which might not be true since PHP allows for definitions to appear anywhere in the code. In fact, RIPS assumes good coding practices, since the way RIPS parses the code through tokenization allows it to make the analysis even if the code is broken; some tokens such as parenthesis are even ignored completely. While this can be seen as an advantage, it reveals the lack of analysis that is done by RIPS and is mostly a disadvantage.

2.4.2 Pixy

Pixy is written in Java and is a command line application that can detect Cross-site Scripting and SQL Injection vulnerabilities using taint analysis, which makes it both less user-friendly and less complete than RIPS. It is run by specifying one file where vulnerabilities will be searched, which are then presented in a summary in the terminal. Alternatively, a DOT file can be produced, which can be visualized using the dot application from Graphviz [14], that represents the taint path that causes the vulnerability.

Pixy's main advantage lies in its static analysis. It contains both a front-end and back-end which correspond roughly to a compiler's front and middle-end. The front-end is composed of a lexer and a parser that are responsible for tokenizing the source code and building a parse tree from it, respectively, after which the parse tree is converted into an intermediate language P-Tac which is similar to the three-address code (TAC) presented in [15]. The front-end is thus able to verify whether the program's syntax is correct before analyzing it, and if it is, converts the source code into an equivalent language that makes further analyses much easier.

The back-end of Pixy is the one responsible for the analysis and actual detection of the vulnerabilities. The analysis occurs over the CFG of the program, where Pixy uses each statement of the

application as a node, i.e., the analysis occurs at the granularity of single statements instead of basic blocks. Pixy uses flow-sensitive, interprocedural and context-sensitive analysis. Path-sensitivity was not implemented due to its potential high performance toll. In order to make the analysis more precise, both literal and alias analyses are performed. Taint analysis is then applied in parallel with these analyses in the same way as it is in RIPS, i.e., starting at PVFs and going up the dependency tree until either sanitization functions or user input are found.

Pixy is thus complementary to RIPS since it has a worse user-interface, vulnerability coverage and even performance, but compensates this with its quite good static analysis which makes results more trustworthy.

Chapter 3

Vulnerabilities Distribution

In order to learn about the distribution of vulnerabilities by type in today's software, seven web applications were analyzed. The number of vulnerabilities per type that each application had in the past was found, and this is used to understand which vulnerabilities are more common in web applications. The following sections will explain how were the applications chosen, the methodology for the analysis and the last section will present the results.

3.1 Applications Analyzed

In the process of deciding which applications would be analyzed, some properties were taken into account. First, all applications had to be written in PHP and be open source, which are obvious requirements if they are to be analyzed for PHP vulnerabilities.

Second, the applications had to be popular, complex and mature. The popularity was mostly needed to guarantee that the results were meaningful to the reader. The complexity and maturity was required in order for the application to have a reasonable amount of vulnerabilities discovered that could be analyzed and also for the vulnerabilities analyzed to represent as best as possible the type and distribution of a typical web application.

The last property that was required was for the application website to have an easy way to browse the disclosed vulnerabilities. Some very popular applications (WordPress [16] and phpBB [17]) were excluded from this study simply because there was no easy way to get information about their past vulnerabilities.

This in no way means that these applications are the only ones that match these properties, since most properties are subjective. Also, the search for applications to analyze stopped as soon as the amount of applications found was deemed reasonable for the type of analysis that had to be done.

The seven resulting applications are:

- Drupal [18] and Joomla [19], which are both web content management systems (WCMS). A WCMS is an application that allows anyone to create and manage a wide variety of web content, from static pages to blogs, forums, or even e-commerce applications, without the need

for programming knowledge. Joomla is used in about 2.7% of all websites worldwide [20], while Drupal is used in about 1.5%. They are the second and third most used WCMSs, with the first being Wordpress.

- Moodle [21], an e-learning software application. It provides features to create fully online courses, or can instead assist regular courses by allowing discussion in forums and chats, creation of online assignments, or simply by providing a way to deliver content to students. It is one of the most popular e-learning platforms, together with Blackboard [22].
- phpMyAdmin [23], a web based front-end for MySQL. It has a very large community of users making it one of the most popular PHP applications. It has won the Community Choice Award for Best Tool or Utility for SysAdmins in three consecutive years, from 2007 to 2009.
- MediaWiki [24], a web-based wiki software application. As a wiki application its main focus is on providing ease of content management at the expense of extended control over the layout or functionality. It is the application used to run not only Wikipedia [25] but also all other Wikimedia Foundation websites and many other websites around the world.
- Zend Framework [26], an object-oriented web application framework, meaning it consists of object-oriented libraries designed to make common activities in web development easier. It won a Bossie Award in 2010 for "The best open source application development software" [27].
- SquirrelMail [28], a web-based email application. It has the usual functionality from most email clients, including address books, and folder manipulation. It is the webmail application used at CMU [29].

3.2 Analysis Methodology

After choosing the applications from Section 3.1 they were analyzed to find out which vulnerabilities had been disclosed for each. This section explains the methodology used to achieve this.

Gathering information about past vulnerabilities of each application was done by looking at a section in each application's website where past vulnerabilities are disclosed to the public. This could be for example a security section or a news section with each old release and its changelog. This webpage varied a lot between each application's website, both in format, content and detail, so different approaches had to be taken for each application. While some contained enough detail so that looking at the code was unnecessary, others required a deeper analysis of the code since the disclosed information was little more than the type of the vulnerability.

To get recent and thus meaningful results, only vulnerabilities patched in the last three years (2008-2010) were analyzed. Table 3.1 shows the versions that were studied for each application and the number of vulnerabilities that were analyzed. The displayed versions represent the ones that patched some vulnerability. However, we do not present versions that patch vulnerabilities that were also patched in higher versions, i.e., if a lower version appears in the table that means it patched some vulnerability that did not exist in a higher version.

Table 3.1 displays a great difference among applications in both the total number of vulnerabilities (the number of vulnerabilities analyzed ranges from 6 to 48) and in the vulnerabilities patched per version (Moodle has almost four vulnerabilities patched per version while in phpMyAdmin the rate is almost one). This is probably due to different policies used by the developers of each application in relation to the time interval between the release of each version.

Table 3.1: Versions and number of vulnerabilities analyzed per application

Web Application	Versions analyzed	Total Vulns
Drupal	5.6, 5.9, 5.11, 5.20, 6.1, 6.2, 6.3, 6.4, 6.5, 6.6, 6.7, 6.9, 6.10, 6.11, 6.12, 6.13, 6.14, 6.15, 6.16, 6.18	48
Moodle	1.9.10, 1.9.9, 1.9.8, 1.9.7, 1.9.6, 1.9.5, 1.9.4, 1.9.3, 1.9.2, 1.8.4, 1.7.5, 1.7.3	46
phpMyAdmin	2.11.5, 2.11.5.1, 2.11.5.2, 2.11.7, 2.11.7.1, 2.11.8, 2.11.9.1, 2.11.9.2, 3.0.1.1, 3.1.1.0, 3.1.3.1, 3.1.3.2, 3.2.0.1, 3.2.2.1, 2.11.10, 2.11.10.1, 3.3.5.1, 3.3.6, 3.3.7, 3.3.8.1, 3.4.0-beta1	29
Joomla	1.5.22, 1.5.21, 1.5.20, 1.5.18, 1.5.16, 1.5.15, 1.5.14, 1.5.13, 1.5.12, 1.5.11, 1.5.10, 1.5.9, 1.5.8, 1.5.7, 1.5.6	27
MediaWiki	1.16.2, 1.16.1, 1.16.0, 1.16.0beta3, 1.16.0beta2, 1.15.2, 1.15.1, 1.13.4, 1.13.3, 1.13.2, 1.11.2, 1.11.1	18
Zend Framework	1.7.5, 1.7.6, 1.9.7	9
SquirrelMail	1.4.21, 1.4.20, 1.4.19, 1.4.18, 1.4.17, 1.4.16	6

About the categorization of the vulnerabilities, the decision was to base it on the CWE (Common Weakness Enumeration) [30]. CWE is, according to its own webpage, "A Community-Developed Dictionary of Software Weakness Types". In other words, CWE is an encyclopedia with the objective of containing information about all known weaknesses, including the relation between different flaws.

In this context a weakness is synonym to a vulnerability, but for ease of readability, during the rest of this chapter, weakness will mean a type (or class) of vulnerability while vulnerability will mean a specific instance found in an application.

In order to categorize the vulnerabilities they were thus associated with a weakness from CWE, which made the analysis easier since it was based on a well known standard classification.

CWE classifies weaknesses according to their abstraction level, with Class being the most abstract type, Base being a specific type of weakness but independent of specific resources or technologies, and Variant the less abstract type, being specific to a particular resource or technology. CWE also specifies Categories which group weaknesses that share common attributes.

The first approach was thus to assign each vulnerability with the CWE weakness Base or Variant that best matched it, since they are more specific and thus would make the analysis results better represent reality, but two problems emerged from this approach. Sometimes it was very hard to find the right category when not enough details were disclosed by the application developers. It required inspection of the code, but the changed code was not always easily available, making the task nearly impossible. Still, this approach was followed at first, and even though the analysis was not as perfect as expected due to the lack of information, it was possible to categorize most vulnerabilities and start analyzing the results.

This was when the second problem revealed itself. Due to the categorization using less abstract weakness types, there was an high discrepancy in the rate of vulnerabilities per weakness, with a couple weaknesses representing the majority of vulnerabilities and many weaknesses with very little vulnerabilities found. This could be seen as just another case of the Pareto Principle [31], since about 20% of weaknesses represented about 80% of the vulnerabilities. This meant that about 80% of the weaknesses represented such a small fraction of the vulnerabilities found that they seemed almost irrelevant.

After a deeper look at these results a conclusion was reached that this reasoning was fallacious. In fact, what was happening was the fact that some weaknesses had variants which were being seen as entirely different weaknesses, i.e., the more variants a weakness had the more the vulnerabilities found were spread between the variants hiding the relevance that the original weakness should have. A different way to categorize the vulnerabilities was needed where similar weaknesses would be grouped together making their relevance independent of the number of variants they had.

In order to address these two problems, the approach that was followed was to categorize the vulnerabilities in a more abstract way, using not only CWE Bases and Variants but also Classes or Categories where deemed appropriate. This made the categorization much easier as much less details were needed and also gave the deserved relevance to some weaknesses that were spread through many variants before.

After the categorization of the vulnerabilities, the results were ordered by number of vulnerabilities per weakness and all but the top ten of the weaknesses were discarded since they were very uncommon. This top ten contains more than 90% of all vulnerabilities found thus covering enough vulnerabilities to be considered meaningful.

The weaknesses from top ten are (in order from the most to the least common):

Cross-Site Scripting (CWE 79) A Cross-Site Scripting weakness allows an attacker to inject a mallicious client-side script into the code of a website trusted by the user, such that the script will run with the same permissions as legitimate scripts from the target website. The script can thus access its cookies and session tokens and might even be able to modify the whole layout of the page.

Cross-Site Request Forgery (CWE 352) When this weakness exists, an attacker can forge a request to a server and force a client to make that request without his explicit consent. If the client is authenticated the server will see it as a legitimate request. This weakness occurs because the server does not verify whether the action being requested by the client corresponds to a form that was requested by him, which would make the above attack fail.

Improper Access Control (CWE 285) This weakness includes all vulnerabilities that have to do with problems at the level of access control, i.e., vulnerabilities that allow an attacker to do actions that it should not have authorization to do.

SQL Injection (CWE 89) This weakness occurs when user input is used in a SQL query without being sanitized. This allows an attacker to pass specially crafted input that modifies the structure of the query itself in a way that is advantageous to him. The consequences can range from having private data being read by the attacker to having the whole database deleted.

Code Injection (CWE 94) Contains all vulnerabilities that allow code to be injected and eventually be run by the application, allowing the attacker to modify the control flow of software. The injection is usually due to the use of non-neutralized input in the construction of the target code segment.

File Handling (CWE 632) This weakness is a category containing all flaws that are the result of problems when handling files or directories. It includes some very well-known vulnerabilities like Path Traversal (CWE 22) where a failure to canonize and restrict paths to known good files or directories leads to an attacker being able to access files outside the scope expected by the developer, and Unrestricted Upload of File with Dangerous Type (CWE 434) where failure to validate the type of file being uploaded might lead to an attacker uploading a dangerous file.

Information Exposure (CWE 200) A common weakness where sensible information gets leaked by the application, allowing an attacker to learn things that he should not. The information might be leaked unintentionally, in which case secret information might be leaked like passwords or credit card numbers (CWE 359), but it might also be leaked intentionally, in the case of debug information in error messages that are useful for the developer but are also useful for the attacker to learn more about how the application works (CWE 209).

Session Fixation (CWE 384) This weakness allows an attacker to force a user to use a predefined session identifier, giving the attacker the same privileges in the application as the victim user. This weakness might manifest if the application fails to regenerate session identifies on a privilege change and accepts session identifiers given by the user. This allows the attacker to find a valid session identifier which he might then force upon the victim, and since the session identifier is not regenerated on login the session identifier used by the victim will be the same as the one given by the attacker.

Insecure Temporary File (CWE 377) The term temporary might lead developers to disregard the sensibility of temporary files, leading to this weakness. Typical instances of this weakness are temporary files with predictable names or which are created in directories where the attacker has write access, which might allow an attacker to tamper the file as it is created or written to. The danger of this weakness varies depending on the purpose of the temporary file.

Improper Authentication (CWE 287) THis weakness, although uncommon, is very critical in the sense that a problem with the authentication might allow an attacker to authenticate as another user giving him privileges equal to the rightful owner of the account. It is much more

critical than improper authorization as the attacker does not gain access only to a specific functionality where access control has failed, but to the whole range of functionality that the victim has access to.

3.3 Results

After the methodology from the previous section was applied to a total of 196 vulnerabilities, the results obtained were analyzed. Figure 3.1 shows the percentage of vulnerabilities per weakness relative to the number of vulnerabilities that composes the top ten. While this figure only represents the top ten, it contains 183 individual vulnerabilities, about 93% of all vulnerabilities analyzed. The rest of the weaknesses were not represented both to make the graph easier to read and because most weaknesses in the 7% left had but a single vulnerability.

The first thing that catches the eye is how nearly half of the vulnerabilities disclosed belong to the Cross-Site Scripting (XSS) weakness. This does not come as a surprise since XSS comes in first in the CWE Top 25 [32] and second in the OWASP Top 10 [33].

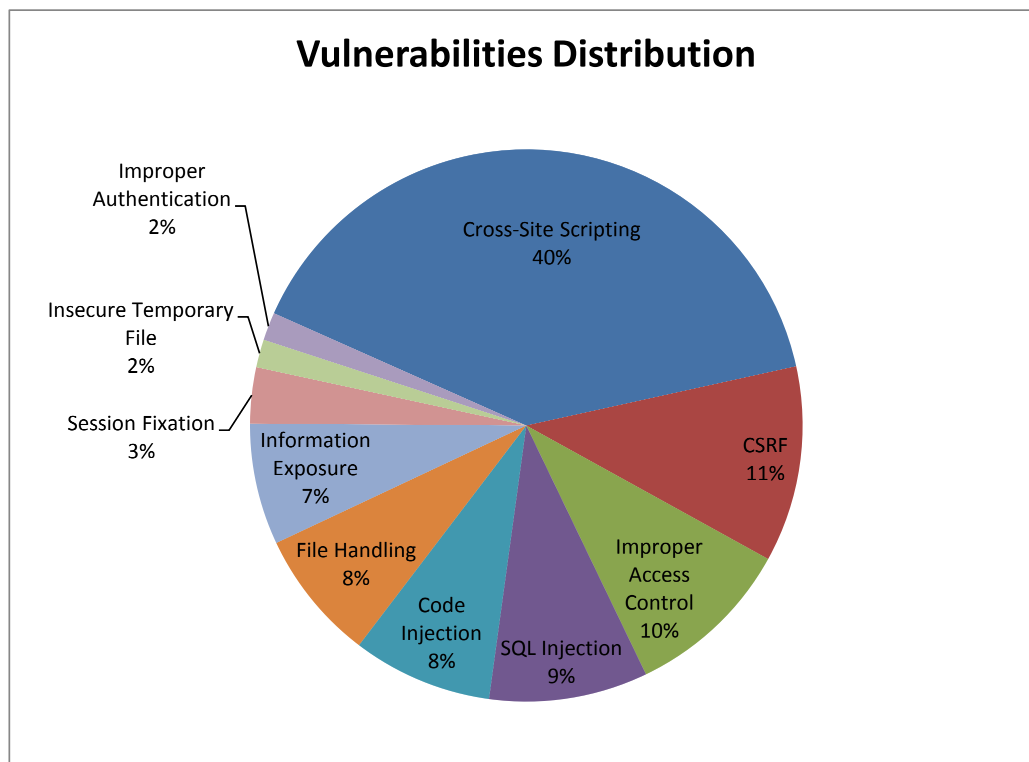


Figure 3.1: Distribution of vulnerabilities per type

Besides the fact that the vulnerability is really prevalent in today's web applications, another interpretation can be seen under the light that this analysis only takes into account vulnerabilities that have been already been found in the applications. While this is obvious (one cannot infer anything from vulnerabilities that have not yet been found), its implications might not be. In this case, what

it means is that vulnerabilities that are easier to find will be reported in a higher number than the harder to find ones, giving the idea that they are more prevalent when this might not be the truth.

The exaggerated high number of XSS vulnerabilities might be a result of this fact, since it is a well known vulnerability and is relatively easy to discover, both manually and automatically. Still, this weakness is also easy to introduce, and can appear in a wide array of places, which contributes to the high number.

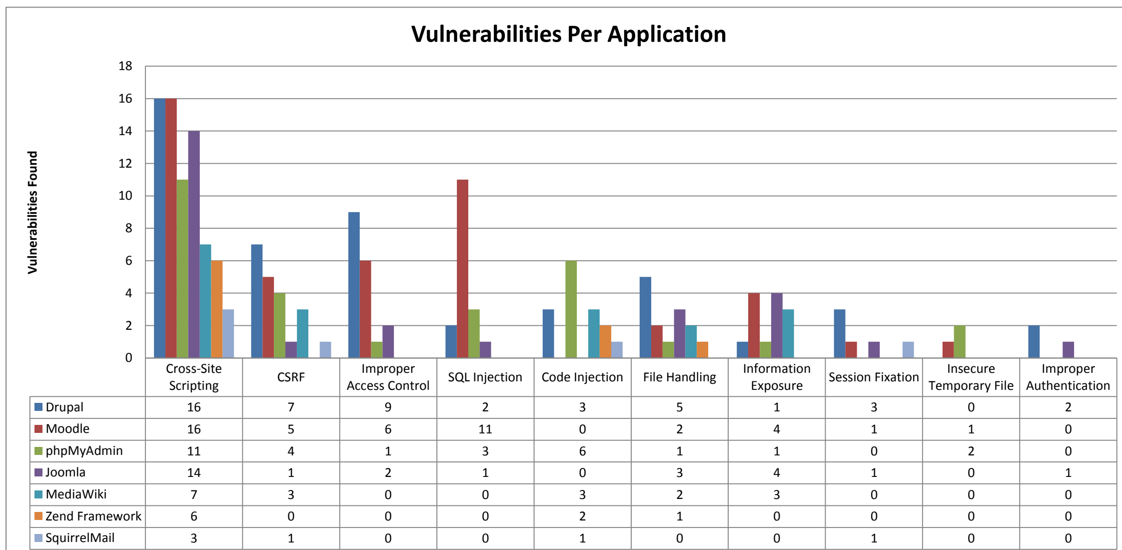
The next six weaknesses in the top ten, Cross-Site Request Forgery (CSRF), Improper Access Control, SQL Injection, Code Injection, File Handling and Information Exposure, had a quite similar amount of vulnerabilities. This makes their actual prevalence when compared to the other weaknesses less than the XSS weakness, i.e., while there is a great chance that CSRF vulnerabilities are more common than Information Exposure vulnerabilities, a couple more vulnerabilities analyzed could easily change the order of any two consecutive weaknesses in these six. While this makes it difficult to say for sure which is the next most common weakness after XSS, it does allow the conclusion that all of these six weaknesses are worth great attention as they are very common in most applications.

Comparing these results to the ones from the CWE Top 25 shows that CSRF, SQL Injection, Improper Access Control and File Handling (as Path Traversal and Unrestricted Upload) can be found in-between the first 8 positions in the rank. Code Injection can be found as OS Command Injection and PHP File Inclusion in the 9th and 13th position and Information Exposure can also be found in the 16th position.

Analyzing this under the light that the CWE Top 25 contains all types of weaknesses and not only weaknesses specific to the web, and also that their rank is based not only on the prevalence of weaknesses but also on their threat, explains the slight deviation in the results. For example, Information Exposure can be quite common but usually its consequences are limited to give information to the attacker, and while this should not be underestimated, it is certainly less dangerous than a SQL Injection. This could in part explain why Information Exposure has such a low rank in the CWE Top 25.

The results from OWASP Top 10 also share similarities with our experiment. XSS ranks 2nd and is only beat by Injection, which includes both CWE weaknesses of Code and SQL Injection. CSRF is also present, ranking 5th, and Improper Access Control can be matched with Insecure Direct Object References and Failure to Restrict URL Access since the latter are a consequence of the former. File Handling and Information Exposure do not have such a direct correspondence. The reason for this is due to the fact that this analysis was made based on CWE weaknesses, which are quite different in nature from the ones found in OWASP Top 10, and thus trying to compare these weaknesses with those in OWASP Top 10 would be unnatural.

The last three weaknesses in the top ten, Session Fixation, Insecure Temporary File and Improper Authentication are much less common. One thing that is worth noticing is the fact that ranking 3rd in the OWASP Top 10 is Broken Authentication and Session Management, which encompasses both Improper Authentication and Session Fixation. Even if these two weaknesses were put together, they would still be at the bottom of the top 10, so the reason for it ranking 3rd in OWASP Top 10 might have to do with the threat these weaknesses represent, which is not taken into account in this analysis but is a warning that they should not be underestimated.



	Cross-Site Scripting	CSRF	Improper Access Control	SQL Injection	Code Injection	File Handling	Information Exposure	Session Fixation	Insecure Temporary File	Improper Authentication
Drupal	16	7	9	2	3	5	1	3	0	2
Moodle	16	5	6	11	0	2	4	1	1	0
phpMyAdmin	11	4	1	3	6	1	1	0	2	0
Joomla	14	1	2	1	0	3	4	1	0	1
MediaWiki	7	3	0	0	3	2	3	0	0	0
Zend Framework	6	0	0	0	2	1	0	0	0	0
SquirrelMail	3	1	0	0	1	0	0	1	0	0

Figure 3.2: Number of vulnerabilities by type per application

Figure 3.2 tries to show with a little more detail the vulnerabilities analyzed for each application. Applications are ordered by total number of vulnerabilities to help better analyze the graph. For the most part the results follow what would be expected, that applications with higher number of analyzed vulnerabilities would show a higher number of vulnerabilities for each weakness, still, there are some results that do not follow this rule. Moodle, for example, is responsible for two thirds of the SQL Injection vulnerabilities found while Drupal, having about the same number of vulnerabilities analyzed, is responsible for little more than a tenth. Another example is phpMyAdmin and Joomla which despite also having about the same number of vulnerabilities analyzed, have very different distributions of vulnerabilities, with phpMyAdmin having more CSRF, SQL Injections and Code Injections, and Joomla having more File Handling problems and Information Exposures.

These exceptions can be explained as being a natural result of the differences between the applications, from the developers that work on them which might have a tendency to insert a certain weakness more than another, to the security team auditing the application which might be better at finding certain weaknesses, or even to the type of application (CMS, Wiki, Webmail client, etc.) which might influence which weaknesses become more common.

Figure 3.2 shows also that SQL Injection vulnerabilities are not as common as they look. Moodle is an obvious outlier in its number of SQL Injection vulnerabilities from some of the reasons discussed above, and if it is ignored, SQL Injection vulnerabilities become about as common as Session Injections. This is no reason to ignore them since the attacks allowed by a SQL Injection vulnerability are a real threat, but makes them a little less of a priority. On the other hand, the percentage of Cross-Site Scripting vulnerabilities is confirmed with the fact that no outliers exist, in fact it seems to make up about 40% of the vulnerabilities of every application, with no exception. Other weaknesses have not such perfect distributions but also do not have any outliers that require a re-analysis, which confirms the results from Figure 3.1.

Chapter 4

Architecture

This chapter describes a generic architecture for a vulnerability injection tool. The goals that this architecture must achieve are presented in Section 4.1, the architecture itself is described in Section 4.2 and Section 4.3 provides some examples of the location plugins used in this architecture.

4.1 Goals

The goals for the architecture are described in the list below as a set of three functionalities that it must provide and five properties it must have.

Vulnerability Injection This is the main goal of this work, and is thus the one that results in the most important architecture requirements: the need to allow both discovery of vulnerability injection locations and the actual injection of the vulnerabilities.

Vulnerability Detection While this work is not about vulnerability detection, it has many common characteristics with vulnerability injection as stated before in Section 2.4, and it should thus be possible to create an architecture that allows for both with little overhead.

Besides, as long as the architecture tries to make a clear separation of the common and specific components of detection and injection, a tool following this architecture will have great advantages compared to having different tools for each functionality.

The most obvious advantage is that upgrading the common components makes both functionalities better with half the effort. A less obvious one lies in the fact that there is already vast research on vulnerability detection that vulnerability injection could profit from, and having an architecture that tries to separate the common from the specific components makes it easier to know where and how could that research be integrated.

Static analysis While it is obvious that the architecture must allow some static analysis to be done in order to search for the location where to insert the vulnerability, the architecture should allow for different types of static analysis as each has its advantages. For example, RIPS, presented in Section 2.4.1, does not parse the tokens from the program which does not allow

it to verify whether program's syntax is correct, while Pixy, presented in Section 2.4.2, does it and also uses alias analysis. The architecture should allow the implementation of vulnerability detectors/injectors based on both approaches since each has its own advantages.

Modular The architecture must be modular in order to allow for separation of concerns. This is not only a goal but is also a requirement if both vulnerability detection and injection are to be supported, since modules of the architecture common to both goals must be clearly separated from modules specific to each. One way of making the architecture modular is to follow the example of a compiler, where front, middle and back-end have completely different concerns. Since the architecture must allow for an analysis similar to that of a compiler, the analogy is quite good and easy to follow.

Configurable The user of the tool must be able to configure it to better tackle his or her current needs. Some examples of configuration are allowing the user to decide between vulnerability detection and injection, which vulnerabilities to detect/inject, whether to replace the input files or to create new ones on injection, which analyses to perform, and how verbose the output should be.

Extensible This goal means that the architecture must allow for the vulnerabilities that are detected or injected to be added or modified without requiring modifications to the rest of the tool. The same must happen for the types of analyses that the tool uses, and therefore it must be possible to add and modify them without having to change any code besides the one for that specific analysis.

Language Independent The architecture must be independent of any language, which is not to be confused with being language extensible. Language independence means that any tool for injecting/detecting vulnerabilities can be implemented following the architecture, independently of the language it targets.

Language extensibility on the other hand means that a tool following the architecture must allow for new languages to be added, without requiring modifications to most of the tool. While the concept is quite easy to understand, such a tool would need to work over some abstract language that is able to represent any other language, such that new languages could be added simply by providing a translation to the abstract language. To ou knowledge, such abstract language has not yet been described, and approximations to the problem would be too complex to consider in this work.

Implementable The last goal is to avoid falling into the pitfall of making an architecture that is so generic and complex that no tool will ever be able to follow it. It should thus be simple enough to understand and implement but at the same time versatile enough so that it can be used by both a single person creating a simple tool for research or by a company trying to create a very complete tool that can be commercialized.

4.2 Architecture

The architecture from Figure 4.1 was designed to achieve the goals from the previous section.

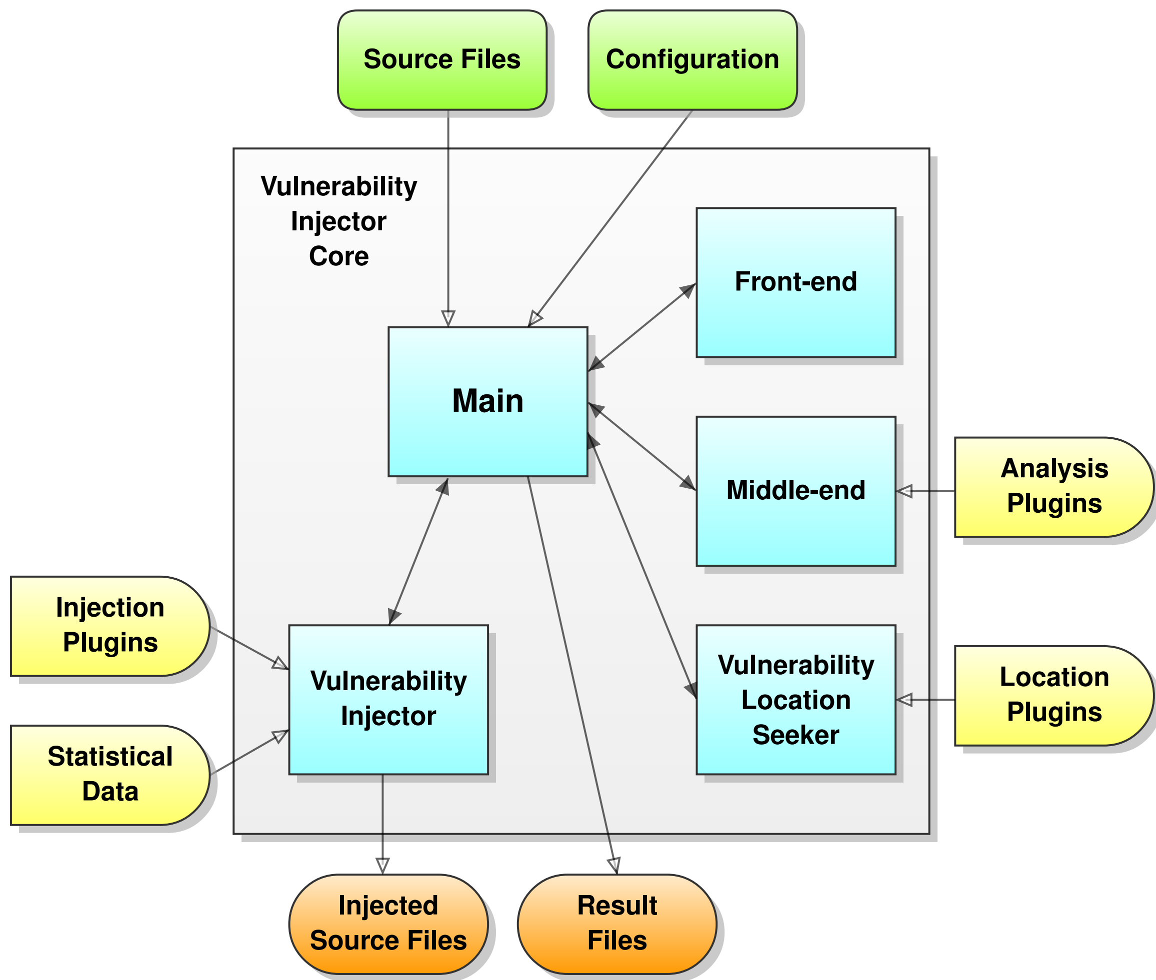


Figure 4.1: Architecture for a generic vulnerability injection tool

The architecture is composed of a vulnerability injector core composed of 5 modules, which are explained in the following 5 subsections. The last subsection will explain the typical work flow of the architecture.

4.2.1 Main module

The main module is the first one to run, and might range from a script called from a command line to a graphical user interface. It is responsible for receiving the input, which is composed of the source files that the user wants to analyze plus the configuration given by the user. The configuration may consist of the vulnerabilities the user wants to detect or inject, or the analyses he wants to run.

The main purpose of the main module is to allow each of the other modules to be self-contained and unaware of the existence of each other. This is accomplished by having the main module work as an intermediary that uses the other modules as if they were stateless functions or libraries, i.e., the modules receive all their input from the main module each time they are used and after

returning the results of their execution they do not need to store any state (as the main module will do it instead).

Modules will still use the output of other modules as their input, but only through the main module. The fact that no module calls another module directly makes the architecture loosely coupled, allowing modules to be easily reused in other tools, since they do not depend on any other part of the architecture. Without the presence of a main module, modules would need to call each other in turn, making the architecture more difficult to extend.

The main module is also responsible for carrying out the session according to the configurations. This is achieved by having the main module use each module according to what the configuration demands, e.g., telling the middle-end module (which is responsible for doing the static analysis) to run only the analyses that the user specified, or using or not the vulnerability injector module depending on whether the user wants to inject or detect vulnerabilities. With this organization each module only sees the configurations that affect it.

Another purpose of the main module is to take care of keeping any state that must be maintained between runs of the tool. One example might be if a run of the tool is stopped during the execution and the user wants to resume it at a latter time: the main module can stop the current running module and store the current state in a persistent format, e.g., a file or a database. When the tool is resumed, the main module will restore the state, making it unnecessary to redo the whole process again. The alternative without a main module would require that all modules had the capability to stop and resume a session, or at least had knowledge of another module that did it, since any module could be running at the time the user decided to stop the tool.

Finally, the main module is also responsible for producing the actual output of the application, except for the injected source files which are the responsibility of the vulnerability injector. This output includes information about which vulnerabilities where detected/injected, graphs explaining the dependency path that led to the vulnerability, and in case of vulnerability detection it might output an explanation on how to correct these vulnerabilities.

4.2.2 Front-end

The front-end of this architecture is analogous to the front-end of a compiler. Its purpose is thus to perform the translation from the source code into an intermediate representation. This can range from doing as little as tokenizing the source code, similar to what RIPS (Section 2.4.1) does, to performing lexical analysis, syntactic analysis and semantic analysis, resulting in the parse tree, the symbol table and the attribute grammar. It might even perform intermediate code generation such as Pixy (Section 2.4.2), resulting in an intermediate representation of the code in the format of TAC (three address codes), which are language independent, and thus allow the modules that use it to perform their work independently of the language that originated the TAC. Complete language

independence is unfortunately impossible since vulnerabilities can be language dependent or at least manifest themselves differently in each language.

4.2.3 Middle-end

The middle-end is also analogous to the middle-end of a compiler since its purpose is to perform static analysis over an intermediate representation of the source files. The difference to a typical compiler middle-end lies in the analysis plugins that this architecture uses. Since each analysis is performed somewhat independently of each other, it is possible to define each analysis separately as a plugin and have the middle-end run it as needed. The advantage lies in the fact that the user can easily update the tool with newer versions of the analyses that are more effective or efficient, or even add new types of analysis in order to make the results more precise.

Plugins require meta information on the analyses in order for the middle-end to make decisions on which ones to run and when. Since analyses might depend on other analyses, their plugins should state their dependencies, e.g., dependence analysis requires reaching definitions analysis which needs control-flow-analysis. This would result in a dependency tree that would allow the middle-end to calculate, based on the desired analyses what is the minimum set of analyses to run; all other would be optional and could be set to be executed by modifying the configurations.

Another type of meta information that is needed is the granularity of each analysis, since some might work over individual statements but others might work over basic blocks (see Section 2.3.2). In this thesis, it was not possible to go into more detail on the analyses plugins; further study is required in order to understand exactly what the plugins must describe and how such that new analysis can be added without requiring modifications to the middle-end.

4.2.4 Vulnerability Location Seeker

This module is the one responsible for actually finding vulnerabilities and places to inject them, and is thus of high importance to the performance of the tool. It uses plugins that describe how vulnerabilities manifest themselves in the source code. These plugins are used both in the detection and injection of vulnerabilities.

In case the user wants to detect vulnerabilities, the plugin must describe how the vulnerability manifests itself, specifying both what must happen for the vulnerability to occur (such as using user input in a SQL query) and what must not (such as sanitizing that input). The vulnerability location seeker will then use the result from the analyses made by the middle-end to find whether the code contains patterns that match the ones in the plugins, and for each one it will report to the main module the type of vulnerability it has found, the line where it occured and the data from the analysis that supports this claim. The plugins must thus contain the vulnerability each pattern corresponds to in order for it to be reported back to the user.

If the user asked instead for vulnerability injection, a similar process occurs, with the difference that the vulnerability location seeker must now search for places where the vulnerabilities might be injected. This uses the same plugins as above, and requires the patterns described in the plugins

to have information that allows the vulnerability location seeker to infer where a vulnerability can be injected. For example, if the pattern for a vulnerability claims that there is a set of conditions that must be verified in order for the vulnerability to occur, and the vulnerability location seeker finds a pattern in the code where all the conditions are verified except for that one, then it might be possible to modify the code in a way that allows that condition to be verified also, thus injecting the vulnerability.

While this method is correct, not all conditions can easily be modified and thus the vulnerability pattern must specify which conditions can. As an example, SQL injection requires that a SQL query contains unsanitized data originating from the user input, and therefore the conditions, somewhat simplified, are the existence of a **SQL query**, the fact that the data it contains is **unsanitized** and that it originates or depends on **user input**.

While it is a realistic injection to modify the **unsanitized** condition by removing or tampering with the sanitization, since the apparent behavior of the application would remain the same and it is a common mistake [34], it is much harder to modify the other two conditions in order to make a realistic injection.

Adding a nonexistent **SQL query** requires finding out whether a database is accessed and how in order to make a correct query. This is complex but possible, but even then making a realistic SQL query would require knowledge of the purpose of the application, something that is usually too complex to be described or analyzed correctly.

If the last condition is not verified, i.e., the input to the SQL query does not depend on **user input**, the modification required to verify that condition is also non-trivial since it requires applying one of the two following non-trivial methods.

The first method is to add a new entry point of user input into the application while taking care not to modify its apparent behavior, such as adding a statement reading from some unused cookie or POST parameter and using its value somewhere in the SQL query. While this would not modify the apparent behavior of the application, since the user input's entry point is hidden, it is difficult to make such a modification look realistic.

The second method consists of using some pre-existing variable that depends on user input in the pre-existing SQL query, and has the advantage of not requiring new entry points of user input to be added. It loses its advantage however due to the complexity of finding a triple of user input dependent variable, SQL query and location inside the SQL query, such that adding the variable to that location of the SQL query does not modify the apparent behavior of the application and seems realistic when looking directly at the code.

Before ending the description of the vulnerability location seeker it is important to notice that it is only responsible for finding the locations where vulnerabilities can be found or injected. In vulnerability detection, it is the main module that actually presents the results and potential solutions, while in vulnerability injection, it is the vulnerability injector that does the actual injections.

The vulnerability location seeker is thus oblivious to how the injections are actually done, it only returns information about which types of vulnerabilities can be injected and where.

4.2.5 Vulnerability Injector

The vulnerability injector is the component responsible for the actual injections in the source code. It receives the locations where it can inject from the main module and proceeds to do the actual injections. As most modules in the architecture, it uses a set of plugins to allow for extensibility and updates to the injection procedure. It also uses a configuration file (or other type of persistent data) that provides statistical data for the injections.

The statistical data file contains information about the distribution of vulnerabilities in real applications, such as the one gathered in Chapter 3, and can be used in order to inject vulnerabilities with a realistic distribution. The user should thus be able to specify whether he wants all the potential vulnerabilities returned by the vulnerability location seeker to be injected, or whether he wants the vulnerabilities injected to follow a realistic distribution.

Since there are applications where some types of vulnerabilities cannot be injected, e.g., an application without databases cannot have SQL Injection vulnerabilities, the vulnerability injector should not try to follow the distribution exactly, since it would mean that no vulnerability at all would be injected. Instead, it should try to do its best to follow the distribution while still injecting a reasonable part of the discovered vulnerabilities. For example, if it finds 70 Cross-Site Scripting vulnerabilities, 15 CSRF and 10 SQL Injections, it can injects 44, 12 and 10 vulnerabilities respectively, since this respects the relative distribution for these three types of vulnerability (according to the analysis from Chapter 3) while still injecting about 70% of all potential vulnerabilities found. If besides those vulnerabilities it had also found a single Code Injection vulnerability, it would be best to inject it and keep the same number of other injected vulnerabilities, since reducing the numbers of the others to 5, 1, 1 respectively in order to respect the distribution would only create a very little percentage of all potential vulnerabilities that were found.

The statistical data might be more or less detailed, depending on the analysis that originates it. In the analysis from Chapter 3, all Cross-Site Scripting vulnerabilities were considered to be equal, but this is not true as is explained in Section 4.3, since these vulnerabilities may be for instance Reflected or Stored. The reason for the analysis to consider them all the same is that the number of vulnerabilities analyzed was not enough to take conclusions with such detail, i.e., if the vulnerabilities were divided in their sub-categories the resulting numbers would be too small to take any conclusions with statistical meaning. If an analysis that goes into more depth exists, it is possible to use it instead to make the results even more realistic.

While the typical way to get the statistical data is through third-party analysis, another interesting method can be used. Since the architecture allows for both vulnerability detection and injection, it is in principle possible to have the vulnerabilities gathered though vulnerability detection feed the statistical data used for vulnerability injection. This would require a mechanism to detect duplicates in order to prevent double counting of vulnerabilities, and might not be as precise as a careful

manual analysis, but has a couple advantages that make the idea interesting: it would allow for higher amounts of data to be collected as long as the tool had a high level of utilization, such as in a big software company or a publically available web service.

The injection plugins describe how the injections are actually done. They use the information about how the parse tree (returned by the front-end) can be modified to insert the vulnerability, and uses the modified parse tree to reconstruct the source code with the injected vulnerability.

Each injection plugin must contain the type of the injection and its parent in the vulnerabilities hierarchy if there is one, e.g., Cross-Site Scripting is parent to Cross-Site Scripting in HTML, but has no parent itself. The parent property is useful both for the user and injection plugin developer to think of the relations between vulnerabilities and is needed to apply the statistical data, since there might be injection plugins available for some child vulnerabilities while there is only statistical data for parent vulnerabilities. In that case, the specific vulnerabilities are injected, while their distribution is based on the parent vulnerability.

Most injection plugins also contain an actual description of the injection. Some plugins might also represent a parent vulnerability with no description, since it always manifests as one of its children. Cross-Site Scripting is a good example because it is always possible to categorize it as one of the children, and so it does not make sense to describe a generic injection for it, only the more specific injections.

If an injection plugin does have an injection description, it must consist of a series of available modifications to the source code that insert that specific vulnerability. For example, if a sanitized variable must be made unsanitized, this can be done by removing the sanitization function by implementing a dummy function with the same name such that it overrides the original function, or even by wrapping the statement where the sanitization occurs in an **`if`** condition that can never occur.

In order to choose between the available modifications, several mechanisms can be used, such as selecting at random, allowing the user to decide in the configuration, allowing the user to decide at runtime, or even using statistical data to suggest the most realistic alternative.

4.2.6 Typical Work Flow

After the description of the five modules that compose the vulnerability injector core plus its plugins, it should not be difficult to guess how a typical run would play out, but there are some subtleties that make it worth explaining in more detail.

A typical run starts with a user specifying the source files that are to be analyzed and the configuration that he wants for that run. This input can be passed to the application either as arguments in the command-line interface of the tool or through a graphical interface.

The main module can now resume a previous session if that was the user's intent or start a new one, which is what will be described next. The front-end module is the first one to be used by the main

module, and its purpose is to receive the source code and return an intermediate representation of it.

The middle-end can now be used to perform static analysis over the intermediate representation, returning data structures that can be used by the vulnerability location seeker. There is a performance optimization that can be applied in this phase: since the user might want to detect or inject only a subset of the available vulnerabilities, there are some analyses that might not need to be performed. Each location plugin defines the analyses it depends on, or in other words, the analyses that must be run in order for the location plugin to work. If the main module asks the vulnerability location seeker for analyses that must be run for the subset of vulnerabilities that the user selected, then it can tell the middle-end module to execute the needed analysis, resulting in a faster run.

After the analyses are performed, the vulnerability location seeker is ready to start using its location plugins to find vulnerabilities or locations to place them, after which it will return all information needed to report or create them. If the user wanted to generate vulnerabilities, the vulnerability injector is now used by the main module to perform the actual injections in the source code.

In the end, after all modules have run, the main module will output the result files described in Section 4.2.1, after which it stops running.

4.3 Location Plugins

In order to better understand how the location plugins work, the architecture from Figure 4.2 was designed for the Vulnerability Location Seeker.

It is worth noting this is not the only valid architecture for the Vulnerability Location Seeker; due to the modular nature of the architecture from Section 4.2, several different implementations of the Vulnerability Location Seeker are possible as long as its input and output are similar.

In this architecture, the idea is to use program query languages such as JQuery [35] and SOUL [36] to query the program for the location of vulnerabilities or the location where vulnerabilities can be injected. In order to allow for easy extensibility of the queries and language independence, the queries are generated at runtime by the Query Generator using two types of plugins, the query templates and language translators.

The query templates are normal queries in the chosen program query language, except for some embedded template code that they might have. The template code is written in a different language from the query, and its purpose is to create parts of the query dynamically.

For example, if someone wants to create a query template, an easy way is to create some examples of the actual queries he wants to be executed and find the parts that differ in the example queries. The query template will then consist of the common parts of the example queries, with the parts that differ described using template code.

Most times, the queries will be quite distinct from each other, or else the differences are mainly restricted to the parts that are language dependent, such as queries that differ in the function names being searched for, etc. In the former case, the queries should not be merged in one

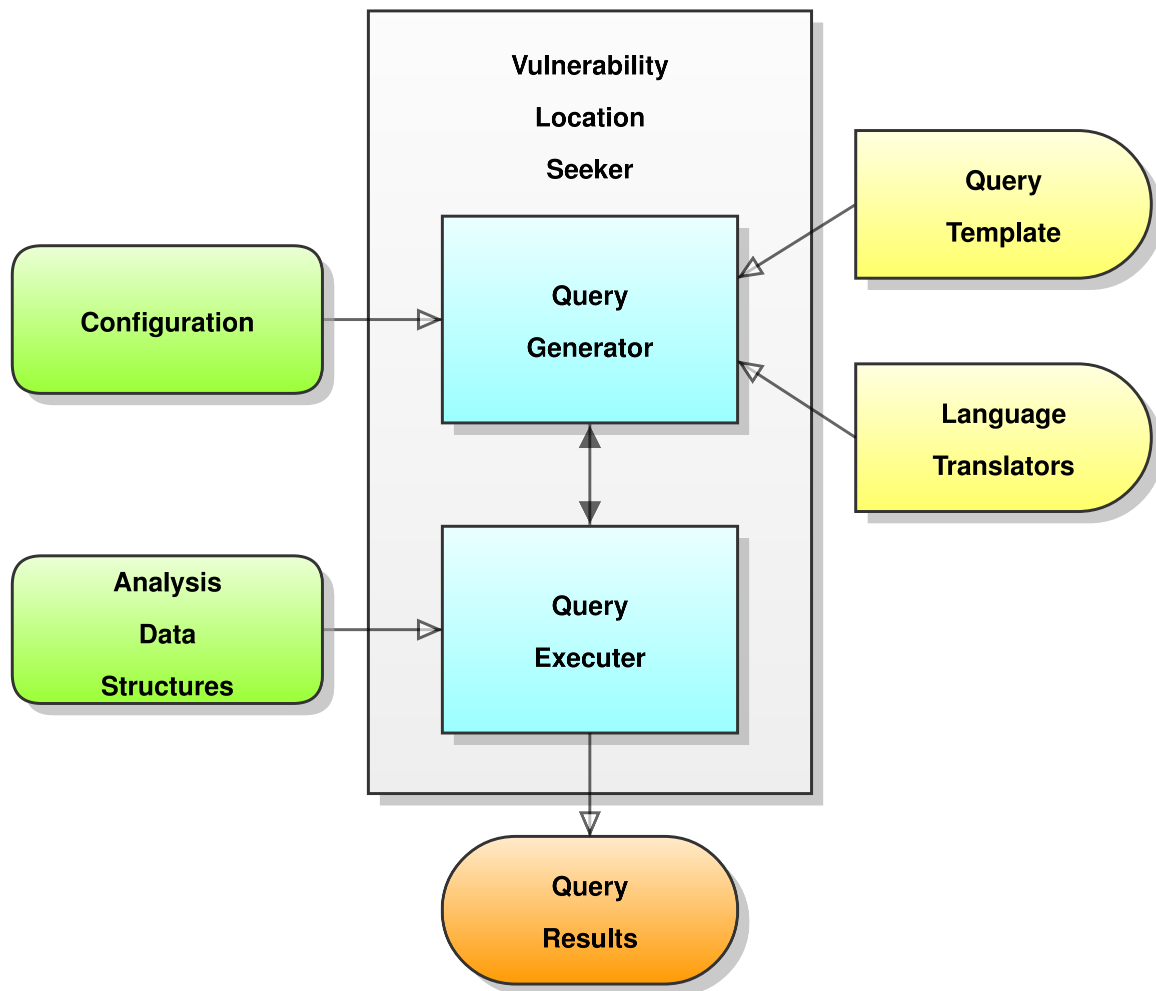


Figure 4.2: Vulnerability Location Seeker Architecture

template since they probably represent different vulnerabilities. In the latter case, using language translators solves the problem.

A language translator is a structured file written in XML, JSON or other data formatting language (XML will be used for the following examples). It contains language specific terms that can be referred by the template code to build a query.

Using an example to make the concept clearer, suppose one needs a query whose purpose is to return all functions that are inherently dangerous, such as the `gets` and `strcpy` functions in C. Those functions are language dependent, which means that if queries were not generated dynamically, a different query would be required depending on the language. Besides, while searching for dangerous functions is a process that typically never changes, the list of dangerous functions might grow over time as new functions are found to create security problems. This would require modifications to the actual queries, which are not practical, as they grow in size and complexity.

Using query templates and language translators makes this process much simpler since there is a separations between the definition of the query process from the definition of the dangerous functions for each language. With this solution, only a single query definition is needed, the query

template, while adding new dangerous functions also gets much easier since the definitions are in a straightforward XML file.

Besides the language specific terms, the language translators must also specify what language they translate, what vulnerability they target and what query template must be used for that vulnerability. This information will then be used by the query generator to create the queries. The query generator needs only to choose the right language translator using the target language and vulnerability as indicated in the configurations, get the query template specified in it, and run the template code for that template using the language translator when needed.

The generated queries are then run over the data structures provided by the middle-end and the results are returned to the main module.

In order to provide some examples, an actual program query language had to be selected. The language of choice is PQL from Jarzabek [37] for the following reasons:

- PQL is easy to understand. The syntax for PQL has some influences from SQL, and is in general not much different from natural language. Since the examples should be easy to understand this is a very important characteristic.
- PQL is language agnostic. Queries in PQL are based on an abstract syntax grammar meaning it is independent of the language they are applied to. As long as the front-end of the architecture translates the source code concrete syntax into an abstract syntax, the same query can be applied to many languages.

  As an example, functions from any language will be represented in the same abstract format, allowing the queries format to be the same for all languages. However, functions names cannot be abstracted, meaning that if the query must know the function names to be searched, such as in the example of searching for dangerous functions, then language translators must be used.
- PQL assumes pre-existing static analysis. Some examples include control flow analysis and dependency analysis. This is an advantage since it matches the approach taken in the architecture, where the vulnerability location seeker will use the static analysis performed by the middle-end.

The example below represents a generic PQL query.

Listing 4.1: Generic format of a PQL query

```
view some_name

some_entity a, b
some_other_entity c

Select <a, b, c.some_attribute>
such that
    Some_Relationship(a, b)
and
    Some_Other_Relationship(b, c)
pattern
    some_pattern(a, c)
and
    some_other_pattern(b, c, _)
with
    a.some_other_attribute="some_value"
or
    b.some_other_attribute="some_other_value"
```

A PQL query begins with the optional keyword **`view`** that specifies the name of the query. This name can be used by other queries to refer to the results from this one.

After the **`view`** line come the declarations where synonyms are given to entities, as can be seen in lines 3 and 4. An entity represents the set of all its instances which are usually components of the abstract syntax tree, such as procedures, variables, or statements. Synonyms are normally used when a query wants to refer to more than one set of instances of the same entity, such as in a query where we want to find all procedures (first set) that call other procedures (second set). Nevertheless, it is also possible to use a synonym even if a single set is used in the query such as in line 4.

The query itself starts at line 6 with the **`Select`** keyword followed by a tuple which represents the query results. The tuple can contain both entity instances and entity attribute values. The tuple from the example above means that the result of the query will consist of all triples of instances of `a`, instances of `b` and values of the attribute `some_attribute` of `c` such that the conditions of the query are satisfied. This is analogous to the `SELECT` statement of a SQL query which returns all values that match the conditions given in the query [38].

The conditions of the query come after the **`Select`** keyword and make use of the first order logic operators **`and`**, **`or`**, **`not`**, **`exists`** and **`forall`**, to allow for more complex conditions. There are three types of conditions, which come after the **`such that`**, **`pattern`** and **`with`** keywords.

The **`such that`** conditions represent relationships that must exist between entities. Examples of relationships include:

- Uses(p,v), which means procedure p uses variable v
- Calls(p1, p2) which means procedure p1 calls procedure p2
- Affects(s1, s2, x, y), which means that variable x at statement s1 is used at statement s2 to compute the value of y
- Affects*(s1, s2, x, y) which is the transitive version of Affects, i.e., it means that variable x at statement s1 affects directly or indirectly the value of y at statement s2.

It must be noted that these relationships are the reason that static analysis is required by PQL, since for example the Calls relationship requires control flow analysis and the Affects and Affects* relationships require dependency analysis.

The **pattern** conditions represent code patterns. When the pattern represents a function, we note that the first parameter of the pattern represents the return value if there is one. This is an addition to the PQL language since it was not possible (or at least not easy) to represent that a variable got the return value of some function. Patterns can then be as follows:

- strlen(r, v) which means somewhere in the code, variable v is used as the parameter for a strlen call and variable r is the returned value.
- plus(r1, v1, v2) **and** times(r2, r1, v3) meaning there is a code pattern with the format (v1+v2)*v3.

Finally, the **with** conditions constrains the attribute values of entities. Examples include:

- v.varName="x" that is used if we want to constrain the variable v to variables with the name "x"
- p.procName="f" that constrains the p procedure entity to procedures named "f"
- s.stmt#>*20* that constrains the statement entity s to the statements that come after line 20.

To finalize the explanation of the generic query, the underscore "_" in line 14 must be explained. Underscores "_" can be used in relationships and patterns as a way to say "I do not care". For example, if used in a relationship such as Calls(p,_) it will constrain p to the set of procedures that p calls, while in Calls(_,p) it constrains it to the set of procedures that are called by some procedure. If used in a pattern such as substr(_, v, _) it constrains v to the set of variables that are passed to the procedure substr as the first parameter (remember, the first parameter in the pattern corresponds to the return value, not the first parameter of the function).

We note that this generic explanation of PQL did not cover all features of PQL, detailed information can be found in [37]. Also, some features might be used in the examples in the rest of this section that have not yet been explained. They will be explained wherever they are used.

We will present next three examples of queries.

Listing 4.2: PQL Example 1

```
1  procedure p, q
2
3  Select q
4  such that
5      Calls(p, q)
6  with
7      p.procName="proc_example"
```

This example starts by declaring `p` and `q` as synonyms of the `procedure` entity. The **`Select`** statement then tells us that it will return the set `q` that meets the conditions below, which in turn say that procedure `p` must call procedure `q` and that procedure `p` must have the name `"proc_example"`. In short, this example returns the set of all procedures that are called by the procedure named `"proc_example"`.

Listing 4.3: PQL Example 2

```
1  view all_affected_by_x_at_20
2
3  var varx, vary
4  statement sx, sy
5
6  Select <sy, vary>
7  such that
8      Affects*(sx, sy, varx, vary)
9  with
10     varx.varName="X"
11 and
12     sx.stmt#=20
```

This example is slightly more complex than the previous one. It begins by defining a **`view`** name that allows this query to be used by another. After the declarations, the **`Select`** statement shows that a tuple of two entities, a statement and a variable, will be returned. The conditions state that variable `varx` in statement `sx` must affect directly or indirectly the variable `vary` at statement `sy`, and also that the variable `varx` must be named `"X"` and statement `sx` must correspond to the twentieth statement. In short, this query will return the pairs of statements and variables that are affected by the variable named `"X"` at statement 20.

Listing 4.4: PQL Example 3

```
1  var vary, varz
2  statement sy
3
4  Select varz
5  such that
6      IS-IN(<sy, vary>, all_affected_by_x_at_20)
7  pattern
8      echo(varz, _*, vary, _*)
9  with
10     varz.varName=~"^some_prefix"
```

This last example shows the features of PQL that are needed to understand the examples that will follow. The `IS-IN` relationship constrains the tuple in the first parameter to the tuples returned by the query that comes in the second parameter. It can be used to create queries that further refine the results of other queries. The next two features are actually additions that were made to the language to overcome the limitations of the original PQL.

The first feature comes in line 8, in the form of the `"_*"` symbol. Its meaning is similar to the `"_"` symbol in that it says “I do not care”, but in the case of the `"_*"` symbol, instead of being valid for a single parameter, it is valid for an arbitrary number of parameters. In the case of line 8, it is used to say that `varz` is used as the first parameter of `echo` (there is no return value in `echo` so the first parameter of the pattern actually matches the first parameter of the function), followed by an arbitrary number of parameters, followed by `vary` and by another arbitrary number of parameters, i.e., `vary` is used as some parameter of `echo` except the first.

The other feature is the one in line 10 and is the `"=~"` symbol. This symbol was added to allow for matching using regular expressions instead of the exact matching that PQL allows. In the example above it constrains `varz` to variables whose name starts with `"some_prefix"`.

The whole query will thus return all variables whose name starts with `"some_prefix"` and are used as the first parameter in some `echo` function where a `vary` is also used, with `vary` being a variable affected by the `"X"` variable at statement 20.

In the next subsections we will present some location plugins for common vulnerabilities according to the analysis from Chapter 3. It is worth noting that the examples might be incomplete, and their purpose is only to give an idea of how the location plugin could be applied to real vulnerabilities. In fact, the PQL language itself has limitations for the use in the location plugins that even the modifications from this work were not able to surpass, and thus further modifications or a completely different language must be used in practice.

4.3.1 Taint-Style Vulnerabilities

Before going in depth in specific vulnerabilities, it is interesting to look first at a well known class of vulnerabilities, called taint-style [39]. This class is very relevant because four of the most common

vulnerabilities in web application, Cross-site Scripting, SQL Injection, Code Injection and File Handling, all belong to it. A taint-style vulnerability occurs when **user input** can reach a **sensitive sink** in the program without being properly **sanitized**.

The **user input** might come from a variety of sources, some of which are quite obvious such as parameters from HTML forms and URLs, and some which a normal user would not modify but an attacker might, such as hidden fields in HTML forms, HTTP header fields, and cookie data. These sources have a common characteristic, the fact that they come from the user directly in requests, which makes them obvious sources of user input, but even data originating from the server can be seen as user input. Examples of this are data coming from databases or files, since user input might have been written to them and therefore they many return user input when read.

A **sensitive sink** is a critical operation where user input must be sanitized before it is used to prevent a security compromise. While it is not required, sensitive sinks are usually functions or more precisely certain parameters of certain functions. A taint-style vulnerability occurs then if user input is used as one of the sensitive parameters of a sensitive function. The type of vulnerability comes from the type of sensitive sink, of which there are many examples, such as functions that output data to HTML (leading to Cross-Site Scripting attacks), functions that output data to HTTP headers (risking HTTP response splitting), functions that query databases (could create SQL Injection), functions that access files (leading to Path Traversal), and functions that execute shell commands (risking Command Injection).

It is thus necessary to **sanitize** the user input before it is utilized in a sensitive sink. Sanitization depends on the exact sensitive sink, e.g., sanitization of user input for functions that output data to HTML is different from functions that query databases. Typically, sanitization of user input is done in one of two ways, using language specific routines or by using regular expressions to validate and replace dangerous character sequences in the input.

In order to find taint-style vulnerabilities, taint analysis can be applied. The purpose of taint analysis is to find if there is a way for user input to reach a sensitive sink without being properly sanitized.

Its name comes from the fact that it considers user input as tainted and that values that depend on a tainted value are tainted as well, e.g., if a string is the result of the concatenation of a tainted and an untainted string, then it is also tainted. The concept of dependence is quite intuitive at first thought but a deeper look reveals some not so obvious situations.

Typical examples of dependence are a string that is the result of the concatenation of two other strings, a string that is the reverse of another string, a substring of another string or an integer that is the result of some operation over another integer. All these examples show a clear dependency between the source and result values, and taint analysis can usually assume safely that taint will propagate through these dependencies.

Less typical examples where it is not easy to define whether taint should be propagated include using an integer as an index of an array or using a boolean in a condition that sets a variable value. In these cases there is no data dependency between the source and result values, only a control dependency [40], i.e., the result value is not the result of some operation over the source value, but

is instead chosen using the source value in the decision, which is still a type of dependency, just not as strong.

The success of taint analysis is thus related to the type of dependency analysis, resulting in more false negatives if only data dependencies are considered and in more false positives if control dependencies are also taken into account.

Assuming dependency analysis has already been performed, taint analysis can be performed in two opposite ways, top-down and bottom-up.

Top-down taint analysis simulates the way taint is propagated at run-time. It begins by marking all user input as tainted and goes down the dependency tree marking all that depends on a tainted value as tainted. If a sanitization routine is found while going down the dependency tree, its branch is cut since it is not worth analyzing. If a sensitive sink is found instead, the taint analysis can store the path from the user input to the sensitive sink so that it can be presented as a new vulnerability.

Bottom-up taint analysis is the exact opposite because it starts at the sensitive sinks and goes up the dependency tree until a sanitization routine or user input is found. While both achieve the exact same results, bottom-up is more common, e.g., RIPS and Pixy both use it. There are two main reasons for this: the first being that bottom-up taint analysis is slightly easier to use, since sensitive sinks are usually only functions and user input might also come from tainted variables such as GET and POST in PHP, which simplifies the discovery of the starting points. The second reason is due to performance, which is better in a bottom-up method due to the fact that sanitization routines usually occur nearer to the sensitive sinks then to the user input. This occurs because as explained before, sanitization is different depending on the sensitive sink, and thus programmers usually put the two together, which in turn results in the cuts in the dependency tree to occur earlier, leading to a smaller search and higher performance.

Taint analysis can also be used to assist in the injection of vulnerabilities. Assuming bottom-up taint analysis, if the dependency tree is searched up and a sanitization routine is found, vulnerability detection would cut the tree and backtrack.

If the purpose is to find where can vulnerabilities be injected then the trick is to not stop at sanitization routines but instead keep going up the tree trying to find user input. If it is eventually found, taint analysis can conclude that there is a dependency path between user input and a sensitive sink that is protected by one or several sanitizations.

Injection is then as simple as the removal or tampering of the sanitizations, since that would clear the path for the user input to reach the sensitive sink.

This concludes the explanation of taint-style vulnerabilities and taint analysis. It is now possible to use this information to define the necessary plugins to inject vulnerabilities that fall into this category.

Listing 4.5: Query template for taint-style vulnerabilities

```
1  @BEGIN
2      function echo_all(&target, $bool_op, $pre, $pos, $vars[]) {
3          for ($i = 0; $i < card(&target); $i++) {
4              if ($i > 0)
5                  echo $bool_op
6              echo $pre . &target[$i]($vars[]) . $pos
7          }
8      }
9  @END
10
11 view @ echo &name
12
13 statement stmt_user_input, stmt_sanit, stmt_sink
14 var ui, var1, var2, var3
15
16 Select <stmt_ui, stmt_sanit, stmt_sink>
17 such that
18     Affects*(stmt_ui, stmt_sanit, ui, var1)
19 and
20     Affects*(stmt_sanit, stmt_sink, var2, var3)
21 pattern
22     (
23         @ echo_all(&sanit.func, "or", "", "", ["var2", "var1"])
24     )
25     and
26     (
27         @ echo_all(&sink.func, "or", "", "", ["var3"])
28     )
29 with
30     @ echo_all(&user_input.regex, "or", "ui.varName=~", "", [])
```

Listing 4.6: Language translator for taint-style vulnerabilities

```
1  <translator>
2      <language>PHP</language>
3      <name>taint_style</name>
4      <query>taint_style</query>
5      <user_input>
6          <regex>"\$_GET\[.*\]"</regex>
7          <regex>"\$_POST\[.*\]"</regex>
8          <regex>"\$_COOKIE\[.*\]"</regex>
9      </user_input>
10 </translator>
```

The two listings above represent the template query and language translator for generic taint-style vulnerabilities, respectively. The template query is mostly a normal query with template code embedded. Template code can be found in individual lines after a @ symbol or in blocks between the *@BEGIN* and *@END* statements. The language of template code has similarities to PHP but no effort has been made to make it a formal language, meaning it is to be interpreted as pseudo-code.

Access from the template code to the language translator is done through the use of variables preceded by the symbol `&`, which represent references to elements of the language translator. For example, for the language translator above, using the variable `&user_input.regex` will return a reference to the `regex` elements inside the element `user_input`, which can be accessed as an array. Line 11, for example, echoes the variable `&name` which corresponds to the element `name` from the language translator.

The language translator of listing 4.6 does not provide enough information for the query, such as the sanitization functions and sensitive functions, accessed with the `&sanit.func` and `&sink.func` variables respectively. The reason is that it is so generic that all it can specify are the user input variables which are common to all vulnerabilities. Translators specific for each vulnerability must exist to allow the injection of the corresponding vulnerability. These will be presented in the subsection respective to each vulnerability.

Let's now look at the query template in detail. The template starts with a block of template code in line 1 that defines a function `echo_all`. This function receives a reference to some elements called `&target`, a boolean operator called `$bool_op`, two strings called `$pre` and `$pos` and an array of variables called `$vars[]`. It then loops through all elements of the `&target` reference using the function `card` that returns the number of elements of the reference. It then proceeds to printing each element of `&target` preceded by `$pre` and followed by `$pos` and between each of the lines printed it prints the boolean operator `$bool_op`.

The call from line 30 for example prints the following:

Listing 4.7: Example of the `echo_all` from line 30

```
1 ui.varName=~"\$_GET\[.*\]"
2 or
3 ui.varName=~"\$_POST\[.*\]"
4 or
5 ui.varName=~"\$_COOKIE\[.*\]"
```

The `echo_all` function can then be used to print all elements of a reference to the language translator, while putting some boolean operator between each line, and also adds some flexibility by allowing to print something behind and in front of each printed element.

The only parameter that was not explained was the `$vars[]` array. The array is passed as a parameter to the each index of the reference `&target`. This allows passing variables to the references that are inserted in predefined places in the referred element.

Suppose a language translator has an element such as `<el>f($1, _, $2)</el>` and it is called like this `&el("vx", "vy")`. When the query generator parses the reference what will be generated is `f(vx, _, vy)`, since `$1` will be replaced by the first parameter of the reference and `$2` will be replaced by the second.

This is used when printing the lines with the sanitization functions and sensitive sinks, to put the variables in the right parameter of the printed patterns since they are not always the same.

Looking at the whole template query, we see that it starts by defining `statement` variables for the user input, the sanitization function and the sensitive sinks. These are the triples that the query returns, since they are the main elements necessary to verify that the vulnerability can really be injected. Actually, it would be more useful to return the whole dependency path for each location found but one of the limitations of PQL is not allowing tuples with arbitrary number of elements to be returned, which would be required for this case.

The query also declares four variables to be used throughout the program. The `ui` variable matches the variables that contain user input due to the **`with`** condition of line 30.

Variables `var1` and `var2` match the relevant parameter and the return value respectively of a sanitization function due to the `echo_all` of line 23. Actually, this depends on the language translator and how it places the parameters but it is safe to assume that it will put them in the right place. Actual examples of the language translators will be presented latter.

As for `var3` it corresponds to the variable using in the sensitive parameter of a sensitive sink according to line 27.

The remaining parts of the query that have to be analyzed are the `Affects*` relationships from lines 18 and 20. What they mean is just that the user input variable `ui` must affect the `var1` variable used as the parameter to a sanitization function, and that the `var2` variable that is returned from the sanitization function affects the `var3` variable that is used as the parameter of a sensitive sink.

This is the formal explanation of a taint-style vulnerability, which is what the query generated by this template will find.

The next two subsections will go in more detail on two of the most common vulnerabilities of the taint-style class and will show examples of query templates and language translators for each.

4.3.2 Cross-Site Scripting

Cross-Site Scripting (XSS) is an attack where a malicious script that runs in the client side is injected into the code of a website trusted by the target user. The purpose of this attack is to have the user's browser run the malicious script as if it was running a script originating from the trusted website. This gives the malicious script the same permissions as any script from the trusted website, such as access to the cookies, session tokens, etc. which the it can then send back to the attacker. The script might even modify the HTML content returned by the trusted website, allowing more sophisticated attacks.

There are three variants of this attack, each requiring slightly different vulnerabilities in order to succeed.

Reflected XSS – The vulnerability here is that the server uses untrusted input from a web request to create the returned page without first validating it.

In order to exploit this vulnerability the attacker creates a web request, usually in the form of an URL, where the untrusted input (that will be used by the server to create the page) is the script that he wants to run in the user's browser. Now, if he manages to trick his target into following that URL, the browser will send the request containing the malicious script and the server will reflect the script into the page, which the browser will run since it came from a trusted site.

Stored XSS – In Stored XSS, the server is vulnerable as long as it stores some untrusted input in a persistent form and later on uses it to create a page without validating it first.

This is much similar to the Reflected XSS except for the fact that the injection is permanent, that is, while in the Reflected XSS, after an attack, subsequent normal requests would not be attacked, in the Stored XSS the injection is stored into the server and each subsequent request for the injected page will include the malicious script.

This makes the attack more powerful since the user now does not even have to be tricked into following a crafted URL, the attacker can just exploit the vulnerability by injecting the script into the page and wait that the users browse the vulnerable page.

DOM based XSS – This variant is the most sophisticated one. It requires that the server includes some client side script, which accesses the DOM environment, while failing to validate the untrusted values of the DOM environment.

The attacker might thus be able to modify the DOM environment in such a way that even the unmodified script received from the server will run in an unexpected way that is advantageous for the attacker.

One example is if the server contains a client side script that gets the value of a parameter username from the URL and puts it into the HTML. The attacker can now create a URL where the parameter username contains a malicious script, and if he manages to trick a user to follow it, the user will receive the original page, with no injections, but the client side script itself will pick the malicious script from the URL and insert it into the HTML, thus allowing the same types of attacks as the other variants.

These three variants differ basically on the source of user input, which might be request parameters in the case of Reflected XSS, databases and files in the case of Stored XSS, and DOM environment in the case of DOM based XSS. Modeling these three types of XSS is thus very easy because they only require adding new types of user input to the language translator for taint-style vulnerabilities.

The recommended preventions against XSS [41] are the following:

Input Validation – Input validation is a prevention against many vulnerabilities, not XSS in particular, but will be explained here since this is the first vulnerability presented in this thesis where it applies.

This technique applies a white-list to the input in order to make sure it only contains acceptable characters or words. While this might prevent many attacks, it has a clear drawback which is that when the input is received it might not be possible to know where it will be used and thus the white-list will need to be overly conservative in order to prevent any bad utilization latter on.

For example, single quotes might not be allowed due to fear of SQL Injection, preventing valid names such as O'Reilly from being used in the system. Due to these limitations, input validation is not enough to protect against XSS by itself without compromising usability, and therefore should be used together with escaping with the only purpose of providing defense in depth.

In order to compromise as little usability as possible while still providing some security, it should allow characters if and only if they are valid in the given context (i.e., allow letters, spaces and single quotes in names, and allow only numbers in an age field).

Escaping – Escaping is used to ensure that data being used is interpreted as data by the parser and not as code. It can prevent XSS, SQL injections, Code injection, and many other vulnerabilities if the data passed to the sensitive sinks of each vulnerability is escaped.

Escaping is used normally only on characters that have syntactic meaning in the context where the data is going to be used. Those characters are converted to a representation that the parser for that context recognizes so that it can use the character as data and not as a meta character with syntactic meaning.

One very common example of escaping is the one used to represent the quote (`"`) character in strings of languages that use the quote as the delimiter of a string. The escaping in that case is done by putting a slash (\) before the quote as in `\"`, which tells the string parser that that quote is not the end of the string but is actually a data quote.

Focusing on the case of XSS, escaping solves the vulnerability because even if a script is embedded in the data being output, it will be interpreted as data and not as the script that the attacker intended. The subtlety lies in knowing how to escape the data before outputting it.

The answer is that it depends on the context where the data is being output. Outputting to HTML, CSS, or JavaScript requires different escaping since the characters with syntactic meaning of each are different. Care must be taken that both CSS and JavaScript can be embedded in HTML in the form of attribute values.

The HTML locations where data must be escaped differently are those below. In the examples, the `...` represents the location being exemplified.

- Elements content – `<body>...</body>` or `<div>...</div>`
- Common attributes – `<div id="...">` or `<form method="...">`
- JavaScript data values – `<script>x='...'</script>` or `<div onmouseover="x='...'">`
- Style property values – `<style>selector{property:...;}</style>` or `<span style="property:...">`
- URL parameter values – `<a href="http://www.somesite.com?param=...">`

Data output to other HTML locations cannot be properly escaped and thus it should be avoided at all costs.

With this information it is now possible to create both language translators and query templates for XSS. The following examples are only valid for XSS vulnerabilities prevented through escaping.

Listing 4.8: Language translator for generic XSS

```
1  <translator>
2      <language>PHP</language>
3      <name>cross_site_scripting</name>
4      <parent>taint_style</parent>
5      <sinks>
6          <func>echo(_*, $1, _*)</func>
7          <func>printf(_, $1, _)</func>
8          <func>printf(_, _, _*, $1, _*)</func>
9      </sinks>
10 </translator>
```

Listing 4.9: Language translator for XSS on HTML elements

```
1  <translator>
2      <language>PHP</language>
3      <name>cross_site_scripting_HTML</name>
4      <query>cross_site_scripting_HTML</query>
5      <parent>cross_site_scripting</parent>
6      <sanit>
7          <func>htmlspecialchars($1, $2, _*)</func>
8          <func>htmlentities($1, $2, _*)</func>
9          <func>ESAPI.encoder().encodeForHTML($1, $2)</func>
10     <sanit>
11 </translator>
```

The two listings above represent the language translators for generic XSS and XSS on HTML elements. In order to understand them it is important to understand first how the Query Generator parses the language translators.

The Query Generator starts by creating trees of language translators using the parent element of each. For example, in the case of the three language translators presented so far in listings 4.6, 4.8 and 4.9, the one named `taint_style` would be the root of the tree with the `cross_site_scripting` as its child and `cross_site_scripting_HTML` as its grandchild. If they had been presented also, language translators for other variants of `cross_site_scripting` would also be children to it, while `sql_injection` and `code_injection` would be children of `taint_style`.

The Query Generator then receives information about which vulnerabilities to inject through the configuration. It uses this information to know which language translator to use, and then merges that language translator to all its ancestors and descendants in the tree. This merging allows for each language translator to have only the information that is common to all its descendants instead of requiring it to have all information that is needed if the user wants to inject that specific vulnerability.

If the user wants to inject a generic vulnerability such as `taint_style`, all information about the descendants is imported, meaning all types of vulnerability get injected. On the other hand, if he only wants to inject a specific vulnerability such as `cross_site_scripting_HTML`, the information specific to it is used plus all the information that its ancestors have because it is common to it.

It is now easy to understand these language translators. The one for `cross_site_scripting` has `taint_style` as his parent and defines the sensitive sinks because they are common to all variations. The one for `cross_site_scripting_HTML` has `cross_site_scripting` as its parent and defines the sanitizations functions since they are specific of XSS on HTML elements but not of other children of `cross_site_scripting`.

Another subtlety must be noted. While for `cross_site_scripting` the `taint_style` query is enough, in `cross_site_scripting_HTML` information is required about where the data is being output to, something that the `taint_style` query does not verify. This is why there is a query

element in cross_site_scripting_HTML while there is none in cross_site_scripting. The Query Generator will look for a query element in the language translator it uses and if none is found will search in its ancestors until one is discovered. The value of the query element specifies which query to use to search for that vulnerability. Since in cross_site_scripting there is no query element, it will use the one in its parent, while cross_site_scripting_HTML has a query element which is the one that is utilized.

Listing 4.10: Query template for XSS on HTML

```
1 view cross_site_scripting_HTML
2
3 statement s1, s2, s3
4
5 Select <s1, s2, s3>
6 such that
7     IS-IN(<s1, s2, s3>, taint_style)
8 and
9     Outputs_To_HTML_Element(s3)
```

The listing above is the query template for searching for cross_site_scripting_HTML vulnerabilities. It is very simple as all it does is returning the same values as the taint_style query, only constraining the statement s3 to statements that output to an HTML element.

The relationship Outputs_To_HTML_Element requires string analysis to be performed in the source code, followed by HTML analysis to determine the places where variables are output in the HTML that is built by the application. As long as these analyses are performed beforehand by the middle-hand the relationships can be safely used in the queries.

4.3.3 SQL Injection

A SQL Injection is an attack where the adversary manages to create a SQL query of his choice or at least modify an existing SQL query to his advantage. The consequences of a SQL injection might range from being able to read sensitive data from the database to modifying it or even deleting the database completely.

The attack is possible due to a taint-style vulnerability, where user input is used to construct a SQL query without being validated.

Listing 4.11: Example of a SQL Injection vulnerability in PHP using MySQL

```
1  $dbLink = mysql_connect('localhost', 'username', 'password');
2  mysql_select_db('db');
3  $query = "SELECT private_data
4            FROM   user_table
5            WHERE  user_name ='".$_POST["username"]."'
6            AND    password  ='".$_POST["password"]."'";
7  $result = mysql_query($query);
8  $row = mysql_fetch_row($result);
9  echo $row[0];
```

The example above is a typical example of a vulnerable piece of PHP code. The developer created this code so that the user had to give the right username and password pair to access his private data, thus preventing users from accessing private data that does not belongs to them.

However, due to the SQL Injection vulnerability, this is not what happens. If an attacker fills his username as "`admin'--` ", since `--` is a comment in MySQL, the SQL query will stop at the username and will not even compare the password. With such a simple attack, the attacker gained access to the private data of the user without knowing its password.

In order to prevent SQL Injection vulnerabilities, one of the following prevention mechanisms [42] should be used:

Prepared Statements – Using prepared statements is the best way to avoid SQL Injections. A prepared statement is just a SQL query that is prepared before the parameters are passed to it. This allows separation of what is the actual query and what is the data that is being passed to it as statements.

Using prepared statements with MySQLi, the vulnerable example above would become like this:

Listing 4.12: Example of how to use prepared statements to prevent SQL Injection

```
1  $mysqli = new mysql('localhost', 'username', 'password', 'db');
2  $stmt = $mysqli->prepare("SELECT priv FROM testUsers WHERE
3                            username=? AND password=?";
4  $stmt->bind_param("ss", $_POST["username"], $_POST["password"]);
5  $stmt->execute();
6  $stmt->bind_result($result);
7  $stmt->fetch();
8  echo $result;
```

In this case, the SQL query is first prepared in line 2 and the parameters are only bound to the query at line 4, allowing MySQLi to distinguish the query from its parameters. Besides, the first parameter of the `bind_param` function, `"ss"`, limits the values of the parameters to being strings, which further adds to the security.

Stored Procedures – Stored procedures are the next best solution besides prepared statements. Under normal use they are very similar, since the SQL query is also prepared in advance and the parameters are passed to the query separately, not allowing for confusion. The only problem with stored procedures is that they also allow for dynamic generation of SQL inside the stored procedure, meaning that the unaware developer might use that functionality and fall in the same mistake as described previously.

Escaping – Sometimes using prepared statements of stored procedures is not possible or is not practical, since it would require modifying too much code. In those cases it is still possible to prevent SQL Injection vulnerabilities by escaping the data before adding it to the query.

Since escaping was already explained in the previous section it will not be explained here. However, one important point must be made: escaping is once again dependent on the context, which in this case is the database being used, such as Oracle or MySQL, and the fact that the data is being put between quotes or not.

While making a template query and language translators for the prepared statements or stored procedures is not impossible, it is too complex to use as an example, and as such only the escaping will be shown.

The language translators are very similar to the ones for Cross-Site Scripting, the only parts that would change are the sensitive sinks that would include functions such as `mysql_query` and `mysql_db_query` and the sanitization functions which would have functions such as `mysql_real-_escape_string` and **`addslashes`**.

Listing 4.13: Query template for the SQL Injection with quotes vulnerability

```
1  view sql_injection_quoted
2
3  statement s1, s2, s3
4
5  Select <s1, s2, s3>
6  such that
7      IS-IN(<s1, s2, s3>, sql_injection)
8  and
9      Outputs_Inside_Quotes(s3)
```

The query template would also be very similar, requiring string analysis to verify whether the input to the query is or not inside quotes.

Chapter 5

Prototype Implementation & Evaluation

In order to evaluate the architecture presented in Chapter 4 a prototype was implemented and evaluated. The prototype was based on Pixy [43] to leverage its strong static analysis capabilities.

5.1 Prototype

The architecture from Figure 5.1 shows the original Pixy architecture in dark with the modifications in red. Pixy is written in Java and its original architecture consists of a Checker that corresponds roughly to the Main module of the architecture described in Chapter 4, a Program Converter that corresponds to the front-end, an Analyzer corresponding to the middle-end, and a DepClient per vulnerability that does the work of both the Vulnerability Location Seeker and Vulnerability Injector.

The Checker is thus the first to run and is responsible for receiving and parsing the input from the command line, which consists of the list of files to analyze and some configurations. It then proceeds to read the model and sink files, where the model files contain the sanitization functions for each vulnerability and models of functions describing how the taint propagates through each. The sink files contain the sanitization functions per vulnerability. For each vulnerability a DepClient is created and initialized with the information taken from the model and sink files. Which DepClient is created per vulnerability depends on a table that is hardcoded in Pixy: an improvement would be to put this table into a configuration file to allow for easy addition of new vulnerabilities to detect, but for the purpose of a prototype this was not needed.

The ProgramConverter is called by the Checker and runs the source code through a PHP lexer and parser. It then converts the parse tree into a variation of TAC (three address codes), which the rest of Pixy uses for analysis instead of the original source code.

The Analyzer is not a real module of Pixy because it is just a method of the Checker class. We describe it as such to allow for comparison with the generic vulnerability injector architecture. The

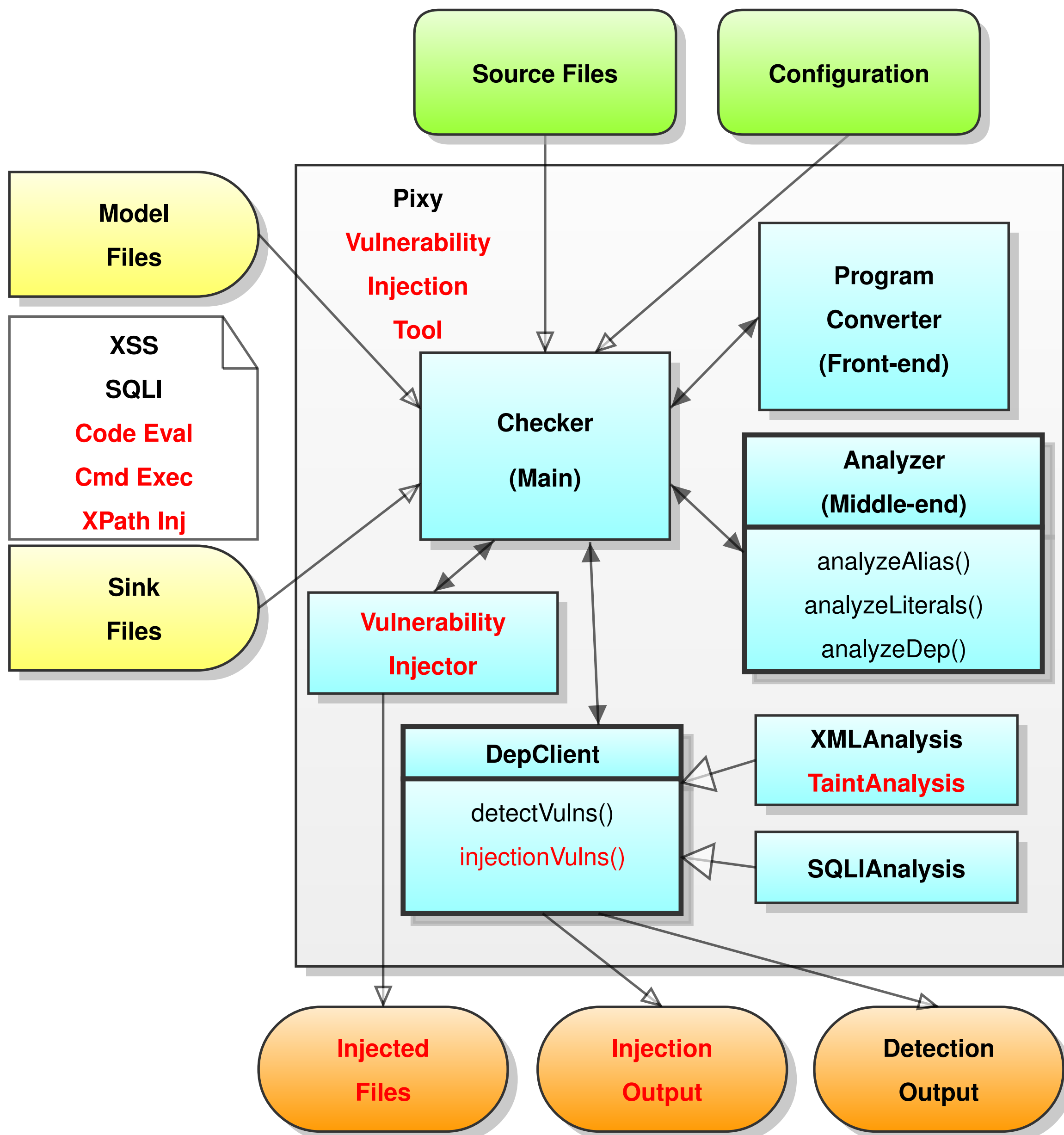


Figure 5.1: Pixy architecture plus modifications for Injection Tool

Analyzer is responsible for doing the static analysis over the TAC representation of the source code. Its main purpose is to create the dependency graphs that are used for taint analysis and thus it always performs dependency analysis, but it can also do alias and literal analysis if they are activated in the configurations.

The DepClients are then run in turn to detect each type vulnerability. Each DepClient has a `detectVulns()` method that performs taint analysis in order to find the vulnerabilities. It does so by collecting the sinks and going up the dependency graph of each sink until either sanitization functions or user input is found. For each found vulnerability it prints some information and outputs the dependency graph to provide a better understanding of why the vulnerability exists.

The modifications made to Pixy were the ones marked in red in Figure 5.1. The addition of a `injectVulns()` method in the DepClients and of a Vulnerability Injector module were required in order to transform it into a Vulnerability Injection Tool. The rest of the modifications were made in order to allow for the Vulnerability Injection Tool to be able to inject and detect some more vulnerabilities.

The `injectVulns()` method is called by the Checker, instead of `detectVulns()`, whenever the user specifies in the configurations that he wants to create instead of discover vulnerabilities. It performs taint analysis similar to `detectVulns()` although it had to be modified in order to find out the locations where the vulnerabilities can be injected.

The modification consisted in making the taint analysis not stop whenever a sanitization function was found in the dependency graph of a sink, and have it store that sanitization function in a list and keep searching for user input. If user input is eventually found, an InjectionLoc object is created containing all information relevant to this potential place to insert a vulnerability, such as the user input source, the sensitive sink, all sanitization functions found between them, and the type of vulnerability that can be injected. This object is then added to a list which is returned when the method is done, resulting in a total of one list per DepClient.

These lists are passed by the Checker to the Vulnerability Injector module that goes through each list and performs the actual injection in the files. The injections made in the prototype are limited to the removal of the sanitization functions received in each InjectionLoc, but more complex injections are possible by just modifying the Vulnerability Injector.

Unfortunately, Pixy only allows the transformation of the source code into a parse tree, not the opposite, so it was not possible to make the modifications in the parse tree and rebuild the modified file from it, which would be more precise and versatile than making the modifications directly in the source code.

In order to add a little extensibility to Pixy some more modifications were made. First, the XMLAnalysis class was modified to represent a generic TaintAnalysis. Although the analysis made by XMLAnalysis was just as simple as the generic TaintAnalysis, it solved some existing limitations, such as having hardcoded sensitive sinks, echo and print, and only allowing one vulnerability per analysis class, making it impossible to use the generic TaintAnalysis for many vulnerabilities.

The next step was to add new model files and sink files for new vulnerabilities and add the new vulnerabilities to the table that links vulnerabilities to the analysis class. The information for the new vulnerabilities was adapted from the RIPS tool because it had much better capabilities than Pixy to detect a diverse class of vulnerabilities.

Unfortunately, RIPS has much information on the sensitive sinks of each vulnerability but its information on the sanitization functions was very limited, and since the Vulnerability Injection Tool requires information on the sanitization functions to inject vulnerabilities, only the vulnerabilities that contained this information were added. In the end, it was possible to have the Vulnerability Injection Tool detect and inject three new vulnerabilities, Code Evaluation, Command Execution and XPath Injection, which occur when user input is used in an unsanitized way in a function that evaluates PHP code, shell code or XPath code, respectively.

5.2 Evaluation

The prototype was evaluated with four applications in order to measure its efficiency. The four applications are a guestbook script called Talkback [44], a webchat server named Voc [45], a document management system called yaDMS [46], and RIPS [47], the vulnerability scanner that contributed to the model and sink files of the prototype.

The selection of these applications was based in three requirements that had to be satisfied. The first was that the prototype had to be able to parse the application files, which was not always true because the original Pixy could not parse PHP5 applications, and thus neither could the prototype. The second is the equivalent for RIPS, i.e., RIPS had to be able to parse the application files too. This was required because RIPS was to be used to cross-check the results of the injections. The third requirement was that the applications were easy to run and test, which was required to verify whether the injections affected the apparent behavior of the application or not.

During the evaluation phase many problems arose that required corrections on the prototype so that it worked as expected. Some of these were bugs from the original Pixy or the modified prototype but the most common problem was with functions that were not yet considered in the model files of Pixy. The function models had to be added to the model files, but the way they were implemented did not make this tasks easy to accomplish.

The problem was that most functions that had to be modeled were common to all vulnerabilities, i.e., the taint propagated in the same way for all vulnerabilities. This required adding a new function model to all model files, which was quite bothersome with the five vulnerabilities that the prototype could inject. To solve this problem, a model file common to all vulnerabilities was created and the prototype was modified so that for each vulnerability it would read first the common models and only then the specific ones. This allowed for models that differed in some vulnerabilities to override the common models.

Another problem with the model files occured when a function had parameters that propagated taint among each other, as the original Pixy did not support this (it only allowed taint to be propagated from one parameter to the return value). However, the addition of this functionality would require a complete revamp of the model files syntax and thus it was not done, making the injections a little less precise whenever those functions appear.

Moreover, it was found during this evaluation that the literal analysis on Pixy was completely unable to solve even the most simple includes as long as they required guessing the contents of an array. It would fail to resolve an include even if it was as simples as

```
$v['a']='foo';
include($v['a'].'bar.php').
```

The solution for this problem was to resolve the includes by hand in order to avoid having to modify the literal analysis of Pixy.

After all these problems were solved or worked around, the injections were performed according to Figure 5.2, where XSS corresponds to Cross-Site Scripting, SQLI means SQL Injection, CODE_EVAL indicates Code Evaluation and CMD_EXEC means Command Execution.

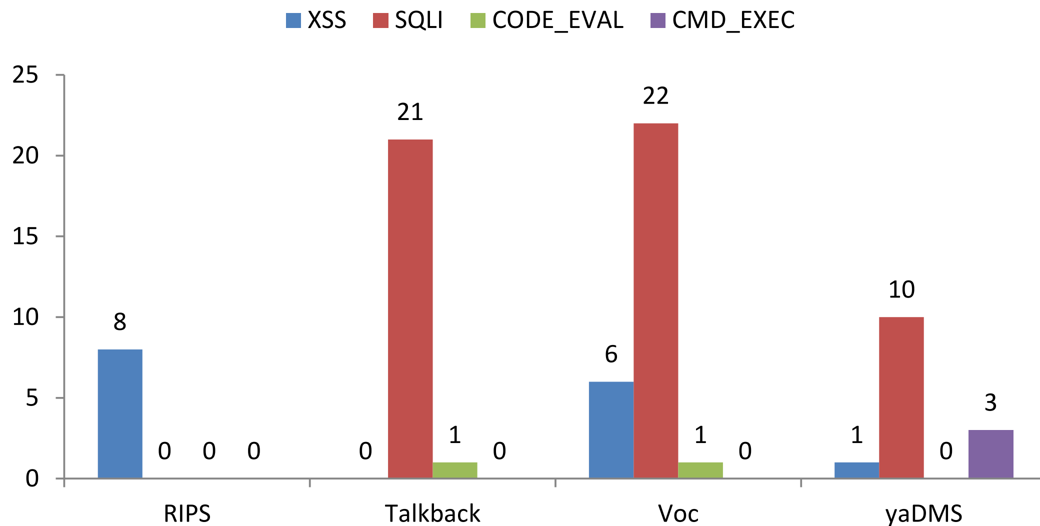


Figure 5.2: Vulnerabilities successfully injected in each application

A remark must be made on the disparity in the number of successful injections for each type of vulnerability, which ranges from dozens of SQL Injections to only a couple of Code Evaluations and not even one XPATH Injection. There are two main reasons for this to happen.

The first one is related to the frequency with which each type of sink appears in applications, e.g., sinks for Code Evaluation vulnerabilities, which evaluate strings as PHP code, are obviously less common than sinks for SQL Injections, which are very common database queries.

The second reason has to do with how well is each vulnerability modeled, e.g., RIPS had a lot of functions that represented sinks and sensitive functions of SQL Injections, and very little for all other vulnerabilities. Therefore, this made the injection of SQL Injection vulnerabilities more successful than other injections because these models were used in the prototype. RIPS was the only application without any injection of SQL Injection vulnerabilities because it did not use databases to begin with.

To ensure that the injections did not affect the normal operation of the applications, the applications were tested after they were injected. Since only sanitization functions were removed the result was as expected, and they all had the same apparent behavior after the injection.

The injections were thus successful, so the only thing left to verify was that the injections corresponded to actual vulnerabilities. To do this, RIPS was run over each application in search of vulnerabilities both before and after inserting the vulnerabilities. The results are in Table 5.1.

The values presented in Table 5.1 correspond to the information provided by RIPS, and therefore include both vulnerabilities that were correctly and incorrectly discovered.

While most values from the above table are more or less expected, there is an outlier on the number of XSS vulnerabilities found in yaDMS. This number is very high due to a dozen of echo statements that are embedded in JavaScript instead of the usual HTML. RIPS cannot distinguish this because it does not recognize JavaScript and thus assumes that they will result in a XSS vulnerability, which is false.

Table 5.1: Vulnerabilities indicated by RIPS before/after injection

	RIPS	Talkback	Voc	yaDMS
XSS	3/14	19/19	3/3	88/89
SQLI	0/0	5/10	1/3	7/16
CODE_EVAL	0/0	0/0	2/2	1/1
CMD_EXEC	0/0	0/0	0/0	2/6

This table also shows a problem that this evaluation suffered from, which is related to the fact that the applications were already vulnerable even before the injection was performed. Since the prototype's approach is to remove sanitizations, restoring vulnerabilities that had been prevented, each vulnerability that is not prevented in the application is one less vulnerability that the prototype is able to inject. In other words, if an application has $n + m$ potential vulnerabilities of which n are prevented and m are not, after injection there will always be $n + m$ vulnerabilities in the application. However, the lower n is, the less vulnerabilities the prototype can inject. The number of injected vulnerabilities in a vulnerable application is therefore lower than it could be if all vulnerabilities had been prevented.

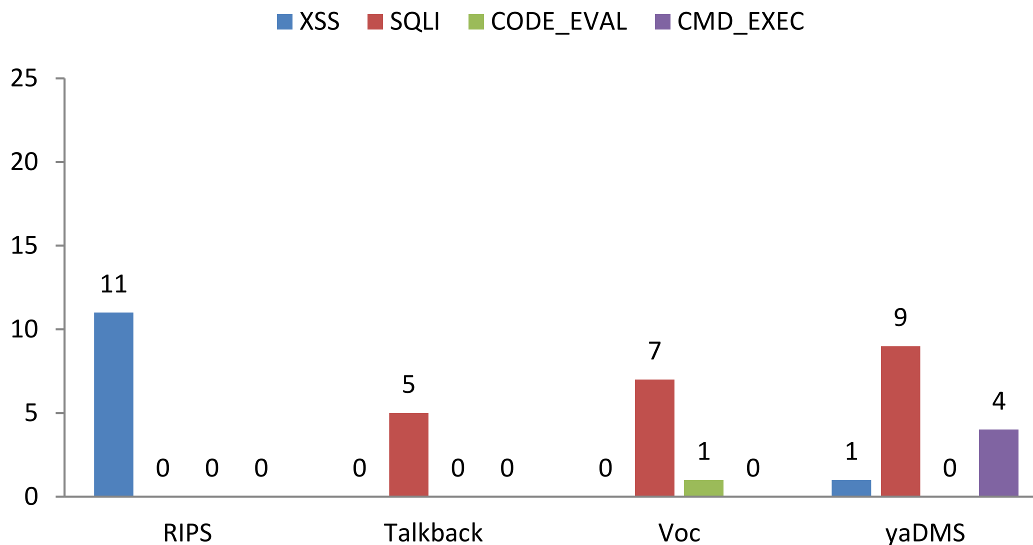


Figure 5.3: Difference of the vulnerabilities detected by RIPS before and after injection

Figure 5.3 represents the difference in the number of detected vulnerabilities by RIPS before and after the injections were done with the prototype. As expected, the types of vulnerability that RIPS detected after the injection (but not before) match the types of vulnerability that were injected (see Figure 5.2). However, there are some discrepancies in the numbers that can be explained by the following facts.

First, RIPS only makes a single pass over the source code, meaning it will fail to model the functions

behavior if they are used before they are declared, while Pixy has a much more complex front-end that allows it to inject vulnerabilities that RIPS cannot detect.

Also, the interpretation of what is counted as a vulnerability differs between RIPS and the prototype, since the prototype only counts one vulnerability per sensitive sink and RIPS counts one vulnerability per each pair of user input and sensitive sink. This is the why there are some cases where the number of vulnerabilities detected by RIPS after injection is even higher than the number of vulnerabilities injected.

Finally, it could have also be explained due to the conservative nature of the prototype. If a unsanitized value is stored in a database or a file and is latter read from it and used in a sensitive sink, it causes a vulnerability. The original Pixy however cannot detect this because it cannot keep track of whether a file is tainted or not, and thus considers all files to be untainted. In the prototype the opposite policy was chosen, i.e., every value read from a file is considered tainted. While both options have their advantages and disadvantages, this option is more conservative and thus is able to inject and detect more vulnerabilities, at the price of an higher rate of false positives. Since RIPS does not consider files as tainted, it will not consider as vulnerabilities some of the injections from the prototype, which is the reason for the discrepancies in the number of SQL Injection vulnerabilities injected by the prototype and detected by RIPS.

In order to understand exactly the reason behind the discrepancies, there was the necessity of knowing whether the vulnerabilities inserted were actually attackable. To find this out, the vulnerabilities were analyzed by looking at the code and when this was not enough, by trying to exploit the vulnerability. The result was that almost all vulnerabilities were found to be attackable, with the exception being vulnerabilities that require that a file or a database contains tainted data when there is no way to insert tainted data in it. We can thus conclude that the prototype is able to inject vulnerabilities that RIPS is unable to detect.

Chapter 6

Conclusions and Future Work

6.1 Conclusion

The thesis has described a vulnerability injection tool that can insert realistic and attackable vulnerabilities. In fact, since PHP is such a challenging language, this gives us some level of confidence that it should be possible to inject vulnerabilities in other languages.

The study on vulnerability distribution has shown that the majority of the most common vulnerabilities on web applications belong to the class of taint-style vulnerabilities. For this reason, the prototype that was developed focuses on this class of vulnerabilities. It is true that some of those vulnerabilities have variants that cannot be injected using only taint analysis, but that is only a limitation of the prototype. The proposed architecture allows for far more complex injections than were actually implemented in the prototype, making it much more powerful.

Another interesting result of the thesis was the comparison of vulnerability injection to vulnerability detection. It was found during this work that they had many points in common to each other, to the point that much of the research on vulnerability detection can be reused for vulnerability injection. In fact, the only module of the architecture that cannot profit from research on vulnerability detection was the Vulnerability Injector, since vulnerability discovery usually does not involve program transformations like vulnerability injection does.

The thesis thus concludes that despite being a relatively new research area, vulnerability injection is a viable technique to improve the security of applications, for example by testing vulnerability scanners, support educational activities, and evaluate defense in depth mechanisms.

6.2 Future Work

While this work was done to the best of our ability, there were still many tasks in the end that had to be left undone. This section enumerates some of these tasks, such that someone interested might know how to continue the work that has been done in this thesis.

- Use a front-end that can not only parse source code into an Abstract Syntax Tree (AST), but also pretty print an AST back into source code. Use it then to make the Vulnerability Injector inject the vulnerabilities in the AST and pretty print it to get the injected source codes since this allows for much more precise and complex injections. One example of a front-end that can do this is phc [48], the open source PHP compiler.
- Further study the most common types of static analysis in order to find out how can they actually be implemented as plugins.
- Find or define a language to be used for the location plugins. Although it is not required, program query languages are a good choice since they match the querying nature of the Vulnerability Location Seeker. The difficulty lies in finding a program query language that is complex enough to describe the vulnerabilities that are to be injected, allows for analyses pre-made by the middle-end to be used, and is not limited in the type of results it returns so that the Vulnerability Injector can use those results to make complex injections.
- Find or define a language to be used for the injection plugins. Transformation languages such as Stratego [49] or TXL [50] are good examples of the type of language required, but further research must be made to see whether they fill all the requirements needed for the injection plugins.
- Allow the Vulnerability Injection Tool to generate and execute attacks automatically for each vulnerability injected. This can be used to verify whether the injections actually correspond to attackable vulnerabilities, thus removing most false positives.

Bibliography

[1] S. McConnell. 'Gauging software readiness with defect tracking'. In: *Software, IEEE* 14.3 (1997), pp. 136–135. ISSN: 0740-7459.

[2] *WebGoat*. URL: https://www.owasp.org/index.php/Category:OWASP_WebGoat_Project.

[3] P. Biggar and D. Gregg. 'Static Analysis of Dynamic Scripting Languages'. Aug. 2009.

[4] *Usage of server-side programming languages for websites*. Feb. 2011. URL: http://w3techs.com/technologies/overview/programming_language/all.

[5] J. Fonseca, M. Vieira and H. Madeira. 'Training Security Assurance Teams using Vulnerability Injection'. In: *2008 14th IEEE Pacific Rim International Symposium on Dependable Computing*. IEEE. 2008, pp. 297–304.

[6] J. Fonseca, M. Vieira and H. Madeira. 'Vulnerability & attack injection for web applications'. In: *Dependable Systems & Networks, 2009. DSN'09. IEEE/IFIP International Conference on*. IEEE. 2009, pp. 93–102.

[7] J.V. Carreira, D. Costa and J.G. Silva. 'Fault injection spot-checks computer system dependability'. In: *Spectrum, IEEE* 36.8 (Aug. 1999), pp. 50 –55. ISSN: 0018-9235. DOI: 10.1109/6.780999.

[8] *Compiling phases*. URL: http://en.wikipedia.org/wiki/Compiling.

[9] N. Chomsky. 'Three models for the description of language'. In: *Information Theory, IRE Transactions on* 2.3 (1956), pp. 113–124. ISSN: 0096-1000.

[10] Etienne Kneuss. 'Static Analysis for the PHP Language'. 2010.

[11] N. Jovanovic, C. Kruegel and E. Kirda. 'Precise alias analysis for static detection of web application vulnerabilities'. In: *Proceedings of the 2006 workshop on Programming languages and analysis for security*. ACM. 2006, pp. 27–36. ISBN: 1595933743.

[12] Y. Xie and A. Aiken. 'Static detection of security vulnerabilities in scripting languages'. In: *15th USENIX Security Symposium*. 2006, pp. 179–192.

[13] J. Dahse. 'RIPS - A static source code analyser for vulnerabilities in PHP scripts'. In: Aug. 2010.

[14] *Graphviz - Graph Visualization Software*. URL: http://www.graphviz.org/.

[15] Alfred V. Aho, Ravi Sethi and Jeffrey D. Ullman. *Compilers: principles, techniques, and tools*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1986. ISBN: 0-201-10088-6.

[16] *WordPress*. URL: http://wordpress.org/.

[17] *phpBB*. URL: http://www.phpbb.com/.

[18] *Drupal*. URL: http://drupal.org/.

[19] *Joomla*. URL: http://www.joomla.org/.

[20] *Usage of content management systems for websites*. Feb. 2011. URL: http://w3techs.com/technologies/overview/content_management/all.

[21] *Moodle*. URL: http://moodle.org/.

[22] *Blackboard*. URL: http://www.blackboard.com/.

[23] *phpMyAdmin*. URL: http://www.phpmyadmin.net/.

[24] *MediaWiki*. URL: http://www.mediawiki.org/.

[25] *Wikipedia*. URL: http://en.wikipedia.org/wiki/Wikipedia.

[26] *Zend Framework*. URL: http://framework.zend.com/.

[27] *Bossie Awards 2010: The best open source application development software; Zend Framework*. Aug. 2010. URL: http://www.infoworld.com/d/open-source/bossie-awards-2010-the-best-open-source-application-development-software-140¤t=4&last=1#slideshowTop.

[28] *SquirrelMail*. URL: http://squirrelmail.org/.

[29] *CMU Webmail*. URL: https://webmail.andrew.cmu.edu/src/webmail.php.

[30] *CWE - Common Weakness Enumeration*. URL: http://cwe.mitre.org/.

[31] *Pareto Principle*. URL: http://en.wikipedia.org/wiki/Pareto_principle.

[32] *2010 CWE/SANS Top 25 Most Dangerous Software Errors*. URL: http://cwe.mitre.org/top25/.

[33] *OWASP Top 10 for 2010*. URL: http://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project.

[34] J. Fonseca and M. Vieira. 'Mapping software faults with web security vulnerabilities'. In: *Dependable Systems and Networks With FTCS and DCC, 2008. DSN 2008. IEEE International Conference on*. IEEE. 2008, pp. 257–266.

[35] D. Janzen and K. De Volder. 'Navigating and querying code without getting lost'. In: *Proceedings of the 2nd international conference on Aspect-oriented software development*. ACM. 2003, pp. 178–187. ISBN: 1581136609.

[36] R. Wuyts et al. 'A logic meta-programming approach to support the co-evolution of object-oriented design and implementation'. In: *PhD, Vrije yiversity of Brussel* (2001).

[37] S. Jarzabek. 'Design of flexible static program analyzers with PQL'. In: *Software Engineering, IEEE Transactions on* 24.3 (1998), pp. 197–215. ISSN: 0098-5589.

[38] *SQL Queries*. URL: http://en.wikipedia.org/wiki/SQL#Queries.

[39] V.B. Livshits and M.S. Lam. 'Finding security vulnerabilities in Java applications with static analysis'. In: *Proceedings of the 14th conference on USENIX Security Symposium-Volume 14*. USENIX Association. 2005, pp. 18–18.

[40] B. Scholz, C. Zhang and C. Cifuentes. 'User-input dependence analysis via graph reachability'. In: *Eighth IEEE International Working Conference on Source Code Analysis and Manipulation*. IEEE. 2008, pp. 25–34.

[41] *Cross Site Scripting Prevention Cheat Sheet*. URL: https://www.owasp.org/index.php/XSS_(Cross_Site_Scripting)_Prevention_Cheat_Sheet.

[42] *SQL Injection Prevention Cheat Sheet*. URL: https://www.owasp.org/index.php/SQL_Injection_Prevention_Cheat_Sheet.

[43] N. Jovanovic, C. Kruegel and E. Kirda. 'Pixy: A static analysis tool for detecting web application vulnerabilities'. In: *Security and Privacy, 2006 IEEE Symposium on*. IEEE. 2006, pp. 6–263.

[44] *Talkback guestbook script*. URL: http://www.scripts.oldguy.us/talkback/.

[45] *Voc webchat server*. URL: http://vochat.com/.

[46] *yaDMS document management system*. URL: http://yadms.sourceforge.net/.

[47] *RIPS vulnerability scanner*. URL: http://sourceforge.net/projects/rips-scanner/.

[48] *phc, the open source PHP compiler*. URL: http://www.phpcompiler.org/.

[49] *Stratego Program Transformation Language*. URL: http://strategoxt.org/.

[50] *The TXL Programming Language*. URL: http://www.txl.ca/.