ALGORITMO EFICIENTE DE ANÁLISE ESTÁTICA PARA PROCURAR ATAQUES DO TIPO VARIÁVEIS CONTAMINADAS

ANDREI RIMSA ÁLVARES

ALGORITMO EFICIENTE DE ANÁLISE ESTÁTICA PARA PROCURAR ATAQUES DO TIPO VARIÁVEIS CONTAMINADAS

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação do Instituto de Ciências Exatas da Universidade Federal de Minas Gerais como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação.

Orientador: Roberto da Silva Bigonha
Co-Orientador: Fernando Magno Quintão Pereira

Belo Horizonte
03 de outubro de 2010

ANDREI RIMSA ÁLVARES

EFFICIENT STATIC ANALYSIS TO FIND TAINTED VARIABLE ATTACKS

Dissertation presented to the Graduate Program in Computer Science of the Federal University of Minas Gerais in partial fulfillment of the requirements for the degree of Master in Computer Science.

Advisor: Roberto da Silva Bigonha
Co-Advisor: Fernando Magno Quintão Pereira

Belo Horizonte
October 3, 2010

Álvares, Andrei Rimsa.

A473a Algoritmo eficiente de análise estática para procurar ataques do tipo variáveis contaminadas / Andrei Rimsa Álvares. — Belo Horizonte, 2010.
xxvi, 72 f. : il. ; 29cm

Dissertação (mestrado) — Universidade Federal de Minas Gerais. Departamento de Ciência da Computação.

Orientador: Roberto da Silva Bigonha.
Co-Orientador: Fernando Magno Quintão Pereira.

1. Computação - Tese. 2. Compiladores (Computadores) - Tese. 3. Algoritmos - Tese.
I. Orientador II. Título.

CDU 519.6*33 (043)

UNIVERSIDADE FEDERAL DE MINAS GERAIS
INSTITUTO DE CIÊNCIAS EXATAS
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

FOLHA DE APROVAÇÃO

Algoritmo eficiente de análise estática para procurar ataques do tipo variáveis contaminadas

ANDREI RIMSA ÁLVARES

Dissertação defendida e aprovada pela banca examinadora constituída pelos Senhores:

PROF. ROBERTO DA SILVA BIGONHA - Orientador
Departamento de Ciência da Computação - UFMG

PROF. FERNANDO MAGNO QUINTÃO PEREIRA - Co-orientador
Departamento de Ciência da Computação - UFMG

PROF. MARCELO D' AMORIM
Departamento de Ciência da Computação - UFPE

PROF. CLAUDIONOR JOSÉ NUNES COELHO JÚNIOR
Departamento de Ciência da Computação - UFMG

PROFA. MARIZA ANDRADE DA SILVA BIGONHA
Departamento de Ciência da Computação - UFMG

Belo Horizonte, 03 de dezembro de 2010.

I dedicate this work to my parents, Frederico and Zilza, and to my beloved girlfriend, Anna Izabel.

Acknowledgments

I would like to thank everyone that participated in this dissertation, either directly or indirectly. I'm very thankful to have Fernando Pereira helping me during this whole research. He believed in me and granted me the opportunity to pursuit the problem tackled in this work that was something that I always wanted to do. I could never forget the opportunity given to me for a three months internship at UFPE with professor Marcelo d'Amorim. His insights along this work have played a crucial role to its accomplishment. I thank Roberto Bigonha for taking me as a student in the early stage of my masters, and allowing me to join as one of his students. Big thanks to all my lab friends, in which we shared moments of despair and happiness during these two years. They are, in no particular order, André Tavares, Rodrigo Sol, Ricardo Terra, César Couto and Giselle Machado.

A special thanks goes to my girlfriend Anna Izabel, who is present in another important step of my life. Her unconditional love and support in every moment of my life has given me the strength to keep pursuing my dreams. Thanks to my parents that always supported me and gave me the education that I will carry with me forever.

I am thankful to all of the `phc` compiler developers, who made an outstanding job. In particular, to Paul Biggar who implemented an optimizer for PHP that we rely so extensively in this work. Paul has helped me a lot during the initial difficulties in understanding the compiler intrinsics and has trusted me as one of the `phc` developers recently.

"Program testing can be a very effective way to show the presence of bugs, but is hopelessly inadequate for showing their absence."
(Edsger W. Dijkstra)

Resumo

Ataques do tipo variáveis contaminadas ocorrem quando entradas de programas são manipuladas maliciosamente a fim de explorar falhas de segurança inerentes ao *software* afetado. Ataques deste tipo são comuns em linguagens de *scripts* como PHP, originadas no lado do servidor. Em 1997, Ørbæk e Palsberg formalizaram o problema de detectar essas explorações como uma instância de verificação de tipo e mostraram que um algoritmo com complexidade de tempo $O(V^3)$ é capaz de resolver o problema. Um algoritmo similar foi implementado dez anos depois pela ferramenta `open source` Pixy para encontrar falhas de segurança em programas PHP. Este trabalho mostra que o mesmo problema pode ser resolvido em tempo $O(V^2)$. A solução proposta utiliza uma representação intermediária chamada e-SSA (extended Static Single Assignment) proposta por Bodik *et al.* em 2000. Essa representação pode ser computada eficientemente e permite que o problema seja tratado como um problema de alcance em grafos, onde arestas modelam dependências de dados entre variáveis de programas. Usando a mesma infra-estrutura de implementação, foi comparada a técnica proposta neste trabalho com e-SSA e a técnica que utiliza análise de fluxo de dados (*data-flow*). Ambas as abordagens possuem a mesma eficácia e foram capazes de detectar 36 vulnerabilidades em programas PHP conhecidos. Nos experimentos é possível observar que a abordagem proposta neste trabalho tem um melhor desempenho que o algoritmo utilizando *data-flow*. As falhas de segurança encontradas foram reportadas e a implementação do algoritmo proposto está disponível para o compilador de PHP `open source phc`.

Palavras-chave: Análise Estática, Detecção Automática de Falhas de Segurança, PHP, compilador `phc`.

Abstract

Tainted variable attacks are originated from program inputs maliciously crafted to exploit software vulnerabilities. These attacks are common in server-side scripting languages, such as PHP. In 1997, Ørbæk and Palsberg formalized the problem of detecting these exploits as an instance of type-checking, and has give an $O(V^3)$ algorithm to solve it. A similar algorithm was, ten years later, implemented on the Pixy free-software tool. In this dissertation we give an $O(V^2)$ solution to the same problem. Our solution uses Bodik *et al.*'s extended Static Single Assignment (e-SSA) program representation that can be efficiently computed and it enables us to treat the problem as an instance of graph reachability, where graph edges model data dependencies between program variables. Using the same infrastructure, we compared the data-flow solution with our technique. Both approaches have detected 36 vulnerabilities in well known PHP programs, and our tests show that our approach tends to outperform the data-flow algorithm for larger PHP programs. We have reported the bugs that we found, and the implementation of our algorithm is now available for the `phc open source` PHP compiler.

Palavras-chave: Static Analysis, Automatic Security Vulnerability Detection, PHP, `phc` compiler.

List of Figures

List of Tables

Contents

Chapter 1

Introduction

Web applications are pervasive over the Internet. They permeate sites as diverse as facebook, google and blogger, are broadly used, and often manipulate sensitive information. It comes to no surprise that web applications are common targets of *cyber attacks*, which typically initiate with a remote attacker carefully forging inputs to corrupt or gain access to a running system. In a study from CVE[1], is possible to observe a shift between the type of vulnerabilities being reported over the years. In the past, most of the attacks reported were caused by operational system vulnerabilities, such as buffer overflows; while nowadays web application vulnerabilities are being more commonly reported. Supposedly, web applications are being developed by less skilled programmers and without the proper security awareness. This is also justified by the same study from CVE, focused on statistics for the year of 2006. *Cross-site scripting* accounts for 18.5% of the vulnerabilities reported in that year. *SQL injection* and *PHP includes* account for other 13.6% and 13.1% respectively of the bug reports. All three vulnerabilities are commonly found in web applications and account for almost 50% of all bugs reported. To put the significance of these threats in perspective, the annual SANS's report[2] estimates that a particular type of attack – malicious SQL injection – has happened approximately 19 million times in July of 2009. Static detection of potential vulnerabilities in web applications is therefore an important problem.

Many web vulnerabilities are described as *Tainted Variable Attacks*. Examples include: SQL-injection, cross-site scripting, malicious file inclusion, unwanted command executions, eval-injection, and file system attacks (see Section 2.3) This pattern consists of a remote individual exploring potential leaks in the system via its public interface. In this context, the interface is the web and the leak is the lack of "sanity" checks on user-provided data before using it on sensitive operations. To detect this kind of attack one needs to answer the following question: does the target program contains a path on which data flows from some input to a sensitive place without going through a sanitizer function? A sanitizer is a function that either filters malicious data or strips them completely. We call the previous question the *Tainted Flow Problem*.

[1] `http://cve.mitre.org/docs/vuln-trends/index.html`

[2] `http://www.sans.org/top-cyber-security-risks/origin.php`

The tainted flow problem was formalized by Ørbæk and Palsberg [1997] as an instance of type-checking. They wrote a type system to the λ-calculus, and proved that if a program type-checks, then it is free of tainted flow vulnerabilities. Ten years later, Jovanovic et al. [2006b] provided an implementation of an algorithm that solves the tainted flow problem for PHP programs. This algorithm, embedded into the Pixy tool, was a data-flow version of Ørbæk *et al.*'s type system. This algorithm has an average $O(V^2)$ runtime complexity, yet, the Pixy's implementation suffers from worst case $O(V^4)$ complexity, and Ørbæk and Palsberg's solution, when seen as a data-flow problem, admits a worst case $O(V^3)$ solution [Ørbæk and Palsberg, 1997].

This work improves on the complexity of these previous results. The algorithm that we propose is, in the worst case, quadratic on the number of variables in the source program, both in terms of time and space. The low asymptotic complexity is justified by the use of a program representation called *extended Static Single Assignment* (e-SSA) form, introduced by Bodik et al. [2000], which can be computed in linear time in the program size. This intermediate representation makes it possible to solve the tainted flow problem as a *sparse* analysis, which associates constraints directly to program variables, instead of associating them to variables at every program point as traditional data-flow based approaches. Therefore, it allow us to model the tainted flow problem as a graph with constraints binded to variables – a graph reachability based approach.

We chose to evaluate our solution on PHP applications, although our analysis can be generalized to any other procedural languages. This choice was driven by two reasons. First, it is a popular language for developing server-side web applications, according to the *TIOBE Index*[3]. Also, PHP programs can be found in over 21 million Internet domains worldwide[4]. Second, PHP has been the focus of previous research on static detection of tainted flow vulnerabilities. Since PHP is widely spread, there are enormous resources available; hence benchmarks are easily found.

1.1 Objectives

We aimed at two main objectives in this work. We implemented our tainted flow analysis to be:

1. **Precise:** Our analysis has the same precision of other data-flow based approaches. However, our analysis is accurate because it was able to find 36 previously unknown vulnerabilities in five well-know PHP packages. We reported our findings to vendors and to the bugtraq[5].

2. **Efficient:** Our analysis is efficient because we rely on the e-SSA program representation to model the problem as graph reachability. We improved the complexity of previous solutions, from $O(V^3)$ to $O(V^2)$ in both terms of space and time; whereas V is the number of variables on the program. We show that our algorithm is faster than data-flow in most cases for larger PHP files.

[3] `http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html`
[4] `http://php.net/usage.php`
[5] `http://www.securityfocus.com`

1.2 Contributions

This work brings forward the following contributions:

- An efficient algorithm to solve the tainted flow problem. Two distinguishing features of the algorithm are: (i) the use of the e-SSA representation to generate constraints; and (ii) an on-demand constraint solving algorithm. See Section 4.4.
- An implementation of the algorithm on top of `phc` [Biggar et al., 2009a], an `open source` PHP compiler[6]. During our research, we have found and fixed many bugs and extended the compiler with new features by submitting patches to the `phc` project.
- An evaluation of the proposed approach on public PHP applications, including benchmarks used in previous works [Jovanovic et al., 2006a; Xie and Aiken, 2006]. See Chapter 5.
- The discovery of unknown bugs in five well-known PHP packages by our analysis. We reported our finds to both vendors and bugtraq. See Chapter 5.
- A comparative analysis between our approach and the standard data-flow used in previous works [Jovanovic et al., 2006a]. See Chapter 5.

This dissertation resulted in two papers: (i) a paper in the SBLP 2010 conference that covers our graph reachability approach to the tainted flow problem; and (ii) an accepted paper to appear in the CC 2011 that compares our approach against the standard data-flow analysis. The SBLP paper was given the third best paper award in the conference and received an invitation to be submitted to the Science of Computer Programming journal.

- Rimsa, A., d'Amorim, M., and Pereira, F. M. Q. Efficient Static Checker for Tainted Variable Attacks. In SBLP 2010.
- Rimsa, A., d'Amorim, M., and Pereira, F. M. Q. Efficient Tainted Flow Analysis. In CC 2011.

1.3 Dissertation Organization

This dissertation is organized into seven chapters. Below we list the remaining chapters with a brief summary.

- **Chapter 2:** We provide background information on key aspects used along this dissertation. We cover the SSA and e-SSA intermediate representations in Section 2.1. We give an overview of the PHP languages in Section 2.2 discussing its features, why is it difficult to statically analyze PHP programs and a survey of PHP compilers that we considered as baseline for our analysis. We introduce the tainted variable attack with examples of real-world attacks in Section 2.3.

[6] `http://www.phpcompiler.org/`

- **Chapter 3:** This chapter covers how to model the tainted flow using data-flow. We designed a small language called Nano-PHP in Section 3.1 to help us model the problem. We provide the language semantics in Section 3.2 and the standard data-flow algorithm for the tainted flow problem in Section 3.3. We give an example to illustrate the algorithm and give its complexity in terms of time and space.

- **Chapter 4:** This chapter discusses how to model the tainted flow problem as graph reachability using the e-SSA representation. In Section 3.1 we augment the Nano-PHP language to cover the two new instructions inserted by this representation: ϕ- and σ-functions. We show how to build the constraint graph in Section 4.2. We handle alias in validators in Section 4.3. In Section 4.4 we show an equivalent algorithm to solve the tainted flow problem without actually building the constraint graph and give its complexity in terms of space and time.

- **Chapter 5:** This chapter presents the evaluation of our proposed solution. We list our experiment configuration set-up in Section 5.1. In Section 5.2 we describe the 32 benchmarks considered in our evaluation. Later, we discuss our solution in terms of efficiency (Chapter 5.3) and precision (Chapter 5.4). We show that our approach is faster for larger PHP files, and that it can actually find real security vulnerabilities. We give a real-world example of a bug that we found in the MODx CMS version 1.0.3 in Section 5.5.

- **Chapter 6:** In this chapter we review other works related to ours. We give an historic overview of web application security in Section 6.1, and then we specialize in works treating the tainted flow problem in Section 6.1.1. We also review works that modeled their problem using a graph reachability approach in Section 6.2.

- **Chapter 7:** We present the conclusions of our work in this chapter. We also discuss the limitations and possible directions for future works.

Chapter 2

Background

In this chapter, we provide background information that covers key concepts used in this dissertation. In Section 2.1 we describe the intermediate representation, namely e-SSA (extended Static Single Assignment) proposed by Bodik et al. [2000]. This representation is the core of our analysis that enable us to perform a sparse analysis and decrease in one order-magnitude the complexity of standard data-flow analysis on tainted flow problems, from $O(V^3)$ to $O(V^2)$ – details in Chapter 4.

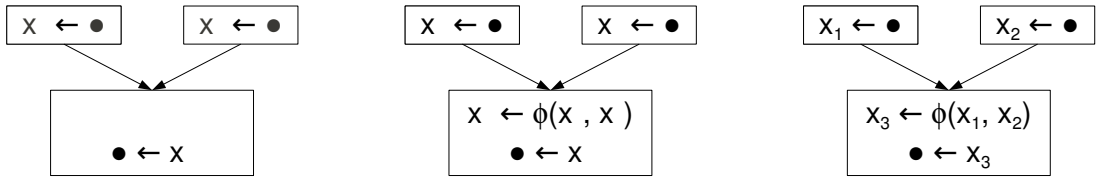
In Section 2.2 we detail PHP features that fascinate users, but complicate static analysis. Later, we provide a discussion of PHP compilers and the motivations behind the choice of `phc` as our baseline compiler. In Section 2.3 we describe tainted variable attacks in terms of sources, sinks and sanitizers with examples of security vulnerabilities that fit this description. We conclude this chapter in Section 2.4.

2.1 Intermediate Representation (IR)

We rely on an intermediate program representation called extended Static Single Assignment (e-SSA) proposed by Bodik et al. [2000] to build our analysis on. This representation is a superset of the Static Single Assignment (SSA) proposed by Cytron et al. [1989] that is addressed in Section 2.1.1. The e-SSA is the core of our analysis that enabled us to treat the tainted flow problem as a graph reachability problem. We give further details about the e-SSA representation in Section 2.1.2 and how we model it as a graph reachability problem in Chapter 4.

2.1.1 Static Single Assignment (SSA)

The Static Single Assignment is a program representation proposed by Cytron et al. [1989] that stipulates that each variable on the program has a single definition; hence the `Single Assignment` in the name. The `Static` means that the definition can be computed statically. This representation simplifies many compiler analyses, including register allocation. Hack [2005] proved that programs in SSA form have chordal interference graphs, which can be efficiently colored in polynomial time [Pereira and Palsberg, 2005].



(a) Original program. (b) Insert *phi*-functions. (c) Rename variable.

Figure 2.1: SSA algorithm on the original program (left) based on two steps. First, insert *phi*-functions (center). Second, rename variables (right).

Because of SSA's single reaching definition property, a special function is required on control-flow join points. The *phi*-function is used to merge the live range of variables. Each *phi*-function receives as argument variables associated to each control-flow predecessor. In order to convert the program into SSA form, we must follow two steps as exemplified by Figure 2.1. Given the original program, calculate the dominance frontier [Appel and Palsberg, 2002; Cooper et al., 2001] to identify precisely the points where *phi*-functions must be inserted. A block x is said to dominate a block y iff there is no path from the entry point to y without passing through x. A block z is in the dominance frontier of x iff x dominates the predecessor of z, but not z itself. For instance, a definition of a variable v in block x has the effect of inserting a *phi*-function in block z, since there is another path to z that does not pass through x – because z is not dominated by x. These new *phi*-functions produce fresh definitions of v; thus, the process continues until the program stabilizes. After all *phi*-functions are properly inserted, a stack based algorithm renames the variables by assigning different version numbers. The renaming is based on the nearest reaching definition. There exist almost linear time algorithms to convert programs to SSA form [Lengauer and Tarjan, 1979].

2.1.2 Extended Static Single Assignment (e-SSA)

In our work we use the e-SSA representation [Bodik et al., 2000] as baseline for the algorithm we proposed in Chapter 4. This representation has similarities with the Static Single Information (SSI) form, which was first proposed by Ananian [1999]. Later, Singer [2006] provided a high-level definition for SSI believing that his definition matched Ananian's. Boissinot et al. [2009] proved that these two are not the same, and reference Ananian's definition as **strong** SSI, and Singer's definition as **weak** SSI. Interesting to say that the algorithm proposed by Singer [2006] in fact constructs the strong SSI. We will use strong SSI to compare with e-SSA.

Both e-SSA and SSI include a special instruction that has the opposite effect of SSA's *phi*-function. While *phi*-functions merge the live range of variables in blocks with more than one predecessor, *sigma*-functions split them in blocks with more than one successor, i.e., branch blocks. The difference between these two algorithms resorts on how they find which variables to convert in *sigma*-functions. SSI algorithms rely on the reverse dominance frontier to identify precisely the points where *sigma*-functions must be inserted, while e-SSA only converts variables used in the conditional clause of

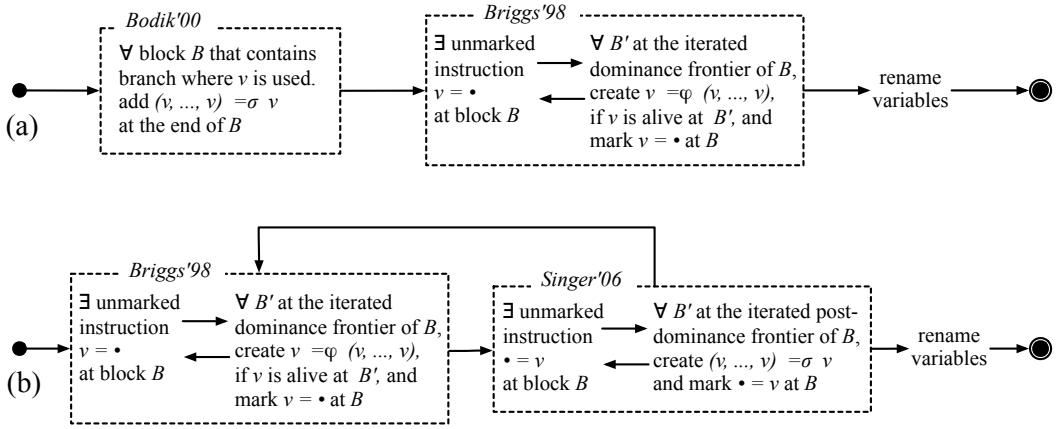


Figure 2.2: e-SSA construction (a) and SSI construction (b).

branch blocks. Therefore, it is a straight-forward conversion that does not require the calculation of the expensive reverse dominance frontier to find which variable must be converted. Figure 2.2 summarizes the differences between these two algorithms. The algorithm to construct e-SSA (algorithm a) does a single pass to insert *sigma*-functions and then triggers the algorithm to construct SSA [Brisk, 2006] in a second stage, which includes a pass to insert *phi*-function and to rename the variables. However, SSI construction (algorithm b) iterates between the insertion of *sigma*- and *phi*-functions until the program stabilizes – no new functions are inserted. Only then we can rename the variables following SSA renaming algorithm. Therefore, we can build e-SSA faster than SSI and possibly with less *sigma*-functions. It is possible to reduce the cost to transform the program into SSI by choosing which variables to convert on demand [Tavares et al., 2010]. In essence, their partial SSI works similarly to e-SSA, which can be computed efficiently. Later, full SSI can be requested which uses partial SSI as base; thus avoiding the need to convert variables already in SSI form.

The e-SSA representation is useful for many static analysis: eliminate redundant array bound checks [Bodik et al., 2000], bitwidth analysis [Stephenson et al., 2000] and conditional constant propagation [Wegman and Zadeck, 1991]. In Section 4.1 we clarify its advantages and show how we applied it in our problem domain.

2.2 PHP

In this Section we review the PHP[1] programming language. We chose to find tainted attacks in PHP because it is a popular language and it is widely used in web application. We show this languague important features in Section 2.2.1 and why it is difficult to statically analyze in Section 2.2.2. As an early design decision, we chose to rely on a mature compiler infra-structure to perform our analysis. In Section 2.2.3 we survey many PHP compilers and discuss why we ultimately chose `phc` as our baseline compiler.

[1] http://www.php.net

2.2.1 Language Features

PHP is a general purpose language used mainly to create dynamic web pages. PHP stands for the recursive acronym `PHP: Hypertext Preprocessor`. It was designed by *Rasmus Lerdorf* in 1994 as a set of Perl[2] scripts to organize his personal web page. Because of that, PHP has substantial influence in Perl's syntax, semantics and interpreted nature. In its early days, PHP was implemented as a Common Gateway Interface[3] (CGI) with the ability to work with web forms and embedded database support. This first version was initially called *Personal Home Page/Forms Interpreter*, or PHP/FI for short. This extension allowed programmers to build simple and dynamic web applications. An example of a simple PHP script is shown below:

```
<?php
  echo "Hello world!";
?>
```

PHP is been under constantly development ever since. There was a major reformulation in 1997 that formed the base for PHP 3.0. In 2000, PHP 4.0 was released powered by the Zend Engine 1.0 and in 2004 the PHP 5.0 was launched in Zend Engine II with a complete new object-orientation model. PHP 6.0 is still under development and will provide full unicode support.

PHP is a popular language[4] and it is present in more than 20 millions of web server world wide[5]. PHP most important features include:

- Open source
- Server-side programming language
- Embedded HTML support
- Embedded Database support
- Portability

One of PHP advantages is that it is an `open source` product. This means that it is been actively developed by a community, referred to as the PHP group, and it is free of charge. Anyone can download, install and start writing PHP application with no costs involved. PHP has an extensive documentation of the language features with many examples and user comments[6]. PHP is also a server-side language. Every processing is done by the server and the result is the html code generated by the page that will be rendered by the browser later. This allows hiding the business logic from end users. PHP contains embedded html and database support. In a PHP script, the presentation (html) logic can be separated from the code logic. PHP code must

[2] `http://www.perl.org`
[3] `CGI: RFC 3875 - http://tools.ietf.org/html/rfc3875`
[4] `http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html`
[5] `http://www.php.net/usage.php`
[6] `http://www.php.net/manual`

be enclosed between the start (`<?php`) and end tags (`?>`). Everything else outside these tags is considered html code to be dumped without processing. PHP program can easily be integrated with databases. In fact, PHP has embed support for many different database vendors, such as MySQL[7] and PostgreSQL[8]. This is another PHP feature enjoyed by PHP developers, because they do not need to install third party software to have seamless database connectivity. PHP is also portable across different operational systems. It runs on UNIX flavors and Windows machines.

2.2.2 Static Analysis on PHP

The first barrier to analyze PHP programs statically relies on the language interpreted nature. Therefore, many features that only exist in interpreted languages are also present in PHP, such as dynamic file inclusion and run-time code evaluation. To complicate matters further, the PHP language has no formal definition. If someone wants to stay compatible with PHP, they must mirror the Zend reference implementation provided by the latest release of PHP [PHP, 2010]. Below we list some of these features that make static analysis complicated for scripting languages such as PHP. This list was extracted from Biggar and Gregg [2009].

- Run-time source inclusion
- Run-time code evaluation
- Dynamic, weak, latent typing
- Duck-typed objects
- Implicit object and array creation
- Run-time aliasing
- Run-time symbol-table access
- Overloading of simple operations

The major difficulty to static analyze PHP programs relies on the ability to infer values that can only be know at run-time. For example, PHP allows files to be dynamic included; where the intended file name can sometimes only be know during execution. In the other hand, languages such as `Java` and `C` can resolve these files statically, and include them at compile-time. PHP code evaluation dispatcher suffers from the same problem. PHP commands represented as string are executed by the PHP interpreter, but sometimes we are unable to determine its content priorly. These two features bring uncertainty to the analysis, that can no longer rely on the complete view of the program. Therefore, we must do a very conservative analysis or, in worst case, completely abort the analysis [Biggar et al., 2009a].

[7] `http://www.mysql.org`
[8] `http://www.postgresql.org`

Scripting languages such as PHP are famous for being dynamic typed. Programmers do not need to annotate type information to variables; thus making it harder to static check. The type of the variable depends on its run-time value. To be able to perform a type analysis, we must follow every value that a variable may hold in order to estimate which types the variable can have. However, sometimes the type cannot be inferred directly; for instance, objects and arrays can be implicit created. An assignment `$x->y = 0` – where x is undefined – has the effect of creating a standard PHP object with y as attribute, while `$x[0] = 0;` creates an array with a single index. PHP is also duck-typed, because it allows methods and attributes to be inserted in previous defined classes at run-time; therefore changing the classes initial signature.

Another problem to cope with is alias analysis. Jovanovic et al. [2006b] acknowledge that alias analysis for imperative languages is not suitable for PHP. PHP references are mutable: can be created, used and destroyed during the program execution. The alias is also bidirectional. In fact, a variable in the main scope, such as `$x` is an alias of the global array `$GLOBALS['x']` and vice versa. PHP's global symbol table is even exported to the run-time environment, which can be used, modified or even unset, i.e. remove all its symbols. We can have alias between variables, formal parameters, global variables, array elements. In order to model alias accurately, we need to keep track of which variables may- or must-alias each other. A may-alias is a conditional alias between two variables, while must-alias is unconditional.

All these complicated features from PHP make static analysis in this language difficult. To avoid dealing with all these features, we decided to rely on a compiler that already tackles these issues. In the next section we review many PHP compilers and discuss their advantages and limitations. We decided towards the `phc` compiler, which handles many of these problems for us [Biggar et al., 2009a] .

2.2.3 PHP Compilers

Previous approaches to handle the tainted flow problem in PHP programs followed a similar strategy. The program is parsed into an abstract syntax tree (AST) using the official grammar obtained from the PHP group[9]. The syntactic elements are extracted from the original source without syntactic sugar like commas and semi-colons; thus making it easier to handle. However, this approach only does the syntactic parsing of the program, leaving developers with the herculean task of writing their analysis compatible with the language semantics from the ground-zero. The Pixy tool decided towards this approach by creating their own alias analysis to increase the precision of its tainted analysis for PHP programs [Jovanovic et al., 2006b] directly on the AST. In our work, we would like to avoid the trouble of remodeling the whole language by using a compiler infra-structure that already deals with the complex features of the language. Note that we do not care about the compiler code generator, but the static analysis that it may provide.

Working with the AST level has another disadvantage. Because the AST closely resembles the original program, it uses the PHP commands exactly as they were on the source, demanding the analysis to cope with many language constructs. We could

[9] `http://www.php.net`

Figure 2.3: `phc` compiler internal pipeline, in which each stage represent an intermediate representation that the pass could work upon.

take advantage of a lower-level representation that deconstructs these instructions into simpler ones. For instance, PHP `foreach` command could be transformed into a `while` command. At the same time, a `while` command could be transformed into labeled conditionals with gotos pointers. Hence, we could build a control-flow graph (CFG) using this lower-level representation to be used by our analysis.

The `phc` [Biggar et al., 2009a] compiler does this job for us precisely using a three way transformation in the program, according to the pipeline in Figure 2.3. Initially, the compiler was designed as a front-end for PHP, which they designed their own parser for the AST [de Vries and Gilbert, 2007]. The compiler was later expanded to generate C code using the embedded PHP library which turns `phc` resilient to changes in the PHP reference implementation [Biggar et al., 2009b]. This addition introduced the other two internal representations shown in Figure 2.3. The AST is transformed to three-address code in the high-level IR (HIR). Later, high-level instructions are deconstructed to form the medium-level IR (MIR). From the MIR, the compiler derivates a control-flow graph that it is used across its optimizations passes. We could enjoy this representation to build our analysis on.

Other works on the tainted flow problem transforms the program into a low-level representation as well. Xie and Aiken [2006] and Wassermann and Su [2007] represented their functions as control-flow graphs to leverage their analysis. In the latter, the CFG is even transformed into SSA form. However, neither works considered the effect of aliases in their evaluation, possibly leading to an imprecise intermediate representation. These limitations is addressed by the `phc` compiler that provides a powerful alias analysis [Pioli et al., 1999] combined with the ability to transform the CFG into SSA form [Chow et al., 1996]. Since e-SSA is a super-set of SSA, `phc` sounded like a natural choice for our work.

We have surveyed other tools that parse PHP programs. The only `open source` tool available that handles the tainted flow problem that we are aware of is Pixy [Jovanovic et al., 2006a]. Unfortunately, it is based on PHP 4.0 and has no support for classes, a widely used feature by most PHP programs nowadays. The Roadsend compiler can generate binary code for x86 machines, but it is written in `O'Caml` making it unsuitable for us [Roadsend, 2010]. As far as we know, this compiler does not provide any analysis or optimization that we could benefit from. A subproject of the Roadsend compiler is the Roadsend Raven [Raven, 2010]. The compiler was completely rewritten in C++ and is now capable of generating intermediate code for the industrial-strength LLVM compiler[10]. The obvious advantage is that LLVM can generate code for a wide number of target architectures. Unfortunately, their project is still incipient and does

[10] `http://www.llvm.org`

not aim for 100% compatibility with the Zend Engine [PHP, 2010]. They did a trade-off between difficult language features to support, such as aliasing, to a more conservative and simple analysis. A recently launched compiler HipHop[11], that transforms PHP to C++ code, followed a similar strategy by sacrificing hardly used features, such as code evaluation. They reported gains on CPU usage of 50% on average depending on the web page[12]. Other tools compile PHP programs to be used by virtual machines. Phalanger [Benda et al., 2006] ported PHP to run on .NET platforms[13], while Quercus [Quercus, 2010] did it for the JVM[14]. In Phalanger, PHP classes and interfaces are represented directly into built-in models of classes and interfaces provided by the .NET framework. PHP can also take advantage of features available only for ASP scripts, an approach shared by Quercus that can benefit from Java's database pooling, for instance. Unfortunately, both tools decided to rewrite the entire PHP standard library to stay compatible with the Zend implementation, a herculean and error-prone task. Because of these limitations, we chose to write our analysis on the `phc` compiler.

2.3 Tainted Variable Attacks

A tainted variable attack is characterized by a sub-path from a source to a sink function that does not include any calls to sanitizing function. A source is a program input which holds user data, e.g. HTML form. The attacker's goal is to carefully craft these input in order to circumvent the system to its own control – possibly leading to arbitrary code execution – or to leverage other types of attack. Sinks are functions that perform sensitive operations that may have security consequences to the running application, e.g. database access. Sanitizers are functions that validate data, either by proving that it does not contain harmful data, or by replacing dangerous characters with filtered values whenever possible. By setting these three parameters properly (sources, sinks and sanitizers) we could configure our tool to statically find security vulnerabilities. This problem is best known as the Tainted Flow Problem.

We can describe the tainted flow problem as a tuple $T = (P, SO, SI, SA)$, such that: P is the subject program; SO is a set of input functions, referred to as sources; SI is a set of security-sensitive operations, referred to as sinks; and SA is a set of functions that validate inputs, referred to as sanitizers. The tainted flow problem consists of determining if program P can make a call to a sink function $si \in SI$ passing a value v, which has been generated by a source function $so \in SO$, and that has not been sanitized by any function $sa \in SA$. In Section 3.2 we give a formal definition (Definition 1) of this problem by using a small language that we called Nano-PHP.

Usually, the same source inputs (SO) are shared between several instances of tainted flow attacks. The attacks usually originate from user-interaction between the browser request and the requested web server, so they can be configured according to these sources. In case of PHP, it can be configured through inputs such as get (`$_GET`) and post (`$_POST`) http requests. The sanitizers (SA) can be further categorized into

[11] `https://github.com/facebook/hiphop-php`

[12] `http://developers.facebook.com/blog/post/358`

[13] `http://www.microsoft.com/net`

[14] `http://www.oracle.com/technetwork/java/javase/overview/index.html`

two types: (i) filters; and (ii) validators. Filters are straight-forward functions that strip malicious contents from a string or transform them into safe inputs to use. These filters are normally configured for each different instance of tainted variable attacks. In the other hand, validator functions ensure that an input is not harmful on conditional branches; thus proving that a variable is free of tainted values for a specific branch. These validators can also be shared across multiple instances of security vulnerabilities, since these validators are commons PHP security routines, such as *is_numeric*. The last configurable option, sinks (*SI*), is set dependending on the type of security vulnerability intended to find. Different security vulnerabilities have different sensitive operations; therefore sinks must be set according to each bug.

Many vulnerabilities described in the literature can fit this description of tainted variable attacks. Some noticeable examples are cross-site scripting (XSS) [Chugh et al., 2009], SQL injection attacks [Wassermann and Su, 2007; Xie and Aiken, 2006], malicious evaluations[15], local/remote file inclusions[16], and unwanted command execution[17]. In the next sections we show examples of these vulnerabilities illustrating how can we configured their respective sinks, sources and sanitizers.

2.3.1 Cross-Site Scripting (XSS)

Cross-site scripting typically occurs when a user is able to dump html code within a dynamically generated page. An attacker uses this vulnerability to inject JavaScript code into the page, usually trying to steal cookie information to acquire session privileges. As an example, the program below contains a vulnerability that allows an external user to output any given text, including html commands.

```
<?php $name = $_GET['name']; echo $name; ?>
```

Consider the scenario where the user assigns the following data to variable `name`: "`<script>does.something.evil;</script>`". A potentially malicious JavaScript program might be executed in the client side of the application. A workaround for this bug is to strip malicious html content from the user input. The built-in function `htmlentities` does the trick by encoding special characters into their respective html entities. For example, "`<`" gets translated to "`&lt;`".

```
<?php $name = htmlentities($_GET['name']); echo $name; ?>
```

Cross-site scripting attacks fit the tainted flow problem framework. A possible input configuration, in this case, would be:

Sources : `$_GET`, `$_POST`, ...

Sinks : `echo, print, printf`.

Sanitizers : `htmlentities, htmlspecialchars, strip_tags`

[15] `http://cwe.mitre.org/data/definitions/95.html`
[16] `http://projects.webappsec.org/Remote-File-Inclusion`
[17] `http://secunia.com/advisories/26201/`

2.3.2 SQL Injection Attacks

The SQL injection attack is another common type of security flaw. The attacker injects malicious database commands via SQL query parameters. The effect can go from reporting incorrect results to the user to modifying the database. The program below contains an example of SQL injection vulnerability:

```
<?php
  $userid = $_GET['userid'];
  $passwd = $_GET['passwd'];
  ...
  $result = mysql_query("SELECT userid FROM users WHERE " .
                        "userid=$userid AND passwd='$passwd'");
?>
```

Note that this program does not sanitize its inputs. A malicious user could obtain access to the application by providing the text `1 OR 1 = 1 --` in the `userid` field. The double minuses start MySQL comments; hence, the resulting query will be `SELECT userid FROM users WHERE userid=1 OR 1 = 1 -- AND passwd='passwd'`, which always outputs one row and therefore bypass the authentication procedure.

One can sanitize variable `userid` by ensuring that it contains only numerical values; a task that we perform either casting it to integer or checking its value with functions like `is_numeric`. One can sanitize variable `$passwd` using the `addslashes` function, which inserts slashes (escape characters) before a predefined set of characters including single quotes. A typical configuration of SQL injection is given below:

Sources : `$_GET`, `$_POST`, ...

Sinks : `mysql_query, pg_query, *_query`

Sanitizers : `addslashes, mysql_real_escape_string, *_escape_string`

2.3.3 Unwanted Command Execution

A PHP script may rely on an external program to compute data. PHP has a handful of functions to trigger command execution. The example below illustrates the use of the *system* function.

```
<?php
 $file = $_GET['file'];
 system("/usr/bin/file $file");
?>
```

An attacker could insert a semi-colon (;) to trick the system to execute another program of its choice. In order to sanitize the input, we could escape the command arguments via function *escapeshellarg* or in some cases escape the whole command through *escapeshellcmd*. A possible description of this type of a attack is:

Sources : `$_GET`, `$_POST`, ...

Sinks : `exec, system, passthru, shell_exec, proc_open, pcntl_exec`

Sanitizers : `escapeshellarg, escapeshellcmd`

2.3.4 Unauthorized Filesystem Access

Allowing PHP scripts to access the file system is a desirable feature. PHP permits many functions to handle the file system, from reading a file to change its properties, like usernames and permissions. The following example shows the *unlink* function, which is used to delete files. Since no sanitizers are present, an attacker could delete any file that he has permission to.

```
<?php
  $file = $_GET['file'];
  unlink($file);
?>
```

There is no specific PHP built-in operator to sanitize inputs for this type of attack. We could write our own regular expression for that purpose, or rely on validators to do the trick. A possible configuration for unauthorized file system access could be:

Sources : `$_GET`, `$_POST`, ...

Sinks : `chdir, mkdir, rmdir, rename, unlink, copy, chgrp, chown, chmod, touch, symlink, link, move_uploaded_file, show_source,highlight_file readfile, file_get_contents`

Sanitizers : `is_numeric, type casts`

2.3.5 Other Attacks

Other vulnerabilities can fit this same tainted analysis framework. Most noticeable examples are local/remote file includes and malicious evaluations. However, these two type of security flaws have inner difficulties to our analysis that is addressed in Section 7.1. Nevertheless, if we could solve these issues, we could enjoy the benefits of our tainted flow analysis (Chapter 4) for finding these bugs.

2.4 Conclusion

In this chapter we covered important concepts used along this dissertation. We discussed the intermediate representation that is the core of the solution proposed by this work. We provided background information concerning PHP, why it a language suitable for the web and why is difficult to static analyze. We survey several PHP compilers, and decided towards `phc` as our baseline compiler. Finally, we reviewed the tainted flow problem giving example of security vulnerabilities.

Chapter 3

Tainted Analysis as Data-flow

In this chapter we describe how to model the tainted flow problem using data-flow. Data-flow is not a new concept, and it has been used by compiler designers for decades [Kam and Ullman, 1976]. Even in this same problem domain, in order to find tainted variable attacks [Jovanovic et al., 2006b]. We implemented a data-flow analysis for the tainted flow problem described in this chapter in order to compare with our solution using e-SSA described in Chapter 4.

Data-flow analysis usually relies on an algebraic structure called a lattice. A lattice is simply a set, plus a partial ordering between the elements of this set. A data-flow analysis contains a number of equations, and each of these equations has the effect of reading a point in the lattice and returning another – possibly different – point. We know that the data-flow algorithm terminates when: (i) each equation is a monotone function; and (ii) the lattice is finite. A monotone function $f(x)$ always returns an element y, such that $y \geq x$, given a total ordering between the elements in f's domain. To know more about the theoretical foundations of data-flow analysis, refer to [Aho et al., 2006, Chapter.9].

We design a small language called Nano-PHP used to model the data-flow analysis. This language cover all the aspects of the original PHP language pertinent to our analysis, but with less instructions. Therefore, we provide a concise semantics of the language by means of an abstract state machine in Section 3.2. In Section 3.3, we list the data-flow equations that are used to solve the tainted flow problem with the associated complexity. We discuss in details a worklist algorithm with an example. We conclude this chapter in Section 3.4.

3.1 Nano-PHP

We use the assembly-like Nano-PHP language to define the tainted flow problem. A label $l \in L$ refers to a program location and is associated to one instruction. A Nano-PHP program is a sequence of labels, $l_0, l_1 \ldots, l_{exit}$. Table 3.1 shows the six instructions of the language. We use the symbol $\otimes$ to denote any operation that (i) uses a number of variables, and (ii) defines a variable. We use two different symbols to address assignments from source ($\circ$) and assignments to sink ($\bullet$).

Name	Instruction	Example
Assignment from source	$x = \circ$	`$x = $_POST['content']`
Assignment to sink	$\bullet = v$	`echo($v)`
Simple assignment	$x = \otimes(x_1, \ldots, x_n)$	`$a = $t1 * $t2`
Branch	`bra` $l_1, \ldots, l_n$	general control flow
Filter	$x =$ `filter`	`$a = htmlentities($t1)`
Validator	`validate` x, l_c, l_t	`if (!is_numeric($t1)) abort();`

Table 3.1: The Nano-PHP syntax.

In Figure 3.1 we give an example of a simple PHP program (left) that uses a fictional database structure to perform operations. We transform this program into Nano-PHP (middle) and give its control-flow graph representation (right). We will use this example along this chapter to demonstrate the data-flow algorithm.

```
<?php
  $v = DB.get($_GET['child']);
  $x = "";
  if (DB.isMember($v)) {
    while (DB.hasParent($v)) {
      echo $x;
      $x = $_POST['$v'];
      $v = DB.getParent($v);
   }
   echo($v);
}
?>
```

```
l0: v = ○
l1: x = filter
l2: validate(v, l3, l9)
l3: bra l4,l8
l4: ● = x
l5: x = ○
l6: v = ⊗(v)
l7: bra l3
l8: ● = v
l9: bra l9
```

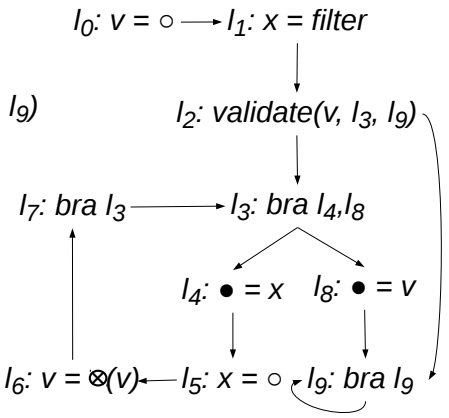


Figure 3.1: A simple PHP program (left), its equivalent Nano-PHP version (middle) and its control-flow representation (right). We let `DB` to denote a global database, and we assume that `DB.get` might produce tainted data. We use label l_9 to mark the end of the program.

3.2 Semantics

We define the semantics of Nano-PHP programs by means of an abstract machine. The state M of this machine is characterized by a tuple (Σ, F, I), informally defined as follows:

Store $\Sigma \subseteq \mathit{Var} \to \mathit{Abs}$	e.g., $\{x_1 \mapsto \mathit{clean}, \ \ldots, \ x_n \mapsto \mathit{tainted}\}$
Code Heap $F \subseteq L \to [\mathit{Ins}]$	e.g., $\{l_1 \mapsto i_1 \ldots i_a, \ \ldots, \ l_n \mapsto i_b\}$
Instruction Sequence $I \subseteq [\mathit{Ins}]$	e.g., $i_5 i_6 \ldots i_n$

$\sqcap$	$\bot$	**clean**	**tainted**
$\bot$	$\bot$	clean	tainted
clean	clean	clean	tainted
tainted	tainted	tainted	tainted

Table 3.2: Definition of least upper bound over pairs of abstract values.

The set *Abs* is a lattice formed with the following elements and order $\bot <$ clean $<$ tainted. The $\bot$ element is associated to yet undefined variables, while clean and tainted states denote safe and dangerous variables. The order $<$ describes the partial ordered of the elements lattice. Table 3.2 shows the result of the pairwise join operation ($\sqcap$) over lattice elements. For instance, the element *clean* when join with *tainted* result in *tainted*.

The store Σ binds each variable name, say $x \in Var$, to an abstract value $v \in Abs$. The code heap F is a map from a program label to a sequence of instructions. Each sequence corresponds to one *basic block* of the Nano-PHP program. Only labels associated to entry *basic block* instructions appear in F. The list I denotes the next instructions for execution. We say that the abstract machine can *take a step* if from a state M it can make a transition to state M'. More formally, we write $M \rightarrow M'$. We say the machine is *stuck* at M if it cannot make any transition from M.

Figure 3.2 illustrates the transition rules describing the semantics of Nano-PHP programs. Rule S-SOURCE states that an assignment from source binds the left-hand side variable to the tainted abstract state. Rule S-SINK is the only one that can cause the machine to get stuck: to execute an assignment to sink the variable on the right hand side *must* be bound to clean. Rule S-SIMPLE says that, given an assignment $x = \otimes(x_1, x_2, \ldots, x_n)$, the abstract state of x is defined by folding the join operation (as described on Table 3.2) onto the list of variables in the right hand side, e.g: $x_1 \sqcap x_2 \ldots \sqcap x_n$. Rule S-BRANCH defines a non-deterministic branch choice: the machine chooses one target in a range of possible targets and branches execution to the leading instruction on its defining basic block.

Nano-PHP also divides the sanitizers into filter and validator functions. Filters correspond to functions that take a value, typically of string type, and return another value after removing malicious fragments from the input. For simplicity we do not show the input parameter in the syntax of Nano-PHP. Rule S-FILTER shows that an assignment from a filter binds the variable on the left side to the clean state. Validators are instructions that combine branching with a boolean function that checks the state for tainting. The instruction `validate`(x, l_c, l_t) has two possible outcomes. If x is bound to the clean state, the machine branches execution to $F(l_c)$. If x is bound to the tainted state, execution branches to $F(l_t)$. Rules S-VALIDC and S-VALIDT define these cases. We assume that every Nano-PHP program is *strict* [Budimlic et al., 2002], i.e., all variables must be defined before used; therefore, we rule out the possibility of passing x to a validator when $\Sigma \vdash x = \bot$.

$$\text{[S-SOURCE]} \qquad (\Sigma, F, x = \circ; S) \rightarrow (\Sigma \backslash [x \mapsto \text{tainted}], F, S)$$

$$\text{[S-SINK]} \qquad \frac{\Sigma \vdash t = \text{clean}}{(\Sigma, F, \bullet = v; S) \rightarrow (\Sigma, F, S)}$$

$$\text{[S-SIMPLE]} \qquad \frac{\Sigma \vdash \sqcap(x_1, \ldots, x_n) = v}{(\Sigma, F, x = \otimes(x_1, \ldots, x_n); S) \rightarrow (\Sigma \backslash [x \mapsto v], F, S)}$$

$$\text{[S-BRANCH]} \qquad \frac{\{l_i\} \subseteq \text{dom}(F) \qquad F(l_i) = S' \qquad 1 \leq i \leq n}{(\Sigma, F, \texttt{bra}\ l_1, \ldots l_n; S) \rightarrow (\Sigma, F, S')}$$

$$\text{[S-FILTER]} \qquad (\Sigma, F, x = \texttt{filter}; S) \rightarrow (\Sigma \backslash [x \mapsto \text{clean}], F, S)$$

$$\text{[S-VALIDC]} \qquad \frac{\Sigma \vdash x = \text{clean} \qquad \{l_c\} \subseteq \text{dom}(F) \qquad F(l_c) = S'}{(\Sigma, F, \texttt{validate}(x, l_c, l_t); S) \rightarrow (\Sigma, F, S')}$$

$$\text{[S-VALIDT]} \qquad \frac{\Sigma \vdash x = \text{tainted} \qquad \{l_t\} \subseteq \text{dom}(F) \qquad F(l_t) = S'}{(\Sigma, F, \texttt{validate}(x, l_c, l_t); S) \rightarrow (\Sigma, F, S')}$$

Figure 3.2: Operational Semantics of Nano-PHP.

We define the tainted flow problem as follows.

Definition 1 THE TAINTED FLOW PROBLEM
***Instance**: a Nano-PHP program P.*
***Problem**: determine if machine can get stuck with execution of P.*

Before we move on to describe the previous solution to the tainted flow problem, a quick note about functions is in order. In this dissertation we describe an intraprocedural analysis. Hence, we conservatively consider that input parameters and the return values of called functions are all definitions from source. A context insensitive, interprocedural version of the algorithms in this dissertation can be produced by simply creating assignments from actual to formal parameters. We opted for not doing it due to an engineering limitation: our limited knowledge of `phc` has hindered us so far from crossing the boundaries of functions.

3.3 Data-flow Analysis

Given a Nano-PHP program, we can solve the tainted flow problem using a *forward-must* data-flow analysis. The analysis is *forward*, because data flows from predecessors to successors labels, and it is *must*, because properties must hold on all joined paths.

Our analysis binds information to *program points*. Following Appel and George [2001], we define a program point as the region between a pair of consecutive Nano-PHP labels, i.e., edges on a control-flow graph. We decided towards this approach, because validators may bind different taint values for each successor. We represent data-flow information with the function $[\![\_]\!] : L \to L \to Var \to Abs$. This function associates to each program point (l, l') a map storing the abstract values of each program variable at that program point. We use the notation $[\![l_1, l_2]\!]$ to denote information at (l_1, l_2); it abbreviates the function application $(([\![\_]\!]l_1)l_2)$. It is important to note that $[\![\_]\!]$ is a lattice and that this lattice is finite since the sets L, Var, and Abs are finite.

Table 3.3 defines the transfer functions $(Var \to Abs) \to (Var \to Abs)$ associated to each instruction. The initial state of the analysis associates undefined ($\bot$) to all program variables, i.e., $[\![\_]\!] = \lambda l_1 \,.\, \lambda l_2 \,.\, \lambda v \,.\, \bot$. We use $[\![l_1, l_2]\!]$ in this definition as abbreviation for the function application $(([\![\_]\!]l_1)l_2)$. We let $PRED(l)$ be the set of program points immediately before label l, and we define the auxiliary function $JOIN$ as follows:

$$JOIN(l) = \bigsqcap [\![l_i, l]\!] \;, \qquad l_i \in PRED(l)$$

Given two functions $[\![k', k]\!]$ and $[\![l', l]\!]$, we define $\bigsqcap([\![k', k]\!], [\![l', l]\!])$ as $\lambda v \,.([\![k', k]\!]v) \sqcap ([\![l', l]\!]v)$, where the meet operator $\sqcap$ is given by Table 3.2. We denote the successor of a given label l by l_+, whenever this successor is unique. The combined transfer function $tr : [\![\_]\!] \to [\![\_]\!]$ is defined as usual with the composition of all individual transfer functions. Very important to note is that tr admits fix-point as the lattice is finite and all individual transfer functions are monotone.

The join operation denotes accumulation of information across control flow edges; in this case, predecessor edges. Note that we define operation $JOIN$ using the join operator over function elements. The semantics of this operation is to apply the join over abstract values elementwise on the image of the functions according to the definition on Table 3.2. For example $\{x \mapsto clean, y \mapsto clean\} \sqcup \{x \mapsto tainted\} = \{x \mapsto clean \sqcup tainted, y \mapsto clean\} = \{x \mapsto tainted, y \mapsto clean\}$.

l	$[\![\_]\!]$
$x = \circ$	$[\![l, l_+]\!] = JOIN(l) \setminus [x \mapsto \text{tainted}]$
$\bullet = x$	$[\![l, l_+]\!] = JOIN(l)$
$x = \otimes(x_1, \ldots, x_n)$	$[\![l, l_+]\!] = JOIN(l) \setminus [x \mapsto JOIN)(l)(x_1) \sqcap \ldots \sqcap JOIN(l)(x_n)]$
`bra` $l_1, \ldots, l_n$	$[\![l, l_i]\!] = JOIN(l), 1 \leq i \leq n$
$x =$ `filter`	$[\![l, l_+]\!] = JOIN(l) \setminus [x \mapsto \text{clean}]$
`validate` x, l_c, l_t	$[\![l, l_c]\!] = JOIN(l) \setminus [x \mapsto \text{clean}]$ $[\![l, l_t]\!] = JOIN(l)$

Table 3.3: Data-flow equations to solve the Tainted Flow Problem.

```
worklist ← l0
while (not worklist.empty) do
 l ← worklist.pop
 foreach (l+, SUCC(l)) do
  execute ⟦ l, l+ ⟧
  if (changed l+)
   worklist.insert l+
 done
done
```

Figure 3.3: Worklist algorithm for data-flow.

The data-flow equations illustrated in Table 3.3 can be solved using a chaotic algorithm [Schwartzbach, 2010]. Consider for instance a bag containing all data-flow equations required to solve our problem. We could remove from the bag and apply one equation at a time. If the solution does not change, select another equation and continue with the process. However, if the solution changes, we must throw all applied equations back to the bag in order to be processed again. The algorithm reaches a fix-point when the bag is empty. We can guarantee that the algorithm stops, because we are using a finite lattice. There is another algorithm based on a worklist that can solve these equations efficiently.

The worklist data-flow algorithm is shown in Figure 3.3. The algorithm starts by initializing a worklist with the first instruction, here represented by the label $l_0 \in L$. The algorithm ends when the list becomes empty. In every loop iteration, a label is removed from the list, and for each of its successors (l_+), we calculate their transfer equation according to Table 3.3. Note that *SUCC* contain a list of successor labels, like *PRED*'s contains predecessors. After an equation is applied, we check whenever the solution has changed for that edge. If it did, we must insert the successor on the list to be processed later, so that its successors can account this new updates.

Illustrative Example: The right side of Figure 3.4 illustrates the result of the data-flow analysis for the program in Figure 3.1 using the worklist algorithm. On the left, we interrupt the computation before processing the label l_7 in order to demonstrate the worklist algorithm in details. At this point, we process l_7 and insert l_3 in the list to be processed again. Later, we remove l_3 from the worklist head and execute its respective data-flow equation according to Table 3.3. We update its successors edges, (l_3, l_4) and (l_3, l_8), with the new value of x obtained after processing l_5, from *clean* to *tainted*. We propagate the *tainted* value for x after processing l_4 and l_8. However, after executing the transfer function $[\![l_5, l_6]\!]$ we notice that the state has not changed, therefore we do not insert l_5 back to the list. In this example, we update twice the paths $l_3 \to l_4 \to l_5$ and $l_3 \to l_8 \to l_9$ before reaching a fix-point. Note that this example contains a tainted flow vulnerability, given by the path $l_5 \to l_6 \to l_7 \to l_3 \to l_4$, along which it is possible to feed a tainted value for variable x to a sink function.

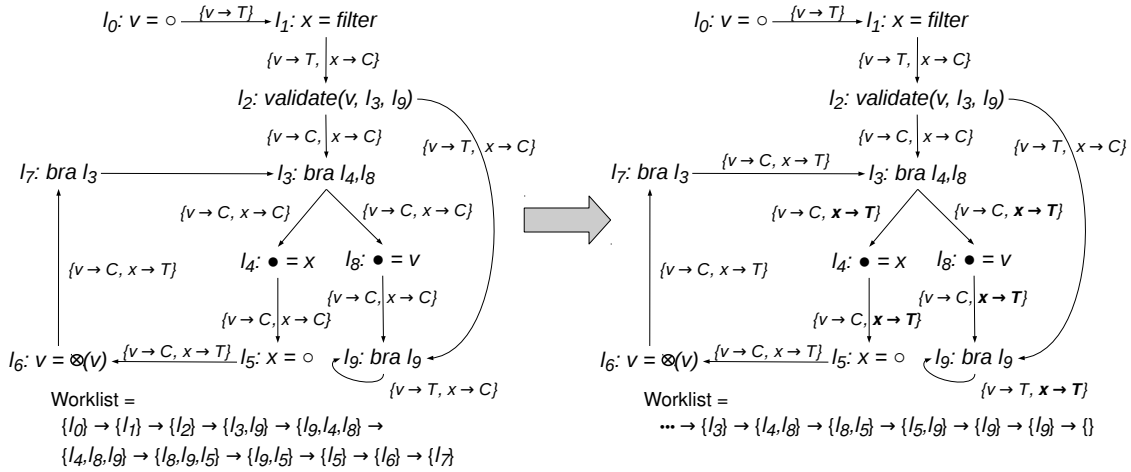


Figure 3.4: Partial (left) and full (right) computation of the data-flow worklist algorithm for the example in Figure 3.1. In the bottom we show the intermediate worklist stages for each algorithm iteration.

Complexity: To estimate the worst-case complexity of this analysis, we notice that if the control-flow graph of the input program has N nodes and V variables then we can perform at most $2 \times N \times V$ iterations, because we can change our abstract state at most twice – from `undefined` to `clean` to `tainted`. Each union operation is $O(V)$, and for each iteration we may have to perform $O(N)$ unions. Thus, our data-flow analysis has the standard [Schwartzbach, 2010] complexity $O(V^2 \times N^2)$. Notice; however, that it is possible to reduce this complexity by executing the transfer function in a topological order of the program's dominator tree [Appel and Palsberg, 2002]. The worklist algorithm presented in Figure 3.3 cope with this ordering, because the instructions are processed before their successors; thus respecting the dominance tree ordering. In particular, Palsberg [1995]; Ørbæk and Palsberg [1997] give an $O(N^3)$ type-inference algorithm that solves the tainted flow problem.

3.4 Conclusion

In this chapter we gave a formal definition of the tainted flow problem in terms of data-flow equations based on a small language definition designed specifically to deal with PHP programs. We gave the semantics of this language in terms of an abstract state machine. We implemented this approach to serve as a baseline to compare the analysis that we proposed in this work in Chapter 4.

Chapter 4

Tainted Analysis as Graph Reachability

In this chapter we describe our solution to the tainted flow problem. Our approach is divided into the three parts below. We determine time complexity in terms of the number of variables (V) in the source program, discussed further in Section 4.4.

1. Convert the input program to the *extended static single assignment* (e-SSA) form. The construction of the dominator tree is $O(V\alpha(V))$, where α is the inverse Ackerman function [Sundblad, 1971], normally regarded as constant, and the insertion of ϕ-functions is $O(V^2)$.

2. Traverse the e-SSA form program collecting use-chains: $O(V)$.

3. Use the algorithm in Figure 4.5 to find tainted flow vulnerabilities: $O(V^2)$.

In Section 4.1 we show the benefits of using the e-SSA intermediate representation and how to augment the Nano-PHP language to include two new instructions added by this representation: ϕ- and σ-functions. In Section 4.2 we model the tainted flow problem as a graph reachability, and demonstrate how to build the constraint graph. The SSA form used by the `phc` makes explicit the effect of alias for definitions, but not for variable uses [Chow et al., 1996]. For validators, we must filter variables and its aliases that is addressed in Section 4.3. In Section 4.4 we provide an equivalent algorithm to solve the tainted flow problem without actually building the graph. We discuss the complexity in terms of space and time with an illustrative example. We conclude in Section 4.5.

4.1 e-SSA form is the Linchpin of Fast Tainted Flow Analysis

We use the *extended Static Single Assignment* (e-SSA) representation to simplify our tainted variable analysis. Information about this representation can be found in Section 2.1. The e-SSA representation has two advantages:

1. Bind information to variables.

2. Validate inputs in conditionals.

The first advantage of e-SSA is that it is possible to bind information directly to variables. In data-flow algorithms, we must associate variables for every program point turning the analysis dense. Since every variable definition is unique in this representation, a state associated to a variable may persist unmodified during the whole analysis. In our work we associate information as either *tainted*, meaning the variable may contain harmful data, or as *clean*, which indicates that it is safe to use. Our analysis is spare because we avoid tracking pairs of variable with states for each program point. The second advantage is the possibility of acquiring useful information from the outcome of conditional tests. In our work, we identify function that validates conditional inputs such as *is_ numeric* and bind information accordingly.

In Section 3.1 we described a language called Nano-PHP to model data-flow algorithms. Now we convert a Nano-PHP program to e-SSA form using the following algorithm:

1. For each instruction $i =$ `validate` x, l_c, l_t:

   a) replace i by a new instruction `validate` x, x_c, l_c, x_l, l_t, where x_c and x_l are fresh variables;

   b) rename every use of x dominated by l_c to x_c. A label l dominates a use of variable x at label l_u if, and only if, every path from the program's entry point to l_u goes across l.

   c) rename every use of x dominated by l_t to x_t;

2. Convert the resulting program into SSA form. For a fast algorithm, see [Appel and Palsberg, 2002, p.410].

In order to represent Nano-PHP program in e-SSA form, we modify the syntax of this language in two ways. First, we extend the language with ϕ-functions according to SSA standards [Cytron et al., 1989]. A ϕ-function such as $x_n = (x_1, \ldots, x_m)$, placed at label l has the effect of assigning $x_i, 1 \leq i \leq m$ to x_n, depending on which predecessor of l was last visited before execution reached l. Second, we modify the syntax of the validator instruction, which become `validator` (x, x_c, l_c, x_t, l_t)[1]. Conceptually, the validator splits the live range of variable x in two parts, depending on whether or not its abstract value is tainted. Note that when converting a program into e-SSA form, we rename every use of x in labels dominated by l_c to x_c, and rename every use of x in labels dominated by l_t to x_t. The new instruction has the following semantics:

[1]Bodik et al. [2000] use special instructions called π-functions to create x_c and x_t.

$$\text{[S-EssaC]}\quad \frac{\Sigma \vdash x = \text{clean} \qquad \{l_c\} \subseteq \text{dom}(F) \qquad F(l_c) = S'}{(\Sigma, F, \texttt{validate}(x, x_c, l_c, x_t, l_t); S) \rightarrow (\Sigma \setminus [x_c \mapsto \text{clean}], F, S')}$$

$$\text{[S-EssaT]}\quad \frac{\Sigma \vdash x = \text{tainted} \qquad \{l_t\} \subseteq \text{dom}(F) \qquad F(l_t) = S'}{(\Sigma, F, \texttt{validate}(x, x_c, l_c, x_t, l_t); S) \rightarrow (\Sigma \setminus [x_t \mapsto \text{tainted}], F, S')}$$

Rule S-EssaC says that after passing a clean variable x through a validator we can rely on the fact that it will be clean henceforth. Given that every use of x dominated by l_c has been renamed to x_c beforehand, we simply continue the program execution in an environment where x_c is bound to clean. Rule S-EssaT does the opposite: if a validator fails on a variable x, we know that x is tainted; hence, we continue the program execution in an environment where x_t is bound to tainted.

The e-SSA representation allows us to acquire static information from the outcome of conditionals. Hence, we can associate unique constraints to variables, as Figure 4.1 illustrates. The original program in Figure 3.4 contains two variables, x and v. We know that these variables are clean in some program points, but not all. The e-SSA representation allows us to identify these program points precisely. The modified program has five variables created after v: $\{v_0, v_2, v_3, v_4, v_7\}$, plus three variables created after x: $\{x_1, x_5, x_6\}$. Let's consider this first group of variables. Given that v_0 is produced by source assignment, we know that it is tainted. Variable v_2 must be necessarily clean, as it is produced by the validation of v_0. On the other hand, v_3 must be necessarily tainted, for the opposite reason. Variable v_7, which results from the application of an operation – assignment – on a clean variable, is also clean. Finally, v_4, which may be assigned either a clean or a tainted value, is tainted, as this is the most conservative choice to detect security vulnerabilities.

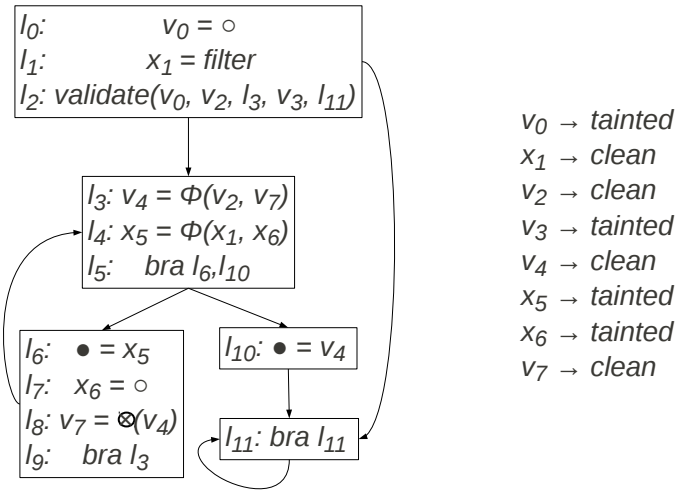


Figure 4.1: The example of Figure 3.1 converted into e-SSA form.

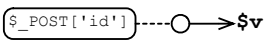
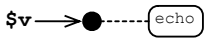
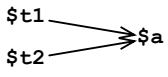
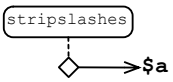
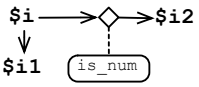
Instruction	Example	Nodes
$v = \circ$	`$v = $_POST['id']`	
$\bullet = v$	`echo($v)`	
$v = \otimes(v_1, \ldots, v_n)$	`$a = $t1 * $t2`	
$v =$ `filter`	`$a = stripslashes($t1)`	
$v = \phi(v_1, \ldots, v_n)$	`$v = phi($v1, $v2)`	
`validate` (v, v_c, l_c, v_t, l_t)	`if(is_numeric($i))`	

Table 4.1: Mapping program instructions to nodes in the reachability graph.

4.2 Graph Reachability Model

Given a Nano-PHP program P, we represent it as a graph G, in which each node $n_v \in G$ denotes a variable $v \in P$. We build the reachability graph directly from the e-SSA form Nano-PHP program. Each particular type of instruction produces a specific configuration of nodes in the reachability graph, as Table 4.1 shows. Roughly, there is an edge linking n_u to n_v if information flows from variable u to v. Notice that, were it not for filters and validators, our reachability graph would represent the def-use chains of the Nano-PHP program [Appel and Palsberg, 2002]. The program from Figure 4.1 gives origin to the disconnected reachability graph in Figure 4.2.

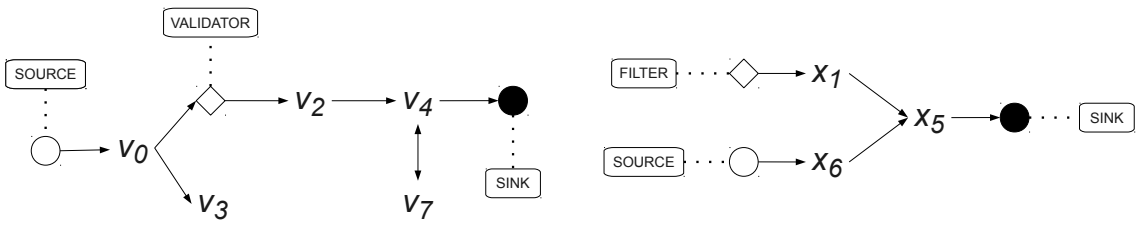


Figure 4.2: The reachability graph for variable v (left) and x (right) built after the program in Figure 4.1.

Definition 2 rephrases the tainted flow problem as an instance of graph reachability. The traversal of the reachability graph is related to the notion of program slicing [Weiser, 1981]. Any node u that reaches a node v is part of the program slice that defines the behavior of v.

Definition 2 The Tainted Flow Problem as Graph Reachability

Instance: *a graph G that describes a Nano-PHP program P.*

Problem: *determine if G contains a path from a source to a sink that does not cross any sanitizer.*

4.3 Addressing Aliasing with HSSA

Aliasing is a phenomenon typical of imperative languages, in which two names reference the same memory location. Aliasing makes static analyses difficult, because it requires the analyzer to understand that updates in the state of a variable may also apply to other variables. To see the implications of aliasing on tainted flow analysis, let's consider the PHP program in Figure 4.3 (Left). Assuming that `$_GET` is a source and `echo` is a sink, then the program is logically bug free. That is, the name `$i`, which is used in a sink, has been sanitized as name `$j`, because both names, `$i` and `$j` represent the same variable. The ordinary e-SSA representation will not catch this subtlety, as Figure 4.3 shows. There is a clear path from `$i0` to the sink that does not go across any sanitizer.

In order to deal with aliasing we use an augmented flavor of the e-SSA representation that we derive from a representation called *Hashed Static Single Assignment (HSSA)* form [Chow et al., 1996]. This last program representation is used internally by `phc`. For each assignment $v = E$ in a SSA form program, the equivalent HSSA form program contains an assignment $(v, a_1, \ldots, a_n) = E$, where $a_1, \ldots a_n$ are the aliases of v at the assignment location. Following this strategy, our augmented representation generates new names for each variable created by sanitizer. The literature contains a

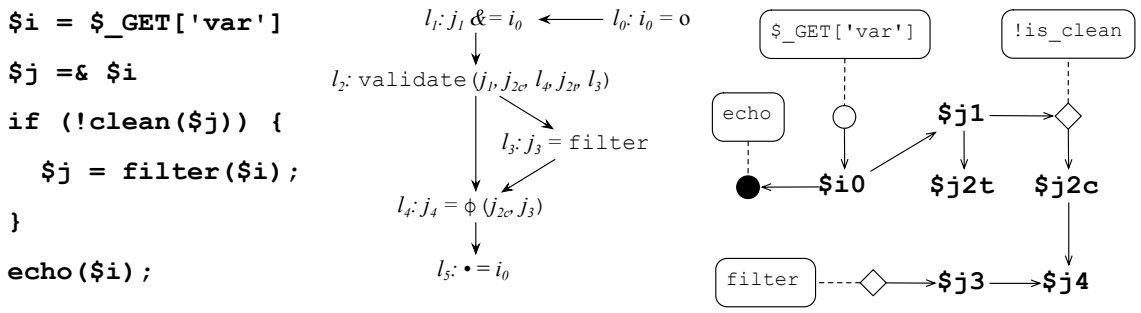


Figure 4.3: An example of how aliasing complicates the tainted flow analysis. In the right side we show the reachability graph built for the e-SSA form program.

plethora of methods to conservatively estimate the set of aliases of a variable. In our implementation we use the flow sensitive, interprocedural analysis [Pioli et al., 1999] that we obtain from `phc`. Figure 4.4 shows the program and the reachability graph after augmenting the e-SSA form program in Figure 4.3 with the results of alias analysis. In the new reachability graph there is no path from a source to a sink that does not go across a sanitizer. Thus, we can prove that the program is bug-free.

4.4 A Solution Quadratic in Time and Space

The function *markTaintedVars*, given in Figure 4.5 finds bugs in e-SSA form Nano-PHP programs. We will be using SML/NJ's syntax to describe the algorithms, adding Erlang's guards in pattern matching, as in the auxiliary function *hasTaintedChild*. This function simulates a traversal of the reachability graph that we described in Section 4.2, but it does not really build the graph. Instead, it relies on the *use-chains* of the variables to guide the traversal. The use-chain of a variable x is a function USE that maps x to every instruction where this variable is used.

The *markTaintedVars* function receives three parameters: a set $\{i, i_1, \ldots, i_n\}$ of instructions to process, an environment Σ that maps variables to one of the abstract states clean or tainted, and a set of visited instructions, which we keep to avoid visiting the same instruction twice. Function *markTaintedVars* processes each instruction forwardly, i.e., an instruction that defines a variable x is buggy if any of the instructions that use x is buggy. We assume that every variable used in a sink function is buggy. We use the auxiliary function *hasTaintedChild* to check if any of the instructions in the use chain of a variable x defines a variable that has been set as tainted in the environment. Notice that neither *markTaintedVars* nor *hasTaintedChild* deals with switches or filter instructions. These instructions will never define or use tainted variables, and will never be found by any of these functions.

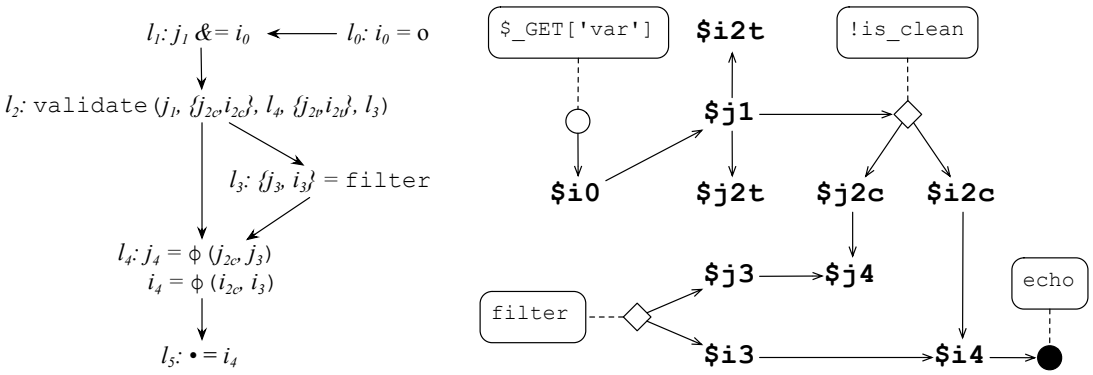


Figure 4.4: (Left) input program in e-SSA form augmented with the results of alias analyses. (Right) final reachability graph.

```
fun hasTaintedChild _ {..., (• = x), ...} ⇒ true
  | hasTaintedChild Σ {..., (x = ⊗(...)), ...} ∧ Σ ⊢ x = clean ⇒ true
  | hasTaintedChild Σ {..., (x = φ(...)), ...} ∧ Σ ⊢ x = clean ⇒ true
  | hasTaintedChild Σ {..., (validate(_, _, _, x, _)), ...} ∧ Σ ⊢ x = clean ⇒ true
  | hasTaintedChild _ _ ⇒ false
fun markTaintedVars ∅ Σ _ ⇒ Σ
  | markTaintedVars {i, i_1, ..., i_n} Σ V ⇒
    let
      val V' = {i} ∪ V
      fun auxiliary v =
        let
          val N = USE(v) \ V'
          val Σ' = markTaintedVars ({i_1, ..., i_n} ∪ N) Σ V'
        in
          if hasTaintedChild Σ' USE(v)
          then Σ'[v ↦ tainted]
          else Σ'
        end
    in
      case i of
        • = x → markTaintedVars {i_1, ..., i_n} Σ[x ↦ tainted] V'
        x = ∘ → auxiliary x
        x = ⊗(...) → auxiliary x
        x = φ(...) → auxiliary x
        validate x, x_c, l_c, x_t, l_t → auxiliary x_t
    end
```

Figure 4.5: The algorithm that finds bugs in Nano-PHP programs.

Complexity: The function *markTaintedVars* is quadratic in time and space. Because *markTaintedVars* keeps the use-chains of every variable, this function uses $O(V \times I)$ space, where V is the number of variables in the input program, and I is the number of instructions in this program. The function is recursively called at most once per each program instruction. When the function is called, it might do a linear search on the use-chain of a variable, inside the function *auxiliary*. Therefore, this function has time complexity $O(I^2)$.

4.5 Conclusion

In this chapter we augmented the Nano-PHP language designed in Section 3.1 to cover the two new instructions added by the e-SSA representation: (i) ϕ-functions; and (ii) σ-functions for validators. The e-SSA representation is the reason behind our

ability to model the tainted flow problem as graph reachability discussed in Section 6.2. We showed how to address aliasing in our program representation and provided an algorithm to solve the tainted flow problem without actually building the reachability graph. We determined the algorithm complexity that is quadratic in terms of time and space; hence improving the data-flow complexity of $O(V^3)$ to $O(V^2)$, where V is given by the number of program variables.

Chapter 5

Experiments and Evaluation

We implemented the data-flow analysis discussed in Section 3 and our e-SSA based analysis from Chapter 4 on top of the `phc open source` compiler [Biggar et al., 2009a]. Our e-SSA intermediate representation implementation is now part of this compiler codebase. We use both implementations to evaluate our solution under two scopes: (i) efficiency; and (ii) precision. In Section 5.3, we compare our analysis in terms of time against the data-flow approach. We show that our approach is generally more efficient than its data-flow counterpart for larger PHP programs. In Section 5.4, we show that our tool is able to find real security vulnerabilities on well-known PHP programs. Both approaches, our solution with e-SSA and the data-flow, have the same precision; hence they found exactly the same bugs. We describe our experiments set-up in Section 5.1 and the benchmarks used in this work in Section 5.2.

5.1 Set Up

We have configured our tool to use sets of sinks, sources and sanitizers to identify two different attack vectors. Cross-site scripting attacks, which we described in Section 2.3.1, and SQL injections, described in Section 2.3.2. We implemented our tool to be easily extensible to cover other types of bugs, such as unwanted command execution (Section 2.3.4). We run the experiments shown in this work on a Pentium Core2Duo 2.0Ghz computer with 3Gb of RAM memory running Ubuntu 9.04.

The `phc` compiler considers each PHP file as a possible entry point, exactly how it happens on a web server. Every PHP file is eligible to be executed as if it is the main function. The `phc` compiler does a simple symbolic execution in order to follow the instructions as they would execute, and only analyzes functions that are reachable by the program's execution path. To analyze a PHP file, we enter its directory base and start the compiler with our analysis embedded. This permit us to resolve includes relative to the execution path. To increase the ability of `phc` to resolve include files automatically, we transform the include query path into a regular expression and search the file system starting from the root directory of the PHP package. We wrote this feature that it is now available for `phc`. We could resolve include files manually, but the enormous amount of PHP files considered make this approach prohibitive.

5.2 Benchmarks

We have run our analysis on 32 publicly available PHP applications. All selected benchmarks were content management systems (CMS) that contains user interaction via the web page and extensive database connection and access. We tried to use as many benchmarks seen in previous works as possible [Jovanovic et al., 2006a,b; Xie and Aiken, 2006]. We installed these CMS on the target machine to simulate an actual configuration on a production server.

Out of the 20,900 PHP files considered in our evaluation, our tool was able to process 13,297 (63.6%). The Table 5.1 summarizes the problems that hinder us to perform our analysis on the remaining 7,603 files (36.4%). These failures occur before we get the chance to run our tainted flow analysis. The first two columns describe the benchmark with their respective versions used in this work. The following two columns indicate parsing and internal failures on the compiler. The fifth column shows the number of failures due to insufficient memory. Columns six and seven show missing functions and missing classes respectively and a count of how many of them had unresolved includes. The eighth column shows the number of missing features not supported by the compiler. The ninth column sums the total failures with their respective failures rate per benchmark.

CMS	Version	Parse	Int.	Mem.	Funct.(Incl.)	Class(Incl.)	Feat.	Total(%)
Wordpress	2.9.2	71	2	1	70(4)	62(34)	0	206(73.6%)
Drupal	6.16	0	0	0	14(0)	0(0)	5	19(30.2%)
Joomla	1.5.17	0	0	6	3(0)	420(74)	0	429(40.6%)
Frog CMS	0.9.5	0	0	0	24(0)	46(3)	4	74(58.3%)
SilverStripe	2.3.7	0	2	2	0(0)	449(15)	5	458(92.0%)
Mambo	4.6.5	6	0	3	16(5)	66(25)	0	91(19.8%)
TYPOlight	2.8.3	0	1	5	2(1)	306(28)	5	319(55.9%)
Concrete5	5.4.0.5	0	0	3	377(11)	471(196)	42	893(88.9%)
Textpattern	4.2.0	2	0	5	1(0)	15(7)	0	23(30.3%)
Symphony	2.0.7	0	1	0	3(0)	69(28)	45	118(77.1%)
MODx CMS	1.0.3	4	19	23	14(0)	24(11)	7	91(20.6%)
CMS Made Simple	1.7.1	1	0	103	68(29)	184(34)	0	356(38.8%)
ImpressCMS	1.2.1	0	1	56	14(2)	448(278)	18	537(49.8%)
Exponent CMS	0.97	127	1	2	75(56)	95(62)	8	308(8.9%)
Mia CMS	4.9.0	14	0	3	16(5)	65(19)	0	98(19.6%)
Jojo CMS	1.0rc2	0	2	3	18(1)	358(88)	6	387(63.6%)
Elxis CMS	2009.1 hecate	34	0	11	12(3)	635(38)	1	693(41.5%)
Chyrp	2.0	1	20	0	3(0)	31(3)	3	58(68.2%)
DCP Portal	7.0beta	3	0	3	22(2)	106(23)	1	135(25.2%)
Dragon Fly CMS	9.2.1	0	0	2	5(4)	15(8)	0	22(7.7%)
MemHT Portal	4.0.1	0	1	62	63(23)	36(4)	2	164(46.3%)
Pligg	1.0.4	2	0	3	103(73)	30(17)	1	139(39.6%)
RunCMS	2.1	4	0	5	15(1)	334(216)	0	358(49.3%)
SPIP	2.1.0	3	0	3	9(0)	14(12)	0	29(4.0%)
TangoCMS	2.5.3	0	1	0	1(0)	285(6)	6	293(84.9%)
WebsiteBaker	2.8.1	5	0	22	1(1)	17(4)	120	165(29.4%)
Xoops	2.4.4	1	0	2	9(2)	432(152)	5	449(45.5%)
sNews	1.7	0	0	0	0(0)	2(0)	0	2(66.7%)
PostNuke	0.764	1	1	42	55(5)	103(29)	1	203(21.0%)
Zikula	1.2.3	0	0	2	39(3)	167(29)	0	208(28.4%)
PHP Fusion	7.0	0	0	3	6(0)	84(84)	0	93(18.3%)
e107	0.7.20	0	1	15	165(162)	3(2)	1	185(24.6%)
Total	-	279	53	390	1223(393)	5372(1529)	286	7603(36.4%)

Table 5.1: List of PHP failure reasons for 32 benchmarks.

We can see in Table 5.1 that we were unable to process 7,603 PHP files. This accounts for 36.4% of all files considered in our evaluation, a relative high number. From this, the most failures were caused by missing functions and classes: 6,595 (86.7% of the total failures). Even if we resolve all the include files we would still fail on 4,673 PHP files (61.5% of the total failures), possibly because these files should not be reachable directly, but through an inclusion. However, resolving this includes statically would lead to a higher number of instructions to analyze, possibly failing later due to memory restrictions because of `phc`'s expensive whole-program analysis [Biggar et al., 2009a]. Therefore, we are limited to execute our analysis only on small PHP files. Nevertheless, the compiler was able to process a significant fraction of files and to report previously unknown bugs when augmented with our analysis.

5.3 Efficiency

In order to show the efficiency of our application, we selected PHP files containing more than 100 instructions. We striped the `phc` optimizer from its dead code elimination pass in order to obtain even larger programs. We found 1,122 files matching this criterion along 30 benchmarks. The SilverStripe and Concrete5 had no candidates. We set both the data-flow and our solution with e-SSA to run 10 times for each of these files, and we collect average times for them. For our analysis, we calculated the average time to build the intermediate representation (e-SSA) and to run our tainted flow analysis.

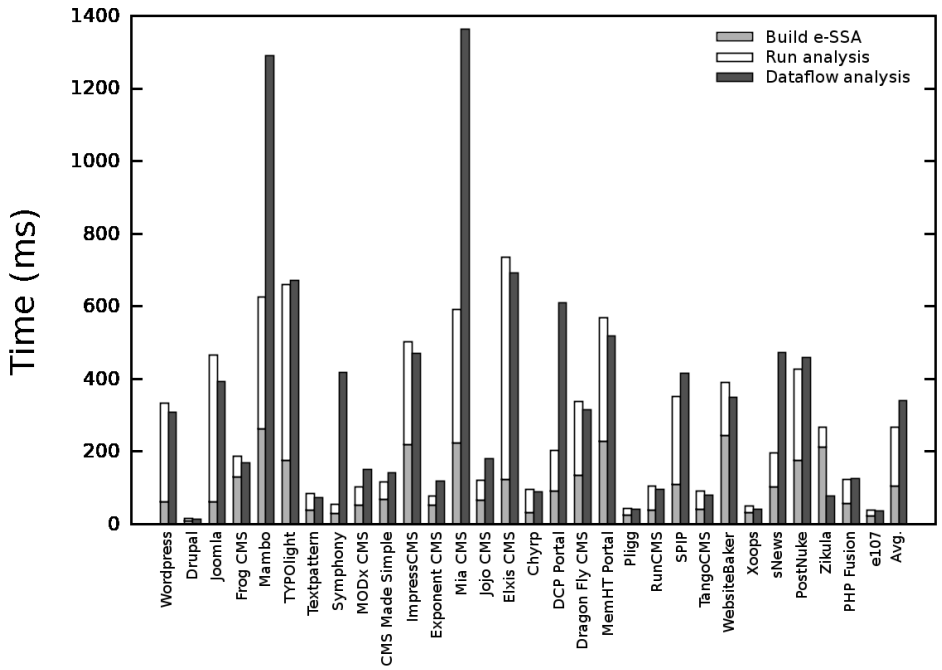


Figure 5.1: Average execution time for each benchmark in milliseconds for the data-flow analysis and our solution with e-SSA – split into build the IR and run the analysis).

10%	20%	30%	40%	50%	60%	70%	80%	90%	100%
68.75%	57.14%	38.10%	31.25%	28.70%	26.00%	23.69%	21.40%	20.21%	18.98%
97.32%	97.32%	98.21%	98.66%	98.75%	98.81%	98.98%	99.11%	99.21%	99.23%

Table 5.2: From the 1,122 PHP files selected, we group larger files in rates ranging from 10% to 100% (Line 1). We compared the rate gain of our approach versus data-flow analysis (Line 2) and our approach gain without the time to build the IR (Line 3).

In Figure 5.1 we show the average execution time for our analysis, divided into building the IR and running the tainted flow analysis against the data-flow approach. Note that we are not considering the time consumed by the `phc` to perform its whole-program analysis. It is possible to observe that our approach has executed faster in 13 (43.34%) of the total benchmarks used. If we do not take the time to build the IR into consideration, our approach is only slower than the data-flow on the *Joomla* benchmark. If we implement the conversion to the e-SSA intermediate representation more efficiently [Cooper et al., 2001], we may achieve further improvements.

The largest function that we have analyzed contains 1,141 basic blocks – a small size to stress the run-time of both solutions. Neither analyzed file surpassed the threshold of 1.5s – a very low limit. However, it is possible to observe that our approach has stayed beneath 800ms, while two benchmarks using data-flow almost reached 1.4s. We speculate that once we cross the boundaries of functions, and start analyzing whole PHP applications, which might contain thousands of functions, and millions of lines of code, our analysis will be much more efficient than the data-flow approach. Our base compiler, `phc`, does not give us enough infra-structure to perform whole program analyses yet, but its developing community is working to overcome this limitation.

From the 1,122 PHP files selected with more than 100 instructions, we organized in Table 5.2 the largest files ranging from 10% to 100% in order to show how fast our analysis was compared to data-flow analysis. For instance, for the 10% largest files, our analysis with e-SSA was faster than data-flow almost 70% of times. As we increase the number of files, the average number of instructions per file decreases; thus decreasing the rate gain of our approach. We speculate that expressive gains could be achieved by processing larger files that could amortize the time to convert the program to e-SSA. If we desconsider the time to build the IR, our approach consistently surpasses data-flow analysis as shown in line 3 of Table 5.2. This confirms the low cost of our analysis.

5.4 Precision

Both our e-SSA based analysis and the data-flow analysis have reported 63 warning messages across 25 distinct PHP files. Table 5.3 details these numbers for the subjects that contain confirmed vulnerabilities. Manual inspection of each of these warnings revealed actual vulnerabilities in 36 of these reports; around 45% false positive ratio. We used this list of bugs to perform malicious injections of html code containing JavaScript. We submitted all these vulnerabilities to the bugtraq[1]. We show the original advisory sent with detailed information about the bugs found in Appendix B.

[1] `http://www.securityfocus.com/`

subject	version	files					warnings	
		total		processed		affected	TP	FP
		#	LOC / #	#	LOC / #			
MODx	1.0.3	472	231	308	228	3	1	1
Exponent CMS	0.97	3456	42	2833	32	3	28	11
DCP Portal	7.0beta	535	97	392	61	7	5	11
Pligg	1.0.4	380	146	179	154	3	1	0
RunCMS	2.1	737	134	361	86	2	1	6
avg.	-	-	-	-	-	3.6	7.2	5.8

Table 5.3: Experimental results of subjects with true alarms.

Table 5.3 details these numbers for the subjects that contain confirmed vulnerabilities. Column *subject* and *version* list the vulnerable benchmark. Column *total* (files) shows the total number of files in the programs, column *processed* shows the total number of these files that `phc` was able to process, and column *affected* shows the number of files involved in warning reports – for the purpose of inspection it is preferable to have the warnings confined in a small set of files. Average LOCs (commented lines of code) per PHP file are given by columns *total* and *processed*. Column *TP: true positive*, (respectively, *FP: false-positive*) shows the number of confirmed (respectively, spurious) vulnerabilities. False positives arise due to several reasons including: the imprecision of `phc`'s analysis, our intraprocedural analysis, and the logical infeasibility of some program paths, which static analysis typically fail to identify. Some of our test applications use third party software. In particular, *Exponent CMS* and *RunCMS* employed the same spell checking module – a library responsible for six false-positives in each of these two benchmarks. Moreover, the 28 bugs reported for *Exponent CMS* were produced by the output of the same tainted variable in different points of the same program. These are all different, yet similar bugs.

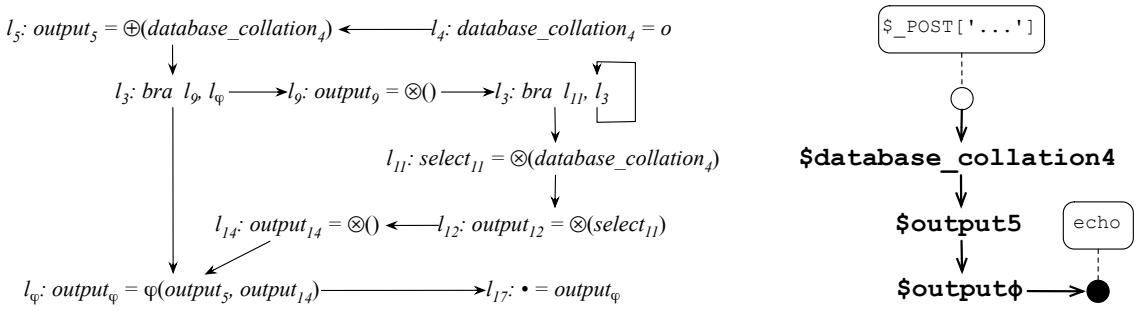


Figure 5.2: (Left) the Nano-PHP representation of the program in Figure 5.2 – we show only the highlighted lines. (Right) The reachability graph.

5.5 An example of a real-world bug

In order to illustrate our analysis, we will show an actual bug that our implementation found in the content management system MODx CMS version 1.0.3. We have reported this bug to the developers[2], and the bug has been fixed since. In this example we use the PHP program of Figure 5.3, which was publicly available on May, 2010.

One of the steps of the installation process lets the user choose a database collation from a small suite of options. Users specify this database via three parameters: `host`, `uid` and `pwd`. Users also specify their choice for a collation system via a string, which the PHP program stores in the variable `database_collation`. The PHP file queries a database, using this variable as a key. However, in case the parameters `host`, `uid` or `pwd` do not determine a valid database, the module receives a single collation option from a variable originated from a post request, i.e., a form. This form contains a string, which is never sanitized inside the program in Figure 5.3. The string in `database_collation` is printed in the output, as we can see in Line 17 of Figure 5.3. Therefore, in order to print a malicious script in the user's webpage, we can choose an invalid host for the database, and write the script code directly in the form that feeds `database_collation`. For instance, we can steal cookies from the user's browsing environment with the string

[2] http://www.securityfocus.com/bid/41454

```
   <?php
1  $host = $_POST['host'];
2  $uid = $_POST['uid'];
3  $pwd = $_POST['pwd'];
4  $database_collation = $_POST['database_collation'];
5  $output = '<select id="database_collation" name="database_collation">
   <option value="'.$database_collation.'" selected >'
     .$database_collation.'</option></select>';
6  if ($conn = @ mysql_connect($host, $uid, $pwd)) {
       // get collation
7      $getCol = mysql_query("SHOW COLLATION");
8      if (@mysql_num_rows($getCol) > 0) {
9          $output = '<select id="database_collationse_collation"
             name="database_collation">';
10         while ($row = mysql_fetch_row($getCol)) {
11             $selected = ( $row[0]==$database_collation ? ' selected' : '' );
12             $output .= '<option value="'.$row[0].'"'.$selected.'>'.$row[0].
                 '</option>';
13         }
14         $output .= '</select>';
15     }
16 }
17 echo $output;
   ?>
```

Figure 5.3: An installation file used by MODx CMS version 1.0.3. This file contains a XSS vulnerability, which we have highlighted in boldface.

"</option></select><script>window.alert(document.cookie);</script>". Our analysis can easily find this vulnerability, as we illustrate in Figure 5.2. The reachability graph that we build for the example program contains a path from the variable `database_collation`, which is initialized from a source, to the function `echo`, which we qualify as a sink.

5.6 Conclusion

We did an evaluation of the tainted flow analysis with the approach proposed by this work using e-SSA against the data-flow solution. We selected 32 `open source` PHP CMS benchmarks to conduct both analyses on them. We have used two criteria in our evaluation: (i) the precision of our analysis; and (ii) the efficiency our of analysis compared to data-flow based approaches. Both approaches have the same precision, and both were able to find the same bugs: 36 vulnerabilities across 5 different CMS systems. We explained in details one of these bugs, demonstrating how the tainted values flows from the source to the sanitizer without proper sanitization. We have also discussed the efficiency of our analysis. In large files, our solution with e-SSA has constantly improved the time to run the tainted flow analysis.

Chapter 6

Related Works

In this chapter we review works that are related to ours. We have organized them into two sections. On the first, we review the broad area of web applications security by giving an historic overview of it. Then, we focus on the tainted flow analysis that our work is based on, and compare our analysis with other security approaches towards improving overall web security in Section 6.1.1. On the last section of this chapter, we discuss works that modeled other problems using a graph reachability approach.

6.1 Web Applications Security

The World Wide Web (WWW) was conceived by Tim Berners-Lee in 1990 to exchange hypertext data between a web server and a web browser. The communication is typically initiated when a browser requests a web page located on the server through the HTTP[1] protocol. The web server identifies the requested page and responds using the same protocol. Initially, web servers were only capable of serving static contents - physical files located on the file system. Therefore, there was no user interaction besides the requesting of the page. To add dynamism to the web, a server extension called Common Gateway Interface[2] (CGI) was created to delegate the page generation capability to a command-line application. This application, often called CGI script, is now capable of processing user input that is carried along the request. CGI scripts are commonly used to process web forms. However, security measures are necessary when handling user data.

Perl[3] has gained popularity as a language for writing CGI scripts. Perl is an imperative dynamic-typed scripting language that is executed by an interpreter. As it happens with other languages, Perl provides critical operations that can be abused by attackers when no security enforcements are employed by the programmers. Puppy [1999] provides a list with several attack venues for Perl scripts. When executed as a CGI scripts, these programs can be exploited by attackers to take control over the

[1] HTTP: Hypertext Transfer Protocol
[2] CGI RFC: http://tools.ietf.org/html/rfc3875
[3] http://www.perl.org

affected web server. Perl has a security mechanism called tainted mode[4] that can be used to prevent attacks from succeeding. When enabled, Perl is executed in a special mode that tracks program inputs across the program execution, marking as tainted dangerous character often used to exploit security flaws. If a value marked as tainted reaches a sensitive operation, Perl blocks that script; thus preventing it from harming the system. However, the tainted mode adds overhead to the program execution and to the web server. The most effective way to prevent such attacks is to identify bugs in advance and fix them. Our approach tackles the latter, finding bugs in source programs so that vendors can provide patches for the security vulnerabilities encountered.

Nowadays, languages such as PHP, ASP, and JSP are directly embedded in web servers through server-integrated modules. Nevertheless, web vulnerabilities in these languages affect the web server in the same way as CGI scripts. In this work we proposed an efficient technique to identify security vulnerabilities described in Chapter 4. Although we focused our efforts in finding bugs in PHP programs, our technique is general enough to handle other languages. We have modeled our solution to fit the tainted flow problem framework discussed in Chapter 3 and we review other works that handle web applications security in Section 6.1.1.

6.1.1 Tainted Flow Analysis

The foundations of the tainted flow analysis were covered in Section 2.3. In this section, we review other works that have used this model and compare to other techniques that provide security to web applications. It is possible to organize different security approaches into three categories, as listed below:

- Whitebox vs. Blackbox Testing
- Server-side vs. Client-side Protection
- Static vs. Dynamic Analysis

Whitebox and blackbox are test design methods. Whitebox takes advantage of the knowledge of the programs internal structure; thus requiring the original source code to be available. In the other hand, blackbox techniques can test a program without the actual source code, by stressing its inputs and observing its consequences in order to spot problems [Godefroid et al., 2005]. Other security enforcements can be provided according to client or server perspectives. The client-side perspective focuses on prevention mechanism on the client browser [Vogt, 2006], while server-side techniques provide security to web server. We can also perform our analysis statically or dynamically. Dynamic analysis relies on the executed program behavior, while static analysis tries to determine the program behavior without actually executing the software. It is important to highlight that the analysis proposed by this work in Chapter 4 is a whitebox, server-side static analysis to find security vulnerabilities in PHP programs.

[4] `http://perldoc.perl.org/perlsec.html#Taint-mode`

The tainted flow problem is well known in the literature [Jovanovic et al., 2006a; Pistoia et al., 2005; Wassermann and Su, 2007; Xie and Aiken, 2006]. WebSSARI [Huang et al., 2004] is possibly the first tool to address the tainted flow problem in the context of web application vulnerabilities. It uses an intraprocedural analysis based on user-specified entries to find tainted variable attacks on PHP programs. However, incomplete or malformed specifications may lead to both false positives and false negatives. Also, many PHP files were not processed because of problems in their parser. Their analysis is in fact a hybrid analysis, whereas they insert run-time guards at critical points in the program that can prevent attacks from succeeding.

Static analysis has been used in another problem domain, to verify html conformance [Brabrand et al., 2001] generated by dynamic web pages. Christensen et al. [2003] provided a string analysis that is capable of modeling every string operation on Java programs using a regular language. Minamide [2005] coped with the same problem, but for PHP programs. Unfortunately, string operations on PHP programs cannot be modeled by a regular language; hence requiring a more expressive language. Therefore, Minamide [2005] modeled its string analysis with context-free grammars. Su and Wassermann [2006] borrowed his technique to verify that no tainted values flow to SQL queries in Java programs, while Wassermann and Su [2007] did the same for PHP programs. Later, Wassermann and Su [2008] extended their analysis to find cross-site scripting as well. Their approach differs from ours because it resorts to a *string analysis* instead of a data-flow analysis. Hence, it is more precise, yet more expensive.

Another strategy to solve the tainted flow problem was proposed by Xie and Aiken [2006] who use a three-tier architecture. Using symbolic execution [King, 1976] it writes block summaries in the basic block level that is later consumed by the intraprocedural analysis. The intraprocedural analysis writes function summaries to be used by the interprocedural analysis; thus making a clear separation between several layers of the analysis. While our analysis has conditional validators powered by the e-SSA representation, their approach tries to infer new functions as validators. However, a direct comparison between their strategy and ours is not possible, because their tool is not publicly available. We can only speculated that, by using symbolic execution, their analysis is more expensive than ours, although possibly more precise.

However, there are publicly available tools that perform tainted variable analysis. One of them is MARCO [Pistoia et al., 2005], a Java bytecode analyzer. Another is Pixy [Jovanovic et al., 2006a], a PHP analyzer. MARCO relies on program slicing [Weiser, 1981] to find the set of tainted variables, whereas Pixy uses a monotone framework that associates to each variable, at each program point, a boolean state that defines if the variable is tainted or clean. Neither tool takes the results of conditional tests into consideration; hence, both are path insensitive – a problem that our intermediate representation permits us to circumvent.

The Pixy tool improves on tainted flow analysis problems because of its embedded alias analysis [Jovanovic et al., 2006b]. They acknowledge that the alias analysis for type-safe languages is unsuitable for PHP and precision can only be achieved with a powerful alias analysis. We solve this issue by using the alias analysis provided by the `phc` compiler, which uses an interprocedural analysis [Pioli et al., 1999]. We have implemented Pixy's data-flow analysis in `phc` in order to compare with our technique

using e-SSA. Data-flow has been widely used, including on the same problem domain. The ASPCW tool provides a data-flow analysis combined with type-inference to find XSS and SQL injectionvulnerabilities in ASP programs [Zhang and Wang, 2010]. We relyed on `phc`'s type-inference algorithm as well to decrease the number of false positives in our analysis, as we show in Chapter 5.

Other works improved their bug finding analysis by providing an efficient interprocedural context-sensitive analysis. Nanda and Sinha [2009] tackles the problem of finding null dereference, a common problem in Java programs, by providing an accurate interprocedural analysis. Livshits and Lam [2005] find security vulnerabilities in Java programs using an efficient context-sensitive representation [Whaley and Lam, 2004], while Bond et al. [2010] did the same using another context representation [Bond and McKinley, 2007]. Tripp et al. [2009] created an industrial-strength tool called TAJ that scales to analyze larger programs in Java. Other works improved on the precision of their tool, using a more accurate analysis. Cadar et al. [2008] created a tool that symbolic executes LLVM intermediate code, while Fu and Qian [2008] applied symbolic execution to find SQL injection in Java programs. However, neither of these techniques is suitable to scripting languages that are difficult to static analyze.

6.2 Graph Reachability

Data-flow analyses are old allies of compiler designers [Kam and Ullman, 1976]. The first influential work to see data-flow analysis as a graph reachability problem was introduced by Reps et al. [1995]. The mapping adopted by Reps et al. [1995] deals with general programs, whereas we use programs in e-SSA form. A disadvantage of the previous approach was the size of the graph that it produces: the number of nodes in the graph is $O(V \times B)$, where V is the number of variables and B is the number of basic blocks in the source program. We avoid this growth, because the e-SSA form tends to increase linearly on the number of variables in the source program, and our graph contains $O(V)$ nodes.

Scholz et al. [2008] have also mapped a flow problem, the *user-input dependence analysis* [Snelting et al., 2006] to an instance of graph reachability. Scholz *et al.* are interested in finding which program variables might be influenced by input data. Contrary to our approach, Scholz *et al.* use a program representation called *Augmented Static Single Assignment (a-SSA)* form. The a-SSA form provides information not present in e-SSA form, because it determines which control structures influence program data. That is, in the program `a := read(); c := (a > 0) ? b[0] : 0;` the value of `a` influences the value of `c`, even though these two variables are not related in e-SSA form. However the user-input dependence analysis does not take sanitizers, e.g., validators, into consideration. Thus, Scholz *et al.*'s a-SSA cannot use information learnt from the outcome of conditionals to bind constraints to variables.

6.3 Conclusion

In this chapter we discussed related works that covered security in web applications. We compared our approach with several others found in the literature. Then we focused on the tainted flow analysis covering works that find security vulnerabilities in PHP applications. We also covered works that solve flow analyses using a reachability graph approach.

Chapter 7

Conclusion

This dissertation has presented a new static analysis technique to identify security vulnerabilities related to tainted variable attacks in PHP programs. We model the tainted flow problem as an instance of a graph reachability that was only possible due to the intermediate program representation called e-SSA [Bodik et al., 2000]. In addition, this representation allowed us to bind tainting information directly to variables avoiding the need to track pairs of variables and programs points; thus making our analysis sparse. Moreover, the e-SSA representation facilitates our analysis to be path-sensitive, because we can take into consideration the outcome of conditional validators.

To evaluate the solution proposed in this work, we decided to implement an iterative data-flow algorithm to cope with the tainted flow problem that has already been addressed in the literature by Pixy [Jovanovic et al., 2006b]. Although data-flow algorithms are not a new concept, the formal model provided in Chapter 3 is a contribution of our work. We have designed a small language subset called Nano-PHP with its operational semantics included. We gave the algorithm in terms of data-flow equations that are used to solve the tainted flow problem.

We have implemented both approaches on top of `phc`, an `open source` PHP compiler. Although Pixy is also an `open source` tool, we decided to re-implement its data-flow algorithm using the same infra-structure in order to perform a more controllable evaluation. We have assessed our solution under two scopes: precision and efficiency. Our tool is precise, because it was able to find real security vulnerabilities in well-known web applications. We have reported the bugs that we found to the maintainers of the affected applications, and some of these developers have acknowledged and fixed the vulnerabilities. Our tool is more precise than Pixy's [Jovanovic et al., 2006b], because our analysis is path-sensitive, but we did not improve the precision compared to other works [Xie and Aiken, 2006; Wassermann and Su, 2007]. It is important to note that our analysis with e-SSA and data-flow are similar in precision; therefore they found exactly the same vulnerabilities. Our analysis is efficient, because it is equivalent to a graph reachability problem, which we have been able to code as a non-iterative data-flow algorithm. In our experiments, we show that our solution with e-SSA tends to become faster than its data-flow counterpart when processing larger PHP files. Our implementation is currently available for the `phc` compiler, and can be found at `http://homepages.dcc.ufmg.br/~rimsa/`.

7.1 Limitations

We decided to perform our tainted flow analysis for PHP programs because its is a widespread language to write web applications. Unfortunately, this language presents unique challenges to static analyzers. To avoid the herculean task of modeling the whole language semantics, we decided to rely on a compiler infra-structure that already performs many useful analyses. We choose the `phc` compiler [Biggar et al., 2009a] for our purposes. Unfortunately, our analysis is conditioned to the same limitations of the compiler [Biggar et al., 2009a, p.9]. Unsupported or missing features affect our pass directly, specially in two of PHP most complicated features to analyze statically: (i) run-time code inclusion; and (ii) run-time code evaluation. These two functionalities can be abused by attackers to gain access to a running system. The `phc` compiler copes with such features by resolving their values at compile-time, if they can be inferred statically. Otherwise, the analyzer is forced to abort due to an incomplete view of the whole program [Biggar et al., 2009a]. Once resolved, the original instruction `include` or `evaluate` is replaced with the correspondent PHP code, leaving us crippled to analyze the original call in search for tainted attacks. Nevertheless, if we could cope with these limitations, these two security vulnerabilities could fit the tainted flow problem and enjoy the benefits of our tainted flow analysis (Chapter 4).

7.2 Future Works

As future works, we would like to transform our analysis from intra- to inter-procedural, to become more precise, and hence report fewer false positives. To achieve this goal, we must modify `phc` whole program analysis to convert the program into SSA (and in e-SSA) on demand, as described in [Biggar and Gregg, 2009, p.63]. The conversion must run alongside other client analysis, such as alias-analysis and type-inference. However, we must overcome `phc` scalability issues concerning memory management before attempting to modify our analysis to surpass the function boundaries. The inter-procedural analysis can only be beneficial if we can process larger files, which unfortunately we still cannot with `phc`. We could modify the compiler points-to analysis to a more efficient context-sensitive analysis by using the bddbddb [Whaley and Lam, 2004] technique or the probabilistic calling contexts [Bond and McKinley, 2007]. Nevertheless, our algorithm is general enough to handle tainted variable attacks in different programming languages and in different application domains.

Bibliography

Aho, A. V., Lam, M. S., Sethi, R., and Ullman, J. D. (2006). *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley.

Ananian, S. (1999). The static single information form. Master's thesis, MIT.

Appel, A. W. and George, L. (2001). Optimal spilling for CISC machines with few registers. In *PLDI*, pages 243–253. ACM.

Appel, A. W. and Palsberg, J. (2002). *Modern Compiler Implementation in Java*. Cambridge University Press, 2nd edition.

Benda, J., Matousek, T., and Prosek, L. (2006). Phalanger: Compiling and running PHP applications on the microsoft .net platform. In *.NET Technologies 2006*, pages 31--38.

Biggar, P., de Vries, E., and Gregg, D. (2009a). A practical solution for scripting language compilers. In *SAC*, pages 1916–1923. ACM.

Biggar, P., de Vries, E., and Gregg, D. (2009b). A practical solution for scripting language compilers. In *SAC '09: Proceedings of the 2009 ACM symposium on Applied Computing*, pages 1916--1923, New York, NY, USA. ACM.

Biggar, P. and Gregg, D. (2009). Static analysis of dynamic scripting languages. Paper Draft.

Bodik, R., Gupta, R., and Sarkar, V. (2000). ABCD: eliminating array bounds checks on demand. In *PLDI*, pages 321–333. ACM.

Boissinot, B., Brisk, P., Darte, A., and Rastello, F. (2009). SSI properties revisited. Technical Report 00404236, LIP Research Report.

Bond, M. D., Baker, G. Z., and Guyer, S. Z. (2010). Breadcrumbs: Efficient context sensitivity for dynamic bug detection analyses. *SIGPLAN Not.*, 45(6):13--24.

Bond, M. D. and McKinley, K. S. (2007). Probabilistic calling context. In *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, OOPSLA '07, pages 97--112, New York, NY, USA. ACM.

Brabrand, C., Møller, A., and Schwartzbach, M. I. (2001). Static validation of dynamically generated HTML. In *Proc. ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, PASTE '01*, pages 221--231.

Brisk, P. (2006). *Advances in Static Single Assignment Form and Register Allocation*. PhD thesis, UCLA - University of California, Los Angeles.

Budimlic, Z., Cooper, K. D., Harvey, T. J., Kennedy, K., Oberg, T. S., and Reeves, S. W. (2002). Fast copy coalescing and live-range identification. In *PLDI*, pages 25–32. ACM.

Cadar, C., Dunbar, D., and Engler, D. (2008). Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 209--224. USENIX Association.

Chow, F. C., Chan, S., Liu, S.-M., Lo, R., and Streich, M. (1996). Effective representation of aliases and indirect memory operations in ssa form. In *CC*, pages 253--267. Springer.

Christensen, A. S., Møller, A., and Schwartzbach, M. I. (2003). Precise analysis of string expressions. In *SAS'03: Proceedings of the 10th international conference on Static analysis*, pages 1--18, Berlin, Heidelberg. Springer-Verlag.

Chugh, R., Meister, J. A., Jhala, R., and Lerner, S. (2009). Staged information flow for javascript. In *PLDI*, pages 50--62. ACM.

Cooper, K. D., Harvey, T. J., and Kennedy, K. (2001). A simple, fast dominance algorithm. Submitted to Software–Practice and Experience.

Cytron, R., Ferrante, J., Rosen, B. K., Wegman, M. N., and Zadeck, F. K. (1989). An efficient method of computing static single assignment form. In *POPL*, pages 25–35.

de Vries, E. and Gilbert, J. (2007). Design and implementation of a PHP compiler front-end. Technical Report TR-2007-47, Dept. of Computer Science, Trinity College Dublin.

Fu, X. and Qian, K. (2008). Safeli: Sql injection scanner using symbolic execution. In *Proceedings of the 2008 workshop on Testing, analysis, and verification of web services and applications*, TAV-WEB '08, pages 34--39, New York, NY, USA. ACM.

Godefroid, P., Klarlund, N., and Sen, K. (2005). Dart: directed automated random testing. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 213--223, New York, NY, USA. ACM.

Hack, S. (2005). Interference graphs of programs in SSA-form. Technical Report ISSN 1432-7864, Universitat Karlsruhe.

Huang, Y.-W., Yu, F., Hang, C., Tsai, C.-H., Lee, D.-T., and Kuo, S.-Y. (2004). Securing web application code by static analysis and runtime protection. In *WWW '04: Proceedings of the 13th international conference on World Wide Web*, pages 40--52, New York, NY, USA. ACM.

Jovanovic, N., Kruegel, C., and Kirda, E. (2006a). Pixy: A static analysis tool for detecting web application vulnerabilities (short paper). In *S&P*, pages 258--263. IEEE.

Jovanovic, N., Kruegel, C., and Kirda, E. (2006b). Precise alias analysis for static detection of web application vulnerabilities. In *PLAS*, pages 27--36. ACM.

Kam, J. B. and Ullman, J. D. (1976). Global data flow analysis and iterative algorithms. *Journal of the ACM*, 23(1):158--171.

King, J. C. (1976). Symbolic execution and program testing. *Commun. ACM*, 19(7):385--394.

Lengauer, T. and Tarjan, R. E. (1979). A fast algorithm for finding dominators in a flowgraph. *ACM Trans. Program. Lang. Syst.*, 1(1):121--141.

Livshits, V. B. and Lam, M. S. (2005). Finding security vulnerabilities in java applications with static analysis. In *Proceedings of the 14th conference on USENIX Security Symposium - Volume 14*, pages 18--18, Berkeley, CA, USA. USENIX Association.

Minamide, Y. (2005). Static approximation of dynamically generated web pages. In *WWW*, pages 432--441. ACM.

Nanda, M. G. and Sinha, S. (2009). Accurate interprocedural null-dereference analysis for java. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 133--143, Washington, DC, USA. IEEE Computer Society.

Ørbæk, P. and Palsberg, J. (1997). Trust in the λ-calculus. *Journal of Functional Programming*, 7(6):557--591.

Palsberg, J. (1995). Efficient inference of object types. *Inf. Comput.*, 123(2):198--209.

Pereira, F. M. Q. and Palsberg, J. (2005). Register allocation via coloring of chordal graphs. In *APLAS*, pages 315–329. Springer.

PHP (2010). Zend engine: PHP interpreter. http://www.zend.com.

Pioli, A., Burke, M., and Hind, M. (1999). Conditional pointer aliasing and constant propagation. Technical Report 99-102, SUNY at New Paltz.

Pistoia, M., Flynn, R., Koved, L., and Sreedhar, V. (2005). Interprocedural analysis for privileged code placement and tainted variable detection. In *ECOOP*, pages 362--386.

Puppy, R. F. (1999). Perl cgi problems. `http://www.phrack.org/issues.php?issue=55&id=7#article`.

Quercus (2010). Quercus: PHP in Java. http://quercus.caucho.com.

Raven (2010). Roadsend PHP: Raven (rphp). http://code.roadsend.com/rphp.

Reps, T., Horwitz, S., and Sagiv, M. (1995). Precise interprocedural dataflow analysis via graph reachability. In *POPL*, pages 49--61. ACM.

Roadsend (2010). Roadsend PHP. http://www.roadsend.com.

Scholz, B., Zhang, C., and Cifuentes, C. (2008). User-input dependence analysis via graph reachability. Technical report, Sun Microsystems, Inc.

Schwartzbach, M. I. (2010). Lecture notes on static analysis. `http://www.brics.dk/~mis/static/`.

Singer, J. (2006). *Static Program Analysis Based on Virtual Register Renaming*. PhD thesis, University of Cambridge.

Snelting, G., Robschink, T., and Krinke, J. (2006). Efficient path conditions in dependence graphs for software safety analysis. *TOSEM*, 15(4):410--457.

Stephenson, M., Babb, J., and Amarasinghe, S. (2000). Bidwidth analysis with application to silicon compilation. In *PLDI*, pages 108--120. ACM.

Su, Z. and Wassermann, G. (2006). The essence of command injection attacks in web applications. In *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 372--382, New York, NY, USA. ACM.

Sundblad, Y. (1971). The ackermann function. a theoretical, computational, and formula manipulative study. *BIT Numerical Mathematics*, 11:107–119. 10.1007/BF01935330.

Tavares, A. L. C., Pereira, F. M. Q., Bigonha, M. A. S., and Bigonha, R. (2010). Efficient SSI conversion. In *SBLP*.

Tripp, O., Pistoia, M., Fink, S. J., Sridharan, M., and Weisman, O. (2009). TAJ: Effective taint analysis of web applications. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '09, pages 87--97, New York, NY, USA. ACM.

Vogt, P. (2006). *Cross Site Scripting (XSS) Attack Prevention with Dynamic Data Tainting on the Client Side*. PhD thesis, Technical University of Vienna.

Wassermann, G. and Su, Z. (2007). Sound and precise analysis of web applications for injection vulnerabilities. In *PLDI*, pages 32--41. ACM.

Wassermann, G. and Su, Z. (2008). Static detection of cross-site scripting vulnerabilities. In *ICSE*, pages 171--180. ACM.

Wegman, M. N. and Zadeck, F. K. (1991). Constant propagation with conditional branches. *TOPLAS*, 13(2).

Weiser, M. (1981). Program slicing. In *ICSE*, pages 439--449. IEEE.

Whaley, J. and Lam, M. S. (2004). Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *PLDI*, pages 131–144. ACM.

Xie, Y. and Aiken, A. (2006). Static detection of security vulnerabilities in scripting languages. In *USENIX-SS*. USENIX Association.

Zhang, X. and Wang, Z. (2010). A static analysis tool for detecting web application injection vulnerabilities for asp program. In *e-Business and Information System Security (EBISS), 2010 2nd International Conference on*, pages 1--5.

Appendix A

Tainted Flow Analysis

In this appendix we show how to write a pass to find a specific type of vulnerability desired and how to register the pass to be executed by our tainted flow analysis on `phc`.

A.1 Writing a Vulnerability Pass

In order to write a tainted flow pass, first we must define the set of sanitizers, validators and sinks for the type of security vulnerability we want to find. To write the pass we must inherit from `Tainted_problem` and provide implementation for the abstract function `initialize`. This function will be used to load the set of sanitizers, validators and sinks. Note that you do not need to specify the sources, since they are the same for every type of security vulnerability; hence shared by every tainted flow pass. We provide a set of common sanitizers and filters as well, that can be used by calling the functions `load_default_sanitizers` and `load_default_validators`. The sinks are bug specific and have no default values. We must define which sink parameters are sensitive so that our analysis can report bugs if taint values reach them. The default behavior of `insert_sink` is to consider only the first argument, unless the second argument is set to true, which has the effect of marking every sink argument as sensitive. Another option is to specify which sink arguments are sensitive through the `insert_sink_arg` function. We have implemented four tainted flow passes according to the discussion on Section 2.3: (i) Cross-Site Scripting (XSS); (ii) SQL Injection Attacks; (iii) Unwanted Command Execution; and (iv) Unauthorized Filesystem Access. The implementation of each one of them can be found in the attachments at the end of this dissertation (Attachments A through D).

A.2 Registering the Vulnerability Pass

We can write a tainted pass directly on the compiler codebase, or we could use `phc` ability to load passes on-the-fly and write the tainted pass as a plugin. Either way, we must register our pass so that the `phc` can identify and execute it. Below we show how to insert the pass into the `phc` pass manager.

```
// tainted attack passes
pm->add_tainted_analysis(new XSS_attack(), s("xss-attack"),
  s("Search for XSS attacks"));
pm->add_tainted_analysis(new SQL_injection(), s("sql-injection"),
  s("Search for SQL injections"));
pm->add_tainted_analysis(new Command_exec(), s("cmd-exec"),
  s("Search for command execution injections"));
pm->add_tainted_analysis(new Filesystem_access(), s("fs-access"),
  s("Search for unauthorized filesystem access"));
```

Appendix B

Security Advisories

In our evaluation (Chapter 5), we showed that our tool was able to find 36 previously unknown vulnerabilities in 5 commercial CMS products. In order to disclosure the bugs we found, first we contact the vendors with detailed information about the discovered flaw. Hence, vendors can acknowledge the bug and provide a fix for the problem. Later, we send a security advisory to specific security lists. In our case, we wrote and sent 5 advisories to the security focus bugtraq[1]. We list the original advisories below.

B.1 MODx 1.0.3

```
Title: MODx Instalation File XSS Vulnerability
Vendor: MODx
Product: MODx CMF
Tested Versions: 1.0.3, 1.0.4
Threat Class: XSS
Severity: Medium
Remote: yes
Local: no
Discovered By: Andrei Rimsa Alvares

===== Description =====

MODx CMF is prone to a XSS vulnerability caused by unsanitized user
input data. The bug occurs in a file used in the installation process.
A description of the affected file is shown below:

--- install/connection.collation.php ----
01: <?php
    ...
06: $database_collation = $_POST['database_collation'];
    ...
```

[1] http://www.securityfocus.com/archive/1

```
08: $output = '<select id="database_collation" name="database_collation">
09: <option value="'.$database_collation.'" selected >'.$database_collation.
'</option></select>';
    ...
23: echo $output;
24: ?>
--- install/connection.collation.php ----
```

The variable $database_collaction (line 6) receives user data via
http post request and gets propagated to variable $output (line 9)
without proper sanitization. Later the $output variable is outputted
to the page in every program path causing the bug (line 23).

```
===== Impact =====
```

Malicious java script code can be executed in the context of the
affected web site.

```
===== Proof of Concept =====

<form action="http://target/install/connection.collation.php"
  name="evil" method="post">
  <input type="hidden" name="database_collation" value="</option></select>
<script>window.alert(String.fromCharCode(88,83,83));</script>" />
</form>
<script>
document.evil.submit();
</script>

===== Workaround =====
```

Remove all installation files after MODx is successfuly installed.

```
===== Disclosure Timeline =====
```

June, 16 2010 - Vendor notification.
July, 06 2010 - No vendor reply. Public disclosure.

```
===== References =====

http://modxcms.com
```

B.2 Exponent CMS 0.97

```
Title: Exponent Slideshow XSS Vulnerability
```

```
Vendor: Exponent
Product: Exponent CMS
Tested Version: 0.97.0
Threat Class: XSS
Severity: High
Remote: yes
Local: no
Discovered By: Andrei Rimsa Alvares

===== Description =====

The file "modules/slideshowmodule/slideshow.js.php" is prone to
a XSS vulnerability. Multiple instance of variable $_GET['u'] gets
outputted to the page without proper sanitization.

===== Impact =====

Malicious java script code can be executed in the context of the
affected web site.

===== Proof of Concept =====

http://target/modules/slideshowmodule/slideshow.js.php?
u=%3Cscript%3Ewindow.alert(String.fromCharCode(88,83,83));%3C/script%3E

===== Workaround =====

No workaround is available at the time.

===== Disclosure Timeline =====

June, 16 2010 - Vendor notification.
July, 06 2010 - No vendor reply. Public disclosure.

===== References =====

http://www.exponentcms.org
```

B.3 DCP Portal 7.0beta

```
Title: DCP-Portal Multiple XSS Vulnerabilities
Vendor: Worxware
Product: DCP-Portal
Tested Version: 7.0beta
Threat Class: XSS
```

```
Severity: High
Remote: yes
Local: no
Discovered By: Andrei Rimsa Alvares

===== Description =====

Multiple XSS vulnerabilities encountered in the DCP-Portal.

1. common/components/editor/insert_image.php,
modules/newsletter/insert_image.php, php/editor.php
  The variable $upload_failure_report gets user input from http
get request variable "Image" when the action of deleting an
uploaded file fails. Later this variable is outputted to the
page without proper sanitization.

2. modules/gallery/view_img.php
  Page title can be modified by changing the http request variable
"imgtitle". Since no sanitizer is used, an XSS occurs on line 2.
  Another vulnerability exists if magic quotes is turned off.
The http request variable "imagename" gets outputted on the java
script function document.write between simple quotes on line 27.

3. modules/tips/show_tip.php
  Http request variable "newsId" gets outputted to the page without
proper sanitization on line 14.

===== Impact =====

Malicious java script code can be executed in the context of the
affected web site.

===== Proof of Concept =====

All proof of concepts display a java script alert containing the
message "XSS".

1. common/components/editor/insert_image.php,
modules/newsletter/insert_image.php, php/editor.php
  http://target/common/components/editor/insert_image.php?MyAction=Delete&
Image=%3Cscript%3Ewindow.alert(String.fromCharCode(88,83,83));%3C/script%3E
  http://target/modules/newsletter/insert_image.php?MyAction=Delete&
Image=%3Cscript%3Ewindow.alert(String.fromCharCode(88,83,83));%3C/script%3E
  http://target/php/editor.php?MyAction=Delete&
Image=%3Cscript%3Ewindow.alert(String.fromCharCode(88,83,83));%3C/script%3E
```

```
2. modules/gallery/view_img.php
  http://target/modules/gallery/view_img.php?imgtitle=
%3C/title%3E%3Cscript%3Ewindow.alert(String.fromCharCode(88,83,83));%3C/script%3E
  (requires magic_quotes_gpc = off) http://target/modules/gallery/view_img.php?
imagename=%22');window.alert('XSS');document.write('%22

3. modules/tips/show_tip.php
  http://target/modules/tips/show_tip.php?
newsId=%3Cscript%3Ewindow.alert(String.fromCharCode(88,83,83));%3C/script%3E

===== Workaround =====

No workaround is available at the time.

===== Disclosure Timeline =====

June, 16 2010 - Vendor notification.
July, 06 2010 - No vendo reply. Public disclosure.

===== References =====

http://www.dcp-portal.org
http://www.worxware.com
```

B.4 Pligg 1.0.4

```
Title: Pligg Instalation File XSS Vulnerability
Vendor: Pligg
Product: Pligg CMS
Tested Version: 1.0.4
Threat Class: XSS
Severity: Medium
Remote: yes
Local: no
Discovered By: Andrei Rimsa Alvares

===== Description =====

Pligg is prone to a XSS vulnerability in the installation file:
install/install1.php. The variable "language" - obtained from a
http request - can be manipulated to execute java script code via
onmouseover like functions. Even with the two sanitizers used
(strip_tags and addslashes) is possible to bypass the double quote
jail of the value field in the input tag by passing a double quote
```

```
via the "language" variable.

----- install/install1.php -----
20:  <input type="hidden" name="language"
value="<?php echo addslashes(strip_tags($_REQUEST['language'])); ?>">
----- install/install1.php -----

The sanitizer strip_tags prevents new tags to be used (like <script>
and </script>) but it can still be abused to pass onmouseover style
attacks. Addslashes adds backslashes to escape special characters like
double quote, but since html does not process escape sequences, this
sanitizer is useless to prevent breaking the double quote jail.

===== Impact =====

Malicious java script code can be executed in the context of the
affected web site.

===== Proof of Concept =====

Attack vector extracted from [1]. This attack attempts to increase the
area of the affected input field to cover the whole screen. Once the
mouse is moved anywhere on the screen, the onmouseover java script can
be triggered to execute the malicious code. In this proof of concept,
an alert containing the message "XSS" should be shown on the screen once
the mouse is moved on the screen.

http://target/install/install1.php?language=%22%20style=a:b;
margin-top:-1000px;margin-left:-100px;width:4000px;height:4000px;
display:block;%20onmouseover=alert%28String.fromCharCode%2888,83,83%29%29;%3E

This attack venue exploited in this proof of concept had no effect on
Google Chrome web browser, because the input field is hidden. But can
be exploited on Mozilla Firefox and possibly others.

===== Workaround =====

Remove the instalation directory after installation, as recommended
during installation.

===== Disclosure Timeline =====

June, 16 2010 - Vendor notification.
June, 22 2010 - Vendor replied but did not acknowledging the bug.
June, 22 2010 - New contact attempted to further explainining the bug.
```

```
July, 06 2010 - No vendor reply. Public disclosure.

===== References =====

1. http://www.packetstormsecurity.org/papers/bypass/workaround-xss.txt
2. http://www.pligg.com
```

B.5 RunCMS 2.1

```
Title: RunCMS XSS Vulnerability via User Agent
Vendor: RunCMS
Product: RunCMS
Tested Version: 2.1
Threat Class: XSS
Severity: Medium
Remote: yes
Local: no
Discovered By: Andrei Rimsa Alvares

===== Description =====

RunCMS is proned to a XSS vulnerability by mangling the user-agent
field on a http request to a script within the forum module.

----- modules/forum/check.php -----
01: <?php
    ...
10:           echo "BROWSER: ".$_SERVER['HTTP_USER_AGENT'];
----- modules/forum/check.php -----

===== Impact =====

Malicious java script code can be executed in the context of the
affected web site.

===== Proof of Concept =====

wget --user-agent="<script>window.alert('XSS');</script>"
  http://target/modules/forum/check.php

===== Workaround =====

Remove the affected file form the system: modules/forum/check.php.

===== Disclosure Timeline =====
```

```
June, 16 2010 - Vendor notification.
June, 17 2010 - Vendor response.
July, 06 2010 - Public disclosure.

===== References =====
http://www.runcms.org
```

Attachment A

Cross-Site Scripting (XSS)

A.1 XSS_attack.h

```
#ifndef PHC_XSS_ATTACK
#define PHC_XSS_ATTACK

#include "tainted/Tainted_problem.h"

class CFG;

class XSS_attack : public Tainted_problem {
public:
  XSS_attack();
  void initialize();
};

#endif // PHC_XSS_ATTACK
```

A.2 XSS_attack.cpp

```
#include "XSS_attack.h"

XSS_attack::XSS_attack() : Tainted_problem("XSS Attacks") {
}

void XSS_attack::initialize() {
  // Sanitizers.
  load_default_sanitizers();
  insert_sanitizer("htmlentities");
  insert_sanitizer("htmlspecialchars");
  insert_sanitizer("strip_tags");
  insert_sanitizer("highlight_string");
```

```
  // Validators.
  load_default_validators();

  // Sinks.
  insert_sink("print");
  insert_sink("printf", true);
}
```

Attachment B

SQL Injection Attacks

B.1 SQL_injection.h

```
#ifndef PHC_SQL_INJECTION
#define PHC_SQL_INJECTION

#include "tainted/Tainted_problem.h"

class CFG;

class SQL_injection : public Tainted_problem {
public:
  SQL_injection();
  void initialize();
};

#endif // PHC_SQL_INJECTION
```

B.2 SQL_injection.cpp

```
#include "SQL_injection.h"

SQL_injection::SQL_injection() : Tainted_problem("SQL Injection") {
}

void SQL_injection::initialize() {
  // Sanitizers.
  load_default_sanitizers();
  insert_sanitizer("addslashes");
  insert_sanitizer("mysql_escape_string"); // deprecated.
  insert_sanitizer("mysql_real_escape_string");
  insert_sanitizer("pg_escape_string");
```

```
  // Validators.
  load_default_validators();

  // Sinks.
  insert_sink("mysql_query", true);
  insert_sink("pg_query", true);
}
```

Attachment C

Unwanted Command Execution

C.1 Command_exec.h

```
#ifndef PHC_CMD_EXEC
#define PHC_CMS_EXEC

#include "tainted/Tainted_problem.h"

class CFG;

class Command_exec : public Tainted_problem {
public:
  Command_exec();
  void initialize();
};

#endif // PHC_CMD_EXEC
```

C.2 Command_exec.cpp

```
#include "Command_exec.h"

Command_exec::Command_exec()
  : Tainted_problem("Unwanted Command Execution") {
}

void Command_exec::initialize() {
  // Sanitizers.
  load_default_sanitizers();
  insert_sanitizer("escapeshellarg");
  insert_sanitizer("escapeshellcmd");
```

```
  // Validators.
  load_default_validators();

  // Sinks.
  insert_sink("exec");
  insert_sink("system");
  insert_sink("passthru");
  insert_sink("shell_exec");
  insert_sink("proc_open");
  insert_sink("pcntl_exec");
}
```

Attachment D

Unauthorized Filesystem Access

D.1 Filesystem_access.h

```
#ifndef PHC_FS_ACCESS
#define PHC_FS_ACCESS

#include "tainted/Tainted_problem.h"

class CFG;

class Filesystem_access : public Tainted_problem {
public:
  Filesystem_access();
  void initialize();
};

#endif // PHC_FS_ACCESS
```

D.2 Filesystem_access.cpp

```
#include "Filesystem_access.h"

Filesystem_access::Filesystem_access()
  : Tainted_problem("Unauthorized Filesystem Access") {
}

void Filesystem_access::initialize() {
  // Sanitizers.
  load_default_sanitizers();

  // Validators.
  load_default_validators();
```

```
  // Sinks.
  insert_sink("chdir");
  insert_sink("mkdir");
  insert_sink("rmdir");
  insert_sink("rename");
  insert_sink("unlink");
  insert_sink("copy");
  insert_sink("chgrp");
  insert_sink("chown");
  insert_sink("chmod");
  insert_sink("touch");
  insert_sink("symlink");
  insert_sink("link");
  insert_sink("move_uploaded_file");
  insert_sink("show_source");
  insert_sink("highlight_file");
  insert_sink("readfile");
  insert_sink("file_get_contents");
}
```