Inferring Specifications for Web Application Security

by

Maliheh Monshizadeh
B.S., Shahid Beheshti University, 2008
M.S. Sharif University of Technology, 2011

Thesis submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Chicago, 2017

Chicago, Illinois

Defense Committee:
V. N. Venkatakrishnan, Chair and Advisor
Prasad A. Sistla
Lenore Zuck
Mark Grechanik
Prasad Naldurg, IBM Research India

To my Parents.

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

CHAPTER **PAGE**

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

AIV	Application Inconsistency Vulnerability
CFG	Control Flow Graph
CMS	Content Management System
DB	DataBase
DBMS	DataBase Management System
DDG	Data Dependence Graph
DAC	Discretionary Access Control
DIFT	Dynamic Information Flow Tracking
EAR	Execution after Redirect
HPE	Horizontal Privilege Escalation
LVA	Live Variable Analysis
PHP	Hypertext Preprocessor scripting language
SDG	System Dependence Graph
SQLI	SQL Injection attack
VM	Virtual Machine
VPE	Vertical Privilege Escalation
XSS	Cross-Site Scripting

SUMMARY

Over the past two decades, we have been witnessing the evolution of the web applications from simple static pages into complex, interactive platforms. With increasing demand to have more features added to the applications, we also have observed an increase in the frequency and significance of data breaches due to web application vulnerabilities. The need to secure the applications, however, has not been met promptly. The current practice of web application development does not address security concerns even against known vulnerabilities, let alone new unknown attacks.

The goal of the thesis is to improve the security of web applications. To achieve this goal, we would like to detect, and retrofit vulnerabilities. In studying the cyber threat landscape, we observed common web development practices and mistakes, which cause security flaws in design and implementation of web applications. By examining the existing security analysis tools, we identify their capabilities and their limitations. Most of these tools require some program specifications to be available to generate sound reports. However, specifications are often missing in web applications due to market demands for fast releases.

The lack of program specification in web applications makes it challenging to analyze and verify web applications. In the absence of program specifications, the only source of information about the web developer's design intentions with respect to security policies in the application source code. While this source code obscures the high-level logic of the application among so many low-level details, there still are some development patterns available to us to infer the intention of the developers. Based on this belief, it is very much possible to infer program specifications from low-level artifacts and leverage

them in order to detect and retrofit vulnerabilities in legacy applications. We are also able to use this knowledge to build newer development frameworks for automated synthesis of secure code.

This thesis develops techniques to infer security specifications from the web application source. As a result of using the inferred specifications, we can improve the security of the applications in numerous ways. First, we are able to examine the inferred authentication and authorization policies to find authorization inconsistencies. Such inconsistencies are the main source of privilege escalation vulnerabilities in web applications. To present the effectiveness of our approach, we evaluated it on various web applications. The results suggest that we are able to detect previously unknown vulnerabilities by precise inference of access control policies.

Secondly, we are able to generate security patches for the reported vulnerabilities in web applications. Traditionally, the applications were being patched manually due to the poor quality of the automated generated patches. Using specification inference techniques, we can generate correct security patches for the vulnerable applications and suggest suitable placement of these patches in complex applications, reducing the effort of developers and security analysts.

Lastly, we examine how inferred security specification can be used for synthesis of secure code in web development frameworks. We believe that by automated synthesis of security policies, we reduce the possibility of redundancy and human-error.

Our results in each of the areas mentioned above show that inferring security specifications from the application source code is not only possible but also practical and scalable.

CHAPTER 1

INTRODUCTION

Since the emerge of the World Wide Web in the early 90s, it had a tremendous effect on how people live, work and interact with each other. There are over 1 billion websites on the Web today including websites which deal with our economy, healthcare and education systems, and around 40% of the world population (3.5 billion people) has an Internet connection today. (1)

A website is typically a set of web pages, usually powered by *web applications* hosted on one or more web servers. Web applications allow a website owner (publisher) to provide dynamic content and interactive user experience. A web application is a client/server application in which a web browser is used as the client. At the client side, users retrieve data and interact with webpage components through the web browser.

Figure 1 shows a typical web application design in three layers. The layered architecture of web applications allows the developers and users to be able to communicate with a broad range of technologies and frameworks. While the web servers can manage the static content (such as static HTML files and images), the underlying application servers use more dynamic technologies (e.g., PHP and JSP) to create more flexible, scalable web applications. With dynamic web applications, the content can be updated and customized much faster than in static web applications.

While the data is imported to the storage layer, the application layer manages to retrieve and store the data. The workflow of the application, written in a dynamic scripting language such as PHP, reflects the high-level logic of the interactions between application entities. Although this data-centric view of

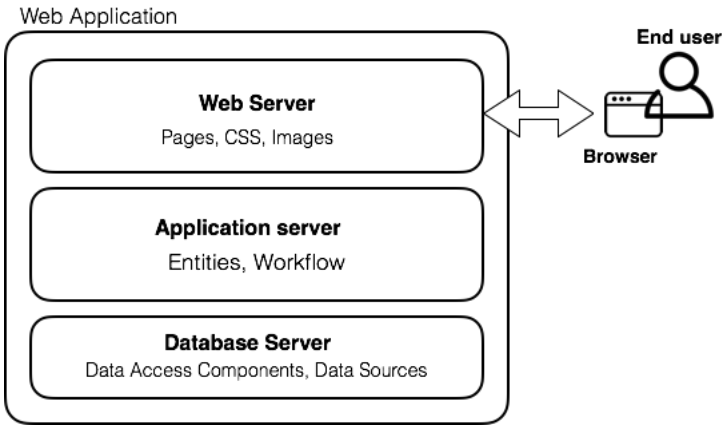

Figure 1: Typical Web Application Architecture

the web applications leads to more interactive and attractive websites, the dynamic web makes it much more difficult to pin down the defect logic in the applications.

The complexity in the applications, along with their dominance in the market share lure hackers to exploit these technologies. More than half of all breaches involve web applications in organizations and about 89% of the breaches had a financial or espionage motive. (2) This year, Yahoo confirmed that user data associated with at least 500 million user accounts have been stolen, including names, email addresses, telephone numbers, dates of birth, hashed passwords, and encrypted or unencrypted security questions and answers. (3)

Although traditional applications are also prone to insider and outsider attacks, there are some properties associated with web applications which make them particularly unique in the area of computer security. (4) The reasons as to why web application security concerns stand out in software security is twofold:

Rapid Development of Web Applications Over the past decade, it has become easier and simpler for users to generate content and customize websites for their purposes. Newer web application platforms facilitate the development and maintenance of new web applications. The market demands make web developers want to release their products faster, using rapid building and frequent update deliveries. Although there has been a tremendous effort toward security education, most web developers are still unaware of improper programming habits which can lead to severe attacks.

Stateless Web Protocols For performance reasons, the web protocols have been developed to be stateless, meaning that the web server and the underlying operating system do not keep any connection information and each request is treated independently.

However, most distributed applications, including web applications, are stateful. Therefore, the burden of design and implementation of a stateful application is entirely on the web application developers. In such design, both the client and the server should be able to share a common state in each session. Each request also should have sufficient information about the session and the user in it so that the server can distinguish the users and sessions. If the web developers do not design session management thoroughly and based on guidelines (5), their applications can be vulnerable to numerous attacks, including session hijacking and session fixation attacks.

These two factors, rapid cycles of development along with de facto standards used in web protocols, lead to defects in the design and implementation of web applications. This, in turn, lures attackers to exploit various vulnerabilities. To prevent such attacks, session management libraries have been developed for various platforms, which are used to develop new web applications. For vulnerable legacy

web applications, however, fundamental security flaws often require a re-architecture, which can be quite expensive for website owners.

Alternatively, there are other measures cyber security teams can take to protect flawed applications. Prevention and retrofitting efforts are among the security measures one can take to safeguard vulnerable legacy applications. To be able to prevent attacks and retrofit the vulnerable applications, we first need to identify the vulnerabilities. As the applications become larger and more sophisticated, manual detection of the vulnerabilities become more time-consuming and tedious. The complexity and the size of web applications direct the security analysts to find automated or semi-automated approaches to speed up the detection procedure as well as to make the analysis process more precise.

1.1 Automated Security Analysis of Web Applications

To prevent security attacks in web applications, developers and website publishers (publisher in short) need to actively maintain their websites. This maintenance includes testing, regularly updating and generating patches for security vulnerabilities. A developer or a publisher needs a deep understanding of the design of the application and the security concepts, to be able to perform this security maintenance. However, a web developer without proper security knowledge may eventually re-use custom libraries or code snippets for fast development.

Although manual inspection of the code by a security expert can lead to precise results, the growing size of web application code base due to added new features to websites has made the manual inspection challenging. The complexity of application - usually expressed in the number of possible execution paths - grows exponentially with the number of conditions in the application code. It has become a

tedious - if not impossible - task to inspect medium- to large-sized applications without using automated analysis tools.

Program analysis tools facilitate the detection of security bugs. These tools usually inspect the code against the design artifacts (e.g., class diagrams, interface models) and detect implementation vulnerabilities. The documentations about the expected behavior of the program are called program specifications. These specifications are usually developed in design stages and before the implementation cycles.

However, the rapid development of web applications leads to the release of these programs with almost no specification documentations. Without program specifications, it is challenging to reason about the expected functionality of the application versus its deviant behavior which may result in data breaches. Most of the security analysis tools are not fully automated and they use some annotations provided by the analysts. But even in these cases, analysis of medium to large applications may lead to the need for a considerable amount of annotation efforts. Therefore, automated analysis of applications with minimal knowledge about the program is desirable for both developers and security analysts.

Using the inferred security policies from the applications, we are able to reason about the overall security of the application and its users. Not only we can detect the vulnerabilities due to implementation bugs and faulty policies, but also we would be able to synthesize code to secure the application.

1.2 Thesis Goals

In the web applications context, vulnerabilities may manifest in one of two categories: 1) as an error in a computation (data dependent) or 2) as an error in program control, i.e., a vulnerability caused by a missing or an incorrect control check in the program. Web applications often do not come with

correctness specifications – properties that attest behavior at various program points. The lack of such program specifications makes it very difficult to identify original functional requirements and validate their correctness. Therefore, fixing the first category, i.e., computation errors is challenging, as the true intention of the computation is unknown to the analysis tools. However, the second category of logic vulnerabilities is relatively easier to generate patches for, given that this subset of logic vulnerabilities are detectable through inconsistency analyses on applications.

We term this set of vulnerabilities as application inconsistency vulnerabilities (AIVs), the type of logic vulnerability that arises from inconsistent design and implementation of security checks in an application. We believe that despite the challenges involved in the automatic analysis of web applications, there are still some clues in the source code which can help us to detect security vulnerabilities and synthesize secure patches for them. Software patterns and language-based constructs used in programs, designed schemas and interfaces are examples of such clues which can be used to build abstract models of the programs. It is important to take advantage of the available artifacts, such as the source code, the database schemas, as much as possible in order to build more precise models. We are especially interested in the inference of specifications for legacy applications which are already ubiquitously in use.

Furthermore, we believe that although the analysis tools are unaware of the true intention of the developers, they can still build proper abstract models of the security policies to detect inconsistent policies implemented in the source code and retrofit them. Although these inconsistencies may be intentional, most of them lead to actual vulnerabilities in the code which can have severe impacts on the security of the whole application and its users. Existing analysis tools for AIVs can not only detect that

something is missing or is incorrect, but also infer *what exactly* is missing and suggest ways to correct it.

We categorize the top web application vulnerabilities and reason about their common causes. For each of these categories, we define the vulnerability at a high level and then build an abstract model of the vulnerabilities and the program. We believe that this approach helps us in a thorough understanding of the vulnerabilities which lead to more effective detection methods and secure code synthesis. We also believe that this categorization based on the common causes of vulnerabilities can help us in detecting newer vulnerabilities. It also helps us in focusing on fixing the vulnerabilities with common causes for security holes rather than fixing them individually.

Chapter 2 provides a detailed background of the vulnerabilities that can be detected through automated analysis. We discuss the abstraction interpretations and the challenges involved in building abstract models about the programs. We also go over the challenges and the limitations of automated tools.

In Chapter 3 we present MACE, an automated tool for detection of authorization vulnerabilities in web applications. MACE detects these vulnerabilities with minimum information about the application's authorization policies. MACE uses static code analysis to build authorization models for each application resource (such as files, and database tables). It then checks each resource access in the code against the built models to detect vulnerabilities. MACE is the first tool reported in the literature to identify a new class of web application vulnerabilities called Horizontal Privilege Escalation vulnerabilities. MACE works on large codebases and discovers serious, previously unknown vulnerabilities

Chapter 4 presents LOGICPATCHER, an automated tool for retrofitting logic vulnerabilities in web applications. Given a vulnerability description, LOGICPATCHER generates candidate security patches for a web application. LOGICPATCHER focuses on logic vulnerabilities due to inconsistent security checks in programs and works across a broad variety of application types, including e-commerce servers, news servers, wikis etc. By using path profiling, LOGICPATCHER emphasizes on correct patch placement, i.e., identifying the precise location in the code where the patch code can be introduced without interfering with existing functionalities.

In Chapter 5 we take a step further and discuss our research on the synthesis of secure code in web applications. Our tool WAVES generates client-side input validation code for PHP applications. Not only our tool reduces the possibility of inconsistency vulnerabilities due to human error, it also saves developers' time and energy on software upgrades. We believe that this new paradigm of secure code generation is a scalable new solution to securing under-develop web applications as well as legacy applications.

In Chapter 6 we discuss the related work and compare our research to similar projects. We conclude our work in Chapter 7. Technical details related to our security mechanisms are provided in Appendices.

2

BACKGROUND

Parts of this chapter have been published as Maliheh Monshizadeh, Prasad Naldurg, V. N. Venkatakrishnan. MACE - Detecting Privilege Escalation Vulnerabilities in Web Applications. In Proceedings of 21st ACM Conference on Computer and Communications Security (CCS'14), Scottsdale, AZ, 2014.

The 2011 CWE/SANS (6) ranked the 25 most dangerous software errors. Vulnerabilities in Web were and have remained among the top 10 vulnerabilities identified in this list. In order to address these vulnerabilities, we first need to understand them and identify why they happen. We also need to examine the associated risk with their exploitation and educate the web developers about this risk.

Among various type of vulnerabilities, we set our focus to the top vulnerabilities which occur at the server-side and in the web application source code; rather than vulnerabilities in the back-end databases or client-side or browser vulnerabilities.

In Section 2.1 we discuss these vulnerabilities and the reason for their ubiquity. We then categorize these vulnerabilities based on the amount of program specifications needed in order to detect and fix them. Each subsection explains the vulnerability in more detail along with the underlying programming challenges that may lead to the vulnerability. In Section 2.2 we will discuss different approaches to

inferring program specifications. We explain our choice to use static analysis and abstraction as the high-level approach.

2.1 Web Application Vulnerabilities

Due to Web's open nature and wide deployment, they make appealing targets to criminals who want to gain access to users' data and resources. No one on the Internet is immune from the threat of security breaches. Security of web applications therefore has become an important concern.

However, the Web protocols and many legacy applications have been deployed and widely used before the web security becomes a crucial must. This leads to fundamental design defects in the web and puts the burden on web developers to protect their application and its users from security threats.

Stateless Protocol For performance reasons, the HTTP protocol is designed to be stateless, meaning that each request is processed independently by the web server. Therefore, the underlying operating system and the web server do not establish any connection for each request and they don not hold any state related to them.

However, many of the transactions (e.g., shopping transactions) in web applications are stateful. Hence, the task of keeping the history and the sequence of this connection, the state, is on the web applications and each request should contain enough state information on its own in order to be processed. Web applications use cookies, sessions, special URLs, and hidden field values to keep the state information.

Untrusted Input Similar to all application which deal with data entering the system, web applications need to validate the data before performing any operation on it. Proper input validation minimizes the risk of many attacks as well as application malfunction.

Input validation becomes extremely crucial when it affects the application's security decisions or its internal state. For instance, session management decisions based on cookie and hidden field values should be performed carefully. Both of these values come from the client-side and as far as the server-side is concerned, no data from the server-side should be trusted. Poor session management leads to severe attacks which we will discuss in the following sections.

2.1.1 Injection Vulnerabilities

An injection attack happens when input data has not been validated properly. In an injection attack, the attacker will provide some form of input and attach additional malicious data to perform some other or additional command. By especially crafting the input, the attacker is able to break the confidentiality and integrity of resources, gain control over the application, or abrupt the application service.

SQL Injection This attack is ranked first in the top 25 most dangerous vulnerabilities. In this attack, the malicious user manipulates the input data in order to craft new SQL queries and affect their execution. Based on the type of query execution, this attack may lead to breaches in confidentiality and data integrity as well as authorization.

The reason for this attack is that the data entering the application is not validated properly before using them in sensitive database operations. To avoid SQLI, developers use prepared statements, as well as standard sanitization functions to validate the data before sensitive operations.

Cross-Site Scripting (XSS) Cross-site scripting occurs when the attacker can successfully inject malicious scripts into a webpage, in order to read users' sensitive information or perform malicious activity on behalf of the victim webpage on the users.

XSS is ranked fourth in the top 25 vulnerabilities and is caused by improper validation of the user input which then is going to be reflected in the webpages. By injecting malicious data and scripts, the attacker is able to take control of how to attacker the users of the victim webpage.

Proper sanitization of the data can prevent these attacks. There are well-established, standard libraries developed in each web development kit to validate the data against XSS attacks.

2.1.2 Logic Vulnerabilities

Logic vulnerabilities are a special category of security vulnerabilities that cause a program to operate incorrectly or exhibit unexpected behavior. In contrast with *injection* vulnerabilities (such as Cross-Site Scripting, or SQL Injection) which are generic type of vulnerabilities, logic vulnerabilities are *application-specific* and caused by faulty logic in the application. It means that the generic sanitization procedures and libraries cannot be used arbitrarily to fix these vulnerabilities. We will explain this issue in more details in Chapters 3 and 4.

Parameter Tampering Vulnerabilities Lack of input validation in web applications cause a type of logic exploits called Parameter tampering attacks. These type of attacks are based on crafting the input to the server in order to modify the application data and usual control flow of the application. Modifying cookies, query strings, quantities and hidden values in web forms are examples of such attacks.

The following line of HTML code shows that the developer used a hidden HTML field to store the cost of the item in the shopping cart:

```
1 <input type="hidden" id="1008" name="cost" value="70.00">
```

Although the field is hidden from in rendered page, an attacker can tamper this value and submit it back to the server. A vulnerable application relies on this tampered value provided from the untrusted client-side to perform sensitive computations on the shopping cart cost amount.

Not only the web developers should validate the client-side data thoroughly, but also they should not trust the state information that comes from the client-side. In this example, the `cost` value should not be sent back to the server, as the server-side already has the cost information.

Authorization Vulnerabilities (Privilege Escalation) The problem of privilege escalation in web applications has roots in the way a web application is being authenticated/authorized to the application Database Management System (DBMS). In a typical web server, the whole web application is authenticated through a single pair of credentials, giving the application the maximum set of privileges (administrative privileges). In addition, this authenticated session is usually persistent.

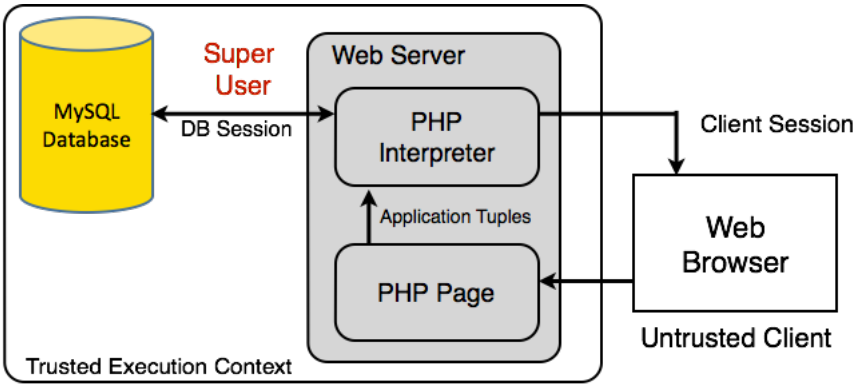

Figure 2: Active Sessions in Web Applications

To protect from these threats, web applications implement *access control* (a.k.a. authorization) policies. A typical authorization check in a web application involves verifying whether a given authenticated user with an associated functional *role* has the required *privilege* to access a given resource such as a database table. Since authorization is expected to be performed before every resource access, it therefore forms the basis for security of the web application.

However, web developers often fail to use proper techniques to minimize the privileges of the users. There are several reasons why such authorization errors are numerous. First, unlike conventional operating systems, web applications (such as those written using PHP) do not come with built-in support for access control. The access control policy is often coded by a developer into the application. Developers often focus on other key functionalities of the applications, and often make errors in programming authorization code, as illustrated by the 2011 CWE / SANS report (6), in which missing authorization and improper authorization are ranked 6th and 15th in the top 25 most dangerous software errors. Second, a web application (such as one written using PHP and SQL) often connects directly to the database resource as a superuser who enjoys all administrative privileges on the database, and any flaws in the authorization logic often lead to catastrophic data breaches. Further, web application developers often implement roles (7) as a privilege management solution. However, the unavailability of a standard framework, and the lack of developer's knowledge of access control design, have led to buggy and inconsistent role implementations in applications (8).

Several high-profile data breaches were caused due to the privilege escalation attacks in web applications. A most notable one is the Citibank data breach (9), wherein more than 360k credit card numbers were stolen. Such breaches suggest that web application authorization errors could be disastrous for

organizations. Furthermore, such vulnerabilities appear to be widespread, as a recent Cenzic technical report (10) listed that authorization vulnerabilities occurred in 56% of the applications that were tested in the 2013 study.

There are two types of privilege escalation: 1) Vertical Privilege Escalation (VPE), in which users can gain access to higher level role privileges, and 2) Horizontal Privilege Escalation (HPE) which happens when the user has the same level of privileges, but she can gain access to resources of users within the same group. In a banking website, a malicious customer may be able to forge requests and access other customers accounts, which is a type of HPE.

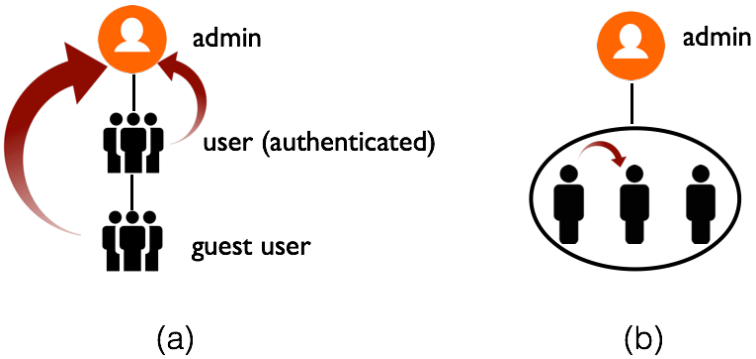

Figure 3: Privilege Escalation Types

The academic and industrial communities have identified several solutions to the problem. Virtual private databases (11) provide a way for applications to execute queries on behalf of users, and provide effective privilege separation. Web application frameworks such as Rails (12) provide software

engineering solutions to structure the access control logic of an application effectively. Despite these advances, a vast majority of web applications continue to be developed in languages such as ASP, Java and PHP where the onus of developing and enforcing the access control policy largely falls on developers. In Chapter 3, we will discuss privilege escalation vulnerabilities in more detail and discuss how we detect these vulnerabilities in legacy web applications.

2.2 Program Specification Inference

The problem of identifying whether an existing web application contains vulnerabilities is often exacerbated by lack of program documentations. It is indeed possible for a vulnerability analyst to look for errors by understanding the functionality of the application, inspecting the source for missing security checks and measures. However, manual inspection of the application source code in detail can be time-consuming and tedious for large web applications. Therefore, automated solutions that identify errors are desirable.

Such solutions usually need some input about the application's logic. Unfortunately, most of the open source web applications come with almost no documentation (except their source code) of their functionality. Security policies implemented, as a component of the overall functionality of the application is therefore obscure to the analysis tools. Thus, use of analysis tools which infer partial program specifications is a must.

Program specifications are useful, since they are the only detailed documentations beside the source code. While the application source code sheds light on the implementation decisions, the program specification defines the design of the application. Program specification is the description of what the program is expected to do, and is more high-level than the source code.

A program specification is often defined in terms of program invariants which are some properties which hold at particular program points. These properties are usually defined as relations between program data structures.

There are three approaches to defining the specifications.

1. Annotation: Developers often use annotation tools (e.g., Houdini (13) for ESC/Java applications) or instructions while developing their code, to explicitly define the program invariants. For instance, java developers can use `assert` instructions to define pre-conditions and post-conditions for verifying the correctness of different operations. The following code asserts that by incrementing the lower bound also increases:

```
assert x > 0;
x++;
assert x > 1;
```

2. Dynamic Detection of Invariants: In this method, the program gets executed over a set of sample inputs and the execution traces are being recorded. By statistically analyzing the values at each program location, one can infer the likely program invariants from the execution traces.

   In Daikon (14), program invariants are detected from program executions by instrumenting the source code to trace the variables of interest, running the program over a set of test cases and inferring invariants over both the instrumented variables and over derived variables that are not manifested in the original program. The key idea is that those invariants that are tested to a significant degree without falsification should be reported with more confidence.

Although this approach is very helpful in finding the likely invariants, it is not complete. To be confident of the results of such approaches, we should be able to reason about the coverage of the inputs, and the we have executed the program for enough times. That is why these likely program invariants need to be verified to be actual invariants.

3. Static Analysis: In this approach we analyze the program without actually executing the code. Using techniques such as symbolic execution, we are able to reason about almost all possible executions of the program. The key idea in using static analysis is to generalize the testing cases by using symbolic formulas rather than actual values.

Rather than designing new tools for each vulnerability type, we like to be able to detect several classes of vulnerabilities, including new vulnerabilities (e.g., privilege escalation), with the same techniques. Static analysis techniques can lead us to detection of general classes of vulnerabilities as they can detect underlying cause of vulnerabilities in the source-code.

In this dissertation we focus on static analysis of web applications as we are interested in detection and retrofitting of vulnerabilities by inspecting the underlying causes for vulnerabilities. In the remaining part of this chapter, we will discuss static analysis and abstraction techniques in more detail.

2.2.1 Static Analysis and Predicate Abstraction

Static analysis techniques try to reason about the specific properties of the program without actually running the program. To reason about the properties of each program, we have to reason about the *semantics* of the program first. According to Cousot & Cousot: (15)

> The (concrete) semantics of a program P is a computation (execution) model describing the effective executions $[[P]]$ of the program in all possible environments.

But the challenge is that in practice, $[[P]]$ is infinite or large enough not to be computable (16). It is not possible to write a program able to represent and to compute all possible executions of any program in all its possible execution environments. According to Rice's theorem, all non-trivial properties of programs written in common programming languages are mathematically undecidable. (17)

Alternatively, we have the option of building abstract models of possible execution traces, by ignoring some details in concrete semantics. Since we are ignoring some details and use generalization, the abstract models are supersets of the concrete semantics.

Abstract Interpretation formalizes the idea that a semantics can be more or less precise according to the considered observation level. The challenge then becomes to find a suitable abstraction model of our program semantics. In other words, we want to extract semantics of the code, but the extent to which we need these detailed semantics and how we model it are important questions. A good abstraction avoids state space explosion problem and leads to more efficient models in practice.

For an abstract semantics model to be acceptable it should have three properties: 1) soundness so that no possible error can be forgotten (false negatives); 2) precision (to avoid false positives); and 3) scalability (to avoid combinatorial explosion). However, in practice these properties may not be achievable all at once. We have a trade-off between scalability and precision while choosing an acceptable abstraction. If the approximation is coarse, the abstraction model of the semantics provides a model which is less precise (and therefore less questions can be answered) but is scalable and computable. Therefore, in choosing a suitable level for abstraction, we can examine the questions or the properties we want to verify in our program, dealing only with those elements in the semantics which are related to the considered property.

As an instance, we can consider a case of verifying a safety property in a computer program. In a safety property, we define what the unsafe condition is, highlighted in red areas in Figure 4 (18). The green zone in the figure, the abstract semantics, is a superset of all possible executions shown by trajectories. Since the abstraction model is a superset of the semantics of the program, if the abstraction model is safe, then the semantics is also safe.

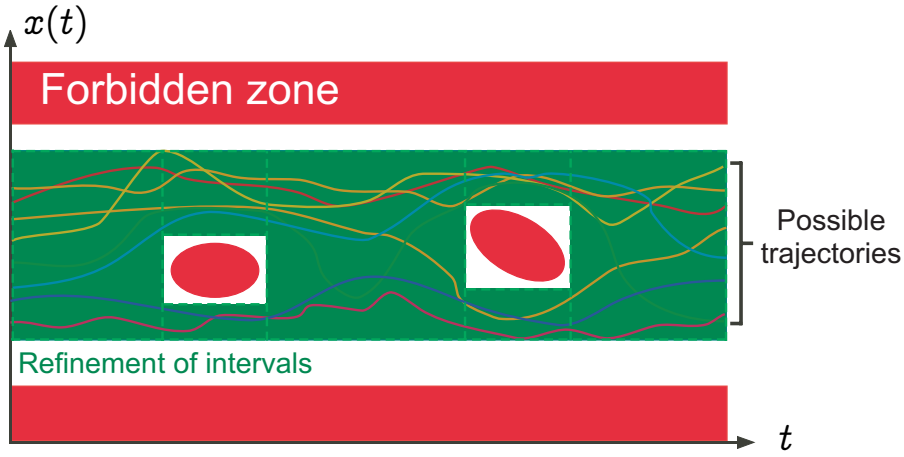

Figure 4: Precise Abstraction of a Safety Property (18)

There are several approaches to abstract semantics of a program. Model checking, deductive methods and static analysis are among these methods. Static analysis is particularly interesting for us since the abstract semantics is computed automatically from the program source based on predefined abstractions. According to Cousot & Cousot (19) *abstract interpretation provides a general theory behind all*

programs analyzers, which only differ in their choice of considered programming languages, program properties and their abstractions. By effective computation of the abstract semantics models (such as data-flow, data types and control-flow models), the analyzer tools can examine the behavior of programs before executing them.

In practice, Predicate Abstraction (20) is one the most popular and widely used techniques for semantic abstraction. In predicate abstraction we only keep track of certain predicates on data. Instead of tracking program variables of various types we then need to track the predicates of boolean type. Static analysis based on predicate abstraction of programs works based on finding approximate predicates (conditions) at each program location via a *predicate transformer*. These predicates, similar to assertions, define the properties which hold at that particular point in the program. A predicate transformer describes how the execution of different instructions changes these predicates. (21) According to Hoare logic (22), there are three components in predicate transformation:

$\{P\}\ C\ \{Q\}$

P is the set of pre-conditions, Q is the set of post-conditions and C is the command (instruction). Hoare logic provides the axioms to infer the relationship between these three components. Following the axioms and tracing the predicates as *pre-conditions*, one can determine what would be the *post-conditions* after symbolically executing an instruction. Similarly, assuming some predicates as post-conditions, one can reason about the pre-conditions which must hold, so that after execution of an instruction, the predicates are true. The following example shows an axiom schema for a simple assignment instruction:

$\{x+1 \leq N\}\ x := x+1\ \{x \leq N\}$

Based on the direction of the analysis to find the predicates, program analyses can be categorized into two classes: 1) forward semantics analysis, and 2) backward semantics analysis. Either way, the main challenge is to define the transformation function precisely to identify and capture *sound* predicates. Standard data- and control-flow analyses, and type checking systems adapt Hoare axioms in generating the predicates. The combination of these predicates can then be abstracted to define more high-level properties.

In practice, static analysis may lose its precision when encountering complex language commands, such as for-loops. However, as we will discuss in the following chapters, some properties (e.g., authorization properties) are generally independent of the loop invariants and they are generally not affected by the number of iterations of the loops.

In the following chapters we will discuss our approach in building abstraction models for web applications with more depth. To reason about security vulnerabilities in web applications, we need to find an appropriate abstract definition of the problem and an acceptable abstraction model for the program states. We will show that by using a proper abstraction, we are able to categorize and solve various security vulnerabilities in the same problem scope.

3

DETECTION OF AUTHORIZATION VULNERABILITIES IN WEB APPLICATIONS

Previously published as Maliheh Monshizadeh, Prasad Naldurg, V. N. Venkatakrishnan. MACE - Detecting Privilege Escalation Vulnerabilities in Web Applications. In Proceedings of 21st ACM Conference on Computer and Communications Security (CCS'14), Scottsdale, AZ, 2014.

As discussed in Chapter 1, authorization vulnerabilities are a subset of logic vulnerabilities. In a privilege escalation attack, the attacker or the malicious insider targets the security holes in the authorization logic of the application. To be able to detect these vulnerabilities, we first need to understand the authorization policy of the application. Lacking the program specification, we do not know exactly what the correct authorization policy is. However, we can find the inconsistencies in such policies which lead us to detection of authorization vulnerabilities.

In order to find authorization inconsistencies in applications, we define a notion of *authorization context* for web applications, that associates an authorization state to every program point in the application. We then develop a notion called *authorization context consistency*, which is satisfied when the application uses the same authorization context in order to access the same resource along different paths of a web application. When there is a mismatch in authorization contexts along two different paths, we flag that as an potential access control violation.

Employing the authorization context, we use one level of abstraction, in which we do not trace whether the user is authenticated at all times. Alternatively, we build a tuple of authorization related properties and we only examine them at the resource accesses. Using the static analysis framework, we only track these authorization contexts throughout the program.

We develop algorithms for computing authorization contexts and checking for authorization context consistency. These algorithms involve a variety of program analysis techniques that include control flow analysis, data flow analysis and symbolic evaluation. These algorithms are implemented in a tool that we call MACE (Mining Access Control Errors). These algorithms are bootstrapped by a small set of annotations provided by the vulnerability analyst, and we show that the effort for providing these annotations is small.

Using our approach, we are able to detect two kinds of privilege escalation vulnerabilities: the conventional (1) *Vertical Privilege Escalation (VPE)* when an attacker (outsider) or a malicious user (insider) tries to change her privilege level (access more than they are entitled to, say according to their role) and (2) *Horizontal Privilege Escalation (HPE)* when a malicious user tries to access the system resources of other users. In particular, our modeling of authorization context and our detection algorithms facilitate the detection of the latter kind of privilege escalation, thus making MACE the first tool in the literature that is capable of identifying HPE vulnerabilities in web applications.

In this chapter, we will demonstrate the evidence of the usability and usefulness of MACE is demonstrated by showing introductory examples, explaining the approach and running it against a large number of open-source code-bases.

3.1 Introductory Example

We illustrate the key aspects of the authorization problem for web applications with the help of an extended example. The traditional authorization or privilege escalation problem is tied to the functional role of a user in this context. If this user can exercise privileges that are not usually associated with their functional role, a vertical privilege escalation vulnerability is detected. In addition, as described in CWE-639 (23) the horizontal authorization problem describes a situation where two users may have the same role or privilege level, and must be prevented from accessing each other's resources.

Listings 1 to 7 present the source of a running example that illustrates these authorization vulnerabilities. The example is a simplified version of real-world code samples and describes typical vulnerabilities that were reported. The particular web application here is a blog that permits its registered users to insert, edit, delete, or comment on blog articles. There are two functional roles: admin and user, with the admin having control over all posts in the blog, whereas the individual users should only be able to insert, edit, or delete their own blog, and comment on other blogs.

Listings 3.1, 3.2 and 3.3 refer to a secure implementation of the application. Function `verifyUser`, shown in Listing 3.1, checks if the request is coming from an authenticated user. In Listing 3.2, an article is being added to the `articles` table in the database. The user name of the current logged-in user specifies the owner of the article, and the request includes the article text that is inserted into the database. Note that this insert implementation is secure, as the user is verified, and is found to have the required permission. Listing 3.3 refers to the delete operation, where the user can delete any post that she owns. Additionally, an admin user, as specified by the role `userLevel`, can delete all entries in a blog as shown by the second `DELETE` operation.

Listings 3.4, 3.5, 3.6 and 3.7 show example PHP files that implement the delete operation. Each implementation of the delete operation is vulnerable as described below:

- *No authorization* In Listing 3.4, the application performs a delete without checking if the user is authorized.
- *Improper permissions*. In Listing 3.5, the application does not check if the user has the appropriate permissions to delete an article.
- *Improper Delete-all* In Listing 3.6, the application does not check if the user trying to delete-all belongs to the `Admin` role, and therefore permits a privilege escalation attack.
- *Improper Delete* In Listing 3.7, the application does not check whether the user requesting the delete is the owner of the article, and is authorized to delete it. It therefore allows the currently logged in user to delete articles owned by any other user in the system, as long as the (public) article-ID is supplied as query argument.

The last two examples in the above list deserve special mention. Listing 3.6 is the conventional form of privilege escalation allowing an ordinary user to assume admin privileges, i.e., vertical privilege escalation (VPE). In contrast, Listing 3.7 allows for an ordinary user to assume privileges of any other ordinary user in the system, a form of privilege escalation known as *horizontal privilege escalation* (HPE) (24). To the best of our knowledge, this research was the first to discuss an approach for detecting HPE vulnerabilities automatically, in addition to detecting VPEs.

```
1 function verifyUser(){
2   if(!isset($_SESSION['userID']))
3     header('Location: /login.php');
4   else $userID = $_SESSION['userID'];
5   return;
```

```
6  }
```

Listing 3.1: verifyUser.php

```
1  verifyUser();
2  if($permission['canWrite']&&$action == 'insert')
3    query("INSERT INTO tbl_articles VALUES (
4    sanit($_GET['article_code']),
5    $_SESSION['userID'],
6    sanit($_GET['article_msg']))");
```

Listing 3.2: insert.php

```
1  verifyUser();
2  if($permission['canWrite']&&$action=='delete')
3    query("DELETE FROM tbl_articles WHERE
4    article_ID = '" + sanit($_GET['article_ID']) + "' and
5    author_ID = '" + $userID + "'");
6  else if($_SESSION['userLevel'] == 'Admin' && $action == 'deleteAll')
7    query("DELETE FROM tbl_articles");
```

Listing 3.3: delete.php

```
2.a  if($action == 'delete')
3.a    query("DELETE FROM tbl_articles WHERE article_ID = '" +
           sanit($_GET['article_ID']) + "'");
```

Listing 3.4: delete1.php (vulnerable version)

```
1.b  verifyUser();
2.b  if($action == 'delete')
3.b    query("DELETE FROM tbl_articles WHERE article_ID = '" +
           sanit($_GET['article_ID']) + "'");
```

Listing 3.5: delete2.php (vulnerable)

```
1.c  verifyUser();
...  ...
6.c  if($permission['canWrite'] && $action == 'deleteAll')
7.c    query("DELETE FROM tbl_articles");
```

Listing 3.6: delete3.php (vulnerable)

```
1.d verifyUser();
2.d if($permission['canWrite'] && $action == 'delete')
3.d   query("DELETE FROM tbl_articles WHERE article_ID = '" +
        sanit($_GET['article_ID']) + "'");
```

Listing 3.7: delete4.php (vulnerable)

TABLE I: The Authorization Context for different queries in Running Example

Query	Authorization Context
Listing 3.2	`$_SESSION['userID'], $permission['canWrite'], Column<$_SESSION['userID']>`
Listing 3.3 Line 3	`$_SESSION['userID'], $permission['canWrite'], Column<author_ID>==$_SESSION['userID']`
Listing 3.4	∅
Listing 3.5	`$_SESSION['userID']`
Listing 3.7	`$_SESSION['userID'],$permission['canWrite']`
Listing 3.3 Line 8	`$_SESSION['userID'], $userLevel == 'Admin'`
Listing 3.6	`$_SESSION['userID'],$permission['canWrite']`

Lack of Policy Specification Note that our techniques are designed to work directly on the source code of our target applications, without relying on the existence of a well-articulated policy manifest to clarify these functional roles. In order to know whether the web application is implementing its access control correctly, one needs to know what access control policy is implemented. Unfortunately, the only documentation of this policy is in fact the source code of the web application. Furthermore, we also face the problem that this policy implementation can be incomplete or incorrect. This makes the problem of checking for access control errors quite challenging.

3.2 Approach

To reason about a web application's authorization correctness, one must examine each sensitive operation (e.g., each SQL query execution) of the program and examine the authorization information required to perform that operation. Recall from Section 3.1, the running example identifies what can go wrong in the implementation of access control, including the absence of any authorization checks, improper ownership or privileges corresponding to user role, and untrusted session variables.

Authorization state Applications should ideally have a well-defined policy manifest of what authorizations should be granted to what users, taking into account the session context, but unfortunately this is not always explicit. Even in applications that manage to have policy documents, the implementation may not match the specification. The best understanding of access policy therefore is the operating context of each access request in the implementation. For each access request in a user session, corresponding to a particular control and data-flow in the program execution, we argue that the four tuple $\langle U, R, S, P \rangle$ represents the associated access control rule explicitly, with U the set of authenticated users, R the set of roles defined over the users, capturing different authorizations, S the set of session identifiers or session variables, and P the permissions defined on the resources (e.g., `read`, `write`). This set $\langle U, R, S, P \rangle$ is our authorization state. We illustrate this in our running example:

- In Listing 3.4 the identity of user is not checked. The value of `$action` (i.e., delete) comes from the input form and therefore is controllable by the user, and cannot be trusted. We infer that the access rule checked here is $\langle -, -, -, - \rangle$, which means any user in any role can actually execute this `DELETE` query providing any article if they know the corresponding *article_id*, which is a placeholder for session context.

- In Listing 3.5, the following access rule is being checked $\langle user, non\_admin, -, - \rangle$. Access is allowed to any user, and it is not checked if they are an admin or not.
- In Listing 3.6 the appropriate role is not being checked, the incorrect (inferred) access rule here is $\langle user, -, -, canWrite \rangle$ whereas the actual rule needs to include a check that the user is admin.
- In Listing 3.7 the correct ownership information, corresponding to the *user* who created the $article\_id$ is missing in the access check. Instead, the rule inferred here is $\langle user, non\_admin, -, canWrite \rangle$.

From these examples, we now see that the correct access rule associated with the delete query on $tbl\_articles$, depending on the role of the user, should be:

- $\langle user, non\_admin, -, canwrite \rangle$ and
- $\langle user, admin, -, - \rangle$

However, determining that this is the access rule, and that this is correct is not at all obvious. As mentioned earlier, all we have is the implementation, where the access rule is both control and data sensitive. Depending on whether the user is admin or not, different rules apply, indicating dependence on control. The ability to delete an article also depends on whether the same user had created the article, or had permissions to create it, requiring knowledge of data variables using data-flow analysis. Also implicit is the notion of the underlying access model. In this example, though the `admin` user may not be the owner of the article, an implicit role hierarchy lets her delete items and use the permission `canWrite`. Unfortunately, if the implementation is incorrect, the task of finding authorization errors becomes even more difficult.

Authorization Context One of the main ideas in our approach is the notion of matching what we call the authorization context, across related or complementary security sensitive operations, in terms *four-tuple* we have identified. We have no prior assumption about the authorization policy used by the web application authors. This authorization context is garnered by examining the code and trying to fill out the four-tuple at a given program point automatically. To do this we will first need to annotate the code to identify some of these fields manually. We populate our analysis by tagging the variables corresponding to user-ids, roles, session identifiers (i.e., those sets of variables that change every session) and permissions.

With each security sensitive operation identified, our goal is to try to infer what access rule is being enforced by the code. Using the annotated variables and the clauses in the query, we can now compute the *actual* authorization context at a given program point using a combination of control flow and data flow analysis. More details of these techniques are presented in Section 3.3. In Table I we show the actual context from a correct program from Listings 2 and 3 for `INSERT` and `DELETE`, and the actual context inferred from each of the incorrect inserts and deletes from Listings 4 through 8. Independent of what is correct, we observe that there are missing gaps in the conditions checked for access, across `INSERT`s and `DELETE`s to the same rows in the same table. Once we construct the authorization context, this acts as a specification for the access policy as implemented by the developer. The obvious question now is whether this policy is correct. However, we do not have any information about whether this is the case or not. It is possible for us to take this actual context to the developers and ask them to establish its validity, but this may not be always possible.

Authorization Context Consistency We observe that we can also compare the authorization context for different, but matching queries on the same tables. When the application uses the same authorization context in order to access the same resource along different paths of a web application, we term its authorization contexts to be *consistent* across the application. Any inconsistencies identified in this manner could indicate a potential problem with the access control implementation. Note that we do not know the correct access policy here, what we are trying to do is detect inconsistencies across related operations. We illustrate this idea with an example:

`INSERT` queries in a database are a good example of code segments that contain rich authorization information. For example, during creation of a row in a database table, we can expect to find some information about the *owner* of the row. Consider table `articles` with columns `(article_id, article_author, article_text)`. The following query adds an entry to this table:

```
INSERT INTO articles (article_author, article_text) VALUES
    ($_SESSION['userID'], sanitize($_GET['post']));
```

The variable `article_id` is incremented automatically. The ownership information, as to who can insert into this table can be inferred from `article_author` and the value for this column comes from `$_SESSION['userID']`. This ownership information is being checked on access, the authorization column tuple being $\langle \texttt{\$\_SESSION}[\prime\texttt{userID}\prime], -, -, - \rangle$.

Let us now examine the corresponding `DELETE` query on the same table `article`. Here, the only parameter for delete comes from the user input (via `GET`).

```
DELETE FROM article WHERE article_id = $_GET['post_ID'];
```

From the listing, it is clear that the authorization context for the delete does not check ownership, i.e., the inferred context is $\langle -, -, -, - \rangle$. If any user can guess the range of the current IDs in the table, she can delete any row owned by any other user. This simple example now suggests that it is useful to make the constraints or authorization states for these queries *consistent* and add the userID to the authorization context of `DELETE` as:

```
DELETE FROM article WHERE article_id = $_GET['post_ID'] AND article_author =
    $_SESSION['userID'];
```

The notion of computing the actual context and comparing it with those obtained from matching rules as discussed, using the authorization state four-tuple is both powerful and general. It accommodates a variety of different application access control models, being agnostic to the actual models directly. No a priori definitions or models are required, and the violations detected can encompass scenarios such as dynamic authorization and separation of duty (SoD), and the DAC model as shown. In fact, the DAC model is implied with the ownership information in the `INSERT` query. As long as the attributes that determine access can be captured by the authorization state abstraction, rich context variables such as time of day, location, integrity constraints, keys and shared secrets, etc., can all fit easily with the techniques discussed.

Note that normal sanitization of the user input, without associating it with the current session token is not sufficient. Of course, there are many challenges associated with this kind of matching. The obvious one is that this could be intended behavior, i.e., `DELETES` may have different permissions from Inserts. The actual context on the `INSERT` could be incorrect. Further, there could be more than one insert, corresponding to different roles or different session characteristics and the corresponding delete

has to be matched up accurately. Nevertheless, searching for inconsistencies in matching operations helps us resolve errors in real applications. The question is how often they lead to false positives or negatives and we explore this in detail in Section 3.4.

Algorithm Overview We now describe our algorithm to compute the authorization context, and compare contexts across matching requests to discover inconsistencies. To start the analysis, we need to find all the queries in the code, and proceed to compute the context at these query locations. Next, we compare the contexts of similar access locations (i.e., query locations). Inconsistency in the contexts or in the way the authorization tokens are used in accessing the DB, e.g., using `where` clauses can lead to detecting vulnerabilities as described.

Algorithm 1: Algorithm Overview

input : Application source, Authorization variables, Possible values for role variables

```
1  cfg := ControlFlowAnalysis();
2  dda := createDependencyGraphs(cfg);
3  sinkPaths := enumeratePaths(dda);
4  foreach sp ∈ sinkPaths do
5  |  AuthzContextAnalysis(sp);
6  |  if sink ∈ {INSERT, UPDATE, DELETE} then
7  |  |  queries += <symbolicQuery(sink), authz-c(sink)>;
8  analyzeInserts();
9  analyzeDeletes();
10 analyzeUpdates();
```

As shown in Algorithm 1, the main input to our analysis is the application source code (PHP code), annotated variables that correspond to user ids, roles, session specific attributes and permissions appro-

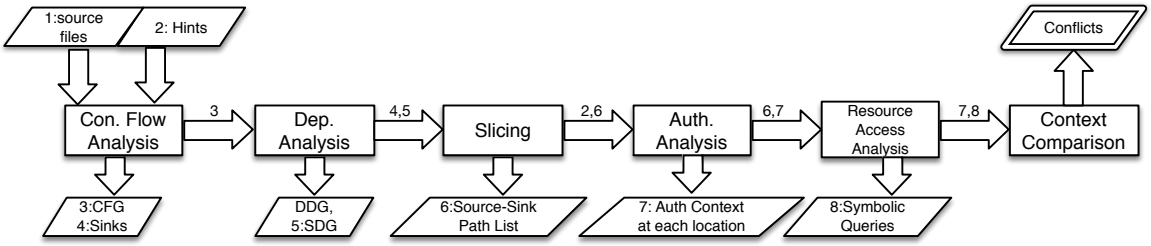

Figure 5: System Architecture. The numbers shown refer to outputs produced during various components, which are used as inputs for subsequent components.

priately. Once we have these annotations we perform a control flow analysis to identify paths involving these authorization sensitive variables. Next, we construct a data dependency graph using the annotated control flow graph to capture data-flows between the identified variables. A source-sink graph corresponding to entry points (sources) in web applications to particular sensitive queries (sinks) is now ready. On this graph, we gather the constraints at each sink as well as the annotated information flow context and construct our query context (lines 4-8). Context for different queries is computed in Algorithms 2 and 3 and checked for consistency to find errors. Further details are presented in Section 3.3.

3.3 Implementation

Figure 5 shows the architecture of MACE, identifying the various components of our tool, with (numbered) outputs produced by each component that are subsequently used as (numbered) inputs to other components.

Inputs. There are two sets of inputs to MACE: (1) source files and (2) annotations or hints provided by the developer / end user. Specifically, the set of hints provided by the developer is a small set of super

global variables that constitute the various components of the authorization 4-tuple as described earlier. In PHP, typically these annotations are on super-global variables such as `SESSION`. In our running example, the hint provided to the tool is that the super-global `userID` constitutes the specification of the user component of our 4-tuple. In our experience, the effort required to specify these hints is not high, as it took only a few minutes for each application that we tested (as discussed in our evaluation). With this information, together with the source code, MACE is able to identify potential privilege escalation errors.

Control Flow Analysis To identify authorization errors in the program, MACE uses static analysis methods to analyze the code. The advantage of using static analysis is that it can identify *all* sensitive accesses to important resources of a given type (e.g., SQL queries), and analyze all execution paths that lead to them. MACE includes a front-end to parse source files (in PHP). Subsequently a control-flow analysis is performed that results in a CFG (numbered output 3 in the figure) for the application, which explicitly identifies control flows throughout the whole application. In addition, this component also identifies a set of sensitive *sinks* in the application. Currently, sink identification in MACE is performed for SQL query locations (as identified by calls to `mysql_query`).

Data Dependency Analysis The next step in MACE is to compute a data dependency analysis. To illustrate the need for data dependency analysis, let us consult the running example. In Listing 3.1, the variable `$userID` holds the user information, which it receives from the super-global `$_SESSION['userID']`. `$userID` is subsequently used, and we need to capture these types of dataflows to reason about authorization. This requires a data dependency analysis, which is done by constructing data dependence graphs (DDGs) for each procedure.

In addition, MACE's analysis is inter-procedural. To see the need for inter-procedural analysis, let us consult the running example again. The assignment of `$userID` happens in procedure `verifyUser` (Listing 3.3), whereas the use of `$userID` happens in another file (`delete.php`). Since this analysis requires us to see such data-flows across all procedures, MACE also builds a system dependence graph (SDG) (25), which is essentially an inter-procedural DDG. The output of this step is a SDG (numbered output 5) in Figure 5.

Slicing In order to look for authorization errors at a particular sink, MACE analyzes paths that lead from the sources (entry points in a web application) to that sink. Such analysis needs to be *path sensitive*. Consider the running example in Listing 3.1. In this function, there are two paths, one that successfully checks if the user ID has been set (through a prior authentication step, not shown in the example for brevity), and the other that exits the application. The authorization context therefore exists in only one of the paths, and our analysis must be able to select such paths for further analysis, as well as ignore the other path, as it would not lead to a sink. Therefore we require a path sensitive analysis.

In order to analyze each path, we perform inter-procedural slicing using the SDG (System Dependence Graph) (25). Intuitively, for a given sink such as a SQL query, the corresponding SDG captures all program statements that construct these queries (data dependencies) and control flows among these statements. MACE performs backward slicing for the sinks such that each slice represents a *unique* control path to the sink. Each of these control paths is therefore an instance of sensitive resource-access. A number of steps are performed during the slicing operation. Loops are expanded by unrolling, either 0 or 1 or 2 times. Expanding the loops for more scenarios is not necessary in our approach, as we observed that authorization context is generally not modified in loops. In addition, the conditional expressions

are preserved in the SDG by NOP (No Operation) nodes, so that the information that is checked by these conditional expressions can be used in computing the authorization context. For instance, in our running example, the condition `isset($_SESSION['userID'])` is stored along each control path. Paths that do not reach a sensitive sink are omitted from subsequent analysis. At the end of this slicing step, MACE outputs a list of source-sink paths (numbered 6 in Figure 5. For our running example involving Listings 3.1, 3.2 and 3.3, we have three, one that reaches the `INSERT` query, another that reaches the `DELETE` query, and a third that reaches the same query but corresponds to deleteAll.

Authorization Context Analysis Using the paths computed during the slicing step, MACE computes the authorization state along each such path from the source to the sink. To do this, it starts with the super-globals identified from the user provided annotations (numbered by 2 in Figure 5) and checks if they (or other program variables that get receive values from these super-globals through data-flows) are consulted in conditions along the path from the source to the sink. If so, that information is symbolically represented in the authorization context 4-tuple. For instance, for our running example involving Listings 3.1 and 3.3, deleting (Line 3) involves an authorization context that checks both the user has logged on, the permission (`canWrite`) and therefore the corresponding context is inferred: $\langle \texttt{userID}, \_, \_, \{canWrite\} \rangle$ whereas Listing 3.1 and 3.5 involve the following authorization context: $\langle \texttt{userID}, \_, \_, \_ \rangle$. This step is computed using our sliced SDGs, which provide the data-flow information. At the end of this step, MACE outputs source-sink paths with annotated authorization context at each location along the path.

Resource Access Analysis Having computed the authorization context at all program points in any given path, the next step in MACE is to see how these are used towards accessing application resources.

The main resources in web application are DB tables, and we need to check whether the authorization contexts across all resources are consistent. However, the access control context at the query location may still be incomplete in capturing the access restrictions the application places along the current path being analyzed. To see this, we refer to Listing 3.3. In the first `DELETE` operation, the query is constrained by two `WHERE` clauses: (1) the specific article to be deleted and (2) the `author_ID` field from postAuthor of the table restricted to the current `userID`. The latter constrains the authorization context by way of *row restriction*: i.e., each article can be deleted only by a user who has her `userID` stored in the same row of the table. This happens only when the corresponding `INSERT` query for the same row (shown in Listing 3.2) inserts the `userID` in the field `author_ID`. This implicit form of "ownership" needs to be captured by MACE in the authorization context. Thus, the relationship between `author_ID` and `userID` that is implicit in the code needs to become explicit. The tool captures this information as *Authorization Columns* for each table. Each *authorization column* element captures the column name and the symbolic value used to constrain the query. Returning to our running example, the *authorization columns* for the table `tbl_articles` is: `[author_ID = $_SESSION['userID']]`

Note that, in cases where the developer did not specify column names in `INSERT` queries, MACE uses DB schemas, which are automatically generated by parsing database creation files (`CREATE TABLE` queries) to mark the *authorization columns*.

For this purpose, MACE symbolically evaluates the source-to-sink path that leads to the query so that the relationship between the super-global variables that eventually reach the query becomes explicit. Symbolically executing the path that leads to the first `DELETE` query in Listing 3.3 makes this relationship explicit. For instance, we obtain the following symbolic query for the above example.

Algorithm 2: Analysis of INSERT Queries

input : List of queries and their authorization contexts
output: List of Tables, Tables' authorization contexts, Tables' authorization columns

```
1  sortSymbolicQueries();
   // based on Table names
2  foreach table t do
3      foreach ins-q ∈ getInsertQueries(t) do
4          authzContexts(t)+=getAuthzContext(ins-q); authzCols+=getAuthzColumns(ins-q);
5      if AuthzContext(t).size > 1 then
6          diff := compare(authzContexts(t));
7          if diff then
8              raiseWarning(INSERT_Conflict);
9      if authzCols(t).size > 1 then
10         diff := compare(authzCols(t));
11         if diff then
12             raiseWarning(INSERT_AuthzColumn_Conflict);
13 return authzCols, authzContexts;
```

```
DELETE FROM tbl_articles WHERE article_id = $_GET['article_ID'] AND author_ID
    = $_SESSION['userID'];
```

Notice that the query is now entirely expressed in terms of symbolic super-globals such as user inputs `$_GET['post_id']` and `$_SESSION['userID']`. Having the symbolic query for each query location allows us to compare similar accesses to the column `authorID` in the DB, and compare the authorization contexts for *similar* accesses to the same table.

In addition to identifying inconsistencies, the previous steps allow one to see if there is a possibility of privilege escalation due to insecure use of user-supplied parameters in authorization decisions. User supplied parameters such as `$_COOKIE` can be tampered, and therefore authorization decisions must not refer to them. Having a symbolic query that makes any such use explicit also facilitates MACE to identify these types of errors. At the end of this step, MACE outputs a set of symbolic queries. These

queries contain the resources (tables) and together with the authorization context at each resource along every path that leads to the resource. The specific type of access (`INSERT, UPDATE, DELETE`) is also available through the symbolic query. As mention earlier, paths that lead to `SELECT` queries or those that do not lead to sensitive resources are discarded.

Context Comparison The goal of this step is to compare the authorization context at each resource access and identify inconsistencies. To this point, we have gathered two sets of information with respect to authorization in previous steps: 1) the authorization contexts for query locations and 2) the resource access parameters (authorization columns) for each table.

The first step in context comparison is to group the query-path pairs based on the *table names* in the query (Line 1 in Algorithm 2. During symbolic execution analysis done in the previous phases, we are able to resolve the table names for static and dynamic queries in each possible sink-path pair. Then, for each table, we gather the authorization context and authorization columns, which are used in table accesses. The number of distinct authorization contexts and authorization columns may be more than one, for different queries in different program locations. Therefore, we need to resolve these differences and in any case report these conflicts. Lines 7-11 in Algorithm 2 compare the contexts for `INSERT` queries on a given table. The discussion about the analysis of these conflicts comes in Section 3.3.1. After finishing all `INSERT` queries, we proceed to analyze `DELETE` and `UPDATE` queries. The reason we start this way is because *ownership information* is added to the resources at their creation time, in a DAC model, which is typical of such applications. `INSERT` queries show us where the ownership information comes from in the program and in which column of the table they are going to be stored.

Algorithm 3: Analysis of DELETE Queries

```
   input : authzCols, authzContexts
1  foreach table t do
2      foreach del-q ∈ getDeleteQueries(t) do
           // Compare Authorization Contexts
3          if getAuthzContext(del-q) == getAuthzContext(t) then
               // Compare Authorization Cols
4              authzClause = getAuthz(getWhereClauseCols(del-q));
5              if authzClause != getAuthzColumns(t) then
6                  raiseWarning(HORIZONTAL_ESC);
7          else if getRole(getAuthzContext(q)) < getRole(getAuthzContext(t)) then
8              raiseWarning(VERTICAL_ESC);
9          else
               // Other inconsistencies may have various authorization
                  vulnerabilities
10             raiseWarning(INCONSISTENCY);
```

In the next step, to analyze the rest of the queries (i.e., `DELETE` and `UPDATE` queries) we compare their authorization contexts shown by 1) the resource accesses (where clauses in queries) and 2) the authorization context annotations, with the information we gathered from `INSERT` queries. Algorithm 3 shows the analysis of delete queries. The analysis of `UPDATE` queries is done in the same way.

The data in database tables should be changed exclusively with a privilege level equal or more than the level specified in their authorization context so that the integrity of these tables remains intact. Lines 9 and 10 in Algorithm 3 show how we check for vertical privilege escalation vulnerabilities in our tool.

In addition to the authorization context, we check table access parameters (lines 4-6 in Algorithm 3) to detect horizontal privilege escalation vulnerabilities. These attacks happen in the same privilege level as the legitimate users, however, a malicious insider can manipulate the data stored in DB tables owned by other users. This additional check of accesses tries to prevent such attacks.

Lines 12-15 in Algorithm 3 detect any other inconsistency in the authorization contexts. These inconsistencies are also reported back to the user for further analysis.

3.3.1 Conflicting Contexts

During the comparison phase, both for authorization contexts and the resource access comparisons, there may be scenarios in which the contexts or access parameters do not match entirely. In these cases, the authorization context or `WHERE` clause with *more restrictions* is viewed as the *stronger* context / clause. This is intuitive – quite often, the *more restrictive* the context, the more specific (or precise) it is about the access rules regarding the resource being referred to. Below, we discuss different conflict scenarios and how MACE addresses these conflicts.

In the case of one `INSERT` query present in the application for a given resource, we assume that the authorization information extracted from the query must be present in the authorization context of further accesses (`UPDATE`s or `DELETE`s). In case of any conflict after the comparison, if the authorization context of the `INSERT` query is *stronger* than the other query's context, we raise a warning. Depending on the missing element in the authorization 4-tuple, the type of the warning may vary. A missing or weaker *role* information is generally an indication of a vertical privilege escalation vulnerability caused by the current privilege role level being less than what was present in the `INSERT` query's context. A missing *user* element in the context, while it was present at the time of the insert, indicates a horizontal privilege escalation vulnerability. Relying on user provided inputs (such as `GET`, `POST` or `COOKIE` variables) in the 4-tuple is an indication of a general mismanagement of sessions and authorization in the application.

3.3.2 Precision and minimizing warnings

MACE is a 'best effort' tool to detect missing or inconsistent authorization information. It is based on the intuition that developers aim to get most cases right, and some occasional cases wrong. In the rare case that the developer gets *none* of the cases right, then MACE's approach cannot detect errors.

To have a more effective tool with a better confidence rate, there are a few general steps we took in order to improve the overall precision and lower the false positive rate.

`INSERT` queries with missing authorizations So far, we set authorization contexts in `INSERT` queries as the base for consistency analysis. If an `INSERT` query misses some crucial authorization information, then our tool may not report faulty `DELETE` or `UPDATE` queries (as long as they are consistent with the faulty `INSERT`), causing possible false negatives.

User-controllable query parameters. There are some applications that permit user-controlled parameter (such as those of `GET, POST, COOKIE`) values in authorization decisions. Since MACE uses data flow analysis, it is able to observe these incorrect authorizations and reports them (we identify and report 10 such errors in our evaluation). In such cases of vulnerabilities, MACE does not further proceed to analyze the queries that are impacted by these flaws, so that the number of warnings reported by the tool is minimized. After fixing these vulnerabilities, the user can re-run MACE to identify if there are still missing authorizations.

`SELECT` queries. Technically speaking, it is possible for MACE to include `SELECT` queries and analyze them for inconsistencies, as the analysis required to identify authorization for a `SELECT` query (e.g., dataflow analysis) is no different compared to an `INSERT`. However, including `SELECT` queries is

primarily a question of the user's tolerance of the signal / noise ratio for an policy-agnostic tool such as MACE. To see this, let us consider an example of a news article website. The news articles are publicly viewable and so at the corresponding `SELECT` query there would be no authorization information, whereas the users of the website often have to authenticated and authorized to be able to post news articles. Comparing such `SELECT` queries with `INSERT` queries often will lead to false alarms. Therefore, in order to eliminate such false alarms, a user might decide to omit analyzing `SELECT` queries, as we did in the evaluation of MACE. Another choice that a user has, which involves additional manual effort, is to provide additional annotations that identify the tables that store sensitive data (and therefore require authorization on `SELECT` queries).

3.3.3 Other Issues

Unsupported `PHP` features MACE is implemented for PHP, and makes use of the Pixy (26) tool for control flow analysis. A small set of features in PHP language are not handled by Pixy and therefore MACE does not deal with them. For instance, dynamic inclusions and certain object-oriented features of PHP are not handled entirely. However, these have not limited the applicability of MACE to the application suite that used in our evaluation. Note that, while MACE works in the context of PHP, which is widely used, our technique (using authorization context differentials, symbolic execution, dataflow analysis) is independent of any platform.

Counting the number of vulnerabilities. Many vulnerable queries might be fixed by a single common authorization check at a shared program location. However, MACE treats each query location as an independent operation and reports and counts vulnerable queries separately. We prefer to do so because we think each of the reported vulnerable queries might lead to a different instance of attack. By treating

TABLE II: PHP APPLICATIONS

Application	SLOC	# php files	# of query locations	# of DB tables	Analysis times
phpns 2.1.1alpha	4224	30	40	13	8220
DCPPortal 5.1.44	89074	362	308	34	982
DNScript	1322	60	27	7	35093
myBloggie 2.1.3	6261	59	24	5	373
miniBloggie 1.1	1283	11	5	2	35
SCARF 1.0	978	19	13	7	54
WeBid 1.0.6	27803	266	687	47	1492

the queries in isolation, we can identify the vulnerability type with more precision. As we see in Section 3.4, we may report large number of vulnerable locations because of one single missing authorization check, but in each such case of multiple vulnerabilities due to a single reason, we explicitly indicate so.

3.4 Evaluation

Implementation MACE is designed to analyze PHP Web applications. MACE is implemented in Java and is about 10K lines of code. We use an open-source tool and library (`TAPS` (27)) to get the control flow graphs and enumerate execution paths for PHP applications. The experiments described in this section were performed on a MacBook Pro (2.4 GHz Intel, 4.0 GB RAM).

3.4.1 Effectiveness

Experiments. We ran our tool to analyze the effectiveness of MACE on a suite of seven small to large `PHP` free and open-source Web applications. As shown in Table II, the applications range from approximately 1k to 89k source lines of code (SLOC). These applications were used as benchmarks in previous research studies (28; 29; 8). The results of our evaluation fall under the following categories: (1) vulnerabilities identified by MACE and detailed statistics about the vulnerabilities identified in our

experiments, (2) performance, scalability of MACE and (3) the annotation effort required from the developers. We have verified by hand all of the authorization vulnerabilities in the applications, which were reported by the tool.

For each application, we annotate the authorization tuple variables and then run MACE with given annotations and the source code. MACE then lists all of the conflicts, the vulnerabilities, their locations in the source code and the values, which cause the inconsistencies.

Results summary. Table III presents the summary of our experiments. The table lists the number of conflict reports, and also shows how many of these were indeed vulnerabilities (true positives - TP) and how many of them were reported incorrectly (false positives -FP) by our tool. Furthermore, the breakdown of the vulnerabilities (true positives) between the two types of privilege escalations namely HPEs and VPEs is presented in columns 4 and 5. The last column in this table gives the breakdown of the identified vulnerabilities were known (i.e., previously reported in CVEs or by previous studies) versus unknown (i.e., zero-day vulnerabilities).

As reported in the last column of the table, MACE is able identify zero-day authorization vulnerabilities in the following applications: `phpns`, `DCP-Portal`, `mybloggie` and `SCARF`. In the following subsection, we will go through the details of these vulnerabilities.

3.4.2 Vulnerabilities Identified

phpns The `phpns` application is an open-source news system. The application allows three roles in the system for the users: 1) guest users (unauthenticated users) who can only view the news articles; 2) normal users who must be logged into the system and use the article management panels and 3) *admin* user who also must be logged into the system and can access both article and user management panels.

TABLE III: OVERVIEW OF VULNERABILITIES

Application	# query conflicts		VPE	HPE	Known/Unknown
	TP	FP			
phpns	7	0	✓	✓	0, 7
DCPPortal	46	0	✓	✓	0, 46
DNScript	0	0	-	-	-
myBloggie	6	0	✓	✓	3,3
miniBloggie	1	0	✓	-	1, 0
SCARF	11	0	✓	✓	1, 10
WeBid	0	0	-	-	-

Basic permissions such as adding, deleting and updating articles are set by default for the new users of the system.

```
$new_res = general_query('INSERT INTO articles
                (article_title, artcle_sbtitle,
                article_author, article_cat,
                article_text,...)
            VALUES('.$data['article_title'].','
            .$data['article_subtitle'].','
            .$_SESSION['username'].','
            .$data['article_cat'].','
            .$data['article_text']'');
```

Listing 3.8: Inserting an article item in in inc/function.php, `phpns`

```
$items = $_POST; //get vars
...
$sql = general_query("DELETE FROM ".$databaseinfo['prefix']."".'articles'."
    WHERE id IN (".$items.")");
```

Listing 3.9: Deleting an article item in `article.php`, `phpns`

Using MACE, we found seven vulnerabilities in this application, all of which are previously unknown, two of which we describe below. Consider the actual code for the inserting and deleting users, shown in Listings 3.8 and 3.9 respectively. The first vulnerability allows an unauthenticated user can delete any comment without any authorization checks. MACE was able to identify this because the relevant authorization context is not consulted at the delete operation. The implication of this vulner-

TABLE IV: DETAILS OF WARNINGS

Application	Number of violations		
	insert-insert	**insert-update**	**insert-delete**
phpns	0	5	2
DCPPortal	0	21	25
DNScript	0	0	0
myBloggie	0	3	3
miniBloggie	0	0	1
SCARF	1	8	3
WeBid	0	0	0

ability is that an outside attacker (who has no credentials in a given installation of `phpns`) can delete *any* comment item in the application. This is an example of a vertical privilege escalation attack (VPE). Another detected vulnerability is found in `manage.php` that allows for an *authenticated* user to delete other users' news articles by providing arbitrary article IDs (which are available to all users through inspection of URLs). This vulnerability is a horizontal privilege escalation (HPE). We have reported these and other vulnerabilities in `phpns`.

dcp-portal The `dcp-portal` application is an open-source content management system. This application allows two authenticated roles: admin user and non-admin user (normal user). Consider Listings 3.10, 3.11, and 3.12, which refer to the authorization operation, insertion and deletion of agenda items in a calendar table. Variable `$_COOKIE["dcp5_member_admin"]` is being used to determine whether the user is an admin user or not. While inserting an item in the agenda, this variable is consulted, and the agenda item is entered in the table `t_agenda`. However, while deleting the item, while the authorization function is consulted, the deletion is based on a (user supplied) value `$_REQUEST["agid"]`, thus making the requests inconsistent. The implication of this vulnerability is that it allows any user in the

system to delete another user's agenda entries, thus making it a HPE, which was a previously unknown vulnerability.

```
1 if (UserValid($_COOKIE["dcp5_member_id"])) {
2   ...}
```

Listing 3.10: Authorization function in lib.php, `dcp-portal`

```
1 if ((isset($_REQUEST["action"])) && ($_REQUEST["action"] == "add") &&
      ($_REQUEST["mode"] == "write")) {
2   $sql = "INSERT INTO $t_agenda (user_id,
3   subject, message, date) VALUES ($_COOKIE['dcp5_member_id'] ," .
4   htmlspclchars($_REQUEST['subject']).",".
5   htmlspclchars($_REQUEST['aktivite']).",
6   $date)";
7   $result = mysql_query($sql);}
```

Listing 3.11: Inserting an agenda in calendar.php, `dcp-portal`

```
1 if ((isset($_REQUEST["action"])) && ($_REQUEST["action"]=="delete")) {
2   $sql = "DELETE FROM $t_agenda WHERE id = '".$_REQUEST["agid"]."'";
3   $result = mysql_query($sql);}
```

Listing 3.12: Deleting an agenda in calendar.php, `dcp-portal`

We also found 44 other VPEs due to the incorrect implementation of `UserStillStillAdmin` function in `dcp-portal`. The first argument of this function takes the value of `$_COOKIE["dcp5_member_id"]` and determines whether the user with this userID is an admin. The value for the userID comes from a cookie variable and not from an established authorization state at the server side, which makes all 44 distinct queries in the admin path vulnerable to VPE.

myBloggie The `MyBloggie` application is an open-source blogging software. When we ran MACE on this application, we found six privilege escalation vulnerabilities. In three of these vulnerabilities, the validity of a session is not checked in many instances as the check shown in Listing 3.13 does not appear in del.php, delcat.php, deluser.php files. Even in the files that do check this constraint, MACE found horizontal escalation attacks. The parameters used to delete rows do not check for authorization information. For instance, the parameter used to access and delete the rows in `POST_TBL` is coming

from user-supplied values such as `GET["post_id"]` and is prune to HPE. MACE found three such unreported vulnerabilities in this application.

```
if (!isset($_SESSION['username']) && !isset($_SESSION['passwd'])) //go to
    login;
```

Listing 3.13: authorization Check in addcat.php, `MyBloggie`

miniBloggie The `miniBloggie` application is also a blogging Web application. In this application, there is no role or privilege level defined for the users. Thus, users are either authenticated (`$_SESSION['user']` is set) or not. Even with this simple authorization rule, the application is vulnerable to privilege escalation as detected by MACE. These scenarios involved missing checks that need to be present in order to ensure the user is a valid one before access to table rows is granted. In `del.php` (Listing 3.15), function `verifyuser()` is omitted, making way for the vulnerability, which was previously unknown.

```
session_start();
if (!verifyuser()){
  header( "Location: ./login.php" );
}else {...
  if (isset($_POST["submit"])) {
    $sql = "INSERT INTO blogdata SET
    user_id='$id',
    subject='$subject',
    message='$message'"...";
```

Listing 3.14: Inserting a blog user in add.php, `miniBloggie`

```
session_start();
if (isset($_GET['post_id'])) $post_id = $_GET['post_id'];
if (isset($_GET['confirm'])) $confirm = $_GET['confirm'];
if ($confirm=="yes") {
  dbConnect();
  $sql = "DELETE FROM blogdata WHERE post_id=$post_id";
```

Listing 3.15: Deleting a blog user in del.php, `miniBloggie`

SCARF The `SCARF` application is an open-source conference management software which helps the user to submit and review papers. The possible roles in `SCARF` are admin and normal user. Both roles

must be authenticated to interact with the software. Variable `$_SESSION['privilege']` indicates whether a user is an admin or not.

MACE detects several types of authentication and authorization bypass vulnerabilities in `SCARF`. For example, in `generaloptions.php`, the admin can delete users and modify the `option` table. The page has no authorization check before it proceeds to performing `admin` tasks. As a result of this vulnerability, a normal user of the system who is legitimately authenticated can delete other users. To fix the problem this method, `require_admin()`, should be added at the beginning of the file which verifies whether the current session is the admin session or not. If it is not the admin session, the program exits.

```
1 if (isset($_GET['delete_email'])) {
2   query("DELETE FROM users WHERE email='" . escape_str($_GET['delete_email']) .
        "'");
3 }
```

Listing 3.16: Deleting a user in generaloptions.php, `SCARF`

```
1 function require_admin() {
2   if (!is_admin()) {
3         die ("...");}
4 }
5 function is_admin() {
6   if ($_SESSION['privilege'] == 'admin') return TRUE;
7   else return FALSE;
8 }
```

Listing 3.17: Missing Authorization in generaloptions.php, `SCARF`

Ten other vulnerabilities reported by MACE in this application can be attributed to a single reason. The reason for these vulnerabilities being reported is that the constraining parameter used in certain `UPDATE` or `DELETE` queries derives it value from `$_GET['session_id']`, which is an untrusted source (i.e., the HTTP client). The corresponding `INSERT` query uses the `$_SESSION['user_id']` which is an authorization variable as shown in the following code snippets. The column `session_id`

in table `sessions` is an auto-increment key. Since untrusted values are never part of server authorization state, the authorization contexts for these queries were reported empty. Since the parameter `$_GET['session_id']` is provided by the user, and the values are guessable (auto-incremented value), an attacker can impose himself on any guessable session.

```
INSERT INTO sessions (name, user_id, starttime, duration) VALUES
    (mysql_real_escape_string($_POST['name']), $_SESSION['user_id'], $date,
    $duration)
```

Listing 3.18: Inserting a session addsession.php, SCARF

```
UPDATE sessions SET user_id=$_POST['chair'] WHERE
    session_id=$_GET['session_id']
```

Listing 3.19: Updating a session in editsession.php, SCARF

Webid and DNScript MACE did not report any conflicts in these two applications.

Vulnerability & Inconsistency Reports. Table IV shows the breakdown of the number of inconsistencies reported by our tool. The inconsistencies between various types of query pairs (insert-insert, insert-update and insert-delete). Together with Table III, we see that MACE is precise and produces no false positives. This low FP rate is due to the use of authorization 4-tuple to model the authorization state of sessions at the server. Using the reports generated by the tool (including the locations of the queries and the missing authorization), a developer can proceed to fix the application.

3.4.3 Performance & Scalability

We evaluated MACE on a suite of Web applications with different sizes ranging from 1K to 90K. Columns 2-3 in Table II show the size and number of php files in the applications, and column 4 gives an estimation of the number of query (insert-update-delete) locations (in source).

TABLE V: ANALYSIS OF QUERIES

Application	# query-path pairs			# query-authzInfo pairs
	insert	update	delete	
phpns	2564	222	920	78
DCPPortal	56	60	58	158
DNScript	8	2	13	26
myBloggie	5	0	2	40
miniBloggie	3	9	1	3
SCARF	4	26	12	19
WeBid	131	22	7	323

Table II (column 6) shows the total analysis time for each Web application ranging from 35 seconds to 35093 seconds. About %95 of the analysis time has been spent to create the dependency graphs and enumerate execution paths.

The increase in the number of possible paths increases the number of created symbolic queries. However, the number of distinct symbolic queries may still remain relatively low as shown in the last column of Table V, where we present the number of unique symbolic queries and their authorization information 4-tuple. Currently, MACE analyzes each file separately and builds the aggregated contexts when all the queries are gathered. The performance of MACE can be improved, especially if we summarize recurring contexts for basic user-defined functions. Since MACE is a static tool, the analysis times are quite acceptable for the benefits provided by the tool.

3.4.4 Annotation Effort

To run MACE, we manually identify the 4-tuple variables for each of the applications as hints for our tool. Developers typically use global and super-global variables (e.g., in `SESSION` or `COOKIE`) to represent user roles, user IDs, and the possible permissions for the logged-in users. These variables

are further used to hold the authentication and authorization-related values throughout the program. Table XI in Appendix A shows the variables we identified as hints for our programs.

The manual annotations are developed by observing the session management functions in login procedures. In our experience, developing these annotations is not hard for users familiar with the application, and certainly for developers who coded the application. To objectively measure the annotation effort, we performed a user-study experiment. To assist this experiment, MACE was extended to automatically generate a list of global and superglobal variables, which are used in if-statements, which is a superset of authorization variables. This list is then refined to exclude user input variables (such as `GET` and `POST` superglobals), and is provided as a starting point to the user.

To measure the effort needed to identify the 4-tuple, we asked a graduate student who had basic knowledge about Web applications to develop these annotations. We provided the application sources and the globals list generated by MACE. The student was provided the `mybloggie` and `phpns` applications, which are mid-tier applications in our benchmark suite. She was able to produce annotations that matched our own annotations for both the applications, and took about 50 minutes for generating and verifying the annotations. This experiment lends evidence that only modest efforts are required in providing annotations. We also note that our experience with providing such annotations is consistent with prior work in web access control that makes use of similar annotations (30; 31). Given the number of unknown vulnerabilities identified by MACE, we believe such annotation-assisted automated bug finding is an attractive alternative to weeks of human effort and manual code inspection.

3.5 Summary

This Chapter presents MACE, a static analysis tool to detect authentication and authorization vulnerabilities in web applications. By analyzing the resource access operations and building an access model for each resource, MACE is able to find inconsistencies in access control logic throughout the web application.

We have shown the MACE is scalable and effective in detecting privilege escalation vulnerabilities precisely. MACE is also the first tool reported in the literature to identify a new class of web application vulnerabilities called Horizontal Privilege Escalation (HPE) vulnerabilities. MACE works on large codebases, and discovers serious, previously unknown, vulnerabilities in 5 out of 7 web applications tested. Without MACE, a comparable human-driven security audit would require weeks of effort in code inspection and testing.

4

RETROFITTING LOGIC VULNERABILITIES IN WEB APPLICATIONS

Previously published as Maliheh Monshizadeh, Prasad Naldurg, V. N. Venkatakrishnan. Patching Logic Vulnerabilities for Web Applications using LogicPatcher. In Proceedings of The 6th ACM Conference on Data and Application Security and Privacy (CODASPY) 2016, New Orleans, LA, 2016.

Logic vulnerabilities cause a program to operate incorrectly or exhibit unexpected behavior. The ability to fix security-sensitive logic vulnerabilities in web applications, caused by incorrect control checks or improper data computation, is an important requirement in the SANS Critical Security Control (CSC) (32) for effective cyber-defenses.

Application Inconsistency Vulnerabilities (AIVs) - as a subset of logic vulnerabilities - are caused by lack of consistency in the design or implementation of security checks. Some of them include:

- *E-commerce logic inconsistencies* These vulnerabilities result from inconsistent checking of business validation logic in the application code. Prior work on detecting these vulnerabilities include using model checking (33), modeling correct payment logics combined with static analysis (34), and invariant generation and blackbox testing (35). In all these cases, the vulnerability analysis tools report inconsistencies in checks, and these inconsistencies are subsequently verified for the presence of vulnerabilities.

- *Client-server inconsistencies* The validation performed by client-side JavaScript can be used as a specification to check the server-side for vulnerabilities. (36; 29; 37) take this approach towards vulnerability detection. In this case, the vulnerability is a client-side check that must have been performed by the server.
- *Access control inconsistencies* Inconsistency in application authorization logic along different execution paths result in application vulnerabilities. (38; 39; 40; 41) look for these types of inconsistencies, which are subsequently confirmed for the presence of actual vulnerabilities.

To fix these vulnerabilities, we have designed and implemented LOGICPATCHER (42), a static analysis tool that takes a patch condition, i.e., a reported vulnerability and a path descriptor as input, and suggests optimal or near-optimal candidate path placement locations. Using our automated tool, application customers or system administrators can confirm and test the candidate patches instead of going through the arduous work of manual code inspection.

4.1 High-level Goals and Challenges

The problem of fixing errors automatically is not straightforward, as it requires a thorough analysis of large amount of source code. In the case of legacy web applications, if the vendor or the developer of a web application is no longer in business, or simply failed to fix the problem, the burden is on the customers of the application to either change their application or patch the vulnerabilities themselves. Patching an application requires a thorough understanding of the application and the error, something that deployment professionals do not have time for.

Detected vulnerabilities may be embedded deeply in the code, beyond such easily analyzable or stand-alone constructs such as user interfaces or end-user communication modules. Manually locating

these errors may require rigorous analysis of the source-code, with many interdependencies. Modifying or adding code snippets to fix these errors will now imply we take extra care not to change the logic and the functionality of the interdependent code, and only fix the vulnerable path(s). Therefore, the requirements for a successful patch is twofold: 1) generating correct conditions for missing security checks, and 2) the correct placement of these conditions.

At a high level, we need to identify the candidate patch locations for the application while preserving the logic of the program. This goal becomes challenging as we do not know much about the functionality of the program in the first place. The only information we have aside from the source code of the application, is the reported vulnerabilities and their locations in the code.

Patch Generation: In order for the generated patch to work, it should first have the necessary instructions which will prevent the exploits. While the generated patch should work, it should not interfere with the main functionality of the program.

Patch Placement: We need to find the proper location for the patch, which assures that the patch will not change the logic or the functionality of the application along other execution paths. For patch generation, we rely on outputs from AIV detection tools such as Rolecast and MACE (38; 39). Our work in patch placement is related to FixMeUp (43), a static analysis tool that suggests patches for access control vulnerabilities. Our research finds a wider scope for solving this problem. We believe that we can generate security patches for any type of logic vulnerability caused by missing or inconsistent checks, with minimal guidelines about the vulnerability, and find optimal or near optimal placement of the suggested changes directly. We summarize the contributions of our research: 1) Precise formulation of logic vulnerabilities in order to patch the applications, 2) Design and implementation of an analysis

tool called LOGICPATCHER, 3) Finding candidate placements of the generated patches along the vulnerable path(s), and 4) Generation of security patches for reported logic vulnerabilities for 9 open-source applications.

4.1.1 Vulnerabilities to Be Patched

To patch logic vulnerabilities, the developer needs to have some basic knowledge about the vulnerability to be able to patch it effectively. In particular three different items affect the patching process: $< C, P, E >$ in which C is the missing condition which needs to be added to vulnerable paths, P is the particular vulnerable path, and E is the exception handling policy the developer considers if the condition failed at execution time.

The Conditions Set The set of missing conditions C is expressed in terms of variables, values and conditional operators. For example, $\{\texttt{strlen}(\$\texttt{password}), 8, >=\}$ defines a condition on the length of the variable `$password`. The variables used in C can be functions of input variables (e.g.,`strlen($password)`) or they can be internal variables related to server-state (e.g.,`$_SESSION['username']`). The values also can be constants or derived from some server-state (e.g., the result of a DB query).

The Vulnerable Path(s) P is the path from the source to the sensitive operation (*sink*). Security analysis tools can usually generate execution traces which shows the possibility of the exploiting the vulnerability.

The Exception Handling Policy Set The set E defines the set of actions which are allowed to be executed if the conditions C do not hold. For example, the developer may choose to use `exit()` or `die("message")` or she may choose to log the failed operation in the system. The semantics of the actions specified in E depends on the application, the usage and developer's choice.

4.1.2 High-level Challenges

There are some high-level challenges involved in the process of generating security patches for vulnerable web applications. Given that most web applications lack program specifications, our approach should be able to work on the source code with minimal input from the developers/application admins. That is, the patch generation and placement modules should rely on the extracted logic of program and inferred security policies.

Missing Check Dependencies A security patch is basically a generated code snippet which is going to be placed somewhere along the vulnerable path(s). The code snippet may contain variables which should have semantical and syntactic values at the location they are going to be placed. Therefore, the data dependencies for these variables should be kept intact and meaningful. Forward and backward data dependencies of the variables used in the code snippet will assure that the dependencies are correct and consistent along different paths and throughout the application.

The following example, derived from a real-world vulnerable application, shows a sink (the DELETE query) in which the variables used in the query are dependent on some previous instructions. Although the value used in the sink depends on the user input (e.g.,`$_POST['username']`), it also depends on some server state (e.g., the data stored in `users` table which determines if the username is valid in the application). If the sink missed the authentication, along with the check in Line 2, other instructions (on validation of the username and password) should accompany the check in the patch. Therefore, we need analysis to determine the instructions with dependencies on the input values and the server-state.

```
1 session_start();
2 if(isset($_POST['username']) && isset($_POST['password'])){
3   $result = mysql_query("SELECT userid FROM users
4                 WHERE username = " + $_POST['username'] + " AND password = " +
                      $_POST['password']);
```

```
5   if ($result) {
6     $_SESSION['username'] = $_POST['username'];
7     if($action == 'delete'){
8       $res = mysql_query("DELETE FROM posts WHERE author=" +
            $_SESSION['username'] + "AND post_ID=" + $_POST['post_id']);
9       if($res) echo("Deleted post successfully");}}}
```

Listing 4.1: Sink Control and Data Dependency

Overprotecting The patching process includes adding some code snippets (patches) to the source code which will retrofit certain vulnerable sinks. Inserting instructions to a code will change the state of the program (the post-conditions) from that point forward:

$$\{P\}\ C\ \{Q\} \Rightarrow \{P\}\ C\prime\ \{Q\prime\}$$

Changing the conditions after the security patch may affect the current functionality of the program. Therefore, we need to make sure that the changes made to the application source code do not in any way affect the functionality and logic of the program *in the execution paths which were not vulnerable.* In particular, we need to assure that the logic of the program in other non-vulnerable paths is not changed due to the insertion of the patch.

The *path-sensitivity* of the patching problem requires us to know which path it is going to retrofit. Our analysis can gain this information about the vulnerable path from the vulnerable location (sink) and can use this information to distinguish execution paths.

Optimization Although the main goal of automatic retrofitting of the web application against logic vulnerabilities is to make sure that the vulnerabilities are correctly patched, but we also prefer that the resulting web application source remains optimized in terms of both time and space. That is, we prefer to add one patch at a location where it prevents several vulnerable locations than adding multiple patches into multiple locations in the code. To place the patches to optimal locations in an application, our

approach considers various scenarios with different candidate places, it then identifies which candidate place is optimal for an application with one or multiple vulnerable sinks. More detailed discussion about optimization of the patching process is in Subsection 4.2.4.

4.2 Approach

Figure 6 shows the overview of our approach. There are two two main steps: 1) generate the appropriate patch, 2) place the patch in an appropriate location in the original code.

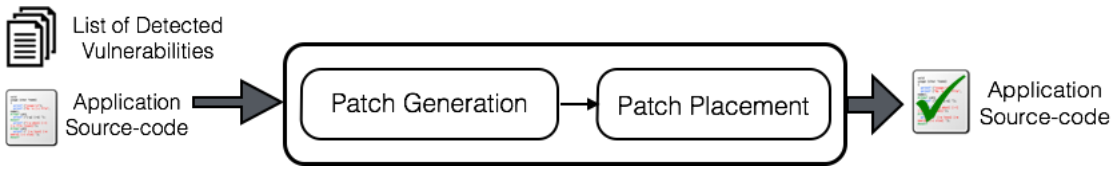

Figure 6: LOGICPATCHER Overview

Input Our approach uses general information about the vulnerability to start its analysis. This information includes: 1) the missing condition, C, 2) the path to the vulnerable sensitive operation (*sink*), P, and 3) the exception policy E.

Error Handling Policy LOGICPATCHER requires an (optional) exception handling logic E as input. If no inputs are provided, the default exception handling strategy is to terminate of the program. This way if the placed security condition C is not met by the program state, the program terminates without entering in a non-secure state. Though termination is one option, the developers may want to take other

actions if a malicious input is given to the program. Logging and sending alerts to system administrators are among popular actions that one may take after a malicious attack. To better help users of the tool in deciding the exception handling methods, LOGICPATCHER provides an option to analyze exception handling options in the source code (see Appendix B).

4.2.1 Patch Generation

The Missing Conditions Automatic detection tools usually report AIVs in terms of the missing conditions as well as the details about the location of the vulnerability. They express these conditions in terms of conditions on 1) user inputs or 2) conditions on the internal server state. A condition is a tuple of $< Var, Val, Rel >$ in which *Var* is the variable, *Val* is the value for the variable which may be a constant value or a dynamic one, and *Rel* is the relational operator. Our approach can use the missing conditions C and the vulnerable execution trace (path) P to compute the program slice for these constraints. This program slice is called the *patch*.

The security conditions to be added to the patch include variables which specify the available information *context*. This information context defines the type of the patch as well as the instructions which need to be added to patch.

Program Slicing A missing condition, when inserted in code, may need to be accompanied by some other data- and control-dependent instructions if necessary. To preserve the data dependencies, we may add other instructions to the patch. That is, given a condition $< Var, Val, Rel >$, we perform backward program slicing so that the variables *Var* and values *Val* involved in the condition set C will be meaningful and valid at the patch location.

4.2.2 Patch Placement

The problem of patch placement may seem to be trivial at first glance. However, the location of the vulnerability is not necessarily the location where the security patch should be placed. A logic vulnerability patch is basically a constraint written as an conditional statement. To insert an conditional statement we need to find the starting point and the ending point for the conditional statement block.

An important challenge in patching an execution paths is to ensure that the patch does not affect other execution paths, an occurrence we call *overprotection*, since adding a security condition to a path which was not vulnerable may make that execution path unavailable, and prevent it from being executed under legitimate circumstances. To place the generated patch, the approach should find candidate places in the code where it can insert the beginning and ending of the conditional statement block without interfering with other execution paths and other sinks.

Multiple Paths to one Vulnerable Sink When only some of the execution paths to a sink are vulnerable, we need to make sure that we are fixing those vulnerable paths only and are not changing other paths. When there are multiple execution paths to a vulnerable sink, our approach must assure that the vulnerable(s) path is patched, as well as ensure other non-vulnerable paths remain intact. Based on this goal, we categorize different scenarios when placing the patch.

To show these scenarios, we provide an example of a sink (a DELETE query). The code example deletes some records from the table `$table`. This example derived from a real-world example in `phpNS` application (simplified for more clarity). The main command to delete is executed in `function.php`.

```
1  function delete_item($table,$where) {
2    $res = mysql_query("DELETE FROM ".$table ." ".$where);
```

```
3       log_this('delete','User <i>'.$_SESSION['username'].'</i> has
            <strong>deleted</strong> a tables contents. Table: '.$table.'
            '.$where);}
```

Listing 4.2: Delete Query phpNS, function.php

Analysis shows that this query is vulnerable to authorization bypass in some of the paths, for instance it lets the authenticated normal users to delete *any* comment in the system if they can guess the comment ID (which is an incremental integer value and not hard to guess). The missing condition is to check whether the current user is the owner (the author) of the comments and if so, she can delete the comments. Now let us consider different scenarios where the other execution paths in the application affect our decision about patch placement. For code in Listing 4.3, all of the paths leading to the sink are vulnerable since both paths allow authenticated users to delete any comments.

```
1   if ($do == "comments") {
2     if ($_GET['action'] == 'delete') {
3       $where = "WHERE id =".$_POST['id'];
4       delete_item('comments',$where);
5       log_this('delete_comments','User <i>'.$_SESSION['username'].'</i> has
            <strong>deleted</strong> the comments: "'.$where"');}}
6   else if($do == "all") {
7     if ($_GET['action'] == 'delete') {
8       delete_item('comments',"");
9       log_this('delete all comments','User <i>'.$_SESSION['username'].'</i>
            has <strong>deleted</strong> all comments');}}
```

Listing 4.3: Deleting a comment, article.php

An abstract control flow graph of this example is depicted in Figure 7 (a) where both paths to the sink are vulnerable and therefore we can simply place the patch at the sink location. Since all of the paths to a sink are vulnerable, then patching the query or patching the path to the query is straightforward: it can insert the patch just before sink location. We can place the patch, which is a check for the author of

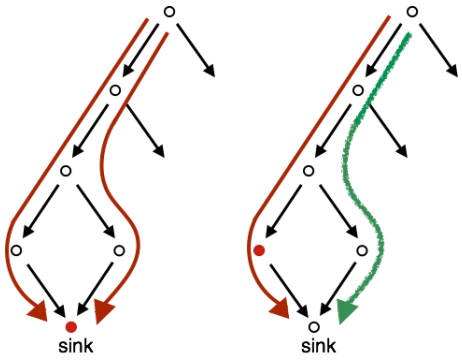

Figure 7: a) Execution paths for Listing 4.3, b) Execution paths for Listing 4.5

the comment (users should be able to delete their own comments), in the `delete_item` function if this function is used only for deleting comments. Listing 4.4 shows the candidate place for the patch.

```
1  function delete_item($table,$items) {
2    XXXX: placing the Patch here : XXXX
3    $res = general_query("DELETE FROM ".$table." WHERE id IN (".$items.")");
4    log_this('delete','User <i>'.$_SESSION['username'].'</i> has
         <strong>deleted</strong> a tables contents. Table: '.$table.'. Items:
         '.$items);}
```

Listing 4.4: Patching scenario 1, phpNS, function.php

Another scenario is shown in Listing 4.5 in which only some of the paths to the query are vulnerable. The path to deleting a comment is the vulnerable one but the other path to delete an article is not since it restricts the query to the author of the article.

Figure 7 (b) shows the control flow representation of Listing 4.5. If we place the patch inside `delete_item` function, then we are adding code to non-vulnerable paths, in which we will check for the author of comments, and so we are injecting an irrelevant check to the code which makes the query unavailable for those non-vulnerable paths.

```
1   if ($do == "comments") {
2     if ($_GET['action'] == 'delete') {
3       $where = "WHERE id =".$_POST['id'];
4       delete_item('comments',$where);}}
5   else if($do == "articles") {
6     if ($_GET['action'] == 'delete') {
7       $where = "WHERE id =".$_POST['id'] "AND author=".$_SESSION['userID'];
8       delete_item('articles',$where}}
```

Listing 4.5: Non-vulnerable Paths to the Sink, article.php, phpNs application

Unlike the earlier scenario, in these case, we certainly cannot patch the path at query location. To patch the vulnerable path, we must ensure that the patch is only accessible along the vulnerable path. Our approach finds the nearest location from the sink where the patch would not interfere with other paths. The candidate location for this example is shown in Listing 4.6.

```
1   if ($do == "comments") {
2     if ($_GET['action'] == 'delete') {
3       $where = "WHERE id =".$_POST['id'];
4       XXXX: placing the patch here XXXX
5       delete_item('comments',$where);}}
6   else if($do == "articles") {
7     if ($_GET['action'] == 'delete') {
8       $where = "WHERE article_ID = " .$_POST['id']." AND
            author=".$_SESSION['userID'];
9       delete_item('articles',$where);}}
```

Listing 4.6: Patching scenario 2, phpNS, article.php

Other Paths to Other Sinks Now consider a scenario in which some paths to the sink are vulnerable, but the vulnerable paths are not entirely disjoint from other non-vulnerable paths to other sinks. Listing 4.7 shows the example where the code is shared with other users in the system (in this case `admin` user(s)). Based on the previous solution, we may place the patch before calling function `delete_item`, however, we are restricting the admin users from deleting comments of other users, where there is no such policy in the application demanding that.

```
1   if($_SESSION['group'] == "admin"){
2     $admin = true;}
3   if ($do == "comments") {
4     if ($_GET['action'] == 'delete') {
5       $where = "WHERE id =".$_POST['id'];
6       delete_item('comments',$where);}}
7   else if($do == "articles") {
8     if ($_GET['action'] == 'delete') {
9       delete_item('articles',"WHERE article_ID = " .$_POST['id']." AND
            author=".$_SESSION['userID']);}}
```

Listing 4.7: Other Execution Paths in article.php

Figure 8 (a) shows this scenario. Although the nearest node to `sink1` is node `x`, we cannot place the the patch right after this node, because it will interfere with the non-vulnerable path.

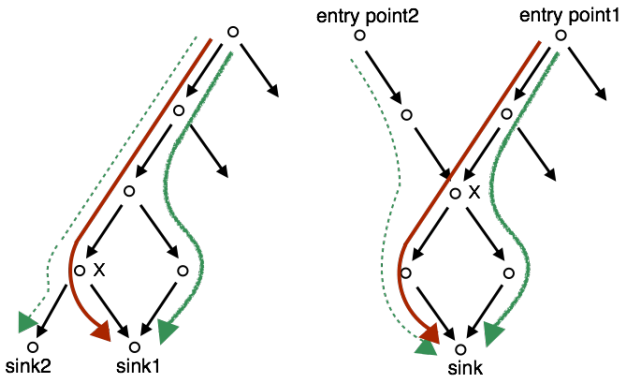

Figure 8: a) Interference with Other Execution Paths: CFG representation of Listing 4.7, b) Interfering Execution Paths: Non-disjoint Paths

The above example, or even the one shown in Figure 8 (b)), where two or more non-vulnerable paths are not entirely disjoint from the vulnerable path, show that in some cases the overlap between the paths

make it impossible to place the patch along the vulnerable path without interfering with non-vulnerable ones.

In order to uniquely patch the vulnerable path, we need to be able to uniquely identify it. Therefore, our approach uses *Path Profiling* techniques (44) to uniquely identify the execution paths in the instrumented application. Path Profiling algorithm assigns a unique state to each execution path and instruments the program with state transitions on each path. The changes in the transition state should be in a way that by reaching the end of each path, we reach the same state which was assigned to the path. Path Profiling algorithm introduced in (44) computes the state transitions efficiently.

After identifying the path, and checking for interfering paths, if inserting the condition *C* alone results in interference between vulnerable and non-vulnerable paths, then the path id (hash ids generated in path profiling) will be merged with condition *C*, to make it non-interfering, as shown in Listing 4.8.

```
1 if(C && PATH == <hash_path_x>){
2   /*vulnerable sink and its dependent instructions*/}
```

Listing 4.8: Merging Path Profiling info

4.2.3 Algorithm

A high-level overview of our algorithm is shown in Algorithm 4. We start our analysis by parsing and generating the control-flow graph (CFG) of the source code. After creating the CFG, we should identify the CFGNodes which are data- or control-dependent on the sink. This is a crucial step since we are going to wrap the sink inside an introduced security condition *C*; therefore we need to also include these dependent instructions inside the condition block so that the program executes consistently during

Algorithm 4: Overview of the Patching Algorithm on a Single File

```
   input : C: Missing Check
   input : P: Vulnerable Path to the Vulnerable Sink
   input : E: Security Exception Handling Policy
   // Step 1:  Generating the Patch
1  CFG := getCFG(sourceFile);
2  foreach Condition cond ∈ C do
3  |  instructions_before_sink,Vars, Vals := ∅;
4  |  Vars.add(getVar(cond));
5  |  Vars.add(getVal(cond));
   |  // non-constant values
   |  // Step 1.1:  Backward Program Slicing
6  |  for cfgNode ∈ CFG do
   |  |  // Starting from sink backward
7  |  |  foreach Variable newVar ∈ cfgNode do
8  |  |  |  if newVar is data-control dependent on Vars then
9  |  |  |  |  Vars.add(newVar);
10 |  |  if cfgNode.include(Vars) then
11 |  |  |  instructions_before_sink.add(cfgNode);
   // Step 2:  Finding the Patch Scope:  Algorithm 5
   // Step 3:  Patch Placement
   // Step 3.1:  Forward Live Variable Analysis
12 instructions_after_sink := ∅;
13 for cfgNode ∈ CFG do
   |  // Starting from sink forward
14 |  foreach Variable v ∈ cfgNode do
15 |  |  if v is data-control dependent on sink operation then
16 |  |  |  instructions_after_sink.add(cfgNode);
17 |  |  |  break;
18 return instructions_before_sink, instructions_after_sink, functionScope;
```

run-time. Sets `instructions_before_sink` and `instructions_after_sink` are used to gather these dependent instructions.

Finding the right Scope The previous scenarios show that it is important to reason about the patch and the placement of the patch with respect to other execution paths in the application.Our approach uses static analysis techniques to generate and place the patch in a proper location and it also reasons

about the possibility of multiple vulnerabilities and how LOGICPATCHER should generate and place the patches so that they would also not interfere with each other.

Recall that the input to LOGICPATCHER is a vulnerable sink location *P*, the missing (authorization) condition *C*, and the security exception handling policy *E*. LOGICPATCHER generate 1) the patch *S* which is a set of instructions; 2) the candidate location *L* for the patch to be inserted in. *L* consists of the patch context (i.e., filename where the patch introduced), and two candidate locations to insert the start and ending of the missing condition.

First our approach needs to identify the function context where we should inject the missing condition. Since the execution paths are intra- and inter-procedural according to where in path we inject the missing condition, LOGICPATCHER may work with different function contexts.

Consider this sequence of function calls `f()` $->$ `g()` $->$ `h()` where function `f`calls function `g` and also function `g` can call function `h` and function `h` includes the vulnerable sink. The first challenge is identify the proper function context; that is, whether LOGICPATCHER should add the missing condition to function `h()`, `g()` or `f()`.

To answer this question we should think about our second goal: not to overprotect the application, which means that we should not add the condition to the execution paths which were not vulnerable in the first place. So we first examine the number of function calls to each of this functions – starting from the innermost function `h()` and going backward to `f()` – and if the sequence of such function calls includes paths which are not vulnerable we go one step backward.

After this step, the location of the function is found and the scope of the variables involved in the patch is identified. Now LOGICPATCHER needs to find the start and end block for the condition itself.

Algorithm 5: Finding the candidate file and scope

```
   input  : CFGNode vuln_sink, List vulnerablePaths, missingAuthzCheck
   input  : Candidate file name, Candidate start point, candidate end point
1  Function (findFunction(h)) currentFunction := h;
2  callerFunctions := all callGraphNodes who call currentFunction;
3  if callerFunctions.size == 1 then
       // only one path
4      candidate patch function := currentFunction;
5  else if callerFunctions are all in vulnerable paths then
       // all paths leading to the currentFunction are vulnerable
       // so we can patch the current function
6      candidate patch function := currentFunction;
7  else
       // traverse backward
       // and put the check before calling the currentFunction
8      foreach function f ∈ callerFunctions do
9          findCandidatePatchFunction(f);
```

The start point for the condition can come just before the vulnerable sink inside the candidate function f.

For the end point of the check, we first need to analyze the data dependencies after the sink. To address this problem, we do an inter-procedural Live Variable Analysis (LVA) on the output of the sink operation to find the scope of the code where the output is still live. This analysis may add other variables to the set due to data dependencies.

Live Variable Analysis A patch puts the vulnerable sink inside an conditional statement block with the condition *C*. If we simply put only the sink inside a conditional statement, we may change the original semantics of the application. Merely including the sink instruction to the conditional statement block may make other instructions after the sink invalid. Listing 4.9 shows an example where the result of the query is used to other operations.

```
result = execute_query(...);
if(result){ echo("successful transaction.");}
else{ handle_exception();}
```

Listing 4.9: Result of sink is used for another operation

Now one only puts the sink (the query) inside the condition block, then the result value would not be accessible for paths which do not execute the true branch of condition *C*. Therefore, our approach needs to reason about a candidate location for ending the condition block.

```
if(missing_condition {\mytt C}){
  result = execute_query(...);}
if(result){
  do_something();}
```

Listing 4.10: Wrongly patched version of 4.9

The correct patch for Listing 4.9 is shown in Listing 4.11.

```
if(missing_condition {\mytt C}){
  result = execute_query(...);
  if(result){
    do_something();}}
```

Listing 4.11: The correct patch for 4.9

What is changed in Listing 4.11 is that every instruction which is control- or data-dependent on the value of the sink operation is now also contained by the condition block. To find all of the instructions which are dependent on the sink value, we should first find the variables which are dependent on the value of the sink. Each instruction which includes such variables, will be added to the instruction list which is going to be added to the condition block.

To find the *sink-dependent* variables LOGICPATCHER performs forward data-control dependency analysis starting from the sink location and put the ending of the block where the result of the sink is

no longer used. LOGICPATCHER uses similar ideas for *Live Variable Analysis* to find which variables are dependent on the sink. The implementation details of this algorithm is shown in the implementation Section.

4.2.4 Discussion

Patching the Sink Based on the type of vulnerability and the information context available in the execution paths, there may be two options to patch the application at the source level: 1) patching the execution paths leading to the sensitive operations (*sinks*), and 2) patching the sink itself by changing the access parameters of the sensitive operation.

For example, when the sink type is a DB query, to patch the vulnerability we may be able to augment the query where clause if the columns in the DB table hold the condition variables. This way the query becomes more restrictive. However, it is not always the case that the values are available for access parameters. For example, when the missing authorization information is not available in the query or in any of the columns of the table, we need to add the missing conditions to the execution paths leading to the vulnerable sink.

As an example, consider the following query. This query is accessible for all logged-in users of a blog application. The following query is detected as vulnerable because it lets the authenticated user to delete *any* blog post if she can provide a valid post ID.

```
1 $sql = "DELETE FROM blogdata WHERE post_id=$post_id";
```

Listing 4.12: Query to be Patched

To patch the vulnerability, we may have the option to place the condition `authorID == $_SESSION['userID']` in the `WHERE` clause in the query as shown in Listing 4.13 if the `blogdata` table contains the column

`author_ID` . Alternatively, we can place the condition in the path leading to the query as shown in Listing 4.14.

```
$sql = "DELETE FROM blogdata WHERE post_id=$post_id AND author_ID=$author_id";
```

Listing 4.13: Patched Query

```
if(getAuthor($post_id) == $author_id || $author_id=="admin"){
  $sql = "DELETE FROM blogdata WHERE post_id=$post_id";}
```

Listing 4.14: Patched Path

Patching Multiple Vulnerabilities In real world, it is often the case that an application may have more than one logic vulnerability. It is because some logic vulnerabilities go together, i.e., they may have the same or similar root cause(s). In patching logic vulnerabilities therefore we should examine how we can patch the application in a way that is both 1) correct and 2) efficient. Reasoning about patching multiple vulnerabilities is crucial because we do not want to repeat the patch(es) firstly because it may change the logic of the program and secondly because repeating the patch at multiple locations is not time- and space-efficient. In finding the optimal location for patching multiple vulnerabilities in an application several scenarios may occur:

1) Vulnerabilities of the same type: A patch consists some security checks which would control values on some variables or some server-side state. Vulnerabilities of the same type may share these checks entirely or partially. In this scenario, we should place the checks if they are being added previously.

2) Vulnerabilities with various types, but the same root cause: In another scenario, the vulnerabilities may be of various type and therefore they do not share the security checks in the patch, however, they share a common prefix in the code. i.e., they share paths.

In this case, even though the checks themselves are not shared, but the patch may change data-dependencies for the variables used in the patch or they may change the server-side state. Hence, we must watch out for the instructions included in each patch to make sure they do not interfere with each other.

3) Independent Vulnerabilities: Vulnerabilities are independent when they do not share a common prefix in the code or they do not have the same cause. These vulnerabilities can be patch independently. Our approach takes each vulnerability one by one and generated the candidate patch for it.

4.2.5 Limitations

LOGICPATCHER is a 'best effort' tool which suggest security patches to developers/system administrators. A separate formal verification or manual effort must be made to verify the correctness of the generated patches and the resulting source code. This is because of some high-level limitations which are involved in using automated tools to patch security vulnerabilities which we discuss in the rest of this subsection.

Lack of Program Specification The main reason that our approach cannot assure the correctness of the patches is that our approach does not require specifications on the functionality of the program, functions and program statements in the source code. For instance, during the patch generation, our tool must decide which instructions should be wrapped by the security condition C. This would be much easier if the tool had more knowledge about the sensitive instructions rather than performing static data- and control-dependency analyses.

Although we do not verify the correctness of the patches automatically, we believe that the proposed approach *by design* minimizes undesirable side-effects on the functionality of the applications. For

instance, in the presence of interfering execution paths, as discussed in Section 4.2, we use path profiling to prevent the introduction of new control instructions in non-vulnerable paths.

The generated patch consists of the missing condition, and its control and data dependencies. Thus, the correctness of the generated patch depends on the correctness of the data and control dependency analysis. We use backward program slicing and live variable analysis techniques to create the patch. These data and control dependency techniques are performed statically and therefore they may be imprecise. However, we use a very conservative approach to analyzing the dependencies along the execution paths and so we do not miss any dependent instructions. For instance, in backward program slicing, even though the dependencies of DB query instructions are not available statically, our tool adds those instructions in the patch if condition *C* is dependent on the result of the query and also it adds the query string variables to the set of variables of interest. This may seem to cause overprotection problems, however, the path profiling procedures ensure that the instructions protected by *C* are going to be used only in the vulnerable path.

Cascading Sinks There are some complications involved with patching applications in which the developers use cascading sinks. Assume that the developers perform two sensitive operations (reading and writing to db tables, or files) in the same execution path, and one of these sensitive operations is vulnerable. The decision to whether include both sinks in the missing condition statement block or just include the vulnerable sink depends on whether the data- and control-dependency analyses detect the two operations as related or not. The problem may cause some side effects (as we will see in Section 4.4 under correctness discussion) on the generated patch and correctness of the code.

4.3 Implementation

We have designed and implemented a tool which, given a list of application vulnerabilities in a web application, it will generate security patches for the vulnerabilities and will place them in the source-code of the application so that each sink is no longer vulnerable to that particular vulnerability. Figure 9 shows the architecture of LOGICPATCHER in two major steps: 1) patch generation and 2) patch placement. As shown in this figure, each phase has various analysis components. In this section we go over the implementation of each of these components.

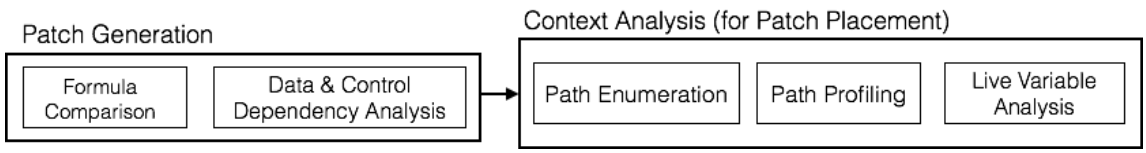

Figure 9: Patch Generation and Patch Placement in LOGICPATCHER

4.3.1 Inputs to LOGICPATCHER

LOGICPATCHER starts the analysis with two sets of inputs:(1) source files and (2) list of detected vulnerabilities which is expressed as the tuple $< C, P, E >$ (ref Sec 4.1.1) . In our experience, providing such input does not need any additional effort. Security tools usually report the vulnerabilities with these details.

4.3.2 Patch Generation

To generate the security patch, LOGICPATCHER uses static data- and control-flow analysis to preserve the original semantics of the program in the presence of a new security condition *C*. A security condition has a set of variables and values which should be have the same semantics as other consistent paths while used in the security condition. Therefore LOGICPATCHER uses static analysis to identify the instructions which include these variables and values. These instruction then are then added to the patch.

LOGICPATCHER starts with using a PHP parser and Control Flow Graph (CFG) generator to get the CFG of the application. It then traverses the CFG of one of the consistent paths backward from the sink location to find the related instructions to condition *C*'s variables and values. This analysis is intra- and inter-procedural which enables LOGICPATCHER to reason about consistency of the patch throughout the entire application. After adding all the related instructions to the patch, LOGICPATCHER compares these instructions to the instructions which are in the vulnerable path *P*. The reason for this comparison is that our approach avoids the insertion of instructions to a path twice as it may cause side-effects.

4.3.3 Patch Placement

As discussed in Section 4.2, along the execution path leading to the vulnerable query, there are several candidate locations to put the patch. If we put the patch at the beginning of the entry point, we may cause so many side-effects because other instructions are going to be affected by the patch. So the best choice is to put it as close as possible to the query itself to minimize the side-effects.

To find a correct place for the patch to be inserted in the code, LOGICPATCHER uses several code analysis techniques. These techniques assure that LOGICPATCHER inserts a patch into the vulnerable path and it does not affect other execution paths.

Path Enumeration To enumerate all of the execution paths, LOGICPATCHER uses Pixy (26) to parse all of the PHP files in the source code and build the Control Flow Graph (CFG) of the application. LOGICPATCHER then traverses the CFG nodes in the graph and lists all of the paths for each entry point. It finally gathers all of the execution paths for all of the entry points in the application. Our path enumeration is intra- and inter-procedural.

LOGICPATCHER starts uses one entry point at a time to enumerate all of the paths starting from that entry point. For non-control instructions LOGICPATCHER adds the CFG node for the instruction to the current path to be constructed. Each time LOGICPATCHER visits a control instruction (i.e., if-statement, function call, function return) it changes the list which holds the paths.

For if-statements it clones a path into two different paths and puts the paths back into the list of paths. For a function call instruction, LOGICPATCHER first saves the return address (the next instruction to be executed after the function return) in a hash structure and then traverses the CFG for the function. By visiting a function return, LOGICPATCHER searches the hash and adds the CFG node for the return address to the current path.

Path Profiling As discussed in Listing 4.5, there may be cases where placement of the patch in a vulnerable path interferes with other execution paths since it is not disjoint from the other paths and therefor we may need to introduce extra control flags to assure that the patch is going to be added to the vul-

nerable path only. In order to patch the vulnerable path and only the vulnerable path, LOGICPATCHER needs to keep track of the vulnerable path and uses some criteria to uniquely identify it.

Path profiling techniques (44) enable LOGICPATCHER to achieve this goal. The path profiling concept first enumerates all of the paths and assigns a unique number to paths to each sink or exit location. It then computes the transition numbers and instruments the application. At execution time, by each transition, the computed value for the transition is going be added to the state of application. Eventually when it reaches the sink or exit location it arrives to the same assigned number to the path. LOGICPATCHER uses the same ideas to keep track of the vulnerable path and for efficiency it uses an abstracted version of the path which only includes control-instructions: conditions, loops, exit, return, and function calls.

Live Variable Analysis As discussed in Section 4.2, there are instructions after the sink which need to be included in the condition block during the patching. Since we need to know the last location where the dependent variables are used (the result value of the sink operation is still *alive*), we need to perform a backward dependency analysis, which is going to show the last location where the variables are used. LOGICPATCHER uses *Live Variable Analysis* (LVA) (45) ideas to keep track of the variables which are data- or control-dependent on the result of the sink operation.

To perform the analysis, LVA algorithm takes the current context (i.e., the current file which the patch is going to be inserted in) as input and performs a backward analysis to compute the `In`, `Out`, `Def` and `Used` sets. The last instruction where all sink-dependent variables are still alive is going to be marked. The end of the condition block will be just after the marked instruction. Since the value of the sink operation may be used in return statements and be variables in other file contexts may be dependent on the sink, LOGICPATCHER performs intra- and inter-procedural LVA. Note that the end block will

be inserted to the file context where the condition is going to be added, therefore the block syntax is preserved by LOGICPATCHER.

Output LOGICPATCHER generates a candidate security patch which includes the missing condition *C*. It also gives out the candidate location to place the patch which is somewhere along path *P*. In cases where path profiling is needed, LOGICPATCHER produces an instrumented version of the code along with new control conditions, which assure that the patch is only executable along the vulnerable path.

4.3.4 Discussion

There are some technical limitations associated with static analysis of programs for patching PHP applications with current open-source tools. These problems however are not derived from the design of LOGICPATCHER, and they do not limit the applicability of our approach for patching applications.

Object-Oriented PHP features LOGICPATCHER uses Pixy (26) for parsing and creating the control flow graph of the application. Some complex features in PHP such as object-oriented features are not handled by Pixy.

Path enumeration in the presence of loops Path enumeration is the most crucial component of our approach. It is used by both for patch placement and path profiling algorithms. Therefore the precision of path enumerates affects the precision of our tool. However, with static analysis of PHP code and limitations in the libraries we use, there are some cases where enumerating the paths statically is a challenge. Currently our path enumeration approach treats loops similar to if-statements and therefore each loop is treated in two different branches: 1) it is not going to be executed and 2) it is going to be executed once.

TABLE VI: PHP APPLICATIONS

Application	SLOC	# PHP Files	# of Sink locations	# of Resources	Vulnerability Type
phpns 2.1.1alpha	4224	30	40	13	Authz/Authc
DCPPortal 5.1.44	89074	362	308	34	param tampering/Authc/Authz
myBloggie 2.1.3	6261	59	24	5	param tampering/Authc/Authz
miniBloggie 1.1	1283	11	5	2	Authc/Authz
SCARF 1.0	978	19	13	7	Authc/Authz
SnipeGallery	9.1k	37	25	3	param tampering
SPHPBlog	26.5k	125	122	11	param tampering
PHPNews	6.4k	20	57	6	param tampering
Landshop	15.4k	88	541	9	param tampering

4.4 Evaluation

LOGICPATCHER, designed for patching PHP web applications, is implemented in Java in about 1.5K lines of code. We use open-source tools and libraries (TAPS (27) and Pixy (26)) to get the control flow graphs for PHP applications. The experiments described in this section were performed on a MacBook Pro (2.4 GHz Intel, 4.0 GB RAM).

We examine 9 open-source PHP applications which are summarized in Table VI. The test suite was picked from reported logic vulnerabilities (39; 38; 43) in PHP web applications from previous research studies, which also gave missing conditions and path locations. The results of our evaluation are categorized into three subsections: (1) the generated patches by LOGICPATCHER and sample patch and patch locations suggested by our tool, (2) the precision of LOGICPATCHER on generation and placement of the patches, and (3) scalability of our approach.

4.4.1 Candidate Patch Locations

myBloggie is a simple blogging application which lets users add, delete and update blog posts to/from a database. The application has two types of security issues: access control and parameter tampering

vulnerabilities. There are six privilege escalation problems in `myBloggie`, which allow an unauthenticated user to delete and update blog posts. LOGICPATCHER generated a patch which checks the validity of the session variables and the level of the user `$userid['level']`. The patch includes termination instructions if the security check is not met.

miniBloggie is a blogging software with standard features such as adding, deleting, and updating blog posts and comments. This application has one vertical privilege escalation(i.e., access to privileges in higher role) vulnerability. LOGICPATCHER fixed `miniBloggie` by adding the following constraint:

```
1 if (isset($_SESSION['user']) && isset($_SESSION['pwd'])){
2   $sql = "DELETE FROM blogdata WHERE post_id=$post_id";
3   $query = mysql_query($sql) or die("Cannot query the database.<br>" .
        mysql_error());}
4 else{ header( "Location: ./login.php" );}
```

Listing 4.15: Patch generated for del.php, miniBloggie

SCARF is an open-source conference management software. This application is vulnerable to both vertical and horizontal privilege escalation(same role different user). The vertical escalation vulnerability is caused by lack of authorization checks in certain files. For example, in generaloptions.php the check for administrative role is omitted which lets other users to gain access to operations in this file. Our tool generated the following patch for the file, and placed the patch before the first sensitive operation (DB query) in the file.

```
1 if ($_SESSION['privilege'] == 'admin'){...}
2 else{ exit();}
```

Listing 4.16: Patch generated for generaloptions.php, SCARF

Several horizontal escalation vulnerabilities found in SCARF are caused by use of `$_GET['session_id']` instead of `$_SESSION['user_id']`. This vulnerability happens at the query location in the access parameters used in the WHERE clause. Therefore, for the database columns which were identified to hold the `$_SESSION['user_id']`, LOGICPATCHER augmented the WHERE clause in the vulnerable queries with the necessary constraints on the column values. For the WHERE clauses in which the column was present but the value was set to `$_GET['session_id']`, LOGICPATCHER replaced the value with `$_SESSION['user_id']`.

DCPPortal is an open-source content management system which is vulnerable to both vertical and horizontal privilege escalation. The reason for vertical privilege escalation vulnerabilities is the use of cookies from untrusted sources. LOGICPATCHER suggested using session variables instead of cookies, preserving a one-to-one correspondence between the cookie and the session variables (username, role, permissions, etc). We also generated the program slices with necessary computations for each of the new session variables. For example, for checking the admin role we added:

```
1 session_start();
2 if($_SESSION"dcp5_member_id"] == 5){...}
```

Listing 4.17: Patch generated for DCPPortal

SnipeGallery a photo album application, allows users to arrange albums hierarchically by selecting a parent category for each new album from a drop down list. By selecting a value not in that list, the new album becomes invisible. LOGICPATCHER generated the patch to check for the availability of the values in the list, before performing any sensitive operations, preventing the vulnerability.

SPHPBlog is a blog system. It uses files to store blog posts, comments, rating, etc. This application is vulnerable to parameter tampering as it does not check if the user selects values from the drop-down list, or if it is an arbitrary value. Entered values are stored in various files, which is a security threat to the server LOGICPATCHER generated the patch to check the values for the drop-down lists.

PHPNews is a news management software and is vulnerable to parameter tampering attacks. In admin.php, the application allows administrators to modify certain files through a form which contains name of the file as a hidden field. The server-side code fails to validate that the file name is not tampered and as a result attackers can update existing files, create arbitrary files and / or corrupt files of other applications deployed on the same web server.

Landshop is a real estate application which is vulnerable to both parameter tampering and horizontal privilege escalation attacks. This application includes a form with a hidden field not relevant to that form. When the value of this field is set to the ID of an existing listing (which are displayed prominently on the site), that listing is deleted from the application whether the user is the owner or not. LOGICPATCHER patched this application by 1) augmenting the WHERE clause query to include the ownership constraints and 2) generate the program slice to create the ownership value.

`phpns`, In `phpns`, application, we have a case of non-disjoint paths because of the existence of a generic function `delete_item`. This function is used to delete rows from any table in the DB and the table (the resource) name depends on the path leading to this function. That is, in `manage.php` and `article.php` we have:

```
1 delete_item('articles',$items_f);
```

Listing 4.18: manage.php, `phpns`

```
1 delete_item('comments',$items_f);
```

Listing 4.19: article.php before the patch, `phpns`,

However, only one of these execution paths is vulnerable to vertical privilege escalation, and therefore only one of these function calls should be patched by an authorization check. In this case, we use path profiling to restrict the path which deletes comments (Listing 4.19) to authenticated users. The generated candidate patch is shown in Listing 4.20:

```
1 if($_SESSION['username']){//added security check, checks if the a valid
      username exists in the session
2   delete_item('comments',$items_f);}
```

Listing 4.20: article.php after the patch, `phpns`,

4.4.2 Effectiveness

We evaluated the effectiveness of our tool by manually inspecting the generated patches. We also compare the newer versions of applications in our test suite (if available) with our the patches. There are two aspects to the effectiveness of LOGICPATCHER: correctness of the patches and optimizing patch placement.

Correctness The correctness of the patching depends on: 1) the correctness of the generated patch, and 2) the correctness of the scope and location of the patch. We have confirmed that LOGICPATCHER could correctly patch 27 of 29 vulnerable files in 9 web applications in our test suite. As discussed in the Limitations subsection in Section 4.2, cascading sinks are one of the reasons LOGICPATCHER might generate incorrect or inconsistent patches. The following example in `Scarf` application shows the code before and after the patch:

TABLE VII: APPLICATION COMPLEXITY

Application	# of paths	# of Vulnerable Files	# of Entry-point Locations	Analysis time (s)
phpns 2.1.1alpha	709	1	21	3759
DCPPortal 5.1.44	588	12	210	2452
myBloggie 2.1.3	98	7	45	1620
miniBloggie 1.1	14	1	6	732
SCARF 1.0	86	3	16	250
SnipeGallery	530	1	32	2415
SPHPBlog	251	1	76	5613
PHPNews	36	2	10	179
Landshop	362	1	44	1205

```
1 if (isset($_POST['paper_id'])) {
2   query("UPDATE papers SET title='$title', abstract='$abstract', $pdfSetString
        session_id='$session' WHERE paper_id='$id'");
3   query("DELETE FROM authors WHERE paper_id='$id'");}
```

Listing 4.21: editpaper.php, Scarf Application before the patch

```
1 if (isset($_POST['paper_id'])) {
2   if($_POST['authors'] == $_SESSION['user'] && PATH == <hash-path1>){
3     query("UPDATE papers SET title='$title', abstract='$abstract',
          $pdfSetString session_id='$session' WHERE paper_id='$id'");}
4   if($_POST['authors'] == $_SESSION['user'] && PATH == <hash-path2>){
5     query("DELETE FROM authors WHERE paper_id='$id'");}}
```

Listing 4.22: editpaper.php, Scarf Application after applying the candidate patch

As it is shown in Listing 4.22, because two different sinks exist in the same path and they are not disjoint, the path profiling procedure will create two different hash values for the functions, and each sink is wrapped in a different set of conditions. However, these two queries should come together to preserve the consistency of the DB data.

Optimization The second aspect of the effectiveness of our tool is to check whether the patch placement was optimal. For the same missing condition *C*, our tool should be able to find the best location to inject *C* so that multiple vulnerabilities are prevented. Currently, the path enumeration and path profiling procedures use the information about vulnerable paths and missing conditions one at a time. If the same missing condition causes multiple vulnerabilities and paths are disjoint from correct paths, optimal patches are generated. However, if any interference exists in the code, then the path profiling procedure may suggest several patch locations for multiple vulnerabilities for the same cause. We plan to address optimizing of multiple patch placement in future work.

4.4.3 Scalability

We evaluate LOGICPATCHER on variety of web applications with sizes ranging from 1K to 90K. as shown in Columns 2-3 in Table VI show the size and number of PHP files in the applications, and column 4 gives an estimation of the number of sink locations (query locations, file operations, etc) in the source-code of the applications. Column 5 in Table VI shows number of resources to be analyzed. By resources, we mean the number of DB queries in case the vulnerabilities are related to DB operations and number of different files when the sink type is file operation.

Table VII shows statistics about the complexity of each application. In particular, it shows the total number of paths (column 2), the number of files which are vulnerable and need to be patched (column 3). Column 4 shows the number of program entry points, which together with the number of paths affect the patch placement process. That is, if the number of paths increase while the number of entry points remain the same, the number of disjoint paths would decrease. This in turn will increase the analysis times which are shown in column 5 of Table VII. About 90% of the analysis time is spent on path

enumeration which is used by the patch placement module. This analysis is only needed once and it does not add any overhead to the application execution at run-time.

4.5 Summary

This Chapter discussed retrofitting of logic vulnerabilities, which are an important class of programming flaws in web applications. The challenge in retrofitting vulnerabilities in web applications is to patch the vulnerable locations without changing the functionality of other components in the applications. We address this challenge by developing an approach and tool called LOGICPATCHER for patching of logic vulnerabilities. We focus on correct *patch placement*, i.e., identifying the precise location in code where the patch code can be introduced, based on path profiling. We showed that identifying the appropriate patch location as well as generating the right patch can get complicated and require deep code analysis. We demonstrated the utility of LOGICPATCHER by testing it on large vulnerable web applications, and we were able to successfully patch 27 of the 29 vulnerabilities.

5

SYNTHESIZING SECURE CODE FOR WEB APPLICATIONS

Previously published as Nazari Skrupsky, Maliheh Monshizadeh, Prithvi Bisht, Timothy Hinrichs, V.N. Venkatakrishnan, and Lenore Zuck. "WAVES: Automatic Synthesis of Client-side Validation Code for Web Applications". In ASE Science Journal Vol. 1, Issue 3, pp. 121-136, Dec. 2012.

Current practices in mainstream web development isolate the construction of the client component of an application from the server component. Not only are the two components developed independently, but they are often developed by different teams of developers. The client component is often written using a different programming language and platform (HTML and JavaScript in a web browser) than the server (e.g., PHP, Java, ASP), therefore necessitating developers with different skill sets. When the client and server are supposed to share application logic but do not, an “impedance mismatch" occurs.

In this chapter we are concerned with a specific kind of application logic: the input validation logic. Input validation logic is the set of predicates, which are related to the input values, and should hold during computation and storage operations. Examples of input validation include input character validation (“username does not contain special characters”), required fields (“phone number is required”) and logical checks (“credit card expiry date in past”).

Input validation on the client improves the user experience because it provides the user immediate feedback about errors; furthermore, it often reduces network and server load. Input validation on the server is necessary for security. For if the server assumes all the data it has been sent has been validated by the client, a malicious user can circumvent the client, submit invalid data to the server, and exploit the lack of server-side validation, leading to Cross-site Scripting (XSS) and SQL Injection (SQLI) attacks. has uncovered impedance-mismatch vulnerabilities that enable takeovers of accounts and unauthorized financial transactions in commercial and open-source websites as well as third-party cashiers (such as `PayPal` and `Amazon Payments`).

Existing web application frameworks offer some support for client code synthesis. For example, Google web toolkit (46) offers a way by which programmers can write new applications with a common module (in Java) for the server and client, and client JavaScript code is automatically generated from this common module specification. However, the burden of identifying validation logic rests entirely on the programmer.

Compared to frameworks, a more challenging problem exists for legacy applications. In this case, we are aware of no methodologies or tools that provide automatic support for automatic client code synthesis or validation logic identification. Augmenting such a legacy application therefore involves manual effort by the programmer, and often leads to independent development and therefore has the potential to lead to security vulnerabilities.

For legacy applications manually retrofitting them with extensive client-side and server-side validation is error-prone and expensive, especially since the client and server validation must be synchronized every time the application is updated.

In this Chapter, we would like to address the impedance mismatch problem for legacy web applications that have no interactive client-side input validation. Our approach is to automatically examine the source code of a web application, identify the server-side input validation logic (predicates), and replicate that logic on the client. While designed for legacy applications, our approach can also be deployed in modern web development frameworks, thereby enabling a developer writing a new application to author only the server-side validation code while the framework automatically installs the corresponding client-side validation code. While such technology is most obviously beneficial because it simplifies a web developer's job, it can also help to improve the security of newly written applications. Thus our approach has several high-level benefits:

- *Improved Security.* The development team can devote more resources to the design and implementation of the server code, thereby being more likely to include all the input validation necessary for the application's security.
- *Improved Usability.* Applications whose client input validation has been automatically generated provide end users with all the input validation expected of today's web apps.
- *Greater Development Efficiency.* Developers no longer write the same validation code twice since the client code is automatically synthesized from the server code.

Our realization of this approach, WAVES (103; 102), uses program analysis to automatically extract a logical representation of the input validation checks on the server and then synthesizes efficient client-side input validation routines. Of particular note is that WAVES also generates code for validation checks that involve server-side state by utilizing asynchronous requests (AJAX) to perform the required

validation. Because such validation routines can increase server and network load, WAVES allows a developer to choose the extent to which such validation checks are generated.

This chapter is organized as follows: Section 5.1 presents the problem by means of a running example and the challenges our approach must overcome. In Section 5.2 we present a high level overview of our approach. In Section 5.3 we present a detailed description of the different components of our approach. Section 5.4 presents an evaluation. We evaluate our approach and tool over three real-world web applications, and in Section 5.5 we conclude. Our experience indicates that our approach offers a promising improvement to current mainstream web development practices

5.1 Running Example and Challenges

Figure 10 presents the client side interface of a simple user registration application. We will use this application as the running example throughout the chapter. In this application, a user supplies her user ID and her password twice (for confirmation purposes). There are three validation checks performed by this application:

1. The characters in user ID belong to a specified character set, which in this case is all alphanumeric characters along with a hyphen and underscore.
2. The two supplied passwords match.
3. The user ID is available for creating an account (i.e., it is not already taken by another user).

Suppose the developer authors the server component of the application and implements these checks in server code. Our goal is to *automatically synthesize* the corresponding client side input validation routines. The high-level challenges in achieving our goal include:

Figure 10: Running Example of A Registration Form

- *Automatic inference of server-side constraints*. While the client side validation constraints are expressed in terms of form fields, the server side validation may be performed in terms of server-side variables within deeply nested control flows of the application. The server-side constraints must be extracted and expressed in terms of the form fields.
- *Validation involving the server*. Sometimes validation involves server-side state (such as the database), but moving that data to the client is often impractical because of performance, security, privacy, and/or staleness issues. For example, when a user ID is submitted to the server, the server checks if the ID is unique in the database. Moving all the user IDs to the client is impractical; thus, some clients asynchronously contact the server to check if the ID is unique. The code that is generated must allow the client to asynchronously contact the server (and for the developer to control which asynchronous validations are performed).
- *Preservation of application logic and security*. The code that is generated must neither compromise the security of the application nor disable existing functionality.

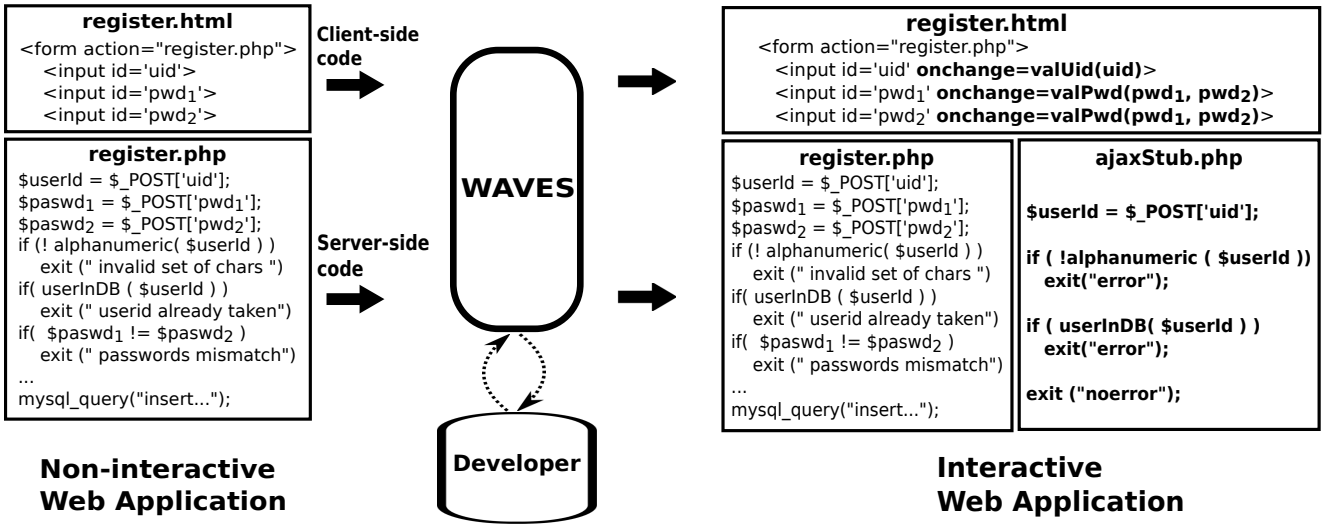

Figure 11: WAVES: Synthesizing Client-side Validation Code.

5.2 Approach

We present an approach for improving the web application development process that alleviates the problem of inconsistent client and server input validation: WAVES (Web Application Validation Extraction and Synthesis). WAVES requires developers to only maintain the input validation code on the server. WAVES then automatically synthesizes the corresponding validation code for the client. Figure 11 shows the desired transformation of the running example[1].

[1]For concreteness, the example shows the client implemented in JavaScript, and AJAX, and the server implemented in PHP. While these languages are the ones addressed by our current prototype, the underlying techniques used by our approach are agnostic to programming languages. Our implementation can be easily extended to other server platforms (e.g., JSP, .NET) and client platforms (e.g., ActionScript).

The non-interactive version of the web application is shown on the left and is comprised of the client-side code (`register.html`) and server-side code (`register.php`). Guided by validation checks in `register.php`, WAVES generates the interactive version of this application shown on the right (newly added code in bold font). The retrofitted client validates each of its three fields as soon as the data in any field changes. For instance, when the user changes `uid`, the client checks that only alphanumeric characters, hyphens, and underscores appear in the user ID; additionally, the client asks the server if the user ID is unique in the database.

WAVES breaks this transformation into four conceptually distinct phases:

(1) Server analysis WAVES performs dynamic program analysis–submitting form inputs to the server and inspecting the sequence of instructions that the server executes. The key insight is that when the server is given an input it accepts, the sequence of if-statements it executes contain all the input validation constraints it checks. So after submitting form field inputs that the server accepts and rewriting the if-statements in terms of the original form field inputs, we have a list of potential input validation constraints. We then analyze each one to determine if it is truly an input validation constraint –a constraint that when falsified causes the server to reject the input . Once the list has been reduced to the set of actual input validation constraints, we identify which constraints are dependent on the server's environment (the *dynamic* constraints) and which are not (the *static* constraints).

In our running example, we first submit legitimate values for `uid` and the two passwords. The server checks if the `uid` contains only the permitted characters, that the `uid` is unique in the database, and that the passwords match. Finally, we separate the static constraints (the alphanumeric constraint on `uid`

and the password equivalence constraint) from the dynamic constraints (the fact that the `uid` is unique in the database).

(2) Client-side code generation In WAVES, once the static and dynamic constraints have been extracted from the server, we synthesize client-side code to check those constraints. The static constraints can be checked directly by JavaScript code, but the dynamic constraints can only be checked after communicating with the server. So for each form field, we generate code that performs two tasks: checking if any errors arise because of static constraints and if not, checking if any errors arise because of dynamic constraints by asynchronously contacting the server.

(3) Server-side code generation The asynchronous messages sent by the client to check the dynamic constraints for a form field can only be responded to by special-purpose server-side code. (The original code assumes the user provided values for all form fields, but the client's asynchronous messages aim to check constraints even before the user completes the form.) These server stubs behave the same as the original server code but operate properly when data for only one or two form fields is provided. Different techniques can be used to generate server stubs, but we recommend code slicing. To minimize server communication, we also recommend checking all of the dynamic constraints for a form field via one asynchronous message.

(4) Integration Once the new client and server code has been generated, it must be integrated into the existing client and server codebases. In this step, the developers can choose to disallow some generated code parts to be integrated into the application since there are some constraints which may reveal information about the server state or data. How the integration is done depends on the programming

languages for client and server, but ideally regenerating client and server code to reflect changes in the application will require minimal additional integration effort.

5.3 Technical Description

In this section we describe each of the four phases of our approach in more detail.

5.3.1 Server Analysis

The server analysis phase of WAVES aims to discover all of the constraints on form fields that the server enforces (Algorithm 15). Besides the URL of the web form, WAVES is given inputs for the form that the web server accepts, i.e., a single error-free input. WAVES begins by submitting this initial input (the *success* input) to the server, which returns a trace of the instructions that the server executed in response (Algorithm 15 Line 1). Instrumenting the server to return such a trace is done offline and was described in prior work (29). Since the success input is accepted by the server, those inputs satisfy all of the constraints the server enforces, and consequently all the input validation constraints will appear as if-statements in the resulting server trace. By rewriting those if-statements in terms of the original inputs (using taint analysis of (29)), WAVES extracts the set of conditions that were true of the form field inputs: $\{C_1, \ldots, C_n\}$ (Line 2).

Not each of the resulting conditions, if falsified, leads to an error. Thus, WAVES next identifies which of the conditions (C_i) if falsified lead to an error. For each C_i, WAVES constructs inputs that satisfy $\neg C_i$ using a string solver (47) (Line 5) but is otherwise as similar to the original success input as possible (Line 6). The intent is that this *failure input*, if rejected by the server, demonstrates that $\neg C_i$ is an error condition. If the server rejects a failure input, we know that the conjunction of conditions in that trace (after rewriting them in terms of the original form field inputs) is an error condition: $D_1 \wedge \cdots \wedge D_m$

Algorithm 6: WAVES(url, suc_input, indep)

```
   input  : url, suc_input, indep
   output: Client validation code in JavaScript and server stubs in PHP.
1  trace := SUBMIT(url, suc_input);
2  C_1 ∧ ··· ∧ C_n := CONSTRAINTS(trace);
3  safe := PARTITION(C_1 ∧ ··· ∧ C_n, indep);
4  foreach C_i ∈ C do
5  |  bl := SOLVER(¬C_i);
6  |  bl := bl ∪ ELIMINATEVARS(suc_input, VARS(bl));
7  |  trace := SUBMIT(url, bl);
8  |  D_1 ∧ ··· ∧ D_m := CONSTRAINTS(trace);
9  |  P = PARTITION(D_1 ∧ ··· ∧ D_m, indep);
10 |  if SERVERACCEPTED(trace) then
11 |  |  safe := safe ∪ P;
12 |  else
13 |  |  errors := errors ∪ (P − safe);
14 (static, dynamic) := SPLITSTATICDYNAMIC(errors);
15 return (GENCLIENT(static),GENSERVER(dynamic));
```

(Line 7-8). That is, every input satisfying $D_1 \wedge \cdots \wedge D_m$ contains at least one error. The constraints that WAVES extracts is a collection of such error conditions (Algorithm 15 Line 13).

Simplification The algorithm described above is sound by construction (proof in Appendix C) : if WAVES finds an error condition, then any input satisfying that condition will cause an error. But in practice each of these error conditions is usually too weak to be useful because it includes checks on all of the form fields. The only time the error condition is satisfied is therefore when all of the form fields have values. One of the design goals of WAVES is to give the user real-time feedback each time she enters a new form field value, a goal that the error conditions described so far fail to achieve. To illustrate the issue, consider a failure input where the user ID satisfies the necessary conditions but where the two passwords are unequal. The above algorithm would identify the following conjunction as an error condition.

$$(uid \in [0-9a-zA-Z\_-]*) \wedge$$
$$isUnique(\ uid\ ) \wedge$$
$$(pwd_1 \neq pwd_2)$$

The problem is that this constraint can only be evaluated once there are values for all three form fields. Moreover, this constraint only ensures that if `uid` is alphanumeric and not already present in the database then the passwords must be equal. While the correct simplification of this example is obvious from our description of the application ($pwd_1 \neq pwd_2$), in general we cannot soundly eliminate conjuncts from an error condition.

The basic premise behind our simplification routine is that we have two kinds of server traces: those with errors and those without errors. The conjunction of conditions in a trace with errors is an error condition: any input that satisfies *all* the constraints is rejected by the server. The conjunction of conditions in a trace without errors is a *safe condition*: no input that satisfies a safe condition is rejected by the server. Thus, we can simplify an error condition by removing all safe conditions contained within it (Algorithm 15 Line 13).

Unfortunately, it is just as important and difficult to simplify a safe condition as it is to simplify an error condition. All we know is that no input satisfying all the conjuncts together causes an error. But if WAVES knows which form fields are independent of which others in terms of all control paths (the $indep$ argument to Algorithm 15), it can break large safe conditions and error conditions into independent conjunctions of constraints (Lines 3, 9). WAVES then records each independent conjunction of constraints as either a safe condition (Line 11) or as an error condition (Line 13). Any error condition

that is also a safe condition is eliminated as an error condition (Line 13). We found this independence information crucial to generating practically useful error conditions.

Static and Dynamic Constraints The constraints WAVES extracts from the server are one of two kinds: static or dynamic. Dynamic constraints depend on the server's environment (e.g., file system or database), while static constraints do not. The difference is important because static constraints will never change and hence can easily be synthesized on the client, but dynamic constraints change each time the server's environment changes and hence for correctness can only be checked by the server. The way WAVES identifies dynamic constraints is straightforward: any constraint referencing the server's environment (e.g., the database, files, sessions, global variables, time, etc.) is a dynamic constraint; all others are static (Algorithm 15 Line 14).

Discussion One of the limitations of server-side analysis is that if the constraints enforced by the server are complex enough, it may be that a single success input is insufficient to extract all of the constraints enforced by the server. While we did not encounter this limitation in the applications we evaluated, to address such forms we would apply the algorithms we developed in prior work to construct additional success inputs automatically (29).

5.3.2 Client-side Code Generation

Generating the client code to check a collection of static and dynamic constraints is broken into two distinct components: generating code that checks the static constraints and generating code that checks the dynamic constraints. Recall that the static constraints can be checked directly on the client, and the dynamic constraints require communicating with the server. For each form field, WAVES generates an

event handler that first checks the static constraints for an error and if none is found then checks the dynamic constraints.

Static constraints Each static constraint is basically a conditional test on form fields that can include any number of string and integer manipulation functions (e.g., $len(trim(x)) > 6$ ensures the length of field x after removing whitespace from both ends is greater than 6). Formally, each constraint is represented in the logic of strings and integer arithmetic.

Given the static constraints that must hold of the form, we must identify which constraints are pertinent to each form field so that each time that form field changes we can check the right constraints. Choose too many, and the user may see error warnings for form fields that she has not even filled in; choose too few, and she will not be warned of errors when they exist. This identification is quite simple after converting the constraints to a canonical form (conjunctive normal form): for form field f collect all those constraints where f occurs.

There are some static constraints which may reveal secret information about the server. For example, the constraint `password == "secret"` (revealing the hard-coded password "secret", which is a poor security practice) should not be added to client-side code. These constraints occur rarely, and we have not encountered any warnings of this type. The string solver can recognize constraints in which a form field value is checked against a constant value, however it cannot identify whether this constant value is a server-related secret. Therefore, the developer should choose to allow these type of static checks to appear on the client-side or not.

Generating client-side code that checks the constraints for a given form field is a linear time and space procedure, assuming the client has implementations of all the string and integer functions.

Dynamic constraints A dynamic constraint is essentially a static constraint, which is additionally deemed to be volatile. More precisely, constraints which directly involve the server's environment (e.g., session data, database and file operations) are classified dynamic. Nested constraints are also considered dynamic when present within the scope of a dynamic condition. Because the server's environment may change from the time a form is generated to the time it is submitted (e.g., the set of available user names changes), dynamic constraints can only be checked by consulting the server.

To this end, WAVES generates and makes use of *server-side stubs*, which check dynamic constraints on the server (described in §5.3.3). When the client needs to check a form field with a dynamic constraint, it communicates with the server asynchronously. The client-side code for checking dynamic constraints consists of sending requests with form field values to the server and processing status changes from the server's responses into real-time feedback for the user.

Triggering Validation Once the client-side code is generated, we must instruct the client to execute that code at the appropriate time and inform users when constraints have been violated. For modern web clients, it is usually a simple matter to provide snippets of code to be executed for each of a fixed number of events (e.g., each time the user changes the *uid*). Thus it is a simple matter to tell the client to run the code that checks the appropriate constraints each time a form field changes and provide error messages when appropriate.

5.3.3 Server-side Code Generation

The main goal in this step is to create server code that responds to an asynchronous client request to check the dynamic constraints for a given form field. That code invokes a stub for each of the dynamic

constraints extracted by WAVES. If any of the stubs produces an error, the server returns an error. Stub generation is a three-step process, which we explain below with our running example.

Dependency Analysis Given a dynamic constraint in the server code, WAVES first performs a data and control dependency analysis to compute the set of all program variables (not just form fields) on which the dynamic constraint depends (either implicitly or explicitly). We call these the *related variables*. We do this via backward analysis, starting from the dynamic constraints and working backwards in the server code. In the running example for the dynamic constraint `userInDB( $userId )`, the set of related program variables includes `$userId` and `$_POST['uid']`.

Program Slicing WAVES then employs off-the-shelf program slicing techniques (48) to generate the server stubs. More precisely, we begin at the top of the code and prune out any instructions not relevant to the related variables, stopping once we reach the dynamic constraint. The efficiency of the resulting stubs is a direct consequence of how effective our pruning of the server code is. Prune too little, and the stub is inefficient; prune too much, and the stub is unsound. Our pruning process was guided by the following three criteria.

First, the server stub includes all those instructions that the result of the dynamic constraint depends on. All assignments that have a related variable on the left hand side are retained in the server stub. For our running example, this ensures the assignment `$userId = $_POST['uid'];` is not pruned from the stub. Second, the server stub includes environment variables and functions that affect these variables, such as functions that read or write session values. These functions and variables may indirectly change the control flow of the server code. Third, some instructions change the state of the server while executing, e.g., inserts and updates in database operations, database schema changes, writing to files, as

well as changing and setting session and cookie variables. Including statements with side-effects can lead to inconsistent server state, since the user has not actually submitted the form, but excluding such statements can lead to security vulnerabilities (e.g., an application outfitted to defend against denial-of-service attacks by logging IP addresses and dropping large bursts of requests from a single IP). Thus, WAVES allows a developer to choose whether statements with side-effects are allowed in stubs or not. If side-effects are not allowed, and a stub includes a side-effect after pruning, that stub is eliminated and the dynamic constraint is not checked. Note that failure to check a dynamic constraint is a source of incompleteness, not unsoundness. In addition, none of our test applications (§5.4) required allowing the use of side effects.

Simplification and Optimization There are some cases in which constraints on *unrelated* form fields may appear in a server stub. This happens because of control dependencies introduced by if-else constructs in the server code, which will cause unwanted errors. As discussed in Section 5.3.1, we can alleviate this problem by using independence information for the form fields.

5.3.4 Integration

WAVES is designed to incorporate client side validation code in new as well as legacy applications. In the previous steps, WAVES generated the code necessary to enable client-side validation of user inputs. The integration of this generated code in an application requires minimal changes to the application's codebase. Installing the server code only requires uploading it to application's directory on the server. Installing the client code is almost as easy—it simply requires augmenting the client's source code to include the JavaScript file containing the generated code. Thus when that file is loaded by the browser, it attaches all the event handlers to appropriate fields to perform validation.

5.4 Evaluation

Implementation. The server-side analysis is implemented in Java and Lisp and builds upon our prior work WAPTEC (29) as well as the state-of-the-art SMT solver Kaluza (47). The client-side code generation is implemented in LISP and Java and builds on Plato (49) (a web form generator), php.js (50) (a library of PHP functions implemented in JavaScript), and the jQuery validation module (51). The server-side code generation is implemented in Java and builds on Pixy (26) (a tool for PHP dependency analysis).

Test suite. We selected three medium to large and popular PHP applications. The application test suite was deployed on a Mac Mini (1.83 GHz Intel, 2.0 GB RAM) running the MAMP application suite, and the WAVES prototype was deployed on an Ubuntu virtual machine (2.4 Ghz single core Intel, 2.0 GB RAM).

Experiments We chose one form in each of the three applications. Two of the chosen forms (`B2Evolution` and `WeBid`) do not contain any client-side validation; the other form (`WebSubRev`) already includes client-side validation. The first two forms allowed end-to-end testing of our prototype tool while the third form allowed us to compare WAVES synthesized code with validation code written manually by developers. We discuss our experiments and experiences below.

5.4.1 Effectiveness

For each of the selected forms, we first manually analyzed the server-side code for processing the chosen form and identified the constraints being checked — we call this the “ideal" synthesis and use it to assess effectiveness of WAVES. For each application, Column 2 of Table VIII shows the ideal number of constraints (static + dynamic). Static constraints, those that do not rely on server-side state,

Application	Ideal Synthesis	WAVES Synthesis	False Negatives	False Positives	Existing Validation
B2Evolution	10+1	7+1	3	0	0
WeBid	17+8	16+6	3	0	0
WebSubRev	5+1	4+1	1	0	5+0

TABLE VIII: WAVES SYNTHESIZED OVER 83% CONSTRAINTS SUCCESSFULLY.

dominated the total number of constraints synthesized by WAVES (27 / 35). As shown in the next column, WAVES was able to synthesize over 83% of the constraints identified by the ideal synthesis.

False Negatives WAVES suffered from a small number of false negatives due to missed constraints (Column 4 of Table VIII). Constraints that WAVES failed to synthesize were those it failed to extract during the server analysis phase. One of the problems encountered was that WAVES generated form field inputs intended to detect whether or not a particular constraint leads to an error, but the form field inputs happened to falsify a different constraint, hence WAVES never inferred the original constraint that caused an error. For example, a constraint in `WeBid` required the e-mail field to include the @ character while another constraint required the e-mail field to satisfy a regular expression. WAVES was unable to uncover the regular expression constraint, because the input used to test if the regular expression constraint was actually an error condition so happened to include no @, therefore, the server rejected due to the first constraint and not the second. We attempted to avoid this problem by generating inputs that satisfy the combination of the two constraints, where one was negated and the other was not, but found that such constraint sets were often too complex for Kaluza to solve efficiently.

The second reason for missing constraints was a fundamental mismatch between the constraints we needed to solve and the language supported by Kaluza. For example, the PHP function `explode` takes a string and splits that string into an array of strings. Since Kaluza does not implement the theory of arrays, we could not encode `explode` into its constraint language, and hence simply ignored any constraint with `explode`. We expect that as SMT solvers that support the theory of strings mature (there have only been two developed to date), many of these issues will be overcome, and the results for WAVES will improve as a consequence.

False Positives Cases where the synthesized client ends up rejecting inputs that the server actually accepts are considered to be false positives (Column 5 of Table VIII). In our experiments, we did not encounter any false positives; however, we discuss at least one conceivable case that could cause false positives. When input validation is performed inside a loop, the number of iterations can influence the constraint that gets extracted from a particular trace. For example, the constraint extracted from a loop that iterates over the characters of an input of length n will check exactly n characters each time regardless of the subsequent lengths. In this case, any input whose length is not the same would be rejected by the client. Properly handling this type of validation contained within loops would require assistance from developers in the form of loop invariants. An automatable approach is to discard constraints that are derived from within loops. We would like to note that such a solution would decrease false positives at the expense of increasing false negatives – an advantageous tradeoff which would produce all the benefits of client validation without any impedance of usability.

Form Interactivity One of the benefits from using WAVES is that forms retrofitted with interactivity should improve the overall usability of the application. A synthesized client provides instant feedback

as the user interacts with the form. For example, when the user inputs valid data, a green check mark will appear next to the form field; conversely, invalid data will appear next to a red X, and an error message will convey the mistake.

Applications that rely solely on the server to validate form input can be discouraging for the end-user. For example, in the `WeBid` application, we noticed that the server sends a single error message at a time. This particular form contains 25 constraints, so the user may need to resubmit that many times–correcting a single invalid value each time. This problem is eliminated when WAVES introduces validation into the client, because by the time the user submits the form, the values will already be error-free.

Improved Performance The above `WeBid` example also illustrates that insufficient client-side validation can cause repeat submissions, which result in additional server workload and bandwidth use. In the original form submission logic, whenever the user commits an error she needs to retransmit all form data to the server, and the server needs to reprocess the input. Since WAVES effectively offloads validation onto the client, the server spends less resources on form processing, and the overall performance of the application improves. In general, the reduction of resource consumption at the server is expected when most of the constraints are static, but if there are many dynamic constraints, our approach could have the opposite effect. In our experiments, we observed over 75% of form fields have no dynamic constraints; moreover, WAVES allows the developer to choose which form fields to outfit with dynamic constraint checks.

Application	Formula Complexity	Time (sec)	Average Stub Size(KB)	Average Stub RT(ms)
B2Evolution	52+9	522	0.7k	23
WeBid	17+18	281	1.1k	104
WebSubRev	25+0	12921	0.9k	117

TABLE IX: PERFORMANCE MEASURES

5.4.2 Synthesized Code vs. Developer Written Code

We also compared the code WAVES synthesized with code written manually by application developers. The third application in our test suite, `WebSubRev`, rejected invalid inputs by employing JavaScript. For this form, the server-side code checked 6 constraints (Column 2 in Table VIII), and the developer written client-side code checked 5 constraints (all of which were static). WAVES generated 4 static constraints and 1 dynamic constraint, therefore synthesizing 80% of the static constraints and 100% of the dynamic constraints.

The one static constraint that WAVES could not synthesize was a regular expression check on an array obtained from the `explode` function, which as described previously was problematic for Kaluza. The one dynamic constraint discovered by WAVES but not included in the manually written client dictates which filename extensions are accepted by the server. This constraint was not included in the manually written client because (i) the list of permitted extensions is stored in the database and (ii) the constraint is only checked by the server when the administrator has configured the application so that the file field is mandatory. Checking this constraint dynamically can yield a potentially large savings since before a potentially large file is transmitted to the server, the form can warn the user about an

improper file type, thereby saving a potentially lengthy wait for the user while the file is transmitted over the network. The server and network also benefit from decreased loads.

5.4.3 Other Experimental Details

We evaluated WAVES prototype on our test suite and recorded various performance measures during execution (Table IX). In the offline phase, when WAVES performs code analysis, client and server code generation, and installation, we measured the formula complexity of static and dynamic constraints. The second and third columns show static and dynamic formula complexities, which are the total number of boolean operators and atomic constraints. The total time taken by WAVES to extract the formula and synthesize the client is shown by the fourth column. We noted that WAVES spent most time in either analyzing traces or solving constraints. Because WAVES is designed as an offline program transformation tool, even if these numbers are not reduced via additional system engineering, they should be acceptable in many situations. For each dynamic constraint, WAVES synthesized an AJAX stub. As shown in the fifth column, the generated stubs were much smaller in size than the portion of the application relevant to validation – in most cases less than 25% of the original LOC (stub sizes measured in effective Lines of Code using CLOC (52)).

Once WAVES finishes execution and the results are installed, the application is ready for production. The seventh column of Table IX shows average round trip time taken by stubs in responding to AJAX requests. The round trip time averaged in the range of 43 to 164 milliseconds. For comparison, the sixth column shows the average round trip time taken between client and server when users submit the full form. We believe that in real deployment scenarios such overheads are acceptable as user interactions typically last in the order of a few seconds and will overshadow delays associated with AJAX requests.

5.5 Summary

The current practice of web application development treats the client and server components of the application as two separate but interacting pieces of software. Each component is written independently, usually in distinct programming languages and development platforms — a process known to be prone to errors when the client and server share application logic. When the client and server are out of sync, an “impedance mismatch" occurs, often leading to software vulnerabilities as demonstrated by recent work on parameter tampering.

This chapter outlines the groundwork for a new software development approach, WAVES, where developers author the server-side application logic and rely on tools to automatically synthesize the corresponding client-side application logic. WAVES employs program analysis techniques to extract a logical specification from the server, from which it synthesizes client code. WAVES also synthesizes interactive client interfaces that include asynchronous callbacks whose performance and coverage rival that of manually written clients while ensuring no new security vulnerabilities are introduced. The effectiveness of WAVES is demonstrated and evaluated on three real-world web applications.

6

PREVIOUS WORK

Parts of this chapter have been published as:

–Maliheh Monshizadeh, Prasad Naldurg, V. N. Venkatakrishnan. MACE - Detecting Privilege Escalation Vulnerabilities in Web Applications. In Proceedings of 21st ACM Conference on Computer and Communications Security (CCS'14), Scottsdale, AZ, 2014.

–Maliheh Monshizadeh, Prasad Naldurg, V. N. Venkatakrishnan. Patching Logic Vulnerabilities for Web Applications using LogicPatcher. In Proceedings of The 6th ACM Conference on Data and Application Security and Privacy (CODASPY) 2016, New Orleans, LA, 2016.

–Nazari Skrupsky, Maliheh Monshizadeh, Prithvi Bisht, Timothy Hinrichs, V.N. Venkatakrishnan, and Lenore Zuck. "WAVES: Automatic Synthesis of Client-side Validation Code for Web Applications". In ASE Science Journal Vol. 1, Issue 3, pp. 121-136, Dec. 2012.

This chapter covers the related research in the area of web application security. We categorize previous work into three broad approaches: 1) techniques for prevention of the vulnerabilities, 2) techniques for detection of vulnerabilities, and 3) techniques for synthesizing secure code for web applications. While we concentrate on web applications, we also discuss similar generic software security solutions outside the realm of web applications if necessary.

6.1 Techniques for Prevention of Vulnerabilities

Rather than trying to find and fix the vulnerabilities, the prevention techniques focus on the enforcement of the security policies at runtime.

Nemesis (30) enforces authorization properties at runtime by using Dynamic Information Flow Tracking (DIFT) to establish a shadow authentication system that tracks user authentication state. Access control lists can be specified by programmers, which help the system in enforcing them at runtime. Ganapathy et al. (53), add checks to enforce authorization rules in legacy software systems, such as X SERVER. They use a reference monitor for enforcing defined authorization policies at runtime.

CLAMP (54) uses virtual web servers to prevent authorization vulnerabilities in web applications: by migrating the user authentication module of a web application into a separate, trusted virtual machine (VM). All database access requests (queries) are mediated by a trusted VM that enforces defined access control rules and restricts the queries if necessary. Diesel (55) provides a proxy-based framework to limit database accesses at runtime. It uses the principle of least-privilege to secure the database through developer-defined policies.

Capsules (56) develops a language-based technique which uses Object-Capability languages to isolate objects from each other, in order to separate web applications into components. All application state, including capabilities to application-specific resources, are stored in a per-session data store. Component isolation in Capsules ensures that application components are not able to tamper with each other and a component is not able to escalate its privilege by invoking functionality provided by another component. Capsules encourages using this capability model in developing new applications. In contrast to

MACE, these techniques seek to re-engineer web server code, maintaining separate components/VMs and changing trust assumptions.

Swaddler (57) is a dynamic anomaly detection tool which is able to detect several types of bugs including workflow bugs. Swaddler has a Daikon-based invariant learning phase (14) followed by an analysis phase which checks the invariants against the application state model. Although the Swaddler tool uses the notion of session and checks for the presence of session variables in execution paths, it does not take into account the access control model of the application with respect to various resources.

While these works are focused on *dynamic prevention* of vulnerabilities, MACE is focused on *static detection* of access control vulnerabilities, and LOGICPATCHER uses static analysis of the execution paths to precisely retrofit the vulnerabilities.

6.2 Techniques for Detection of Vulnerabilities in Legacy Web Applications

The problem of finding vulnerabilities in web applications has been studied originally in the context of e-commerce validation logic. Vulnerability detection is a crucial step in the security analysis of an application, whether we use detection to prevent or to retrofit the application.

Based on the program specification provided to detection tools, prior work towards detecting of vulnerabilities can be categorized into two categories: 1) model checking techniques in which, given pre-defined patterns and program invariants, the tool finds the anomalies; and 2) inconsistency analysis techniques in which a differential analysis is performed on two software components to find the potential vulnerabilities.

In the first category, we have Waler (33), which uses modeling checking combined with static analysis (34) to detect a wide range of logic vulnerabilities. Waler (33) uses a combination of static and

dynamic analysis techniques to extract program specifications in terms of *likely invariants* and then uses model checking to verify the extracted invariants. MACE is similar in its objectives to Waler as both approaches aim to work with source code as the only specification. However, Waler is more focused on logic errors. This limits its ability to identify access control discrepancies that require global reasoning across the entire web application, especially related to how a particular resource is accessed in various operations. MACE computes a more precise authorization contexts and is able to detect horizontal privilege escalation vulnerabilities.

Engler et al. (58) also try to extract program specifications, through behavioral patterns called *beliefs*. They use static analysis techniques to infer these patterns and rank them using statistical analysis of the patterns. The patterns specified can be used to detect certain types of vulnerabilities caused by inconsistency in the programs, such as pointer dereference and use of locks on resources.

Application Inconsistency Vulnerabilities (AIVs), as a subset of logic vulnerabilities, exist because of inconsistent development of applications. The inconsistencies may arise from different origins, whether it is caused by a mismatch between client and server-side code, or whether it occurs because of dissimilarities between developer's specified policies.

Inconsistency analysis approaches detect vulnerabilities (AIVs) by finding inconsistent design or implementation components. RoleCast (38), Srivastava et al. (59), (39) and (40) are among the research projects which use these techniques. Table X summarizes the research tools based on their approach and type of vulnerability they find.

In terms of finding security logic errors, RoleCast (38) is one of the first works for web applications, using patterns to model authorization requirements and check if any sensitive operations are performed

TABLE X: INCONSISTENCY CHECKING ANALYSIS TOOLS

Tool	Description
JIGSAW (60)	Detection of resource access inconsistencies
MACE (39)	Detection of authorization & authentication inconsistencies
RoleCast (38)	Detection of authorization & other logic inconsistencies within authorization roles
WAPTEC (29)	Detection of client and server input validation inconsistencies
NoTamper (36)	Black-box detection of client and server input validation inconsistencies
Viewpoints (37)	Detection of client and server input validation inconsistencies
AutoISES (40)	Detection of bugs in C libraries by finding security pattern inconsistencies
Srivastava et al. (59)	Detection of vulnerabilities in Java APIs by finding inconsistent API implementations

after authorization. Relatedly, MACE and AutoISES (39; 40) look at conditions along program paths to detect inconsistencies. MACE (39) employs a precise and fine-grained authorization model that is supported by user annotations, comparing the consistency of checking conditions across different requests to the same resources along different code paths, giving it the ability to detect a larger class of vulnerabilities. AutoISES (40) can detect bugs in standard C libraries through mining for common security-related patterns and identifying deviations from these as vulnerabilities. Srivastava et al. (59) detect security vulnerabilities through comparing different implementations of the same API using security policies as inputs. Any inconsistency between the security policy and any of the implementations or between different implementations are reported as errors.

Blackbox approaches (NoTamper (36) and the approach proposed by Pellegrino et al. (61)) have some potential to reason about access control vulnerabilities in an application, but they are inherently limited in their ability to reason about authorization errors that manifest as a result of missing checks along specific paths present in source code which can only be effectively gleaned through access to the application source code.

Input validation inconsistency detection approaches (36; 29; 37), try to find the injection attacks (e.g., SQLI, XSS) through comparison of the client and server side sanitization logic. While NoTamper (36) is a black-box approach to finding vulnerabilities, WAPTEC(29) and Viewpoints(37) are white-box analysis tools which analyze PHP and Java web applications respectively.

RoleCast (8) uses common software engineering patterns to model authorization requirements and develops techniques to check if any sensitive operation is performed after authorization. While the advantage of using patterns is that it frees the need for developer annotations, we have noticed that the RoleCast patterns do not hold consistently across all web applications. The approach proposed by Sun et al. (62) detects vertical escalation vulnerabilities using static analysis. This approach builds a sitemap of the web application, modeling the accesses to *privileged* webpages per role. It then checks if forced browsing causes the privileged pages to be accessed.

Both approaches ((8) and (62)) use coarse-grained modeling of authorization requirements through grouping the roles. They only accommodate detection of vertical escalation vulnerabilities. In contrast, MACE employs a precise and fine-grained authorization model that is supported by user annotations of modest effort, giving it the ability to detect a larger class of vulnerabilities, including horizontal privilege escalation.

Doupe et al. (63) present an analysis of Execution after Redirect (EAR) vulnerabilities in web applications. They discuss a static control flow analysis for web applications that detect EAR attacks. While MACE is not built to detect EARs, the analysis infrastructure of MACE could be extended in a straightforward way to detect EAR vulnerabilities. In addition, the context inference for sinks in MACE could form the basis for automatically distinguishing benign EARs from vulnerable EARs.

6.3 Techniques for Synthesis of Secure Code

We broadly divide the secure code generation work related to WAVES and LOGICPATCHER into three categories: a) patching logic vulnerabilities applicable to legacy applications, b) input validation synthesis for legacy applications, and c) applicable to newly written code. For each category, we discuss the introduction of interactivity and its security implications.

6.3.1 Retrofitting Vulnerabilities in Legacy Applications

The problem of fixing security errors has received less attention than detection and prevention techniques. Ganapathy et al. (64) study correct enforcement of authorization rules in legacy applications, such as X SERVER using a reference monitor for authorization policies. Also, static analysis tools have been used to generate patches in vulnerable software automatically, including repair by generating invariants from correct executions statically (65), placement of sanitization functions by taint analysis (66), and searching for violations in pre-defined patterns (67), requiring to a few lines of edits in the source code, or restricted to specific code transformations within a single procedure (68).

The work closest in spirit to LOGICPATCHER is FixMeUp(43), for fixing access control bugs in web applications due to incorrect conditions. At a high level, LOGICPATCHER is tackling a problem of broader scope, that of fixing logic vulnerabilities caused by missing or inconsistent checks, with minimal guidelines about the vulnerability. FixMeUp requires an explicit and correct high-level specification of access control checks to generate a low-level policy specification and a program transformation template, computed using inter-procedural backward slicing similar to LOGICPATCHER. In LOGICPATCHER work, the focus is on correct patch placement in existing code, different from their statement matching and replacement semantics. Also, we do not require explicit roles or a specification of cor-

rect access control check in advance. LOGICPATCHER works using only correct conditions and path identifiers as input, and optimizes patch placement directly.

6.3.2 Retrofitting Input Validation in Legacy Applications

Improper input validation, where the server fails to reject malicious inputs, allows for the possibility of well-known security vulnerabilities such as SQL-injection, Cross-site scripting, etc. Many existing works try to reason about missing and/or insufficient validation to detect as well as prevent these problems e.g., (69; 70; 71; 72; 73; 74). The goal of WAVES is orthogonal to these prior works because it allows the developer to devote the entirety of her input validation development to the server and rest assured that the client validation code will be correct by construction.

Inconsistent Client- and Server-side Validation Inconsistent client and server validation can lead to problems, such as the parameter tampering vulnerabilities (inputs the client rejects but the server accepts) that our recent work (36; 29) established as pervasive in open source and commercial applications.

WAVES avoids these inconsistencies for applications where the server validation code is correct by simply generating that code for the client. Two related works Ripley (75) and (76) also avoid these inconsistencies but for applications where the *client* validation is correct. These two classes of work are therefore complementary for legacy applications. Some prior works have made advances in the direction of offering analysis that spans multiple modules (77), including the application code and database layers.

6.3.3 Synthesis of Input Validation for New Applications

The key goal of WAVES is to enable developers to write input validation routines once (on the server) and have them replicated elsewhere (on the client). The most germane work, Ripley (75) and

(76), could seemingly be used to meet the same objective: write validation code once (on the client) and allow the system to automatically replicate it elsewhere (on the server). However, there is a crucial benefit to writing validation code on the server instead of the client: all constraints, whether static (not dependent on the server's database, file system, etc.) or dynamic (dependent on the server's state) can uniformly be written on the server, but only the static constraints can easily be written on the client. Implementing dynamic constraints on the client requires AJAX and server-side support; thus, dynamic constraints cannot be implemented solely on the client. Furthermore, even if they could be implemented on the client there may be privacy or security reasons to avoid doing so.

Outside the research arena, the most sophisticated tools to aid web development are found within web development frameworks like Ruby on Rails (RoR) (78), Google Web Toolkit (GWT) (46), and Django (79). Google Web Toolkit allows a programmer to specify which code is common to the client and the server. However, it offers no support for a programmer in the problem of identifying and extracting static or dynamic checks that can be performed by the client. We are only aware of the following two tools that allow a developer to write validation in one place and have it enforced in other places: (a) Ruby on Rails with the SimpleForm plugin (80), and (b) Prado (81). With RoR, a developer writes the constraints that data should satisfy on the server, and SimpleForm enforces those constraints on the client. The limitation, however, is that the constraints extracted are limited to a handful of built-in validation routines and are implemented on the client using built-in validation of HTML5. Prado's collection of custom HTML input controls allows a developer to specify required validation at server-side which is also replicated in the client using JavaScript. However, it also allows developers to specify custom validation code for server and client thus introducing avenues for inconsistencies in client and

server validation. WAVES, in contrast, extracts any constraints checked by the server and implements them on the client using custom-generated JavaScript code.

7

CONCLUSION

This dissertation highlights the importance of program specification inference in maintaining the security of web applications. Current practices in web development require publishers and web application server administrators to actively and frequently test their applications and their updates for vulnerabilities. Lack of web development knowledge of the publishers and admins, along with the growing complexity of current web applications calls for automated techniques in prevention, detection and retrofitting of vulnerabilities. Traditional approaches to achieve this goal require some type of program specification to be available to test the applications. Lack of program specifications in most of the free web applications today, demands security analysis tools which are capable of reasoning about the security state of the programs, and require minimal information about the functionality of the applications.

In Chapter 3, we presented MACE, a program analysis tool for automatic detection of authorization vulnerabilities in Web applications. The tool is based on our study and characterization of different authorization attacks and the underlying vulnerabilities. We find privilege escalation vulnerabilities by finding inconsistencies in the authorization contexts at access request points without knowing the correct access control policies. While the analysis is best-effort, the greatest value of MACE is in identifying flaws in these applications using fundamental abstractions, in the absence of any authorization policy specifications, with the benefit of finding important vulnerabilities that were not discovered earlier.

Chapter 4 discussed LOGICPATCHER, a tool for automatically patching application inconsistency vulnerabilities in web applications. LOGICPATCHER focuses on logic errors due to inconsistent security checks in programs and works across a broad variety of application types, including e-commerce servers, news servers, wikis etc. LOGICPATCHER takes a vulnerability description as input, which includes the conditions that need to be fixed and the associated context, a description of the path in the program where the vulnerability was found, as well as expected exception handling details. Using a combination of backward program slicing and inter-procedural live variable analysis, in addition to standard control and data flow analysis, LOGICPATCHER preserves data dependencies and finds the right scope to insert the patches, without changing the logic in other non-vulnerable paths in the program.

Though LOGICPATCHER is best-effort and works without explicit functional or policy specifications, we were able to generate near-optimal patches and fix important vulnerabilities on 9 open source PHP web applications that were previously studied in literature from the point of view of vulnerability detection, in spite of inherent limitations such as cascading sinks. Verifying the correctness of these patches by hand demonstrates that LOGICPATCHER works well in identifying the correct scope and placing the patch in optimal code locations.

In Chapter 5, we introduced a new methodology for developing client validation code for web applications. Our approach, WAVES, allows the developer to improve the security of the web application by focusing only on the server side development of validation. We developed novel techniques for automatic synthesis of the client side validation. Our experimental results are promising: they indicate that automated synthesis can result in highly interactive web applications that are competitive in terms of performance and rival human-generated code in terms of coverage.

APPENDICES

Appendix A

ANNOTATION EFFORT FOR MACE

Table XI shows the set of input information (hints) we provided for our tool MACE. Gathering this information about each application requires minimal effort and some familiarity with the applications, as discussed in Section 3.4.4.

TABLE XI: PROVIDED ANNOTATIONS TO MACE

Application	Input Variables	Role Values
phpns	`$globalvars['rank']`, `$_COOKIE['cookie_auth']`, `$_SESSION['auth']`, `$_SESSION['username']`, `$_SESSION['userID']`, `$_SESSION['permissions']`, `$_SESSION['path']`	(dynamic)
DCPPortal	`$_COOKIE["dcp5_member_id"]`, `$_COOKIE["dcp5_member_admin"]`, `$HTTP_COOKIE_VARS--["dcp5_member_admin"]`	(dynamic)
DNScript	`$_SESSION['admin']`, `$_SESSION['member']`	1for admin, 0 for non-admin
myBloggie	`$_SESSION['username']`, `$userid['level']`, `$_SESSION['user_id']`	1(for admin), 2 (for normal)
miniBloggie	`$_SESSION['user']`	-
SCARF	`$_SESSION['privlege']`, `$_SESSION['user_id']`	'admin', 'user'
WeBid	`$_SESSION['WEBID_LOGGED_IN']`, `$user_data['groups']`, `$_SESSION['WEBID_ADMIN_USER']`, `$_SESSION['WEBID_ADMIN_IN']`, `$group['can_sell']`, `$group['can_sell']`, `$group['auto_join']`	admin role flag, user groups have dynamic values

Appendix B

ERROR HANDLING

As described in Section 4.3 LOGICPATCHER mines the security exception handling information from source-code. This pre-processing step helps the user of LOGICPATCHER to consistently handle exceptions throughout the whole application. LOGICPATCHER users can decide which of the mined instructions should be included in E. Table XII shows some sample handling methods used by different applications.

TABLE XII: MINED SECURITY EXCEPTIONS BY LOGICPATCHER

Application	**Security Exception Handling Method**
DCPPortal	no else branch Termination Redirect to Login page
SCARF	Termination Redirect to Login page
SPHPBlog	Redirect to Login page
SnipeGallery	Redirect to Login page
PHPNews	Termination Redirect to Login page
MiniBloggie	Termination Redirect to Login page
MyBloggie	Termination
PHPNS	Termination Redirect to Login page
Landshop	Termination

Appendix C

PROOFS

Definition 1 (Constraint Semantics)**.** *Each constraint over variables X describes a possibly infinite set of variable assignments to X. If C is the constraint, we denote the set of variables appearing in C as Vars*(C) *and the set of assignments described by C as VA*(C)*. The semantics of a conjunction of constraints (which we also consider a constraint) is defined as usual.*

$$\begin{aligned} VA(C_1(\bar{x},\bar{y}) \wedge C_2(\bar{x},\bar{z})) = \\ \{\bar{x}/\bar{a},\bar{y}/\bar{b},\bar{z}/\bar{c} \mid \bar{x}/\bar{a},\bar{y}/\bar{b} \in VA(C_1(\bar{x},\bar{y})), \\ \bar{x}/\bar{a},\bar{z}/\bar{c} \in VA(C_2(\bar{x},\bar{z}))\} \end{aligned}$$

Definition 2 (Input Semantics)**.** *The input semantics for a form is the (possibly infinite) set of variable assignments permitted by that form. A variable assignment* X/A *is consistent with the input semantics* Δ *if there is an extension of* X/A *that belongs to* Δ*. A variable assignment* X/A *is inconsistent if there is no extension of* X/A *belonging to* Δ*.*

Definition 3 (Error and Safe Conditions)**.** *A constraint C is an* error condition *for input semantics* Δ *if every* $v \in VA(C)$ *is inconsistent with* Δ*. A constraint C is a* safe condition *for* Δ *if every* $v \in VA(C)$ *is consistent with* Δ*.*

Definition 4 (Success and Failure Traces)**.** *The conjunction of constraints checked on a success trace is a safe condition, and the conjunction of constraints checked on a failure trace is an error condition.*

Appendix C (Continued)

Definition 5 (Independence)**.** *A set of variables X is independent of the set of variables Y (where X and Y are assumed disjoint) for input semantics* Δ *if whenever the variable assignment* X/A *is consistent with* Δ *and the variable assignment* Y/B *is consistent with* Δ *then the assignment* $\{X/A, Y/B\}$ *is consistent with* Δ*. We say that a partitioning of variables* $X_1 \cup \cdots \cup X_n$ *is independent if* X_i *is independent for* X_j *for every* $i \neq j$*. We say a partitioning is strongly independent if* X_i *is independent of* $\bigcup_{j \neq i} X_j$*.*

Note that not all independent partitionings are strongly independent. Consider 3 variables *x*,*y*,*z* where Δ is all variable assignments except $\{x/a, y/b, z/c\}$. Then $\{x\}, \{y\}, \{z\}$ is an independent partitioning because any variable assignment for *x*,*y* can be extended to an assignment in Δ; any variable assignment for *x*,*z* can be extended; and any variable assignment for *y*,*z* can be extended, but $\{x/a, y/b, z/c\}$ cannot be extended to an assignment in Delta.

Theorem 1. *Suppose* Δ *is the input semantics for a web form. Suppose* $D_1 \wedge \cdots \wedge D_k \wedge C_{k+1} \wedge \cdots \wedge C_n$ *is the conjunction of constraints for some failure trace for that form, where* $Vars(D_1) \cup \cdots \cup Vars(D_k) \cup Vars(C_{k+1}) \cup \cdots \cup Vars(C_n)$ *is a strongly independent partitioning for* Δ *and for* $VA(D_1 \wedge \cdots \wedge D_k \wedge C_{k+1} \wedge \cdots \wedge C_n)$*. Suppose that for each* D_i *there is some* E_i *where (i)* $E_i \wedge F_1 \wedge \cdots \wedge F_m$ *are the constraints checked on a success trace, (ii)* $Vars(E_i)$ *is independent of the rest of the variables in the conjunction for* $VA(E_i \wedge F_1 \wedge \cdots \wedge F_m)$*, and (iii)* $VA(D_i)$ *intersects* $VA(E_i)$*. Then* $C_{k+1} \wedge \cdots \wedge C_n$ *is an error condition for* Δ*.*

Proof. Let $X = Vars(D_1 \wedge \cdots \wedge D_k \wedge C_{k+1} \wedge \cdots \wedge C_n)$. Since $D_1 \wedge \cdots \wedge D_k \wedge C_{k+1} \wedge \cdots \wedge C_n$ is the conjunction of constraints for a failure trace, $D_1 \wedge \cdots \wedge D_k \wedge C_{k+1} \wedge \cdots \wedge C_n$ is an error condition, ensuring that each X/A in $VA(D_1 \wedge \cdots \wedge D_k \wedge C_{k+1} \wedge \cdots \wedge C_n)$ is inconsistent with Δ.

Appendix C (Continued)

Consider an assignment $Vars(D_1)/B$ such that $Vars(D_1)/B$ is in the intersection of $VA(D_1)$ and $VA(E_1)$. Since $E_1 \wedge F_1 \wedge \cdots \wedge F_m$ is on a success trace, it is a safe condition, ensuring that each assignment in $VA(E_i \wedge F_1 \wedge \cdots \wedge F_m)$ is consistent with Δ. Since $Vars(D_1)/B \in VA(E_i)$ and $Vars(E_i)$ is independent of the variables in $F_1 \wedge \cdots \wedge F_m$, we know that $Vars(D_1)/B \in VA(E_i \wedge F_1 \wedge \cdots \wedge F_m)$ and hence $Vars(D_1)/B$ is consistent with Δ.

By strong independence of $Vars(D_1)$ and $Vars(D_2 \wedge \cdots \wedge D_k \wedge C_{k+1} \wedge \cdots \wedge C_n)$, we know that we can combine $Vars(D_1)/B$ and any assignment $(X - Vars(D1))/C$ in $VA(D_2 \wedge \cdots \wedge D_k \wedge C_{k+1} \wedge \cdots \wedge C_n)$ to produce an assignment in $VA(D_1 \wedge \cdots \wedge D_k \wedge C_{k+1} \wedge \cdots \wedge C_n)$; thus, $\{Vars(D_1)/B, (X - Vars(D_1))/C\}$ must be inconsistent with Δ. By strong independence with respect to Δ, we see that either $Vars(D_1)/B$ or $(X - Vars(D_1))/C$ or both must therefore be inconsistent (since if both were individually consistent, their combination would be consistent. Since $Vars(D_1)/B$ is consistent by construction, we know that $(X - Vars(D_1))/C$ must be inconsistent, i.e., every element of $VA(D_2 \wedge \cdots \wedge D_k \wedge C_{k+1} \wedge \cdots \wedge C_n)$ is inconsistent, and thus $D_2 \wedge \cdots \wedge D_k \wedge C_{k+1} \wedge \cdots \wedge C_n$ is an error condition. Since we chose D_1 arbitrarily, the argument applies to all D_i and hence by straightforward induction we conclude that $C_{k+1} \wedge \cdots \wedge C_n$ is an error condition. □

CITED LITERATURE

1. Internet live stats. http://www.internetlivestats.com/.

2. Verizon 2016 data breach investigations report. http://www.verizonenterprise.com/verizon-insights-lab/dbir/2016/.

3. 500 million yahoo accounts stolen by state-sponsored hackers. https://threatpost.com/500-million-yahoo-accounts-stolen-by-state-sponsored-hackers/120818/.

4. Du, W., Jayaraman, K., Tan, X., Luo, T., and Chapin, S.: Position paper: Why are there so many vulnerabilities in web applications? In Proceedings of the 2011 Workshop on New Security Paradigms Workshop, NSPW '11, pages 83–94, New York, NY, USA, 2011. ACM.

5. Owasp session management cheat sheet. https://www.owasp.org/index.php/Session_Management_Cheat_Sheet.

6. Mitre top 25. http://cwe.mitre.org/top25/.

7. Ferraiolo, D. F., Sandhu, R., Gavrila, S., Kuhn, D. R., and Chandramouli, R.: Proposed nist standard for role-based access control. ACM Trans. Inf. Syst. Secur., 4(3):224–274, August 2001.

8. Son, S., McKinley, K. S., and Shmatikov, V.: Rolecast: finding missing security checks when you do not know what checks are. In Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications, OOPSLA '11, pages 1069–1084, New York, NY, USA, 2011. ACM.

9. Citi breach: 360k card accounts affected. http://www.bankinfosecurity.com/citi-breach-360k-card-accounts-affected-a-3760.

10. Application vulnerability report. Technical report, http://www.cenzic.com/downloads/Cenzic_Vulnerability_Report_2014.pdf, 2014.

11. Virtual private database. http://www.oracle.com/technetwork/database/security/index-088277.html.

12. Ruby on rails website. http://rubyonrails.org/, 2011.

13. Flanagan, C. and Leino, K. R. M.: Houdini, an annotation assistant for esc/-java. In Proceedings of the International Symposium of Formal Methods Europe on Formal Methods for Increasing Software Productivity, FME '01, pages 500–517, London, UK, UK, 2001. Springer-Verlag.

14. Ernst, M. D., Perkins, J. H., Guo, P. J., McCamant, S., Pacheco, C., Tschantz, M. S., and Xiao, C.: The daikon system for dynamic detection of likely invariants. Sci. Comput. Program., 69(1-3):35–45, December 2007.

15. Cousot, P.: Abstract interpretation: Achievements and perspectives. Compact Disk Paper 224 and Electronic Proceedings. Scuola Superiore G. Reiss Romoli, 2000.

16. Cousot, P. and Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, POPL '77, pages 238–252, New York, NY, USA, 1977. ACM.

17. Robic, B.: The Foundations of Computability Theory. Springer Publishing Company, Incorporated, 1st edition, 2015.

18. A tutorial on abstract interpretation. http://homepage.cs.uiowa.edu/~tinelli/classes/seminar/Cousot--A%20Tutorial%20on%20AI.pdf.

19. Cousot, P.: Abstract Interpretation: Theory and Practice, pages 2–5. Berlin, Heidelberg, Springer Berlin Heidelberg, 2002.

20. Graf, S. and Saïdi, H.: Construction of abstract state graphs with pvs. In Proceedings of the 9th International Conference on Computer Aided Verification, CAV '97, pages 72–83, London, UK, UK, 1997. Springer-Verlag.

21. Cousot, P. and Cousot, R.: Systematic design of program analysis frameworks. In Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, POPL '79, pages 269–282, New York, NY, USA, 1979. ACM.

22. Hoare, C. A. R.: An axiomatic basis for computer programming. Commun. ACM, 12(10):576–580, October 1969.

23. Cwe-639. http://cwe.mitre.org/data/definitions/639.html.

24. OWASP: Testing for privilege escalation. https://www.owasp.org/index.php/Testing_for_Privilege_escalation_(OWASP-AZ-003).

25. Horwitz, S., Reps, T., and Binkley, D.: Interprocedural slicing using dependence graphs. In Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation, PLDI '88, pages 35–46, New York, NY, USA, 1988. ACM.

26. Jovanovic, N., Kruegel, C., and Kirda, E.: Pixy: A static analysis tool for detecting web application vulnerabilities (short paper). In Proceedings of the 2006 IEEE Symposium on Security and Privacy, SP '06, pages 258–263, Washington, DC, USA, 2006. IEEE Computer Society.

27. Bisht, P., Sistla, A. P., and Venkatakrishnan, V. N.: Taps: Automatically preparing safe sql queries. In Proceedings of the 17th ACM Conference on Computer and Communications Security, CCS '10, pages 645–647, New York, NY, USA, 2010. ACM.

28. Xie, Y. and Aiken, A.: Static detection of security vulnerabilities in scripting languages. In Proceedings of the 15th Conference on USENIX Security Symposium - Volume 15, USENIX-SS'06, Berkeley, CA, USA, 2006. USENIX Association.

29. Bisht, P., Hinrichs, T., Skrupsky, N., and Venkatakrishnan, V. N.: Waptec: Whitebox analysis of web applications for parameter tampering exploit construction. In Proceedings of the 18th ACM Conference on Computer and Communications Security, CCS '11, pages 575–586, New York, NY, USA, 2011. ACM.

30. Dalton, M., Kozyrakis, C., and Zeldovich, N.: Nemesis: Preventing authentication & access control vulnerabilities in web applications. In USENIX Security Symposium, pages 267–282. USENIX Association, 2009.

31. Yip, A., Wang, X., Zeldovich, N., and Kaashoek, M. F.: Improving application security with data flow assertions. In Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles, SOSP '09, pages 291–304, New York, NY, USA, 2009. ACM.

32. Sans critical security controls for effective cyber defense, 2015.

33. Felmetsger, V., Cavedon, L., Kruegel, C., and Vigna, G.: Toward automated detection of logic vulnerabilities in web applications. In Proceedings of the 19th USENIX Conference on Security, USENIX Security'10, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association.

34. Son, S. and Shmatikov, V.: Saferphp: Finding semantic vulnerabilities in php applications. In ACM PLAS, 2011.

35. Pellegrino, G. and Balzarotti, D.: Toward black-box detection of logic flaws in web applications. In NDSS 2014, Network and Distributed System Security Symposium, 23-26 February 2014, San Diego, USA, 2014.

36. Bisht, P., Hinrichs, T., Skrupsky, N., Bobrowicz, R., and Venkatakrishnan, V. N.: Notamper: Automatic blackbox detection of parameter tampering opportunities in web applications. In Proceedings of the 17th ACM Conference on Computer and Communications Security, CCS '10, pages 607–618, New York, NY, USA, 2010. ACM.

37. Alkhalaf, M., Choudhary, S. R., Fazzini, M., Bultan, T., Orso, A., and Kruegel, C.: Viewpoints: Differential string analysis for discovering client- and server-side input validation inconsistencies. In Proceedings of the 2012 International Symposium on Software Testing and Analysis, ISSTA 2012, pages 56–66, New York, NY, USA, 2012. ACM.

38. Son, S., McKinley, K. S., and Shmatikov, V.: Rolecast: finding missing security checks when you do not know what checks are. In Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications, OOPSLA '11, pages 1069–1084, New York, NY, USA, 2011. ACM.

39. Monshizadeh, M., Naldurg, P., and Venkatakrishnan, V. N.: Mace: Detecting privilege escalation vulnerabilities in web applications. In Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, CCS '14, pages 690–701, New York, NY, USA, 2014. ACM.

40. Tan, L., Zhang, X., Ma, X., Xiong, W., and Zhou, Y.: Autoises: Automatically inferring security specifications and detecting violations. In Proceedings of the 17th Conference on Security Symposium, SS'08, pages 379–394, Berkeley, CA, USA, 2008. USENIX Association.

41. Ganapathy, V., Jaeger, T., and Jha, S.: Automatic placement of authorization hooks in the Linux Security Modules framework. In Proceedings of the 12th ACM Conference on Computer and Communications Security, pages 330–339, November 2005.

42. Monshizadeh, M., Naldurg, P., and Venkatakrishnan, V. N.: Patching logic vulnerabilities for web applications using logicpatcher. In CODASPY, eds. E. Bertino, R. Sandhu, and A. Pretschner, pages 73–84. ACM, 2016.

43. Son, S., Mckinley, K. S., and Shmatikov, V.: Fix me up: Repairing access-control bugs in web applications. In In Network and Distributed System Security Symposium, 2013.

44. Ball, T. and Larus, J. R.: Efficient path profiling. In Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture, MICRO 29, pages 46–57, Washington, DC, USA, 1996. IEEE Computer Society.

45. Aho, A. V., Sethi, R., and Ullman, J. D.: Compilers: Principles, Techniques, and Tools. Boston, MA, USA, Addison-Wesley Longman Publishing Co., Inc., 1986.

46. Google web toolkit. `http://www.gwtproject.org/`.

47. Saxena, P., Akhawe, D., Hanna, S., Mao, F., McCamant, S., and Song, D.: A Symbolic Execution Framework for JavaScript. In SP'10: Proceedings of the 31st IEEE Symposium on Security and Privacy, Oakland, CA, USA, 2010.

48. Tip, F.: A survey of program slicing techniques. Journal of Programming Languages, 3:121–189, 1995.

49. Hinrichs, T. L.: Plato: A Compiler for Interactive Web Forms. In PADL'11: Proceedings of the 13th International Conference on Practical Aspects of Declarative Languages, Austin, TX, USA, 2011.

50. php.js project. `http://phpjs.org/`, 2011.

51. jQuery. `http://jquery.com`.

52. CLOC: Count Lines of Code. `http://cloc.sourceforge.net`.

53. Ganapathy, V., King, D., Jaeger, T., and Jha, S.: Mining security-sensitive operations in legacy code using concept analysis. In Proceedings of the 29th International Conference on Software Engineering, ICSE '07, pages 458–467, Washington, DC, USA, 2007. IEEE Computer Society.

54. Parno, B., McCune, J. M., Wendlandt, D., Andersen, D. G., and Perrig, A.: Clamp: Practical prevention of large-scale data leaks. In Proceedings of the 2009 30th IEEE Symposium

on Security and Privacy, SP '09, pages 154–169, Washington, DC, USA, 2009. IEEE Computer Society.

55. Felt, A. P., Finifter, M., Weinberger, J., and Wagner, D.: Diesel: Applying privilege separation to database access. In Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security, ASIACCS '11, pages 416–422, New York, NY, USA, 2011. ACM.

56. Krishnamurthy, A., Mettler, A., and Wagner, D.: Fine-grained privilege separation for web applications. In Proceedings of the 19th international conference on World wide web, WWW '10, pages 551–560, New York, NY, USA, 2010. ACM.

57. Cova, M., Balzarotti, D., Felmetsger, V., and Vigna, G.: Swaddler: An approach for the anomaly-based detection of state violations in web applications. In Proceedings of the 10th International Conference on Recent Advances in Intrusion Detection, RAID'07, pages 63–86, Berlin, Heidelberg, 2007. Springer-Verlag.

58. Engler, D., Chen, D. Y., Hallem, S., Chou, A., and Chelf, B.: Bugs as deviant behavior: A general approach to inferring errors in systems code. In Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles, SOSP '01, pages 57–72, New York, NY, USA, 2001. ACM.

59. Srivastava, V., Bond, M. D., McKinley, K. S., and Shmatikov, V.: A security policy oracle: Detecting security holes using multiple api implementations. In Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation, 2011.

60. Vijayakumar, H., Ge, X., Payer, M., and Jaeger, T.: Jigsaw: Protecting resource access by inferring programmer expectations. In Proceedings of the 23rd USENIX Conference on Security Symposium, SEC'14, pages 973–988, Berkeley, CA, USA, 2014. USENIX Association.

61. Pellegrino, G. and Balzarotti, D.: Toward black-box detection of logic flaws in web applications. In NDSS 2014, Network and Distributed System Security Symposium, 23-26 February 2014, San Diego, USA, San Diego, UNITED STATES, 02 2014.

62. Sun, F., Xu, L., and Su, Z.: Static detection of access control vulnerabilities in web applications. In Proceedings of the 20th USENIX conference on Security, SEC'11, pages 11–11, Berkeley, CA, USA, 2011. USENIX Association.

63. Doupé, A., Boe, B., Kruegel, C., and Vigna, G.: Fear the ear: Discovering and mitigating execution after redirect vulnerabilities. In Proceedings of the 18th ACM Conference on

Computer and Communications Security, CCS '11, pages 251–262, New York, NY, USA, 2011. ACM.

64. Ganapathy, V., Jaeger, T., and Jha, S.: Retrofitting legacy code for authorization policy enforcement. 2012 IEEE Symposium on Security and Privacy, 0:214–229, 2006.

65. Jin, G., Song, L., Zhang, W., Lu, S., and Liblit, B.: Automated atomicity-violation fixing. In Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation, 2011.

66. Livshits, B. and Chong, S.: Towards fully automatic placement of security sanitizers and declassifiers. In Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, pages 385–398, New York, NY, USA, 2013. ACM.

67. Perkins, J. H., Kim, S., Larsen, S., Amarasinghe, S., Bachrach, J., Carbin, M., Pacheco, C., Sherwood, F., Sidiroglou, S., Sullivan, G., Wong, W.-F., Zibin, Y., Ernst, M. D., and Rinard, M.: Automatically patching errors in deployed software. In Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles SOSP 09, 2009.

68. Andersen, J. and Lawall, J. L.: Generic patch inference. In 23rd IEEE/ACM International Conference on Automated Software Engineering ASE 08, 2008.

69. Saxena, P., Hanna, S., Poosankam, P., and Song, D.: FLAX: Systematic Discovery of Client-side Validation Vulnerabilities in Rich Web Applications. In NDSS'10: Proceedings of the 17th Annual Network and Distributed System Security Symposium, San Diego, CA, USA, 2010.

70. Balzarotti, D., Cova, M., Felmetsger, V., Jovanovic, N., Kruegel, C., Kirda, E., and Vigna, G.: Saner: Composing Static and Dynamic Analysis to Validate Sanitization in Web Applications. In SP'08: Proceedings of the 29th IEEE Symposium on Security and Privacy, Oakland, CA, USA, 2008.

71. Xie, Y. and Aiken, A.: Static Detection of Security Vulnerabilities in Scripting Languages. In SS'06: Proceedings of the 15th USENIX Security Symposium, Vancouver, B.C., Canada, 2006.

72. Minamide, Y.: Static Approximation of Dynamically Generated Web Pages. In WWW'05: Proceedings of the 14th International Conference on World Wide Web, Chiba, Japan, 2005.

73. Wassermann, G. and Su, Z.: Sound and Precise Analysis of Web Applications for Injection Vulnerabilities. In PLDI'07: Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation, San Diego, CA, USA, 2007.

74. Xu, W., Bhatkar, S., and Sekar, R.: Taint-Enhanced Policy Enforcement: A Practical Approach to Defeat a Wide Range of Attacks. In SS'06: Proceedings of the 15th USENIX Security Symposium, Vancouver, B.C., Canada, 2006.

75. Vikram, K., Prateek, A., and Livshits, B.: Ripley: Automatically Securing Distributed Web Applications Through Replicated Execution. In CCS'09: Proceedings of the 16th Conference on Computer and Communications Security, Chicago, IL, USA, 2009.

76. Bethea, D., Cochran, R., and Reiter, M.: Server-side Verification of Client Behavior in Online Games. In NDSS'10: Proceedings of the 17th Annual Network and Distributed System Security Symposium, San Diego, CA, USA, 2010.

77. Balzarotti, D., Cova, M., Felmetsger, V. V., and Vigna, G.: Multi-Module Vulnerability Analysis of Web-based Applications. In CCS'07: Proceedings of the 14th ACM Conference on Computer and Communications Security, Alexandria, VA, USA, 2007.

78. Ruby on Rails. `http://rubyonrails.org/`.

79. django: Python Web Framework. `https://www.djangoproject.com/`.

80. Simpleform website. `http://blog.plataformatec.com.br/2010/06/simpleform-forms-made-easy/`, 2011.

81. Component Framework for PHP5. `http://www.pradosoft.com`.

82. Cwe-566. `http://cwe.mitre.org/data/definitions/566.html`.

83. Epic face palm. `http://cci.uncc.edu/sites/cci.uncc.edu/files/media/files/Epic%20Facepalm.pdf`.

84. Wang, R., Chen, S., Wang, X., and Qadeer, S.: How to Shop for Free Online – Security Analysis of Cashier-as-a-Service Based Web Stores. In Oakland'11: Proceedings of the 2011 IEEE Symposium on Security and Privacy, Oakland, CA, USA, 2011.

85. Alkhalaf, M., Bultan, T., Choudhary, S. R., Fazzini, M., Orso, A., and Kruegel, C.: ViewPoints: Differential String Analysis for Discovering Client and Server-Side Input Vali-

dation Inconsistencies. In ISSTA'12: Proceedings of the 2011 International Symposium on Software Testing and Analysis, Minneapolis, MN, USA, 2012.

86. What is a privilege escalation. http://www.wisegeek.com/what-is-a-privilege-escalation.htm.

87. Choosing a content management system. http://www.hugeinc.com/ideas/report/choosing-a-content-management-system.

88. Wordpress vs joomla vs drupal? https://websitesetup.org/cms-comparison-wordpress-vs-joomla-drupal/.

89. Framework usage statistics. https://trends.builtwith.com/framework.

90. Tomb, A. and Flanagan, C.: Detecting inconsistencies via universal reachability analysis. In Proceedings of the 2012 International Symposium on Software Testing and Analysis, ISSTA 2012, pages 287–297, New York, NY, USA, 2012. ACM.

91. Furia, C. A. and Meyer, B.: Fields of logic and computation. chapter Inferring Loop Invariants Using Postconditions, pages 277–300. Berlin, Heidelberg, Springer-Verlag, 2010.

92. Nimmer, J. W. and Ernst, M. D.: Automatic generation of program specifications. In ISSTA 2002, Proceedings of the 2002 International Symposium on Software Testing and Analysis, pages 232–242, Rome, Italy, July 22–24, 2002.

93. De Capitani di Vimercati, S., Foresti, S., Jajodia, S., Paraboschi, S., and Samarati, P.: Assessing query privileges via safe and efficient permission composition. In Proceedings of the 15th ACM Conference on Computer and Communications Security, CCS '08, pages 311–322, New York, NY, USA, 2008. ACM.

94. Rizvi, S., Mendelzon, A., Sudarshan, S., and Roy, P.: Extending query rewriting techniques for fine-grained access control. In Proceedings of the 2004 ACM SIGMOD international conference on Management of data, SIGMOD '04, pages 551–562, New York, NY, USA, 2004. ACM.

95. Son, S., McKinley, K. S., and Shmatikov, V.: Fix me up: Repairing access-control bugs in web applications. In *NDSS* (96).

96. 20th Annual Network and Distributed System Security Symposium, NDSS 2013, San Diego, California, USA, February 24-27, 2013. The Internet Society, 2013.

97. Sandhu, R. and Munawer, Q.: How to do discretionary access control using roles. In Proceedings of the Third ACM Workshop on Role-based Access Control, RBAC '98, pages 47–54, New York, NY, USA, 1998. ACM.

98. Kim, D., Nam, J., Song, J., and Kim, S.: Automatic patch generation learned from human-written patches. In Proceedings of the 2013 International Conference on Software Engineering, ICSE '13, pages 802–811, Piscataway, NJ, USA, 2013. IEEE Press.

99. Ray, B., Kim, M., Person, S., and Rungta, N.: Detecting and characterizing semantic inconsistencies in ported code. In Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on, pages 367–377, Nov 2013.

100. Meng, N., Kim, M., and McKinley, K. S.: Lase: Locating and applying systematic edits by learning from examples. In Proceedings of the 2013 International Conference on Software Engineering, ICSE '13, pages 502–511, Piscataway, NJ, USA, 2013. IEEE Press.

101. Alkhalaf, M., Aydin, A., and Bultan, T.: Semantic differential repair for input validation and sanitization. In Proceedings of the 2014 International Symposium on Software Testing and Analysis, ISSTA 2014, pages 225–236, New York, NY, USA, 2014. ACM.

102. Skrupsky, N., Monshizadeh, M., Bisht, P., Hinrichs, T., Venkatakrishnan, V. N., and Zuck, L.: Waves: Automatic synthesis of client-side validation code for web applications. In 2012 International Conference on Cyber Security, pages 46–53, Dec 2012.

103. Skrupsky, N., Monshizadeh, M., Bisht, P., Hinrichs, T., Venkatakrishnan, V. N., and Zuck, L.: Don't repeat yourself: Automatically synthesizing client-side validation code for web applications. In Proceedings of the 3rd USENIX Conference on Web Application Development, WebApps'12, pages 11–11, Berkeley, CA, USA, 2012. USENIX Association.

Copyright Permission Statement

ACM Publishing License and Audio/Video Release

Title of the Work: MACE: Detecting Privilege Escalation Vulnerabilities in Web Applications

Publication and/or Conference Name: CCS'14: 2014 ACM SIGSAC Conference on Computer and Communications Security Proceedings

Author/Presenter(s): Maliheh Monshizadeh (Univ. of Illinois at Chicago); Prasad Naldurg (IBM Research India); V. N. Venkatakrishnan (Univ. of Illinois at Chicago)

Auxiliary Materials (provide filenames and a description of auxiliary content, if any, for display in the ACM Digital Library. The description may be provided as a ReadMe file):

1. Glossary

2. Grant of Rights

(a) Owner hereby grants to ACM an exclusive, worldwide, royalty-free, perpetual, irrevocable, transferable and sublicenseable license to publish, reproduce and distribute all or any part of the Work in any and all forms of media, now or hereafter known, including in the above publication and in the ACM Digital Library, and to authorize third parties to do the same.

(b) In connection with software and "Artistic Images and "Auxiliary Materials, Owner grants ACM non-exclusive permission to publish, reproduce and distribute in any and all forms of media, now or hereafter known, including in the above publication and in the ACM Digital Library.

(c) In connection with any "Minor Revision", that is, a derivative work containing less than twenty-five percent (25%) of new substantive material, Owner hereby grants to ACM all rights in the Minor Revision that Owner grants to ACM with respect to the Work, and all terms of this Agreement shall apply to the Minor Revision.

☑ A. Grant of Rights. I grant the rights and agree to the terms described above.

☐ B. Declaration for Government Work. I am an employee of the national government of my country and my Government claims rights to this work, or it is not copyrightable (Government work is classified as Public Domain in U.S. only)

Country:

---

3. Reserved Rights and Permitted Uses.

(a) All rights and permissions the author has not granted to ACM in Paragraph 2 are reserved to the Owner, including without limitation the ownership of the copyright of the Work and all other proprietary rights such as patent or trademark rights.

(b) Furthermore, notwithstanding the exclusive rights the Owner has granted to ACM in Paragraph 2(a), Owner shall have the right to do the following:

(i) Reuse any portion of the Work, without fee, in any future works written or edited by the Author, including books, lectures and presentations in any and all

media.

(ii) Create a "Major Revision" which is wholly owned by the author

(iii) Post the Accepted Version of the Work on (1) the Authors home page, (2) the Owner's institutional repository, or (3) any repository legally mandated by an agency funding the research on which the Work is based.

(iv) Post an "Author-Izer" link enabling free downloads of the Version of Record in the ACM Digital Library on (1) the Author's home page or (2) the Owner's institutional repository;

(v) Prior to commencement of the ACM peer review process, post the version of the Work as submitted to ACM ("Submitted Version" or any earlier versions) to non-peer reviewed servers;

(vi) Make free distributions of the final published Version of Record internally to the Owner's employees, if applicable;

(vii) Make free distributions of the published Version of Record for Classroom and Personal Use; (viii) Bundle the Work in any of Owner's software distributions; and

(xi) Use any Auxiliary Material independent from the Work.

Authors should understand that consistent with ACMs policy of encouraging dissemination of information, each work published by ACM appears with the ACM copyright and the following notice:

When preparing your paper for submission using the ACM templates, you will need to include the rights management and bibstrip text blocks below to the lower left hand portion of the first page. As this text will provide rights information for your paper, please make sure that this text is displayed and positioned correctly when you submit your manuscript for publication.

"Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org."

---

4. ACM Citation and Digital Object Identifier.

(a) In connection with any use by the Owner of the Definitive Version, Owner shall include the ACM citation and ACM Digital Object Identifier (DOI).
(b) In connection with any use by the Owner of the Submitted Version (if accepted) or the Accepted Version or a Minor Revision, Owner shall use best efforts to display

the ACM citation, along with a statement substantially similar to the following:

> "© [Owner] [Year]. This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive version was published in {Source Publication}, http://dx.doi.org/10.1145/{number}."

5. Audio/Video Recording

I hereby grant permission for ACM to include my name, likeness, presentation and comments in any and all forms, for the Conference and/or Publication.

I further grant permission for ACM to record and/or transcribe and reproduce my presentation as part of the ACM Digital Library, and to distribute the same for sale in complete or partial form as part of an ACM product on CD-ROM, DVD, webcast, USB device, streaming video or any other media format now or hereafter known.

I understand that my presentation will not be sold separately by itself as a stand-alone product without my direct consent. Accordingly, I give ACM the right to use my image, voice, pronouncements, likeness, and my name, and any biographical material submitted by me, in connection with the Conference and/or Publication, whether used in excerpts or in full, for distribution described above and for any associated advertising or exhibition.

A. Do you agree to the above Audio/Video Release? ◉ Yes ○ No

B. Auxiliary Materials, not integral to the Work

Do you have any Auxiliary Materials? ○ Yes ◉ No

I hereby grant ACM permission to serve files named below containing my Auxiliary Material from the ACM Digital Library. I hereby represent and warrant that my Auxiliary Material contains no malicious code, virus, trojan horse or other software routines or hardware components designed to permit unauthorized access or to disable, erase or otherwise harm any computer systems or software, and I hereby agree to indemnify and hold harmless ACM from all liability, losses, damages, penalties, claims, actions, costs and expenses (including reasonable legal expense) arising from the use of such files.

Do you agree to the above Auxiliary Materials permission statement? Yes ○ No ○

6. Third Party Materials

In the event that any materials used in my presentation or Auxiliary Materials contain the work of third-party individuals or organizations (including copyrighted music or movie excerpts or anything not owned by me), I understand that it is my responsibility to secure any necessary permissions and/or licenses for print and/or digital publication, and cite or attach them below.

◉ We/I have not used third-party material.
○ We/I have used third-party materials and have necessary permissions.

7. Artistic Images

If your paper includes images that were created for any purpose other than this paper and to which you or your employer claim copyright, you must complete Part IV and

be sure to include a notice of copyright with each such image in the paper.
(•) We/I do not have any artistic images.
( ) We/I have any artistic images.

---

8. Representations, Warranties and Covenants

The undersigned hereby represents, warrants and covenants as follows:

(a) Owner is the sole owner or authorized agent of Owner(s) of the Work;

(b) The undersigned is authorized to enter into this Agreement and grant the rights included in this license to ACM;

(c) The Work is original and does not infringe the rights of any third party; all permissions for use of third-party materials consistent in scope and duration with the rights granted to ACM have been obtained, copies of such permissions have been provided to ACM, and the Work as submitted to ACM clearly and accurately indicates the credit to the proprietors of any such third-party materials (including any applicable copyright notice), or will be revised to indicate such credit;

(d) The Work has not been published except for informal postings on non-peer reviewed servers, and Owner covenants to use best efforts to place ACM DOI pointers on any such prior postings;

(e) The Auxiliary Materials, if any, contain no malicious code, virus, trojan horse or other software routines or hardware components designed to permit unauthorized access or to disable, erase or otherwise harm any computer systems or software; and

(f) The Artistic Images, if any, are clearly and accurately noted as such (including any applicable copyright notice) in the Submitted Version.

☑ I agree to the Representations, Warranties and Covenants.

---

9. Enforcement.

At ACM's expense, ACM shall have the right (but not the obligation) to defend and enforce the rights granted to ACM hereunder, including in connection with any instances of plagiarism brought to the attention of ACM. Owner shall notify ACM in writing as promptly as practicable upon becoming aware that any third party is infringing upon the rights granted to ACM, and shall reasonably cooperate with ACM in its defense or enforcement.

---

10. Governing Law

This Agreement shall be governed by, and construed in accordance with, the laws of the state of New York applicable to contracts entered into and to be fully performed therein.

DATE: **08/11/2014** sent to mmonsh2@uic.edu; pnaldurg@in.ibm.com; venkat@uic.edu at **07:08:26**

ACM Publishing License and Audio/Video Release

Title of the Work: Patching Logic Vulnerabilities for Web Applications using LogicPatcher
Submission ID:codaf111
Author/Presenter(s): Maliheh Monshizadeh (University of Illinois at Chicago); Prasad Naldurg (IBM Research India); V. N. Venkatakrishnan (University of Illinois at Chicago)
Type of material: Full Paper

Publication and/or Conference Name: CODASPY'16: Sixth ACM Conference on Data & Application Security and Privacy Proceedings

1. Glossary

2. Grant of Rights

(a) Owner hereby grants to ACM an exclusive, worldwide, royalty-free, perpetual, irrevocable, transferable and sublicenseable license to publish, reproduce and distribute all or any part of the Work in any and all forms of media, now or hereafter known, including in the above publication and in the ACM Digital Library, and to authorize third parties to do the same.

(b) In connection with software and "Artistic Images and "Auxiliary Materials, Owner grants ACM non-exclusive permission to publish, reproduce and distribute in any and all forms of media, now or hereafter known, including in the above publication and in the ACM Digital Library.

(c) In connection with any "Minor Revision", that is, a derivative work containing less than twenty-five percent (25%) of new substantive material, Owner hereby grants to ACM all rights in the Minor Revision that Owner grants to ACM with respect to the Work, and all terms of this Agreement shall apply to the Minor Revision.

☑ A. Grant of Rights. I grant the rights and agree to the terms described above.

☐ B. Declaration for Government Work. I am an employee of the national government of my country and my Government claims rights to this work, or it is not copyrightable (Government work is classified as Public Domain in U.S. only)

Are any of the co-authors, employees or contractors of a National Government?
○ Yes ◉ No
Country:

---

3. Reserved Rights and Permitted Uses.

(a) All rights and permissions the author has not granted to ACM in Paragraph 2 are reserved to the Owner, including without limitation the ownership of the copyright of the Work and all other proprietary rights such as patent or trademark rights.

(b) Furthermore, notwithstanding the exclusive rights the Owner has granted to ACM in Paragraph 2(a), Owner shall have the right to do the following:

(i) Reuse any portion of the Work, without fee, in any future works written or edited by the Author, including books, lectures and presentations in any and all

media.

(ii) Create a "Major Revision" which is wholly owned by the author

(iii) Post the Accepted Version of the Work on (1) the Authors home page, (2) the Owner's institutional repository, or (3) any repository legally mandated by an agency funding the research on which the Work is based.

(iv) Post an "Author-Izer" link enabling free downloads of the Version of Record in the ACM Digital Library on (1) the Author's home page or (2) the Owner's institutional repository;

(v) Prior to commencement of the ACM peer review process, post the version of the Work as submitted to ACM ("Submitted Version" or any earlier versions) to non-peer reviewed servers;

(vi) Make free distributions of the final published Version of Record internally to the Owner's employees, if applicable;

(vii) Make free distributions of the published Version of Record for Classroom and Personal Use;

(viii) Bundle the Work in any of Owner's software distributions; and

(ix) Use any Auxiliary Material independent from the Work.

When preparing your paper for submission using the ACM TeX templates, the rights and permissions information and the bibliographic strip must appear on the lower left hand portion of the first page.

The new Authorized ACM TeX template .cls version 2.8, automatically creates and positions these text blocks for you based on the code snippet which is system-generated based on your rights management choice and this particular conference.

Please copy and paste the following code snippet into your TeX file between \begin{document} and \maketitle, either after or before CCS codes.

```
\CopyrightYear{2016}
\setcopyright{acmlicensed}
\conferenceinfo{CODASPY'16,}{March 09 - 11, 2016, New Orleans, LA, USA}
\isbn{978-1-4503-3935-3/16/03}\acmPrice{\$15.00}
\doi{http://dx.doi.org/10.1145/2857705.2857727}
```

If you are using the ACM Microsoft Word template, or still using an older version of the ACM TeX template, or the current versions of the ACM SIGCHI, SIGGGRAPH, or SIGPLAN TeX templates, you must copy and

paste the following text block into your document as per the instructions provided with the templates you are using:

CODASPY'16, March 09 - 11, 2016, New Orleans, LA, USA

ACM 978-1-4503-3935-3/16/03...$15.00
DOI: http://dx.doi.org/10.1145/2857705.2857727

NOTE: Make sure to include your article's DOI as part of the bibstrip data; DOIs will be registered and become active shortly after publication in the ACM Digital Library

---

4. ACM Citation and Digital Object Identifier.

(a) In connection with any use by the Owner of the Definitive Version, Owner shall include the ACM citation and ACM Digital Object Identifier (DOI).
(b) In connection with any use by the Owner of the Submitted Version (if accepted) or the Accepted Version or a Minor Revision, Owner shall use best efforts to display the ACM citation, along with a statement substantially similar to the following:

> "© [Owner] [Year]. This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive version was published in {Source Publication}, http://dx.doi.org/10.1145/{number}."

5. Audio/Video Recording

I hereby grant permission for ACM to include my name, likeness, presentation and comments in any and all forms, for the Conference and/or Publication.

I further grant permission for ACM to record and/or transcribe and reproduce my presentation as part of the ACM Digital Library, and to distribute the same for sale in complete or partial form as part of an ACM product on CD-ROM, DVD, webcast, USB device, streaming video or any other media format now or hereafter known.

I understand that my presentation will not be sold separately as a stand-alone product without my direct consent. Accordingly, I give ACM the right to use my image, voice, pronouncements, likeness, and my name, and any biographical material submitted by me, in connection with the Conference and/or Publication, whether used in excerpts or in full, for distribution described above and for any associated

advertising or exhibition.

Do you agree to the above Audio/Video Release? (•) Yes ( ) No

6. Auxiliary Material

Any additional materials, including software or other executables that are not submitted for review and therefore not an integral part of the work, but are included for publication.

Do you have any Auxiliary Materials? ( ) Yes (•) No

I hereby grant ACM permission to serve files containing my Auxiliary Material from the ACM Digital Library. I hereby represent and warrant that my Auxiliary Materials do not knowingly and surreptitiously incorporate malicious code, virus, trojan horse or other software routines or hardware components designed to permit unauthorized access or to disable, erase or otherwise harm any computer systems or software.

☐ I agree to the above Auxiliary Materials permission statement.

☐ This software is knowingly designed to illustrate technique(s) intended to defeat a system's security. The code has been explicitly documented to state this fact.

7. Third Party Materials

In the event that any materials used in my presentation or Auxiliary Materials contain the work of third-party individuals or organizations (including copyrighted music or movie excerpts or anything not owned by me), I understand that it is my responsibility to secure any necessary permissions and/or licenses for print and/or digital publication, and cite or attach them below.

(•) We/I have not used third-party material.
( ) We/I have used third-party materials and have necessary permissions.

8. Artistic Images

If your paper includes images that were created for any purpose other than this paper and to which you or your employer claim copyright, you must complete Part IV and be sure to include a notice of copyright with each such image in the paper.

(•) We/I do not have any artistic images.
( ) We/I have any artistic images.

---

9. Representations, Warranties and Covenants

The undersigned hereby represents, warrants and covenants as follows:

(a) Owner is the sole owner or authorized agent of Owner(s) of the Work;

(b) The undersigned is authorized to enter into this Agreement and grant the rights included in this license to ACM;

(c) The Work is original and does not infringe the rights of any third party; all permissions for use of third-party materials consistent in scope and duration with the rights granted to ACM have been obtained, copies of such permissions have been provided to ACM, and the Work as submitted to ACM clearly and accurately indicates the credit to the proprietors of any such third-party materials (including any applicable copyright notice), or will be revised to indicate such credit;

(d) The Work has not been published except for informal postings on non-peer reviewed servers, and Owner covenants to use best efforts to place ACM DOI pointers on any such prior postings;

(e) The Auxiliary Materials, if any, contain no malicious code, virus, trojan horse or other software routines or hardware components designed to permit unauthorized access or to disable, erase or otherwise harm any computer systems or software; and

(f) The Artistic Images, if any, are clearly and accurately noted as such (including any applicable copyright notice) in the Submitted Version.

☑ I agree to the Representations, Warranties and Covenants.

---

10. Enforcement.

At ACM's expense, ACM shall have the right (but not the obligation) to defend and enforce the rights granted to ACM hereunder, including in connection with any instances of plagiarism brought to the attention of ACM. Owner shall notify ACM in writing as promptly as practicable upon becoming aware that any third party is infringing upon the rights granted to ACM, and shall reasonably cooperate with ACM in its defense or enforcement.

---

11. Governing Law

This Agreement shall be governed by, and construed in accordance with, the laws of the state of New York applicable to contracts entered into and to be fully performed therein.

Funding Agents

1. National Science Foundation award number(s):1065537, 1514142, 1069311

2. Defense Advanced Research Projects Agency award number(s):FA8750-12-C-0166, FAFA8650-15-C-7561

DATE: **01/17/2016** sent to mmonsh2@uic.edu; pnaldurg@in.ibm.com; venkat@uic.edu at **13:01:39**

IEEE COPYRIGHT FORM FOR US GOVERNMENT EMPLOYEES

To ensure uniformity of treatment among all contributors, other forms may not be substituted for this form, nor may any wording of the form be changed. This form is intended for original material submitted to the IEEE and must accompany any such material in order to be published by the IEEE.Please read the form carefully and keep a copy for your files.

TITLE OF PAPER/ARTICLE/REPORT, INCLUDING ALL CONTENT IN ANY FORM, FORMAT, OR MEDIA (hereinafter, "The Work"):**WAVES: Automatic Synthesis of Client-side Validation Code for Web Applications**

COMPLETE LIST OF AUTHORS:**Maliheh Monshizadeh**

IEEE PUBLICATION TITLE (Journal, Magazine, Conference, Book):**2012 ASE International Conference on Social Informatics (SocialInformatics 2012) / 2012 ASE International Conference on Cyber Security (CyberSecurity 2012) / 2012 ASE International Conference on BioMedical Computing**

U.S. GOVERNMENT EMPLOYEE CERTIFICATION AND TRANSFER OF INTERNATIONAL COPYRIGHT

1. This will certify that all authors of the Work are U.S. govemment employees and prepared the Work on a subject within the scope of their official duties. As such, the Work is not subject to U.S. copyright protection. In order to enable IEEE to claim and protect copyright rights in international jurisdictions, the undersigned hereby assigns to the IEEE all such international rights under copyright that may exist in and to the above Work, and any revised or expanded derivative works submitted to the IEEE by the undersigned based on the Work.

CONSENT AND RELEASE

2. ln the event the undersigned makes a presentation based upon the Work at a conference hosted or sponsored in whole or in part by the IEEE, the undersigned, in consideration for his/her participation in the conference, hereby grants the IEEE the unlimited, worldwide, irrevocable permission to use, distribute, publish, license, exhibit, record, digitize, broadcast, reproduce and archive, in any format or medium, whether now known or hereafter developed: (a) his/her presentation and comments at the conference; (b) any written materials or multimedia files used in connection with his/her presentation; and (c) any recorded interviews of him/her (collectively, the "Presentation"). The permission granted includes the transcription and reproduction ofthe Presentation for inclusion in products sold or distributed by IEEE and live or recorded broadcast ofthe Presentation during or after the conference.

3. In connection with the permission granted in Section 2, the undersigned hereby grants IEEE the unlimited, worldwide, irrevocable right to use his/her name, picture, likeness, voice and biographical information as part of the advertisement, distribution and sale of products incorporating the Work or Presentation, and releases IEEE from any claim based on right of privacy or publicity.

4. The undersigned hereby warrants that the Work and Presentation (collectively, the "Materials") are original and that he/she is the author of the Materials. To the extent the Materials incorporate text passages, figures, data or other material from the works of others, the undersigned has obtained any necessary permissions. Where necessary, the undersigned has obtained all third party permissions and consents to grant the license above and has provided copies of such permissions and consents to IEEE.

[X] Please check this box ifyou do not wish to have video/audio recordings made ofyour conference presentation.

See below for Retained Rights/Terms and Conditions, and Author Responsibilities.

AUTHOR RESPONSIBILITIES

INFORMATION FOR AUTHORS

IEEE Copyright Ownership

It is the formal policy of the IEEE to own the copyrights to all copyrightable material in its technical publications and to the individual contributions contained therein, in order to protect the interests of the IEEE, its authors and their employers, and, at the same time, to facilitate the appropriate re-use of this material by others.The IEEE distributes its technical publications throughout the world and does so by various means such as hard copy, microfiche, microfilm, and electronic media.It also abstracts and may translate its publications, and articles contained therein, for inclusion in various compendiums, collective works, databases and similar publications. While the IEEE recognizes that works authored by U.S. Government employees on a subject within the scope of their employment are not subject to U.S. copyright protection, it is the formal policy of the IEEE to own any international copyright rights that may exist in such works(Employee Certification and Transfer below).

GENERAL TERMS

1. The undersigned represents that he/she has the power and authority to make and execute this form.
2. The undersigned agrees to identify and hold harmless the IEEE from any damage or expense that may arise in the event of a breach of any of the warranties set forth above.
3. In the event the above work is not accepted and published by the IEEE or is withdrawn by the author(s) before acceptance by the IEEE, the foregoing grant of rights shall become null and void and all materials embodying the Work submitted to the IEEE will be destroyed.
4. For jointly authored Works, all joint authors should sign, or one of the authors should sign as authorized agent for the others.

Maliheh Monshizadeh — **13-11-2012**
Author/Authorized Agent For Joint Authors — **Date(dd-mm-yy)**

THIS FORM MUST ACCOMPANY THE SUBMISSION OF THE AUTHOR'S MANUSCRIPT.
Questions about the submission of the form or manuscript must be sent to the publication's editor.Please direct all questions about IEEE copyright policy to:
IEEE Intellectual Property Rights Office, copyrights@ieee.org, +1-732-562-3966 (telephone)