DETECTION AND PREVENTION OF LOGIC ATTACKS AGAINST WEB APPLICATIONS THROUGH BLACK-BOX ANALYSIS

By

Xiaowei Li

Dissertation

Submitted to the Faculty of the

Graduate School of Vanderbilt University

in partial fulfillment of the requirements

for the degree of

DOCTOR OF PHILOSOPHY

in

Computer Science

August, 2013

Nashville, Tennessee

Approved:

Professor Yuan Xue

Professor Janos Sztipanovits

Professor Bradley Malin

Professor Douglas C. Schmidt

Professor Gautam Biswas

To my parents and Shan

ACKNOWLEDGMENTS

I feel so lucky to be sitting here at this moment. I could never go this far without the people who have helped me during my PhD journey. My deepest gratefulness goes to my advisor, Professor Yuan Xue, for her invaluable guidance and support. She has been a great advisor and my role model, who has taught me how to think, how to solve problems and how to manage my future career. I will always be grateful for her encouragement and belief in me during my PhD study.

I would like to thank the members of my thesis committee: Professor Janos Sztipanovits, Professor Bradley Malin, Professor Douglas C. Schmidt and Professor Gautam Biswas, for their insightful comments on my thesis. I am also grateful for the help and support from all the members of the VANETS research group and all my friends at Vandy. Their friendship comprises my best memories at this country music capital.

This work was sponsored by NSF TRUST (The Team for Research in Ubiquitous Secure Technology) Science and Technology Center (CCF-0424422). In addition to financial support, it provides me valuable opportunities to meet with world-class researchers, which greatly broaden my horizons and benefit my future career.

Last but not least, I would give my special gratitude to my parents for their unconditional love and endless support from across the ocean. Without them, I could never be who I am. It is their sacrifice that makes my dream come true. I am also very grateful to my wife and love Shan, who has been accompanying me for over ten years. She completes me and my life. This work is dedicated to them.

TABLE OF CONTENTS

Page

ACKNOWLEDGMENTS . . . iii

LIST OF TABLES . . . vii

LIST OF FIGURES . . . viii

Chapter

I. INTRODUCTION . . . 1
- Background . . . 2
- Problem Description and Research Goal . . . 3
- Research Approach and Dissertation Contributions . . . 6
  - Detection of Logic Attacks . . . 7
  - Prevention of Logic Attacks . . . 8
- Dissertation Organization . . . 8

II. RELATED WORK . . . 10
- Existing Defenses against Logic Attacks . . . 10
  - Secure Construction of New Web Applications . . . 11
  - Security Analysis/Testing of Legacy Web Applications . . . 12
  - Runtime Protection of Legacy Web Applications . . . 15
  - Summary and Our Contributions . . . 16
- Software Specification Inference . . . 17
- Web Database Security . . . 18
- Test Input Generation . . . 19

III. A BLACK-BOX APPROACH FOR DETECTION OF STATE VIOLATION ATTACKS . . . 21
- Overview . . . 21
- Attacks and Running Example . . . 22
- System Model . . . 25
- Approach . . . 26
  - Approach Overview . . . 26
  - Web Page Symbolization . . . 28
  - Invariant Extraction . . . 30
  - Runtime Detection . . . 31
- Implementation . . . 32
  - Training Mode . . . 33
  - Detection Mode . . . 34
- Evaluation . . . 35
  - Experiment Setup . . . 35
  - Detection Effectiveness . . . 37
  - Performance Overhead . . . 40
- Discussion . . . 41

IV. SECURING DATABASES FROM LOGIC FLAWS IN WEB APPLICATIONS . 42

Overview . . . 42
Attacks and Running Example . . . 43
System Model . . . 46
Approach . . . 48
Approach Overview . . . 48
SQL Signature Construction . . . 49
Application State Inference . . . 50
Data Constraint Inference . . . 51
Invariant Extraction . . . 52
Runtime Detection . . . 55
Implementation . . . 55
Evaluation . . . 57
Experiment Setup . . . 57
Detection Effectiveness . . . 59
Performance Overhead . . . 60
Discussion . . . 60

V. AUTOMATIC DISCOVERY OF LOGIC VULNERABILITIES WITHIN WEB APPLICATIONS . . . 62

Overview . . . 62
Problem Description . . . 63
Illustrative Example . . . 63
Problem Formulation . . . 64
Approach . . . 67
High-level Overview . . . 67
State Construction . . . 68
Input Symbolization . . . 69
Output Symbolization . . . 70
Test Input Generation . . . 72
Output Evaluation . . . 73
Implementation . . . 73
Phase I: Trace Collection . . . 74
Phase II: FSM Inference . . . 75
Phase III: Testing . . . 75
Evaluation . . . 77
Experiment Results . . . 77
Discussion . . . 81

VI. EXPLOITING WEB APPLICATIONS FOR LOGIC VULNERABILITIES OF DATABASE ACCESS . . . 82

Overview . . . 82
Problem Description . . . 83
Illustrative Example . . . 83
System Model . . . 86
Problem Formulation . . . 87
Approach . . . 89
High-level Overview . . . 89
Trace Collection . . . 90
SQL Signature Construction . . . 91
Constraint Inference . . . 92

Vulnerability Exploitation . . . 95
Implementation . . . 99
Phase I: Constraint Inference . . . 100
Phase II: Vulnerability Exploitation . . . 100
Evaluation . . . 101
Experiment Results . . . 101
Discussion . . . 104

VII. AUTOMATED BLACK-BOX DETECTION OF ACCESS CONTROL VULNERABILITIES IN WEB APPLICATIONS . . . 106
Overview . . . 106
Problem Description . . . 107
Database Access Model . . . 107
Access Control Vulnerability . . . 110
Approach . . . 111
High-level Overview . . . 111
Policy Inference . . . 112
Vulnerability Detection . . . 118
Implementation . . . 120
Crawler . . . 120
Trace Collection . . . 122
Inference & Testing . . . 123
Evaluation . . . 123
Details of Vulnerabilities . . . 125
Discussion . . . 127
Comparison with LogicScope and EXPELLER . . . 128

VIII. CONCLUSIONS AND FUTURE WORK . . . 129
Summary of Chapters . . . 129
Limitations . . . 130
Future Research Directions . . . 131

BIBLIOGRAPHY . . . 133

LIST OF TABLES

Table Page

II.1. Summary of Existing Defenses against Logic Attacks 17

III.1. Summary of Evaluated Web Applications (Evaluation of BLOCK) 36

III.2. Summary of Training Set (Evaluation of BLOCK) 37

III.3. Summary of Detection Result (Evaluation of BLOCK) 38

IV.1. Summary of Web Applications (Evaluation of SENTINEL) 58

IV.2. Summary of Training Set (Evaluation of SENTINEL) 59

IV.3. Summary of Detection Result (Evaluation of SENTINEL) 60

V.1. Summary of Traces and Inferred FSMs (Evaluation of LogicScope) 78

V.2. Summary of Testing Results (Evaluation of LogicScope) 78

VI.1. Summary of Constraint Inference (Evaluation of EXPELLER) 102

VI.2. Summary of Testing Results (Evaluation of EXPELLER) 103

VII.1. Summary of Web Applications for Evaluation (Evaluation of BATMAN) . . . 123

VII.2. Summary of Code Coverage (Evaluation of BATMAN) 124

VII.3. Summary of Policy Inference (Evaluation of BATMAN) 124

VII.4. Summary of Testing Results (Evaluation of BATMAN) 125

LIST OF FIGURES

Figure		Page
I.1.	Overview of Web Application	2
I.2.	Summary of Prototype Systems	7
III.1.	An Example of a Vulnerable Web Application	23
III.2.	A Stateless View of a Web Application	26
III.3.	The Procedure of Web Page Symbolization	28
III.4.	Overview of BLOCK System	33
III.5.	Training Mode (BLOCK)	33
III.6.	Detection Mode (BLOCK)	34
III.7.	Number of Invariants vs. Training Set Size (Scarf)	38
III.8.	Summary of Performance Overhead (Evaluation of BLOCK)	40
IV.1.	An Example Vulnerable Web Application: *SimpleOAK*	45
IV.2.	An EFSM Model of a Web Application	47
IV.3.	Partial EFSM Model of *SimpleOAK*	48
IV.4.	An SQL Query Skeleton Extraction	50
IV.5.	Invariant Extraction for SQL Signature Dependency	54
IV.6.	Overview of SENTINEL System	55
IV.7.	The Analyzer Component	57
IV.8.	Summary of Performance Overhead (Evaluation of SENTINEL)	61
V.1.	Example Application	63
V.2.	The FSM Representation of the Example Application	65
V.3.	Approach Overview	68
V.4.	Output Symbolization	71
V.5.	Test Input Generation	72
V.6.	Prototype System Architecture (LogicScope)	74

V.7. Workflow of Testing Controller 76

VI.1. Example Application: *SimpleOAK* 85

VI.2. System Model 87

VI.3. Approach Overview 90

VI.4. HTTP Interaction Example 90

VI.5. Query Parameter Membership Constraint 94

VI.6. Prototype System Architecture (EXPELLER) 99

VII.1. Example of Database Access via a Web Application 107

VII.2. An Example Vulnerable Web Application 110

VII.3. Trace Structure 112

VII.4. Column-based Filter Inference 113

VII.5. Second-Order Relationship Example 116

VII.6. Prototype System Architecture (BATMAN) 120

CHAPTER I

INTRODUCTION

Three-tier web architecture has become the de-facto solution for delivering information and business services over the Internet. The center of this architecture is a web application, which implements the business logic, accesses the sensitive information (e.g., financial or health) stored at the back-end database and interacts with the users through the front-end web server. The increasing popularity of web applications can be attributed to several factors, including remote accessibility, cross-platform compatibility, and fast development and deployment. AJAX (Asynchronous JavaScript and XML) technology also greatly enhances the user experiences of web applications with better interactiveness and responsiveness.

On the other side of the coin, web applications have also become a primary and valuable target for cyber attacks, which brings serious security concerns for all users and businesses that rely on web applications. According to a recent survey [65], the attacks against web applications now account for 63% of all the Internet exploits in 2009. There are several reasons behind this trend. First, web applications are designed to be open and accessible to a large number of users. This also facilitates easy access from malicious attackers. Second, a dominating percentage of web applications are designed to connect with database backend information systems and an enormous amount of sensitive information exists. This provides economic incentives to the attackers to compromise web applications as they can gain a huge amount of profit on the underground market from the sales of breached data. A breach report from Verizon [66] shows that web applications now reign supreme in both the number of breaches and the amount of data compromised. Third, as web applications become deeply embedded in business activities and are required to support sophisticated functionalities, the design and implementation of web applications are becoming more complicated. The increasing complexity is confronted with the insufficient security assurances from both the widely-used web application development and testing frameworks and the web developers with insufficient security skills. As a result, a high percentage of web applications deployed on the Internet are exposed with security vulnerabilities. According to a report by the Web Application Security Consortium, about 49% of the web applications they reviewed contain vulnerabilities of high risks and more than 13% of the websites can be compromised

completely [72]. Another report [73] reveals that over 80% of the websites on the Internet used to have at least one serious vulnerability.

Background

A web application is a client-server application that is executed over the Web platform. It is an integral part of today's Web ecosystem that enables dynamic information and service delivery. As shown in Figure I.1, a web application consists of code on both the server side and the client side. On the server side, the web application receives user inputs via HTTP requests from the client (i.e., browser), and interacts with the local file system, the back-end database or other components for data access and information retrieval. Its outputs (i.e., HTML page) are sent to the client through HTTP responses. On the client side, HTML pages are rendered and the client-side code (i.e., JavaScript) embedded in the HTTP responses is executed by the web browser. The client-side code can also communicate with the server-side code asynchronously without interfering with the display of the existing HTML page via AJAX, which dynamically updates the page. We describe two unique characteristics of web applications as follows.

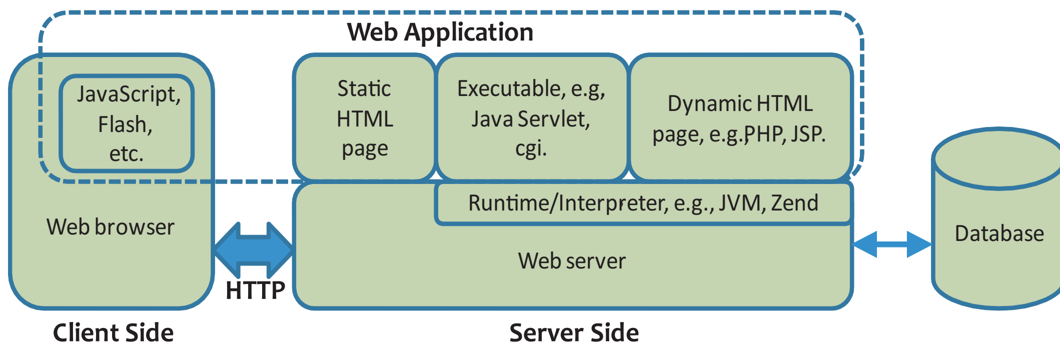


Figure I.1: Overview of Web Application

Session Management: Web applications adopt the abstraction of a web session to identify and correlate a series of web requests from the same user during a certain period of time. A set of session variables (or session data) is associated with a web session, which can be used by the application to record the states from the historical web requests that affect the future execution of the web application (i.e., application session state). The session variables can be maintained either at the client side (via a cookie, a hidden form or a URL) or at the server

side (in a file or using a database). In the latter case, a unique identifier (session ID) is defined to index the explicit session variables stored at the server side and issued to the client. Most web programming languages (e.g., PHP and JSP) and frameworks offer developers a collection of functions for managing web sessions. For example, in PHP, *session_start()* can be called to initialize a web session and a pre-defined global array *$_SESSION* can be employed to contain session variables. In either case, the client plays a vital role in maintaining the session information.

Logic Implementation: The implementation of a web application's logic is usually manifested as enforcing the control flow of the application and protecting its sensitive information and operations. This is usually achieved through explicit security checks in the source code or implied by the navigation paths presented to users (i.e., interface hiding). Explicit security checks examine the conditions over certain security critical variables, whose values are drawn from session variables or persistent data objects in the database before sensitive information and operations can be accessed. Interface hiding only allows accessible resources and operations to be exposed to users as web links.

Problem Description and Research Goal

As web applications get increasingly complex to support sophisticated business functionalities, an emerging class of vulnerabilities, which are referred to as logic vulnerabilities, (a.k.a, logic flaws) have attracted increasing attention in recent years. The attacks that target on these vulnerabilities, which are referred to as logic attacks or state violation attacks, have posed serious security threats. For instance, in June 2010, it was reported that a vulnerability of the AT&T website allowed an attacker to harvest Apple iPAD subscribers' emails by enumerating ICC-ID numbers, which affected over 100,000 Apple customers [2]. As another example, in 2011, a huge amount of credit card information was leaked due to a logic vulnerability within the Citigroup website [15]. A recent report shows that three among top ten security risks for web applications [49][1] can be attributed to logic vulnerabilities within web applications.

Logic vulnerabilities are closely related with the intended functionality of a web application. Some vulnerabilities occur within the business logic patterns that are commonly seen in web applications. Example of such vulnerabilities include access control vulnerabilities, which allows

[1] Missing Functional Access Control (newly added in 2013), Insecure Direct Object Reference and Unvalidated Redirects and Forwards.

the attackers to access unauthorized sensitive information or operations. Some vulnerabilities are linked to the functionalities that are specific to an application. For example, an e-commerce website, which supports coupon during checkout, may be vulnerable to a vulnerability which allows repeated application of the same coupon.

Ensuring correct implementation of logic specification faces several challenges that are fundamental to the development of a web application. First, web applications are usually structured in a "decentralized" way and rely on additional state maintenance mechanisms to keep the application state over the "stateless" HTTP protocol. As a result, the possible execution paths leading to sensitive operations that are dispersed throughout the web application, making it very difficult to identify all the security check points. Second, a large number of web applications, developed in object-oriented programming languages, meanwhile rely on relational databases for persistent data storage. The impedance mismatch between the relational data model and the in-memory data model complicates the security checks over these data objects. Third, modern web application development heavily relies on development frameworks (e.g., Django, Rails). There exist gaps between the expectations from developers and the actual functionalities provided by these frameworks. Security vulnerabilities can be introduced when certain functions are used under the wrong assumptions. A common case is that redirection headers are sent to users when the security checks fail. However, even the users are redirected to a different page, the issuance of the redirection header does not stop further program execution, which may trigger sensitive operations after the security checks fail. This vulnerability is referred to as Execution After Redirection (EAR) [23].

A web application with logic vulnerabilities is vulnerable to logic attacks. Common logic attack vectors include:

- Forceful browsing [63], where the attackers directly access hidden but predictable web links to retrieve unauthorized web pages.
- Parameter tampering [6], which is launched by manipulating certain values in web requests to craft inputs that are beyond the expected input domain under the current user.

Logic attacks are radically different from input validation attacks. Input validation attacks manipulate the syntax of web requests to generate malformed web application inputs, while logic

attacks are disguised as syntactically valid web requests that carry malicious intentions to violate the intended application logic.

To date, very few works have been devoted to the study of the logic vulnerabilities and effective measures to mitigate logic attacks are yet to be developed. Most existing works only target on one specific type of logic vulnerability, and are limited by the availability of application source code and the applicability to specific development languages and platforms. There are three major challenges for addressing logic flaws and logic attacks.

- **Application logic is specific to each web application**. Logic vulnerabilities and attacks are specific to the functionality of a web application. There is no general specification which can characterize the logic vulnerabilities and attacks for all web applications.
- **Logic specification of a web application is rarely available**. Although new tools are emerging to facilitate the specification-based design of web applications [13, 19, 77], the majority of online web applications come without any logic specifications.
- **The implementation of a web application is heterogeneous**. Web applications are built using diverse programming languages and frameworks (e.g. PHP and JSP). As such, techniques that infer application specifications from the source code face the following limitations: (1) the technique developed for one language and platform cannot be readily applied to another, especially when the application logic is implemented across different platforms (e.g. for AJAX applications, the application logic is split at both client and server side); (2) the quality (i.e., correctness and accuracy) of the inferred logic specification is limited by their capabilities for handling language-level details. For example, it is extremely difficult to analyze PHP web applications that contain object-oriented code, which leads to inaccurate representations. (3) the quality of the inferred specification is also highly dependent on the quality of the web application implementation. The definitions of program blocks and variables greatly affect the specification inference.

In the light of the above challenges, *the research goal of this dissertation is to secure legacy web applications from logic attacks and vulnerabilities*. In particular, we aim to develop security techniques that can:

- **detect** logic attacks launched against web applications and

- **prevent** logic attacks by identifying logic vulnerabilities within web applications.

These techniques should be 1) automated - they require minimal amount of human intervention; 2) effective - they can detect a broad range of logic attacks and vulnerabilities with minimum number of false positives incurred; 3) scalable - they can be easily applied to a variety of web applications that are developed with different programming languages and frameworks.

Research Approach and Dissertation Contributions

Our approach consists of two major steps: *specification inference* and *specification utilization*. Our specification inference technique does not require application source code but solely relies on the observation over the external behavior of a web application. Note that the functionality of an application is correctly and fully manifested when users follow its navigation paths [63]. We infer the application specification from its traces (including web requests/responses, SQL queries/responses and session variables) collected during its normal execution. Based on the inferred specification, we take two approaches to defending against logic attacks: runtime detection of of logic attacks (i.e., the *defensive* approach) and discovery of logic vulnerabilities within web applications (i.e., the *preventive* approach). The defensive approach, based on misuse detection, aims at identifying the deviation of the application behavior from the inferred specification. The preventive approach, based on directed fuzzing, generates concrete attack vectors from the inferred specification to expose the logic vulnerabilities within the application. The defensive approach can be utilized to protect the potentially vulnerable web applications that cannot be taken offline for vulnerability analysis, while the preventive approach, which constructs test can help developers to identify and fix logic flaws within the application implementations so that they are immune to logic attacks.

In this dissertation, we present several techniques for automatically deriving logic specifications based on different application models and utilizing the specifications to mitigate logic attacks. We implement a prototype system for each technique and evaluate it over a set of open source web applications. Figure I.2 summarizes the five prototype systems we developed. BLOCK and SENTINEL can be used to detect logic attacks at runtime, while LogicScope, EXPELLER and BATMAN can be used to identify logic vulnerabilities within web applications. LogicScope leverages the same amount of information as BLOCK, including web requests, web responses

and session variables. EXPELLER leverages the same amount of information as SENTINEL, including SQL queries, SQL responses and session variables. BATMAN integrates information of both web requests/responses and SQL queries/responses, and does not require any session variables. The experiment results demonstrate the effectiveness of our techniques and prototype systems. The contributions of this dissertation are listed as follows.

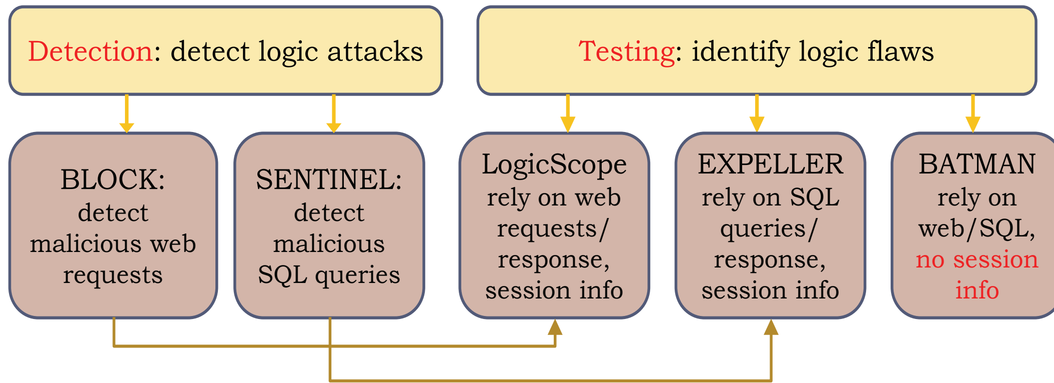


Figure I.2: Summary of Prototype Systems

Detection of Logic Attacks

- We present a black-box approach (and the first) for detecting logic attacks. We present a stateless model of web application by flattening the finite state machine (FSM) model. Based on this model, we characterize the logic specification by inferring the relations among web requests, responses and session variables in the form of invariants and use the invariants for detecting malicious web requests. We implement a prototype detection system BLOCK for PHP web applications. Details are provided in Chapter III.

- We present an approach to detecting malicious SQL queries, which are triggered to exploit the database access logic flaws within web applications. We model the web application as an Extended Finite State Machine (EFSM) by associating data constraints with state transitions, and extract invariants from SQL queries, responses and session variables. This technique systematically utilizes the persistent information in the database (i.e., SQL response) for deriving a more complete and accurate logic specification. We implement a

prototype system SENTINEL for PHP web applications. Details are provided in Chapter IV.

Prevention of Logic Attacks

- We present the first systematic source-code free approach to identifying logic flaws within web applications based on BLOCK. We present a formalization of logic vulnerability based on the FSM model and exploit the discrepancies between the intended FSM and the implemented FSM by constructing test inputs that reflect the logic attack vectors. We implement a prototype system LogicScope. Details are provided in Chapter V.

- We present a source-code free approach, which focuses on identifying logic flaws of database access within web applications, based on SENTINEL. We characterize the logic specification as a set of security constraints over SQL queries and exploit the application by constructing malicious web requests that violate the inferred constraints. We implement a prototype system EXPELLER for PHP web applications. Details are provided in Chapter VI.

- We present a black-box technique for identifying the most prevalent logic vulnerability: access control vulnerability, within web applications. This technique is based on a novel data access model that unifies the SQL queries for write operations and web responses for read operations, and automatically extracts the intended access control policy by analyzing data access operation patterns within and across different users and roles. This technique does not rely on any server-side session variables, thus it can naturally handle web applications developed in different languages and platforms without additional efforts. We implement a prototype system BATMAN and evaluate it over open source PHP and JSP web applications. Details are provided in Chapter VII.

Dissertation Organization

The remainder of this dissertation is organized as follows. Chapter II presents the existing literature related with the work in this dissertation. Chapter III and Chapter IV present our black-box approach to detecting logic attacks, as well as the implementation and evaluation of the BLOCK and SENTINEL systems. Chapter V and Chapter VI present our source-code

free approach to identifying logic vulnerabilities within web applications, as well as the implementation and evaluation of the LogicScope and EXPELLER systems. Chapter V presents our black-box technique for discovering access control vulnerabilities, as well as the implementation and evaluation of the BATMAN system. Finally, chapter VIII concludes this dissertation.

CHAPTER II

RELATED WORK

The work in this dissertation is mostly related with developing security techniques to address logic attacks and vulnerabilities within web applications. Our work is also related to several other areas of research, including software specification inference, web database security and test input generation. In this chapter, we review the existing work and highlight our contributions.

We summarize the existing work in Table II.1. We highlight two important dimensions: *specification source* and *defense mechanism*. We also fit our techniques described in this dissertation into the big picture. From the table, we can see that our major contribution is a new category for specification inference through black-box analysis, which does not require application source code for either static analysis or instrumentation.

Existing Defenses against Logic Attacks

A large body of existing work has been proposed to secure web applications and defend against a variety of web attacks. However, most investigate input validation attacks (e.g., [36, 71, 41, 35]), while only a few try to address logic attacks. This is not surprising because input validation vulnerabilities are independent of the specific web application and can be captured via a general specification: insecure information flow. The information flow model can be applied to identify or fortify the insufficient or erroneous sanitization routines within web applications to defend against input validation attacks.

In contrast, logic vulnerabilities are multi-faceted without such a general specification. Existing works that address logic vulnerabilities have followed two directions: 1) targeting on special types of vulnerabilities that are associated with common application functionalities, such as authentication, access control, etc; 2) aiming at general logic vulnerabilities that can be dependent on the functionalities of each application (referred to as application-specific logic vulnerabilities hereinafter). In this case, the application's intended functionality (i.e., specification) is required to tackle the logic vulnerabilities and attacks. Such a specification can be explicitly specified by developers during software development. In the absence of such a specification, which is commonly seen in practice, it needs to be inferred from the application implementation. Application

specification inference is challenging, since a general method needs be developed to handle a number of heterogeneous web applications and platforms to minimize manual efforts involved. In what follows, we categorize the existing works by the phase, when they can be applied along the lifecycle of a web application, into three classes: *Secure Construction of New Web Applications*, *Security Analysis/Testing of Legacy Web Applications* and *Runtime Protection of Legacy Web Applications*.

Secure Construction of New Web Applications

this class of techniques aim to *construct* secure web applications, ensuring that no potential vulnerabilities are introduced during the web application development. These techniques are usually carried out through the design of new web programming languages or frameworks that are built with security mechanisms, which root out the target vulnerabilities. These techniques are most applicable to the development of new secure web applications, but not suitable for fixing the vulnerable legacy applications due to the huge amount of redevelopment effort that is required.

To enforce authorization policies in web applications, existing works have adopted information flow model to prohibit sensitive information from flowing to unauthorized principals. Recall that, to address input validation vulnerabilities, information flow model has been applied to prevent the untrusted user data from flowing into trusted web contents. Thus, SIF [13] and Swift [12], which use security-typed language Jif to track the information flow, provide a web application framework that can enforce both input validation policies and authorization policies.

SELinks [19] is a programming framework that extends the LINKS web programming language with FABLE [64], a type system for defining and enforcing custom, label-based security policies. Similar to Jif, the type of sensitive data in Fable is annotated with a security label; different from Jif, the semantics of this label is user-defined, where programmers define the interpretation of labels in special enforcement policy functions separated from the rest of the program. FABLE can be used to define and enforce a wide range of policies including access control, data provenance and information flow policies, while Jif only supports information flow policy.

Resin [77] is a system that allows programmers to specify application-level data flow assertions using policy objects and define the data flow boundaries using filter objects. RESIN operates within a language runtime (e.g, Python or PHP interpreter). It tracks application data as it flows

through the application, checks data flow assertions on every executed path and invokes filter objects when data crosses a data flow boundary, such as when writing data to the network or a file. A variety of vulnerabilities can be mitigated using Resin, such as script injections, missing access control checks. Compared to SIF [13] and SELinks [19], Resin allows programmers to reuse the application's existing code and thus avoids the large amount of annotations and instrumentations required by security-typed languages. However, Resin cannot track implicit data flow, such as program control flow, data structure layout, which is the capability provided by security-typed languages.

In addition to information flow models, the principle of least privilege and privilege separation have also been applied to facilitate constructions of web applications that minimize the side effects of an attack. Capsules [40] is a web development framework based on an object-capability language Joe-E [44] for enforcing isolation and facilitating the practice of the principle of least privilege. An web application is partitioned into isolated components, each of which is given a limited set of explicitly-specified privileges. This technique minimizes the damages caused by vulnerable components, especially third-party code, and facilitate security reviews and verification. However, it cannot guarantee each application component is free of vulnerabilities.

Security Analysis/Testing of Legacy Web Applications

This class of techniques aim to identify vulnerabilities within web applications through program *analysis* (usually referred to as vulnerability analysis) and *testing* techniques. Additional efforts have to be spent on fixing the discovered vulnerabilities and retrofitting the web applications, either manually or automatically. They are usually designed for handling a specific programming language or framework and can not be easily generalized for another one. A key challenge for this class of techniques is the trade-off between the completeness and correctness of vulnerability discovery. Static analysis, which examines application source code without execution, tends to be more complete in vulnerability discovery than dynamic analysis (including testing), which observes the application behavior through execution. On the other hand, static analysis is likely to introduce more false alerts, while dynamic analysis can typically guarantee the correctness of the identified vulnerabilities through its capability of generating concrete attack vectors.

Logic vulnerabilities within a legacy web application originate from the discrepancies between its intended functionalities (i.e., specification) and its implementation. Once the specification of an application is defined, the existence of logic vulnerabilities can be identified.

UrFlow [11] is able to statically verify a variety of security policies, including both information flow and access control policies, within database-backed web applications. UrFlow requires developers to specify policies in the form of SQL queries and employs symbolic execution and theorem proving to automatically verify if the program behaviors conform to those policies.

Rubyx [9] is a symbolic execution framework for Ruby-on-Rail web applications. Rubyx allows developers to specify security policies using a set of programming interfaces and verifies those policies automatically. Rubyx is able to identify input validation vulnerabilities, CSRF, insufficient authentication as well as application-specific logic flaws, depending on the security policies defined.

When the application specification is not provided by the developers, it has to be first inferred from its implementation for identification of logic flaws. There are two general approaches for specification inference: 1) static analysis, which extracts the logic specification from the application's source code, and 2) dynamic analysis, which generates the specification by observing the application's runtime behavior.

Static Analysis

Sun et al. [63] perform role-specific analysis of PHP web applications for identifying access control vulnerabilities. They first specify a set of roles with total order. Then sitemaps are constructed for different roles in a web application based on explicit navigation links. By comparing per-role sitemaps, privileged pages can be identified. Then they analyze whether direct access to privileged pages from unauthorized roles is allowed, which indicates missing access control checks.

RoleCast [61] also tries to identify missing access control checks in PHP web applications. In RoleCast, roles are defined based on the common functionality and security logic. The set of user roles are inferred from the partitions of the file contexts on which security sensitive events are control dependent. Within a role, RoleCast identifies security-critical variables and performs role-specific consistency analysis over security-critical variables to find missing security checks.

Doupé et al. [23] address a particular type of logic vulnerability called Execution After Redirect (EAR), where the application continues execution after developer-intended redirection,

resulting in violation of intended control flow and unauthorized execution. In particular, they present a static analysis technique to identify such vulnerabilities within Ruby-on-Rail web applications. They extract the control flow graph from the application source code and identify the control paths that lead to privileged code after redirection routines are called as potential vulnerabilities.

MiMoSA [4] identifies workflow violations that are introduced by unintended navigation paths, in addition to data flow attacks that exploit the input validation vulnerabilities, among multiple modules. MiMoSA infers a workfow graph of the application as a specification through two steps: first, intra-module analysis extract a "state view" of each module (a PHP file in this case) is analyzed by determining its pre-condition, post-conditions, and sinks; then inter-module analysis links the state views to derive the workfow graph of the entire application. Model checking technique is applied on the workflow graph to identify unintended navigation paths.

Dynamic/Hybrid analysis

Waler [29] can automatically discover application-specific logic flaws. First, Waler extracts value-based invariants for session variables and function parameters by observing normal executions and uses them as the logic specification. Then model checking technique is applied to identify possible violations of inferred invariants. They also filter spurious invariants by analyzing the program control paths and capturing the relationship between session variables and database objects.

While Waler focuses on the analysis of server-side code, NoTamper [6] aims at detecting parameter tampering opportunities behind web forms within AJAX applications. Such vulnerabilities are caused by the inconsistencies between the client-side and server-side validation. NoTamper extracts the constraints over parameters within the form from the client-side JavaScript code and generates malicious input vectors by negating those constraints. The web responses triggered by both benign and malicious inputs are examined to determine whether the form is vulnerable to parameter tampering. The black-box technique used by NoTamper may produce both false positives and false negatives. WAPTEC [7] enhances NoTamper by applying white-box analysis to the server-side code to reduce false positives and identify the vulnerabilities that NoTamper fails to discover.

Runtime Protection of Legacy Web Applications

This class of techniques aim to harden and *protect* potentially vulnerable web applications against external exploits by building a runtime environment that supports its secure execution. They usually either 1) place safeguards (e.g., proxy) that separate the web application from other components (e.g., the browser and the database) in the Web ecosystem or 2) instrument the infrastructure components (e.g., language runtime and web browser) to monitor its runtime behavior and identify/quarantine potential exploits. These techniques are usually scalable to handle a large number of web applications, even with different languages or platforms, with the price of additional performance overhead due to instrumentation.

CLAMP [53] addresses access control vulnerabilities that can be exploited by a variety of attacks, including logic attacks, SQL injections and even web server compromises, and protects sensitive user data by isolating application components running on behalf of different users through virtualization. CLAMP assigns a virtual web server instance to each user's web session, so that the user can only access his own data. However, CLAMP cannot be applied to applications with shared data among users. Also, CLAMP requires a small amount of changes to the application code.

Nemesis [22] addresses a wide range of authentication bypass and access control vulnerabilities in legacy web applications. It uses a shadow authentication system to infer successful user authentication without depending on the potentially vulnerable authentication mechanism in the application. It employs dynamic information flow tracking technique to track the flow of user credentials through the application's language runtime and combines authentication information with programmer-supplied access control rules to ensure that only properly authenticated users are granted access to any privileged resources or data. Nemesis does not require changes to the application code, but need the application developers to provide annotation information for verifying the authentication credentials and explicitly specify the access control policies.

Swaddler [20] applies anomaly detection technique for the discovery of state violation attacks. In particular, Swaddler establishes statistical models of session variables for each program block during its normal execution, which indicate the application state when that program block is executed. At runtime, this set of statistical models are evaluated to determine whether the application state is legitimate when the current program block is executed. Different from Nemesis

and CLAMP, which only focus on certain types of common logic vulnerabilities (e.g., authentication, access control), Swaddler provides a unified approach for a wide range of application-specific logic vulnerabilities.

Besides the vulnerabilities that are embedded in the server-side code, recent works [33, 67] also tackle the logic vulnerabilities within the client-side code for AJAX applications. [33] extracts a control-flow graph of URLs from the client-side HTML and JavaScript code as the client specification using static analysis. This graph is then used in a reverse proxy to monitor client behaviors and detect malicious activities against server-side web applications.

Ripley [67] detects malicious user behaviors within AJAX applications by leveraging replicated execution. Essentially, the client-side computation is exactly emulated on the trusted server tier, where each client-side event is transferred to the replica of the client for execution. The discrepancies between the execution results are flagged as exploits.

Summary and Our Contributions

Despite the existence of the above research efforts, as summarized in Table II.1, securing web applications from logic flaws and attacks still remains an under-explored area. Among the limited number of works, most of them only address a special type of application logic vulnerabilities [23, 63, 61, 53, 22, 6, 7], such as authentication and access control vulnerabilities, and inconsistencies between client and server validations. The fundamental difficulty for tackling general logic flaws is the absence of application logic specification. The absence of a general and automatic mechanism for characterizing the application logic may be the inherent reason for the inability of application scanners and firewalls to handle logic flaws and attacks [25, 5].

Several recent works try to develop a general and systematic method for automatically inferring the specifications for web applications, which in turn facilitates automatic and sound verification of the application logic. One of the key observations that these work [4, 29, 20] leverage in establishing the application specification is that the application intended behavior is usually revealed under its normal execution, when users follow the navigation paths. In [33], similar assumption is made for well-behaved clients, where they are expected by the server to invoke the URLs in a particular sequence with particular arguments. In order to infer the application logic, one class of methods leverage the program source code [20, 29]. As a result, the inferred specification is highly dependent on the development language and platform. Our techniques try

to infer the application specification by observing and characterizing the application's external behavior. The noisy information observed from external behaviors may lead to an inaccurate specification through this method. Moreover, web application maintains a large amount of persistent states in the database. Correctly identifying these states to accurately characterize the application logic is still challenging.

Table II.1: Summary of Existing Defenses against Logic Attacks

	Secure Construction	Security Analysis/Testing	Runtime Protection
Specification Source			
Specified by Developer (explicitly or implicitly)	SIF [13, 12], SELINKS [19], Resin [77], UrFlow [11], Capsules [40]	EAR [23]	Nemesis [22], CLAMP [53]
Static Inference (source code)	N/A	MiMoSA [4], Sun's work [63], RoleCast [61], NoTamper [6], WAPTEC [7]	Arjun's work [33]
Dynamic Inference (via instrumentation)	N/A	Waler [29], WAPTEC [7]	Swaddler [20], Ripley [67]
Dynamic Inference (black-box analysis)	N/A	*LogicScope, EXPELLER, BATMAN*	*BLOCK, SENTINEL*
Defense Mechanism			
Static Checking	SIF [13, 12], UrFlow [11]	MiMoSA [4], RoleCast [61], EAR [23], Waler [29], Sun's work [63]	N/A
Dynamic Checking/Enforcement	SIF [13, 12], SELINKS [19], Resin [77], Capsules [40]	NoTamper [6], WAPTEC [7], *LogicScope, EXPELLER, BATMAN*	Nemesis [22], CLAMP [53], Swaddler [20], Ripley [67], Arjun's work [33], *BLOCK, SENTINEL*

Software Specification Inference

Software specification is essential for verification of program behaviors and program testing. However, a complete and machine understandable specification is rarely available. Thus,

researchers are motivated to study the problem of inferring software specifications. Static inference techniques analyze the program code to extract the partial orders of function calls [39], while dynamic inference techniques try to profile the program behavior by mining program execution traces. The Daikon engine [27], the most famous tool in this field, extracts value-related invariants by matching invariant templates to expressions. Strauss [1] formalizes the specification mining as a grammar inference problem and learns probabilistic finite state automata (PFSA) from traces. Perracotta [76] mines two-letter alternating patterns of functions from imperfect traces. Gk-tail [43] builds extended finite state machine (EFSM) combining both value-related and temporal properties.

Our techniques fall into the category of dynamic inference techniques. Different from these generic software specification inference methods, our work observes the external information (i.e., web requests/responses and SQL queries/responses), does not require instrumentation to generate execution traces and can be applied to distributed client/server web applications.

Web Database Security

Most database systems currently in use implement role-based access control mechanisms to regulate database accesses. However, when the database is connected to a web application, which we refer to as web database, it cannot differentiate between the end users who are actually operating the application and trusts all the queries issued by the application. Thus, if an attacker exploits the web application successfully, he can trick the application into sending malicious SQL queries, leading to a "confused deputy" problem [17].

Access Control

A straightforward approach to addressing the aforementioned confused deputy problem is to identify the individual users that operate the web application and perform a fine-grained access control for each user. Roichman et al. [56] leverage parameterized views for user identification and implement access control at a finer granularity at the database layer. However, such a "user-based" approach depends on the adoption of parameterized views, which requires retrofitting the entire database system and all the schema. As an alternative approach, Felt et al. [30] propose Diesel to enforce privilege separation between different application modules. The "module-based"

approach is only capable of confining the potential damages to a certain vulnerable module and cannot defend against the attacks that exploit the vulnerable module.

Our technique (SENTINEL in Chapter IV) neither relies on user identification nor fine-grained access control. Instead, we characterize the application state and data constraints when each SQL query is issued by the application. Our "state-based" approach requires minimal efforts, can be easily integrated with existing systems and effectively identify malicious SQL queries.

Intrusion Detection

Another approach is to build an intrusion detection system (IDS) for database systems. Lee et al. [42] build a signature-based IDS by learning a set of legitimate SQL query fingerprints from database transactions. While the set of fingerprints is exactly the same as the set of SQL signatures we extract from traces, we further associate each SQL signature with invariants to detect state violation attacks. Chung et al. [14], Kamra et al. [37] and Chen et al. [10] establish anomaly-based IDS by deriving user profiles of database accesses from audit logs, while Roichman et al. [57] rely on parameterized database views [56]. However, all of their systems face the same challenge of scaling up to handle a huge number of users. Instead, our technique (i.e., SENTINEL in Chapter IV) characterizes the application behavior, not the individual user behavior. Thus, our approach is naturally scalable for handling a large population of users.

Test Input Generation

Random test generation (i.e., fuzz testing) tends to achieve limited coverage for testing. Thus, it is usually enhanced with guided/directed mutations of well-formed inputs to discover vulnerabilities. FLAX [54] is a taint-guided black-box fuzzing technique for identifying client-side input validation vulnerabilities in web applications. By leveraging the knowledge of sensitive sinks, a large proportion of the input space can be pruned. DART [31] blends testing with model checking, which systematically explores the program space for directed input generation. Another technique that can be applied for directed input generation is symbolic execution, which is usually combined with constraint solving. The program execution is tracked using symbolic rather actual values to gather constrains on inputs capturing how the program uses them. Then, the collected constraints are negated one by one and solved with a constraint solver, producing new inputs that

exercise different control paths in the program. SAGE [32] leverages this technique on well-formed input in white-box manner to discover vulnerabilities. EXE [8] is able to automatically generate attack vectors. Kudzu [58] is a symbolic execution framework for JavaScript and aims to identify code injection vulnerabilities in the client-side code that result from untrusted data provided as arguments to sensitive operations. Additionally, Halfond et al. [34] apply this strategy to infer the web application interfaces for improved testing and analysis of web applications.

Symbolic execution can also be combined with concrete execution to avoid redundant test cases and false warnings [60], which is referred to concolic testing. Emmi et al. [26] *concolically* (i.e., both symbolically and concretely) execute server-side code and analyze executed SQL queries to find missing database records to improve branch coverage in testing. Pan et al. [52] focus on generating effective inputs for testing database applications given an existing database state, without incurring the overhead of generating new database state. All of the above works require analyzing the application source code to incorporate techniques, such as symbolic execution and model checking and cannot identify logic vulnerabilities within web applications. Instead, our test input generation (i.e., LogicScope in Chapter V, EXPELLER in Chapter VI, BATMAN in Chapter VII) is based on an inferred specification in a black-box manner and echoes logic attack vectors to expose logic vulnerabilities.

CHAPTER III

A BLACK-BOX APPROACH FOR DETECTION OF STATE VIOLATION ATTACKS

In this chapter, we present BLOCK, a BLack-bOx approach for detecting state violation attaCKs. To our knowledge, this is the first black-box technique that addresses this problem. We regard the web application as a stateless system and infer the intended web application behavior model by observing the interactions between the clients and the web application. We extract a set of invariants from the web request/response sequences and their associated session variable values during its attack-free execution. The set of invariants is then used for evaluating web requests and responses at runtime. We develop a system prototype based on the WebScarab proxy and evaluate our detection system using a set of real-world web applications. The experiment results demonstrate that our approach is effective at detecting state violation attacks and incurs an acceptable performance overhead. Our approach is valuable in that it is independent of the web application source code and can fit into a large variety of web application hosting scenarios based on different application frameworks, where the source code may not be available.

The rest of this chapter is organized as follows. We first give an overview of our approach. Then, we present an example web application to show the state violation attacks we focus on. Our system and approach are discussed in detail in the following two sections. Then, we discuss the implementation and demonstrate the evaluation results of the prototype detection system. We conclude with discussions on the limitations of this work.

Overview

The key idea of BLOCK is to infer the intended behavior model of the web application (i.e., specification) by observing the web request/response sequences and their associated session variable values during attack-free executions. In particular, we leverage the stateless property of HTTP and regard the vector of current values of session variables as part of the input along with the web request to the application, the web responses and the updated session variables as the output. In this way, the web application can be approximated as a stateless system. Under this stateless system model, we characterize the application behavior from three aspects

in the form of likely invariants: 1) input invariants, which model the relationship between the web requests and the session variable values, 2) input/output invariants, which capture the relationship between the web request and response as well as the changes in the session variables after the web request is processed, and 3) input/output sequence invariants, which leverage the historical web request/response pair sequences to capture the application states that are not revealed by defined session variables.

Attacks and Running Example

A web application manages the clients' session states to control the access over its restrictive functions and sensitive information, as well as enforce desired state transitions. Although most current web application development frameworks provide session management mechanisms, it is still the developer's responsibility to define and check session variables at appropriate program points, which is usually done in an ad-hoc manner. Three types of vulnerabilities are possibly introduced into the web application: (1) insufficient definition of session variables for differentiating all possible states; (2) insufficient checking of session variables at appropriate program points; (3) erroneous checking of session variables that can be bypassed. All three of these problems make the web application vulnerable to state violation attacks (also referred to as the workflow violation attack in Swaddler [20]). The attacker can launch state violation attacks by sending web requests to the web application, which violate the underlying requirements of expected web requests by the developers at the current application state. We use a small PHP web application (shown in Fig. III.1) which contains several state management vulnerabilities to illustrate state violation attacks. This example is used throughout the chapter to demonstrate how we address these attacks.

The first example of the state violation attack is an authentication/authorization (simplified as auth hereafter) bypass. The web application controls the access over its functions by checking session variables indicating the user privilege before its restrictive functions can be executed. If the application is not at the required state, the web application will redirect the user to the login page, authorization page or an error page. However, if there exists a path leading to the restrictive function with insufficient or erroneous checking of session variables, the attacker is able to bypass the authentication/authorization. The example application demonstrates three

```
<?php
include_once("header.php");
if (isset($_GET['logout'])){
        session_start();
        unset($_SESSION['username']);
        unset($_SESSION['privilege']);
        session_destroy();
        print "You are logged out.<br>";
} else if (isset($_POST['email'])){
        if (validateLogin($_POST['email'], $_POST['passwd'])){
                $_SESSION['username'] = $_POST['email'];
                if ($_POST['email'] == $admin_email){
                        $_SESSION['privilege'] = "admin";
                } else {
                        $_SESSION['privilege'] = "user";
                }
                header("Location:index.php?username
                                =" . $_SESSION['username']);
                exit();
        } else {
                die("Wrong username or password");
        }
}
?>
<form action='login.php' method=post>
username: <input name="email" type="text"><br>
password: <input name="passwd" type="password"><br>
<input name="submit" type="submit"> </form>
<?php  include_once  'footer.html';?>
```

login.php

```
<?php  include_once 'header.php';
logIdentity();
print "<a href='admin2.php'>Next step: change the
title</a>";
include 'footer.html'; ?>
```

admin.php

```
<?php
include_once 'header.php';
if (isset($_GET['username'])){
        $userid = $_GET['username'];
        showUserInfo($userid);
        if ($_SESSION['privilege'] == "admin"){
                print "<a href='admin.php'>Admin link</a><br >";
        }
}
print "<a href='login.php?logout=1'>Logout</a><br>";
include_once 'footer.html';
?>
```

index.php

```
<?php
include_once 'header.php';
if ($privilege != "admin"){
        header("Location: index.php?username
                        =".$_SESSION['username']);
}
if (isset($_POST['title'])){
        modifyTitle($_POST['title']);
}
?>
<form action='admin2.php' method=post>
New title: <input name="title" type="text"><br>
<input name="submit" type="submit">
</form>
<a href='login.php?logout=1'>Logout</a>
<?php
include_once 'footer.html';?>
```

admin2.php

Figure III.1: An Example of a Vulnerable Web Application

cases of auth bypass attacks. *admin.php* and *admin2.php* contain restrictive functions, which should only be accessed by admin users when the session variable *$_SESSION['privilege']* is set to the value of *admin*.

- In *admin.php*, there is no check on the session variable *$_SESSION['privilege']*. The attacker, being either a guest or a regular user, can directly request the page and access the admin functions.

- In *admin2.php*, even though there is an *if* condition check on the session variable $*privilege*, the attacker can append an additional parameter *privilege* to the URL, for example *http://example.com/admin2.php?privilege=admin*, and bypass the auth check. The reason is when the *register_global* option of PHP interpreter is enabled, the parameter attached to the web request will be automatically bound to a global variable, if such a variable does not exist in the current session state. This vulnerability results from the inappropriate or erroneous check on the session variable.

- In *admin2.php*, even when the auth check fails, the attacker is able to execute the restrictive functions after the redirection (i.e., *header* function) by submitting a POST request with the parameter *title* and change the application's title successfully. This is because there is no *exit* function or an additional check after the redirection.

The second example of a state violation attack is parameter manipulation. In many cases, the web application assumes implicit relations between the user's input parameters within web requests and the session state. Such a relationship may also be reflected by web responses returned by the web application. If the application does not check the session state when accepting the web request, the attacker is able to manipulate the input parameters and gain access to unauthorized information. In the example application, after the user logs in, they will be redirected to the *index.php* page, which displays their personal information. The web application assumes the request parameter *username* is always equal to the value of session variable *$_SESSION['username']*. If the equality relationship is not examined when the user's personal information is retrieved, the attacker is able to view any user's information by modifying the *username* parameter within the web request.

The third state violation attack is workflow bypass. A web application usually has an intended workflow, which requires the user to perform a predefined sequence of operations to complete a

certain task. For example, an e-commerce website has a predefined checkout procedure, which instructs the customer to first fill-in the shipping information and then the credit card information before the order can be confirmed and submitted. Such a temporal relationship is enforced by the restrictions over the session state transitions. However, if the session variables are insufficiently defined or checked for guarding the desired state transitions, the attacker is able to bypass certain required steps and violate the intended workflow. The example application requires the admin user to first access *admin.php*, which logs his/her identify (by *logIdentity* function) before they can modify the application title in *admin2.php*. The two steps indicate two different session states and the transition between them should be guarded by the web application. However, there is no session variable defined for indicating whether the identity of the admin user has been logged or not. The attacker can directly point to the *admin2.php* page without their identity being logged.

System Model

As shown in Fig.III.2, a web application is regarded as a stateless system F, which accepts an input m_{in} and emits an output m_{out}, expressed as $F(m_{in}) = m_{out}$. An input m_{in} consists of a web request and a set of session variable name/value pair $S(m_{in})$. To facilitate detection, we further decompose a web request into two components: a web request key $r(m_{in})$, which includes the HTTP request method and the target file, and a set of input parameter name/value pair $P(m_{in})$. In this chapter, we only consider GET and POST methods and focus on PHP pages. For example, the web request keys include *GET-login.php*, *POST-login.php*, in the application shown in Fig. III.1. Similarly, an output consists of a web response and a set of session variable name/value pair $S(m_{out})$. A web response is a synthesized web page, which is usually generated by filling dynamic contents into a static web page structure (i.e., template). To deal with the infinite number of possible web responses, we decompose a web page into a web template, the number of which is finite, with a set of dynamic contents, which become output parameters. If we assign a unique ID to each static template, a web response can be symbolized as a web template ID (i.e., web response key $v(m_{out})$) and a set of output parameter name/value pairs $Q(m_{in})$. In the next section, we illustrate how to symbolize a web page as a web template with a set of output parameters.

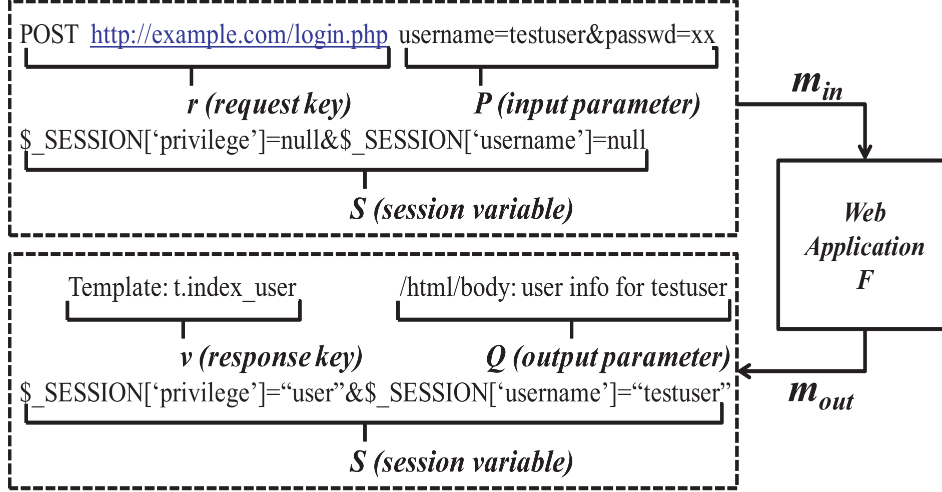


Figure III.2: A Stateless View of a Web Application

Approach

Approach Overview

Our approach for detecting state violation attacks has two key phases. In the training phase, the intended behavior model of the web application (i.e., the specification) is derived by observing the web request/response sequences and the corresponding session variable values during its attack-free execution. In the detection phase, the inferred model is used to evaluate each incoming web request and outgoing web response and detect any violations.

Due to the stateless nature of HTTP, session variables are explicitly defined in web applications to maintain the state of a web session. There are two ways for maintaining session states: 1) client-side only, where session states are directly carried in cookies, hidden forms, or URLs and 2) collaboration of the client and the server, where the server stores the session states and issues a session ID to the client for indexing its session states. In either case, session states can be retrieved at runtime for each web request independent of the web application implementation. For example, when session states are carried in cookies, hidden forms, or rewritten URLs, they can be directly retrieved from the web requests. When session states are kept in the server side, they can be found either in a file or a database table. In the case of PHP, the session state is

stored in temporary files located at */var/lib/php5* by default, which is indexed by a session ID within web requests, while in the case of JSP, the session state is persisted in database tables.

One straightforward approach to modeling the application behavior is to derive its states from the session variables and their values directly. Yet, this approach has several drawbacks: 1) at one application state, session variables may exhibit a large range of values. For example, *$_-SESSION['username']* can assume as many possible values as the number of registered users in the same application state. Thus, directly using session variable values to differentiate application states may result in a large number of spurious states; 2) definition of session variables may be missing from the application implementation. As a result, two application states in the specification can not be differentiated by the collection of all session variables. For example, in the application shown in Fig. III.1, there is no session variable defined for indicating whether the admin user identity has been logged.

Our approach follows the stateless property of HTTP and regards the session variables as part of the input to the web application along with web requests. Similarly, the output of the application consists of the web response and the session variables. In this way, the web application can be regarded as a stateless system, as shown in Fig. III.2. Under this stateless system model, we characterize the application behavior in the form of three types of likely invariants. 1) Type I input invariants: recall that the web application input consists of the web request and the values of the session variables when the request is made. This type of invariant models the relationship between the web requests and the session variable values. Essentially, it tries to capture the constraints on the web requests at certain session states. By identifying the invariant component of session variables, this approach avoids the introduction of spurious states by unnecessary session variables. 2) Type II input/output invariant: this type of invariants models the relationship between the web request and response as well as the changes in the session variables after the web request is processed. Essentially, it tries to capture the constraints on the application state transition and the input/output dependency at a certain state. Both type I and II invariant rely on the session variables to infer the application states. When the session variables are not sufficiently defined, we need a third type of invariant. 3) Type III input/output sequence invariants: this type of invariants models the relationship between consecutive web request/response pairs. Essentially, it tries to capture the application states that are not revealed by defined session variables by leveraging the historical request/response information. In the

following sections, we first formalize our system model and then illustrate how to extract three types of invariants and apply them into runtime detection.

Web Page Symbolization

To symbolize a web page, we first extract the web templates (O) from all the observed web pages (D). Then, given a web page $d \in D$, we classify it into the most probable template (v) and extract the set of output parameters (Q) accordingly. Techniques for extraction of templates from web pages have been presented in existing literatures [55, 38]. In this chapter, we leverage the method from TEXT [38], which expresses the DOM tree structure of a web page as a set of essential paths. Our template extraction procedure contains the following four steps. Step 1 and 2 are similar to TEXT and step 3 and 4 are designed to fulfill the purpose of template extraction in our context.

(1) Transformation: The DOM tree structure of a web page d is first transformed into a set of paths P_d. Here, we focus on the paths that lead to the leaf text nodes, which carry the information sent back to the clients within web pages. An index page from our example application can be expressed as three paths: "*/html/body/Welcome to the application*", "*/html/body/user information for: testuser.*" and "*/html/body/a/logout*", as shown in Fig. III.3.

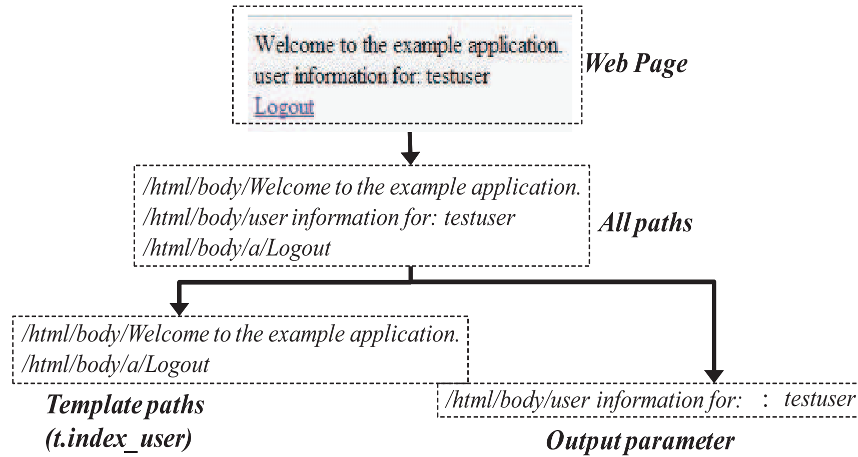


Figure III.3: The Procedure of Web Page Symbolization

(2) Pruning: To extract templates from all the paths, those paths that lead to dynamic contents should be pruned. To do so, we define the support of a path as the number of pages in

D that contain the path. Since the occurrence of a path that belongs to a template is generally higher, paths with low support are most likely dynamic content and should be pruned. For each page d, the minimum support threshold t_d is defined as the mode (i.e., the most frequent value) of the occurrence of paths that are contained in the page. Note that using one threshold for all the pages is inappropriate as each template may generate a different number of pages. After the paths with support lower than the threshold are pruned, each page is expressed as a set of "essential" paths. We use $ep(d)$ to denote the number of essential paths contained in the page d.

(3) Clustering: Two web pages are probably generated from the same web template if they have similar set of essential paths. The similarity ($Dist$) between two pages d_i d_j is defined as follows:

$$Dist(d_i, d_j) = \frac{cp(d_i, d_j)}{\sqrt{ep(d_i) \times ep(d_j)}} \tag{III.1}$$

where $cp(d_i, d_j)$ is the number of common essential paths contained in d_i and d_j and $ep(d)$ is the number of essential paths contained in d. We then perform hierarchical agglomerative clustering (based on average linkage) over all pages based on the above similarity metric. Each resulting cluster corresponds to a web template. The essential path set of a new template is the intersection of path sets from the two templates that are merged together.

(4) Parametrization: For each page in D, after eliminating the essential paths contained in the template it belongs to, the remaining paths in its path set belong to output parameters. The parameter is identified by the path leading to the text node and its value is the content of the text node. We extract the parameters that are observed in all pages that belong to the template as the set of output parameters of the template. For each parameter, we put the common parts (i.e., tokens) from all the observed values into the parameter name and only extract the variable part as its value.

For the example application, we obtain seven templates. They are the login page (*t.login_-form*), logout page (*t.logout*), wrong login page (*t.wrong_login*), regular user information page (*t.index_user*), admin user information page (*t.index_admin*), logging identity page (*t.admin*) and the title change page (*t.title_form*). As shown in Fig.III.3, the template *t.index_user* has a parameter "*/html/body/user information for:*", which displays the user's information and its value is the current user name.

Given a web page d, it is first transformed into a set of paths. Then, it is classified into the template v that has the highest similarity with its path set (i.e., $v = argmax(Dist(d, v_i))$, $v_i \in O$). The corresponding output parameters for the template are finally extracted.

Invariant Extraction

We extract three types of invariants: (1) type I input invariants, indexed by the web request key r; (2) type II input/output invariants, indexed by the key pair (r, v); (3) type III input/output sequence invariants, also indexed by the request key r. We also show some example invariants extracted from the application in Fig.III.1.

Type I Invariant

The inputs with the same request key r are grouped together. We extract the following types of invariants for each request key r.

(1) A set of session variables $S_{inv}(r)$ that are always present. An example invariant of this type is S_{inv}(*GET-index.php*) = {*$_SESSION['username'], $_SESSION['privilege']*}.

(2) A set of input parameters $P_{inv}(r)$ that are always present. An example invariant of this type is P_{inv}(*POST-login.php*) = {*email*, *passwd*}.

(3) For a specific session variable $s \in S_{inv}(r)$, its value is drawn from an enumeration set $V(s, r)$. For example, invariants of this type include: V(*$_SESSION['privilege'], GET-admin.php*) = {*admin*}, V(*$_SESSION['privilege'], GET-index.php*) = {*admin*, *user*};

(4) For a specific input parameter $p \in P_{inv}(r)$, its value is drawn from an enumeration set $V(p, r)$.

(5) The value of an input parameter $p \in P_{inv}(r)$ is always equal to the value of a session variable $s \in S_{inv}(r)$. For the request key *GET-index.php*, the session variable *$_SESSION['username']* is always equal to the input parameter *username*.

Type II Invariant

The input/output pairs with the same key pair (r, v) are grouped together. We first extract the same set of invariants as type I for the key pair. For example, an invariant drawn for the key pair (*GET-login.php*, *t.logout*) is that V(*logout*, (*GET-login.php*, *t.logout*) = {1} and the input parameter *logout* is added into P_{inv}(*GET-login.php*, *t.logout*).

We also extract two new invariants for each key pair (r, v):

(1) The value of an output parameter is always equal to the value of an input parameter and/or a session variable. This invariant reflects the dataflow within the web application. An invariant for the key pair (*POST-login.php*, *t.index_user*) is that the output parameter */html/body/* of the template *t.index_user* is always equal to the session variable *$_SESSION['username']* and the input parameter *username*.

(2) The session state is unchanged. For example, the user's session state always stays the same by observing the key pair (*GET-login.php*, *t.login_form*), but evolves for the key pair (*POST-login.php*, *t.index_user*).

Type III Invariant

For each request key r, we extract the following invariant:

(1) A set of input/output key pairs that always precede the web request key in one session. An invariant of this type is the key pair (*GET-admin.php*, *t.admin*) always precedes the request key *GET-admin2.php* and the key pair (*GET-admin2.php*, *t.title_form*) always occurs before *POST-admin2.php*.

Runtime Detection

Each web request key r is associated with a set of invariants, including both type I and type III invariants. Each input/output key pair (r, v) is also associated with a set of type II invariants. For detection, each invariant is transformed into an evaluation function, which operates on an input or an input/output pair. If the input or input/output pair satisfies the invariant, the function returns true. Otherwise, the function returns false. The runtime detection is performed in two phases:

(1) validating the input m_{in}: The web request is accepted, if and only if the request key has been observed and all the invariants associated with it are satisfied. Otherwise, the web request is dropped.

(2) validating the input/output pair (m_{in}, m_{out}): The web page is sent back to the user if and only if the corresponding key pair has been observed and all the invariants associated with it are satisfied. Otherwise, the web page is blocked.

All the attacks that exploit the example application can be detected by our extracted invariants. (1) Each auth bypass attack instance violates the invariants associated with three request keys *GET-admin.php*, *GET-admin2.php* and *POST-admin2.php* respectively and are detected at the first phase. For example, the first attack instance violates the invariant V(*$_SESSION['privilege']*, *GET-admin.php*) = {*admin*}. (2) the parameter manipulation attack violates the invariant associated with the request key *GET-index.php* where the input parameter *username* is always equal to the session variable *$_SESSION['username']* and is detected in the first phase. It also violates the invariant of the key pair (*GET-index.php*, *t.index_user*) that the output parameter "*/html/body/user information for:*" is equal to both of the input parameters *username* and *$_SESSION['username']*. (3) the workflow bypass attack violates the invariant associated with the request key *GET-admin2.php* that the key pair (*GET-admin.php*, *t.admin*) always precedes the request key and is detected in the first phase.

Implementation

We implement the prototype of our detection system BLOCK as a proxy that sits between the web application and the client, as shown in Fig. III.4. BLOCK is capable of intercepting all the messages exchanged between the web application and the client and taking snapshots of the user's session information stored at the server side. To capture the web requests and responses, we build BLOCK on top of WebScarab [50], an open source web application testing tool, which is deployed at the web server and configured as a reverse proxy. PHP web applications, which are our focus in this chapter, by default store the users' session information in temporary files at the directory */var/lib/php5*. BLOCK is able to locate the correct session files, indexed by the session ID within the web request, and read the user's session information. BLOCK can be operated in two modes: training and detection. In the training mode, BLOCK collects the observed web requests, responses and their associated session information, analyzes those execution traces and extracts the set of relevant invariants. In the detection mode, BLOCK monitors the interactions between the clients and the web application, dynamically detects and blocks those potential attacks that violate the extracted invariants. We note that BLOCK can be easily extended to other platforms other than PHP by just modifying the component that accesses the session information to handle a variety of programming frameworks. For example, in the case of Tomcat

servlet, the component should be able to access database tables via JDBC drivers, which store persistent session information. Our implementation is independent of the web application (i.e., doesn't require the source code for analysis or instrumentation). Thus, it can scale up to protect a large number of web applications.

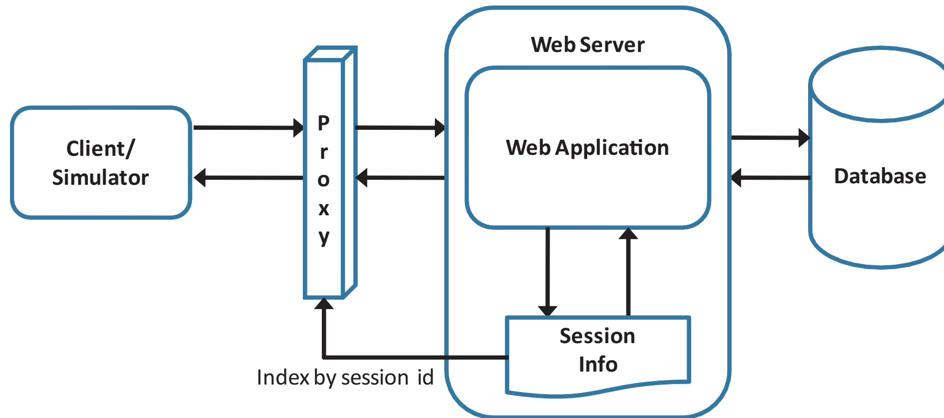


Figure III.4: Overview of BLOCK System

Training Mode

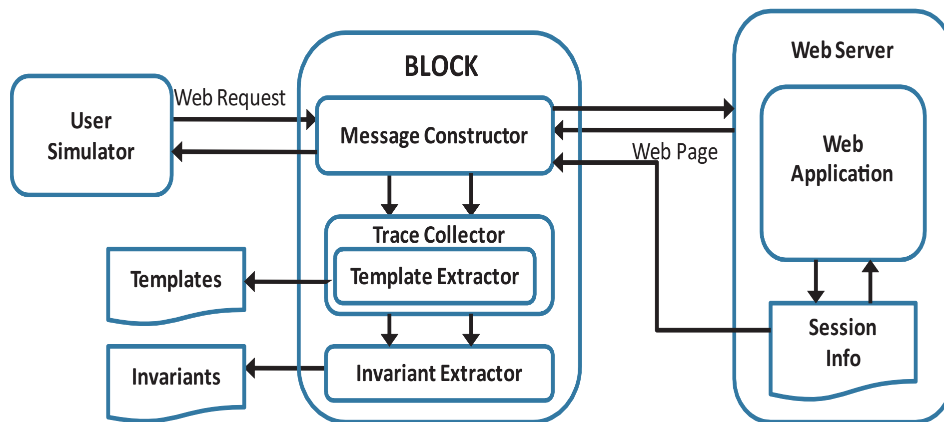


Figure III.5: Training Mode (BLOCK)

The components of BLOCK in the training mode are shown in Fig. III.5. Whenever a web request or a web page is captured, the message constructor takes a snapshot of the current session state and composes the corresponding messages, which is sent to the trace collector. After

sufficient traces have been collected, BLOCK will perform offline learning. The trace processor first extracts web templates from observed web pages, then parses both the input and output messages into the designated format: a request or response key associated with a set of key/value pairs for both parameters and session variables. The parsed traces are fed into the invariant extractor, where all three types of invariants are derived. The value-related invariants (e.g., the equality relationship between variables, the enumeration value set of variables) are inferred by leveraging the Daikon engine [27], a well-known tool for dynamic inference of program invariants. The traces are transformed into the format required by the Daikon engine and the output is a set of invariants extracted for each declared entry. Presence-related invariants are extracted by self-developed programs. All extracted invariants comprise the web application's specification.

Detection Mode

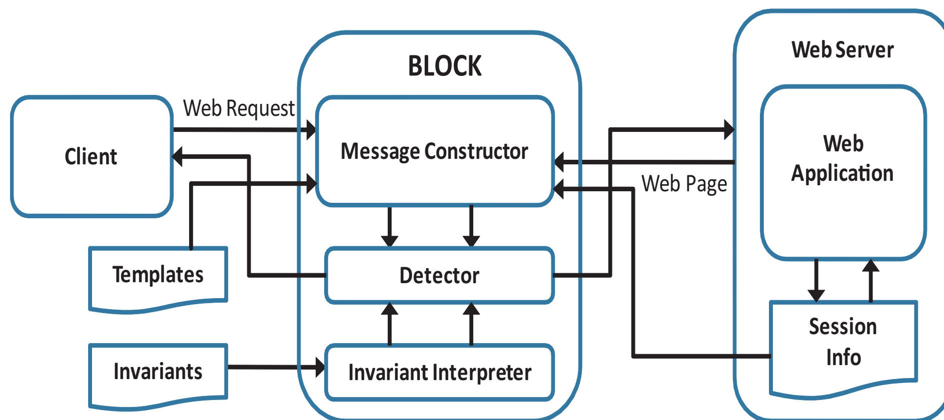


Figure III.6: Detection Mode (BLOCK)

Once the invariants are extracted, BLOCK switches to the detection mode, as shown in Fig. III.6. The invariant interpreter loads and interprets the extracted invariants. At runtime, the message constructor combines session information with the intercepted web request, composes an input and sends it to the detector for evaluation. If the input is accepted, the web request is forwarded to the web application and logged as the current input for the web application. Otherwise, the web request is dropped. When the message constructor receives a web response, if the response is a redirection, the subsequent web request will not be evaluated or logged. If the response is a web page, the message constructor assigns the web page a response key based

on its web template, composes an output and sends it to the detector, where the output is paired with the current input and evaluated. If the output is accepted, the web page is returned to the client and the key pair is logged for the current user session. Otherwise, the web response is blocked and the current input is invalidated. After the user's session has terminated, all of the logged key pairs are cleaned up from the memory.

Evaluation

Experiment Setup

We evaluate our approach using a set of open source PHP web applications, which are representative of different types of functionalities. Table III.1 shows a summary of web applications we use for evaluation. (1) Scarf is a conference management system, which is used for managing sessions, papers, users and comments. It is known to contain an auth bypass vulnerability (CVE-2006-5909). The attacker can directly visit the administrative page *generaloptions.php* and modify the system settings and user accounts, since the admin page does not check the privilege of the current user. It echoes the first case of auth bypass in the example application. (2) Simplecms is a simple content management system that allows the admin to publish and manage contents. It is also vulnerable to an auth bypass attack in *Auth.php* page (BID 19386) because it uses the *register_globals* mechanism insecurely. An attacker can append a parameter *loggedin* to the web request and bypass the authentication check. It echoes the second case of auth bypass in the example application. (3) Bloggit is a blog application that supports web blog management. It also has an auth bypass vulnerability (CVE-2006-7014) in the *admin.php* page where the restrictive code continues being executed after the auth check fails. It echoes the third case of auth bypass in the example application. (4) Wackopicko [68] is an online photo sharing website that allows users to upload pictures, comment on and purchase other people's pictures. It was initially written for testing web application vulnerability scanners. It was designed with a number of vulnerabilities, such as cross-site scripting, SQL injection and file inclusion. Here, we focus on its parameter manipulation vulnerability. After a user logs in, they can view their personal information in *home.php* page. However, an attacker can manipulate the *userid* parameter

to view any other user's information and owned pictures. (5) OsCommerce [48] is a widely-used open source e-commerce application. To evaluate our approach for handling workflow bypass attacks, we instrument one vulnerability into the checkout procedure, which allows the attacker to go directly to the payment page without selecting the shipping method and the total charge does not include shipping fees.

Table III.1: Summary of Evaluated Web Applications (Evaluation of BLOCK)

Application	PHP files	Description	Vulnerability
Scarf	21	Conference management system	Auth bypass (CVE-2006-5909)
Simplecms	23	Content management system	Auth bypass (BID 19386)
BloggIt	24	Blog engine	Auth bypass (CVE-2006-7104)
Wackopicko	53	Photo sharing website	Parameter manipulation
OsCommerce	533	Open source e-commerce solution	Workflow bypass (instrumented)

All the web applications and BLOCK (based on WebScarab) were deployed on a 2.13GHz Core 2 Linux server with 2GB RAM, running Ubuntu 10.10, Apache web server (version 2.2.16) and PHP (version 5.3.3). To collect training traces, each web application is driven by a user simulator, which emulates the interactions between a normal user and the web application. For each web application, user roles and atomic operations are first identified manually. Then, the user simulator is developed based on the Selenium WebDriver [59] to emulate a normal user operating a web application. The simulator leverages a library of user information for all undergraduate students from a network security class and is able to automatically explore the web application, such as clicking the links, filling in and submitting forms. Among the available atomic operations for the currently chosen user, it randomly selects one as the emulated user's next step. The user simulator is set up on a 2.83GHz Core 2 desktop with 8GB RAM running Windows 7 and Firefox 4.0.

Detection Effectiveness

BLOCK first runs in the training mode to collect the execution traces, generated by the user simulators. Table III.2 shows the summary of our collected traces [1]. Then, it analyzes those traces, extracts web request keys, web page templates, as well as all three types of invariants. To observe the impact of the training set size on the number of derived invariants, we vary the training set size and calculate the resulting invariants. Fig. III.7 shows the experiment result we obtain for the Scarf application. We can see that the numbers of type I and III invariants initially decrease and then converge with the increase of training set size, indicating the elimination of false invariants learnt from insufficient training samples. The number of type II invariants first increases, due to the exploration of new state space that was not revealed by the small training set, then it slowly converges. Based on this observation, we use the training set for each application where the number of invariants converges.

Table III.2: Summary of Training Set (Evaluation of BLOCK)

Application	Requests	Web Pages	Request Keys	Web Templates	Key Pairs	Type I Inv	Type II Inv	Type III Inv
Scarf	3225	3200	21	26	69	90	640	11
Simplecms	2661	2555	17	12	34	56	190	28
BloggIt	2657	2645	16	13	47	65	377	9
Wackopicko	2949	2946	20	12	30	36	155	37
OsCommerce	3879	3444	25	36	123	374	4609	26

BLOCK then switches to the detection mode. A clean test set is generated by both the user simulators and the undergraduate students who manually operate the web applications. Ten attack instances are manually generated under different circumstances for each web application. Table III.3 (Req: web request; Resp: web response; TS: clean test set; FP: false positive) shows the summary of the test set and all the detection results. All of the attacks are successfully detected by BLOCK with only a few false positives incurred. This fact demonstrates the effectiveness of our approach at detecting state violation attacks.

Upon further investigation of the false positives we find two major sources. One is the incomplete exploration of the web application performed by the user simulator. The capability

[1] Here, we note that our training only covers the portion of mostly used functions for customers in the OsCommerce application. Also, we do not count redirection headers as web pages.

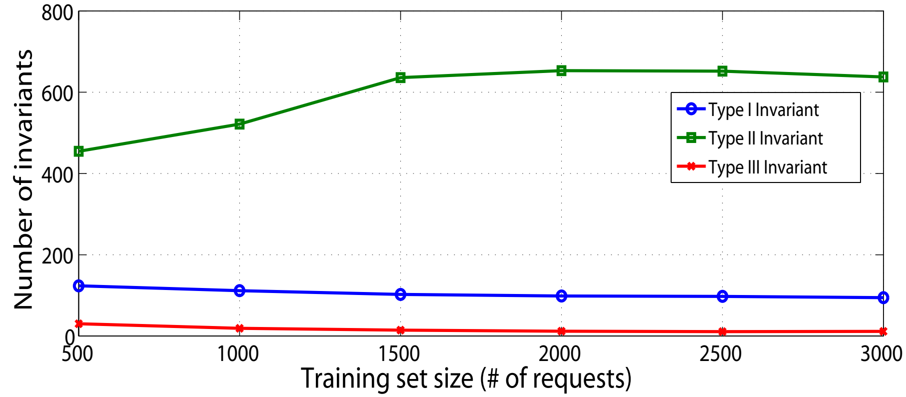


Figure III.7: Number of Invariants vs. Training Set Size (Scarf)

Table III.3: Summary of Detection Result (Evaluation of BLOCK)

Application	Req (TS)	Resp (TS)	Req (FP)	Resp (FP)	Attacks	Detected	Invariant violations
Scarf	1364	1360	0	6	10	10	type I and III
Simplecms	1731	1688	0	8	10	10	type I and III
BloggIt	1044	1024	0	0	10	10	type I and III
Wackopicko	1322	1314	0	1	10	10	type I and II
OsCommerce	1505	1460	3	10	10	10	type I and III

of the user simulator determines the state space that our detection system can characterize for the web application. The more the simulator explores, the richer and more accurate these invariants are. In our evaluation, some false positives result from error pages that are not explored by the simulator and thus not observed and profiled by the invariant extractor. In practice, if real-world traces are available, our detection system can be readily applied and work effectively. The other source of false positives is the inaccurate symbolization of web pages. Page symbolization affects both the training and detection phase. In the training phase, both the number and the quality of the inferred invariants, especially for type II, are closely related with the number of extracted web templates. We can see that the number of type I and III invariants converges very fast, thus leading to an extremely low number of false positives for web requests, while type II invariants bring more false positives of web responses. In the detection phase, due to the content changes of web pages, it is possible that a web page is classified as a template incorrectly, which likely results in an unobserved pair of input/output and thus a false positive. We use the same clustering threshold for all applications to extract web templates, which also introduces certain level of inaccuracies. Since web template extraction is not our focus in this chapter, we adapt the methods from TEXT [38] which appear to work well with the web application in our evaluation. To increase the accuracy and robustness of web page symbolization, advanced algorithms or manual audit can be introduced for guiding the process.

The detection results also show the types of invariants violated by different attacks. Auth bypass attacks on insufficient checking of session variables result in violations of type I invariants that are imposed on the session state, when web requests are received. They would also violate type III invariants due to the missing step of authentication/authorization. Parameter manipulation attacks can be detected by type I invariants, if the input parameters are related to the session variables. They may also be identified by type II invariants, if the corresponding web pages contain output parameters that are related with the session state. Workflow bypass attacks will be blocked in the same manner as auth bypass attacks, if the session variables, which are used for guarding the state transitions, are not checked. If there are no such guarding session variables (e.g., in the example application) type III invariants would help to identify workflow bypass attacks through the constraints imposed on the sequence of operations.

Performance Overhead

Since our detection system sits between the client and the web application, it will affect the response time of the web application for several reasons. First, the WebScarab proxy intercepts and forwards all the messages exchanged between the user and the web application, which increases the response time. Second, the integrated detector evaluates the web requests and web pages, which introduces additional delay. To measure the performance overhead induced by our detection system, we use the simulators to perform a designated sequence of operations and log the response time for every web request. We compare the performance under three configurations: 1) without the WebScarab proxy, 2) with the WebScarab proxy deployed but the detector disabled, 3) with the WebScarab proxy deployed and the detector enabled. Figure III.8 shows the summary of the averaged response time for each application under the above three scenarios. We can see that the average response time increases by a factor of 1.5 to around 5, if BLOCK is deployed and enabled. While the resulting response time is still acceptable, we notice that more than 90% of the overhead is caused by the WebScarab proxy and only a small amount is introduced by the detector. For our current prototype implementation, no modifications or configurations are made to the WebScarab proxy to enhance its performance. If a more lightweight and efficient proxy (e.g, Apache mod_proxy) is employed to integrate our detection system, it would be possible to reduce the response time.

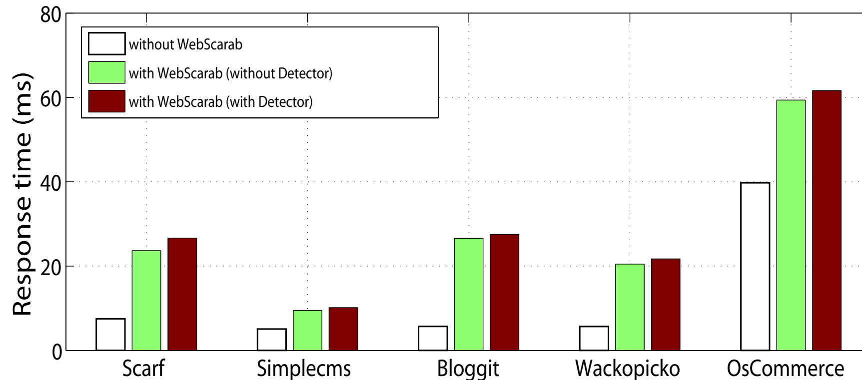


Figure III.8: Summary of Performance Overhead (Evaluation of BLOCK)

Discussion

There is one limitation of BLOCK we would like to point out. BLOCK only observes and models the relations between web requests, web responses and the session variables. Thus it cannot handle attacks that violate the persistent states that exist in database tables. If BLOCK is extended to capture and analyze the SQL queries/responses from a database, it has the potential to handle this type of state violation attack. We address this limitation in our follow-up work on SENTINEL (Chapter IV).

Our technique bears the same limitations as other dynamic analysis techniques. The completeness and correctness of inferred invariants cannot be guaranteed. In order to put BLOCK into practice, introducing some manual intervention is preferable to guarantee sufficient training and suppress false positives. In the future, we hope to investigate mechanisms for automatic verification of likely invariants.

CHAPTER IV

SECURING DATABASES FROM LOGIC FLAWS IN WEB APPLICATIONS

In this chapter, we present SENTINEL, which stands for SEcuriNg daTabase from logIc flaws iN wEb appLication. We model the web application as an extended finite state machine (EFSM) and derive the application specification in a black-box manner. In particular, we systematically extract persistent state information in the database and infer data constraints from observed SQL queries and responses. Suspicious SQL queries that violate corresponding invariants are identified as potential attacks. We implement a prototype detection system and evaluate it using a set of real-world web applications. The experiment results demonstrate the effectiveness of our approach and show that acceptable performance overhead is incurred by our implementation.

The rest of this chapter is organized as follows. We first give an overview of our approach. Then, we describe an example vulnerable application and the attacks we focus on. We present our EFSM model of a web application and approach in the following two sections. The implementation and evaluation results of the prototype system are then demonstrated. Finally, we discuss the limitations of our technique to conclude this chapter.

Overview

The front-end web application usually acts as the lone trusted user that interacts with the database. Thus, the database fully trusts the web application and accepts and executes all the queries submitted by the application. As such, the vulnerabilities within web applications may introduce security concerns for the information stored in the database and lead to information disclosure or tampering. One type of logic attacks trick the application into sending malicious SQL queries at inappropriate application states.

We proposed BLOCK in the previous chapter for inferring the application specification. BLOCK observes the web requests/responses between the web application and its users and extracts the invariants associated within. While BLOCK uses a black-box and source-code-free framework, its capability is limited because it only observes web requests/responses and does not take into account the large amount of information persisted in the database, which results in an incomplete specification. The persistent information in the database may affect the application's

behavior in two ways. First, the application can use persistent objects in the database for maintaining its state across web sessions, while using session variables for managing the state during a session. Second, the persistent objects may embed complex data constraints for web applications. Moreover, BLOCK examines web requests/responses and thus is incapable of handling certain state violation attacks that are targeted at the database.

To address the above limitations, we answer two questions in this chapter: (1) **What external behavior should we observe in order to collect sufficient information for specification inference?** Since we focus on securing the database, we observe the interaction between the web application and the database. For the application to utilize persistent objects stored in the database, they have to be returned within SQL responses first. Thus, we collect all the observed SQL queries and responses, as well as the corresponding session variables.

(2) **How should we infer the application logic from collected information in a systematic way, so that the application behavior can be characterized adequately?** We model the web application as an extended finite state machine (EFSM). EFSM has been employed for modeling the behavior of complex software [43], because it can capture state transitions as well as data constraints associated with transitions. It also fits well in the web application scenario. To derive the EFSM, we first construct SQL signatures from observed SQL queries, which represent the output symbols emitted from the EFSM. Then, we extract a set of invariants for each SQL signature from both session variables and SQL responses, which characterize the application state and the associated data constraints when a SQL query is issued. In particular, we leverage the Daikon engine [27] to derive value-based invariants, including the invariants over variables that are used for indicating the application state and the data constraints. Additionally, we extract the dependencies between SQL signatures to infer other data constraints, which are implicitly reflected from previously issued SQL queries. The set of invariants, indexed by SQL signatures, manifest the application specification and are used for evaluating the incoming SQL queries at runtime. Suspicious SQL queries, which violate any invariant associated with their respective signatures, are identified as potential attacks and blocked.

Attacks and Running Example

Fig. IV.1 shows an example application *SimpleOAK*, which is used to illustrate the attacks we address in this chapter and demonstrate our system model and approach throughout the chapter. A user is first presented with the *index.php* page (the application is at state s_0). After the user inputs correct login credentials, the application will redirect the student to the *user.php* page (the application is at state s_1) and the professor to the *admin.php* page (the application is at state s_2), depending on the role information (i.e., *$row['role']* in *index.php*) retrieved from the database. The *user.php* page shows the student's registrations and grades, as well as the syllabus links for registered courses. The student can only modify his/her own registrations (data constraint c) through the *course.php* page. The *admin.php* page shows all the students' registrations, so that the professor can modify their grades accordingly.

SimpleOAK contains several vulnerabilities. First, the application fails to enforce the correctness of the application state, which allows malicious SQL queries to be issued at an incorrect state. There are two cases of this type of vulnerability in the *SimpleOAK* application.

Case 1: A guest user at state s_0 can directly access a student's information (i.e., issuing *Query1* and *Query2* in *user.php*), because the *user.php* page does not check if the user has logged in (i.e., whether the session variable *$_SESSION['userid']* is null).

Case 2: A student at state s_1 can directly access the professor's page (i.e., issuing *Query3* in *admin.php*) and modify the grades (i.e., issuing *Query4* in *course.php*), because the *admin.php* page fails to verify the application state, which is stored as persistent objects in the database (i.e., whether *$row['role']* in *index.php* is equal to *professor*).

Second, the web application fails to enforce the data constraints (e.g., constraint c) associated with SQL queries, which allows the attacker to issue malicious SQL queries by manipulating query parameters.

Case 3: A student is able to view/change another student's registrations.

- View registration: the *user.php* page fails to check the constraint: *$_GET['userid'] == $_SESSION['userid']* associated with *Query1* and *Query2*.

- Register a course: the *course.php* page fails to check the constraint: *$_POST['userid'] == $_SESSION['userid']* associated with *Query5*.

```
<? php
include_once("header.php");
if (isset($_GET['logout'])){
      unset($_SESSION['userid']);
      session_destroy();
} else if (isset($_POST['username']) && isset($_POST['password'])){
      $Query0 = mysql_query(sprintf("SELECT * FROM users WHERE login = '%s' AND
password = '%s';", $_POST['username'], $_POST['password']));
      if ($row = mysql_fetch_assoc($Query0)){
            $_SESSION['userid'] = $row['id'];
            if ($row['role'] == "professor"){
                  header("Location: admin.php?userid=" . $_SESSION['userid']);
            } else if ($row['role'] == "student"){
                  header("Location: user.php?userid=" . $_SESSION['userid']);
            }
      } else { die("Wrong username or password.");}
} ?>
<form action="index.php" method=post>
username: <input name="username" type="text"><br>
password: <input name="password" type="password"><br>
<input name="submit" type="submit">
</form></body></html>
```

index.php

```
<?php
include_once("header.php");
if (isset($_SESSION['userid'])){
      $Query3 = mysql_query("SELECT * FROM registration;");
      print("<table><tr><td>Name</td><td>Course</td><td>Grade</td></tr>");
      while ($row = mysql_fetch_assoc($Query3)){
            print("<tr><td>" . getUserName($row['user_id']) .
                  "</td><td>" . getCourseName($row['course_id']) .
                  "</td><td><form method=\"post\" action=\"course.php\">" .
                  "<input type=\"hidden\" name=\"register_id\" value=" . $row['id'] . ">".s
                  "<textarea name=\"grade\">" . $row['grade'] . "</textarea>" .
                  "<input type=\"submit\" name=\"action\" value=\"Modify\"></form></td><tr>");
      }
      print("</table><br><a href=\"./index.php?logout=1\"><b>Logout</b></a>");
} else{  die("You are not authorized to view this page"); }
?> </body></html>
```

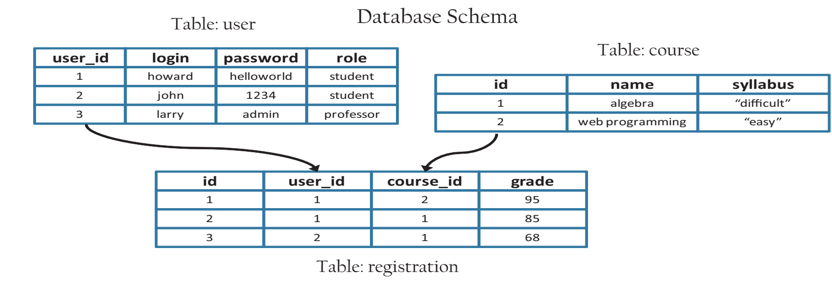
admin.php

```
<? php
include_once("header.php");
if (isset($_GET['userid'])){
      print("<table><tr><td>Course</td><td>Grade</td><td>Syllabus</td><td>Unregister</td></tr>");
      $Query1 = mysql_query("SELECT * FROM registration WHERE  user_id =".
$_GET['userid']. ";");
      while ($row = mysql_fetch_assoc($Query1)){
            print("<tr><td>". getCourseName($row['course_id']). "</td><td>" . $row['grade'].
                  "</td><td><a href=\"./course.php?course_id=" . $row['course_id']. "\">link</a>" .
                  "</td><td><form method=\"post\" action=\"course.php\">".
                  "<input type=\"hidden\" name=\"register_id\" value=" . $row['id'] . ">" .
                  "<input type=\"submit\" name=\"action\" value=\"Unregister\"></form></td></tr>");
      }
      print("<table><tr><td>Course</td><td>Register</td></tr>");
      $Query2 = mysql_query("SELECT * FROM course WHERE course.id NOT IN
(SELECT course_id FROM registration WHERE user_id  =" . $_GET['userid']. ")");
      while ($row = mysql_fetch_assoc($Query2)){
            print ("<tr><td>". $row['name']. "</td><td><form method=\"post\" action=\"course.php\">".
                  "<input type=\"hidden\" name=\"course_id\" value=" . $row['id']. ">" .
                  "<input type =\"hidden\" name=\"user_id\" value=" . $_GET['userid'] . ">" .
                  "<input type=\"submit\" name=\"action\" value=\"Register\"></form></td></tr>");
      }
      print("</table><br><a href=\"./index.php?logout=1\"><b>Logout</b></a>");
} ?>
</body></html>
```

user.php

```
<? php
include_once("header.php");
if (isset($_POST['register_id']) && isset($_POST['grade']) && $_POST['action'] == "Modify") {
      $Query4 = mysql_query("UPDATE registration SET grade=" . $_POST['grade'] .
"WHERE id= " . $_POST['register_id'] . ";");
      if ($Query4) {
            header("Location: admin.php?userid=" . $_SESSION['userid']);
      } else {  die("Fail to update the grade."); }
} else if (isset($_POST['course_id']) && $_POST['action'] == "Register") {
      $Query5 = mysql_query("INSERT INTO registration (user_id, course_id) VALUES
(".$_POST['user_id'] . ", " . $_POST['course_id']. ");");
      if ($Query5) {
            header("Location: user.php?userid=" . $_SESSION['userid']);
      } else {  die("Fail to register the course."); }
} else if (isset($_POST['register_id']) && $_POST['action'] == "Unregister") {
      $Query6 = mysql_query("DELETE FROM registration WHERE id=" .
$_POST['register_id']. ";");
      if ($Query6) {
            header("Location: user.php?userid=" . $_SESSION['userid']);
      } else {  die("Fail to unregister the course."); }
} else if (isset($_GET['course_id'])){
      $Query7 = mysql_query("SELECT * FROM course WHERE id =" . $_GET[' course_id'].
";");
      if (res = mysql_fetch_assoc($Query7)){
            print("<h2>" . $res['name']. "</h2><br>" . "<b>Syllabus: </b>" . $res['syllabus']. "<br>");
      }
} ?>
</body></html>
```

course.php



Database Schema

Table: user

user_id	login	password	role
1	howard	helloworld	student
2	john	1234	student
3	larry	admin	professor

Table: course

id	name	syllabus
1	algebra	"difficult"
2	web programming	"easy"

id	user_id	course_id	grade
1	1	2	95
2	1	1	85
3	2	1	68

Table: registration

Figure IV.1: An Example Vulnerable Web Application: *SimpleOAK*

- Unregister a course: the *course.php* page fails to check if the *user_id* field in the affected row of *registration* table is equal to the session variable *$_SESSION['userid']*, when issuing *Query6*.

Case 4: A student can view the syllabus of a course for which he/she has not registered by manipulating the *course_id* parameter in *Query7*. The *course.php* page fails to check the correlation between the student and the course, which exists within *registration* table (i.e., the directed lines in Fig. IV.1).

To date, there is no automated mechanism that can help developers identify all the above security flaws and detect the corresponding attacks. Developers have to pay extra attention and manually place appropriate checks during development and code auditing.

System Model

We model a web application as an *extended finite state machine (EFSM)* [28], denoted as M. A web application M, as shown in Fig.IV.2, is defined as a seven-tuple: $M = (S, V, I, O, P, U, T)$.

- S denotes the set of application states. A web application maintains its state using either session variables (e.g., *$_SESSION['userid']*) or persistent objects (e.g., the *row* column in *user* table) in the database. We refer to the set of session variables and state-related persistent objects as state variables, which collectively characterize the current application state.

- V denotes the set of context variables, which include global variables (e.g., *$_SERVER* variables) and local variables in scope (e.g., *$row['id']* in *user.php*), except state variables. They represent the general context of the application's current execution.

- I is the set of input symbols, which include the users' web requests (e.g., *GET: user.php?userid=3*) and the SQL responses (e.g., *$Query0*, which contains the response of *Query0*) returned by the database.

- O is the set of output symbols, which include web responses sent to users (e.g., *header("Location: user.php")*) and SQL queries issued to the database (e.g., *Query0*).

- P: $D_S \times D_V \rightarrow \{true, false\}$ is the set of data constraints associated with state transitions (e.g., *if($row[‘role’]==professor)* in *index.php*). D_S and D_V denote the evaluation domains for state variables and context variables respectively.

- U: $D_S \times D_V \rightarrow D_S \times D_V$ is the set of update functions, which update state and context variables (e.g., *$_SESSION[‘uerid’]=$row[‘id’]* in *index.php*).

- T: $S \times I \times P \rightarrow S \times O \times U$ defines the state transitions. In a web application, each state transition can be decomposed into two steps: (1) the application accepts the input, executes update functions and possibly transitions to a new state, just before the output symbol is emitted, which can be expressed as T^U: $S \times I \rightarrow S \times U$. Whether the application transitions to a new state depends on if state variables are updated. (2) the application evaluates the data constraints over current variables and issues the output symbol if the evaluation returns true, which can be expressed as T^O: $S \times P \rightarrow O$. Note here the application state does not change in this step. SQL queries may modify state-related persistent objects in the database. However, the application is aware of the state change only after it retrieves the modified persistent objects later. Thus, we regard the application state before and after the output symbol is emitted as the same.

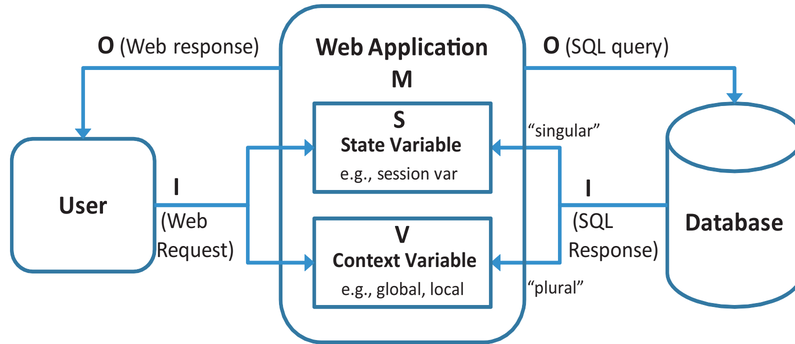


Figure IV.2: An EFSM Model of a Web Application

Fig.IV.3 shows part of the intended EFSM model for *SimpleOAK*, which contains two states and two transitions. First, *SimpleOAK* accepts a POST web request (i_0) at state s_0, which triggers the transition t_0. In the first step t_0^U, the application updates context variables (u_0) and stays at the same state s_0, because no state variables are updated. In the second step t_0^O, the application checks data constraints (p_0) and issues an SQL query to the database (o_0). After the

application receives the SQL response from the database (i_1), a new transition t_1 is triggered. It first executes updating functions (u_1) and transitions to a new state s_1, because the state variable *$_SESSION['userid']* is updated to the current user id (i.e., from null to non-null). Then, the application evaluates data constraints (p_1) and returns a web response to the user (o_1).

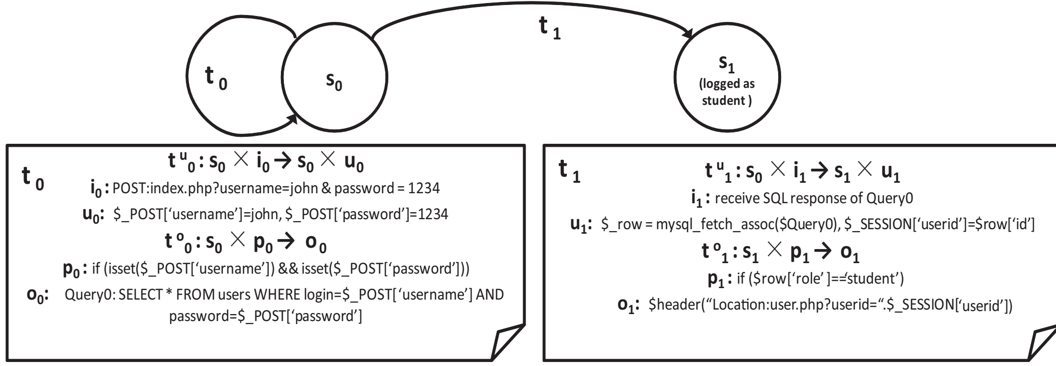


Figure IV.3: Partial EFSM Model of *SimpleOAK*

Approach

Approach Overview

Our objective is to detect malicious SQL queries that exploit logic flaws and violate the application specification. From our system model, such violations manifest during the second step of state transition (i.e., $T^O\colon\ S \times P \rightarrow O$), when the output symbols (i.e., SQL queries) are emitted. There are two scenarios: (1) the output is emitted when the application is at an incorrect state, which can be captured by characterizing the relationship between the output and the application state (i.e., $S \rightarrow O$); (2) the output is emitted when the data constraints are not satisfied, which can be captured by characterizing the relationship between the output and data constraints (i.e., $P \rightarrow O$). Our approach infers the application specification by observing and analyzing the interactions between the application and the database during attack-free sessions. Then, the inferred specification is used for runtime detection of malicious SQL queries issued by the web application. In the following sections, we first illustrate how we identify the set of output symbols (i.e., O) by constructing SQL signatures from observed SQL queries. Then, we present how we

infer the application specification from collected information, including application state inference and data constraint inference. The inference also captures the first step of state transition (i.e., $T^U\colon\, S\times I \rightarrow S\times U$), because we utilize updated state variables and context variables triggered by the application input (i.e., either a web request or an SQL response). The inferred specification is realized as a set of invariants, indexed by SQL signatures. Finally, we describe how we evaluate incoming SQL queries and detect potential attacks at runtime.

SQL Signature Construction

Each SQL query is composed of a skeleton structure, which is programmed in the source code, and a set of query parameters, whose values are dynamically fed by the application at runtime. To represent the output with a finite set of symbols, we need to separate the skeleton structure from query parameters, which have an unbounded set of values. We extract the skeleton structure of SQL queries in three steps, as shown in Fig.IV.4. First, we identify all the literals in the SQL queries and collect all the observed values for each parameter that assumes a literal. Second, for each parameter, we perform a one-sample Komoglov-Smirnov's D statistic test (KS-test) to determine whether the value domain of the parameter is bounded or not. The KS-test is employed to evaluate whether the number of unique values of the parameter linearly increases with the number of sample sizes. Further details can be found in [42]. Third, for each parameter, if the parameter has a bounded value domain (e.g., *role*), we assume its value carries implicit meaning for operations, which should be retained within the structure of the query. If the parameter has an unbounded number of values observed (e.g., *user_id*), we replace its value with a place holder token and record the parameter. The resulting string is the skeleton structure of the query (e.g., *SELECT * FROM registration WHERE user_id = Token*, where *Token* represents a query parameter).

Next, we construct a SQL signature by combining the skeleton structure of the SQL query and the script name, where the query resides. An example signature is {*user.php, SELECT * FROM registration WHERE user_id = Token*}. Each SQL signature represents a unique output symbol that can be issued by the web application.

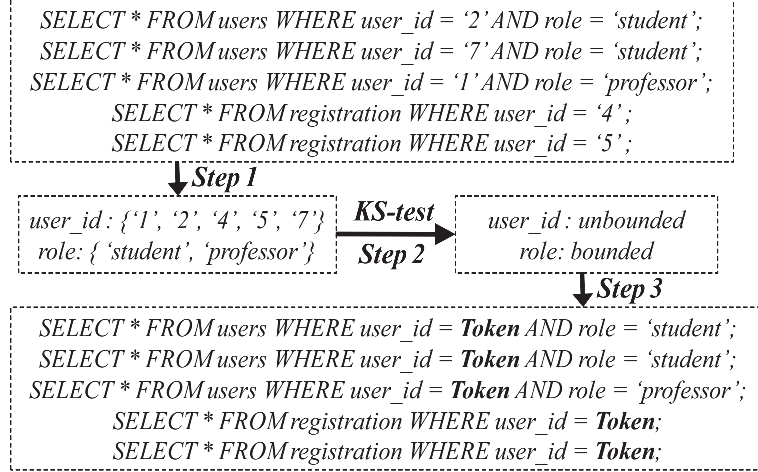


Figure IV.4: An SQL Query Skeleton Extraction

Application State Inference

To infer the relationship between the output and the application state (i.e., $S \rightarrow O$), we associate each SQL signature with the set of state variables when an SQL query is issued. First, we have to identify the correct set of state variables to construct the state space S. State variables come from two sources: session variables and persistent objects in the database. Due to the adoption of the connection pooling technique [18], which shares SQL connections among different sessions, SQL queries from different sessions cannot be differentiated. We reuse the session id and associate it with each SQL query for indexing the set of session variables.

The state variables that come from persistent objects can only be used by the application after they are retrieved from the database, thus can be observed from SQL responses. However, the SQL response may contain a large amount of objects, only part of which are actually used to maintain the application state. As such, we identify three types of SQL responses: (1) Type I: the response is a boolean (i.e., true or false), which applies to UPDATE, INSERT and DELETE queries; (2) Type II: the response of SELECT query, which always contains no more than one row of objects [1]. We say this type of response is "singular", which can possibly be used for retrieving persistent state information from the database and updating state variables (as shown in Fig.2). For example, *Query0* in *index.php* page retrieves the current user information persisted in the database, which determines the current application state (i.e., at either s_0, s_1 or s_2). (3) Type

[1]If a SELECT query returns a false on error, we consider its response to be zero row of object.

III: the response of SELECT query, which contains a variable row of objects (usually more than one). We say this type of response is "plural", which is used for updating context variables by the application (and inferring data constraints in Section 4.4). For example, *Query1* in the *user.php* page returns a number of courses the user has registered. We infer the response type for each SQL signature by examining all the responses, returned by SQL queries with the same signature and associate a flag r with each signature to indicate its response type. In particular, we transform the "singular" SQL response into a set of key-value pairs similar to the representation of session variables (e.g., *$row['id']*, *$row['role']* in *index.php*).

After collecting all the state variables, we employ a Daikon engine to extract value-based invariants over state variables, which characterize the application state associated with each signature.

Data Constraint Inference

To infer the relationship between the output and the data constraints (i.e., $P \rightarrow O$), we extract three types of data constraints for each SQL signature. First, each SQL signature itself captures part of the data constraints, which are directly encoded within WHERE clauses and automatically enforced (e.g., *WHERE course.id NOT IN* in *Query2*). Second, we employ daikon engine to infer the mathematical relationship between state variables and context variables (i.e., $\mid S \mid \times \mid V \mid$), which are fed into SQL queries as query parameters [2]. For example, the variable *$_GET['userid']* is always equal to the session variable *$_SESSION['userid']* in *Query1.* Third, data constraints may also be embedded within previously issued SQL queries, since the application implicitly assumes the dependency relationship between SQL queries. We say an SQL query q is dependent on another query q', if the database objects q performs over also satisfy the constraints specified by the WHERE clause of q'. For example, *SimpleOAK* first retrieves all the registered courses (at least the course id) for one student via *Query1.* Then, the student can view the syllabus of those courses via *Query7.* The application implicitly assumes *Query7* is dependent on *Query1*, which means that the specified course in *Query7* always satisfies the constraint within *Query1* that the course is registered by current student. Since the objects within the SQL response must satisfy the constraints specified by the WHERE clause, we infer this type

[2] We only identify the equality relationship between state variables and context variables here.

of constraint for each signature by evaluating its WHERE clause over previously observed SQL responses, instead of analyzing complex WHERE clauses and their logical relationship among different SQL queries. If the WHERE clause of one signature *sig* is always satisfied by the response of another signature sig' that is always issued earlier and observed during the session, then we express such a constraint in the form that *sig* is dependent on sig'.

Invariant Extraction

We extract a set of invariants from collected traces to represent the application specification and use them for runtime detection. In the traces, each SQL query t is associated with the following information: a set of query parameters Q, a set of state variables S and its SQL response *Resp*. The SQL queries, as well as relevant information, are grouped by signatures. We extract the following types of invariants for each SQL signature.

Type A: Application state invariant

Type A invariants characterize the application state when SQL queries are issued (i.e., $S \rightarrow O$). For each SQL signature *sig*, we identify:

A.1: A set of state variables $S_{inv}(sig)$ ($S_{inv}(sig) \subseteq S$) that are always present (i.e., their value is non-null). These state variables represent the dimension of the application state. An example in *SimpleOAK* is $S_{inv}(sig1)$ = {*$_SESSION['userid']*, *$row['role']*}, where *sig1* = (*user.php*, *SELECT * FROM registration WHERE user_id = Token*), which indicates that the application cannot issue *Query1* at state s_0.

A.2: For a state variable s ($s \in S_{inv}(sig)$) that is always present, its value is drawn from an enumeration set $E(s, sig)$, which indicates the value domain for a specific state dimension. An example in *SimpleOAK* is E(*$row['role']*, *sig2*) = {'professor'}, where *sig2* = (*admin.php*, *SELECT * FROM registration*), which means that *Query2* can only be issued when the application is at state s_2.

Type B: Data constraint invariant

Type B invariants characterize the data constraints associated with SQL queries (i.e., $P \rightarrow O$). For each SQL signature *sig*, we identify:

B.1: The value of a query parameter q ($q \in Q$) is always equal to the value of a state variable that is always present s ($s \in S_{inv}(sig)$). Since the query parameters are drawn from context variables, this type of invariant captures the constraints between the context variables and state variables that the SQL query has to satisfy (i.e., $\mid S \mid \times \mid V \mid$). An example in *SimpleOAK* is the value of *Token1* in *sig3* is always equal to *$_SESSION['userid']*, where *sig3* = (*course.php*, *INSERT INTO registration (user_id, course_id) VALUES (Token1, Token2)*), which means a student can only register a course for himself.

B.2: The SQL signature *sig* is dependent on another SQL signature sig', if and only if: a) the skeleton structure of sig' is a SELECT statement; b) sig' has a Type III response type; c) there is always a SQL query with signature sig' issued before a SQL query with signature *sig* can be issued; d) the WHERE clause of the SQL query with signature *sig* is always satisfied by the set of objects returned by the previous SQL query with signature sig'. This type of invariant captures the constraints, which are not directly encoded within the WHERE clause of current SQL signature but implicitly specified by previously issued SQL queries.

Several examples in *SimpleOAK* are: a) the signature (*course.php*, *UPDATE registration SET grade = Token1 WHERE id = Token2*) is dependent on (*admin.php*, *SELECT * FROM registration*), because the UPDATE query always executes over one of the objects returned by the SELECT statement; b) both the signatures (*course.php*, *SELECT * FROM course WHERE id = Token*) and (*course.php*, *DELETE FROM registration WHERE id = Token*) are dependent on (*user.php*, *SELECT * FROM registration WHERE user_id = Token*).

We present an efficient algorithm to extract this type of invariant, as shown in Fig.IV.5. We use a hashtable (i.e., *SessionStore*) to maintain all the objects, carried by SQL responses, during the session. The key of the hashtable is the signature of an SQL query that has a type III response, while the value is the objects within the response. When a type III SQL response is observed, it is added into the hashtable indexed by the signature of the query (i.e., *Sig*). When an SQL query is issued, its WHERE clause (i.e., *Cond*) will be evaluated over the objects maintained in the hashtable. If the clause is satisfied by the response of a previous SQL query, their pair is added into a candidate set *CandidateSet*. Otherwise, the pair is added into a black list *BlackList* and is no longer evaluated. If the dependency relationship between two signatures holds for all the samples in the trace, it becomes one invariant.

```
SigDepLearn(TRACE Γ)
CandidateSet ← ∅
BlackList ← ∅
SessionStore ← empty
for all queries t IN Γ do
  if a new session then
    SessionStore ← empty
  end if
  Sig ← extractSig(t)
  Cond ← extractWhereClause(t)
  if Cond = null then
    continue
  end if
  for all keys k NOT IN SessionStore do
    Pair ← (Sig, k)
    BlackList.add(Pair)
  end for
  for all keys k IN SessionStore do
    Pair ← (Sig, k)
    if BlackList.exists(Pair) then
      continue
    end if
    if SessionStore.getRespone(k).eval(Cond)=true then
      CandidateSet.add(Pair)
    else
      BlackList.add(Pair)
      if CandidateSet.exists(Pair) then
        CandidateSet.remove(Pair)
      end if
    end if
  end for
  if Sig.getRespType = Type III then
    SessionStore.add(Sig, t.Resp)
  end if
end for
return CandidateSet
```

Figure IV.5: Invariant Extraction for SQL Signature Dependency

Runtime Detection

Each SQL signature sig is associated with a set of invariants $Inv(sig)$ after invariant extraction. The collection of all the invariants serves as the application specification. For detection, each invariant inv is transformed into an evaluation function f_{inv}. If the observed SQL query t satisfies the invariant inv, f_{inv} returns true. Otherwise, the function returns false. An SQL query is accepted and sent to the database if and only if its signature exists and it satisfies all the invariants associated with the signature. Otherwise, the query is blocked and the application receives an error response. In *SimpleOAK*, all the four attack cases can be detected using corresponding invariants extracted in the previous section.

Implementation

We implement a prototype detection system SENTINEL for PHP web applications as two components: the *Sensor* and the *Analyzer*, as shown in Fig.IV.6. The *Sensor* is responsible for collecting information and communicating with the *Analyzer*, while the *Analyzer* is responsible for offline training and runtime detection. To be more specific, the *Sensor* intercepts SQL queries and responses, collects session variable values and script names, and sends them to *Analyzer*. Based on the collected traces, the *Analyzer* extracts SQL signatures and infers the set of invariants associated with signatures. At runtime, the *Analyzer* evaluates incoming SQL queries and instructs *Sensor* to block malicious queries.

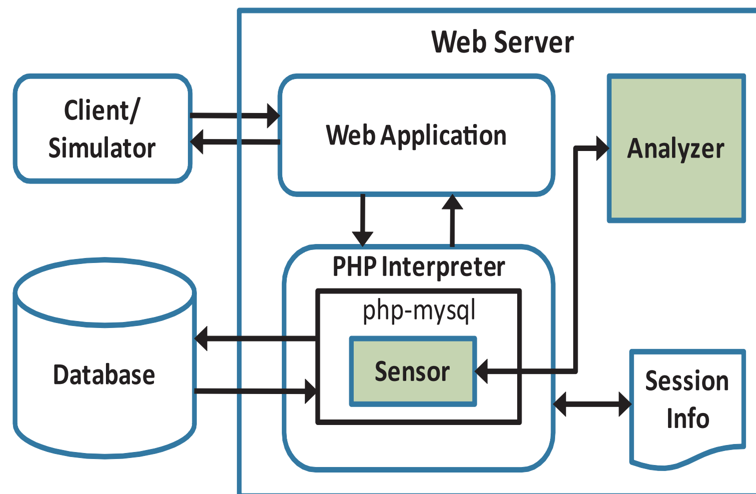


Figure IV.6: Overview of SENTINEL System

To implement the functionalities of the *Sensor*, we modify the *php-mysql* module in the PHP interpreter, which provides the connectivity between the web application and the database. Whenever a SQL query is constructed and is about to be sent to the database, we capture the query string, identify the script currently being executed (via *$_SERVER['SCRIPT_FILE-NAME']*), record the current values of session variables and send them to the *Analyzer*. In PHP, the session variables are by default stored in local files located at */var/lib/php5* and indexed by the current session id. When the SQL response is returned from the database, we capture it and also send it to the analyzer. Our extensions to the *php-mysql* module can be dynamically enabled/disabled through an added PHP directive (i.e., *mysql:enable_proxy*).

The *Analyzer*, the key component of SENTINEL, is implemented as a Java Servlet and can be operated in two modes: training and detection, as shown in Fig.IV.7. In the training mode, *Trace Collector* logs all the information received from *Sensor*. After sufficient traces are collected, *Signature Extractor* generates SQL signatures from observed SQL queries. Then, all of the information (identified by the SQL signatures) are fed into the *Invariant Extractor*, where we leverage the Daikon engine to infer the value-based invariants (i.e., A.1, A.2, B.1). The *Session Manager* is responsible for analyzing sessions and extracting SQL signature dependency invariants (i.e., B.2). In the detection mode, when an SQL query is received from the *Sensor*, the *Signature Extractor* first generates its signature. Then, it is passed to the *Detector* for evaluation based on the set of invariants, identified by its signature. The *Session Manager* is responsible for maintaining the observed objects during the session, when an SQL response is received from the *Sensor*, and evaluating the query with B.2 invariants. If the query is determined to be safe, the *Analyzer* will respond to the *Sensor* and allow the *Sensor* to forward the query to the database. Otherwise, the *Analyzer* will instruct the *Sensor* to block the malicious query and the *Sensor* will return an error response to the application.

The communication between the *Sensor* and the *Analyzer* is based on HTTP. The *Sensor* composes a web request, which contains collected information, sends it to the *Analyzer* and waits for the web response. The *Analyzer* processes the web request via the *Request Handler*. SENTINEL is independent of web applications and database systems and can be easily integrated with existing infrastructures by just replacing the original *php-mysql* module with our extension version. Although our current prototype works for PHP applications, it can be conveniently extended to handle other platforms as long as the functionalities of the *Sensor* (e.g., SQL

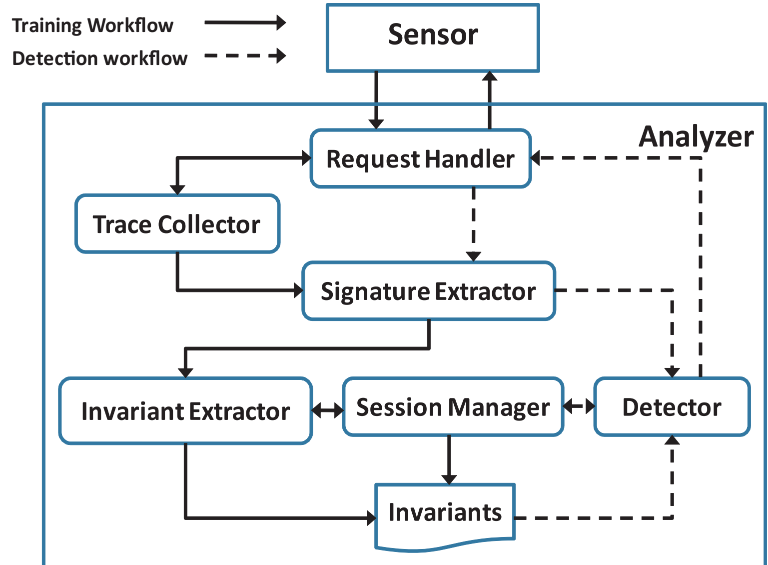


Figure IV.7: The Analyzer Component

interception, the interface for communicating with the analyzer) are implemented. Currently, we co-locate the *Analyzer* with the *Sensor* and the applications, and restrict the access to the *Analyzer*. Our implementation has the potential to be deployed as a security service if additional mechanisms for securing the communication between the *Sensor* and the *Analyzer* (e.g., authenticating the *Sensor*) are implemented.

Evaluation

Experiment Setup

We evaluate SENTINEL using a set of open-source PHP web applications, as shown in Table IV.1. (1) Scarf is a conference management system, which is used for managing sessions, papers, users and comments. It has an authentication bypass vulnerability (CVE-2006-5909), which allows the attacker to directly access the administrative functionalities. The page *generaloptions.php* doesn't check if the session variable *$_SESSION['privilege']*, which indicates the privilege of current user, is equal to *admin* when retrieving and updating system settings and user accounts stored in the database. (2) Wackopicko [68] is an online photo sharing website that

allows users to upload, comment and purchase pictures. It is designed with a number of vulnerabilities, such as cross-site scripting and SQL injection, and used for testing the capabilities of web application vulnerability scanners [25]. In this chapter, we focus on the logic vulnerabilities it contains. The first vulnerability allows an attacker to manipulate the *userid* parameter sent to the *sample.php* page and view any users' information, because the application does not check if the *userid* parameter is equal to the id of the user who is currently logged in. The second vulnerability can be exploited by an attacker to view the high-quality versions of arbitrary pictures without purchasing them by manipulating the *picid* parameter sent to the *highquality.php* page. The application uses a database table *own* to maintain the relationship between the pictures and their buyers. However, it fails to check if the current user has purchased a particular picture when retrieving its high-quality version. (3) OpenIT [47] is an IT management system, which consists of a set of modules, such as inventory, help desk, issue tracking and other features. It has a parameter manipulation vulnerability, which allows the attacker to tamper with a hidden field that stores the employee ID and change other users' information. (4) openInvoice [46] is an invoicing system for keeping track of customers, invoices and items. It has a vulnerability (CVE-2008-6524) that an attacker can exploit to modify the password of an arbitrary user through the *resetpass.php* page.

Table IV.1: Summary of Web Applications (Evaluation of SENTINEL)

Application	# PHP file	Description	Vulnerability
Scarf	21	Conference management system	Auth bypass (CVE-2006-5909)
Wackopicko	52	Photo sharing website	Parameter manipulation, forceful browsing [68]
OpenIT	25	IT management system	Parameter manipulation
openInvoice	327	Invoicing system	Parameter manipulation (CVE-2008-6524)

We deploy all of the web applications on a 2.13GHz Core 2 Linux server with 2GB RAM, running Ubuntu 10.10, Apache web server (version 2.2.16) and PHP (version 5.3.3). The analyzer is hosted by Tomcat 7 on the same machine. To generate traces more efficiently, we build user simulators that automate the procedure of operating web applications and emulate the interaction

between a normal user and the web applications. The details about user simulators can be refered to in Chapter III.

Detection Effectiveness

SENTINEL first runs in the training mode to collect traces, extract signatures and infer invariants. Training traces are generated by both manually operating the web applications and running user simulators and do not include attack instances (e.g., SQL injections). Table IV.2 shows the summary of our training set. In addition to the size of the SQL query set, we also report on the number of extracted signatures, database fields we observed from SQL queries and responses, likely state variables, as well as each type of inferred invariants.

Table IV.2: Summary of Training Set (Evaluation of SENTINEL)

Application	SQL queries	SQL Signatures	Database fields	State variables	A.1 Inv	A.2 Inv	B.1 Inv	B.2 Inv
Scarf	4385	95	58	44	1210	457	71	10
Wackopicko	5062	56	57	29	729	52	130	14
OpenIT	5708	50	145	97	1583	401	596	3
openInvoice	8022	113	51	31	746	375	43	2

Then, SENTINEL runs in the detection mode. The clean test set is also generated by both manually operating the web applications and running the user simulators. Ten attacks are manually launched against each web application under different circumstances, such as logging as a different user and performing a different sequence of actions before launching the attack. Table IV.3 shows the summary of our test sets and detection results. We can see that all of the attacks are successfully identified by corresponding invariants. In Wackopicko for instance, the attack, which allows for the high-quality versions of arbitrary pictures to be viewed without purchase, is detected by one of the B.2 invariants. To the best of our knowledge, none of the existing techniques can capture this type of logic attack, since they do not account for the persistent information (i.e., the table *own*) in the database, which reflects part of the application logic. On the other hand, the false positive rate is fairly low. We analyze the false alerts raised by SENTINEL and find that all of them are introduced by the incomplete exploration of user simulators, which is known as an inherent challenge for dynamic analysis techniques. In summary,

we believe the detection experiments demonstrate the effectiveness of our approach at detecting malicious SQL queries that violate the intended application logic.

Table IV.3: Summary of Detection Result (Evaluation of SENTINEL)

Application	Queries (clean test set)	Queries (false positive)	Attacks	Detected	Invariant violations
Scarf	4694	1	10	10	A.1, A.2
Wackopicko	4984	2	10	10	B.1, B.2
OpenIT	6304	0	10	10	B.1
openInvoice	6585	0	10	10	B.1

Performance Overhead

At runtime, SENTINEL intercepts each SQL query and sends it to the *Analyzer* for evaluation before forwarding it to the database, which inevitably introduces additional SQL response delay. To evaluate the performance overhead induced by SENTINEL, we measure the averaged SQL response time for running each web application under three circumstances: (1) without SENTINEL; (2) with SENTINEL and with the *Detector* disabled; (3) with SENTINEL and the *Detector* enabled to evaluate each SQL query. Fig. IV.8 shows the summary of the performance overhead measured for each application. The results are averaged over a number of rounds. We can see that SENTINEL increases SQL response time by a factor of 1.6 to 4. The performance overhead is introduced mainly through two sources: (1) the communication overhead between the *Sensor* and the *Analyzer*; (2) the analysis time during which the *Analyzer* extracts SQL signature and evaluates the query. While the communication overhead is still acceptable (around 1ms in average), the analysis time is relatively low. Thus, we believe SENTINEL can be integrated into running applications without incurring noticeable performance degradation.

Discussion

Fingerprint-based techniques have been proposed to defend against SQL injection attacks [42]. Our technique extracts SQL signatures to represent the possible output symbols that can be issued by the web application and associates them with invariants, which characterize the application state and data constraints. Similar to a fingerprint, our technique has the potential

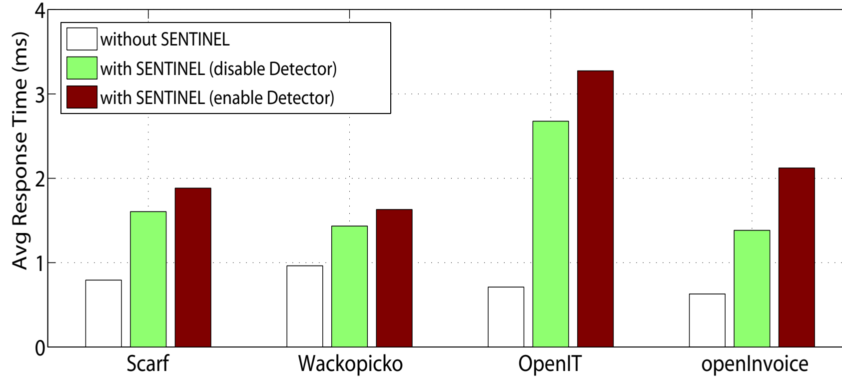


Figure IV.8: Summary of Performance Overhead (Evaluation of SENTINEL)

to be employed to mitigate SQL injection attacks, which tamper with the SQL query structure captured by the SQL signature. Since we focus on state violation attacks in this chapter, we do not evaluate our technique over the detection of SQL injection attacks.

Our technique bears the same limitations as other dynamic analysis techniques. Due to the fact that the user simulator is incapable of exploring all of the possible states of the web application, our inferred invariants can be either incomplete or spurious. The completeness (i.e., coverage) of our technique cannot be directly assessed, because we do not examine the application source code. The attacks that traverse the unexplored state of the application will not be detected by our system, resulting in false negatives. On the other hand, the inferred invariants may be over restrictive for characterizing the application's behavior, so that normal behaviors are identified as attacks, resulting in false positives. In reality, if real world traces are available, our technique can be readily applied.

Since we focus on securing the database access, we collect SQL queries and extract invariants only for SQL signatures. However, our approach can be extended to handle other types of state violation attacks, such as an unauthorized file access. In this case, the *Sensor* should be extended to collect relevant sensitive operations (i.e., file I/O system calls) and the *Analyzer* establishes models for file access behaviors of the web application.

CHAPTER V

AUTOMATIC DISCOVERY OF LOGIC VULNERABILITIES WITHIN WEB APPLICATIONS

In this chapter, we take a first step towards a systematic source-code free approach to identifying logic flaws within web applications. In particular, we model the logic of a web application using a finite state machine (FSM) and formalize logic vulnerabilities as the discrepancies between the intended FSM and the implementation FSM. We infer the intended FSM based on collected web requests, web responses and session variables. We then construct two forms of test inputs to exploit the discrepancies between the intended FSM and the FSM that is actually implemented by the application. We implement a prototype system LogicScope and evaluate it using a set of real world web applications.

The rest of this chapter is organized as follows. We first give an overview of our approach. Then we give an illustrative example and present our system model. The details of our approach and implementation are described in the following two sections. Finally, we conclude this chapter.

Overview

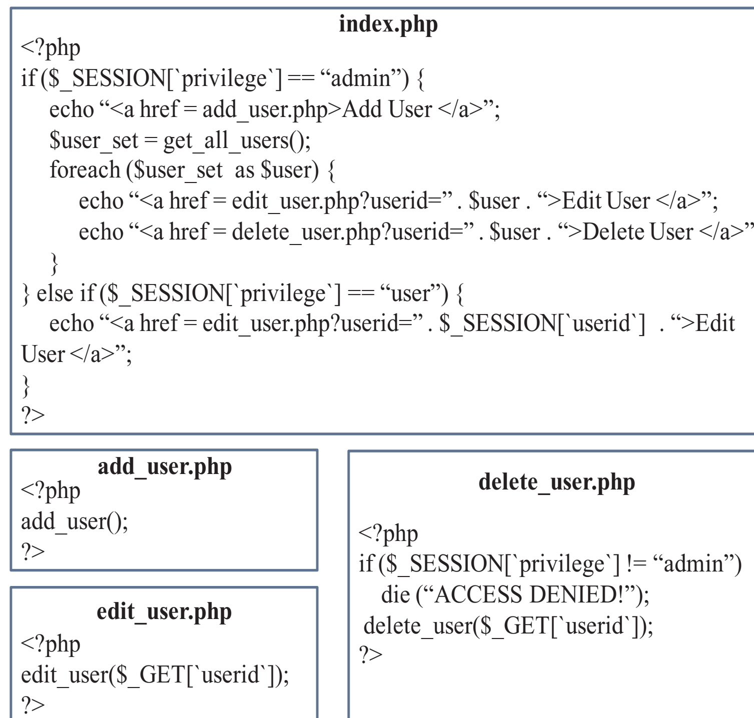
The intended behavior of a web application can usually be observed when benign users follow the navigation paths within the web application [29, 63]. In this case, we say the inputs (i.e., HTTP requests) from the users are *expected*. Over the expected inputs, the behavior of the intended FSM and the implementation FSM is consistent. The discrepancies manifest over the application behaviors when unexpected inputs are fed into the application.

To identify such discrepancies, we first construct the intended FSM as a partial FSM over the observed user inputs (i.e., expected input domain). Then, based on the inferred partial FSM, we test the application over unexpected inputs at each state to identify logic flaws. We present two methods for constructing test input vectors, corresponding to two commonly seen state violation attack vectors (i.e., parameter manipulation and forceful browsing attacks, respectively) and feed them into the application. We provide an evaluation rule to determine whether the corresponding web response leads to a potential logic vulnerability. The reported potential logic vulnerabilities are evaluated manually and classified into real attack vectors and false positives.

Problem Description

Illustrative Example

In Figure V.1, we present a small, vulnerable web application as a running example to illustrate how we formalize and identify logic flaws within web applications. This application uses two session variables $_SESSION['privilege'] and $_SESSION['userid'] to remember the current user's access right and ID. A user, who just logs in, will be redirected to the *index.php* page. If the current user is an admin (i.e., $_SESSION['privilege'] is equal to "admin"), they are presented with links for adding new users, editing and deleting any of the registered users. If the current user is a regular user (i.e., $_SESSION['privilege'] is equal to "user"), they can see only the link for editing their own information.



index.php
```
<?php
if ($_SESSION[`privilege`] == "admin") {
    echo "<a href = add_user.php>Add User </a>";
    $user_set = get_all_users();
    foreach ($user_set  as $user) {
        echo "<a href = edit_user.php?userid=" . $user . ">Edit User </a>";
        echo "<a href = delete_user.php?userid=" . $user . ">Delete User </a>";
    }
} else if ($_SESSION[`privilege`] == "user") {
    echo "<a href = edit_user.php?userid=" . $_SESSION[`userid`]  . ">Edit
User </a>";
}
?>
```

add_user.php
```
<?php
add_user();
?>
```

edit_user.php
```
<?php
edit_user($_GET[`userid`]);
?>
```

delete_user.php
```
<?php
if ($_SESSION[`privilege`] != "admin")
    die ("ACCESS DENIED!");
 delete_user($_GET[`userid`]);
?>
```

Figure V.1: Example Application

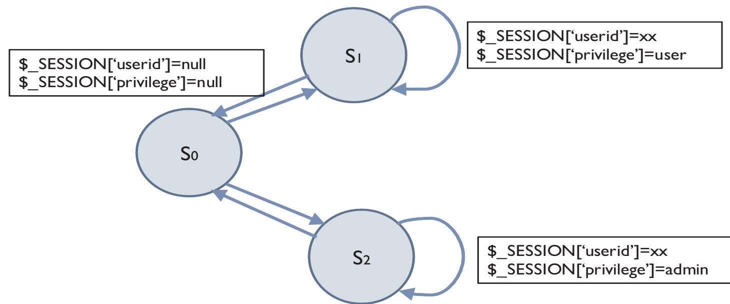
Problem Formulation

We model a web application using a finite state machine (FSM) model (S, s_0, Σ, Λ, T, G), where S is the set of states, $s_0 \in S$ is the initial state, Σ is the set of input symbols (i.e., input alphabet), Λ is the set of output symbols (i.e., output alphabet), $T : S \times \Sigma \rightarrow S$ is the set of transition functions and $G : S \times \Sigma \rightarrow \Lambda$ is the set of output functions. T and G determine the next state and the output symbol, respectively, based on the current state and the input symbol.

To understand the logic flaws within a web application, we need to consider two FSMs associated with it: (1) the ideal FSM (denoted as F_{ideal}) models the intended (or expected) behavior of the web application without any security vulnerabilities; (2) the implementation FSM (denoted as F_{impl}) models the actual behavior of the web application as being implemented by the developer. If F_{impl} is equivalent to F_{ideal}, we say the implemented web application is secure as intended. If there exist discrepancies between F_{impl} and F_{ideal} and the discrepancies involve sensitive information or operations, we say the implemented web application has logic vulnerabilities.

As shown in Figure V.2, the example application has three states: (1) the user is not logged in (s_0), (2) a regular user logs in (s_1) and (3) an admin user logs in (s_2). The state information is maintained by two session variables $_SESSION['privilege'] and $_SESSION['userid'], which are explicitly defined in the application. Each input symbol $I \in \Sigma$ is an abstract representation of a web request, which consists of two parts: 1) the *key* (denoted by K) represents the syntax of the web request (e.g., K_1 = GET-edit_user.php : userid); 2) the *value* (denoted by V) represents the value domains of the parameters, which is related to the semantics of the web request (e.g., V_1 = [userid = $v_{constrained}$($_SESSION['userid'])] means that the userid parameter in the web request is equal to the value of the session variable userid) (we refer the reader to Input Symbolization for details). Thus, $I_1 = K_1.V_1$ and $I_2 = K_1.V_2$ are two different input symbols due to the difference in their parameter value domains, although they have the same syntax structure. Similarly, each output symbol in Λ is an abstract representation of the web responses returned by the application to users.

The ideal FSM (F_{ideal}) for the application works as follows. At state s_1, since it is intended that the regular user can only edit his/her own information, when the regular user sends an input symbol $I_1 = K_1.V_1$, where the userid parameter is equal to the current user id, the application will respond with the edit_user page (output symbol O_1). When the regular user tries to edit another



Input symbols

I_1= K_1. V_1: [GET-edit_user.php : userid] . [userid = $v_{constrained}$(*$_SESSION['userid']*)]
I_2= K_1. V_2: [GET-edit_user.php : userid] . [userid = $v_{unsatisfied}$]
I_3= K_2. V: [GET-delete_user.php : userid] . [userid = *nonnull*]
I_4= K_3. V: [GET-add_user.php : userid] . [userid = *nonnull*]

Output symbols

O_1: edit_user_page
O_2: ACCESS_DENIED
O_3: delete_user_page
O_4: add_user_page

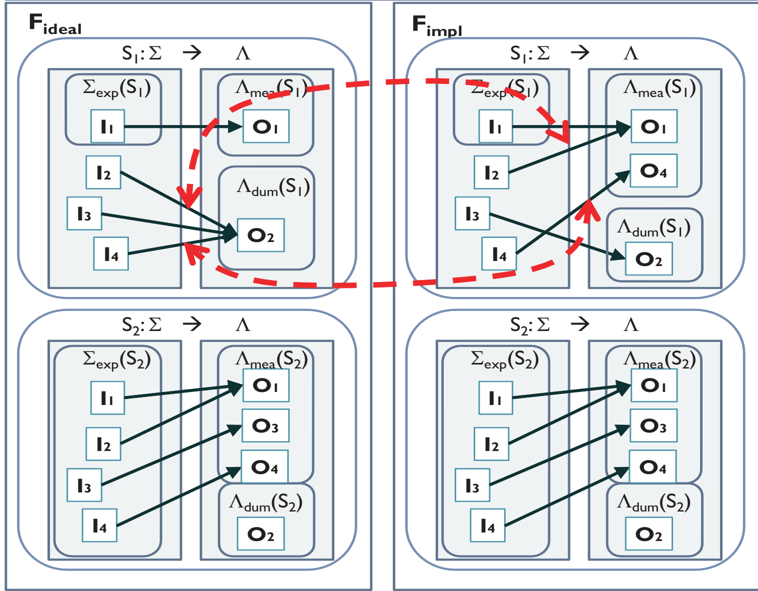


Figure V.2: The FSM Representation of the Example Application

user's information (i.e., by sending $I_2 = K_1.V_2$), delete a user (i.e., by sending $I_3 = K_2.V$) or add a new user (i.e., by sending $I_4 = K_3.V$), the application should reject such input symbols with an ACCESS_DENIED page (output symbol O_2).

However, the implemented application has two logic vulnerabilities, which are reflected as the discrepancies between F_{ideal} and F_{impl}, shown using the dashed lines with arrows in Figure V.2. First, the edit_user.php page does not check if the userid parameter is equal to the current user ID. Second, the add_user.php page does not check if the current user has the admin privilege, as the delete_user.php page does. These vulnerabilities allow two types of attacks. 1) *Parameter manipulation attack*: when input symbol I_2 is sent to the application at state s_1, the output symbol O_1 is returned. This allows a regular user to edit other users' information. 2) *Forceful browsing attack*: when input symbol I_4 is sent to the application at state s_1, O_4 is returned, which allows a regular user to add new users.

Since a user can send any web request to a web application, the input alphabet can be infinite and fed into the application at any state, resulting in corresponding output symbols. However, at a given state s, we observe that only a subset of input symbols are expected by the application (denoted as $\Sigma_{exp}(s)$) and processed to generate "meaningful" output symbols (i.e., $\Lambda_{mea}(s) = G(s, \Sigma_{exp}(s))$). The expected input symbols are the web requests that can be issued (if the user follows the navigation links of the web application and the meaningful output symbols) are the web responses that provide the users with useful information. All the other input symbols, that are not expected at state s, should be rejected/mitigated by the application, resulting in "dummy" output symbols (i.e., $\Lambda_{dum}(s) = G(s, \Sigma - \Sigma_{exp}(s))$). A dummy output symbol means that the application responds to the user with an error page or a redirection header pointing to a previously visited web page, without leaking any useful information. As shown in Figure V.2, for state s_1, the expected input set is $\{I_1\}$, the meaningful output set is $\{O_1\}$ and the dummy output set is $\{O_2\}$. For state s_2, the expected input set is $\{I_1, I_2, I_3, I_4\}$ and the meaningful output set is $\{O_1, O_3, O_4\}$. A web application is expected to fully implement the intended functionality, which means the behaviors of F_{ideal} and F_{impl} over the expected input symbols should be consistent. However, the unexpected input may not be fully mitigated/rejected by F_{impl} as intended by F_{ideal}. Thus, we say a web application has a logic vulnerability at state s, if an input symbol, which is not expected at state s (called a malicious input and denoted by

I_{mal}), is fed into the application, the application generates an output symbol that falls beyond the dummy output set (i.e., $G(s, I_{mal}) \in (\Lambda - \Lambda_{dum}(s))$).

Approach

High-level Overview

As stated in the problem formulation, to identify logic vulnerabilities, we need to construct unexpected (malicious) inputs for each state and evaluate whether their outputs fall beyond the dummy output set. This is nontrivial since we have no knowledge about the entire input alphabet (and thus unexpected input) and the dummy output set at each state. To approach this problem, we first construct a partial FSM over the expected input domain by observing the executions of the application when users follow the navigation paths provided by the application. Then, we leverage the inferred partial FSM to construct unexpected inputs and test the application for logic vulnerabilities. The overview of our approach is shown in Figure V.3.

We collect execution traces, including web requests, web responses and associated session variables, when normal users follow the navigation paths, and analyze the traces to construct the partial FSM through the following essential steps: (1) State Construction, in which we derive the set of application states S using collected session variables. (2) Input Symbolization, in which we abstract concrete web requests into input symbols. This allows us to profile the expected input domain at each state (i.e., $\Sigma_{exp}(s), \forall s \in S$). (3) Output Symbolization, in which we transform web responses (i.e., html pages) into abstract output symbols. This allows us to obtain the mapping between the expected input symbols and the meaningful output symbols (i.e., $G(s, \Sigma_{exp}(s)) \rightarrow \Lambda_{mea}(s), \forall s \in S$). Note that we have no knowledge about the dummy output set (i.e., $\Lambda_{dum}(s), \forall s \in S$), because we only feed expected input symbols into the application. In addition, we learn how the application transitions between the set of states and the corresponding input symbols that trigger the transitions (i.e., $T : S \times \Sigma \rightarrow S$).

Then, based on the partial FSM, we test each application state over the unexpected inputs. In particular, we have to address two issues:

- Test Input Generation: Since we only have knowledge of the expected input set given a state, how do we construct unexpected input symbols to test that state? We present two methods for generating test input vectors in Test Input Generation.

- Output Evaluation: Since we have no knowledge of the dummy output set given a state, how do we determine whether the corresponding output symbol falls beyond the dummy output set, so that we can report a potential logic vulnerability? We provide the evaluation rule in Output Evaluation.

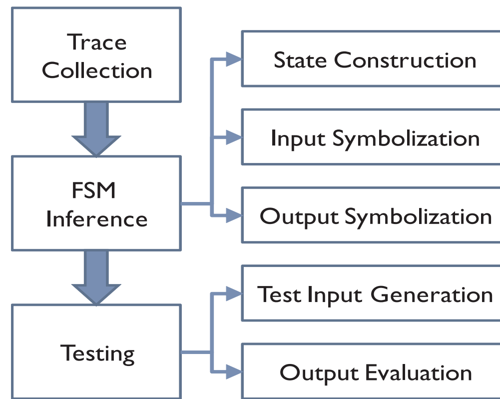


Figure V.3: Approach Overview

State Construction

A web application usually uses session variables to maintain the users' session state. Intuitively, the set of application states can be constructed through the Cartesian product of the value domains of the session variables. However, directly applying this method may result in an infinite number of states, since the value domains of session variables can be infinite. Thus, we first characterize the value domain for each session variable. We employ the KS-test (see Chapter III for details) to analyze the set of values collected for each session variable and categorize each variable into two types: (1) bounded, which means the values the variable may take are bounded to a finite set. For example, $_SESSION['privilege'] only assumes two values: user and admin, indicating two types of users. We represent bounded session variable using the set of observed values; (2) unbounded, which means the number of values the variable may take linearly increases with the number of samples. For example, $_SESSION['userid'] can take the

same number of values as the user number. In this case, we need to consider the value-based invariants between input parameters and session variables. We abstract such variables using null and nonnull for state construction and model such invariants through constrained variables in input symbolization.

Then, we construct the application state using the Cartesian product of the abstract values of session variables, which we refer to as the *state signature*, and identify the set of application states that are observed in the traces. One example state signature is [privilege = user] [userid = nonnull], which represents the application state when a regular user logs into the application. Theoretically, the application state space can be huge depending on the number of session variables and their abstract value domains. In reality, we observe that the actual number of states can be much smaller, since certain session variables are correlated with each other. For example, when a user logs in, both session variables are updated and the following state is impossible: [privilege = user] [userid = null].

Input Symbolization

We need to represent a collection of web requests using abstract input symbols. A web request consists of a HTTP method (we only consider GET and POST here), a URL (the script file name in the case of PHP, e.g., index.php) and a set of input parameters. We symbolize each web request with a two-part structure *Key.Value*: (1) the *key* (denoted by K) represents the syntax of a web request. This part is formed by the combination of the HTTP method and the URL, which we refer to as the request key, with a set of parameters, whose order is sorted alphabetically (e.g., [GET-edit_user.php : userid]). (2) the *value* (denoted by V) represents the value domain of parameters, which indicates certain invariants between the input symbol and the application state. Since a parameter may assume an infinite number of values, we first profile each parameter and construct the value domain by concatenating the sequence of abstract values of each parameter, which is similar to how we deal with session variables.

Since the value domain of parameters reflects the relationship between the input symbol and the application state, we need to profile the value domain of each parameter at each state (i.e.,“local” profiling). In order to construct input symbols that are atomic for the entire application, we need to obtain the global value domain of each parameter by combining their local views (i.e., “global” profiling). Thus, we take two steps to profile each input parameter. In the first step,

we employ the KS-test over the set of values collected for each parameter with the same request key at the same state and categorize the parameter into three types: (1) an unbounded random variable (denoted as p_{ur}), which can take any value - we represent its value domain with two abstract values $\Theta(p_{ur}) = \{null, nonnull\}$, where Θ denotes the value domain. (2) an unbounded constrained variable (denoted as p_{uc}), which can take an infinite number of values, but is subject to certain constraints - here, we only identify one state-related constraint, where the parameter is always equal to a specific session variable (e.g., the parameter userid of request key GET-edit_user.php at state s_1 is always equal to $_SESSION['userid']). We represent its value domain with three values $\Theta(p_{uc}) = \{null, v_{constrained}(sess), v_{unsatisfied}\}$, where $v_{constrained}$ denotes the value satisfying the constraint and $sess$ denotes the session variable name and $v_{unsatisfied}$ denotes all the other values[1]. (3) bounded variable (denoted as p_b), whose value domain is represented with the set of values plus two additional values: null and $v_{outofbound}$, where $v_{outofbound}$ denotes the values that are beyond the bounded set. In the second step, we combine the local views of the value domains of parameters. If the parameter has consistent domain type over all the states, we keep its domain type and re-compute its value domain (i.e., for p_{ub}, the value domain is the set of values divided by multiple constraints; for p_b, additional values are added into the domain). If the parameter has inconsistent domain types over all the states, we use the more restrictive domain type and the further divided value domain $p_b \gg p_{uc} \gg p_{ur}$, where $\gg$ means "is more restrictive than". In the example application, the parameter userid of the request key GET-edit_user.php is constrained by $_SESSION['userid'] at state s_1, but inferred as an unbounded random variable at state s_2. Thus, in the global space, its domain type is identified as an unbounded constrained variable. In this way, we construct two atomic input symbols at state s_2 ($K_1.V_1$, $K_1.V_2$ as shown in Figure V.2).

Output Symbolization

We need to transform a collection of web responses (i.e., html pages) into abstract output symbols. Usually, web responses are generated by feeding dynamic contents into static templates by the application and the number of static templates for a web application is finite. Thus, we represent a web response using its static template, which is the output symbol that is emitted

[1] In a general case, it is possible that one parameter is subject to more than one constraint, whose value domain will be further divided.

by the application. To extract the templates from collected web responses, we leverage the same technique as in Chapter III. We give a brief introduction here, but for more details we refer the reader to Chapter III. We represent the tree-like DOM structure of a html page as a set of paths (like XPath) leading to text nodes in the page, which carry useful information (Step I: transformation). Then, we identify those paths, which occur not so frequently compared to other paths, as dynamic contents and prune them (Step II: pruning). The remaining paths for a web page are more likely to be used in composing the template, which we refer to as "critical paths".

Then, the set of web pages are clustered based on the similarities of its critical paths, to form a number of templates (Step III: clustering). Given a web response, we assign it to a cluster, which it most likely belongs to, and represent it using the corresponding output symbol (Step IV: classification). As shown in Figure V.4, six paths are identified from the *index.php* page seen by the admin user in the example application, three of which are regarded as critical after pruning. Then, one output symbol (i.e., HTML template) *t.index_admin* is identified after clustering and the incoming web page will be assigned to this template because of the high similarities of critical paths.

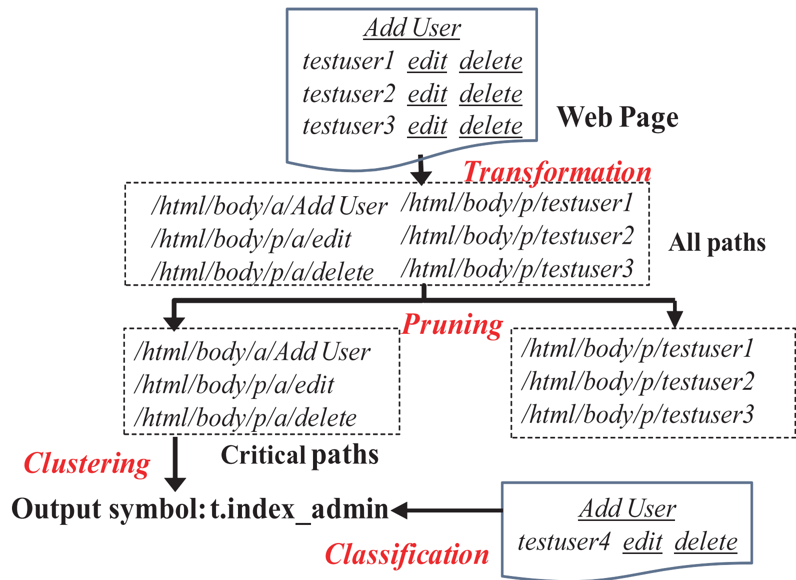


Figure V.4: Output Symbolization

Test Input Generation

We present two methods for generating test input symbols at a given state s, as shown in Figure V.5. Each method is designed to construct one type of attack vector.

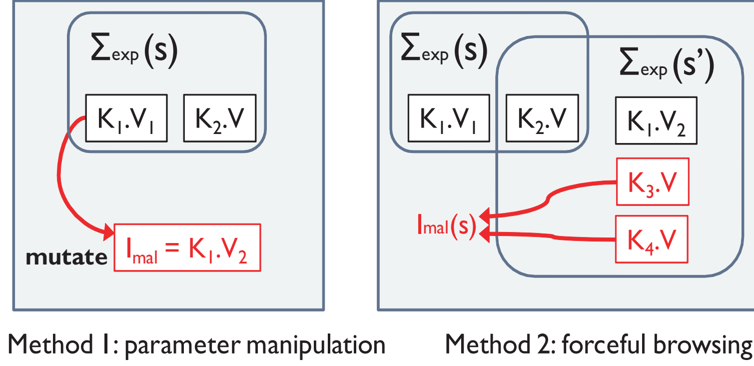


Figure V.5: Test Input Generation

Method 1 – Parameter Manipulation

Given an expected input symbol $I = K.V \in \Sigma_{exp}(s)$ at state s, we directly mutate the input symbol, so that the mutated input symbol falls beyond the expected input set at state s (i.e., $I_{mal} = mutate(I) \notin \Sigma_{exp}(s)$, where $mutate$ is a function over the input symbol). Specifically, the $mutate$ function keeps the *Key* part of input symbol unchanged, but change the *Value* part by varying the values of one or several parameters. For an unbounded constrained variable, we may change its value from $v_{constrained}$ to $v_{unsatisfied}$; for an bounded variable, we may change its value to another in the bounded set or $v_{outofbound}$. For example, as shown in Figure V.5, the *Value* part of input symbol $K_1.V_1$ is mutated to be V_2 as a test input vector for state s. This method mimics the scenario of parameter manipulation attacks, where the attacker tampers the parameter values to violate the constraint between the web request and the current state.

Method 2 – Forceful Browsing

We leverage an expected input symbol from another state s', which falls beyond the expected input set of the current state s (i.e., $I_{mal} \in \Sigma_{exp}(s') - \Sigma_{exp}(s)$). We select input symbols at state s' with a *Key* structure unobserved at state s as test input vectors for state s. For example, as shown in Figure V.5, the input symbols at state s' with *Key* K_3 and K_4 that are not observed

at state s are used as test input vectors for state s. This method mimics the scenario of forceful browsing attacks, where the attacker provides a hidden sensitive link to the application that should not be accessible at current state.

Output Evaluation

Let O_{test} be the output symbol generated by the application after the test input vector I_{mal} is fed into the application at state s. The goal of the output evaluation is to determine whether O_{test} falls beyond the dummy output set (i.e., $O_{test} \notin \Lambda_{dum}(s)$). Since the dummy output cannot be observed directly, we apply the knowledge of the meaningful output set for evaluation. Based on the definition of dummy output, it is straightforward to recognize that there is no intersection between the meaningful and dummy output sets for any two states. Thus, if the test output is the same as any meaningful output, we should report a potential logic vulnerability. Specifically, let I be the original input symbol from which I_{mal} is generated ($I_{mal} = mutate(I)$ in Method 1; $I_{mal} = I$ in Method 2), and O_{orig} be the output symbol generated by feeding the original input symbol I at the corresponding state (i.e., $O_{orig} = G(s, I)$ in Method 1; $O_{orig} = G(s', I)$ in Method 2, where s' is the state from which I is selected from). The output evaluation rule is:

if $O_{test} = O_{orig}$, we report a potential logic vulnerability.

If a test input vector can be generated from more than one input symbol (e.g., $K_1.V_3$ can be generated from both $K_1.V_1$ and $K_1.V_2$), which have different output symbols, the resulting output symbol will be compared to every possible output symbol.

To suppress false positives, we observe that all of the information that is accessible at the initial state (i.e., s_0) should be non-sensitive for all the states. Thus, only when O_{test} does not belong to $\Lambda_{mea}(s_0)$ do we report the violation of the rule as a potential logic vulnerability. Since our technique relies on the collected traces, which may not completely characterize the application behavior, we report those alerts as potential logic vulnerabilities, which require human efforts to analyze and confirm.

Implementation

We implement a prototype system called LogicScope for identifying logic flaws within PHP web applications. Our technique only requires session information from the server side, which is

usually maintained externally to the web application in either local files (e.g., at /var/lib/php5 for PHP) or a database table (e.g., django_session for Python). Note that, though we developed the system for PHP web applications, LogicScope can easily be customized for other platforms (e.g., JSP), because its approach to venerability identification is independent of the programming language and source code. As shown Figure V.6, LogicScope is composed of three major components, including (1) a *Trace Collector*, (2) a *Spec Analzyer* and (3) a *Testing Engine*, which are executed in three phases.

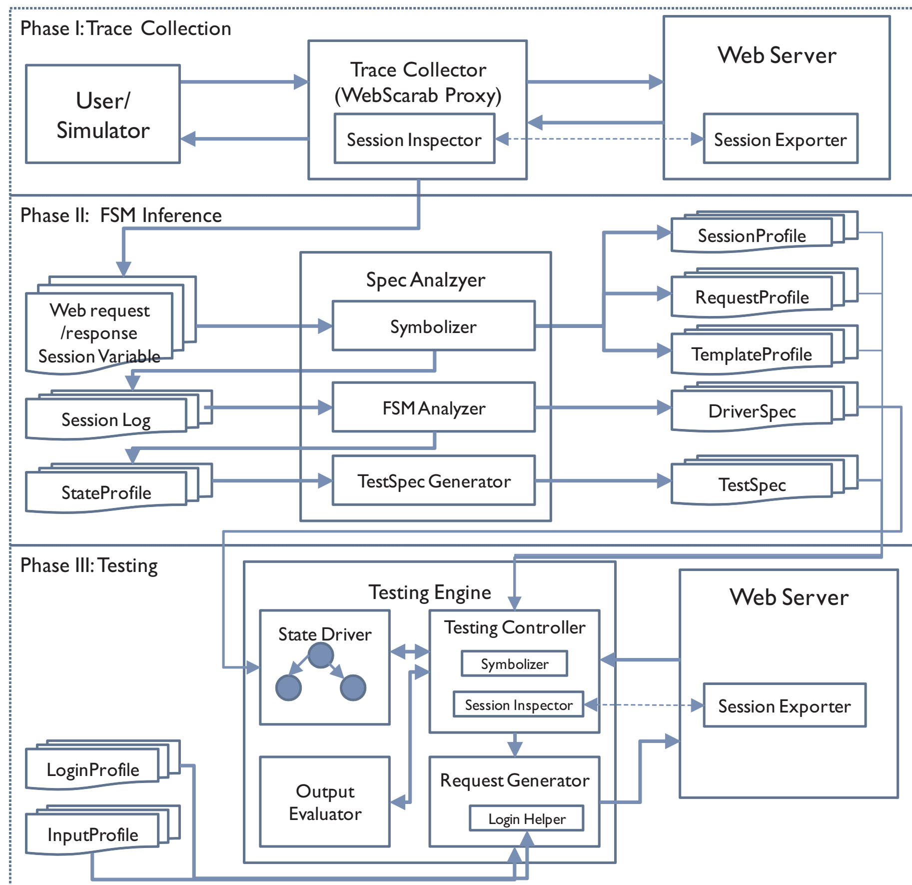


Figure V.6: Prototype System Architecture (LogicScope)

Phase I: Trace Collection

The *Trace Collector* is responsible for collecting web traffic traces when users navigate through the application during attack-free sessions. We implement the *Trace Collector* by enhancing

the WebScarab proxy [50] with a *Session Inspector* module. WebScarab intercepts web requests/responses exchanged between the user and the application, while the *Session Inspector* collects the set of session variables that are associated with each web request/response pair. To do so, we deploy a web service *Session Exporter* on the web server, which will retrieve the set of session variables given a session ID and send them to the *Session Inspector*. To customize LogicScope for other platforms, the only module that needs to be adapted is the *Session Exporter*, since all the other components in LogicScope are independent of the application platform.

Phase II: FSM Inference

The *Spec Analyzer* is executed in Phase II to derive both the partial FSM and the testing specification. The collected traces in Phase I are first fed into the *Symbolizer* module, where session variables are analyzed (resulting in the *SessionProfile*), web requests are profiled (resulting in the *RequestProfile*) and web responses are clustered to form a set of HTML templates (resulting in the *TemplateProfile*). In particular, we implement an invariant inference engine to extract the equality constraint between the parameters and session variables over all samples. Based on the above profiles, the traces are transformed into symbolized session logs, where each set of session variables are replaced with a state signature and each web request/response pair is represented by the corresponding input/output symbol. Then, session logs are used by the *FSM Analyzer* module to derive the partial FSM, resulting in two files: *StateProfile*, which characterizes the mapping between input/output symbols on each state (i.e., output function representation) and *DriverSpec*, which records the transitions between the set of application states, as well as the input symbols that trigger the transitions (i.e., transition function representation). Finally, *StateProfile* is analyzed by *TestSpec Generator* to generate the testing specification, which includes both a set of test input symbols for each state and corresponding output symbols for evaluation.

Phase III: Testing

The *Testing Engine* is executed in Phase III to determine if the application has logic vulnerabilities, based on the above derived profiles and specifications. It instantiates test input vectors into concrete web requests, feeds them into the application and evaluates the corresponding web

responses. The *Testing Controller* is the core module that takes charge of the entire testing procedure. The workflow of this module is shown in Figure V.7.

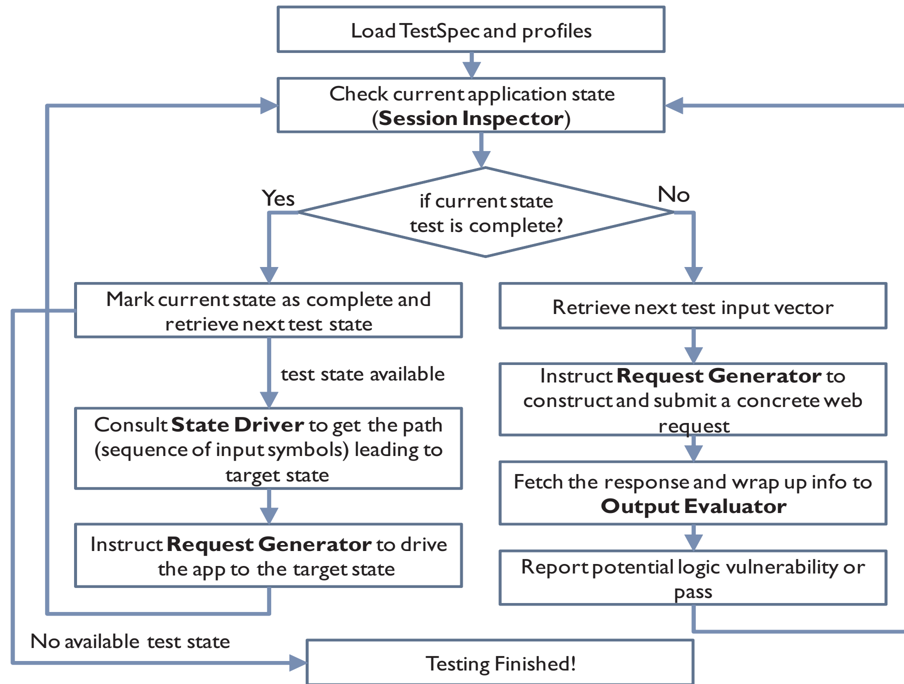


Figure V.7: Workflow of Testing Controller

The *Testing Controller* first loads *TestSpec* and other profiles and checks the current application state using the session ID (usually retrieved from the "Set-Cookie" header within the first web response returned by the application). If the test of the current state is not completed, it retrieves the next available test input vector, delegates *Request Generator* to generate a concrete web request and submits it to the application. After it receives the web response, it will wrap up all the necessary information and send it to *Output Evaluator* for evaluation, where logic vulnerabilities, if exist, will be reported. If the test of the current state is completed (i.e., no test input vectors are left), the *Testing Controller* will move to the next available test state. It will consult the *State Driver*, which loads the *DriverSpec* and keeps track of the transition graph of the application, to get the path leading to the next test state. The path computed by the *State Driver* is essentially the shortest path from the current state to the target state (i.e., a sequence with minimum number of input symbols), which will be instantiated by the *Request Generator* and trigger the state transition step by step to the target state. This mechanism is desirable,

since we cannot directly manipulate the set of session variables to drive the application into our desired abstract state. For the example application, after we test state s_1 for the regular user, we must first log out (i.e., move to state s_0) and log in as an admin user to test state s_2. If all the states have been fully tested, the testing procedure is finished. LogicScope can also perform appropriate URL encoding/decoding, capture essential headers or values (e.g., cookies, encrypted XSRF tokens) and embed them in the following web requests, in order to continue with the live web session, thus be able to handle a variety of web applications well.

One key challenge we need to address is how to instantiate abstract input symbols into concrete web requests with meaningful parameters. In Phase II, when we profile web requests, we also infer the value type (e.g., number, literal string) of each parameter. When the *Request Generator* tries to generate the concrete value for a parameter, it checks its value type and randomly generates a value of that type or retrieves a value from a pre-loaded value store (i.e., *InputProfile*). In particular, *Request Generator* includes *Login Helper* module, which helps *Testing Engine* successfully log into the application. *Login Helper* requires the user to provide a *LoginProfile* file, which specifies the input symbol that represents the login request (e.g., *POST-index.php*) and at least one set of legitimate user credentials (e.g., username and password) for each type of user (e.g., regular user, admin user).

Evaluation

We select six real-world PHP web applications to evaluate our prototype system, LogicScope. We deploy all of the web applications on a 2.13GHz Core 2 Linux server with 2GB RAM, running Ubuntu 10.10, Apache web server (version 2.2.16) and PHP (version 5.3.3). To facilitate trace collection, we developed user simulators for each application based on an open source web application testing tool, Selenium WebDriver [59]. The details of user simulators can be referred to in Chapter III.

Experiment Results

LogicScope first runs in Phase I and Phase II to collect traces and infers the application logic specification. Table V.1 shows the statistics of traces and the inferred FSMs, including the number of files, collected web requests, web responses, session variables, states, input and output

symbols. Then, LogicScope generates the testing specification and launches the testing procedure against each web application. One feature of LogicScope is that it also provides concrete attack vectors and evidences for further inspection. Finally, we manually analyze the reported logic vulnerabilities based on the additional information provided by LogicScope and categorize them as either true vulnerability or false positive. Table VI.2 shows the testing results, including the number of test input vectors generated by each method (*FB* for forceful browsing; *PM* for parameter manipulation), attack instances reported by LogicScope, real attack vectors and false positives (i.e., FP). We also compare the discovered vulnerabilities with the known vulnerabilities from public sources and report the false negatives (FN) (i.e., the number of vulnerabilities LogicScope fails to identify). In summary, we generate 233 test input vectors for all the applications and have 25 attack instances reported, among which, 18 are real attack vectors and 7 are false positives.

Table V.1: Summary of Traces and Inferred FSMs (Evaluation of LogicScope)

Application	Files	Web req.	Web resp.	Session var.	State	Input symb.	Output symb.
Scarf	21	1348	1346	3	3	31	15
Wackopicko	52	2104	1650	1	2	18	7
EventsLister	37	1290	1287	2	2	25	11
Bloggit	24	2657	2645	1	2	18	23
openInvoice	25	1138	1083	5	5	73	11
OpenIT	25	1462	1453	5	5	16	6

Table V.2: Summary of Testing Results (Evaluation of LogicScope)

App	Method	Test input	Flagged attack	Real attack	FP	FN
Scarf	*FB*	5	1	0	1	0
	PM	44	9	8	1	0
Wackopicko	*FB*	5	2	2	0	1
	PM	16	0	0	0	0
EventsLister	*FB*	0	0	0	0	0
	PM	25	4	2	2	0
Bloggit	*FB*	6	2	0	2	0
	PM	20	0	0	0	1
openInvoice	*FB*	19	4	4	0	0
	PM	28	0	0	0	0
OpenIT	*FB*	3	3	2	1	0
	PM	62	0	0	0	0
Summary		233	25	18	7	2

Scarf is a conference management application, which maintains user, paper and session information in the database. It has a known authentication bypass vulnerability (CVE-2006-5909), which allows the attacker to directly access the restrictive page *generaloptions.php* that contains administrative functionalities.

LogicScope successfully identifies this vulnerability and gives eight real attack vectors, which represent different exploits, such as deleting a user or modifying a user's information. Meanwhile, LogicScope introduces two FPs. For example, one FP is triggered by a test vector, which is allowed at the state under test, but never accessible through navigation links. The existence of this FP reflects an inherent challenge of dynamic analysis, where complete exploration of expected input domain over the application states is required during the trace collection phase. We handle this challenge by carefully developing the user emulator and supplying meaningful parameters to explore the application as fully and deeply as possible.

Wackopicko is an online photo sharing website that allows users to upload, comment and purchase pictures. It contains both input validation flaws and logic flaws. Here, we focus on identifying its logic flaws.

LogicScope identifies two logic vulnerabilities. The attacker can tamper with the parameter *userid* within the *sample.php* and *view.php* pages, respectively, to retrieve any other users' information. The corresponding constraints (i.e., the *userid* sent to *sample.php* is always equal to 1 due to a static link, and the *userid* sent to *view.php* always equal to the session variable $_SESSION['userid'] indicating the current user ID) are violated.

LogicScope fails to identify one known logic vulnerability (i.e., false negative), which allows the attacker to view high-quality pictures without purchasing them by manipulating the *picid* parameter sent to the *highquality.php* page. This relationship between users and pictures exists in the database table "own", which cannot be captured by our technique when constructing input symbols. Thus, no attack vectors will be generated to violate such constraint.

EventsLister is an event management application, in which only admin users are allowed to add new events, update or delete events and add new users. It has an known logic vulnerability within the *add_user.php* page, which lacks the security checking function *checkUser()* for verifying if an admin user is logged in. The vulnerability allows the attacker to directly access the *user_-add.php* page and craft a POST request to add a new user.

LogicScope successfully identifies the above logic vulnerability. LogicScope also reports another two attack vectors, which allow the attacker to reset passwords for registered users. Since the attacker has no access to the victims' emails, it cannot pose threats to the application, so we classify them as false positives.

Bloggit is a blog management application, where only admin users are allowed to add/edit/delete blogs, as well as user information and other features. This application has a known Execution After Redirection (EAR) vulnerability (CVE-2006-7014) within the *session.inc.php* page, which allows the attacker to execute administrative functionalities (e.g., adding a user), even after the authorization check fails. All of the files that include *session.inc.php* for authorization checks are vulnerable.

LogicScope is able to construct the attack vectors that exploit the above vulnerability, but fails to report it as such (i.e., false negative). This is because, although the database content has been tampered with by the attacks, LogicScope still observes a redirection response indicating the authorization check fails. Meanwhile, LogicScope introduces two FPs, when a spurious parameter *id* is sent to the *admin.php* page for adding a new user or a new category.

OpenInvoice is an invoicing system for keeping track of customers, invoices and items, etc. LogicScope identifies one logic vulnerability (CVE-2008-6524), which allows the attacker, logged in as a normal user or an inactive user, to modify the password of an arbitrary user, by constructing four attack vectors that violate the constraint over the parameter *uid* (i.e., equal to a session variable indicating the current user ID) sent to the page *resetpass.php*.

OpenIT is an IT management system, which consists of a number of modules, such as employee, news, computer and software. LogicScope identifies one logic vulnerability, which allows the attacker to change arbitrary users' information by tampering with the hidden parameter *employee_EmployeeID* within the form for editing user information. Two attack vectors are constructed to violate the constraint over the parameter (i.e., equal to a session variable indicating the current user ID) under different scenarios. LogicScope also introduces one FP, where the admin user tries to modify his/her own information, which is allowed but not reflected in the traces. As we discussed in the case of Scarf, this is due to a fundamental challenge of dynamic analysis.

Discussion

We analyzed the false positives and identify two major causes. First, our technique relies on complete exploration of expected input domain over the application states, which is an inherent challenge of dynamic analysis. We handle this challenge by carefully developing user simulators and supplying meaningful parameters to explore the application as fully and deep as possible. Still, some false positives are introduced due to missing links to expected input symbols. Second, there may exist correlations or complex constraints (e.g., the database constraint in wackopicko) within input parameters, which we do not capture when we construct input symbols from web requests. The method used by NoTamper [6] in identifying the constraints over parameters by analyzing html pages could be a valuable complement to our technique.

Our prototype LogicScope can handle a large variety of web applications well, since it can perform appropriate URL encoding/decoding, capture essential headers or values (e.g., cookies and encrypted XSRF tokens) and embed them in the following web requests, in order not to break the live web session. However, our technique cannot handle AJAX-heavy applications, where web pages are dynamically updated. The logic flaws within traditional web applications are still not well resolved.

CHAPTER VI

EXPLOITING WEB APPLICATIONS FOR LOGIC VULNERABILITIES OF DATABASE ACCESS

In this chapter, we propose a source-code free approach, which focuses on identifying logic flaws of database access within web applications. We leverage the EFSM model from SENTINEL and infer the intended security constraints over SQL queries issued by the application by collecting SQL queries, SQL responses and session variables. Then, we systematically exploit the application by violating the inferred constraints through manipulating web requests. We implement a prototype system EXPELLER for PHP web applications and evaluate it using a set of real world web applications.

The rest of the chapter is organized as follows. We first give the overview of our approach. Then we give an illustrative example and present our system model. The details of our approach and implementation are described in the following two sections. Finally, we conclude this chapter.

Overview

Chapter IV presents one of the first studies targeting the logic vulnerabilities of database access. SENTINEL is a runtime detection system, which can identify malicious SQL queries issued by the web application. While shown to be effective in preventing malicious SQL queries from accessing the database, this detection approach has several limitations. First, it incurs a non-negligible performance overhead for database access at runtime. Second, as a mitigation mechanism, it does not address the root cause of the malicious SQL query emissions. Built on top of SENTINEL, we aim at identifying the logic vulnerabilities of database access within web applications. In particular, we generate test inputs to exploit the potentially vulnerable application. If the exploit is successful, the concrete attack vectors can greatly help the developers pinpoint and fix the logic vulnerabilities within the web application.

We first identify the intended security constraints for each sensitive operation within the application by observing the interactions between the application and the user, as well as the database. Specifically, we collect the external information of a web application, including web requests, SQL queries/responses and session variables and organize them into web interaction

samples due to the stateless feature of HTTP. Sensitive operations are identified through SQL signature construction based on collected SQL queries. The intended constraints of sensitive operations are inferred by extracting the set of invariants that hold true over all the interaction samples collected for each SQL signature. We model a web application as an Extended Finite State Machine (EFSM) and regard both the web requests and the SQL responses returned by the database as the inputs to the application. Thus, not only do we examine the relationship between the SQL queries and the web requests, but we also explore the constraints that might persist in the database through analyzing SQL responses.

To exploit the web application for logic vulnerabilities, we try to violate the inferred constraints for each SQL signature and examine whether the desired SQL queries can still be triggered. To maximize the possibility of target SQL query emission, we leverage the collected HTTP interaction samples where the queries are emitted during the training phase. We only manipulate selected input fields (within web requests and session variables) of the samples to violate different types of constraints, while keeping the rest of the inputs the same, and replay them to the application. If the SQL signature being tested is observed during the test interaction, we report a SQL violation emission instance, which indicates a potential logic vulnerability. Finally, we manually analyze those instances and confirm real logic vulnerabilities.

Problem Description

Illustrative Example

Fig. VI.1 shows an example application *SimpleOAK*, which is used to illustrate the logic vulnerabilities we address in this chapter and demonstrate our approach throughout this chapter. This application uses two session variables $_SESSION['role'] and $_SESSION['userid'] to remember the current user's privilege and identity. *SimpleOAK* is intended to work as follows when a user follows its navigation paths. The user is first presented with the *login.php* page. After the user inputs the correct login credentials, the application will redirect him to the *user.php* page (for the student role) or the *admin.php* page (for the professor role), depending on the role information (i.e., $_SESSION['role']) retrieved from the database. The *admin.php* page shows all

the students' registrations to the professor via issuing $Query5$ and allows the professor to modify their grades accordingly via issuing $Query6$. The *user.php* page shows the current student's registrations and grades, as well as the syllabus links for registered courses via issuing $Query2$. The student is able to view and modify his own registrations through the *course.php* page via issuing $Query7$, $Query8$ and $Query9$.

SimpleOAK contains several logic vulnerabilities. First, the application fails to enforce the session state constraints for SQL queries, which manifest as two cases.

Case 1: A guest user can directly access a student's information by triggering $Query2$ in *user.php*, since the *user.php* page does not check if the current user has logged in as a student (i.e., *if $_SESSION['role'] == 'student'*).

Case 2: Although a guest user or a student cannot directly access the professor's page due to the security check in *admin.php* (i.e., *if $_SESSION['role'] == 'professor'*), the redirection fails to stop the program execution, which still allows the attacker to trigger $Query5$ and $Query6$. For instance, $Query6$ can be triggered on the attacker's behalf to modify the students' grades in the database. This flaw is also referred to as an Execution After Redirection (EAR) vulnerability [23].

Second, the application fails to enforce the query parameter constraints associated with SQL queries, which allows the attacker to issue malicious SQL queries by manipulating SQL query parameters.

Case 3: A student is able to view/change another student's registrations.

- View registration: the *user.php* page fails to check the constraint: *$_GET['userid'] == $_SESSION['userid']* associated with $Query2$.
- Register a course: the *course.php* page fails to check the constraint: *$_POST['userid'] == $_SESSION['userid']* associated with $Query7$.
- Unregister a course: the *course.php* page fails to check if the *user_id* field in the affected row of *registration* table is equal to the session variable $_SESSION['userid'], when issuing $Query8$.

Case 4: A student can view the syllabus of a course for which they have not registered, by manipulating the *course_id* parameter in $Query9$. The *course.php* page fails to check the

```
login.php
<?php
$uname =$_POST[‘uname’];
$pwd =$_POST[‘pwd’];
if (isset($uname) & isset($pwd) ) {
  $Query0 = mysql_query(“SELECT id FROM users WHERE
login=” . $uname . “ AND pwd=”.$pwd. “;”);
  if ($Query0) {
    $_SESSION[‘userid’] = get_id($Query0);
    $Query1= mysql_query(“SELECT * FROM user WHERE
user_id = ” $_GET[‘userid’] . “;”);
    $_SESSION[‘role’] = get_role($Query1);
    if ($_SESSION[‘role’] == “professor”) {
        header(“Location: admin.php?userid=” .
$_SESSION[‘userid’]);
    } else if ($_SESSION[‘role’] == “student”) {
        header(“Location: user.php?userid=” .
$_SESSION[‘userid’]);
    }
  } else {
    die(“Wrong username or password!”);
  }
}
return_login_form();?>
```

```
admin.php
<?php
if ($_SESSION[‘role’] != “professor”) {
    header(‘login.php’);
}
if (isset($_GET[‘userid’])){
    $Query4 = mysql_query(“SELECT * FROM user WHERE
    user_id = ” $_GET[‘userid’] . “;”);
    print_admin_info($Query4);
    $Query5 = mysql_query("SELECT * FROM registration;");
    print(“<table><tr><td>Name</td><td>Course</td><td>Grade</t
    d></tr>”);
    while ($row = mysql_fetch_assoc($Query3)) {
        print(“<tr><td>” . getUserName($row[‘user_id’]) .
            “</td><td>” . getCourseName($row[‘course_id’]) .
            “</td><td><form method=\“post\”
    action=\“course.php\”>” .
          “<input type=\“hidden\” name=\“register_id\” value=” .
    $row[‘id’] . “>” .
            “<textarea name=\“grade\”>” . $row[‘grade’] .
    “</textarea>” .
            “<input type=\“submit\” name=\“action\”
    value=\“Modify\”></form></td><tr>”);
    }
} else if (isset($_POST[‘register_id’]) && isset($_POST[‘grade’])
&& $_POST[‘action’] == “Modify”) {
    $Query6 = mysql_query(“UPDATE registration SET grade=”
. $_POST[‘grade’] . “WHERE id= ” . $_POST[‘register_id’] . “;”);
} ?>
```

```
user.php
<? php
if (isset($_GET[‘userid’])){
    $Query2 = mysql_query(“SELECT * FROM registration
WHERE user_id=”. $_GET[‘userid’]. “;”);

print(“<table><tr><td>Course</td><td>Grade</td><td>Syllabus</t
d><td>Unregister</td></tr>”);
    while ($row = mysql_fetch_assoc($Query2)){
        print(“<tr><td>”. getCourseName($row[‘course_id’]).
“</td><td>” . $row[‘grade’].
            “</td><td><a href=\“./course.php?course_id=” .
$row[‘course_id’]. “\”>link</a>” .
            “</td><td><form method=\“post\”
action=\“course.php\”>” .
            “<input type=\“hidden\” name=\“register_id\” value=”
. $row[‘id’] . “>” .
            “<input type=\“submit\” name=\“action\”
value=\“Unregister\”></form></td></tr>”); }
    $Query3 = mysql_query(“SELECT * FROM course
WHERE course.id NOT IN (SELECT course_id FROM
registration WHERE user_id=“ . $_GET[‘userid’]. ”)”);
    print(“<table><tr><td>Course</td><td>Register</td></tr>”);
    while ($row = mysql_fetch_assoc($Query3)){
        print (“<tr><td>”. $row[‘name’]. “</td><td><form
method=\“post\” action=\“course.php\”>” .
            “<input type=\“hidden\” name=\“course_id\” value=” .
$row[‘id’]. “>” .
            “<input type =\“hidden\” name=\“user_id\” value=” .
$_GET[‘userid’] . “>” .
            “<input type=\“submit\” name=\“action\”
value=\“Register\”></form></td></tr>”);
    }
} ?>
```

```
course.php
<? php
if ($_SESSION[‘role’] != “student”) {
    die(“You are not authorized!”);
}
if (isset($_POST[‘course_id’]) && $_POST[‘action’] ==
“Register”) {
    $Query7 = mysql_query(“INSERT INTO registration
(student_id, course_id) VALUES (”.$_POST[‘user_id’] . “, ” .
$_POST[‘course_id’]. “);”);
} else if (isset($_POST[‘register_id’]) && $_POST[‘action’] ==
“Unregister”) {
    $Query8 = mysql_query(“DELETE FROM registration
WHERE id=” . $_POST[‘register_id’]. “;”);
} else if (isset($_GET[‘course_id’])){
    $Query9 = mysql_query(“SELECT * FROM Course
WHERE id =” . $_GET[‘course_id’]. “;”);
    print_course_syllabus($Query7);
} ?>
```

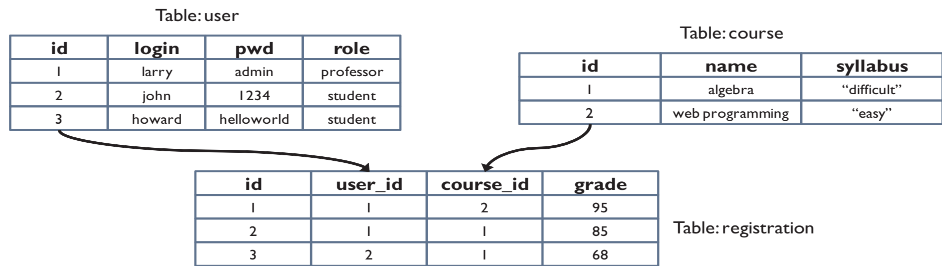


Table: user

id	login	pwd	role
1	larry	admin	professor
2	john	1234	student
3	howard	helloworld	student

Table: course

id	name	syllabus
1	algebra	“difficult”
2	web programming	“easy”

Table: registration

id	user_id	course_id	grade
1	1	2	95
2	1	1	85
3	2	1	68

Figure VI.1: Example Application: *SimpleOAK*

correlation between the student and the course, which exists within *registration* table (i.e., the directed lines in Fig. VI.1).

To date, there is no automated mechanism that can help developers identify all the above security flaws. Developers have to pay extra attention and manually place appropriate checks during development and code auditing.

System Model

We model a web application as an *extended finite state machine (EFSM)* [28], denoted as M. A web application M, as shown in Fig.VI.2, is defined as a seven-tuple: $M = (S, V, I, O, P, U, T)$.

- S denotes the finite set of application states. A web application maintains its state using both session variables (e.g., $_SESSION['userid'])

- V denotes the set of context variables during current execution, which include both global variables and local variables in scope (e.g., $_POST['uname'] in *login.php*).

- I is the set of input symbols, which include users' web requests (e.g., *GET:user.php?userid=3*) and SQL responses (e.g., *$Query0*, which contains the response of *Query0*) returned by the database. Note that the database holds a large amount of information that may affect the application's behavior. The sheer volume of data makes it extremely hard to incorporate as application state directly. We observe that information that comes from persistent objects can only be used by the application after it is retrieved from the database in the form of SQL responses. Thus we model the database state information though the SQL responses as inputs to the application.

- O is the set of output symbols, which include web responses sent to users and SQL queries issued to the database (e.g., *Query1*).

- P: $V \rightarrow \{true, false\}$ is the set of trigger functions evaluated over context variables. Only when specific trigger functions are evaluated to be true, can state transitions be enabled and output symbols emitted.

- U: $V \rightarrow V$ is the set of update functions, which update context variables (e.g., *$_SESSION['userid'] = get_id($Query0)* in *login.php*) based on external input.

- T: $S \times I \times P \rightarrow S \times O \times U$ defines the state transitions. The web application first accepts the external input and executes update functions. If the desired trigger functions over updated variables are evaluated to be true, the application may transition to a new state and emit the corresponding output symbols. For example, after *SimpleOAK* accepts the SQL response of *Query0* (i.e., *$Query0*) from the database, it will update context variables $_SESSION[‘role’] and $_SESSION[‘userid’]. Then, depending on the user role (i.e., professor or student), the application will transition to a new state and return the corresponding output symbols (*header(“Location: admin.php”)* or *header(“Location: user.php”)*) to the user.

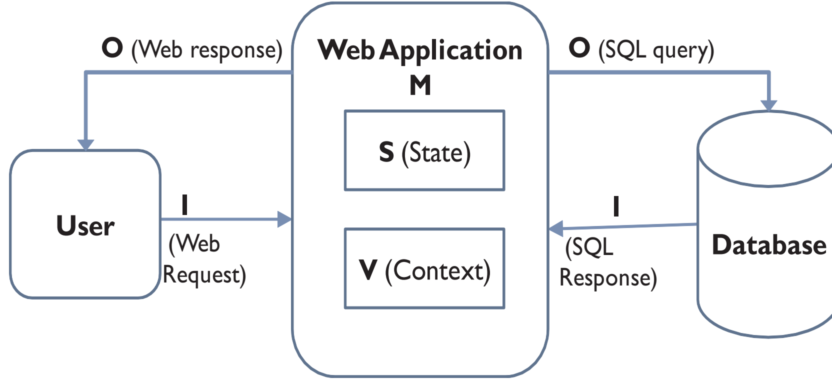


Figure VI.2: System Model

Problem Formulation

The application logic specifies that certain conditions need to be satisfied before sensitive operations can be performed. We focus on identifying logic vulnerabilities of database access within web applications in this chapter. Thus, we regard SQL queries issued by the web application as sensitive operations, which are used for retrieving and manipulating sensitive information in the database. From the above system model, we can see that in order for an SQL query (i.e., O) to be issued, the application has to reside in a certain state, accept desired external input and satisfy the trigger functions (i.e., $S \times I \times P$).

Thus the conditions to be satisfied for triggering SQL queries can be captured through constraints extracted over session variables (for state S) and context variables, whose values come

from the request parameters or the SQL responses (I). In particular, we identify two types of constraints associated with SQL queries, i.e., session state constraint and query parameter constraint. Session state constraint, represented solely by session variables, indicates the application state, while query parameter constraint manifests the desired relationship between context variables (since query parameters also belong to context variables).

Formally, let $\Phi(q)$ be the constraint that is intended to be satisfied before SQL query $q \in O$ is issued, which is built on a set of primitive constraints using boolean operators. Each primitive constraint is represented as a predicate that is evaluated over one or several context variables and returns either true or false. For instance, *$_SESSION['role'] == "student"* is a primitive constraint. In *SimpleOAK*, the intended constraint for issuing $Query3$ in the *user.php* page is $\Phi(Query3) = \Phi_1 \wedge \Phi_2 \wedge \Phi_3$ consists of three primitive constraints. Φ_1 is *$_SESSION['userid'] != null* and Φ_2 is *$_SESSION['role'] == "student"*, which belong to session state constraint and collectively indicate that a student logs into the application. Φ_3 is *$_GET['userid'] == $_SESSION ['userid']*, which belongs to the query parameter constraint and indicates the web request parameter $_GET['userid'] that is propagated into the query as a query parameter has to equal the current user ID.

The intended constraints are usually implemented as explicit security checks in the source code or implicitly implied by the navigation paths offered to users. If the intended constraints are not always enforced in the application implementation, logic vulnerabilities are introduced, which allow sensitive operations to be performed under undesired circumstances. There are two scenarios where such vulnerabilities can exist.

(1) Missing check: when users are assumed to follow the navigation paths provided by the application, security checks may be missing on unexpected paths leading to sensitive operations. Vulnerability examples include Case 1, 3 and 4 in *SimpleOAK*.

(2) Futile check: although security checks are placed before sensitive operations, they fail to stop the attackers from triggering sensitive operations. Case 2 in *SimpleOAK* is one example of this type.

All of the above logic vulnerabilities are rooted in the fact that the application fails to enforce the intended constraints for sensitive operations completely and correctly. Formally, we say a web application has logic vulnerabilities of database access, if there exists one SQL query q that can still be issued by the application while its intended constraint is violated. Our objective is

to identify the existence of such logic vulnerabilities and demonstrate it using concrete malicious inputs towards the web application.

Approach

High-level Overview

To identify logic vulnerabilities within a web application, we first infer the intended constraints for each sensitive operation. The intended behavior of a web application can be captured through observing its normal execution when users follow navigation paths offered by the application [63]. We refer to this step as the *constraint inference* phase. Then, we exploit the application by constructing malicious inputs which violate the inferred constraints and examine whether the sensitive operations can still be triggered. We refer to this step as the *vulnerability exploitation* phase.

In the *constraint inference* phase, we first collect execution traces when users follow navigation paths provided by the application. Then, we identify the set of sensitive operations from collected SQL queries by constructing SQL signatures. We extract a set of invariants from execution traces for each SQL signature as the intended constraints, including both session state constraints and query parameter constraints.

In the *vulnerability exploitation* phase, we try to trigger the SQL queries being tested while violating their intended constraints. In particular, to violate the session state constraints, we directly manipulate the session variables. Since SQL query parameters can not be directly changed by a user, in order to violate the query parameter constraints, we need to manipulate the SQL query parameters through the web request parameters. If the SQL query being tested can still be issued, we report a violating SQL emission instance, which will be manually analyzed to confirm whether it manifests a real logic vulnerability. The overview of our approach is shown in Figure VI.3.

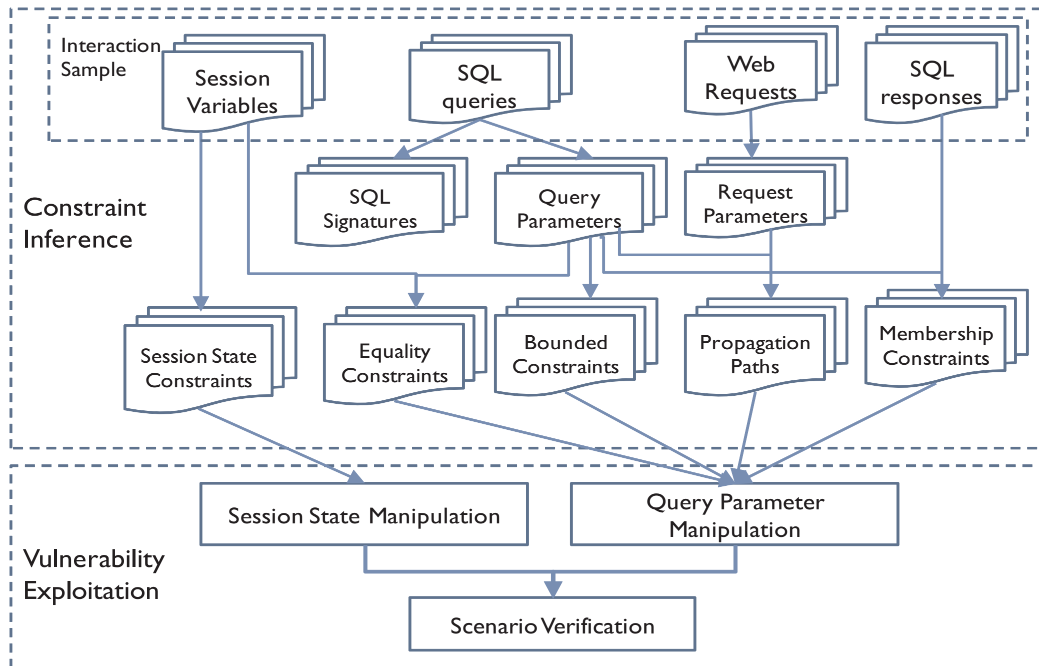


Figure VI.3: Approach Overview

Trace Collection

The stateless nature of HTTP allows us to model the behavior of a web application through independent sequences of web requests/responses. Specifically, we refer to the process starting from the moment when the user sends a web request to the application to the moment when the user receives the web response as one *interaction*. Figure VI.4 shows an example interaction.

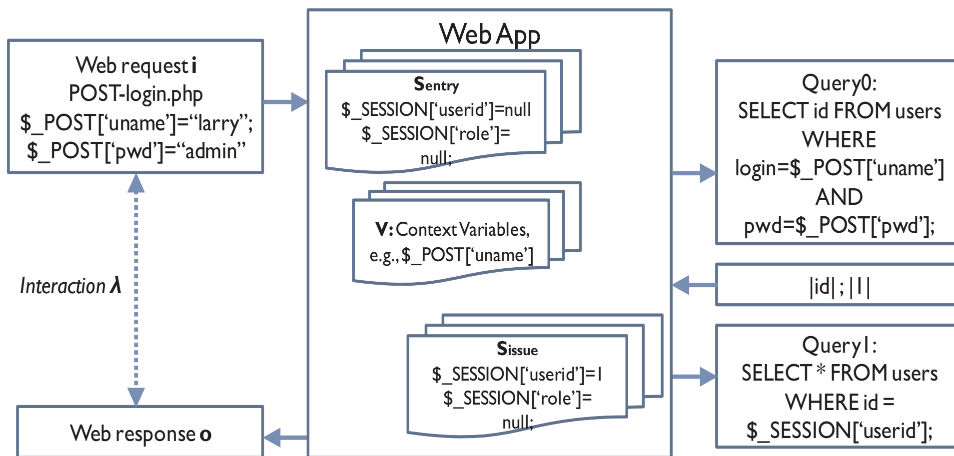


Figure VI.4: HTTP Interaction Example

We collect the set of interactions observed during normal executions when users follow navigation paths as the application execution traces. We organize the traces into interaction samples. Each sample includes four parts: a web request (including a request key and a set of request parameters), the entry session state S_{entry}, a list of SQL queries and the corresponding issuing session states S_{issue} and SQL responses.

We represent web requests using two parts: web request key and request parameters. The web request key is defined as the combination of the HTTP method (here we only consider GET and POST) and the URL path (in php, it is the script file name which the request points to). An example web request key is *GET-login.php*. Web request parameters are essentially a vector of key-value pairs.

Session variables are usually maintained by the web application to serve users across multiple interactions. We represent the session state using a vector of session variable key-value pairs and refer to it as *session state vector*. In particular, we refer to the value of the session state vector when the application receives the web request as *entry session state* (denoted by S_{entry}).

During one interaction, one or more SQL queries can be issued to the database and corresponding SQL responses are returned to the application. We refer to the session state vector when each SQL query is issued as *issuing session state* (denoted by S_{issue}). Note that S_{issue} can be different from S_{entry}, since the session variables may be updated by the application input (including both web requests and SQL responses) during the interaction. For example, as shown in Figure VI.4, a POST web request points to the *login.php* page in the example application, when the user has not logged in (i.e., $_SESSION['userid'] is null). After the application verifies the user's credential by issuing $Query0$ to the database and checking the SQL response, session variables are updated to remember that a user logs in (i.e., $_SESSION['userid'] is nonnull). Then, the application issues $Query1$ to retrieve some configuration information for the user. We can see for $Query1$ the issuing session state S_{issue} is different from the entry session state S_{entry}.

SQL Signature Construction

Each SQL query is composed of a skeleton structure, which is programmed in the source code, and a set of query parameters, whose values are dynamically fed by the application at runtime. To identify the SQL queries that correspond to the same sensitive operation from the trace, we need to separate their skeleton structure from query parameters. In particular, we

replace the value of each query parameter with a place holder and get the skeleton structure of each SQL query (e.g., *SELECT * FROM registration WHERE user_id = Token*). Then, we construct the SQL signature by combining the skeleton structure with the script file name, where the query resides. For example, the SQL signature for $Query1$ is: "*[user.php] [SELECT * FROM registration WHERE user_id = Token]*".

Constraint Inference

To infer the intended constraints for each SQL signature from execution traces, we first identify all the interaction samples within which there exists at least one SQL query that matches the SQL signature q. For each SQL signature, we identify two types of constraints, i.e., session state constraint and query parameter constraint, which manifest as invariants extracted from the collected interaction samples. Each type of the primitive constraint we infer here is a necessary condition for issuing the corresponding SQL query q. As we described in SectionVII, the overall constraint that needs to be satisfied for issuing query q is the conjunction of these primitive constraints.

Session State Constraint

Session state constraint indicates the intended session state whenever the SQL query is issued. Thus, we analyze the set of issuing session states S_{issue} for each SQL signature to infer session state constraint. For example, whenever $Query2$ is issued, we observe that: $_SESSION['userid'] is not null and $_SESSION['role'] is always equal to "student". In particular, we extract two types of constraints:

- Constraint 1.1 (Persistent Session Variable): A specific session variable is required to be set (i.e., its value is not null).

For example, in *SimpleOAK*, when $Query2$ and $Query3$ are issued, both $_SESSION['role'] and $_SESSION['userid'] are always set to indicate a user has logged in.

- Constraint 1.2 (Bounded Session Variable): The value of a specific session variable is bounded to a finite set.

For example, in *SimpleOAK*, when $Query2$ and $Query3$ are issued, $_SESSION['role'] is always equal to "student", indicating the current user is a student, while when $Query5$ and $Query6$ are

issued, $_SESSION['role'] is always equal to "professor", indicating only the professor can trigger these two operations.

We note that session variables are usually correlated to each other. For example, when $_SESSION['role'] is set to a specific value, $_SESSION['userid'] is always set to a non-null value. To capture this relationship between session variables, we use a vector of symbolic values of session variables to represent session state constraints. To transform its concrete value to its symbolic value, we first analyze each session variable over all its observed values and categorize it as either a bounded variable or an unbounded variable based on one-sample Komoglov-Smirnov's D Statistics test (KS-test) [42]. If a session variable has a bounded value domain (e.g., $_SESSION['role'] only takes "professor" or "student"), its symbolic values are the same as the concrete values. If the session variable has an unbounded value domain (e.g., $_SESSION['userid'] can take infinite number of values), we use null and nonnull to represent its symbolic values. In this way, we can represent the above two session state constraints using one symbolic session state vector. For example, whenever *Query*2 and *Query*3 are issued, only one session state vector is observed: {[session.userid = nonnull] [session.role = student]}.

Query Parameter Constraint

Query parameter constraint characterize the value domain of SQL query parameters and the relationship between query parameters and context variables, whose values come from the request parameters or the SQL responses, as well as session variables. Some constraints are directly encoded within WHERE clauses of SQL queries and automatically enforced by the database (e.g., WHERE course.id NOT IN in *Query*3). In addition, we analyze the observed values of SQL query parameters and the relevant context variables to identify the following three parameter constraints.

- Constraint 2.1 (Bounded Query Parameter): The value of a specific SQL query parameter is bounded to a finite set.

This constraint indicates the value of the query parameter takes implicit semantics in the application and should not be set to an arbitrary value that is beyond this set.

- Constraint 2.2 (Query Parameter Equality): The value of a specific query parameter is always equal to the value of a session variable, when the SQL query is issued.

This constraint, which is extracted from the values of query parameters and issuing session state S_{issue}, brings about the value-based relationship between SQL queries and the session state, which cannot be captured by a finite set of symbolic session state vectors (as in Constraint 1.1 and 1.2). For example, in order to trigger $Query2$ and $Query3$ in *SimpleOAK*, the query parameter $_GET['userid'] is intended to always reflect the current user id (i.e., $_SESSION['userid']).

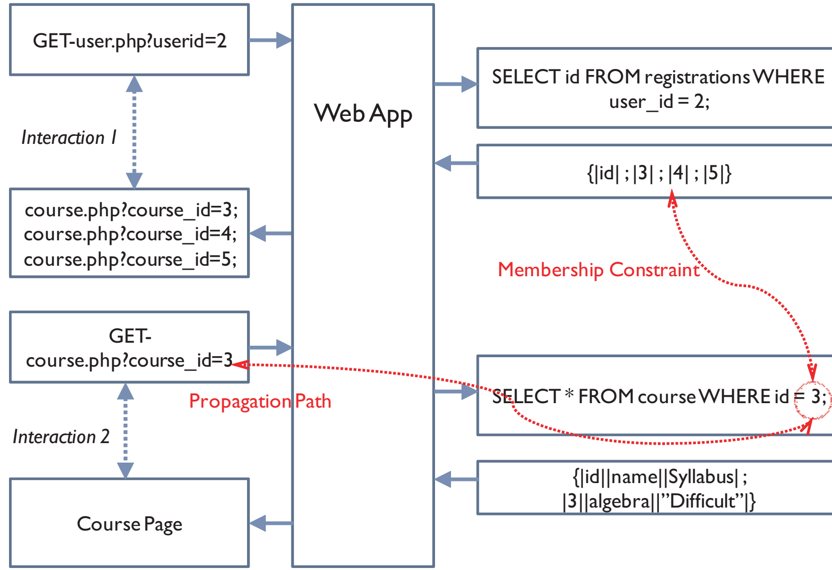


Figure VI.5: Query Parameter Membership Constraint

We observe that the constraints may also be embedded within SQL queries issued from previous interaction, since the application assumes users follow navigation paths (e.g., clicking links or submitting forms within the web response), through which the information as well as the constraint from previous interaction can be passed along to next interaction. For example, as shown in Figure VI.5, *SimpleOAK* first retrieves all the registered courses (at least the course id) for one student via *Query2*. Then, the student can view the syllabus of a course by clicking one of the course links, which will trigger *Query9*. Thus, the query parameter *course_id* in *Query9* always satisfies the constraint (i.e., WHERE clause) in *Query2*, which indicates that the student can only view the syllabus of registered courses.

Such constraints between two subsequential SQL queries can be observed through the relationship between the data object within SQL response of the first query and the parameters

in the second SQL query, as the results returned by the SQL response always satisfy the constraints associated with the SQL query. Thus we infer the following constraint by analyzing query parameters and the list of SQL responses observed within previous interaction.

- Constraint 2.3 (Query Parameter Membership): The value of a specific query parameter always comes from an SQL response returned by the database within the previous interaction.

This constraint captures the requirements over query parameters intended by the application, which are implied by navigation paths. Several examples in *SimpleOAK* include: a) the parameter *id* in the signature (*admin.php*, *UPDATE registration SET grade = Token1 WHERE id = Token2*) always comes from the response variable *id* returned for the signature (*admin.php*, *SELECT * FROM registration*), since the UPDATE query always performs over one of the objects returned by the SELECT statement; b) the parameter *id* in (*course.php*, *DELETE FROM registration WHERE id = Token*) always comes from the response variable *id* returned for the signature (*user.php*, *SELECT * FROM registration WHERE user_id = Token*).

To infer this constraint, we attach every interaction sample with the interaction that precedes it, which contains a list of SQL query-response pairs. Then, we compare the value of each query parameter with the SQL response values from the previous interaction. If the value of query parameter always belongs to the value set of a specific response variable for all interaction samples, we identify it as a constraint.

Vulnerability Exploitation

The existence of logic vulnerabilities can be identified if we are able to construct a sequence of web requests so that some SQL queries can be issued while their intended constraints are violated. First, we need to reconstruct an execution scenario, when the SQL query can be possibly triggered. Second, we need to manipulate the execution scenario to violate the constraints under test and replay it to see whether the desired SQL query can still be issued. If so, we report that there exists a violating SQL emission instance, which indicates a potential logic flaw.

The reconstruction of the execution scenario can be nontrivial, since a random sequence of inputs has a very low chance of triggering the desired SQL query. Here we leverage the interaction samples collected for the desired SQL query for scenario reconstruction. Due to the

stateless feature of HTTP, each interaction sample is able to reproduce the desired SQL query if it is replayed by setting the application state to be the same as the entry session state and feeding the web request to the application. Directly setting the application session state is equivalent to feeding a sequence of web requests to drive the application into the desired session state. To identify the candidate sample for replay, we randomly select one sample from the set which shares the same request key, since the web request key marks the entry point of the application.

Manipulation of the execution scenario to violate the constraint under test also requires careful selection of parameter values, because a randomly generated value might be trivially rejected by the application. For example, we cannot feed a string value when the application is expecting a number value. We capture the legitimate value domains for the parameters appearing in the constraints based on their values from the trace samples and randomly select a value from its domain as test inputs. In the following, we illustrate how we manipulate the selected interaction sample to violate session state constraints and query parameter constraints, respectively.

Session State Manipulation

Violating the session state constraints requires the manipulation of the issuing state variables S_{issue}, which can not be done directly by users during program execution. We notice that S_{issue} is actually derived from S_{entry}. Thus, it is possible that we violate the constraints over S_{issue} by manipulating S_{entry}, which can be manipulated by users through an appropriate sequence of web requests. We perform session state manipulation as follows.

First, we collect the set of entry session state vectors observed in the traces, which we refer to as the *reachable entry state set* (denoted by $\Pi_{reachable}$). This set represents all the possible entry session states for all the interactions. As each reachable entry state S_{entry} can be acquired by feeding the corresponding sequence of web requests to the application during the trace collection phase, in the testing phase we directly set its values of session variables based on selected samples for simplicity.

Second, we identify the set of *leading entry state set*, which are observed to lead to S_{issue} during the execution (denoted by $\Pi_{leading}(S_{issue})$). Then we generate the *testing entry state set* for S_{issue} (denoted by $\Pi_{testing}(S_{issue})$), so that every testing session state vector falls beyond $\Pi_{leading}(S_{issue})$ (i.e., $\forall S_{testing} \in \Pi_{testing}(S_{issue}), S_{testing} \in (\Pi_{reachable} - \Pi_{leading}(S_{issue})))$. Our intuition is that $S_{testing}$ will probably not lead to S_{issue}, so that the constraints over S_{issue}

are violated. It is important to note that whether the session constraints over S_{issue} have been violated needs to be further verified, which will be described in Scenario Verification.

Finally, we have to instantiate each testing session state vector into concrete values of session variables. We randomly pick a set of session variables, whose symbolized session state vector matches the testing one. We set the session state of the application and feed the original web request. For example, S_{issue} for the SQL signature generated from $Query5$ is always [session.userid = nonnull] [session.role = professor]. Thus, we identify two testing session state vectors: [session.userid = nonnull] [session.role = student] and [session.userid=null] [session.role=null] for testing the session state constraints associated with that SQL signature.

Query Parameter Manipulation

To violate the query parameter constraints, we need to manipulate the values of query parameters. However, there is no direct way to control SQL query parameters. We notice that the values of SQL query parameters are usually propagated from web request parameters. For example, as shown in Figure VI.5 the request parameter $_GET['course_id'] is propagated to the query parameter *id* in *Query9*. Thus, we can violate query parameter constraints by manipulating web request parameters. Similar to how we extract Query Parameter Equality constraint, we analyze the collected interaction samples to identify parameter propagation paths.

- Parameter Propagation Path: The value of a specific query parameter is always equal to a specific web request parameter.

For the query parameters that do not have propagation paths from web requests, we do not violate their associated constraints. For query parameters with propagation paths, we manipulate the corresponding web request parameters, while keeping the web request syntax and the entry session state as the same as the interaction sample.

To violate Constraint 2.1, we generate a random value that falls beyond the finite value domain of the bounded parameter. For example, we observe that the query parameter *user_id* in $Query4$ is bounded to 1. We can set the request parameter $_GET['userid'] sent to the *admin.php* page to fall beyond the value set.

To violate Constraint 2.2, we change the value of the web request parameter to be different from the session variable, whose value is intended to be the same as the query parameter. For

example, one constraint we infer for the SQL signature of *Query*2 in *user.php* page is: $_GET[‘userid’] is always equal to $_SESSION[‘userid’]. If the current value of $_SESSION[‘userid’] is 3, we change the value of $_GET[‘userid’] to be a different number.

To violate Constraint 2.3, we need to control two interactions. We first replay the preceding interaction, collect the SQL responses within this interaction, then change the web request parameter to fall beyond the set of values of the response variable, whose value is supposed to contain the query parameter. For example, as shown in Figure VI.5, if the values returned by *Query2* includes 3, 4 and 5, we set the value of the web request parameter *course_id* to fall beyond the set.

We notice that the single random value might not be sufficient for exposing the logic flaws due to the persistent information in the database. For example, when the testing web request tries to add a new user into the application, whose id already exists in the database, the INSERT query will not be triggered. On the other hand, when the testing web request tries to delete a user, whose id does not exist in the database, it will also fail to trigger the DELETE query. Thus, to increase the exposure of logic flaws, we generate two concrete values for each request parameter that we manipulate. One value is from the set of observed values in the traces; the other is randomly generated to fall beyond the observed set and have the same type (e.g., integer or string).

Scenario Verification

After we replay the manipulated interaction sample to the application, we need to verify whether the execution scenario exposes potential logic vulnerabilities within the application. We need to examine if the SQL query, whose SQL signature is under test, is issued by the application and confirm the intended constraints are actually violated through our manipulation. To do so, we collect the set of SQL queries within the replayed interactions and construct SQL signatures for each SQL query to match the one under test. If two SQL signatures match, we need to further verify the validity of the manipulation depending on the type of the constraints being violated. If it is a violation of query parameter constraints, we directly report a violating SQL emission instance, since the manipulated parameter values violate query parameter constraints.

If it is a violation of session state constraints, we have to check whether the constraints over the issuing states are actually violated when the query is issued because our manipulation

is performed over the entry session state and not directly over the issuing session state. For example, when we set the entry state to match the testing session state vector [session.userid = nonnull] [session.role = student] and send a web request with the request key GET-admin.php to the application, *Query4* is triggered while the intended constraints (i.e., the issuing session state vector should be [session.userid = nonnull] [session.role = professor]) are violated. A violating SQL emission instance is thus reported.

Finally, we manually analyze those reported instances and classify them into either real violations of intended constraints or false positives.

Implementation

We implement a prototype system called EXPELLER, as shown in Figure VII.6, for identifying logic vulnerabilities within PHP web applications. EXPELLER operates in two phases: constraint inference and vulnerability exploitation, which are illustrated in details as follows.

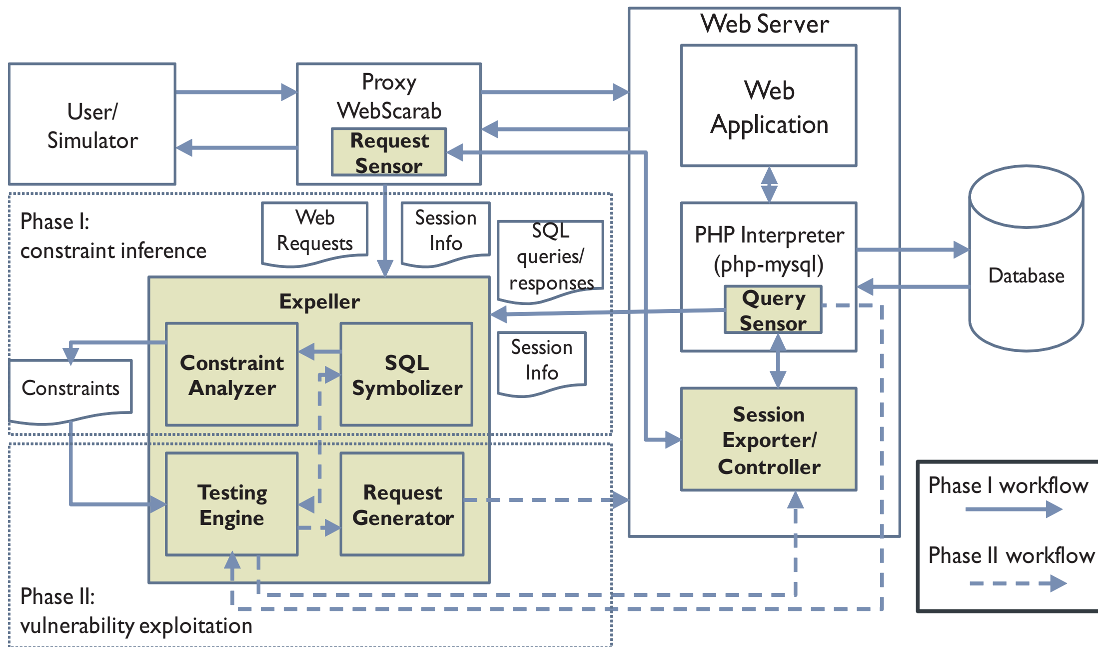


Figure VI.6: Prototype System Architecture (EXPELLER)

Phase I: Constraint Inference

In the constraint inference phase, EXPELLER reconstitutes interactions by collecting the external information during the normal executions of the web application, constructs SQL signatures and infers intended constraints. To collect session variables, EXPELLER has a *Session Exporter* module that is deployed on the web server, which can retrieve the set of session variables indexed by the session id. To collect web requests, we leverage and deploy an open source proxy WebScarab between the user and the application and extend it with a *Request Sensor* module, which forwards the captured web requests to EXPELLER. Similarly, to collect SQL queries, we embed the php-mysql module within the PHP interpreter with a *Query Sensor* module, which delivers the observed SQL queries to EXPELLER. To associate the SQL queries with the corresponding web requests and the issuing session states, both the *Request Sensor* module and the *Query Sensor* module communicate with the *Session Controller* module to retrieve the set of session variables at the moment when the web request is received and the moment when the SQL query is issued, and send the session information to EXPELLER.

After sufficient traces are collected, SQL queries are first fed into the *SQL Symbolizer* module for constructing SQL signatures. Then, all the traces, indexed by SQL signatures, are sent to the *Constraint Analyzer* module, where they are symbolized and the intended constraints are extracted using an invariant inference engine developed by us.

Phase II: Vulnerability Exploitation

In the vulnerability exploitation phase, EXPELLER is responsible for constructing exploit inputs based on inferred constraints, driving the web application with these inputs and reporting potential logic vulnerabilities.

The *Testing Engine* module takes the charge of the entire exploitation procedure. Recall that we employ two methods to construct exploit inputs based on the violation of session constraints and parameter constraints respectively. To violate session constraints, the *Testing Engine* manipulates the values of session variables through *Session Controller*, which is co-located with the *Session Exporter* at the web server. Then, the *Testing Engine* instructs the *Request Generator* module to replay the corresponding web request to the application. Similarly, to violate parameter constraints, the *Testing Engine* instructs *Request Generator* to reconstruct the web

requests with manipulated parameter values. During each interaction, the *Query Sensor* module deployed within the PHP interpreter collects the set of SQL queries issued by the application and sends them back to *Testing Engine* for scenario verification so that potential logic flaws can be identified.

It is worth noting that our approach is independent of the application source code. Among all the components of EXPELLER, only the *Query Sensor* and the *Session Exporter/Controller* have to be deployed at the web server and co-located with the web application. Other components can be deployed as web services, which are independent from the web application. This deployment feature allows EXPELLER to be easily customized for other platforms (e.g., JSP), as long as the *Query Sensor* and the *Session Exporter/Controller* are appropriately implemented and deployed.

Evaluation

We select a set of real world PHP web applications to evaluate our prototype system EXPELLER. We deploy all the web applications on a 2.13GHz Core 2 Linux server with 2GB RAM, running Ubuntu 10.10, Apache web server (version 2.2.16) and PHP (version 5.3.3). To facilitate constraint inference, we develop a user simulator for each application, which automates the procedure of operating the web application and emulates the interactions between users and the web application when users follow the navigation paths of the application. We refer the reader to Chapter III for more details of user simulators.

Experiment Results

In the constraint inference phase, EXPELLER first runs to collect traces. We run the user simulators long enough until the number of constraints stabilizes. Then, EXPELLER constructs SQL signatures and infers constraints from collected traces. Table VII.2 shows the statistics of constraint inference for each application, including the number of PHP files, collected SQL queries, web requests, SQL signatures, request keys, observed symbolic session state vectors, the constraints of each type, as well as the propagation paths identified from traces. In particular, we report the sum of session state vectors observed for different SQL signatures as the number of session state constraints (i.e., Constraint 1.1 & 1.2).

In the vulnerability exploitation phase, EXPELLER manipulates the selected interaction samples by violating each type of constraints. In particular, EXPELLER generates four types of exploit inputs: (1) Test1: violating session state constraints; (2) Test2: violating Constraint 2.1 (Bounded Query Parameter); (3) Test3: violating Constraint 2.2 (Query Parameter Equality); (4) Test4: violating Constraint 2.3 (Query Parameter Membership). EXPELLER feeds the exploit inputs into the application one by one and flags those which trigger a matching SQL query (i.e., indicating the existence of a potential logic flaw). Then, all the flagged exploits will be manually analyzed and categorized into either real violations or false positives.

Table VI.2 shows the summary of testing results, including the number of tests of each type, flagged exploits, false positives, as well as vulnerable files and logic flaws identified from each application. Note that the number of vulnerable files is computed by grouping violating SQL signatures based on their script file names and the number of logic flaws is determined through manual analysis of the source of those violations. One logic flaw may result in a number of vulnerable files and SQL query violations. This is because one faulty security check function in one source file may be included in a number of places. As a result, the vulnerability will manifest through multiple SQL query violations within different vulnerable files. EXPELLER's capability in locating the vulnerable files can greatly help developers pinpoint and fix the vulnerabilities within the web application.

In summary, EXPELLER identifies 9 logic flaws and 2 of them (marked in parenthesis) are previously unknown from published sources. In what follows, we explain the details of the exploits and logic flaws we identify for each application.

Table VI.1: Summary of Constraint Inference (Evaluation of EXPELLER)

Application	PHP files	SQL queries	SQL Sig.	Web req.	Req. keys	Session vectors	Const 1.1 & 1.2	Const 2.1	Const 2.2	Const 2.3	Prop. paths
Scarf	21	26378	78	1746	20	3	125	32	7	17	38
Wackopicko	52	7547	53	1822	28	2	63	3	26	39	18
EventsLister	37	1151	22	1202	24	2	23	5	1	0	28
Bloggit	24	9029	37	2999	18	2	45	4	0	15	34
minibloggie	11	1342	13	566	8	2	16	0	6	8	7

Scarf. It is intended that only the admin user can perform certain sensitive operations, including modifying the application configuration and all the users' information through the

Table VI.2: Summary of Testing Results (Evaluation of EXPELLER)

Application	Test1	Flag	FP	Test2	Flag	FP	Test3	Flag	FP	Test4	Flag	FP	Vuln. file	Logic flaw
Scarf	142	50	35*	0	0	0	0	0	0	6	6	1	1	1
Wackopicko	49	0	0	1	1	0	2	2	0	4	4	1	3	3
EventsLister	26	18	0	2	2	0	0	0	0	0	0	0	8	3 (2)
Bloggit	30	24	0	0	0	0	0	0	0	14	14	0	12	1
minibloggie	15	1	0	0	0	0	0	0	0	2	0	0	1	1
Summary	262	93	35*	3	3	0	2	2	0	26	24	2	25	9

generaloptions.php page. EXPELLER identifies the known authentication bypass vulnerability within the *generaloptions.php* page, where the *require_admin()* function for checking the constraint: $_SESSION['privilege'] == "admin" is missing. This vulnerability allows attackers to issue all of the sensitive SQL queries towards the database through crafted web requests.

We notice that EXPELLER introduces a number of false positives (marked with *) in Test1, which we will explain in detail in Section VII. EXPELLER also introduces one false positive in Test4, where it takes the observation that the parameter *paper_order* comes from the value set of *userid* from the previous interaction as a membership constraint by mistake. This false positive is introduced because the *paper_order* parameter and the *userid* parameter share the same value domain which is populated with natural numbers (e.g.,1,2,...), and our trace collection phase does not supply enough samples to differentiate these two parameters.

Wackopicko. EXPELLER identifies three logic flaws within Wackopicko. The first one exists in the *users/sample.php* page, where the *userid* parameter is fixed to be 1 as part of a static link. However, this page fails to check this constraint (2.1) before the SQL query is issued to retrieve the user information. This allows attackers to view any users' information by manipulating the *userid* parameter. The second flaw resides in the *users/view.php* page, which expects the *userid* parameter to be the same as the current user id (i.e., $_SESSION['userid']). However, this page fails to check this constraint (2.2) and allows the attacker to access unauthorized user information. The third flaw occurs in the *highquality.php page*, which retrieves information for pictures that are purchased by the user. This constraint is implied through the navigation paths, which requires the user to visit the *purchased.php* page first to retrieve the set of purchased pictures. However, the *highquality.php* page does not check this constraint (2.3) when issuing SQL queries to the database, which allows the attacker to manipulate the *pic_id* parameter sent to the *highquality.php* page to view the pictures that are not purchased. EXPELLER introduces one

false positive in Test4. In *add_comment.php* file, EXPELLER infers one parameter propagation path from the POST parameter *picid* to the query parameter *comments.picture_id*. Despite of such an equality relationship, the value of *comments.picture_id* does not come from the POST parameter *picid*. It is actually directly retrieved from the database, and thus cannot be tampered with by changing the request parameter.

EventsLister. EXPELLER identifies three logic flaws within the application. The first is a known vulnerability (CVE-2009-3168). We refer the reader to Chapter V for vulnerability details. The second one is an Execution After Redirection vulnerability. Although the *checkUser()* function is included in a number of script files before issuing SQL queries, it fails to stop the application execution after redirecting the users to the login page. To the best of our knowledge, this is a previously unknown vulnerability, which allows attackers to perform all administrative functions, including adding, deleting events and users and more. The third one is also a unknown vulnerability, which allows attackers to tamper with the parameter "ampm" to carry an arbitrary value and store the value in the database. This parameter is intended to only take either string "am" or "pm". However, the *update.php* page fails to check this Bounded Query Parameter constraint before issuing the query to the database. This vulnerability can serve as a launch pad for further exploits, since the tampered value will be returned within the web responses later.

Bloggit. EXPELLER identifies the known Execution After Redirection vulnerability (CVE-2006-7014) successfully. We refer the reader to Chapter V for vulnerability details.

Minibloggie is a simple blog application, which only allows the admin user to add/edit/delete posts. It has a known logic vulnerability (CVE-2008-6650), which allows the attacker to delete any posts without logging in, since the *del.php* page misses the *verifyUser()* function for security checking.

EXPELLER identifies the above vulnerability, by crafting a POST web request with an arbitrary post_id parameter to trigger the deletion of any posts. EXPELLER generates two test cases in Test4 for trying to edit the posts that are not authored by the current user. Minibloggie successfully identifies such malicious inputs by checking the authorship of posts before issuing UPDATE queries.

Discussion

Our approach makes two assumptions: 1) the intended application behavior is fully reflected when users follow the navigation paths; and 2) SQL queries are all sensitive operations. In practice, these two assumptions may not always hold. As a result, the inferred constraints may not accurately reflect the intended security requirements of the application, causing some ambiguous results.

EXPELLER introduces 10 false positives in Test1 in the Scarf experiment, which are caused by malformed navigation within the application. For example, the *comment.php* page is not accessible by a guest user through navigation links. Thus, several test cases are generated for guest users to forcefully browse this page and trigger queries within, which is actually allowed by the application. It is arguable (and subjective) whether we should report this as a real vulnerability or false positive. Similar cases occur within *showsessions.php* page and *register.php* page.

The other 25 false positives, introduced by EXPELLER in Test1 in the Scarf experiment, are caused by header file inclusion. The *header.php* and *functions.php* files, included in a number of files, perform certain basic operations through SELECT queries, such as retrieving the conference name. These queries are regarded as distinct sensitive operations when they are associated with different files via SQL signature construction. However, the information these queries retrieve from the database is non-sensitive and even accessible to a guest user. Thus, we classify them as false positives. To determine what information is sensitive requires involving human knowledge.

CHAPTER VII

AUTOMATED BLACK-BOX DETECTION OF ACCESS CONTROL VULNERABILITIES IN WEB APPLICATIONS

In this chapter, we present an automated black-box technique for the detection of access control vulnerabilities within web applications. We introduce a virtual SQL query model to accurately represent the database access operations that a web application can perform. Based on this model, complex data relationship (i.e., first-order and second-order) between users and accessed data entities are formulated and considered in the inference of access control policies. This allows our approach to cover a broader range of vulnerabilities compared with existing approaches. Our technique does not require application source code and server-side session information and thus is independent of the application development languages and platforms. We implement a prototype system BATMAN and evaluate it over a set of PHP and JSP web applications. The experiment results demonstrate that our approach is effective, accurate and applicable for applications developed in different languages. Our technique also provides detailed evidences to facilitate the manual analysis and the fix of the identified flaws.

The rest of the chapter is organized as follows. We first give an overview of our approach. Then we present our database access model and an illustrative example. The details of our approach and implementation are described in the following two sections. Finally, we conclude this chapter.

Overview

Access control vulnerabilities stem from the discrepancies between the intended access control policy and the policy that is actually implemented within a web application. Observing the fact that an application implementation usually comes without an explicit specification of the access control policy, deriving the intended access control policy becomes the first critical step in identifying access control vulnerabilities. This is a very challenging task especially for database-backed web applications. First, access control policies are implemented jointly through the proper definition of database access operations (e.g., SQL queries) and the data processing and filtering functions within web applications. The policy enforcement may span multiple program blocks,

files or web interactions. Second, complex data relationship within the databases may complicate the way how access control policies are manifested.

In this chapter, we present a black-box technique, which can accurately identify a broad range of access control vulnerabilities, especially arising from complex data relationship. We present a virtual SQL query model, which captures both the database access action (i.e., SQL queries issued to the database) and the database access result (i.e., the actual information that is presented to the user through web responses). We model the access control policy at two levels – at the role level, as the mapping between roles and the virtual SQL queries; at the user level, as the constraints over virtual SQL query parameters which characterize the relationship between users and the data entities being accessed. Based on the observation that the access control policy is usually correctly implemented under normal user navigation, we employ a crawler for automatically exploring the application, observe its interactions with users and the database and collect the execution traces for inferring the intended access control policy. Based on the inferred policy, we generate test inputs to exploit the application for potential access control flaws.

Problem Description

Database Access Model

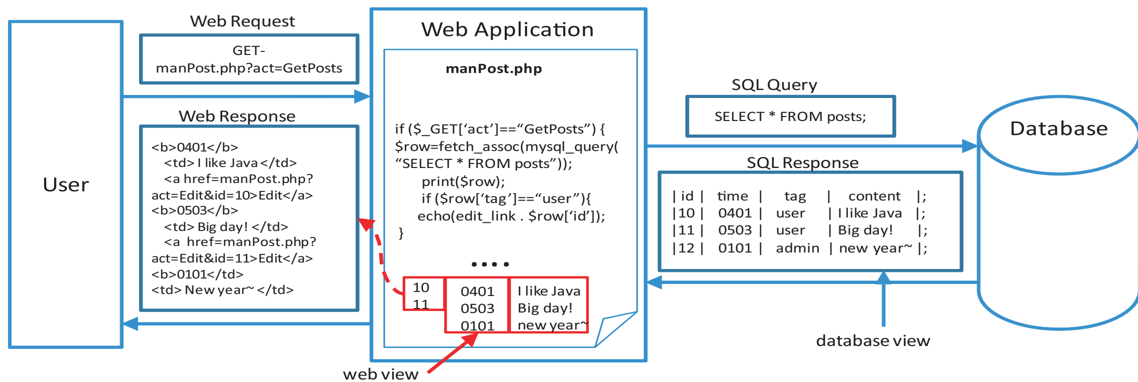


Figure VII.1: Example of Database Access via a Web Application

Figure VII.1 illustrates a typical scenario where a user accesses the information stored at the back-end database through the front-end web application. In this scenario, a user's data access requests are carried in the web requests and processed by the web application. The application first validates the user identity, then grants the user access privilege based on his role by issuing the database access requests on the user's behalf. Based on the results received from the database, the application composes the web responses, which return the information to the user.

In this chapter, we focus on relational database due to its popular deployment. In a web application, a user's privilege to access a relational database is granted through the issuance of SQL query, which specifies the operation and the set of data which the operation can be applied to. In the relational data model, data is represented as a set of n-ary relations, where each n-ary relation is an ordered set of attribute values. Visually, the basic data block can be represented as a table, which we refer to as *database view* in the chapter. There are two types of operations that can be applied:

MODIFY operation, which is performed through SQL queries, including INSERT, UPDATE and DELETE statements. When a user performs a MODIFY operation, the operation immediately takes effect over the database view that is specified in the SQL query.

READ operation, which is achieved through SELECT statement. When a user performs a READ operation, the SQL response is first processed at the web application which filters and embeds the retrieved database view into the web response in the form of HTML. As a result, the actual data observed by the user, which we refer to as *web view*, could be different from the database view, since the application may only allow part of the information within the SQL response to flow into the web response. As shown in Figure VII.1, the returned SQL response (i.e., the database view) is structured as a table. Only selected relations of the table are used by the application for composing the web response (i.e., the web view). In this case, the database view specified in the original SQL query is not sufficient to model the user's data access privilege. The post-processing by the web application needs to be taken into account.

To establish a unified model to express the data access privilege for the above two operations, we introduce *virtual SQL query*. For a MODIFY operation, the virtual SQL query is the same as the original SQL query, as it directly captures the data access privilege. For a READ operation, the virtual SQL query supplements the original SQL query with additional filters to capture the

post processing of the database view. The filters are constructed along two dimensions: *column-based filter*, which selects the attributes from the database view, and *row-based filter*, which selects the tuples from the database view. As shown in Figure VII.1, the application makes use of the *time*, *id* and *content* attributes to construct the web response, which we refer to as the *column-based filter*. Then two of the three rows of *id* values are used for composing the editing links, which we refer to as the *row-based filter*. The column-based filter can be represented by limiting the attributes in the SELECT statement (e.g., SELECT id FROM post) and the row-based filter can be represented by enhancing the SELECT statement of the column-based filter with WHERE clauses (e.g., SELECT id FROM post WHERE tag = 'user').

Formally, we represent a *virtual SQL query* using a skeleton structure and a set of query parameters. The skeleton structure is programmed in the source code, while the values of the parameters are dynamically fed by the application at runtime. For MODIFY operation, both the skeleton structure and the set of query parameters are derived from its original SQL query. For example, a virtual SQL query can be represented as: DELETE FROM user WHERE id = [p1], where p1 represents the value of the query parameter. For READ operation, the skeleton structure is composed of the skeleton structure of the original SQL query and the skeleton structures of the SQL queries that represent the post-processing filters. The set of parameters that are associated with both original SQL queries and the post-processing filter SQL queries collectively constitute the set of parameters for the virtual SQL query. Now we formalize our access control model as follows.

Database Access Operation: A database access operation represents the action performed by the web application on a behalf of users for retrieving or modifying data in the database. We model the database access operation through virtual SQL query. We use $o \in O$ to denote the skeleton of the virtual SQL query and $\Phi(o)$ to denote the set of parameters associated with o.

Role: A role $r \in R$ represents a distinctive set of privileges. We assume the set of roles form a lattice ordering relationship. A less privileged role should not be able to access the data which are only allowed for more privileged roles.

User: A user $u \in U$ represents a concrete principal with one specific role (denoted by $r(u)$) that interacts with the web application. Each user who is registered with the application is identified through an entity defined in the database (e.g., a tuple in the *user* table). When a user u triggers an operation, the parameters passed into the operation are linked to the user's

identity in the database, which are represented as user-based constraints. In this way, the access privileges of users under the same role can be differentiated.

Access Control (AC): The intended access control policy in a web application specifies the set of privileges for each user. It consists of two levels: role level and user level. The role-level access control policy is abstracted as the mapping from roles to the set of virtual SQL query skeletons $P : R \rightarrow 2^O$, where $P(r)$ represents the set of virtual SQL query skeletons that can be triggered under role r. The user-level access control policy attaches user-based constraints to each virtual SQL query skeleton that is accessible under $r(u)$.

Access Control Vulnerability

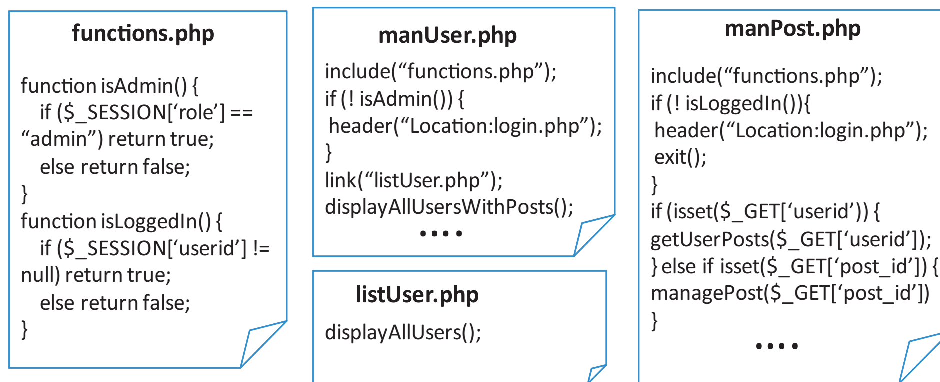


Figure VII.2: An Example Vulnerable Web Application

Access Control Vulnerability: A web application has an access control vulnerability if either of the following scenarios occurs: (1)at the role level, there exists a skeleton o, which is only allowed for role r, but can be triggered by a user with a less privileged role r'; (2)at the user level, there exists a skeleton o allowed for $r(u)$, which can be triggered by the user u while one of the constraints associated with o is violated.

Figure VII.2 shows an example application to demonstrate the access control flaws we focus on. It has three roles: guest user, regular user and admin user. The *manUser.php* and *listUser.php* pages are only intended for admin users, since the *manUser.php* page places *isAdmin()* function to check the user's role and the link to *listUser.php* page is only shown in *manUser.php* page. The *manPost.php* page is intended for regular users to manage their own posts. This example application is specially crafted to contain several access control vulnerabilities that correspond to

the above two scenarios: (1) An attacker, as a guest user, can trigger administrative operations within both *manUser.php* and *listUser.php* pages. The *listUser.php* page misses the *isAdmin()* function for checking the user's role, thus can be directly accessed without following the link. Although the check function within the *manUser.php* page redirects unauthorized users, it does not stop the application execution, so that the attacker's request is still processed. This vulnerability is also referred to as Execution After Redirection (EAR) [23]. (2) An attacker, as a regular user, can retrieve and modify any post created by other users. The *manPost.php* page fails to check whether the request parameter $_GET['userid'] really represents the current user when retrieving the posts for the user and whether the post with id $_GET['post_id'] belongs to the current user.

Approach

High-level Overview

Our approach consists of three major phases: *Trace Collection*, *Policy Inference* and *Vulnerability Detection*. To identify access control vulnerabilities in web applications, we first infer the intended access control policy by observing the application normal execution where users follow the navigation links (including submitting forms) provided by the web application.

In the *Trace Collection* phase, we leverage a web crawler to emulate the behaviors of different users when they follow the navigation links. During crawling, we collect the web interactions between the crawler and the application (i.e., HTTP requests and responses), as well as the interactions between the application and the database (i.e., SQL queries and responses), as the application's execution traces. Figure VII.3 shows an overview of the trace structure. Each SQL sample is retrieved from the traces by including a pair of SQL query and response, as well as the web request and response in the interaction. In the *Policy Inference* phase, the collected traces are grouped into different sample sets for inferring the intended access control policy at two levels: role level and user level. In the *Vulnerability Detection* phase, test inputs are constructed based on the inferred access control policy and fed into the application to generate a testing report of identified potential access control flaws. Finally, we manually analyze the testing report to

confirm real vulnerabilities and false positives. Our approach regards the web application as a black-box and does not require the application source code.

We elaborate *Policy Inference* and *Vulnerability Detection* in the next two sections and describe the implementation details of the crawler in SectionVII.

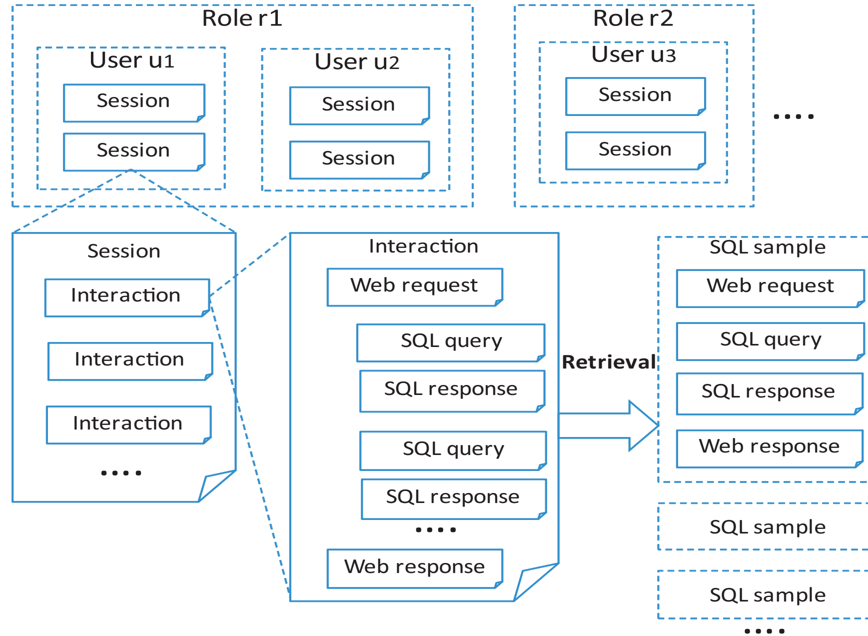


Figure VII.3: Trace Structure

Policy Inference

Role-level Policy Inference

At the role level, the access control policy is derived by identifying the set of virtual SQL query skeletons that are triggered by different roles (i.e., $P(r), r \in R$). Since this skeleton can be trivially constructed for the MODIFY operations from its original SQL queries, we focus on deriving the post-processing filters for constructing virtual SQL skeleton for READ operations. We first infer the column-based filters based on the database view presented in the SQL response and the web view embedded in the web response (i.e. HTML), then infer the row-based filters by leveraging the column-based filters.

Column-Based Filter Inference The key issue for inferring the column-based filter is to identify which attributes of the database view flow into the web response. There are two challenges here. First, database view is represented in a table structure, while the web response retains a tree-like DOM structure. Second, the columns in a database view can be represented through their attribute names, while this attribute information is usually removed when the data is embedded into the HTML page. Even worse, the data can be dispersed in the web page without a fixed representation. This structural and textual level mismatch prevents simple syntax-based filter construction. Thus we look for matching of the value domains between the data from database views and web responses. This is performed in three steps, as shown in Figure VII.4. First, we need to convert the data in HTML into a set of variables. We name these variables as web variables and denote them as $wv \in webResp$, where $webResp$ is a web response. In the second step, we match these web variables to the columns in the database view. Finally, we construct the column-based filter by choosing the columns that match web variables, which can be merged.

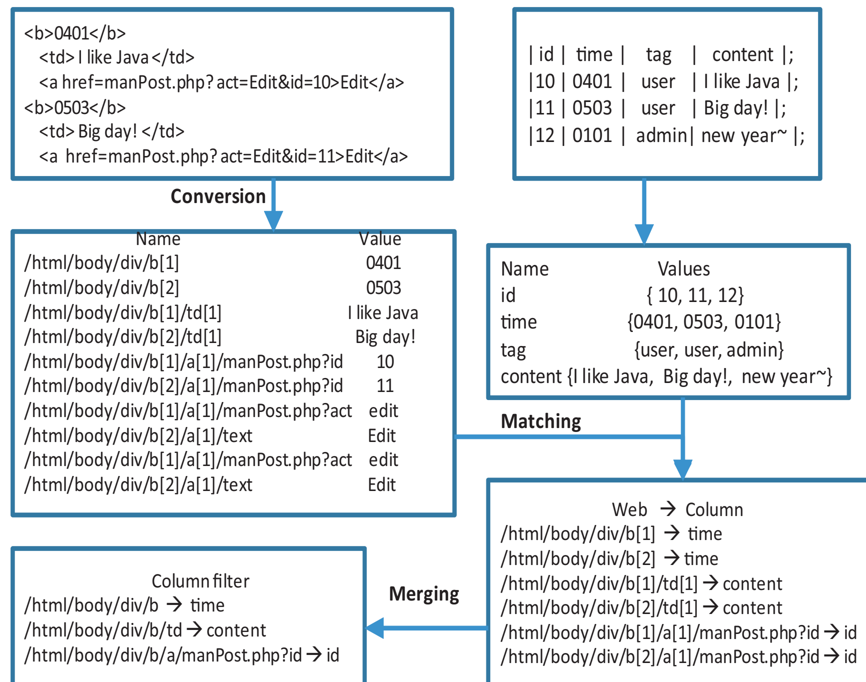


Figure VII.4: Column-based Filter Inference

HTML to variable conversion. To convert the data in HTML into a set of variables, we first identify the locations within web responses where dynamic information usually flows into (i.e. data sink). We observe that there are three types of data sinks: *text shown to the user*; *parameters in URL links*; and *form fields* (including select options, hidden fields, etc.). Each data sink is identified by the unique XPath leading to the sink from the DOM root. For the parameters in URL links, we also append the URL link and the parameter name. The unique XPath is constructed by appending the node along the general XPath (e.g., /html/body/div/b) with a sequence number (e.g., /html/body/div/b[1]).

Web-to-attribute matching. First, we group the SQL samples based on their original SQL query skeleton as well as their web request keys. The web request key is defined as the combination of HTTP method and the request URL (e.g., GET-manPost.php). The reason for grouping by web request key is that the post-processing is determined by the program block that is executed when the application composes the web response, while the web request key specifies the entry point of the execution. For convenience of presentation, we denote a column in the database view using a variable $sv \in sqlResp$, where $sqlResp$ is the database view carried in the SQL response. Let $V(v)$ represents the set of values of variable v. We identify a match between $sv \in sqlResp$ and $wv \in webResp$ if and only if $V(wv) \subseteq V(sv)$ holds for all $sqlResp$ and $webResp$ within the same sample group. To compare two values, we employ approximate string matching since the information from the database view can be manipulated by the application before it flows into the web response. We denote the *response propagation path* from sv to wv as $P_{resp} : sv \rightarrow wv$.

Web variable merging. We examine the web variables under the same general XPath. If they are matched with the same column, which means their values come from one single source, we recognize the attribute that flows into the web response as a column-based filter. The column filters in Figure VII.4 can be represented as: SELECT id, time, content FROM post.

Row-Based Filter Inference The row-based filter inference is performed in two steps. First, for each column selected from the column-based filter, we identify the rows whose values actually flow into the web response. Then, we observe the values of each attribute among the identified rows. If the values keep consistent for a certain attribute, we recognize it as one row-based filter and count the attribute as one operation parameter. In Figure VII.1, for the *time* and *content* attributes, all the rows flow into the web response and no attribute exists, whose values keep consistent. For the *id* attribute, the values of attribute *tag* keep consistent among the two rows

that flow into the web response. Thus, we identify the *tag* attribute as an additional operation parameter and its value is "user".

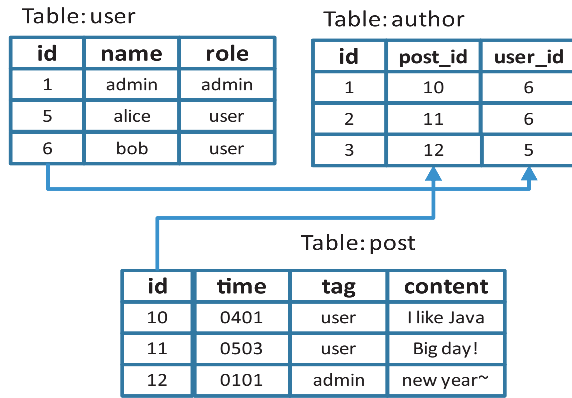
User-level Policy Inference

The role-level policy specifies the set of database access operations that can be performed by all the users with the same role, while the user-level policy restricts the specific view the user is authorized to access through constraints over the operation parameters. Note that the operation parameters include the parameters from the original SQL queries and the parameters from the post-processing filters (for READ operations).

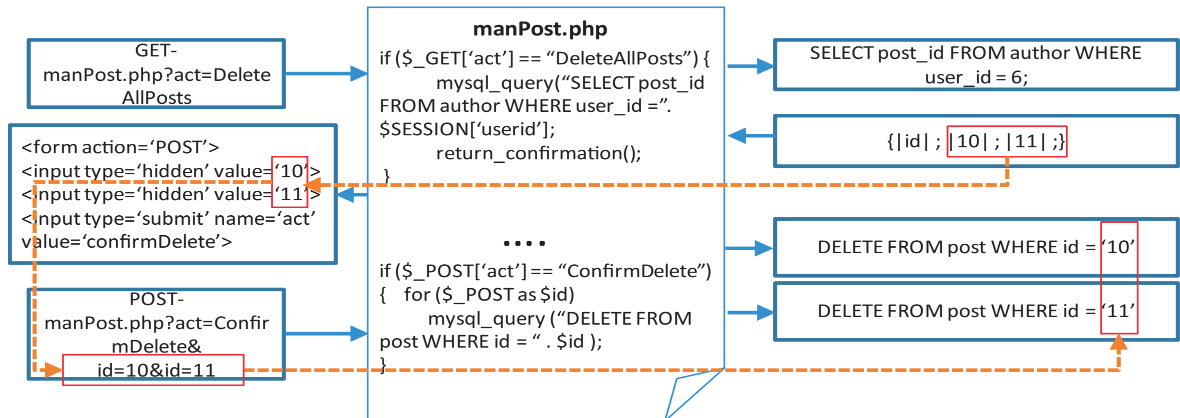
First-Order Constraint To differentiate the views among users, the operation parameters have to be linked with a user's identity, which reflect the relationship between the user and the view being accessed in the database. If the view being accessed is referenced by the user identity directly (e.g., through the primary key in the *user* table), we say there exists *First-Order Relationship* between the user and the view and the constraint on accessing this view is called *First-Order Constraint*. For example, if the application issues a SQL query: SELECT * FROM user WHERE id=5 to retrieve the information for the current user, the *id* parameter is subject to the first-order constraint.

We infer the first-order constraint in two steps. For each user u with role $r(u)$, let $o \in P(r(u))$ be a virtual SQL skeleton that is accessible based on the role-level access control policy and $\Phi(o)$ be the set of parameters associated with o. First, we identify the operation parameters $pv \subseteq \Phi(o)$ whose values are *consistent* for u. Then, from the consistent parameter set, we identify the parameters that are *unique* to each user of the same role by filtering out those whose values coincide across different users. In this way, the identified parameters are bounded to the user and directly linked to the user's identity.

Second-Order Constraint The operation parameters for accessing data may not always be linked with the user's identity directly. If the relationship between the data entity and the user's identity is reflected through foreign keys or a separate table, we say there exists *Second-Order Relationship* between the user and the data entity. As shown in Figure VII.5(1), the post entity is not directly linked with the user's identity, rather through the *author* table.



(1) Database Schema



(2) Second-Order Relationship Across Web Interactions

Figure VII.5: Second-Order Relationship Example

To compare with the first-order constraint which is imposed over the parameters of the virtual SQL query directly, the second-order constraint can be represented through a nested SQL query, e.g., DELETE FROM post WHERE id IN (SELECT post_id FROM author WHERE user_id = '6'). Here the parameter *id* does not directly link to the *user_id*, but is *bounded* to the database view that is subject to first-order constraint. If the second-order constraint is implemented in a nested SQL statement as the above example, it is reduced to a first-order constraint problem where the indirect relationship is automatically enforced.

However, in a web application the second-order constraint is usually implemented through two SQL statements within two web interactions, so that the users can be prompted with intermediate results. As shown in Figure VII.5(2), the application first retrieves all the posts created by the user, then the user can confirm the deletion. When this procedure is separated into more than one interactions, the user may violate this constraint by manipulating the web requests between interactions. As shown by dotted lines in Figure VII.5(2), this constraint actually reflects the value propagation chain across web interactions: the results returned by a previous operation are propagated through the web response, the web request parameters into a new operation. Since this constraint is manifested across interactions, in order to infer it we need to look back for previous web interactions. The number of web interactions we look back for depends on how long this value propagation chain lasts. We observe that this chain is usually no longer than three for most applications. Thus, we look back for two interactions in our inference.

We denote the operation in current interaction by o_{cur}, the operation from preceding interactions by o_{pre} and its result (i.e., SQL response) by $Re(o_{pre})$, the web (response) variables from preceding interactions by $webResp_{pre}$ and the web request parameters of current interaction by $webReq_{cur}$. We identify the second-order constraint if and only if the following rule holds for all the samples in the group:

Second-Order Relationship Rule: (1) o_{pre} is a READ operation; (2) $\exists v \in \Phi(o_{pre}), v$ is subject to first-order constraint, since the returned result set should be specific to each user; (3) there exists a response propagation path from the SQL response to the web response in a preceding interaction, i.e., $\exists P_{resp} : sv \to wv$, where $sv \in Re(o_{pre}), wv \in webResp_{pre}$; (4) there exists a parameter propagation path from the web request to the operation in the current interaction i.e., $\exists P_{para} : rv \to pv$, where $rv \in webReq_{cur}, pv \in \Phi(o_{cur})$; (5) $V(rv) \subseteq V(wv)$, which means

the request parameters come from the previous web response. The value propagation chain is $sv \to wv \to rv \to pv$.

Here the *parameter propagation path* $P_{para} : rv \to pv$, where $rv \in webReq_{cur}$, $pv \in \Phi(o_{cur})$ from web request parameters to virtual SQL query parameters can be inferred in a similar way as how we infer column-based filters. We group the SQL samples by both virtual SQL query skeleton and web request key, since each request URL has a set of predefined request parameters. Within each sample group, we compare the values of web request parameters ($webReq$) against those of operation parameters (i.e., $\Phi(o)$). A parameter propagation path exists between a web request and an operation if and only if the following rule holds for all the samples: $\exists rv \in webReq$ and $\exists pv \in \Phi(o)$ and $V(rv) == V(pv)$.

Vulnerability Detection

Test Input Generation

We generate test inputs based on the inferred policy to check whether the policy is actually enforced by the application. Each test input is designed to examine whether a specific database access operation can be performed while the policy is violated. Concretely, two types of test inputs are generated through *differential analysis*: (1) role-based test, which examines whether a less privileged role can trigger a database access operation, which is only allowed for a more privileged role; (2) user-based test, which examines whether a user can trigger a database access operation with a parameter that is subject to the user-based constraint but takes a value that is linked to another user's identity.

To realistically emulate the attack inputs, our test inputs trigger the desired database access operations through web requests, since end users (including attackers) cannot directly trigger the database access operations. This can be nontrivial, since a random web request has a very low chance of triggering the desired operation. To address this challenge, we leverage the samples that are used for policy inference for constructing test inputs. For each test input, we select a seed sample from each sample group and manipulate the contents of the sample.

Role-based Test. For each pair of roles (r_1, $r_2 \in R$, assuming r_1 is less privileged than r_2) from the role set, we compare their sets of allowed virtual SQL query skeleton to identify the set of privileged operations (denoted by O_{test}), which should not be allowed for the less privileged

role r_1 (i.e., $\forall o \in O_{test}, o \in P(r_2), o \notin P(r_1)$). To test each privileged operation, we first select a seed sample that is collected for users with role r_2, within which the operation under test has been triggered. Then, we change the user identity attached to the sample to be a user with role r_1, who should not be able to perform to this operation according to the inferred policy.

User-based Test. To test user-based constraints associated with the virtual SQL query skeleton, test inputs are constructed to violate such constraints and fed into the application to examine whether the operation can still be triggered. For test construction, we select a seed sample from each sample group and manipulate the value of the web request parameter that has a parameter propagation path leading to the operation parameter, which is subject to the constraint being tested. Specifically, to violate the first-order constraint, we replace the parameter value with a value that is observed for a different user with the same role. We note that first-order constraints, which do not have parameter propagation paths from web request will not be tested, since their parameters can not be manipulated from web requests. To violate the second-order constraints, test inputs need to contain a sequence of web requests. Such a test input is constructed in two steps. First, all the web requests, except the last one, from the sample are kept to trigger the READ operation that returns the set of data entities that are bound to the user. Next, we manipulate the parameter value of the last web request (i.e. test request) to fall outside of the above set.

Test Input Evaluation

Before a test input is fed into the application, if the user attached to the test input is required to log into the application, a login web request is first constructed with the user's login credentials and sent to the application to acquire the session cookie, so that subsequent web requests are recognized as the same user. If the test input is aimed at testing a second-order constraint, the sequence of web requests within the test input, except the test web request, are sent to the application. At the same time, all of the SQL queries and responses are collected. The READ operation that returns the set of data entities will be identified and the test request is manipulated based on its operation results (i.e. web response). Then, the test web request is sent to the application. After the web response is received, we check the sequence of SQL queries and responses collected during this interaction to see whether the database access operation under the test has been triggered. Especially, for a READ operation, we need to analyze whether the

information within the SQL response flows into the web response based on the filters. If the operation under test is identified, we flag the test input as a violation. After all the test inputs are evaluated, we manually analyze the reported violations and classify them as either a true or false positive.

Implementation

We implement a prototype system called BATMAN. Its architecture is shown in Figure VII.6. The crawler, web proxy and MySQL Proxy are adapted from open source projects and cooperate with the *Sync Portal* for collecting traces. The enhancements we make with the crawler are implemented in around 3000 lines of Java code. The *Inference Engine* and the *Testing Engine* are developed by ourselves in around 4600 lines of Java code for policy inference and vulnerability detection.

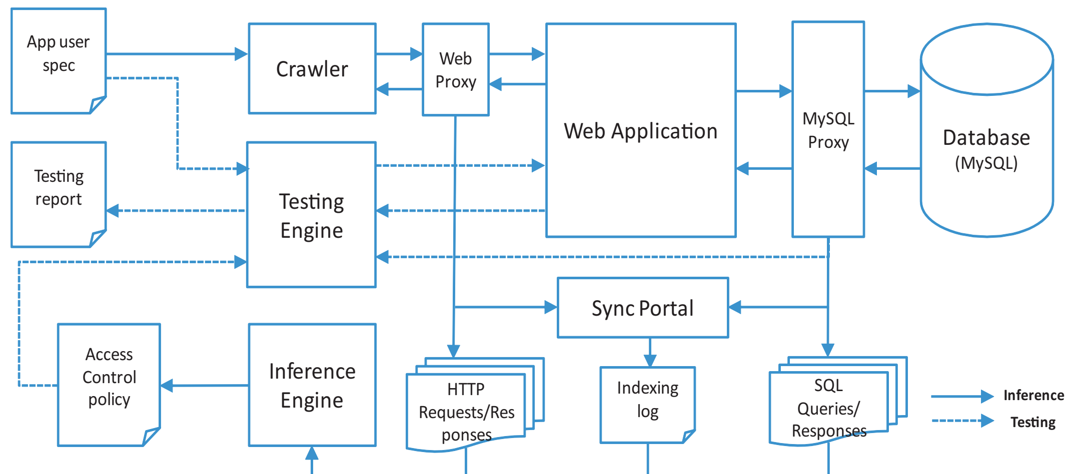


Figure VII.6: Prototype System Architecture (BATMAN)

Crawler

The crawler is built upon Crawljax [21], an open-source crawler designed for exploring modern web applications. Crawljax leverages the web application testing framework Selenium [59] for instantiating browser instances and rending web pages. Crawljax iterates all the clickables on a web page and progressively constructs a graph of web pages for exploring application. Utilizing a

crawler for trace collection allows for efficient exploration of a web application with high coverage. However, Crawjax (and most other open source crawlers) has several limitations and can not be directly applied for trace collection. Thus, we enhance Crawljax with two important functions, which are elaborated below.

Type-enhanced Form Filling. During crawling, whenever Crawljax encounters a web form, it will generate random inputs for filling the form. Since Crawljax is agnostic of the semantics of input fields, the randomly filled form is very likely to be rejected by the application, which prevents it from crawling the portion of the application behind a form submission. For example, the user registration form usually checks the validity of user-provided email, password and other features.

To enhance Crawljax's capability of filling forms, we employ a two-round crawling strategy. In the first round, the crawler follows the identified links, fills random inputs and collects all the encountered input fields. We refer to this round as the *form input discovery* phase. We define a set of commonly seen input data types, such as a single-digit number, random string, email and phone number. and manually attach the data type information to the input fields. For some input fields with special requirements, e.g., password, we directly specify a concrete value. In the second round, when the crawler encounters a form, it will generate a corresponding value based on the attached data type or take the concrete value and feed into the application. This two-round strategy cannot guarantee the crawler successfully makes it through every form checking, but can greatly increase the crawling coverage.

Semantic-based Page Comparison. Crawljax employs a depth-first strategy for crawling and returns to the previous page when it encounters a web page that has been visited. The criteria for determining whether a web page has been visited is the exact DOM matching with any previously seen web page. Considering that the web application state is possibly changed during crawling (e.g., adding an item to the shopping cart), this stringent criteria can easily get the crawler stuck in a local loop (e.g., adding a different item every time), which leads to the crawler unnecessarily exploring deeper with worse efficiency.

To achieve both better coverage and efficiency, we relax this stringent criteria by modifying the underlying page representation and comparison within Crawljax. We observe that although some web pages are not identical to each other, their semantics can be the same, which means

that the crawling paths starting from these pages are exactly the same. Thus, we can merge these pages with same semantics to stop crawling further beyond. To identify web pages with the same semantics, we leverage the intuition, pointed out in [24], that the links and forms within a web page determine where users can go from the current page. Thus, we represent a web page using the set of distinct link and form structures. To be more specific, we extract all the links and forms from the page, remove all the parameters and obtain their structure. An example link structure is */html/body/div/a/view.php?user_id=[p1]*, where [p1] is the place token for the parameter value. The web form is represented in a similar manner by combing the action URL with all input fields. An example form structure is */html/body/div/form/adduser.php?username=[p1]&password =[p2] &submit=[p3]*. In this way, the crawler identifies web pages with the same set of link/form structures, collapses them together and does backtracking from the current point.

Trace Collection

To collect traces, we leverage an open source web proxy WebScarab [50] to intercept HTTP requests/responses exchanged between the user and the application, and MySQL Proxy [45] to intercept SQL queries/response exchanged between the application and the database. Since MySQL Proxy is unaware of the HTTP interactions, we need to correlate the SQL queries and responses with the HTTP interactions during which they are collected. To do so, we implement *Sync Portal*, which is a web service running on Tomcat and can receive web requests coming from the web proxy and the MySQL proxy. We modify the WebScarab proxy, so that every time it intercepts a web request/response, it generates a unique index and sends the index to The *Sync Portal* through a wrapping web request. An example web request is *http://127.0.0.1:8080/Portal? type=HTTP_REQUEST &index=10*. Similarly, the MySQL proxy generates unique indices for SQL queries and responses and send them to The *Sync Portal*. This is implemented through writing a Lua script executed by the MySQL Proxy. The *Sync Portal* writes received indexes into an indexing file for synchronizing the SQL traffic with web interactions. This loosely couple structure enables us to deploy the components on different machines.

Inference & Testing

As shown by solid lines in Figure VII.6, the *Inference Engine* takes the indexing file, collected web requests/responses, and SQL queries/responses together to generate the access control policy file. The dotted lines in Figure VII.6 show the testing workflow. The *Testing Engine* generates test inputs based on the policy file, constructs test web requests and sends them to the application. At the same time, the MySQL Proxy sends SQL queries and responses observed during each interaction to Testing Engine. The *Testing Engine* examines the web response and the sequence of SQL pairs received during the current interaction to identify if the data access operation under test has been executed. The application user specification is also utilized by The *Testing Engine* to construct login requests.

Evaluation

We select a set of open source web applications for evaluation, which include both PHP and JSP applications, to demonstrate the effectiveness of our approach across different platforms. We deploy all the applications on a 2.13GHz Core 2 Linux server with 2GB RAM, running Ubuntu 10.10, Apache web server (version 2.2.16), PHP (version 5.3.3) and Tomcat6 (version 6.0.28). Table VII.1 shows a summary of these applications with their programming language, the lines of executable code (LoC) and a brief description.

Table VII.1: Summary of Web Applications for Evaluation (Evaluation of BATMAN)

Application	Lang.	Files	LoC	Description
Scarf (2007-0227)	PHP	19	797	conference system
Wackopicko	PHP	52	952	photo-sharing
EventsLister v2.03	PHP	27	837	event board
Bloggit v1.0	PHP	24	1071	blogging
minibloggie v2.1.6	PHP	11	838	blogging
JsForum v0.1	JSP	22	2224	web forum
JspBlog v0.2	JSP	17	1365	blogging

We first run the web crawler to collect execution traces. To demonstrate that our crawler is able to cover most functionalities of the application, we show the percentage of the application's executable code that the crawler triggers (i.e., code coverage) in Table VII.2. The PHP code coverage is measured using Spike PHPCoverage [62] and the JSP code coverage is measured using Clover [16]. We also show the improvement of the code coverage achieved by our crawler

compared to a baseline tool, wget. As pointed out in [24], code coverage is not an accurate metric for measuring the completeness of exploration and the performance of a crawler. This is because applications contain code used for installation, debugging, error-handling, and is even “dead” code, which can never be reached by the crawler. We use code coverage as a metric and tool to assist the identification of the portion of the application which the crawler misses, and iteratively improve the crawling depth for better trace quality.

Table VII.2: Summary of Code Coverage (Evaluation of BATMAN)

Application	crawler coverage	wget coverage	crawler improvement
Scarf	64.74%	32.12%	101.56%
Wackopicko	64.02%	30.78%	107.99%
EventsLister	74.81%	42.53%	75.89%
Bloggit	81.87%	68.35%	19.78%
minibloggie	64.98%	42.12%	54.27%
JsForum	69.5%	37.9%	83.38%
JspBlog	58.7%	36.1%	62.60%

The *Inference Engine* runs over the collected traces and generates the intended access control policy. Table VII.3 shows a summary of the policy inference results. For each application, we show the number of roles, web sessions, web requests and SQL queries being collected, along with the virtual SQL query skeletons (denote by skeleton) and two types of user-based constraints being identified.

Table VII.3: Summary of Policy Inference (Evaluation of BATMAN)

Application	role	web session	web request	SQL query	skeleton	First-Order Constraint	Second-Order Constraint
Scarf	3	17	1931	14486	45	2	0
Wackopicko	2	7	4478	16074	26	14	1
EventsLister	2	9	2227	1857	12	0	0
Bloggit	2	13	416	934	22	0	0
minibloggie	2	9	466	655	12	10	9
JsForum	3	17	3666	29749	21	12	1
JspBlog	2	10	270	119	9	0	0

The *Testing Engine* generates test inputs based on the inferred policy and exploits the application. Table VII.4 shows a summary of the testing results. Three types of tests are evaluated: role-based test (denoted as R-Test), tests generated for first-order user-based constraint (denoted

as FU-Test) and tests for second-order user-based constraint (denoted as SU-Test). For each type of test, we show the number of generated test inputs, flagged inputs (denoted by Flagged) and false positives (denoted by FP). We also report the number of real violations (true positives) in total (denoted by TP-Sum) and the number of access control vulnerabilities confirmed by manual analysis (denoted by Vuln). Note that these two numbers can be different, because one access control flaw within one check function allows all the operations that are guarded by the check to be triggered by unauthorized user, resulting in a number of violations. In the following, we describe the details of access control vulnerabilities we identify from each web application.

Table VII.4: Summary of Testing Results (Evaluation of BATMAN)

Application	R-Test	Flagged	FP	FU-Test	Flagged	FP	SU-Test	Flagged	FP	TP-sum	Vuln
Scarf	64	12	0	0	0	0	0	0	0	12	1
Wackopicko	24	0	0	1	1	0	1	1	0	2	2
EventsLister	9	9	0	0	0	0	0	0	0	9	2
Bloggit	19	16	2	0	0	0	0	0	0	14	1
minibloggie	16	1	0	0	0	0	9	2	0	3	2
JsForum	45	12	0	3	3	0	1	0	0	15	6
JspBlog	8	8	0	0	0	0	0	0	0	8	8
Summary	185	58	2	4	4	0	11	3	0	63	22

Details of Vulnerabilities

Scarf has three roles: guest user, regular user and admin user. Only admin users are allowed to manage papers, sessions and registered users. We identify one known authentication bypass vulnerability (CVE-2006-5909) within the *generaloptions.php* page, which misses the *require_-admin()* function for authorization checks. This vulnerability allows an attacker, as a guest or regular user, to tamper the conference and user information, since the page.

Wackopicko We identify two access control flaws within the application. The first one exists in the *users/view.php* page, which fails to check the intended first-order constraint on the parameter *userid*. This allows an attacker to access any user's information by tampering this parameter. The second one exists in the *highquality.php* page, which shows the high-quality pictures that are purchased by the user. This second-order constraint is implemented across two interactions. In the first interaction, the user visits the *purchased.php* page to retrieve a list of thumbnails and ids

of the pictures being purchased. In the second interaction, the high-quality picture is requested via parameter *pic_id* whose value comes from the picture ids returned by the first interaction. The vulnerability within the *highquality.php* page fails to check this constraint on the parameter *pic_id*, which allows an attacker to view any high-quality pictures. The *highquality.php* page calls the same function to retrieve the picture information from the database as other pages, but presents the picture's high quality key value to the user. Our technique can accurately identify this sensitive operation by analyzing the post processing performed by the application.

EventsLister only allows admin users to add new events, update or delete existing events, and manage users. We identify two access control flaws within the application. The first known one exists in the *add_user.php* page, where the function for checking whether the current user is an admin (i.e., *checkUser()*) is missing. This allows an attacker to directly access this page and add new users. The second is an Execution After Redirection vulnerability within the *checkUser()* function. All the PHP files that include this function for authorization checks are vulnerable and allow an attacker to trigger the database access operations. Sun et al. [63] also study this application, but fail to identify the second vulnerability.

Bloggit only allows admin users to manage blogs and user information. We identify the known Execution After Redirection vulnerability (CVE-2006-7014) in the *session.inc.php* page. All of the files that include the *session.inc.php* file for authorization checks allow an attacker to trigger the database access operations even after being redirected. Our tool also generates two false positives. After investigation, we found out they result from spurious filtering rules where the values of two irrelevant variables coincide with each other, so that the same database access operation is identified as different ones for two roles.

Minibloggie only allows the admin user to manager posts and the regular user to edit or delete the posts created by himself. We identify the known access control flaw (CVE-2008-6650) within the *del.php* page, which lacks the *verifyUser()* function for checking the user's role. This allows an attacker, as a guest user, to delete any posts. We also identify another previously unknown access control flaw, which allows the attacker, as a regular user, to delete other users' posts by tampering the parameter *post_id*, since the *del.php* page misses checking the second-order constraint on the parameter. This shows our technique can capture the data relationship between users and data

entities across web interactions and identify access control flaws arising from it. RoleCast [61] also studies this application, but fail to identify the second vulnerability.

JsForum has three roles: guest user, regular user and admin user. Regular users can add threads and replies to the forum, while admin users can add new forum and moderate all threads and replies. We identify several access control flaws within this application. First, the application fails to check the current user's role. As a result, guest users can add new threads, replies and forums, and increase the view counters of certain threads without logging in. Second, the application fails to check the first-order constraint on the parameter within a hidden field *user* in the forms. This allows a regular user to add new threads or replies on behalf of other users by tampering this parameter. It is worth noting that second-order constraints are also identified for this application, where a user should only be able to edit his own replies. But testing results show that the application checks the relationship between the reply and the user. Thus no vulnerability is reported under the SU-test.

JspBlog only allows the admin user to manage the blogs and users. JspBlog implements the access control policy by hiding the links that lead to administrative pages from guest users. Vulnerabilities exist within administrative pages which fail to check the current user's role before any database access operations. Thus an attacker can forcefully access any administrative page and trigger those operations.

Discussion

We use dynamic analysis for policy inference and directed fuzzing for vulnerability detection. Here we discuss the intrinsic challenges and limits of these two techniques and how we address them in this work. First, dynamic analysis cannot guarantee the completeness of vulnerability discovery. Insufficient exploration of the web application may lead to false negatives. We enhance our crawler with two major functions, so that it can cover most functionalities of the application. During experiments, we measure the code coverage and use it as an assistance to determine the crawling depth so that we can make a sound tradeoff between the trace quality and the crawling time. Second, the accuracy of directed fuzzing is closely related with the database state. For example, when a test input of adding a new user is sent to the application, the test output may vary depending on whether the user already exists in the database. If it exists, the application

will reject the request and the INSERT query cannot be triggered even the application contains a vulnerability of missing check. If the user does not exist in the database, then the vulnerability will be discovered. We address this problem by ensure the consistency of database state between the inference phase and the testing phase. First we record the database state before crawling and leverage the collected traces for generating concrete test input values. Before testing, we restore the database state and feed test inputs that don't affect the database state before those which might change the state.

Comparison with LogicScope and EXPELLER

LogicScope only leverages web requests/responses for vulnerability analysis and does not examine the interactions between the web application and the database. Thus, it cannot handle EAR (e.g., the false negative in Bloggit), where malicious SQL queries are triggered towards the database and the effect of the database state tampering is not reflected within web responses. In addition, it can not handle the complex relationship between data entities either (e.g., the false negative in Wackopicko), which can only be revealed through SQL responses. EXPELLER examines SQL queries/responses, thus can address the above limitations of LogicScope. EXPELLER does not examine web responses, which may introduce false positives. Specifically, EXPELLER reports a potential logic flaw, whenever an unexpected SQL query is issued. However, the information returned by the SQL query may not be returned to the user, since the application may perform additional processing and filtering over the SQL response to remove the sensitive information. Further, both LogicScope and EXPELLER require session variables for analysis. To retrieve session variables from a different platform other than PHP, which our prototype systems have been customized for, new session inspector components have to be developed and plugged into the prototype systems. In contrast, BATMAN integrates the information from both web requests/responses and SQL requests/responses, which enables it to handle both EAR and complex data relationship within the database while a minimum number of false positives are incurred. BATMAN also eliminates the usage of session variables by utilizing the role information, which is usually fixed and known a priori for a web application. This feature makes the prototype implementation of BATMAN naturally applicable to different platforms. The evaluation results demonstrate both the effectiveness and the generality of BATMAN system.

CHAPTER VIII

CONCLUSIONS AND FUTURE WORK

In this chapter, we summarize the work presented in previous chapters and discuss the limitations of our work and future research directions.

Summary of Chapters

Chapter I introduced the background of web application security and described the research problem we addressed in this dissertation – logic attacks and vulnerabilities. We also gave an overview of both defensive and preventive approaches and how our approaches address the existing challenges through black-box analysis.

Chapter II reviewed existing works for defending against logic attacks. These works are categorized into three classes. We also present the research areas that are closely related to this dissertation, including software specification inference, database security and test input generation.

Chapter III and IV described two runtime detection systems BLOCK and SENTINEL, which can detect logic attacks launched against web applications. In particular, BLOCK detects malicious web requests and SENTINEL detects malicious SQL queries that target at back-end databases. BLOCK is the first source-code free solution that addresses the logic attacks. SENTINEL can cover more types of logic attacks than BLOCK due to its capability of handling persistent application state in the database.

Chapter V and VI described two testing systems LogicScope and EXPELLER, which can identify logic vulnerabilities within web applications. LogicScope, based on BLOCK, can handle logic vulnerabilities that are reflected through web responses, while EXPELLER, based on SENTINEL, can cover those that are reflected as malicious SQL queries but not through web responses.

Chapter VII described a black-box technique, BATMAN, which formulates the logic vulnerabilities as access control problems and detects access control flaws within web applications. BATMAN has two major advantages over LogicScope and EXPELLER in two aspects: accuracy

and scalability. First, BATMAN covers more logic vulnerabilities than LogicScope by observing the interactions between the application and the database and tends to be more accurate than EXPELLER by taking into account the information propagation from SQL responses to web responses. Second, BATMAN does not require session variables. This allows the prototype implementation to handle applications that are developed in different languages and platforms without any adaptation. The evaluations over open source web applications demonstrate both the effectiveness and the adaptability of our technique.

Limitations

Despite the aforementioned advantages, our techniques have two major limitations.

First, our techniques are based on dynamic analysis and thus confronted with the inherent challenge of addressing the completeness of the analysis. Insufficient exploration of the state space of a web application leads to inaccurate characterization of application logic, resulting in both false positives and false negatives. Although we leverage carefully crafted user simulators and the automated crawler to minimize the chances of insufficient exploration, we cannot reason the coverage of the state space of a web application and improve it automatically.

Formal modeling of the state space of a web application is critical for reasoning the coverage of our techniques. As we have elaborated before, the state of a web application is maintained using two mechanisms – session variables, which keep the intra-session states, and database objects, which keep the inter-session states. Trivial methods which characterize the application state directly using the values of these variables and data objects will lead to state explosion. On the other hand, the application state can be reflected through the execution paths. If we can enumerate all the possible execution paths during crawling, we can cover all of the reachable application states. However, the exploration of all the possible execution paths requires a full population of the database, which itself is a hard research problem [51]. Existing techniques need to analyze the application source code to summarize the constraints along different branches and prepare the database. Without accessing the application source code, we can only start with a partially populated database and progressively enrich the application state space with user inputs until it is "converged". The criteria to determine if the explored state space is converged can only be defined over the external manifestations (through web responses or SQL queries)

and evaluated after state inference. Thus, an iterative technique for state preparation and state inference is desirable during the crawling. This technique consists of the inference of the current application state, construction of user inputs that possibly introduce new application states and refinement of the current application state. Although this method may not achieve a “perfect” database population, it give a certain guarantee or indication of the application state space being explored. The capability of reasoning and exploring the application state space iteratively will help the testing procedure to uncover subtle logic vulnerabilities and reduce false positives.

Second, our techniques cannot handle the scenario where the application logic specification dynamically changes, such as dynamic access control policy. It is interesting to investigate if our techniques can be extended to handle system dynamics. The key to handling system dynamics is to identify the static component behind the dynamics. For example, to handle the scenario of dynamic access control, where new roles can be created and the privileges of a specific role can be modified, we need to identify all of the possible privileges, which are specified in the application implementation and usually unchanged. Instead of characterizing the application logic based on roles, we can describe it in terms of privileges. Then, we can perform the vulnerability analysis over the authorization of each distinct privilege.

Future Research Directions

Web applications have been evolving fast with new technologies emerging, which result in an ever-changing landscape for web application security with new challenges. We outline several future research directions and point out pioneering works as follows.

First, an increasing amount of web application code that embeds business logic is moving to the client side. Since the client-side JavaScript code is exposed, the attackers are able to gain more knowledge about the application logic, thus more likely to discover the logic vulnerabilities within the application. Some researchers have noticed and tried to address this problem. For example, NoTamper [6] studied the inconsistencies of security checks behind web forms between the client side and the server side. Guha et al. [33] detected malicious client behaviors based on the execution graphs extracted from client-side code. However, they only target one specific vulnerability and there is no general approach to handle both client side and server side vulnerabilities. Our techniques currently focus on the server side of the application. As a future

direction, it would be interesting to integrate our techniques with the above techniques to characterize the client-side application logic, which can help us to better understand the application logic as a whole and discover more subtle logic vulnerabilities that arise from the client-side code.

Second, web applications are increasingly built by integrating third-party web services through APIs (i.e., Application Program Interface), which makes the application business logic more complex. Logic vulnerabilities can arise from the integration procedure, where multiple parties get involved in the web interactions. For example, Wang et al. [70] discovered logic vulnerabilities within the checkout procedures of several popular e-commerce websites that rely on third party payment services (e.g, Paypal), which can be exploited by the attackers to shop for free. They also identified logic flaws within web-based single-sign-on services [69], which enable the attackers to impersonate the victims. To defend against this type of logic attacks requires the understanding of the logic specification under the multi-party interaction scenarios. Several recent works try to address this problem by either analyzing the SDK (i.e., Software Development Kit) source code (e.g., [3]) or observing the web interactions (e.g., [75]). Very similar to our techniques, InteGuard [75] performs security checks over a set of invariant relations among HTTP interactions to defeat logic attack at runtime and INDICATOR [74] employs hybrid analysis to infer the dependency constraints on parameters for web services, such as Twitter and Flickr, which can be utilized to verify the correctness of client applications. As a future direction, it would be interesting and promising to extend our techniques to secure the integration among multiple web applications.

BIBLIOGRAPHY

[1] Glenn Ammons, Rastislav Bodk, and James R. Larus. Mining specifications. In *Symposium on Principles of Programming Languages*, volume 37, pages 4–16, 2002.

[2] AT&T website breach. http://www.acunetix.com/blog/web-security-zone/articles/analysis-php-attack-apple-information-disclosure/.

[3] Guangdong Bai, Jike Lei, Guozhu Meng, Sai Sathyanarayan Venkatraman, Prateek Saxena, Jun Suny, Yang Liuz, and Jin Song Dong. AuthScan: Automatic Extraction of Web Authentication Protocols From Implementations. In *NDSS'13: Proceedings of the 20th Annual Network and Distributed System Security Symposium*, 2013.

[4] Davide Balzarotti, Marco Cova, Viktoria V. Felmetsger, and Giovanni Vigna. Multi-module vulnerability analysis of web-based applications. In *CCS '07: Proceedings of the 14th ACM conference on Computer and communications security*, pages 25–35, 2007.

[5] Jason Bau, Elie Bursztein, Divij Gupta, and John Mitchell. State of the art: Automated black-box web application vulnerability testing. *Oakland'10: Proceedings of the 31st IEEE Symposium on Security and Privacy*, pages 332–345, 2010.

[6] Prithvi Bisht, Timothy Hinrichs, Nazari Skrupsky, Radoslaw Bobrowicz, and V. N. Venkatakrishnan. Notamper: automatic blackbox detection of parameter tampering opportunities in web applications. In *CCS '10: Proceedings of the 17th ACM conference on Computer and communications security*, 2010.

[7] Prithvi Bisht, Timothy Hinrichs, Nazari Skrupsky, and V. N. Venkatakrishnan. Waptec: whitebox analysis of web applications for parameter tampering exploit construction. In *CCS'11: Proceedings of the 18th ACM conference on Computer and communications security*, pages 575–586, 2011.

[8] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. Exe: automatically generating inputs of death. In *CCS '06: Proceedings of the 13th ACM conference on Computer and communications security*, pages 322–335, 2006.

[9] Avik Chaudhuri and Jeffrey S. Foster. Symbolic security analysis of ruby-on-rails web applications. In *CCS '10: Proceedings of the 17th ACM conference on Computer and communications security*, 2010.

[10] You Chen and Bradley Malin. Detection of anomalous insiders in collaborative environments via relational analysis of access logs. In *CODASPY'11: Proceedings of the first ACM conference on Data and application security and privacy*, pages 63–74, 2011.

[11] Adam Chlipala. Static checking of dynamically-varying security policies in database-backed applications. In *OSDI'10: Proceedings of the 9th USENIX conference on Operating systems design and implementation*, 2010.

[12] Stephen Chong, Jed Liu, Andrew C. Myers, Xin Qi, K. Vikram, Lantian Zheng, and Xin Zheng. Secure web applications via automatic partitioning. In *SOSP '07: Proceedings of the 21st ACM SIGOPS symposium on Operating systems principles*, pages 31–44, 2007.

[13] Stephen Chong, K. Vikram, and Andrew C. Myers. Sif: Enforcing confidentiality and integrity in web applications. In *USENIX'07: Proceedings of the 16th conference on USENIX security symposium*, 2007.

[14] Christina Yip Chung, Michael Gertz, and Karl Levitt. DEMIDS: A Misuse Detection System for Database Systems. In *Proceedings of the Integrity and Internal Control in Information System*, pages 159–178, 1999.

[15] Citigroup credit card information leakage in 2011. http://www.wired.com/threatlevel/2011/06/citibank-hacked/.

[16] Clover. http://www.atlassian.com/software/clover/overview.

[17] Confused Deputy Problem. http://en.wikipedia.org/wiki/confused_deputy_problem.

[18] Connection Pooling. http://en.wikipedia.org/wiki/connection_pool.

[19] Brian J. Corcoran, Nikhil Swamy, and Michael Hicks. Cross-tier, label-based security enforcement for web applications. In *SIGMOD '09: Proceedings of the 35th SIGMOD international conference on Management of data*, pages 269–282, 2009.

[20] Marco Cova, Davide Balzarotti, Viktoria Felmetsger, and Giovanni Vigna. Swaddler: An Approach for the Anomaly-based Detection of State Violations in Web Applications. In *RAID'07: Proceedings of the 10th International Symposium on Recent Advances in Intrusion Detection*, pages 63–86, 2007.

[21] Crawljax. http://crawljax.com.

[22] Michael Dalton, Christos Kozyrakis, and Nickolai Zeldovich. Nemesis: preventing authentication & access control vulnerabilities in web applications. In *USENIX'09: Proceedings of the 18th conference on USENIX security symposium*, pages 267–282, 2009.

[23] Adam Doupé, Bryce Boe, Christopher Kruegel, and Giovanni Vigna. Fear the EAR: Discovering and Mitigating Execution After Redirect Vulnerabilities. In *CCS'11: Proceeding of the 18th ACM Conference on Computer and Communications Security*, 2011.

[24] Adam Doupé, Ludovico Cavedon, Christopher Kruegel, and Giovanni Vigna. Enemy of the state: a state-aware black-box web vulnerability scanner. In *Proceedings of the 21st USENIX conference on Security symposium*, Security'12, pages 26–26, 2012.

[25] Adam Doupé, Marco Cova, and Giovanni Vigna. Why Johnny Can't Pentest: An Analysis of Black-box Web Vulnerability Scanners. In *DIMVA'10: Proceedings of the 7th Conference on Detection of Intrusions and Malware and Vulnerability Assessment*, pages 111–131, 2010.

[26] Michael Emmi, Rupak Majumdar, and Koushik Sen. Dynamic test input generation for database applications. In *ISSTA '07: Proceedings of the 2007 international symposium on Software testing and analysis*, pages 151–162, 2007.

[27] Michael Ernst, Jake Cockrell, William Griswold, and David Notkin. Dynamically Discovering Likely Program Invariants to Support Program Evolution. *IEEE Transactions on Software Engineering*, 27:99–123, 2001.

[28] Extended Finite State Machine. http://en.wikipedia.org/wiki/extended_finite-state_machine.

[29] Viktoria Felmetsger, Ludovico Cavedon, Christopher Kruegel, and Giovanni Vigna. Toward Automated Detection of Logic Vulnerabilities in Web Applications. In *USENIX'10: Proceedings of the 19th USENIX Security Symposium*, 2010.

[30] Adrienne Felt, Matthew Finifter, Joel Weinberger, and David Wagner. Diesel: Applying Privilege Separation to Database Access. In *ASIACCS'11: Proceedings of 6th ACM Symposium on Information, Computer and Communications Security*, pages 416–422, 2011.

[31] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: directed automated random testing. In *PLDI'05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 213–223, 2005.

[32] Patrice Godefroid, Michael Y. Levin, and David A. Molnar. Automated whitebox fuzz testing. In *NDSS'08: Proceedings of the 16th Network and Distributed System Security Symposium*, 2008.

[33] Arjun Guha, Shriram Krishnamurthi, and Trevor Jim. Using static analysis for ajax intrusion detection. In *WWW'09: Proceedings of the 18th international conference on World Wide Web*, pages 561–570, 2009.

[34] William G.J. Halfond, Saswat Anand, and Alessandro Orso. Precise interface identification to improve testing and analysis of web applications. In *ISSTA '09: Proceedings of the eighteenth international symposium on Software testing and analysis*, pages 285–296, 2009.

[35] Kenneth L. Ingham, Anil Somayaji, John Burge, and Stephanie Forrest. Learning dfa representations of http for protecting web applications. *Computer Networks and Isdn Systems*, 51:1239–1255, 2007.

[36] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. Pixy: A static analysis tool for detecting web application vulnerabilities (short paper). In *Oakland'06: Proceedings of the 27th IEEE Symposium on Security and Privacy*, pages 258–263, 2006.

[37] Ashish Kamra, Evimaria Terzi, and Elisa Bertino. Detecting anomalous access patterns in relational databases. *The VLDB Journal*, 17:1063–1077, 2008.

[38] Chulyun Kim and Kyuseok Shim. TEXT: Automatic Template Extraction from Heterogeneous Web Pages. *IEEE Trans. Knowl. Data Eng.*, 23(4):612–626, 2011.

[39] Ted Kremenek, Paul Twohey, Godmar Back, Andrew Ng, and Dawson Engler. From uncertainty to belief: inferring the specification within. In *OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation*, pages 161–176, 2006.

[40] Akshay Krishnamurthy, Adrian Mettler, and David Wagner. Fine-grained privilege separation for web applications. In *WWW'10: Proceedings of the 19th international conference on World Wide Web*, pages 551–560, 2010.

[41] Christopher Kruegel and Giovanni Vigna. Anomaly detection of web-based attacks. In *CCS'03: Proceedings of the 10th ACM conference on Computer and communications security*, pages 251–261, 2003.

[42] Sin Yeung Lee, Wai Lup Low, and Pei Yuen Wong. Learning Fingerprints for a Database Intrusion Detection System. In *ESORICS'02: Proceedings of 7th European Symposium on Research in Computer Security*, pages 264–280, 2002.

[43] Davide Lorenzoli, Leonardo Mariani, and Mauro Pezzè. Automatic generation of software behavioral models. In *ICSE '08: Proceedings of the 30th international conference on Software engineering*, pages 501–510, 2008.

[44] Adrian Mettler, David Wagner, and Tyler Close. Joe-e: A security-oriented subset of java. In *NDSS'10: Proceedings of the 17th Annual Network and Distributed System Security Symposium*, pages 357–374, 2010.

[45] MySQL Proxy. http://dev.mysql.com/doc/refman/5.0/en/mysql-proxy.html.

[46] OpenInvoice 0.9 beta. http://sourceforge.net/projects/openinv/.

[47] OpenIT. http://sourceforge.net/projects/openit/.

[48] OsCommerce Inc. http://www.oscommerce.com/.

[49] OWASP Top Ten Project 2013 Report. https://www.owasp.org/index.php/Top_10_2013-Top_10.

[50] OWASP WebScarab Project. https://www.owasp.org/index.php/category:owasp_webscarab_project.

[51] Kai Pan, Xintao Wu, and Tao Xie. Database state generation via dynamic symbolic execution for coverage criteria. In *Proceedings of 4th International Workshop on Testing Database Systems (DBTest11)*, pages 4–9, June 2011.

[52] Kai Pan, Xintao Wu, and Tao Xie. Generating program inputs for database application testing. In *ASE'11: Proc. 26th IEEE/ACM International Conference on Automated Software Engineering*, 2011.

[53] Bryan Parno, Jonathan M. McCune, Dan Wendlandt, David G. Andersen, and Adrian Perrig. CLAMP: Practical prevention of large-scale data leaks. In *Oakland'09: Proceedings of the 30th IEEE Symposium on Security and Privacy*, 2009.

[54] Pongsin Poosankam Prateek Saxena, Steve Hanna and Dawn Song. 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*, 2010.

[55] Davi De Castro Reis, Reis Paulo, Alberto H.F. Laender, Paulo B. Golgher, and Altigran S. da Silva. Automatic web news extraction using tree edit distance. In *WWW '04: Proceedings of the 13th international conference on World Wide Web*, pages 502–511, 2004.

[56] Alex Roichman and Ehud Gudes. Fine-grained access control to web databases. In *SACMAT'07: Proceedings of the 12th ACM symposium on Access control models and technologies*, pages 31–40, 2007.

[57] Alex Roichman and Ehud Gudes. DIWeDa - Detecting Intrusions in Web Databases. In *Proceeedings of the 22nd annual IFIP WG 11.3 working conference on Data and Applications Security*, pages 313–329, 2008.

[58] Prateek Saxena, Devdatta Akhawe, Steve Hanna, Feng Mao, Stephen McCamant, and Dawn Song. A symbolic execution framework for javascript. In *Oakland'10: Proceedings of the 2010 IEEE Symposium on Security and Privacy*, pages 513–528, 2010.

[59] SeleniumHQ: Web Application Testing System. http://seleniumhq.org/.

[60] Koushik Sen, Darko Marinov, and Gul Agha. Cute: a concolic unit testing engine for c. In *ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 263–272, 2005.

[61] Sooel Son, Kathryn S. McKinley, and Vitaly Shmatikov. Rolecast: finding missing security checks when you do not know what checks are. In *OOPSLA '11: Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 1069–1084, 2011.

[62] Spike PHPCoverage. http://phpcoverage.sourceforge.net/.

[63] Fangqi Sun, Liang Xu, and Zhendong Su. Static detection of access control vulnerabilities in web applications. In *USENIX'11: Proceedings of the 20th USENIX Security Symposium*, 2011.

[64] Nikhil Swamy, Brian J. Corcoran, and Michael Hicks. Fable: A language for enforcing user-defined security policies. In *Oakland '08: Proceedings of the 29th IEEE Symposium on Security and Privacy*.

[65] Symantec internet security threat report 2009. http://www.symantec.com/business/threatreport/.

[66] Verizon 2010 Data Breach Investigations Report. http://www.verizonbusiness.com/resources/reports/rp_2010-data-breach-report_en_xg.pdf.

[67] K. Vikram, Abhishek Prateek, and Benjamin Livshits. Ripley: automatically securing web 2.0 applications through replicated execution. In *CCS '09: Proceedings of the 16th ACM conference on Computer and communications security*, pages 173–186, 2009.

[68] Wackopicko. https://github.com/adamdoupe/wackopicko.

[69] Rui Wang, Shuo Chen, and XiaoFeng Wang. Signing me onto your accounts through facebook and google: A traffic-guided security study of commercially deployed single-sign-on web services. In *Oakland'12: Proceedings of the 2012 IEEE Symposium on Security and Privacy*, pages 365–379, 2012.

[70] Rui Wang, Shuo Chen, XiaoFeng Wang, and Shaz Qadeer. How to shop for free online - security analysis of cashier-as-a-service based web stores. In *Oakland'11: Proceedings of the 32nd IEEE Symposium on Security and Privacy*, 2011.

[71] Gary Wassermann and Zhendong Su. Static detection of cross-site scripting vulnerabilities. In *ICSE'08: ACM/IEEE 30th International Conference on Software Engineering*, 2008.

[72] Web Application Security Statistics. http://projects.webappsec.org/w/page/13246989/Web ApplicationSecurityStatistics.

[73] WhiteHat Security. WhiteHat website security statistic report 2010.

[74] Qian Wu, Ling Wu, Guangtai Liang, Qianxiang Wang, Tao Xie, and Hong Mei. Inferring dependency constraints on parameters for web services. In *Proceedings of the 22nd international conference on World Wide Web*, WWW '13, pages 1421–1432, 2013.

[75] Luyi Xing, Yangyi Chen, XiaoFeng Wang, and Shuo Chen. InteGuard: Toward Automatic Protection of Third-Party Web Service Integrations. In *NDSS'13: Proceedings of the 20th Annual Network and Distributed System Security Symposium*, 2013.

[76] Jinlin Yang, David Evans, Deepali Bhardwaj, Thirumalesh Bhat, and Manuvir Das. Perracotta: mining temporal api rules from imperfect traces. In *ICSE '06: Proceedings of the 28th international conference on Software engineering*, pages 282–291, 2006.

[77] Alexander Yip, Xi Wang, Nickolai Zeldovich, and M. Frans Kaashoek. Improving application security with data flow assertions. In *SOSP'09: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 291–304, 2009.