ParisTech
INSTITUT DES SCIENCES ET TECHNOLOGIES
PARIS INSTITUTE OF TECHNOLOGY

TELECOM ParisTech

2012-ENST-024

EDITE - ED 130

Doctorat ParisTech

T H È S E

pour obtenir le grade de docteur délivré par

TELECOM ParisTech

Spécialité « Réseaux et Sécurité »

présentée et soutenue publiquement par

Theodoor SCHOLTE

le 11/5/2012

Securing Web Applications by Design

Directeur de thèse : **Prof. Engin KIRDA**

Jury

Thorsten HOLZ, Professeur, Ruhr-Universit at Bochum, Germany	Rapporteur
Martin JOHNS, Senior Researcher, SAP AG, Germany	Rapporteur
Davide BALZAROTTI, Professeur, Institut EURECOM, France	Examinateur
Angelos KEROMYTIS, Professeur, Columbia University, USA	Examinateur
Thorsten STRUFE, Professeur, Technische Universit at Darmstadt, Germany	Examinateur

TELECOM ParisTech
école de l'Institut Télécom - membre de ParisTech

THÈSE

Acknowledgements

This dissertation would not have been possible without the support of many people. First, I would like to thank my parents. They have thaught and are teaching me every day a lot. They have raised me with a good mixture of strictness and love. I believe that they play an important role in all the good things in my life.

I am very grateful to prof. Engin Kirda. It is through his lectures that I have become interested in security. He has been an extraordinary advisor, always available to discuss. After he moved to the United States, he continued to be the person *'next door'*, always available to help me out. Furthermore, I would like to thank prof. Davide Balzarotti and prof. William Robertson. Completing this dissertation would not have been possible without their continuous support.

Thanks to my good friends Jaap, Gerben, Roel, Inge, Luit, Ellen and others I probably forget to mention. Over the years, we have shared and discussed our experiences of the professional working life. More importantly, we had a lot of fun. Although we lived in different parts of Europe, we managed to keep in touch as good friends do. Thanks to my 'local' friends: Claude, Luc, Alessandro, Marco, Leyla, Simone, Gerald and Julia. You have lightened up the years of hard work with activities such as drinking or brewing beer, barbecuing, climbing and skiing. I would like to thank my friends in particular for the moral support as personal life has not always been easy.

I would like to thank my colleagues at SAP, in particular Anderson, Gabriel, Henrik, Sylvine, Jean-Christophe, Volkmar and Agnès. Thank you all for your support and creating a good working environment.

Thanks to the staff at EURECOM, in particular to Gwenäelle for helping me and always being there when needed.

Finally, thanks to prof. Davide Balzarotti, prof. Thorsten Holz, prof. Angelos Keromytis, prof. Thorsten Strufe and Martin Johns for agreeing to be reporters and examinators.

Contents

1 Introduction **1**
1.1 Motivation 2
1.2 Research Problems 6
1.3 Thesis Structure 7

2 Related Work **9**
2.1 Security Studies 9
2.1.1 Large Scale Vulnerability Analysis 9
2.1.2 Evolution of Software Vulnerabilities 10
2.2 Web Application Security Studies 15
2.3 Mitigating Web Application Vulnerabilities 16
2.3.1 Attack Prevention 16
2.3.2 Program Analysis 20
2.3.3 Black-Box Testing 22
2.3.4 Security by Construction 23

3 Overview of Web Applications and Vulnerabilities **25**
3.1 Web Applications 25
3.1.1 Web Browser 26
3.1.2 Web Server 27
3.1.3 Communication 28
3.1.4 Session Management 33
3.2 Web Vulnerabilities 33
3.2.1 Input Validation Vulnerabilities 33
3.2.2 Broken Authentication and Session Management . . . 39
3.2.3 Broken Access Control and Insecure Direct Object References 42
3.2.4 Cross-Site Request Forgery 43

4 The Evolution of Input Validation Vulnerabilities in Web Applications **45**
4.1 Methodology 45
4.1.1 Data Gathering 46

4.1.2 Vulnerability Classification 47
4.1.3 The Exploit Data Set 47
4.2 Analysis of the Vulnerabilities Trends 48
4.2.1 Attack Sophistication 49
4.2.2 Application Popularity 54
4.2.3 Application and Vulnerability Lifetime 56
4.3 Summary 60

5 Input Validation Mechanisms in Web Applications and Languages 63
5.1 Data Collection and Methodology 64
5.1.1 Vulnerability Reports 64
5.1.2 Attack Vectors 65
5.2 Analysis 65
5.2.1 Language Popularity and Reported Vulnerabilities . . 66
5.2.2 Language Choice and Input Validation 68
5.2.3 Typecasting as an Implicit Defense 70
5.2.4 Input Validation as an Explicit Defense 71
5.3 Discussion 72
5.4 Summary 73

6 Automated Prevention of Input Validation Vulnerabilities in Web Applications 77
6.1 Preventing input validation vulnerabilities 78
6.1.1 Output sanitization 78
6.1.2 Input validation 79
6.1.3 Discussion 80
6.2 IPAAS 80
6.2.1 Parameter Extraction 81
6.2.2 Parameter Analysis 81
6.2.3 Runtime Enforcement 83
6.2.4 Prototype Implementation 84
6.2.5 Discussion 85
6.3 Evaluation 86
6.3.1 Vulnerabilities 86
6.3.2 Automated Parameter Analysis 87
6.3.3 Static Analyzer 88
6.3.4 Impact 89
6.4 Summary 90

7 Conclusion and Future Work 91
7.1 Summary of Contributions 91
7.1.1 Evolution of Web Vulnerabilities 92
7.1.2 Input Validation as Defense Mechanism 92

7.1.3 Input Parameter Analysis System 93
7.2 Critical Assessment 94
7.3 Future Work 95

8 French Summary **97**
8.1 Résumé 97
8.2 Introduction 98
8.2.1 Problématiques de recherche 101
8.3 L'évolution des vulnérabilités de validation d'entrée dans les applications Web 103
8.3.1 Méthodologie 103
8.3.2 L'analyse des tendances vulnérabilités 106
8.3.3 Discussion 111
8.4 Les mécanismes pour valider l'entrée des données dans les applications Web et des Langues 112
8.4.1 Méthodologie 113
8.4.2 Analyse 114
8.4.3 Discussion 118
8.5 Prévention automatique des vulnérabilités de validation d'entrée dans les applications Web 118
8.5.1 Extraction des paramètres 119
8.5.2 Analyse des paramètres 120
8.5.3 Runtime Environment 121
8.6 Conclusion 122

A Web Application Frameworks **125**

Bibliography **129**

List of Figures

1.1 Number of web vulnerabilities over time. 2

3.1 Example URL. 29
3.2 Example of a HTTP request message. 30
3.3 Example of an HTTP response message. 31
3.4 HTTP response message with cookie. 32
3.5 HTTP request message with cookie. 32
3.6 Example SQL statement. 34
3.7 Cross-site scripting example: search.php 35
3.8 Directory traversal vulnerability. 37
3.9 HTTP Response vulnerability. 37
3.10 HTTP Parameter Pollution vulnerability. 38
3.11 Cross-site request forgery example. 43

4.1 Buffer overflow, cross-site scripting and SQL injection vulnerabilities over time. 48
4.2 Prerequisites for successful attacks (in percentages). 50
4.3 Exploit complexity over time. 50
4.4 Applications having XSS and SQLI Vulnerabilities over time. 53
4.5 The number of affected applications over time. 54
4.6 Vulnerable applications and their popularity over time. 55
4.7 Popularity of applications across the distribution of the number of vulnerability reports. 56
4.8 Reporting rate of Vulnerabilities 57
4.9 Time elapsed between software release and vulnerability disclosure in years. 59
4.10 Average duration of vulnerability disclosure in years over time. 59

5.1 Distributions of popularity, reported XSS vulnerabilities, and reported SQL injection vulnerabilities for several web programming languages. 66
5.2 Example HTTP request. 69
5.3 Data types corresponding to vulnerable input parameters. . . 74
5.4 Structured string corresponding to vulnerable input parameters. 75

6.1 HTML fragment output sanitization example. 78
6.2 The IPAAS architecture. 81

8.1 Nombre de vulnérabilités web au fil du temps. 98
8.2 Les Buffer overflow, cross-site scripting and SQL injection vulnérabilités au fil du temps. 106
8.3 La complexité des exploits au fil du temps. 107
8.4 Temps écoulé entre la version du logiciel et à la divulgation de vulnérabilité au cours des années. 111
8.5 La durée moyenne de divulgation des vulnérabilités dans les années au fil du temps. 112
8.6 Exemple de requête HTTP. 114
8.7 Les types de données correspondant à des paramètres d'entrée vulnérables. 117
8.8 L'architecture IPAAS. 119

List of Tables

1.1 Largest data breaches. 3

4.1 Foundational and non-foundational vulnerabilities in the ten most affected open source web applications. 58
4.2 The attack surface. 60

5.1 Framework support for various complex input validation types across different languages. 72

6.1 IPAAS types and their validators. 82
6.2 PHP applications used in our experiments. 85
6.3 Manually identified data types of vulnerable parameters in five large web applications. 86
6.4 Typing of vulnerable parameters in five large web applications before static analysis. 87
6.5 Results of analyzing the code. 88
6.6 Typing of vulnerable parameters in five large web applications after static analysis. 88
6.7 The number of prevented vulnerabilities in various large web applications. 89

8.1 Les plus grandes fuites de données. 99
8.2 Vulnérabilités fondamentales et non-fondamentales dans les dix les plus touchés ouverts applications Web source. 110
8.3 IPAAS types et leurs validateurs. 120

A.1 Web frameworks analyzed 127

Chapter 1

Introduction

Global Internet penetration started in the late 80's and early 90's when an increasing number of Research Institutions from all over the world started to interconnect with each other and the first commercial Internet Service Providers (ISPs) began to emerge. At that time, the Internet was primarily used to exchange messages and news between hosts. In 1990, the number of interconnected hosts had grown to more than 300.000 hosts. In the same year, Tim Berners-Lee and Robert Cailliau from CERN started the World Wide Web (WWW) project to allow scientists to share research data, news and documentation in a simple manner.

Berners-Lee developed all the tools necessary for a working WWW including an application protocol (HTTP), a language to create web pages (HTML), a Web browser to render and display web pages and a Web server to serve web pages. As the WWW provided the infrastructure for publishing and obtaining information via the Internet, it simplified the use of the Internet. Hence, the Internet started to become tremendously popular among normal non-technical users resulting in an increasing number of connected hosts.

Over the past decade, affordable, high-speed and 'always-on' Internet connections have become the standard. Due to the ongoing investments in local cell infrastructure, Internet can now be accessed from everywhere at any device. People access the Internet using desktops, notebooks, Tablet PCs and cell phones from home, office, bars, restaurants, airports and other places.

Along with investments in Internet infrastructure, companies started to offer new types of services that helped to make the Web a more compelling experience. These services would not have been realized without a technological evolution of the Web. The Web has evolved from simple static pages to very sophisticated web applications, whose content is dynamically generated depending on the user's input. Similar to static web pages, web applications can be accessed over a network such as the Internet or an Intranet

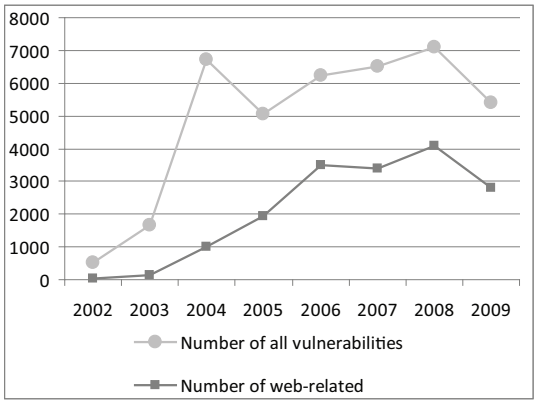


Figure 1.1: Number of web vulnerabilities over time, data obtained from NVD CVE [78].

using a Web browser and it generates content depending on the user's input. The ubiquity of Web browsers, the ability of updating and maintaining web applications without distributing and installing software on potentially thousands of computers and their cross-platform compatibility, are factors that contributed to the popularity of web applications.

The technological evolution of the Web has dramatically changed the type of services that are offered on the Web. New services such as social networking are introduced and traditional services such as e-mail and online banking have been replaced by offerings based on Web technology. Notably in this context is the emerging strategy of software vendors to replace their traditional server or desktop application offerings by sophisticated web applications. An example is SAP's Business ByDesign, an ERP solution offered by SAP as a web application.

The technological evolution has impacted the way how people nowadays use the Web. Today, people critically depend on the Web to perform transactions, to obtain information, to interact, have fun and to socialize via social networking sites such as Facebook and Myspace. Search engines such as Google and Bing allow people to search and obtain all kinds of information. The Web is also used for many different commercial purposes. These include purchasing airline tickets, do-it-yourself auctions via Ebay and electronic market places such Amazon.com.

1.1 Motivation

Over the years, the World Wide Web has attracted many malicious users and attacks against web applications have become prevalent. Recent data from SANS Institute estimates that up to 60% of Internet attacks target

Records	Date	Organizations
130.000.000	2009-01-20	Heartland Payment Systems, Tower Federal Credit Union, Beverly National Bank
94.000.000	2007-01-17	TJX Companies Inc.
90.000.000	1984-06-01	TRW, Sears Roebuck
77.000.000	2011-04-26	Sony Corporation
40.000.000	2005-06-19	CardSystems, Visa, MasterCard, American Express
40.000.000	2011-12-26	Tianya
35.000.000	2011-07-28	SK Communications, Nate, Cyworld
35.000.000	2011-11-10	Steam (Valve, Inc.)
32.000.000	2009-12-14	RockYou Inc.
26.500.000	2006-05-22	U.S. Department of Veterans Affairs

Table 1.1: Largest data breaches in terms of disclosed records according to [31].

web applications [23]. The insecure situation on the Web can be attributed to several factors.

First, the number of vulnerabilities in web applications has increased over the years. Figure 1.1 shows the number of all vulnerabilities compared to the number of web-related vulnerabilities that have been published between 2000 and 2009 in the Common Vulnerabilities and Exposures (CVE) List [78]. We observe that as of 2006, more than half of the reported vulnerabilities are web-related vulnerabilities. The situation has not improved in recent years. Based on an analysis of 3000 web sites in 2010, a web site contained on average 230 vulnerabilities according to a report from WhiteHat Security [105]. Although not all web vulnerabilities pose a security risk, many vulnerabilities are exploited by attackers to compromise the integrity, availability or confidentiality of a web application.

Second, attackers have a wide range of tools at their disposal to find web vulnerabilities and launch attacks against web applications. The advanced functionality of Google Search allows attackers to find security holes in the configuration and programming code of websites. This is also known as Google Hacking [12, 72]. Furthermore, there is a wide range of tools and frameworks available that allow attackers to launch attacks against web applications. Most notably in this context is the Metasploit framework [85]. This modular framework leverages on the world's largest database of quality assured exploits, including hundreds of remote exploits, auxiliary modules, and payloads.

Finally, attackers do have motivations to perform attacks against web applications. These attacks can result into, among other things, data leakage, impersonating innocent users and large-scale malware infections.

An increasing number of web applications store and process sensitive data such as user's credentials, account records and credit card numbers. Vulnerabilities in web applications may occur in the form of data breaches which allow attackers to collect this sensitive information. The attacker may use this information for identity theft or he can sell it on the underground market. Stealing large amounts of credentials and selling them on the underground market can be profitable for an attacker as shown by several studies [124, 13, 39, 104]. Security researchers estimate that stolen credit card numbers can be sold for a price ranging between $2 to $20 each [13, 39]. For bank accounts the price range per item is between $10 and $1000 while for e-mail passwords the range is $4 to $30 [39] per item.

Vulnerable web applications can also be used by attackers to perform malicious actions on the victim's behalf as part of phishing attack. In these types of attacks, attackers use social engineering techniques to acquire sensitive information such as user's credentials or credit card details and/or let the user perform some unwanted actions thereby masquerading itself as a trustworthy entity in the communication. Certain flaws in web applications such as cross-site scripting make it easier for an attacker to perform a successful phishing attack because in such attack, a user is directed to the bank or service's own web page where everything from the web address to the security certificates appears to be correct. The costs of phishing attacks are significant, RSA estimates that the losses of phishing attacks world wide in the first half year of 2011 amounted over more than 520 million Dollars [94].

Legitimate web applications that are vulnerable can be compromised by attackers to install malware on the victim's host as part of a *drive-by-download* [84]. The installed malware can take full control of the victim's machine and the attacker uses the malware to make financial profit. Typically, malware is used for purposes such as acting as a botnet node, harvesting sensitive information from the victim's machine, or performing other malicious actions that can be monetized. Web-based malware is actively traded on the underground market [124]. While no certain assessments exist on the total amount of money attackers earn with trading virtual assets such as malware on the underground market, some activities have been analyzed. A study performed by Mcafee [64] shows that compromised machines are sold as anonymous proxy servers on the underground market for a price ranging between $35 and $550 a month depending on the features of the proxy.

Attacks against web applications affect the availability, integrity and confidentiality of web applications and the data they process. Because our society heavily depends on web applications, attacks against web applications form a serious threat. The increasing number of web applications that process more and more sensitive data made the situation even worse. Table 1.1 reports on the largest incidents in terms of exposed data records in

the past years. While the Web is not the primary source of data breaches, it still accounts for 11 % of the data breaches which is the second place. Although no overall figures exist on the annual loss caused by data breaches on the Web, the costs of some data breaches have been estimated. In 2011, approximately 77 million users accounts on the Sony Playstation Network were compromised through a SQL injection attack on two of Sony's properties. Sony estimated that it would spent 171.1 million Dollars in dealing with the data breach [102].

To summarize, the current insecure state of the Web can be attributed to the prevalence of web vulnerabilities, the readily available tools for exploiting them and the (financial) motivations of attackers. Unfortunately, the growing popularity of the Web will make the situation even worse. It will motivate attackers more as attacks can potentially affect a larger number of innocent users resulting into more profit for the attackers. The situation needs to be improved because the consequences of attacks are dramatic in terms of financial losses and efforts required to repair the damage.

To improve the security on the Web, much effort has been spent in the past decade on making web applications more secure. Organizations such as MITRE [62], SANS Institute [23] and OWASP [79] have emphasized the importance of improving the security education and awareness among programmers, software customers, software managers and chief information officers. Also, the security research community has worked on tools and techniques to improve the security of web applications. These tools and techniques mainly focus on either reducing the number of vulnerabilities in applications or on preventing the exploitation of vulnerabilities.

Although a considerable amount of effort has been spent by many different stakeholders on making web applications more secure, we lack quantitative evidence whether this attention has improved the security of web applications. In this thesis, we study how web vulnerabilities have evolved in the past decade. We focus in this dissertation on SQL injection and cross-site scripting vulnerabilities as these classes of web application vulnerabilities have the same root cause: improper sanitization of user-supplied input that result from invalid assumptions made by the developer on the input of the application. Moreover, these classes of vulnerabilities are prevalent, well-known and have been well-studied in the past decade.

We observe that, despite security awareness programs and tools, web developers consistently fail to implement existing countermeasures which results into vulnerable web applications. Furthermore, the traditional approach of writing code and then testing for security does not seem to work well. Hence, we believe that there is a need for techniques that secure web applications *by design*. That is, techniques that make web applications automatically secure without relying on the web developer. Applying these techniques on a large scale should significantly improve the security situa-

tion on the web.

1.2 Research Problems

The previous sections illustrates that web applications are frequently targeted by attackers and therefore solutions are necessary that help to improve the security situation on the web. Understanding how common web vulnerabilities can be automatically prevented, is the main research challenge in this work.

In this thesis, we tackle the following research problems with regard to the security of web applications:

- *Do developers create more secure web applications today than they used to do in the past?*

  In the past decade, much effort has been spent by many different stake-holders on making web applications more secure. To date, there is no empirical evidence available whether this attention has improved the security of web applications. To gain deeper insights, we perform an automated analysis on a large number of cross-site scripting and SQL injection vulnerability reports. In particular, we are interested in finding out if developers are more aware of web security problems today than they used to be in the past.

- *Does the programming language used to develop a web application influence the exposure of those applications to vulnerabilities?*

  Programming languages often contain features which help programmers to prevent bugs or security-related vulnerabilities. These features include, among others, static type systems, restricted name spaces and modular programming. No evidence exists today whether certain features of web programming languages help in mitigating input validation vulnerabilities. In our work, we perform a quantitative analysis with the aim of understanding whether certain programming languages are intrinsically more robust against the exploitation of input validation vulnerabilities than others.

- *Is input validation an effective defense mechanism against common web vulnerabilities?*

  To design and implement secure web applications, a good understanding of vulnerabilities and attacks is a prerequisite. We study a large number of vulnerability reports and the source code repositories of a significant number of vulnerable web applications with the aim of gaining deeper insights into how common web vulnerabilities can be prevented. We will analyze if typing mechanisms in a language and

input validation functions in a web application framework can potentially prevent many web vulnerabilities. No empirical evidence is available today that show to which extend these mechanisms are able to prevent web vulnerabilities.

- *How can we help application developers, that are unaware of web application security issues, to write more secure web applications?*

  The results of our empirical studies suggest that many web application developers are unaware of security issues and that a significant number of web vulnerabilities can be prevented using simple straight-forward validation mechanisms. We present a system that learns that data types of input parameters when developers write web applications. This system is able to prevent many common web vulnerabilities by automatically augmenting otherwise insecure web development environments with robust input validators.

1.3 Thesis Structure

We start by giving an overview of the related work on vulnerability studies, program analysis techniques to find security vulnerabilities, client-side and server-side defense mechanisms and techniques to make web applications *secure by construction*.

Chapter 3 gives an overview of web vulnerabilities. First, we give an overview of several web technologies to support our discussion on web application security issues. Then, we present different classes of input validation vulnerabilities. We explain how they are introduced and how to prevent them using existing countermeasures.

In Chapter 4, we study the evolution of input validation vulnerabilities in web applications. First, we describe how we automatically collect and process vulnerability and exploit information. Then, we perform an analysis of vulnerability trends. We measure the complexity of attacks, the popularity of vulnerable web applications, the lifetime of vulnerabilities and we build time-lines of our measurements. The results of this study have been published in the International Conference on Financial Cryptography and Data Security 2011 [99] and the journal Elsevier Computers & Security [100].

In Chapter 5, we study the relationship between a particular programming language used to develop web applications and the vulnerabilities commonly reported. First, we describe how we link a vulnerability report to a programming language. Then, we describe how we measure the popularity of a programming language. Furthermore, we examine the source code of vulnerable web applications to determine the data types of vulnerable input. The results of this study have been published in the ACM Symposium on Applied Computing conference 2012 (ACM SAC 2012) [101].

Chapter 6 presents IPAAS, a completely automated system against web application attacks. The system automatically and transparently augments web application development environments with input validators that result in significant and tangible security improvements for real web applications. We describe the implementation of the IPAAS approach of transparently learning types for web application parameters, and automatically applying robust validators for these parameters at runtime. The results have been published in the IEEE Signature Conference on Computers, Software, and Applications 2012 (COMPSAC 2012).

In Chapter 7, we summarize and conclude this thesis. Also, we show future directions for research and provide some initial thoughts on these research directions.

Chapter 2

Related Work

In the past years, we have observed a growing interest on knowledge about the overall state of (web) application security. Furthermore, a lot of effort has been spent on techniques to improve the security of web applications. In this Chapter, we first discuss studies that analyzed general security trends and the life cycle of vulnerabilities in software. Then, we document studies that analyzed the relationship between the security of web applications and the features provided by web programming languages or frameworks. Finally, we give an overview of techniques that can detect or prevent vulnerabilities in web applications or can mitigate their impact by detecting or preventing attacks that target web applications.

Where applicable, we compare the related work with our work presented in this thesis.

2.1 Security Studies

In the past years, several studies have been conducted with the aim of getting a better understanding of the state of software security. In this section, we give an overview of these studies.

2.1.1 Large Scale Vulnerability Analysis

Security is often considered as an arms race between crackers who try to find and exploit flaws in applications and security professionals who try to prevent that. To better understand the security ecosystem, researchers study vulnerability, exploit and attack trends. These studies give us insights in the exposed risks of vulnerabilities to our economy and society. Furthermore, they improve the security education and awareness among programmers, managers and CIOs. Security researchers can use security studies to focus their research on a narrower subset of security issues that are prevalent.

Several security trend analysis studies have been conducted based on

CVE data [20, 75]. In [20], Christey et al. present an analysis of CVE data covering the period 2001 - 2006. The work is based on manual classification of CVE entries using the CWE classification system. In contrast, [75] uses an unsupervised learning technique on CVE text descriptions and introduces a classification system called *'topic model'*. While the works of Christey et al. and Neuhaus et al. focus on analyzing general trends in vulnerability databases, the work presented in this thesis specifically focuses on web application vulnerabilities, and, in particular, cross-site scripting and SQL injection. In contrast to the works of Neuhaus et al. and Christey et al., we have investigated the reasons behind the trends.

Commercial organizations such as HP, IBM, Microsoft and Whitehat collect and analyze security relevant data and publish regularly security risk and threat reports [17, 22, 27, 105]. These security reports give an overview of the prevalence of vulnerabilities and their exploitation. The data on which the studies are based, is often collected from a combination of public and private sources including vulnerability databases and honeypots. The study presented in [17], analyzes vulnerability disclosure trends using the X-Force Database. The authors identify that web application vulnerabilities account for almost 50 % of all vulnerabilities disclosed in 2011 and that cross-site scripting and SQL injection are still dominant. Although the percentage of disclosed SQL injection vulnerabilities is decreasing, a signature-based analysis on attack data suggests that SQL injection is a very popular attack vector. The security risks report of Hewlett Packard [22] identifies that the number of web application vulnerabilities submitted to vulnerabilities such as OSVDB [55] is decreasing. However, the actual number of vulnerabilities discovered by static analysis tools and blackbox testing tools on web applications respectively public websites is increasing. A threat analysis performed by Microsoft [27] shows that the number of disclosed vulnerabilities through CVE is decreasing. Common Vulnerability Scoring System (CVSS) [65] is an industry standard for assessing the severity of vulnerabilities. Their vulnerability complexity analysis based on CVSS data shows that the number of vulnerabilities that are easily exploitable is decreasing, while the number of complex vulnerabilities remains constant.

2.1.2 Evolution of Software Vulnerabilities

Our economy and society increasingly depends on large and complex software systems. Unfortunately, detecting and completely eliminating vulnerabilities before these systems go into production is an utopy. Although tools and techniques exist that minimize the number of vulnerabilities in software systems, it is inevitable that defects slip through the testing and debugging processes and escape into production runs [2, 73]. Economical incentives for software vendors also contribute to the release of insecure software. To gain market dominance, software should be released quickly and building in

security protection mechanisms could hinder this. These factors affects the reliability and security of software and impacts our economy and society.

Life-cycle of vulnerabilities

To assess the risk exposure of vulnerabilities to economy and society, one has to understand the life cycle of vulnerabilities. Arbaugh et al. [5] proposed a life cycle model for vulnerabilities and identified the following phases of a vulnerability:

- *Birth.* This stage denotes the creation of a vulnerability. This typically happens during the implementation and deployment phases of the software. A vulnerability that is detected and repaired before public release of the software is not considered as a vulnerability.

- *Discovery.* At this stage, the existence of a vulnerability becomes known to a restricted group of stakeholders.

- *Disclosure.* A vulnerability may be disclosed by a discoverer through mailing lists such as Bugtraq [106] and CERTs [19]. As we study publicly disclosed vulnerabilities in this thesis, vulnerability disclosure processes become of interest. We will outline this in the next section.

- *Patch.* The software vendor needs to respond on the vulnerability by developing a patch that fixes the flaw in the software. If the software is installed on the customers' systems, then the vendor needs to release the patch to the customers such that they can install it on their systems.

- *Death.* Once all the customers have installed the patch, the software is not vulnerable anymore. Hence, the vulnerability has disappeared.

This model was applied to three cases studies which revealed that systems remain vulnerable for a very long time after security fixes have been made available. In Chapter 4, we study the evolution of a particular class of vulnerabilities: input validation vulnerabilities in web applications. We looked in this study how long input validation vulnerabilities remain in software before these vulnerabilities are disclosed while Arbaugh et al. looked at the number of intrusions after a vulnerability has been disclosed.

Vulnerability Disclosure

The inherent difficulty of eliminating defects in software causes that software is shipped with vulnerabilities. After the release of a software system, vulnerabilities are discovered by an individual or an organization (e.g. independent researcher, vendor, cyber-criminal or governmental organization).

Discoverers have different motivations to find vulnerabilities including altruism, self-marketing to highlight technical skills, recognition or fame and malicious intents to make profit. These motivations suggest that it is more rewarding for white hat and black hat hackers – i.e. hackers with benign respectively malicious intents – to find vulnerabilities in software that is more popular.

The relationship between the application popularity and its vulnerability disclosure rate has been studied in several works. In [2], the authors examined the vulnerability discovery process for different succeeding versions of the Microsoft Windows and Red Hat Linux operating systems. An increase in the cumulative number of vulnerabilities somehow suggests that vulnerabilities are being discovered once software starts gaining momentum. An empirical study conducted by Woo et al. compares market share with the number of disclosed vulnerabilities in web browsers [120]. The results also suggest that the popularity of web browsers leads to a higher discovery rate. While the study of Woo et al. focus on web browser, we focus on web applications and discovered that popular web applications have an higher incidence of reported vulnerabilities.

After a vulnerability is discovered, the information about the vulnerability eventually becomes public. Vulnerability *disclosure* refers to the process of reporting a vulnerability to the public. The potential harms and benefits of publishing information that can be used for malicious purposes is subject of an on-going discussion among security researchers and software vendors [32]. We observe that software vendors have different standpoints on how to handle information about security vulnerabilities and adopt different *disclosure models*. We discuss them below.

- *Security through Obscurity.* This principle attempts to use secrecy of design and implementation of a system to provide security. A system that relies on this principle may have vulnerabilities, but its owners believe that if flaws are unknown it is unlikely to find them.

  Proponents of this standpoint argue that as publishing vulnerability information gives attackers the information they need to exploit a vulnerability in a system, it causes more harm than good. As there is no way to guarantee that cybercriminals do not get access to this information by other means, this is not a very realistic standpoint.

- *Full Disclosure.* In contrast to the *Security through Obscurity* stance, *Full Disclosure* attempts to disclose all the details of a security problem to the public. The vulnerability disclosure includes all the details on how to detect and to exploit the vulnerability. This is also known as the *Security through Transparency* which is also advocated by Kerckhoffs [49] who states that 'the design of a cryptographic system should

not require secrecy and should not cause 'inconvenience' if it falls into the hands of the enemy'.

Proponents of this disclosure model argue that everyone gets the information at the same time and can act upon it. Public disclosure motivates software vendors to react quickly upon it by releasing patches. However, full disclosure is also controversial. With immediate full disclosure, software users are exposed to an increased risk as creating and releasing a patch before disclosure is not possible anymore.

- *Responsible Disclosure.* In this model, all the stakeholders agree to a period of time before all the details of the vulnerability go public. This period allows software vendors to create and release a patch based on the information provided by the discoverer of the vulnerability.

  The discoverer provides the vulnerability information to the software vendor only, expecting that the vendor start to produce a patch. The software vendor has incentives to produce a patch as soon as possible because the discoverer can revert to *full disclosure* at any time. Once the patch is ready, the discoverer coordinates the publication of the advisory with the software vendor's publication of the vulnerability and patch.

To study the relationship between the disclosure model and the security of applications, Arora et al. conducted an empirical analysis on 308 vulnerabilities selected from the CVE dataset [6]. The authors compared this vulnerability data with attack data to investigate whether vulnerability information disclosure and patch availability influences attackers to exploit these vulnerabilities on one hand and on software vendors to release patches on the other. The results of this study suggest that software vendors react more quickly in case of instant disclosure and that even if patches are available, vulnerability disclosure increases the frequency of attacks. Rescorla shows in [87] that even if patches are made available by software vendors, administrators generally fail to install them. This situation does not change if the vulnerability is exploited on a large scale. Cavusoglu et al. found that the vulnerability disclosure policy influences the time it takes for a software vendor to release a patch [18]. In our study, we did not measure the impact of vulnerability disclosure. However, we used the information of disclosed vulnerabilities to measure their prevalence, the complexity and their lifetimes.

Zero-day Vulnerabilities

To conduct malicious activities, attackers exploit vulnerabilities that are unknown to the software vendor. This class of vulnerabilities is called zero-day vulnerabilities. This class of vulnerabilities is alarming as users and

administrators cannot effectively defend against them. Many techniques exist to reduce the probability of reliably exploiting (zero-day) vulnerabilities in software. These techniques include intrusion detection, application-level firewalls, Data Execution Prevention (DEP) and Address Space Layout Randomization (ASLR). To increase the reliability of software in terms of vulnerabilities and bugs, software vendors have adopted initiatives such as Secure Software Development Life Cycle Processes and made them part of their software development life cycle.

Frei et al. performed a large vulnerability study to better understand the state and the evolution of the security ecosystem at large [33]. The work focuses on zero-day vulnerabilities and shows that there has been a dramatic increase in such vulnerabilities. Also, the work shows that there is a faster availability of exploits than of patches.

Improving Software Security

In order to make investment decisions on software security solutions, it is desirable to have quantitative evidence on whether an improved attention to security also improves the security of those systems. One of the first empirical studies in this area investigated whether finding security vulnerabilities is a useful security activitity [88]. The author analyzed the defect rate and did not find a measurable effect of vulnerability finding.

To explore the relationship between code changes and security issues, Ozment et al. [80] studied how vulnerabilities evolve through different versions of the OpenBSD operating system. The study shows that 62 percent of the vulnerabilities are *foundational*; they were introduced prior to the release of the initial version and have not been altered since. The rate at which foundational vulnerabilities are reported is decreasing, somehow suggesting that the security of the same code is increasing. In contrast to our study, Ozment el al.'s study does not consider the security of web applications.

Clark et al. present in [21] a vulnerability study with a focus on the early existence of a software product. The work demonstrates that re-use of legacy code is a major contributor to the rate of vulnerability discovery and the number of vulnerabilities found. In contrast to our work, the paper does not focus on web applications, and it does not distinguish between particular types of vulnerabilities.

To better identify whether it is safe to release software, several works have focused on predicting the number of remaining vulnerabilities in the code base of software. To this end, Neuhaus et al. proposed Vulture [76]; a system that can predict new vulnerabilities using the insight that vulnerable software components such as functional calls and imports share similar past vulnerabilities. An evaluation on the Mozilla code base reveals that the system can actually predict half of the future vulnerabilities. Yamaguchi et al. [122] proposes a technique called vulnerability extrapolation to find

similar unknown vulnerabilities. It is based on analyzing API usage patterns which is more fine grained than the technique on which Vulture is based.

2.2 Web Application Security Studies

Web applications have become a popular way to provide access to services and information. Millions of people critically depend on web applications in their daily lives. Over the years, web applications have evolved towards complex software systems that exhibit critical vulnerabilities. Due to the popularity, the ease of access and the sensitivity of the information they process, web applications have become an attractive target for attackers and these critical vulnerabilities are actively being exploited. This causes financial losses, damaged reputation and increased technical maintenance and support. To improve the security of web applications, tools and techniques are necessary that reduce the number of vulnerabilities in web applications. To drive the research on web application security, several studies have been conducted that analyze the security of web applications, frameworks and programming languages.

To measure the overall state of web application security, Walden et al. measured the vulnerability density of a selection of open source PHP web applications over the period 2006 until 2008 [25, 115]. In this period, the source code repositories of these applications where mined and then the code was exercised by static analysis to find vulnerabilities. While vulnerability density of the aggregate source code decreased over the time period, the vulnerability density of eight out of fourteen applications increased. However, the vulnerability density is still above the average vulnerability density of a large number of desktop- and server- C/C++ applications which somehow suggests that the security of web application is more immature. An analysis on the vulnerability type distribution learns that the share of SQL injection vulnerabilities decreased dramatically while the share of cross-site scripting vulnerabilities increased in this period. The observation of the decrease of SQL injection vulnerabilities in an application's code base is consistent with our study on the evolution of input validation vulnerabilities presented in Chapter 4. Compared to the study of Walden et al., our analysis is performed on a larger scale as it uses NVD CVE data as data source.

Although design flaws and configuration issues are causes of vulnerabilities, most web application vulnerabilities are a result of programming errors. To better understand which tools and techniques can improve the development of secure web applications, several studies explored the relationship between the security of web applications and programming languages, frameworks and tools.

Fonseca et al. studied how software faults relate to web application security [29, 107]. Their results show that only a small set of software fault types

is responsible for most of the XSS and SQL injection vulnerabilities in web applications. Moreover, they empirically demonstrated that the most frequently occurring fault type is that of missing function calls to sanitization or input validation functions. Our work on input validation mechanisms in web programming languages and frameworks, presented in Chapter 5 of this thesis, partially corroborates this finding, but also focuses on the potential for automatic input validation as a means of improving the effectiveness of existing input validation methods.

In [118], Weinberger et al. explored in detail how effective web application frameworks are in sanitizing user-supplied input to defend applications against XSS attacks. In their work, they compare the sanitization functionality provided by web application frameworks and the features that popular web applications require. In contrast to our work presented in Chapter 5, their focus is on output sanitization as a defense mechanism against XSS, while we investigate the potential for input validation as an additional layer of defense against both XSS and SQL injection.

Finifter et al. also studied the relationship between the choice of development tools and the security of the resulting web applications [28]. Their study focused in-depth on nine applications written to an identical specification, with implementations using several languages and frameworks, while our study is based on a large vulnerability data set and examined a broader selection of applications, languages, and frameworks. In contrast to our study in Chapter 5, their study did not find a relationship between the choice of development tools and application security. However, their work shows that automatic, framework-provided mechanisms are preferable to manual mechanisms for mitigating vulnerabilities related to Cross-Site Request Forgery, broken session management and insecure password storage. Walden et al. studied a selection of Java and PHP web applications to study whether language choice influences vulnerability density [114]. Similar to Finifter et al., the result was not statistically significant.

2.3 Mitigating Web Application Vulnerabilities

In the past decade, security researchers have worked on a number of techniques to improve the security of web applications. Generally, the related work has focused on classes of techniques that prevent the exploitation of vulnerabilities, techniques based on program analysis to detect vulnerabilities and the secure construction of web applications. We will discuss each of these classes in a separate section.

2.3.1 Attack Prevention

Due to legacy or operational constraints, it is not always possible to create, deploy and use secure web applications. Developers of web applications do

not always have the necessary security skills, sometimes it is necessary to reuse insecure legacy code and in other cases it is not possible to have access to the source code. To this end, various techniques have been proposed to detect the exploitation of web vulnerabilities with the goal of preventing attacks against web applications. In this section, we discuss web application firewalls and intrusion detection systems. These techniques are common in the sense that they analyze the payload submitted to the web application.

Input Validation

To mitigate the impact of malicious input data to web applications, techniques have been proposed for validating input. Scott and Sharp [103] proposed an application-level firewall to prevent malicious input from reaching the web server. Their approach required a specification of constraints on different inputs, and compiled those constraints into a policy validation program. Nowadays, several open source and commercial offerings are available including Barracuda Web Application Firewall [9], F5 Application Security Manager ASM [26] and ModSecurity [111]. In contrast to these approaches, our approach presented in Chapter 6 is integrated in the web application development environment and automatically learns the input constraints.

Automating the task of generating test vectors for exercising input validation mechanisms is also a topic explored in the literature. Sania [54] is a system to be used in the development and debugging phases. It automatically generates SQL injection attacks based on the syntactic structure of queries found in the source code and tests a web application using the generated attacks. Saxena et al. proposed Kudzu [97], which combines symbolic execution with constraint solving techniques to generate test cases with the goal of finding client-side code injection vulnerabilities in JavaScript code. Halfond et al. [36] use symbolic execution to infer web application interfaces to improve test coverage of web applications. Several papers propose techniques based on symbolic execution and string constraint solving to automatically generate cross-site scripting and SQL injection attacks and input generation for systematic testing of applications implemented in C [16, 51, 50]. We consider these mechanisms to be complementary to the approach presented in Chapter 6, in that they could be used to automatically generate input to test our input validation solution.

Intrusion Detection

An intrusion detection system (IDS) is a system that identifies malicious behavior against networks and resources by monitoring network traffic. IDSs can be either classified as signature-based or anomaly-based:

- Signature-based detection. Signature-based intrusion detection systems monitors network traffic and compares that with patterns that

are associated with known attacks. These patterns are also called *signatures*. Snort [92] is an open source intrusion detection system which is by default configured with a number of 'signatures' that support the detection of web-based attacks. One of the main limitations of signature-based intrusion detection is that it is very difficult to keep the set of signatures up-to-date as new signatures must be developed when new attacks or modifications to previously known attacks are discovered. Almgren et al. proposed in [3] a technique to automatically deduce new signatures by tracking hosts exhibiting malicious behavior, combining signatures and by generalizing signatures. Noisy data are a result of software bugs and corrupt data can cause a significant number of false alarms reducing the effectiveness of intrusion detection systems. Julisch [48] studied the root causes of alarms and identified that a few dozens of root causes trigger 90 % of the false alarms. In [112], Vigna et al. present WebSTAT: a stateful intrusion detection system that can detect more complex attacks such as cookie stealing and malicious behavior such as web crawlers that ignore the `robots.txt` file.

- Anomaly-based detection. Anomaly-based intrusion detection systems first build a statistical model describing the normal behavior of the network traffic. Then, the system can determine network traffic that significantly deviates from the model and identify that as *anomalous* behavior. Kruegel et al. [56, 57] proposed an anomaly-based detection system to detect web-based attacks. In this system, different statistical models are used to characterize the parameters of HTTP requests. Unfortunately, anomaly-based detection systems are prone to produce a large number of false positives and/or false negatives. As the detection of web attacks are relatively rare events, false positives form a big problem. In addition, anomaly-based intrusion detection systems are often not able to identify the type of web-based attack it has detected. To improve existing anomaly-based detection systems, Robertson et al. [91] proposed a technique which generalizes anomalies into a signature such that similar anomalies can be classified as false positives and dismissed. Heuristics are used to infer the type of attack that caused the anomaly.

While intrusion detection systems focus on detecting attacks against web applications that have already been deployed, our focus in this thesis is on improving the secure development of web applications. To that end, we propose in Chapter 6 a system that prevents the exploitation of input validation vulnerabilities in web applications as part of the application development environment.

Similar to our approach, an intrusion detection and prevention system prevents attacks. To stop an attack, such a system may change the payload,

terminate the connection or reconfigure the network topology (e.g. reconfiguring a firewall). If used with the purpose of protecting web applications, an intrusion detection and prevention system can be considered as a special form of a web application firewall.

Client-Side XSS Prevention

Client-side or browser-based mechanisms such as Noncespaces [34], Noxes [52], BEEP [44], DSI [74], or XSS auditor [10] relies on the browser infrastructure to prevent the execution of injected scripts. Noncespaces [34] prevents XSS attacks by adding randomized prefixes to trusted HTML content. A client side policy checker parses the response of the server and checks for injected content that does not correspond to the correct prefix. Unfortunately, this policy checker has significant impact on the performance of rendering web pages.

BEEP [44] is a policy-based system that prevents XSS attacks on the client-side by whitelisting legitimate scripts and disabling scripts for certain regions of the web page. The browser implements a security hook which enforces the policy that has to be embedded in each web page. Although the system can prevent script injection attacks, it cannot prevent against other forms of unsafe data usage (e.g. injecting malicious iframes). BEEP requires also changes to the source code of a web application, a process that can be complicated and error-prone. The goal of DSI [74] is to preserve the integrity of document structure. On the server-side, dynamic content is separated from static content and these two components are assembled on the client-side, thereby preserving the document structure intended by the web developer.

Kirda et al. proposed Noxes [52], a client-side firewall that stops leaking sensitive data to the attackers' servers by disallowing the browser from contacting URLs that do not belong to the web application's domain. Some browser-based XSS filters have been proposed to detect injected scripts. These filter include: Microsoft Internet Explorer 8 [93], noXSS [86] and NoScript [61]. The aim of these approaches is to block reflected XSS attacks by searching for content that is present both in HTTP requests and in HTTP responses. These approaches are either prone to a large number of false positives or lack performance. XSS auditor [10] attempts to overcome these issues by detecting XSS attacks after HTML parsing but before script execution.

Each of these aforementioned approaches requires that end-users upgrade their browsers or install additional software; unfortunately, many users do not regularly upgrade their systems [113].

Server-Side XSS or SQL Injection Prevention

Research effort has also been spent on server-side mechanisms for detecting and preventing XSS and SQL injection attacks. Many techniques focus on the prevention of injection attacks using runtime monitoring. For example, Wassermann and Su [110] propose a system that checks at runtime the syntactic structure of a SQL query for a tautology. AMNESIA [37] checks the syntactic structure of SQL queries at runtime against a model that is obtained through static analysis. Boyd et al. proposed SQLrand [14], a system that prevents SQL injection attacks by applying the concept of instruction-set randomization to SQL. The system appends to aach keyword in a SQL statement a random integer, a proxy server intercepts the randomized query and performs de-randomization before submitting the query to the database management system. XSSDS [46] is a system that aims to detect cross-site scripting attacks by comparing HTTP requests and responses. While these systems focus on preventing injection attacks by checking the integrity of queries or documents, we focus in this thesis on the secure development of web applications using input validation. Recent work has focused on automatically discovering parameter injection [7] and parameter tampering vulnerabilities [83].

Ter Louw et al. proposed BLUEPRINT [59], a system that enables the safe construction of parse trees in a browser-agnostic way. BLUEPRINT requires changes to the source code of a web application, a process that can be complicated and error-prone. In contrast, the IPAAS approach presented in Chapter 6 does not require any modifications to the application's source code. Furthermore, it is platform- and language-agnostic.

2.3.2 Program Analysis

Program analysis refers to the process of automatically analyzing the behavior of a computer program. Computer security researchers use program analysis tools for a number of security-related activities including vulnerability discovery and malware analysis. Depending on whether the program during the analysis is executed, program analysis is classified as static analysis or dynamic analysis. In this section, we discuss related work that has applied program analysis techniques to find input validation vulnerabilities in web applications.

Static Analysis

Analyzing software without executing it, is called static analysis. Static analysis tools perform an automated analysis on an abstraction or a model of the program under consideration. This model has been extracted from the source code or binary representation of the program. It has been proven that finding all possible runtime errors in an application is an undecidable

problem; i.e. there is no mechanical method that can truthfully answer if an application exhibits runtime errors. However, it is still useful to come up with approximate answers.

Static analysis as a tool for finding security-critical bugs in software has also received a great deal of attention. WebSSARI [41] was one of the first efforts to apply classical information flow techniques to web application security vulnerabilities, where the goal of the analysis is to check whether a sanitization routine is applied before data reaches a sensitive sink. Several static analysis approaches have been proposed for various languages including Java [58] and PHP [47]. Statically analyzing web applications has a number of limitations. Many web applications rely on values that cannot be statically determined (e.g. current execution path, current system date, user input). This hinders the application of static analysis techniques. In addition, web applications are often implemented in dynamic weakly-typed languages (e.g. PHP). This class languages make it difficult to infer the possible values of variables by static analysis tools. These limitations result in practice into imprecision [121].

The IPAAS approach presented in Chapter 6 incorporates a static analysis component as well as a dynamic component to learn parameter types. While our prototype static analyzer is simple and imprecise, our evaluation results are nevertheless encouraging.

Dynamic Analysis

In contrast to static analysis, dynamic analysis is performed by executing the program under analysis on a real or virtual processor. Dynamic analysis can only verify properties over the paths that have been explored. Therefore, the target program must be executed with a sufficient number of test inputs to achieve adequate path coverage of the program under analysis. The use of techniques from software testing such as code coverage can assist in ensuring that an adequate set of behaviors have been observed.

Approaches based on dynamic analysis to automatically harden web applications have been proposed for PHP [82] and Java [35]. Both approaches hardcode the assertions to be checked thereby limiting the types of vulnerabilities that can be detected. In contrast, RESIN [123] is a system that allows application developers to annotate data objects with policies describing the assertions to be checked by the runtime environment. This approach allows the prevention of directory traversal, cross-site scripting, SQL injection and server-side script injection. Similar to RESIN, GuardRails [15] also requires the developer to specify policies. However, the policies are assigned to classes instead of objects. Hence, developers do not have to assign manually a policy to all the instances of a class as it is the case with RESIN. Although these approaches can work at a finer-grained level than static analysis tools, they incur runtime overhead. All these approaches aim

to detect missing sanitization functionality while the focus of this thesis is the validation of untrusted user input.

Sanitization Correctness

While much research effort has been spent on applying taint-tracking techniques [47, 58, 77, 82, 117, 121] to ensure that untrusted data is sanitized before its output, less effort has been spent on the correctness of input validation and sanitization. Because taint-tracking techniques do not model the semantics of input validation and sanitization routines, they lack precision. Wassermann proposed a technique based on static string-taint analysis that determines the set of strings an application may generate for a given variable to detect SQL injection vulnerabilities [116] and cross-site scripting vulnerabilities [117] in PHP applications.

Balzarotti et al. [8] used a combination of static and dynamic analysis techniques to analyze sanitization routines in real web applications implemented in PHP. The results show that developers do not always implement correct sanitization routines. The BEK project [40] does not focus on taint-tracking, but proposes a language to model and a system to check the correctness of sanitization functions.

Recent work has also focused on the correct use of sanitization routines to prevent cross-site scripting attacks. Scriptgard [98] can automatically detect and repair mismatches between sanitization routines and context. In addition, it ensures the correct ordering of sanitization routines.

2.3.3 Black-Box Testing

Black-box web vulnerability scanners are automated tools used by computer security professionals to probe web applications for security vulnerabilities without requiring access to the source code. These tools mimic real attackers by generating specially crafted input values, submitting that to the reachable input vectors of the web application and observing the behavior of the application to determine if a vulnerability has been exploited. Web vulnerability scanners have become popular because they are agnostic to web application technologies, the ease of use and the high degree of automation these tools provide. Examples of web vulnerability scanners include: Acunetix WVS [1], HP WebInspect [38], IBM Rational AppScan [42], Burp [60] and w3af [89]. Unfortunately, these tools also have limitations. In particular, these tools do not provide any garantuee on soundness and, as a matter of fact, several studies have shown that web vulnerability scanners miss vulnerabilities [4, 81, 119].

Independent from each other, Bau et al. [11] and Doupe et al. [24] studied the root causes behind the errors that web vulnerability scanners make. Both identified that web vulnerability scanners have difficulties in finding

more complex vulnerabilities such as stored XSS and second order SQL injection. To discover these more complex vulnerabilities, web vulnerability scanners should implement improved web crawling functionality and improved reverse engineering capabilities to keep better track of the state of the application.

2.3.4 Security by Construction

Until now, the discussion on related techniques for securing web applications has concentrated around techniques to prevent web attacks and the analysis of web applications to find vulnerabilities. These techniques can be used to protect web applications when developers use insecure programming languages to develop a web application. With insecure programming languages, vulnerabilities occur in source code because the default behavior of these languages is unsafe: the developer has to apply manually an input validation or an output sanitization routine on data before it can be used to construct a web document or SQL query. *Security by construction* refers to a set of techniques that aim to automatically eliminate issues such as cross-site scripting and SQL injection by providing safe behavior as default. The use of these techniques can automatically result in more secure web applications without much effort from the developer. In this section, we give an overview of these techniques.

Several works have leveraged the language's type system to provide automated protection against cross-site scripting and SQL injection vulnerabilities. In the approach of Robertson et al. [90], cross-site scripting attacks are prevented by generating HTTP responses from statically-typed data structures that represent web documents. During document rendering, context-aware sanitization routines are automatically applied to untrusted values. The approach requires that the web application constructs HTML content using special algebraic data types. This programmatic way of writing client-side code hinders developer acceptance. In addition, the approach is not easily extensible to other client-side languages than HTML, e.g. JavaScript and Flash. In contrast, Johns et al. [45] proposed a special data type which integrates embedded languages such as SQL, HTML and XML in the implementation language of the web application. The application developer can continue to write traditional SQL or HTML/JavaScript code using the special data type. A source-to-source code translator translates the data assigned to the special datatype to enforce a strict separation between data and code with the goal of mitigating cross-site scripting and SQL injection vulnerabilities. While these approaches focus on automated output sanitization, our focus is on automated input validation. In contrast to output sanitization, input validation does not prevent all cross-site scripting and SQL injection vulnerabilities. However, the IPAAS approach presented in Chapter 6 can secure legacy applications written using insecure languages

and it has shown to be remarkably effective in preventing cross-site scripting and SQL injection vulnerabilities in real web applications.

Recent work has focused on context-sensitive output sanitization as countermeasure against cross-site scripting vulnerabilities. To accurately defend against cross-site scripting vulnerabilities, sanitizers need to be placed in the right context and in the correct order. Scriptgard [98] employs dynamic analysis to automatically detect and repair sanitization errors in legacy .NET applications at runtime. Since the analysis is performed per-path, the approach relies on dynamic testing to achieve coverage. Samuel et al. [96] propose a type-qualifier based mechanism that can be used with existing templating languages to achieve context-sensitive auto-sanitization. Both approaches only focus on preventing cross-site scripting vulnerabilities. As we focus on automatically identifying parameter data types for input validation, our approach presented in Chapter 6 can help preventing other classes of vulnerabilities such as SQL injection or, in principle, HTTP Parameter Pollution [7].

Chapter 3

Overview of Web Applications and Vulnerabilities

Web applications have become tremendously popular due to the cross-platform compatibility, the ease of rolling out and maintaining web applications without the need to install software on potentially thousands of computers.

Unfortunately, web applications are also frequently targeted by attackers. Every day, security professionals and hackers discover new vulnerabilities in web applications affecting the security of those applications. Hackers have an increasing list of weaknesses in the web application structure at their disposal, which they can exploit to accomplish a wide variety of malicious tasks.

The goal of this dissertation is to come up with novel techniques that help developers to create secure web applications. In order to design and implement secure web applications, an understanding of the web application paradigm as well as of web vulnerabilities and how they are exploited is required.

In this Chapter, we first provide an overview on the foundations of web applications. Then, we give an overview of web application vulnerabilities and we discuss their countermeasures.

3.1 Web Applications

In order to explain web application related vulnerabilities, we first need to get a better understanding of the working of web applications. In this section, we give an overview of the web application paradigm and technologies involved.

Web applications can be considered as a specific variant of client-server software as it is software that is typically distributed over a Web server and

a Web browser. The Web server implements the application's logic. The client-side of the web application implements the user interface. It consists of HTML, JavaScript and CSS components that are received and interpreted by the Web browser. Web browser and web server communicate with each other via the HTTP protocol. As the HTTP protocol is a stateless protocol, separate mechanisms are required to manage sessions in web applications. In this section, we explore specific topics related to web browsers, web servers, the client/server communication and session management.

3.1.1 Web Browser

In this section, we discuss several browser related web technologies that are relevant in the context of this thesis.

- *HyperText Markup Language (HTML)* is the markup language for web pages. The language provides the means to create structured web documents through HTML elements. These elements are the building blocks of web pages. An element is often structured as follows: a start tag, the content and then an end tag. Tags are enclosed in angle brackets, for example `<img>` is used to specify an image element. Some HTML elements do not contain content or an end tag; these elements are called *empty elements*. HTML documents can embed images, videos and scripts. The embedding of scripts allows manipulating the behavior of web pages and it makes web pages more interactive. HTML can also be used to create interactive forms which are used to submit user-supplied data to a web server.

- *Cascading Style Sheets (CSS)* is a presentation language for web pages. The language can be used to describe the presentation semantics of a document written in a markup language. Although HTML can also be used to describe presentation semantics, CSS allows to separate document content from document presentation which improves the maintainability and extensibility of web applications.

- *JavaScript* is a scripting language and, in the context of the web application paradigm, primarily used as client-side scripting language to achieve a richer user experience. In order to use Javascript scripts within web pages, scripts need to be imported using the `<script>` HTML tag. The language relies on a runtime environment (e.g. Web browser) and enables programmatic access to objects within this runtime environment by invoking functions that are part of APIs. Common examples of programmatic access to objects in the browser are HTTP Cookies and the `XMLHttpRequest` object which is commonly used in AJAX web applications. AJAX is a set of client-side techniques that allow to create asynchronous web applications that send

and retrieve data from a web server in the background, without interfering with the display and behavior of the display of the existing web page.

Web browsers use an internal model to render documents. This model is equivalent to the Document Object Model (DOM), a platform- and language-independent way for representing HTML and XML content. Many implementations of DOM allow client-side languages to manipulate the model through an Application Programming Interface (API).

3.1.2 Web Server

Traditionally, web applications were nothing more than a set of static HTML files that could be fetched from a Web server and then were rendered and displayed by a Web browser. The introduction of *Common Gateway Interface (CGI)* changed this situation. CGI is a standard method for web servers to delegate the generation of web pages to an executable. The executable can be written in any language supported by the web server, e.g. C/C++ but also scripting languages such as Perl and Python are supported as long as a suitable interpreter has been installed on the web server.

With CGI, each request to an executable causes the creation of a new process on the web server. As the time to create and destroy a process might take much more processing time than the generation of the web page itself, performing small computations result into significant overhead. Therefore, several alternative approaches to CGI have been proposed. Examples include web server-specific techniques such as Apache modules, IIS ISAPI plug-ins and FastCGI. Also, complete new architectures for dynamic websites, that execute code as part of the web server process, have been proposed. These solutions span threads instead of creating processes upon requests. Hence, they have lower runtime overhead compared to solutions that require the creation and destruction of processes. Examples of these solutions include Java Enterprise Edition and Microsoft ASP.NET.

Structure

Apart from developments in server-side techniques, also the structure of web applications has evolved over time. While traditionally web applications consisted of only a presentation layer which resides on the client machine, nowadays web applications commonly employ a three tier approach. These three tiers are called: presentation, application and storage. The first tier is the Front End which is the HTML, JavaScript and CSS content rendered by the web browser. The Front End web server serves the static content and cached dynamic content. The application tier is implemented by an application server (ASP.NET, CGI, Coldfusion, Java EE, Perl, PHP, Ruby

On Rails). A back-end database hosted on a database management system implements the the storage tier.

For more complex web applications, it may be beneficial to use an n-tiered approach by splitting up the application tier where the business logic resides in smaller chunks. For example, the application tier can be split up in a logical layer and a data access layer. The data access layer provides an interface to access data from a database. Then, the logical layer can process the data by invoking the data access layer.

Development

Web applications are implemented using a combination of different programming languages. The client-side part of an application is typically implemented using a combination of HTML, Cascading Style Sheets and Javascript. The application-logic that resides on the server can be implemented in ASP.NET, CGI, Coldfusion, Java EE, Perl, PHP, Ruby On Rails.

To facilitate the rapid development of web applications, web application frameworks are available. These frameworks implement functionality common to web applications and allow programmers to reuse that functionality avoiding development overhead. For example, web application frameworks provide libraries for database access, input validation, authentication and session management and templating. Web application frameworks are available for a wide variety of different programming languages. Examples of web application frameworks include: Play, Struts2, Symfony, Django, Ruby On Rails and Yii.

The use of web application frameworks can potentially reduce the number of errors in web applications as it allows the developer to concentrate on application's functionality rather than on non-functional properties and a framework makes the code simpler. Also, by providing common security functionality, the number of vulnerabilities can potentially be reduced resulting in more secure web applications.

3.1.3 Communication

In this section, we discuss several concepts related to web browser and web server communication that are relevant to the content of this thesis.

Uniform Resource Locators (URLs)

Uniform Resource Locators (URLs) provide the means to identify and locate a resource. HTTP URLs (RFC 1738) are a specific type of URLs and, as they are part of HTTP requests, they are key to the communication between browser and server. A web application exposes one or more resources and all those resources can be identified and located through HTTP URLs.

```
http://library.eurecom.fr/scripts/loans.php?userID=5&itemID=3#details
(1)     (2)                 (3)                 (4)                 (5)
```

Figure 3.1: Example URL.

Examples of resources include: server-side scripts (e.g. PHP scripts), client-side scripts (e.g. JavaScript files), markup (e.g. HTML files), graphics (e.g. JPEG files) and stylesheets (e.g. CSS files).

The syntax of an URL consists of the following: the scheme name (http in the case of an HTTP URL), followed by a colon, two slashes, then a domain name (alternatively, IP address), a port number, the path of the resource to be fetched or the program to be run, then, for programs such as Common Gateway Interface (CGI) scripts, a query string and a fragment identifier. Note that the port number and fragment identifier are optional fields. If the port number is omitted, a default port number is used which is 80 for HTTP and 443 for HTTPS. The fragment identifier specifies, if present, a position in a resource or document.

The example in Figure 3.1 shows an HTTP URL pointing to a resource in a library loan web application. This HTTP URL is structured as follows: `http` defines the protocol to be used (1); `library.eurecom.fr` is the hostname and destination location of the URL (2); the resource is located on the server at the following path `/scripts/loans.php` (3); the querystring `?userID=5&itemID=3` represents two attribute-name/value pairs separated by an ampersand that contain the data to be sent to the web server (4); the fragment identifier `#details` specifies the position in the requested document to which the browser jumps after rendering the web page (5).

A resource may contain references to other resources. In these cases, *relative* URLs instead of *absolute* URLs can be used. In contrast to relative URLs, absolute URLs such as the one shown in Figure 3.1 are independent of its context. Relative URLs point to files or directories from the current context.

The HTTP Protocol

The HTTP protocol functions as a request and response protocol. A web browser or another *user agent* submits an HTTP request message to a web server for a given resource identified by a URL. The web server in turn responds with a HTTP response message. The response message contains completion status of the request and in its body may contain the content requested by the user agent.

Figure 3.2 shows an illustrative example of an HTTP request message. The HTTP request message consists of the following components:

```
POST http://library.eurecom.fr:80/login.php HTTP/1.1
Host:  library.eurecom.fr
User-Agent:  Mozilla/5.0 (Windows NT 6.1; WOW64; rv:8.0.1)
Accept:  text/html
Accept-Language:  en-us,en
Accept-Encoding:  gzip, deflate
Accept-Charset:  ISO-8859-1,utf-8

username=johndoe&password=secret
```

Figure 3.2: Example of a HTTP request message.

- A request line containing the request type (`POST`), the requested resource identified by the URL `http://library.eurecom.fr:80/login.php` and the version of the HTTP protocol in use: `HTTP/1.1`.
- HTTP request headers such as `Acccept` and `Accept-Encoding` indicating the media type respectively encodings supported by the user agent.
- An empty line.
- A message body containing the parameters of the POST request. This message body is optional.

The HTTP protocol supports nine different actions that can be performed on an identified resource in a request. These actions are called HTTP request methods and are the following: HEAD, GET, POST, PUT, DELETE, TRACE, OPTIONS, CONNECT and PATCH. The GET and POST methods are the most commonly used ones on the web. GET requests are meant to retrieve a representation of the specified resource and they should not be used to alter the state of the server. A GET request can be used to send information to the server by including a querystring containing attribute-value pairs in the requested URL. POST requests are typically used to submit data to a web server and, in contrast to GET requests, POST requests can be used to create or update new resources on the web server.

Figure 3.3 shows an illusive example of an HTTP response message sent by the web server. The HTTP response message consists of the following components:

- An HTTP response status code. The status code `200` indicates that the processing of the request has succeeded. Several alternative status

```
HTTP/1.1 200 OK
Date:  Mon, 23 Jan 2012 13:12:36 GMT
Server:  Apache
X-Powered-By:  PHP/5.2.9
Expires:  Thu, 19 Nov 1981 08:52:00 GMT
Cache-Control:  no-store, no-cache, must-revalidate, post-check=0, pre-check=0
Pragma:  no-cache
X-Transfer-Encoding:  chunked
Content-Type:  text/html
Content-length:  34735

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html lang="en">
..
</html>
```

Figure 3.3: Example of an HTTP response message.

codes exist such as `301`, indicating that the resource has been moved to a new URL; `400`, the request could not be understood by the web server due to malformed syntax; `401`, the request requires authentication and `404`, the web server has not found any matching resource with the requested URL.

- HTTP response headers such as `Expires` indicating the time/date on which the response is considered to be stale, the `Content-Type` field specifies the internet media type of the content in the message body and the `Content-length` field specifies the length of the content in the message body.
- An empty line.
- The message body. The message body contains the requested content such as an HTML document, an image or some client-side script.

The HTTP protocol is a stateless protocol which means that each request is handled as an isolated transaction and is independent from any previous request. As a consequence, web application developers need to implement session management mechanisms to manage state in web applications.

HTTP Cookies

HTTP Coookies or simply *cookies* are used to exchange state information between web browser and web server. State information can range from

```
HTTP/1.1 200 OK
Content-type:  text/html
Set-Cookie:  name=value1
```

Figure 3.4: HTTP response message with cookie.

```
GET /loans.php HTTP/1.1
Host:  library.eurecom.fr
Cookie:  name=value2
Accept:  */*
```

Figure 3.5: HTTP request message with cookie.

user preferences to shopping cart contents. The only requirement is that the information can be represented in plain text as the web browser on the user's computer stores cookie information in plain text. The exchange of cookies between web browser and web server happens as part of standard HTTP request and response messages.

Cookies have several different purposes including personalization, tracking of users on the web and session management. A cookie may be used to remember the information of a user that visited a particular website and show relevant content to the user in the future. For example, a cookie can be used to store search queries. Another application for cookies is the tracking of users. A website may store a unique identifier in a cookie. Then, each time the user visits the website, the cookie is stored along with the URL and date/time of the request in a logfile. By analyzing the logfile, the website's owner can obtain detailed information about the user including the sequence of webpage views. Finally, cookies can be used by a web application to implement session management mechanisms which we will show in the next section.

A cookie can be set, read and updated on both client-side and server-side. Figure 3.4 shows an HTTP response in which the web application sets a cookie with the name `name` and value `value1`. The web browser may respond to such HTTP response by updating the value of the cookie and sending a subsequent request as shown in Figure 3.5. Web browsers handle cookies per domain name. Cookies may contain sensitive information and, for security reasons, web browsers implement the Same Origin Policy. This prevents web pages from accessing cookies that belong to other domains than the origin domain to which the cookies correspond to. Hence, cookies cannot be shared across multiple domains.

3.1.4 Session Management

Many web applications need to keep track of user's activity across multiple HTTP sessions. Session management is a technique to be implemented by the application developer to maintain state via the otherwise stateless HTTP protocol. For example, a web application may require a user to authenticate. Then, session management allows a user to submit the credentials once and to remain authenticated during multiple page requests for resources of the web application.

Session information is identified and stored on the web server by a session identifier which is generated as a result of a request to a resource on the web server. The web server stores the session information such as session identifier, username, account number, shopping basket content in memory, in flat-files on local disk or in a database.

During a user's session, the session identifier is exchanged as part of HTTP request and response messages between web browser and web server. As described in the previous section, cookies can be used to exchange state information including session identifiers. Besides the use of cookies for holding session identifiers, there are alternative mechanisms to identify sessions. These include the tracking of IP addresses, the use of querystrings and hidden fields in HTML forms to hold session identifiers. In the latter two cases, the web server needs to generate web pages containing hyperlinks with querystrings respectively hidden form fields holding the session identifier.

3.2 Web Vulnerabilities

Web applications have become attractive targets for attackers due to the large degree of authority they possess, their significant user populations, and the prevalence of vulnerabilities they contain [53]. Similar to vulnerabilities in traditional software applications, many classes of vulnerabilities in web application are a result of defects that are introduced during the development lifecycle. Some vulnerabilities affect web technologies only, other classes of vulnerabilities are very common to web applications. In this section, we explore a number of common classes of web vulnerabilities that are caused by programming errors. We describe each vulnerability along with countermeasures.

3.2.1 Input Validation Vulnerabilities

Input validation vulnerabilities are caused by improper input validation. In the essence, input validation is the process of ensuring that data, that has been received from an external source, is clean, correct, useful and does not contain any malicious content. For example, in a library loan administration system, input validation might be used to check that the ISBN number of

```
statement = "SELECT * FROM users WHERE password = '" + passWord +
            "' AND username = '" + userName + "';"
```

Figure 3.6: Example SQL statement.

a book, contains 9, 10 or 13 digits and consists of 4 or 5 parts. Although performing input validation checks is in principle a simple task, performing correct and complete validation of all input to the web application is a complex task in practice. Hence, many web applications in the wild contain input validation vulnerabilities.

In the remainder of this section, we discuss several classes of input validation vulnerabilities.

SQL Injection

Code injection can also be used to read and modify arbitrary values on a back-end database which is called SQL injection, when the database is accessed through SQL, or XPath injection when XPath is used to read or write values to the database. To illustrate a SQL injection vulnerability, Figure 3.6 shows a typical example of a SQL statement to authenticate users of a web application. More specifically, users of the web application enter their credentials using an HTML form and the SQL statement is used to retrieve the users that have the given username and password. Now, if the 'userName' variable is crafted in a specific way by a malicious user, the SQL statement may do other things than intended by the developer. Consider for example an input such as `' UNION SELECT * FROM users WHERE '1'='1` for the variable 'userName' and an empty password field, then the SQL statement `SELECT * FROM users WHERE password = " AND username = " UNION SELECT * FROM users WHERE '1'='1';` is rendered. This input may force the web application to authenticate the user as the condition `'1'='1'` is satisfied.

One can distinguish different types of SQL injection attacks:

- *Reflected attacks* are the simplest forms of SQL injection attacks. An attacker can inject SQL code in a parameter of an HTTP request message, the parameter of an HTTP request is used in the web application to construct a SQL statement using string operations without proper encoding or sanitization. Hence, the submission of SQL code leads to a modified SQL statement that is executed on the back-end database while processing the HTTP request.

- With *stored attacks*, the injected SQL code is first saved on the server.

```
echo "<p>You searched for:  " + searchTerm + "<p>";
```

Figure 3.7: Cross-site scripting example: search.php

Then, the injected code is executed after performing a subsequent HTTP request.

- *Blind* SQL injection might be used when the results of a SQL injection attack are not made visible by the web application to the attacker. The webpage containing the vulnerability may display different data depending on the logical expression in the SQL statement resulting from the injection. A variant on this is the *timing attack* in which the attacker submits time-taking operations as part of SQL statements and verifies whether the response time of the web server is affected.

Cross-Site Scripting

Cross-site scripting (XSS) is a particular class of input validation vulnerabilities that allows attackers to inject client-side code such as HTML or Javascript into web pages viewed by other users. As a result, attackers may gain elevated privileges to sensitive page content viewed by the victims, steal authentication credentials by injecting keylogging scripts or steal sensitive session data by, for example, accessing cookie data. Figure 3.7 shows a HTML fragment of a web page (search.php) displaying the results of a search query. The web script constructs the web page by concatenating HTML and the variable 'searchTerm' that represents the search term entered by the user. Consider now that a user clicks on the following hyperlink provided by the attacker: `http://www.eurecom.fr/search.php?searchTerm=<script type="text/javascript">window.open('http://attacker.com/upload.php?credentials='+document.cookie);</script>`. Then, the web page is opened containing a piece of malicious Javascript that uploads the sensitive content of the user's cookie to the server of the attacker.

Cross-Site scripting vulnerabilities come in different flavors:

- In *non-persistent* or *reflected* cross-site scripting attacks, client-side script is supplied as part of a HTTP query parameter or HTML form field. The victim submits the request with the injected code, the web application constructs a web page using parameter data from the request without proper encoding and the web server returns a web page containing the injected code.

  Phishing attacks often leverage on reflected cross-site scripting vulnerabilities. The attacker sends an e-mail containing a hyperlink to the victim. This hyperlink points to a vulnerable parameter in the web application, the value of this parameter is set to some malicious

scripting code. When, the victim receives the e-mail and clicks on the hyperlink, the victim is directed to the vulnerable entrypoint of the web application and the injected code is sent to the web server. As a result, the malicious code is executed in the trust relationship between browser and web server.

- *Persistent* or *stored* cross-site scripting attacks have often more serious consequences than non-persistent cross-site scripting attacks. In this form of attack, the attacker sends the scripting code to the web application as part of an HTTP request and the application saves the scripting code into persistent storage. As the web application uses this data from persistent storage to construct a web page, users requesting this page receive the malicious script which the browser will execute. A persistent cross-site scripting vulnerability potentially allows an attacker to inject malicious script only once and to affect potentially thousands of visitors. Also in social networking sites such as Myspace, Facebook and Twitter, persistent cross-site scripting vulnerabilities can be used to install cross-site scripting worms that replicate themselves across accounts.

- In contrast to the above classes of cross-site scripting attacks, a *DOM-based* cross-site scripting attack does not rely on server-side code to construct web pages. In many Web 2.0 applications, HTML is written dynamically. Typically, a piece of Javascript on the client-side creates and updates web content by manipulating the Document Object Model (DOM). The content to be shown is often retrieved from the web server by the client-side script accessing the `XMLHttpRequest` object and then updating the DOM. A DOM-based cross-site scripting vulnerability occurs if no encoding function is applied on the data prior to updating the DOM-tree.

Directory Traversal and Open Redirects/Forwards

In a directory traversal attack, an attacker can traverse to parent directories with the ultimate goal of gaining access to files on the web server that are not intended to be accessible. To illustrate the attack, Figure 3.8 shows a PHP code snippet containing a directory traversal vulnerability. Suppose that this script is installed on a Linux server. If an attacker invokes the script on the web server with the parameter having the value `../../../../etc/passwd`, then the PHP script will return the contents of the UNIX `/etc/passwd` file.

Open redirect and open forward vulnerabilities are essentially variants of the directory traversal vulnerability. The vulnerability occurs when input from an untrusted source is used to determine the location where the user should be forwarded/redirected to. This vulnerability allows attackers to redirect users to a destination web site that is specified within a request

```
<?php
     $template = 'default.php';
     if(iss)et($_GET['template'])) {
            $template = $_GET['template']
     }
     include('/www/sites/eurecom/templates/' .  $template)
?>
```

Figure 3.8: Directory traversal vulnerability.

```
<?php
     $username = $_GET['username'];
     setrawcookie('user',$username);
?>
```

Figure 3.9: HTTP Response vulnerability.

to the web application. As part of a phishing attack, attackers can trick victims into submitting a request which would redirect or forward the user to a malicious web page.

HTTP Response Splitting

An HTTP response splitting attack is similar to an open redirect/forward attack in the sense that an attacker can also trick a user into opening a malicious web page. Figure 3.9 shows an example of an HTTP response splitting vulnerability. The vulnerability exists because the input from the GET parameter is not sanitized when using it to set a cookie value. In an HTTP response splitting attack, the attacker submits malicious input that is crafted in two parts. In the first part, the attacker terminates the HTTP response with a carriage return followed by a line feed (e.g. the string '`\r\n`' should do the trick). Then, the second part of the malicious input is used to construct an arbitrary number of HTTP responses over which the attacker has full control.

HTTP Parameter Pollution

As the name suggests, HTTP parameter pollution (HPP) attacks pollute the parameters of a web application. In an HPP attack, an attacker injects encoded query string delimiters along with parameter-name/value pairs in a query string which makes it possible to add new parameters or override

```
<?php
     echo "<a href=\"item.php?action=create&id="
            . $_GET['id'] . " \">Create</a>";
     echo "<a href=\"item.php?action=view&id=
           " . $_GET['id'] . " \">View</a>";
     echo "<a href=\"item.php?action=delete&id=
           " . $_GET['id'] . " \">Delete</a>";
?>
```

Figure 3.10: HTTP Parameter Pollution vulnerability.

existing ones. This may result in altering the normal or intended behavior of the application. Furthermore, HPP can be used to bypass input validation mechanisms such as web application firewalls.

HPP vulnerabilities may occur client-side or server-side. In client-side HPP vulnerabilities, the injection causes the web application to return a web page with hyperlinks containing the injected parameters. Figure 3.10 shows a vulnerable PHP script which generates hyperlinks. In the normal behavior of the application, each hyperlink represents a different action. However, the attacker can inject a malicious string that delimits the query string and adds a second parameter called `action` with a value. If the string `1&action=delete` is injected in the parameter `id`, all the resulting hyperlinks will look like `<a href="item.php?action=create&id=1&action=delete">Create</a>` where the first action parameter and the text of the hyperlinks are similar to the ones shown by Figure 3.10. As PHP always selects the last parameter, the only action the user can perform is `delete`. Server-side HPP vulnerabilities work similar as client-side HPP vulnerabilities, with the difference that then the behavior of the server-side code is altered.

Countermeasures

Several mechanisms exist to mitigate input validation vulnerabilities. These mechanisms focus on the secure handling of input and output such as:

- *Input validation.* Input validation is the process of ensuring that the input to a system is correct, meaningful and secure. It uses validation routines or checks that are implemented as part of a data dictionary or explicit application logic. We can distinguish different forms of input validation such as data type checks (e.g. string or integer), digit checks (e.g. Luhn and ISBN check), allowed characters checks (e.g. a username) and allowed range checks (e.g. month should be between 1 and 12). Chapters 5 and 6 contain a more elaborate description on

input validation.

- *Output sanitization.* Output sanitization processes content that is about to output in such a way that dangerous characters, submitted to the applications, are made safe. Dangerous characters are encoded or escaped meaning that they are translated to another representation using an encoding function prior to its use in the output. Typically, programming languages have libraries containing encoding functions that can be used to prevent different types of input validation vulnerabilities. For example, to prevent SQL injection vulnerabilities in PHP scripts, the function `mysql_real_escape_string()` may be used to escape user-supplied input prior to its use in MySQL database queries. In contrast to the prevention of SQL injection vulnerabilities, defending against cross-site scripting vulnerabilities requires contextual output sanitization. There are several encoding schemes that can be used depending on the location of where the untrusted content is placed in the web document including HTML entity encoding, CSS encoding, URL encoding and Javascript encoding. An example of a function used to perform HTML entity encoding in PHP is the `htmlspecialchars()` function.

- *Prepared statements and web templates.* In code injection attacks, user-supplied data is executed as code. Several mechanisms focus on the prevention of code injection by strictly separating data from code. Prepared statements are used to execute SQL queries or updates on a database. These statements take the form of a template in which the constant values are combined with unspecified values or the parameters or placeholders. During each execution, the parameter is substituted with the actual value. Prepared statements are not only resilient against SQL injection attacks, in many cases they also contribute to greater performance as the overhead of compiling and optimizing the SQL query occurs only once.

  Web templating systems are typically used to separate content from presentation with the goal of developing and deploying web applications that are maintainable and flexible. The separation of content and presentation is also used by web template systems such as Django [30] and Google's ctemplate [43] to provide protection against cross-site scripting vulnerabilities by automatically applying sanitizers to untrusted content.

3.2.2 Broken Authentication and Session Management

Authentication and session management include all aspects of authenticating users and managing their sessions. Web applications need to administer user

accounts in order to be able to authenticate users. Once a user is authenticated, sessions must be established and maintained by the web application to keep track of the user's requests. Session management mechanisms are provided by web application frameworks. However, web application developers often implement their own session handling mechanisms which must be done in a secure way. This section discusses common issues related to user account management and session management capabilities.

Managing User Accounts

Although strong authentication methods such as cryptographic tokens and biometrics are available, users typically authenticate themselves to web applications using cost-effective solutions such as a combination of a username and password. In order to prevent attackers from breaking into accounts, web applications need to handle account data including usernames and passwords in a secure manner. The OWASP project [79] identifies the following best practices for managing user credentials:

- *Password storage.* Web applications need to store passwords securely by either encrypting or hashing the passwords prior to storing them in persistent storage. Hashing is preferred over encryption as it is not reversible. Certain situations require encryption. Then, cryptographic keys must be protected.

- *Password strength.* Web applications need to pose certain restrictions on passwords such as minimum length and complexity. This mitigates the risk of dictionary attacks and brute force attacks [108, 109].

- *Password use.* Users must handle their credentials with care. Writing down the credentials on a note increases the risk of loss or theft of credentials. To mitigate the risks of password cracking, web applications should implement mechanisms such as a maximum number of login attempts, enforce regular password change, inform the user by other channels (e.g. e-mail or SMS) of successful and failed login attempts. A web application should not detail the reasons for authentication failure (e.g. whether the username or password did not match).

- *Account listings.* Web applications should avoid listing information of other accounts including usernames. This would make it easier for an attacker to log on using a certain username.

- *Browser caching.* Credentials should not be cached by the browser. Auto completion on password form fields should be disabled.

- *Secure transit.* Credentials and session identifiers should be encrypted when sending it to the web server using a mechanism such as SSL.

At least the logon transaction should be protected. However, ideally, the whole user session should be encrypted using SSL to avoid the interception of session identifiers.

Managing Sessions

Attackers exploit weaknesses in session management to hijack the sessions of other users resulting in unauthorized access to web applications. Hence, web applications need to implement session management in a secure manner such that attacker cannot take over the ownership of a session. Attackers use different methods to perform session hijacking:

- *Session fixation.* In session fixation attacks, attackers try to set the session identifier of a victim to a particular identifier that is known by the attacker. The attacker waits until the victim logs in. Once logged in, the attacker can perform actions on behalf of the victim.

- *Session ID theft.* Sessions can be hijacked once the attacker has obtained a valid session identifier. There are different ways for stealing the session identifier. An attacker can use packet sniffing to retrieve the session identifier from unencrypted traffic. Often, the login transaction itself uses SSL encrypted link but once the user is authenticated the communication with the web site is unencrypted which allows attackers to intercept the session identifier.

  If the attacker has physical access to the client machine or the web server, he can retrieve the session identifier from files stored on the machine. Web applications may exchange session identifiers between client and server as part of a parameter of a GET request. Then, the session identifiers are visible in the browser and they are stored in the log files of the web server which makes it easier for an attacker to access the session data. Thus, sending session identifiers as part of a cookie is preferred.

- *Cross-site scripting.* As mentioned in the previous section, cross-site scripting vulnerabilities allow to execute malicious code in the trust relationship of the victim's computer and web application. The malicious code may obtain the session identifier from, for example the cookie, and send it to the attacker.

Web application developers can implement a combination of countermeasures against session hijacking. First, the web application should change the session identifier when the user logs in or even renew the identifier at every request. Then, if the attacker tries to use the identifier that has been fixed, this identifier is not valid anymore as it has been replaced by a new one. Second, the web application should store session identifiers in cookies

rather than using GET/POST parameters. Storing the session identifier in GET/POST parameters results into the leakage of session information through referrer headers and log files (e.g. web proxy cache and browser history). Finally, web applications should encrypt the communication between browser and server using SSL to avoid the interception of session identifiers through network sniffing.

3.2.3 Broken Access Control and Insecure Direct Object References

Web applications implement access control mechanisms to control who can interact with particular resources. Examples of resources include files, directories, database records, object keys and URLs. Unfortunately, web applications do not always verify if the user is authorized for the target object.

Many web applications require users to pass certain checks before they are granted access to certain resources. However, by forced browsing, an attacker can get direct access to these URLs that are 'deeper' down in the website. Thus, by directly invoking certain URLs, the attacker bypasses certain checks and gets access to resources he should not have access to.

Internal resources of web applications are often exposed to users through references. By tampering the URLs or the parameters of a web application, an attacker changes the reference and bypasses an access control policy. Since the attacker obtains access by changing the parameter's value that references an internal object, the vulnerability is also known as *Insecure Direct Object References*.

An Insecure Direct Object References vulnerability is often input validation vulnerabilities as well and vise versa. For example, directory traversal and open redirect/forward vulnerabilities are input validation vulnerabilities but are also classical examples of a *Insecure Direct Object References* vulnerability because objects (e.g. websites) are referred to without checking whether the user has permission to access the web page. Another well-known instance in which input validation overlaps with an *Insecure Direct Object References* vulnerability is the use of unverified data in a SQL call. In this scenario, an attacker obtains unauthorized access to information stored in a database by modifying the HTTP request.

Countermeasures

Preventing insecure direct object references requires that each accessible resource is protected. This can be accomplished in two ways:

- Use references to objects specific to a user or a user's session. These indirect references cause that objects can only be accessed authorized users. The web application should map the per-user indirect reference to actual objects such as files, URLs and database records.

```
http://library.eurecom.fr/returnitem.php?itemid=345
```

Figure 3.11: Cross-site request forgery example.

- The web application should check access control. Upon each request for a resource, an access control check should be performed by the web application to verify if the user is authorized to access that particular resource.

3.2.4 Cross-Site Request Forgery

In cross-site request forgery attacks, end-users perform unwanted actions in a web application. Attackers use social engineering techniques to trick the user into executing a request resulting in an unwanted action. The malicious request uses the identity and authorizations of the victim to perform the action on the victim's behalf. A successful attack exploits the trust that a web application in the user agent has.

To illustrate cross-site request forgery, Figure 3.11 shows a URL in the fictive Eurecom library web application. The `returnitem.php` script in this example, is part of a web application that is used by the librarian when items are returned. Only librarians have the privileges to perform this action. To this end, the web application implements user authentication and sessions. Sessions are supported through the use of cookies. Consider now that the librarian has been authenticated to the web application and the session identifier is stored in a cookie. If a user does not want to return a borrowed book (with item identifier 345), he can try to trick the librarian to open the link as shown in Figure 3.11. When the librarian opens the link, the web application would not require any further authentication and the application brings the system into a state reflecting that the item has been returned.

To mitigate the risks of cross-site request forgery attacks, user should avoid the use of 'remember me' and should use 'log out' functionality in web applications. This in addition to preventative techniques implemented by web application developers. A preventative technique commonly used in web applications is to include a user-specific and unique token in each web request. Whenever the web application generates a response for the client, the web application can augment HTML forms and/or local URLs with tokens. Each legitimate request the client makes, contains the token which the web application verifies. Hence, attackers do not have any chance as they cannot put the right token in the submission.

Chapter 4

The Evolution of Input Validation Vulnerabilities in Web Applications

A considerable amount of effort has been spent by many different stakeholders on making web applications more secure. However, we lack quantitative evidence that this attention has improved the security of web applications over time. In this Chapter, we study how common classes of web vulnerabilities have evolved in the last decade. In particular, we are interested in finding out if developers are better at creating secure web applications today than they used to be in the past. We measure the exploit complexity to understand whether vulnerabilities require nowadays more complex attack scenarios to exploit vulnerabilities that are a result of insufficient countermeasures. Furthermore, we study how individual applications are exposed to vulnerabilities. We examine whether popular web applications are more exposed to vulnerabilities than non-popular applications. By measuring the lifetime of vulnerabilities in applications, we try to get an understanding of the difficulty of finding vulnerabilities.

Our study focuses on SQL injection and cross-site scripting vulnerabilities as these classes of web application vulnerabilities have the same root cause: improper sanitization of user-supplied input that results from invalid assumptions made by the developer on the input of the application. Moreover, these classes of vulnerabilities are prevalent, well-known and have been well-studied in the past decade. Thus, it is likely that there is a sufficient number of vulnerability reports available to allow an empirical analysis.

4.1 Methodology

To be able to answer how cross-site scripting and SQL injection vulnerabilities have evolved over time, it is necessary to have access to signifi-

cant amounts of vulnerability data. Hence, we had to collect and classify a large number of vulnerability reports. Furthermore, automated processing is needed to be able to extract the exploit descriptions from the reports. In the next sections, we explain the process we applied to collect and classify vulnerability reports and exploit descriptions.

4.1.1 Data Gathering

One major source of information for security vulnerabilities is the CVE dataset, which is hosted by MITRE [67]. According to MITRE's FAQ [69], CVE is not a vulnerability database but a vulnerability identification system that 'aims to provide common names for publicly known problems' such that it allows 'vulnerability databases and other capabilities to be linked together'. Each CVE entry has a unique CVE identifier, a status ('entry' or 'candidate'), a general description, and a number of references to one or more external information sources of the vulnerability. These references include a source identifier and a well-defined identifier for searching on the source's website. Vulnerability information is provided to MITRE in the form of *vulnerability submissions*. MITRE assigns a CVE identifier and a candidate status. After the CVE Editorial Board has reviewed the candidate entry, the entry may be assigned the 'Accept' status.

For our study, we used the CVE data from the National Vulnerability Database (NVD) [78] which is provided by the National Institute of Standards and Technology (NIST). In addition to CVE data, the NVD database includes the following information:

- Vulnerability type according to the Common Weakness Enumeration (CWE) classification system [68].
- The name of the affected application, version numbers, and the vendor of the application represented by Common Platform Enumeration (CPE) identifiers [66].
- The impact and severity of the vulnerability according to the Common Vulnerability Scoring System (CVSS) standard [65].

The NIST publishes the NVD database as a set of XML files, in the form: `nvdcve-2.0-year.xml`, where year is a number from 2002 until 2010. The first file, `nvdcve-2.0-2002.xml` contains CVE entries from 1998 until 2002. In order to build timelines during the analysis, we needed to know the discovery date, disclosure date, or the publishing date of a CVE entry. Since CVE entries originate from different external sources, the timing information provided in the CVE and NVD data feeds proved to be insufficient. For this reason, we fetched this information by using the disclosure date

from the corresponding entry in the Open Source Vulnerability Database (OSVDB) [55].

For each candidate and accepted CVE entry, we extracted and stored the identifier, the description, the disclosure date from OSVDB, the CWE vulnerability classification, the CVSS scoring, the affected vendor/product/version information, and the references to external sources. Then, we used the references of each CVE entry to retrieve the vulnerability information originating from the various external sources. We stored this website data along with the CVE information for further analysis.

4.1.2 Vulnerability Classification

Since our study focuses particularly on cross-site scripting and SQL injection vulnerabilities, it is essential to classify the vulnerability reports. As mentioned in the previous section, the CVE entries in the NVD database are classified according to the Common Weakness Enumeration classification system. CWE aims to be a dictionary of software weaknesses. NVD uses only a small subset of 19 CWEs for mapping CVEs to CWEs, among those are cross-site scripting (CWE-79) and SQL injection (CWE-89).

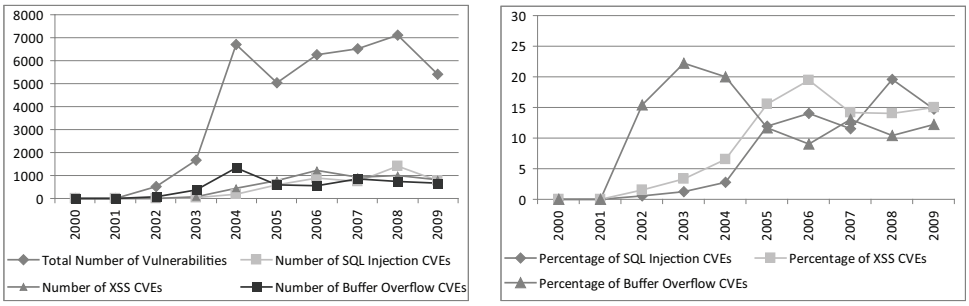
Although NVD provides a mapping between CVEs and CWEs, this mapping is not complete and many CVE entries do not have any classification at all. For this reason, we chose to perform a classification which is based on both the CWE classification and on the description of the CVE entry. In general, we observed that a CVE description is formatted according to the following pattern: {description of vulnerability} {location description of the vulnerability} *allows* {description of attacker} {impact description}. Thus, the CVE description includes the vulnerability type.

For fetching the cross-site scripting related CVEs out of the CVE data, we selected the CVEs associated with CWE identifier 'CWE-79'. Then, we added the CVEs having the text 'cross-site scripting' in their description by performing a case-insensitive query. Similarly, we classified the SQL injection related CVEs by using the CWE identifier 'CWE-89' and the keyword 'SQL injection'.

4.1.3 The Exploit Data Set

To acquire a general view on the security of web applications, we are not only interested in the vulnerability information, but also in the way each vulnerability can be exploited. Some external sources of CVEs that provide information concerning cross-site scripting or SQL injection-related vulnerabilities also provide exploit details. Often, this information is represented by a script or an *attack string*.

An attack string is a well-defined reference to a location in the vulnerable web application where code can be injected. The reference is often a



(a) Vulnerability trends in numbers (b) Vulnerability trends in percentages

Figure 4.1: Buffer overflow, cross-site scripting and SQL injection vulnerabilities over time.

complete URL that includes the name of the vulnerable script, the HTTP parameters, and some characters to represent the placeholders for the injected code. In addition to using placeholders, sometimes, real examples of SQL or Javascript code may also be used. Two examples of attack strings are:

```
http://[victim]/index.php?act=delete&dir=&file=[XSS]
http://[victim]/index.php?module=subjects&func=viewpage&pageid=[SQL]
```

At the end of each line, note the placeholders that can be substituted with arbitrary code by the attacker.

The similar structure of attack strings allows our tool to automatically extract, store and analyze the exploit format. Hence, we extracted and stored all the attack strings associated with both cross-site scripting and the SQL injection CVEs. addtoresethypothesischapter

4.2 Analysis of the Vulnerabilities Trends

The first question we wish to address in this study is whether the number of SQL injection and cross-site scripting vulnerabilities reported in web applications has been decreasing in recent years. To answer this question, we automatically analyzed the 39,081 entries in the NVD database from 1998 to 2009[1]. We had to exclude 1,301 CVE entries because they did not have a corresponding match in the OSVDB database and, as a consequence, did not have a disclosure date associated with them. For this reason, these CVE

[1] At the time of our study, a full vulnerability dataset of 2010 was not available. Hence, our study does not cover 2010.

entries are not taken into account for the rest of our study. Of the remaining vulnerability reports, we identified a total of 5349 buffer overflow entries, 5413 cross-site scripting entries and 4825 SQL injection entries.

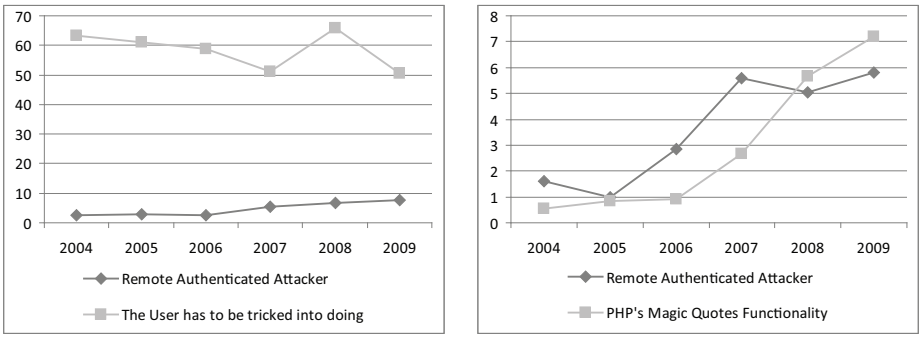
Figure 4.1a shows the number of vulnerability reports over time and figure 4.1b shows the percentage of reported vulnerabilities over the total CVE entries.

Our first expectation based on intuition was to observe that the number of reported vulnerabilities follow a classical bell shape, beginning with a slow start when the vulnerabilities are still relatively unknown, then a steep increase corresponding to the period in which the attacks are disclosed and studied, and finally a decreasing phase when the developers start adopting the required countermeasures. In fact, the graphs show an initial phase (2002-2004) with very few reports about cross-site scripting and SQL injection vulnerabilities and many reports about buffer overflow vulnerabilities. This phase is followed by a steep increase in input validation vulnerability reports in the years 2004, 2005 and 2006 and overtakes the number of buffer overflow vulnerability reports. Note that this trend is consistent with historical developments. Web security started increasing in importance after 2004, and the first XSS-based worm was discovered in 2005 (i.e., "Samy Worm" [71]). Hence, web security threats such as cross-site scripting and SQL injection started receiving more focus after 2004 and, in the meantime, these threats have overtaken buffer overflow problems. Unfortunately, the number of reported cross-site scripting and SQL injection vulnerabilities has not significantly decreased since 2006. In other words, the number of cross-site scripting and SQL injection vulnerabilities found in 2009 is comparable with the number reported in 2006. In the rest of this section, we will formulate and verify a number of hypotheses to explain the possible reasons behind this phenomenon.

4.2.1 Attack Sophistication

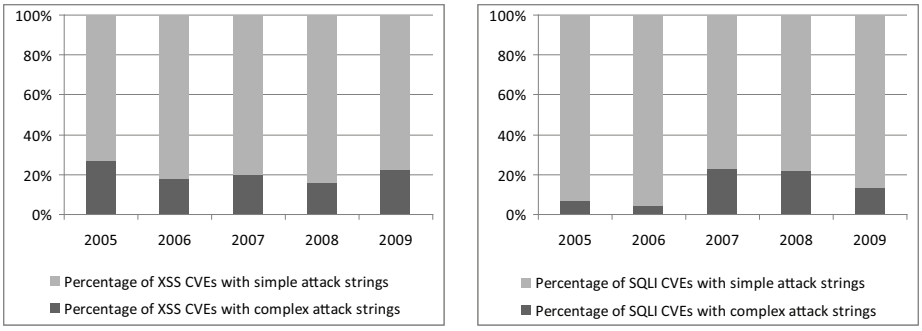
Hypothesis 1 *Simple, easy-to-find vulnerabilities have now been replaced by complex vulnerabilities that require more sophisticated attacks.*

The first hypothesis we wish to verify is whether the overall number of vulnerabilities is not decreasing because the simple vulnerabilities discovered in the early years have now been replaced by new ones that involve more complex attack scenarios. In particular, we are interested in finding out whether the prerequisites for an attack have changed over time. We were inspired by bug reports corresponding to the vulnerabilities to look at the prerequisites for attacks. By investigating the bug reports of web applications, we found out that in some of these cases, software developers are aware of a vulnerability but are unwilling to fix it because the vulnerability is only exploitable in certain scenarios and the risk is minimal. One example



(a) Cross-site scripting (b) SQL injection

Figure 4.2: Prerequisites for successful attacks (in percentages).



(a) Cross-site scripting (b) SQL injection

Figure 4.3: Exploit complexity over time.

of such a scenario is a vulnerability in the administration interface of a web application which is only expoitable by an administrator. Moreover, some vulnerabilities are only exploitable when the user is tricked into performing some action via a phishing attack, for example. Software developers may also decide not to fix SQL injection vulnerabilities if certain configuration settings can prevent the exploitation.

To determine the prerequisites for successful attacks, we searched for particular phrases in the descriptions of the CVE entries. For cross-site scripting vulnerabilities, we looked at the occurrence of the following phrases:

- 'remote authenticated' to identify whether the attacker needs to be authenticated.
- 'trick', 'tricked', 'tricking', 'crafted link', 'crafted url', 'malicious url', 'malicious link', 'malicious website', 'crafted website', 'malicious email',

‘malicious e-mail’, ‘crafted email’, ‘crafted e-mail’, ‘malicious message’, ‘crafted message’ to identify whether the attacker needs to deceive the victim into performing some action.

For SQL injection vulnerabilities, we looked for occurrence of the following keywords:

- ‘remote authenticated’ to identify whether the attacker needs to be authenticated.
- ‘without magic_quotes_gpc enabled’, ‘magic_quotes_gpc is disabled’ to determine whether disabling PHP’s magic_quotes functionality allows a SQL injection attack.

Figures 4.2a and 4.2b plot the percentage of vulnerabilities requiring the given prerequisite over the total number of cross-site scripting or SQL injection vulnerabilities in the given year, respectively. Figure 4.2a suggests that at least 50 percent of the cross-site scripting vulnerabilities require some involvement from the victim. We observe that since 2005, there has been a slight increase of SQL injection vulnerabilities that can only be exploited when the controversial magic_quotes feature is disabled. This trend is consistent with PHP’s development roadmap, which intends to deprecate the feature in PHP version 5.3.0 and remove it in version 6.0. Another trend we observed is a slight increase in vulnerabilities that require an attacker to be authenticated. Although the trend is not significant, it may suggest that developers have started to pay attention to the security of functionality accessible by everyone but fail to secure the functionality used by (website) administrators. Since the trends on prerequisites are not significant, we do not consider them as being the reason behind the steadily increasing input validation vulnerabilities trends. We are also interested in discovering whether the complexity of exploits has increased. Our purpose in doing this is to identify those cases in which the application developers were aware of threats but implemented insufficient, easy-to-evade sanitization routines. In these cases, an attacker has to craft the malicious input more carefully or has to perform certain input transformations (e.g., uppercase or character replacement).

One way to determine the “complexity” of an exploit is to analyze the attack string and to look for evidence of possible evasion techniques. As mentioned in Section 4.1.3, we automatically extract the exploit code from the data provided by external vulnerability information sources. Sometimes, these external sources do not provide exploit information for every reported cross-site scripting or SQL injection vulnerability, do not provide exploit information in a parsable format, or do not provide any exploit information at all. As a consequence, not all CVE entries can be associated with an *attack string*. On the other hand, in some cases, there exist several ways of

exploiting a vulnerability, and, therefore, many *attack strings* may be associated with a single vulnerability report. In our experiments, we collected attack strings for a total of 2632 distinct vulnerabilities.

To determine the exploit complexity, we looked at several characteristics that may indicate an attempt from the attacker to evade some form of input sanitization. The selection of the characteristics is inspired by so-called injection cheat sheets that are available on the Internet [63][95].

In particular, we classify a cross-site scripting attack string as complex (in contrast to simple) if it contains one or more of the following characteristics:

- Different cases are used within the script tags (e.g., `ScRiPt`).
- The script tags contain one or more spaces (e.g., `< script>`)
- The attack string contains 'landingspace-code' which is the set of attributes of HTML-tags (e.g., `onmouseover`, or `onclick`)
- The string contains encoded characters (e.g., `)`)
- The string is split over multiple lines

For SQL injection attack strings, we looked at the following characteristics:

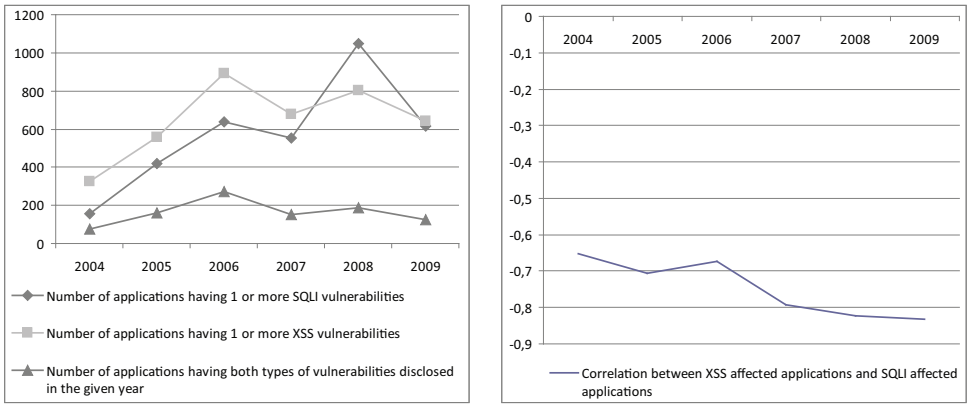
- The use of comment specifiers (e.g., `/**/`) to break a keyword
- The use of encoded single quotes (e.g., '`%27`', '`&#x27`'; '`&#39`', '`Jw==`')
- The use of encoded double quotes (e.g., '`%22`', '`"`', '`&#34`', '`Ig==`')

If none of the previous characteristics is present, we classify the exploit as "simple". Figures 4.3a and 4.3b show the percentage of CVEs having one or more complex attack strings[2]. The graphs show that the majority of the available exploits are, according to our definition, not sophisticated. In fact, in most of the cases, the attacks were performed by injecting the simplest possible string, without requiring any tricks to evade input validation.

Interestingly, while we observe a slight increase in the number of SQL injection vulnerabilities with sophisticated attack strings, we do not observe any significant increase in cross-site scripting attack strings. This may be a first indication that developers are now adopting (unfortunately insufficient) defense mechanisms to prevent SQL injection, but that they are still failing to sanitize the user input to prevent cross-site scripting vulnerabilities.

Although cross-site scripting and SQL injection vulnerabilities share the same root cause, it seems that there is more awareness of SQL injection

[2]The graph starts from 2005 because there were less than 100 vulnerabilities having exploit samples available before that year. Hence, results before 2005 are statistically less significant.

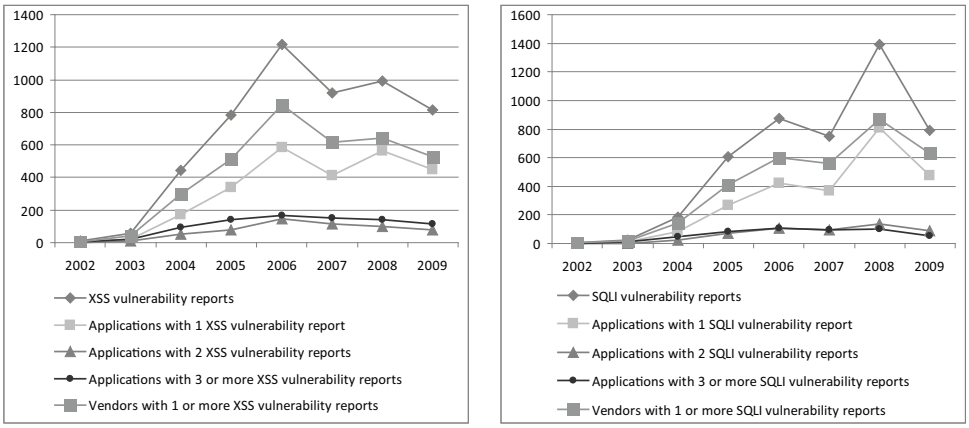


(a) The number of applications over time. (b) Correlation coefficient over time.

Figure 4.4: Applications having XSS and SQLI Vulnerabilities over time.

vulnerabilities than of cross-site scripting vulnerabilities. It is interesting to further investigate whether there is a relationship between the occurrence of the two types of vulnerabilities in web applications. This gives an answer to the question whether a developer who fails to implement countermeasures against SQL injection also fails to implement countermeasures against cross-site scripting. In order to answer this question, we started by extracting vulnerable application and vendor names from a total of 8854 SQL injection and cross-site scripting vulnerability reports in the NVD database that are associated to one or more CPE identifiers. Then, we measured the correlation between cross-site scripting and SQL injection vulnerabilities. Although correlation cannot be used to infer a causal relationship between SQL injection and cross-site scripting vulnerabilities, it can indicate the potential existence of this causal relationship.

Figure 4.4 shows the correlation between applications affected by both cross-site scripting and SQL injection vulnerabilities. More specifically, we measured the number of applications affected by both types of vulnerabilities (Figure 4.4a) and the correlation coefficient over time (Figure 4.4b). Figure 4.4b plots $\rho(X, Y)$ with $X = 0$ or 1 indicating whether the application was affected by a cross-site scripting vulnerability and $Y = 0$ or 1 indicating whether the application was affected by an SQL injection vulnerability. The graph shows a strong negative correlation, meaning that the occurrence of cross-site scripting vulnerabilities is correlated with an absence of SQL injection vulnerabilities in an application. As Figure 4.4b shows, the negative correlation tends to become stronger over time. This might indicate that developers are aware of implementing countermeasures against SQL injection but fail to do so for cross-site scripting vulnerabilities.



(a) Cross-site scripting affected applications. (b) SQL injection affected applications.

Figure 4.5: The number of affected applications over time.

To conclude, the available empirical data suggests that increased attack complexity *is not* the reason behind the steadily increasing number of vulnerability reports. Furthermore, the data suggest that applications become more resilient against SQL injection rather than cross-site scripting vulnerabilities.

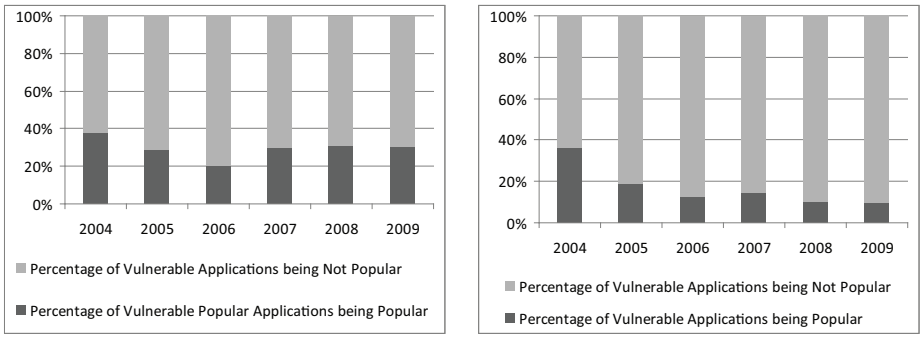
4.2.2 Application Popularity

Since complexity does not seem to explain the increasing number of reported vulnerabilities, we decided to focus on the type of applications. Figures 4.5a and 4.5b plot the number of applications and vendors that are affected by a certain number of vulnerabilities over time. Both graphs clearly show how the increase in the number of vulnerabilities is a direct consequence of the increasing number of vulnerable applications and their vendors. In fact, the number of web applications with more than one vulnerability report over the whole time frame is quite low, and it has been slightly decreasing since 2006.

Based on these findings, we formulated our second hypothesis:

Hypothesis 2 *Popular applications are now more secure while new vulnerabilities are discovered in new, less popular, applications.*

The idea behind this hypothesis is to test whether more vulnerabilities were reported about well-known, popular applications in the past than are today. That is, do vulnerability reports nowadays tend to concentrate on less popular, or recently developed applications?



(a) Cross-site scripting

(b) SQL injection

Figure 4.6: Vulnerable applications and their popularity over time.

The first step in exploring this hypothesis consists of determining the popularity of these applications in order to be able to understand if it is true that popular products are more aware of (and therefore less vulnerable to) cross-site scripting and SQL injection attacks.

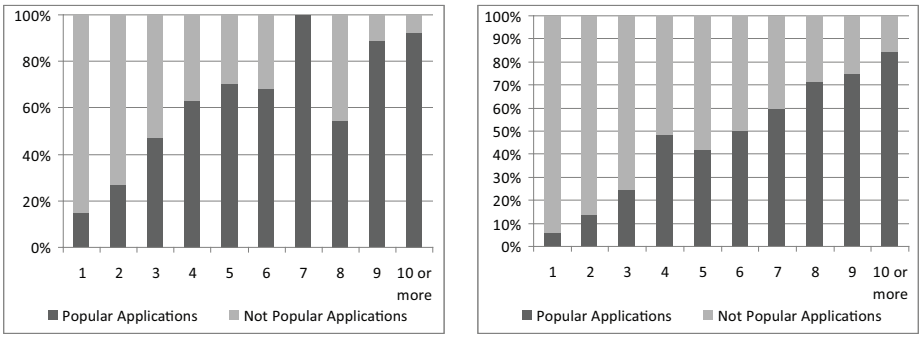
We determined the popularity of applications through the following process:

1. Using Google Search, we performed a search on the vendor and application names within the `Wikipedia` domain.

2. When one of the returned URLs contained the name of the vendor or the name of the application, we flagged the application as being 'popular'. Otherwise, the application was classified as being 'unpopular'.

3. Finally, we manually double-checked the list of popular applications in order to make sure that the corresponding Wikipedia entries described software products and not something else (e.g., when the product name also corresponded to a common English word).

After the classification, we were able to identify 676 popular and 2573 unpopular applications as being vulnerable to cross-site scripting. For SQL injection, we found 328 popular and 2693 unpopular vulnerable applications. Figure 4.6 shows the percentages of popular applications associated with one or more vulnerability reports. The trends support the hypothesis that SQL injection vulnerabilities are indeed moving toward less popular applications – maybe as a consequence of the fact that well-known products are more security-aware. Unfortunately, according to Figure 4.6a, the same hypothesis is not true for cross-site scripting: in fact, the ratio of well-known applications vulnerable to cross-site scripting has been relatively constant in the past six years.



(a) Cross-site scripting (b) SQL injection

Figure 4.7: Popularity of applications across the distribution of the number of vulnerability reports.

Even though the empirical evidence also does not support our second hypothesis, we noticed one characteristic that is common to both types of vulnerabilities: as shown in Figures shown in Figures 4.7a and 4.7b, popular applications typically have a higher number of reported vulnerabilities. There may be many possible reasons to explain this. For example, one possible explanation might be that popular applications are more frequently targeted by attackers, as the application has more impact on potential victims and thus more vulnerabilities are being reported. Another possible explanation could be that developers of popular applications are more security aware or that these applications are better analyzed. Hence, the application meets higher security standards.

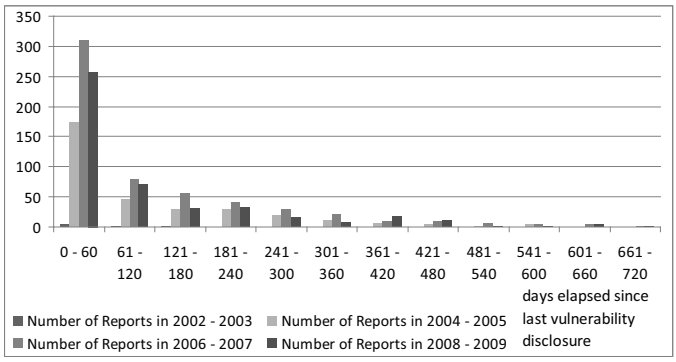
The results, shown in Figures 4.7a and 4.7b, suggest that it would be useful to investigate how these vulnerabilities have evolved in the lifetime of the applications.

4.2.3 Application and Vulnerability Lifetime

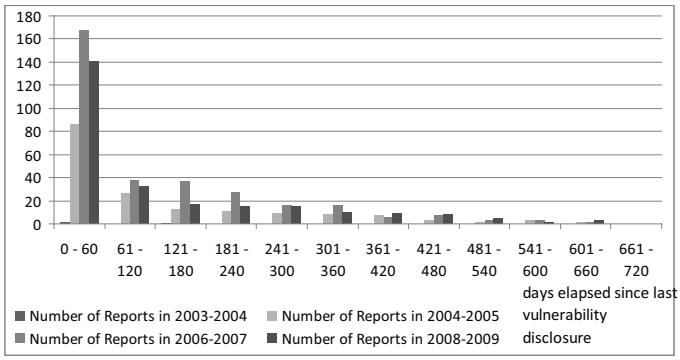
So far, we determined that a constant, large number of simple, easy-to-exploit vulnerabilities are still found in many web applications today. Also, we determined that the high number of reports is driven by an increasing number of vulnerable applications and not by a small number of popular applications. Based on these findings, we formulate our third hypothesis:

Hypothesis 3 *Even though the number of reported vulnerable applications is growing, each application is becoming more secure over time.*

This hypothesis is important, because, if true, it would mean that web applications (the well-known products in particular) are becoming more secure.



(a) Cross-site scripting



(b) SQL injection

Figure 4.8: Reporting rate of Vulnerabilities

To verify this hypothesis, we studied the frequency of vulnerability reports of applications affected by cross-site scripting and SQL injection vulnerabilities.

One way to examine the security of an application is to measure the rate at which vulnerabilities are being reported. The frequency of vulnerability reports about an application can be estimated by measuring the time between them. We applied an analogous metric from *reliability engineering*, the time between-failures (TBF) [70], by defining a vulnerability report as a failure. Figures 4.8a and Figure 4.8b plot the reporting rates of cross-site scripting vulnerabilities and SQL injection vulnerabilities, respectively. The graphs clearly show a steep increase in the reporting rates between 2002 and 2007 and a slight decrease after 2007. In order to verify the hypothesis more precisely, it is also necessary to look at the duration or lifetime of cross-site scripting and SQL injection vulnerabilities. We studied the lifetimes of cross-site scripting and SQL injection vulnerabilities in the ten most-affected open source applications according to the NIST NVD database. By analyz-

	Foundational	Non-Foundational
bugzilla	4	7
drupal	0	22
joomla	5	4
mediawiki	3	21
mybb	9	2
phorum	3	5
phpbb	4	2
phpmyadmin	14	13
squirrelmail	10	4
wordpress	6	9
Total	58	89

(a) Cross-site scripting

	Foundational	Non-Foundational
bugzilla	1	8
coppermine	1	3
e107	0	3
joomla	4	0
moodle	0	3
mybb	9	3
phorum	0	4
phpbb	3	0
punbb	4	2
wordpress	0	4
Total	22	30

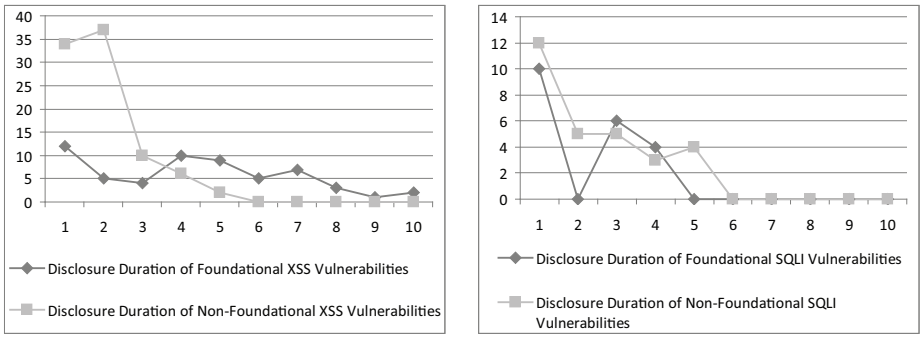
(b) SQL injection

Table 4.1: Foundational and non-foundational vulnerabilities in the ten most affected open source web applications.

ing the change logs for each application, we extracted the version in which a vulnerability was introduced and the version in which a vulnerability was fixed. In order to obtain reliable insights into the vulnerability's lifetime, we excluded vulnerability reports that were not confirmed by the respective vendor. For our analysis, we used the CPE identifiers in the NVD database, the external vulnerability sources, the vulnerability information provided by the vendor. We also extracted information from the version control systems (CVS, or SVN) of the different products.

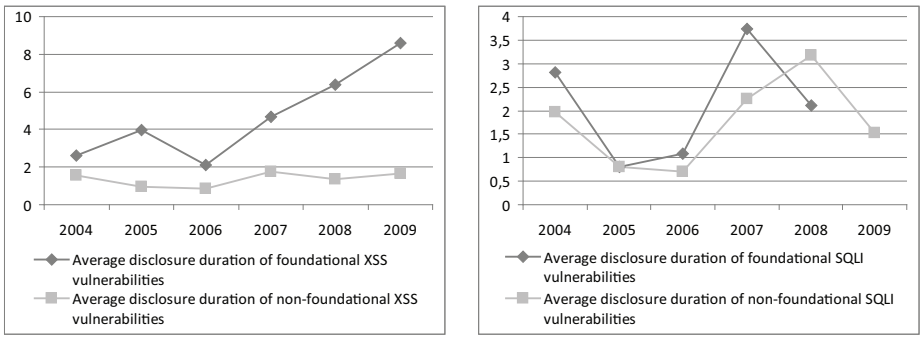
Table 4.1a and Table 4.1b show a total of 147 cross-site scripting and 52 SQL injection vulnerabilities in the most affected applications. The tables distinguish between *foundational* and *non-foundational* vulnerabilities. Foundational vulnerabilities are vulnerabilities that were present in the first version of an application, while non-foundational vulnerabilities were introduced after the initial release.

We observed that 39% of the cross-site scripting vulnerabilities are foundational and 61% are non-foundational. For SQL injection, these percentages are 42% and 58%. These results suggest that most of the vulnerabilities are introduced by new functionality that is built into new versions of a web application. Finally, we investigated how long it took to discover the vulnerabilities. Figure 4.9a and Figure 4.9b plot the number of vulnerabilities that were disclosed after a certain amount of time had elapsed after the initial release of the applications. The graphs show that most SQL injection vulnerabilities are usually discovered in the first few years after the release of the product. For cross-site scripting vulnerabilities, the result is quite different. Many foundational vulnerabilities are disclosed even 10 years after the code was initially released. This observation suggests that it is very problematic to find foundational cross-site scripting vulnerabilities compared to



(a) Cross-site scripting (b) SQL injection

Figure 4.9: Time elapsed between software release and vulnerability disclosure in years.



(a) Cross-site scripting (b) SQL injection

Figure 4.10: Average duration of vulnerability disclosure in years over time.

SQL injection vulnerabilities. This is supported by the fact that the average elapsed time between the software release and the disclosure of foundational vulnerabilities is 2 years for SQL injection vulnerabilities, while for cross-site scripting this value is 4.33 years.

Figures 4.10a and 4.10b plot the average elapsed time between software release and the disclosure of vulnerabilities over time. These results show that cross-site scripting vulnerabilities are indeed harder to find than SQL injection vulnerabilities and that foundational cross-site scripting vulnerabilities become even more difficult to find over time. Also note, that there are no foundational SQL injection vulnerabilities reported in 2009. We believe that difference between cross-site scripting and SQL injection vulnerabilities concerning the lifetime is caused by the fact that the attack surface for SQL

Vulnerable applications reporting about scripts:	1871
Vulnerable scripts:	2499
Average (vulnerable scripts / applications):	1.34
Vulnerable applications reporting about parameters:	1905
Vulnerable parameters:	9304
Average (vulnerable parameters / applications):	4.88

(a) Cross-site scripting

Vulnerable applications reporting about scripts:	2759
Vulnerable scripts:	3548
Average (vulnerable scripts / applications):	1.29
Vulnerable applications reporting about parameters:	1902
Vulnerable parameters:	6556
Average (vulnerable parameters / applications):	3.45

(b) SQL injection

Table 4.2: The attack surface.

injection attacks is much smaller when compared with cross-site scripting attacks. Therefore, it is interesting to further investigate the size of the attack surface of vulnerable applications.

From the CVE descriptions, we extracted the scripts and parameters that are vulnerable to cross-site scripting or SQL injection and we counted them. By measuring the average number of vulnerable scripts and parameters per application for both cross-site scripting and SQL injection vulnerabilities, we get insights into the size of the attack surface. Table 4.2a and 4.2b shows the number of applications that are associated with vulnerabilities related to the affected scripts and/or parameters for cross-site scripting and SQL injection vulnerabilities, respectively. In addition, the number of affected scripts and parameters is shown. We observe that the average number of vulnerable scripts and parameters per application is indeed larger for cross-site scripting vulnerabilities than for SQL injection vulnerabilities. Thus, the results confirm the intuition that the difference in vulnerability lifetime between cross-site scripting and SQL injection vulnerabilities is caused by the size of the attack surface. We believe that the attack surface of SQL injection vulnerabilities is smaller because it is easier for developers to identify (and protect) all the sensitive entry points in the code (e.g. code concerning database access) of the web application than for cross-site scripting vulnerabilities.

4.3 Summary

Our findings in this study show that the complexity of cross-site scripting and SQL injection attacks related to the vulnerabilities in the NVD database has not been increasing. Neither the prerequisites to attacks nor the com-

plexity of exploits have changed significantly. Hence, this finding suggests that the majority of vulnerabilities are not due to sanitization failure, but rather due to the absence of input validation. Despite awareness programs provided by MITRE [67], SANS Institute [23] and OWASP [79], application developers are still not implementing effective countermeasures.

Furthermore, our study suggests that a major reason why the number of web vulnerability reports has not been decreasing is because many more applications of different vendors are now vulnerable to flaws such as cross-site scripting and SQL injection. Although cross-site scripting and SQL injection vulnerabilities share the same root cause, we could not find any significant correlation between applications affected by cross-site scripting and by SQL injection vulnerabilities. In fact, the small negative correlation tends to become stronger. By measuring the popularity of the applications, we observed a trend that SQL injection vulnerabilities occur more often in an increasing number of unpopular applications.

Finally, when analyzing the most affected applications, we observe that years after the initial release of an application, cross-site scripting vulnerabilities concerning the initial release are still being reported. Note that this is in contrast to SQL injection vulnerabilities. By measuring the attack surface of cross-site scripting and SQL injection vulnerabilities, we found out that the attack surface of SQL injection vulnerabilities is much smaller than for cross-site scripting vulnerabilities. Hence, SQL injection problems may be easier to find because only a relatively small part of the application's code is used for database access.

The empirical data we collected and analyzed for this study supports the general intuition that web developers consistently fail to secure their applications. The traditional practice of writing applications and then testing them for security problems (e.g., static analysis, blackbox testing, etc.) does not seem be working well in practice. Hence, we believe that more research is needed in securing applications by design. That is, the developers should not be concerned with problems such as cross-site scripting or SQL injection. Rather, the programming language or the platform should make sure that the problems do not occur when developers produce code (e.g., similar to solutions such as in [90] or managed languages such as C# or Java that prevent buffer overflow problems).

Chapter 5

Input Validation Mechanisms in Web Applications and Languages

In the previous Chapter, we have shown that many web applications are still prone to common classes of vulnerabilities. The complexity of exploits have not been increasing over time and most exploits are still simple by nature. Clearly, web developers often fail to apply existing countermeasures and a new class of solutions is required to help to improve the security situation on the web.

To develop a web application, tools and programming languages are required. The programming language chosen to develop an application has a direct effect on how a system is to be created and the means that are required to ensure that the resulting application behaves as expected and is secure. In this Chapter, we study the relationship between the programming language and web vulnerabilities that are commonly reported.

An important property of a programming language is the type system that is being used. A type system classifies program statements and expressions according to the values they can compute, and it is useful for statically reasoning about possible program behaviors. Some popular web languages such as PHP and Perl are weakly-typed, meaning that the language implicitly converts values when operating with expressions of a different type.

The advantage of weakly-typed languages from a web developer's point of view is that they are often easy to learn and use. Furthermore, they allow developers to create applications quickly as they do not have to worry about declaring data types for the input parameters of a web application. Hence, most parameters are treated as generic "strings" even though they might actually represent an integer value, a boolean, or a set of specific characters (e.g., an e-mail address). As a result, attacks are often possible if the validation is poor. For example, an attacker could inject scripting code

(i.e., a string) into the value of a parameter that is normally used by the application to store an integer.

In order to gain deeper insights into the reasons behind common vulnerabilities in web applications, we analyze in this Chapter around 3,933 cross-site scripting (XSS) and 3,758 SQL injection vulnerabilities affecting applications written in popular languages such as PHP, Python, ASP, and Java. For more than 800 of these vulnerabilities, we manually extract and analyze the code responsible for handling the input, and determined the type of the affected parameter (e.g., boolean, integer, or string). Furthermore, we study 79 web application frameworks available for many popular programming languages.

5.1 Data Collection and Methodology

In order to study the characteristics of vulnerable web applications, it is necessary to have access to a significant amount of vulnerability data. Hence, we collected and classified a large number of vulnerability reports according to the methodology described in Section 4.1. These vulnerability reports were used to identify the programming language each web application was developed in. Furthermore, we used the vulnerability reports we gathered to semi-automatically extract vulnerable input parameters from the source code of web applications. Finally, by automatically collecting data from a number of open source project hosting services, we were able to estimate the popularity of web programming languages.

In the following, we discuss our methodology for extracting information from vulnerability reports, project hosting services, and open source web applications.

5.1.1 Vulnerability Reports

To study the relationship between programming language and vulnerable web applications, we automatically analyzed cross-site scripting and SQL injection–related CVEs. As mentioned in section 4.1, many CVE entries contain a description of the location of a vulnerability in the web application. In general, vulnerability reports use fully–qualified filenames to identify the vulnerable script. We used the filename extension to classify whether an application is written in PHP (.php), ASP/ASP.NET[1] (.asp, .aspx), ColdFusion (.cfm), Java (.jsp), Perl (.pl), and Python (.py).

The programming language of some applications could not be determined in an automated fashion as the corresponding vulnerability reports did not provide any information concerning the vulnerable scripts. In these cases,

[1] We chose to determine the platform instead of the language as we could not automatically identify whether an application was implemented in C# or Visual Basic.

we manually determined the programming language by performing search queries and analyzing the source code of the web application.

5.1.2 Attack Vectors

We analyzed the source code of a significant number of vulnerable web applications with the aim of understanding to what extent data typing and validation mechanisms could help in preventing cross-site scripting and SQL injection vulnerabilities. In order to obtain a test set of applications with a high number of vulnerable input parameters, we chose to focus our study on 20 popular open source PHP web applications that contained the highest incidence of cross-site scripting vulnerabilities, and on 20 with the highest incidence of SQL injection vulnerabilities. The 28 applications belonging to the two, largely overlapping, sets are: claroline, coppermine, deluxebb, drupal, e107, horde, jetbox, joomla, mambo, mantis, mediawiki, moodle, mybb, mybloggie, papoo, phorum, phpbb, phpfusion, phpmyadmin, pligg, punbb, runcms, serendipity, squirrelmail, typo3, webspell, wordpress, and xoops.

For each of these applications, we manually examined the corresponding vulnerability reports to identify the specific application version and any example of attack inputs. Given this information, we downloaded the source code of each application and linked the input vectors to the application's source code to determine an appropriate data type. We repeated this process for a total of 809 vulnerability reports.

In the process of linking vulnerability reports to source code, we first used the version of the source code that was known to be vulnerable. Then, we repeated the process and linked the vulnerability reports to source code in which the vulnerabilities were patched. To determine the data type of the vulnerable input parameter, we manually analyzed how each vulnerability was patched and how the value of the input parameter was used throughout the web application.

5.2 Analysis

In this section, we present the results of our empirical study, and draw conclusions from an analysis of the data. In particular, we first examine whether certain languages are more prone to cross-site scripting and SQL injection vulnerabilities. Then, we analyze the type of the input parameters that commonly serve as attack vectors, and we compare them with particular features provided by the web programming languages, or by the application frameworks available to them.

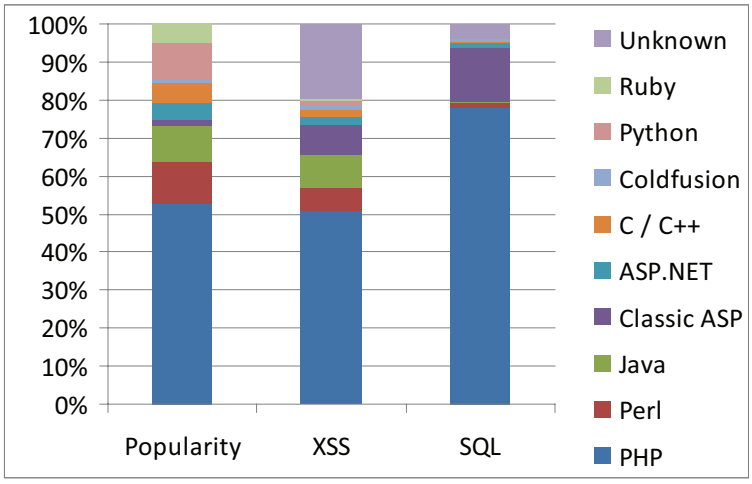


Figure 5.1: Distributions of popularity, reported XSS vulnerabilities, and reported SQL injection vulnerabilities for several web programming languages.

5.2.1 Language Popularity and Reported Vulnerabilities

A central question of this paper concerns whether the choice of programming language used to develop web applications influences the exposure of those applications to cross-site scripting and SQL injection vulnerabilities. To that end, we performed a comparison of the distribution of popular web programming languages to the distribution of reported cross-site scripting and SQL injection vulnerabilities for each of those languages.

Language popularity was calculated by crawling open source project hosting services such as Google Code, Sourceforge, and Freshmeat. For each of these services, we identified web application projects by filtering on project tags using values such as “cms”, “dynamic content”, and “message board”. These projects were classified according to the primary development language by using each service’s built-in search functionality.

The vulnerability data was drawn from the NVD [78] by automatically classifying reports according to the language of the affected application. For each report, our analysis checked whether the report concerned an cross-site scripting or SQL injection vulnerability. Our analysis identified 5,413 cross-site scripting and 4,825 SQL injection CVE entries out of a total of 39,081 entries. Because 104 of these entries did not correspond to any CPE values identifying vulnerable applications, these were excluded from that set. The resulting set of CVE entries used in our analysis was composed of 5,361 cross-site scripting and 4,773 SQL injection reports.

These vulnerability reports correspond to 3,933 and 3,758 applications vulnerable to cross-site scripting respectively SQL injection. An automated classification of the vulnerability reports was able to classify 3,254 and 2,187

of the cross-site scripting and SQL injection CVE affected applications, respectively, as implemented in a particular programming language. The remainder of the vulnerable applications were manually classified.

The distributions of language popularity and vulnerability reports are shown in Figure 5.1. The graph shows the statistics for 9 popular programming languages. Unfortunately, for 3.8% of SQL injection and 19% of cross-site scripting vulnerability reports, we were not able to automatically determine the primary development language of the application. These cases, represented by the "Unknown" category in Figure 5.1, are often related to commercial products for which the software companies do not provide information about the development language on their websites.

Under the null hypothesis — that is, that the choice of programming language does not influence the exposure of applications to cross-site scripting and SQL injection vulnerabilities — one would expect the relative distributions of popularity and reported vulnerabilities to be roughly equivalent. However, the histogram shows that this is not always the case.

The result for PHP-based applications is an illustrative example. 52% of the applications in our test set were developed in PHP, a value that also corresponds to the share of reported cross-site scripting vulnerabilities in PHP applications. Therefore, it may seem that PHP is intrinsically no more or less vulnerable than other languages to this kind of attack. However, PHP-related vulnerabilities comprised almost 80% of the SQL injection reports from our collection of CVE entries. A similar, but opposite trend mismatch can be observed for many other languages. For example, although 10% of the applications in our dataset were written in Java, we found that only 0.5% of SQL injection vulnerabilities are associated with Java-based applications. Clearly, Java applications seem to be less prone to both cross-site scripting and SQL injection vulnerabilities. These differences are too common and too large to be considered statistically insignificant.

This is also shown by Pearson's chi-square tests which we used to assess the goodness of fit. We tested the hypothesis that the language popularity and the number of vulnerabilities per language is not significantly different. For cross-site scripting, we found chi-square 48.4 and for SQL injection we found chi-square 138. The degree of freedom is 8. Looking these numbers up in the chi-square table shows that both probabilities are less than 0.001 meaning that the hypothesis is not true. Thus, the data suggests that the number of cross-site scripting respectively SQL injection vulnerabilities for a given language is not determined by the popularity of that language.

Note that there may be many possible reasons to explain the discrepancies. For example, one possible explanation might be that Java developers are simply more careful than those that favor other languages, and that PHP developers are instead worse, on average, at applying known defense techniques to prevent SQL injection. On the other hand, it might also be

that certain languages, as well as the web development frameworks available for those languages, are intrinsically more resistant to — or provide better defenses against — cross-site scripting and SQL injection vulnerabilities.

Another possible reason could be that the web development frameworks available for a certain language provide a better set of functionality (or a better API for that functionality) to properly sanitize the user inputs, thus making the life easier for the web developers. In the rest of the section, we explore in more detail these possibilities by analyzing the impact of the language type system on the security of the application, and the functionality provided by common application frameworks that can be used to prevent cross-site scripting and SQL injection vulnerabilities.

5.2.2 Language Choice and Input Validation

As we saw from Figure 5.1, the choice of programming language clearly has an influence on the exposure of applications developed in those languages to cross-site scripting and SQL injection vulnerabilities. While there are several plausible explanations for this phenomenon, one likely hypothesis is that some programming languages are intrinsically more robust against the introduction of web application vulnerabilities. In the following, we examine a particular mechanism by which a language or framework might mitigate the potential for web application attacks.

Input Validation

One defensive mechanism that is critical for the correct functioning of applications is *input validation*. In the abstract, input validation is the process of assigning semantic meaning to unstructured and untrusted inputs to an application, and ensuring that those inputs respect a set of constraints describing a well-formed input. For web applications, inputs take the form of key-value pairs of strings. The validation of these inputs may be performed either in the browser using Javascript, or on the server. Since there is currently no guarantee of the integrity of computation in the browser, security-relevant input validation should be performed on the server, and, therefore, we restrict our discussion of input validation to this context.

To elucidate the input validation process for server-side web applications, consider the pedagogical HTTP request shown in Figure 5.2. This figure shows a typical structure for a payment submission request to a fictional e-commerce application. As part of this request, there are several input parameters that the controller logic for `/payment/submit` must handle: `cc`, a credit card number; `month`, a numeric month; `year`, a numeric year; `save`, a flag indicating whether the payment information should be saved for future use; `token`, an anti-CSRF token; and `SESSION`, a session identifier. Each of these request parameters requires a different type of input validation. For

```
POST /payment/submit HTTP/1.1
Host:  example.com
Cookie:  SESSION=cbb8587c63971b8e
[...]

cc=1234567812345678&month=8&year=2012&
save=false&token=006bf047a6c97356
```

Figure 5.2: Example HTTP request.

instance, the credit card number should be a certain number of characters and pass a Luhn check. The month parameter should be an integer value between 1 and 12 inclusive. The year parameter should also be an integer value, but can range from the current year to an arbitrary year in the near future. The save flag should be a boolean value, but as there are different representations of logical true and false (e.g., $\{\text{true}, \text{false}\}$, $\{1, 0\}$, $\{\text{yes}, \text{no}\}$), the application must consistently recognize a fixed set of possible values.

Input validation, in addition to its role in facilitating program correctness, is a helpful tool to prevent the introduction of vulnerabilities into web applications. Were there an attacker to supply the value

```
year=2012'; INSERT INTO admins(user, passwd)
 VALUES('foo', 'bar');--
```

to our fictional e-commerce application as part of a SQL injection vulnerability to escalate privileges, proper input validation would recognize that the malicious value was not a valid year, with the result that the application would refuse to service the request.

Input validation can occur in multiple ways. Validation can be performed implicitly — for instance, through typecasting a string to a primitive type like a boolean or integer. For the example attack shown above, a cast from the input string to an integer would result in a runtime cast error, since the malicious value is not a well-formed integer. On the other hand, input validation can be performed explicitly, by invoking framework-provided validation routines. Explicit validation is typically performed for input values exhibiting complex structure, such as email addresses, URLs, or credit card numbers.

In this respect, the choice of programming language and framework for developing web applications plays an important role in the security of those applications. First, if a language features a strong type system such that typecasts of ill-formed input values to certain primitive types will result in runtime errors, the language can provide an implicit defense against the introduction of vulnerabilities like cross-site scripting and SQL injection.

Second, if a language framework provides a comprehensive set of input validation routines for complex data such as email addresses or credit card numbers, the invocation of these routines can further improve the resilience of a web application to the introduction of common vulnerabilities.

5.2.3 Typecasting as an Implicit Defense

To quantify the extent to which typecasting of input values to primitive types might serve as a layer of defense against cross-site scripting and SQL injection vulnerabilities, we performed an analysis over vulnerability reports for a test set of web applications. Specifically, we examined the source code of these applications to determine the whether the vulnerable input has a primitive type. Where we could not directly identify the type, we looked at the modifications made to the source code to resolve the vulnerability.

We extracted all the input parameters of the applications through the approach described in Section 5.1.2. Following this approach, we were able to link 270 parameters corresponding to cross-site scripting attack vectors, and 248 parameters corresponding to SQL injection vectors to the source of of the test set of applications.[2]

Figures 5.3a and 5.3b show an overview of the types corresponding to input parameters vulnerable to cross-site scripting and SQL injection. Most of the vulnerable parameters had one of the following types: boolean, numeric, structured text, free text, enumeration, or union. Booleans can take either logical true or false values. Examples of numeric types are integers or floating point numbers. By "structured text", we mean that the parameter is a string and, additionally, there is an expected structure to the string. A real name, URL, email address, or a username in a registration form are examples of this type. In contrast, the "free text" type denotes arbitrary, unstructured strings. Input parameters corresponding to the enumeration type should only accept a finite set of values that are known in advance. Examples are genders, country names, or a select form field. Finally, a union type denotes a variable that combines multiple types (e.g., a value that should either be a numeric value or a boolean value).

Only about 20% of the input validation vulnerabilities are associated with the free text type. This means that in these cases, the application should accept an arbitrary text input. Hence, an input validation vulnerability of this type can only be prevented by sanitizing the user-supplied input.

Interestingly, 35% of the input parameters vulnerable to cross-site scripting are actually numeric, enumeration, or boolean types (including lists of values of these types), while 68% of the input parameters vulnerable to SQL injections correspond to these simple data types. Thus, the majority of

[2]Many CVE reports do not mention any attack vectors. Hence, we excluded them for this analysis.

input validation vulnerabilities for these applications could have been prevented by enforcing the validation of user-supplied input based on simple data types. We believe that the large number of parameters vulnerable to SQL injection that correspond to the numeric type is caused by the phenomenon that many web applications use integers to identify objects and use these values in the application logic that interacts with the backend database (e.g. to identify users, messages, or blog posts).

5.2.4 Input Validation as an Explicit Defense

In Section 5.2.2 we argued that a comprehensive support for input validation in popular web application frameworks can improve the resilience of a language to the introduction of cross-site scripting and SQL injection vulnerabilities. Even though developers must remember to explicitly use these validation functions for every possible input of the application, the fact that the right functions are already provided by the framework greatly simplifies the process.

To verify the extension of the support offered by common frameworks, we first need to extract the different classes of structured text responsible for most of the attack against web applications. Figures 5.4a and 5.4b show a detailed overview of which particular "structured text" is responsible for most of the cross-site scripting and SQL injection vulnerabilities. The graph shows that web applications would benefit from input validation routines that are able to sanitize complex data such as URLs, usernames, and email addresses, since these data classes are often used as attack vectors.

However, implementing them and systematically analyzing them for correctness and safety is not a simple task. Therefore, one should expect this functionality to be provided by many application frameworks. In our experiments, we analyzed 78 open source web application frameworks for several web programming languages, including: PHP, Perl, Python, Ruby, .NET, and Java[3]. These frameworks were selected on the basis of factors such as popularity as well as the size and activity level of the developer and user communities. For each framework, we classified the kinds of validation functions for complex input values that are exposed to developers.

Partial results of this classification are shown in Table 5.1. We observe that almost 20% of the frameworks we studied do not provide any validation functionality at all. In fact, of the 78 frameworks we analyzed, only 37 provided any support for validation of complex input data, though these frameworks supported a wide variety of different data types — 31 in all.[4]

Unfortunately, there is a mismatch between the set of validation functions normally provided by these frameworks and the common attack vectors

[3]The web frameworks that have been analyzed are listed in appendix A

[4]In the interests of space, in Table 5.1 we summarize only those validation functions that appeared in five or more frameworks.

Language	PHP	Perl	Python	Ruby	.NET	Java	Total
Frameworks	21	4	2	0	3	7	37 (100%)
Email	16	2	1	0	3	7	29 (78%)
Date	13	4	2	0	2	3	24 (64%)
URL	11	1	2	0	2	5	21 (57%)
Alphanumeric	10	2	1	0	1	0	14 (38%)
Phone	7	1	0	0	0	1	9 (24%)
Time	6	1	2	0	0	0	9 (24%)
Password	4	3	0	0	0	2	9 (24%)
IP Address	6	1	1	0	0	0	8 (22%)
Filename	4	2	1	0	0	0	7 (19%)
Credit card	3	0	0	0	1	3	7 (19%)

Table 5.1: Framework support for various complex input validation types across different languages.

reported in Figures 5.4b and 5.4a. For example, 43% of the frameworks do not provide any way to automatically validate URLs that are often used for cross-site scripting attacks.

5.3 Discussion

Our empirical results from Section 5.2 indicate that the implicit validation resulting from casting input data to primitive types observed in applications written in strongly-typed languages is, indeed, correlated to a decreased exposure to cross-site scripting and SQL injection vulnerabilities. Additionally, the data indicates that the availability of explicit validation functions is also correlated with a reduced count of reported vulnerabilities.

As a result, we can conclude that there is empirical evidence to support the general intuition that input validation serves as an effective layer of defense against cross-site scripting and SQL injection vulnerabilities. In fact, it is likely that the increased usage of strongly-typed languages and explicit input validation functions for web programming would have similar benefits for other classes of vulnerabilities, as well as more general software faults.

Note, however, that we observe that input validation is not a panacea for eradicating vulnerabilities in web applications. For example, a particular drawback of the explicit input validation for complex input data is that the developer is responsible for applying the appropriate validator to each and every possible user input that is not already covered by implicit typecasting. Unfortunately, this is, as operational experience has demonstrated in the

case of web application output sanitization, an arduous and error-prone task [99].

Therefore, we advocate that, analogous to the case of framework support for automatic output sanitization, web development languages and frameworks should support the automatic validation of web application inputs as an additional security measure against both cross-site scripting and SQL injection vulnerabilities, as well as other security-relevant application logic flaws.

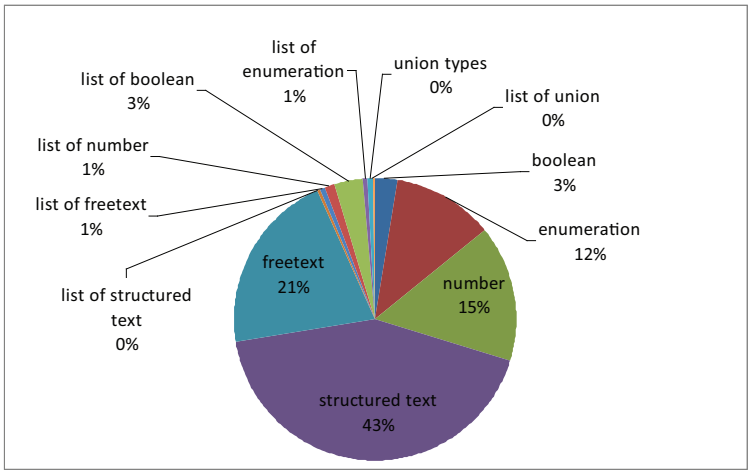
An automatic input validation policy can take several concrete forms. One such instantiation would be to enrich the type system of an appropriate strongly-typed web programming language, such that the language could infer the proper validation routines to apply for a wide variety of common input data. Another possibility would be framework support for a centralized policy description that explicitly enumerates the possible input vectors to an application, as well as the appropriate validation functions to apply. The investigation of these avenues for automatic input validation is promising research work.

5.4 Summary

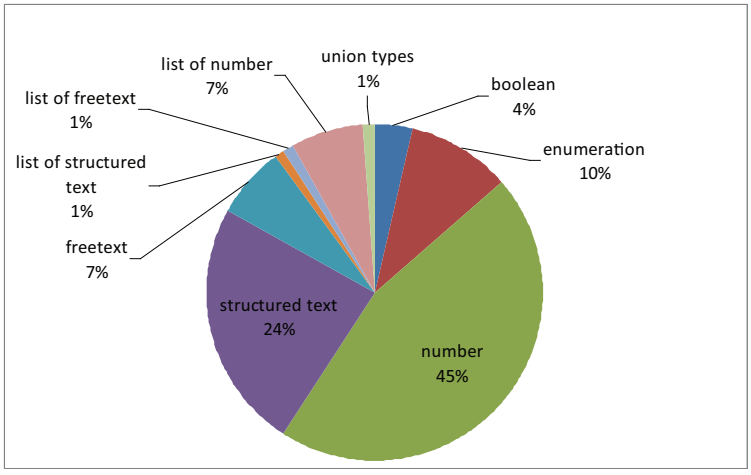
Web applications have become an important part of the daily lives of millions of users. Unfortunately, web applications are also frequently targeted by attacks such as cross-site scripting and SQL injection.

In this Chapter, we presented our empirical study of more than 10.000 web application vulnerabilities and more than 70 web application development frameworks with the aim of gaining deeper insights into how common web vulnerabilities can be prevented. In the study, we have focused on the relationship between the specific programming language used to develop web applications, and the vulnerabilities that are commonly reported.

Our findings suggest that many SQL injection and cross-site scripting could easily be prevented if web languages and frameworks would be able to automatically validate input based on common data types such as integer, boolean, and specific types of strings such as e-mails and URLs.

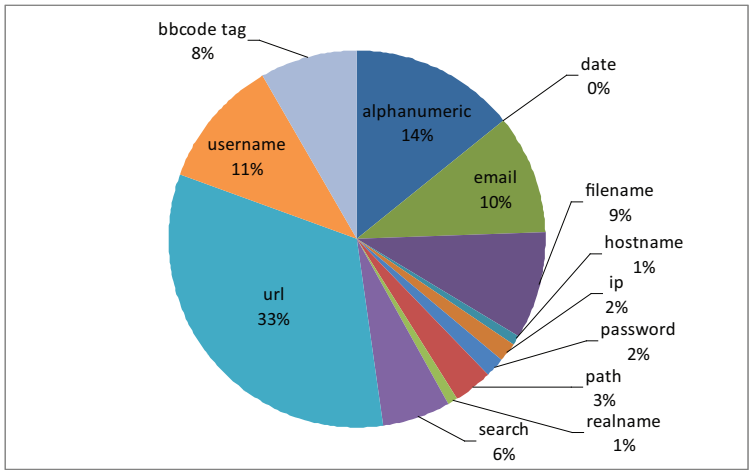


(a) cross-site scripting vulnerabilities.

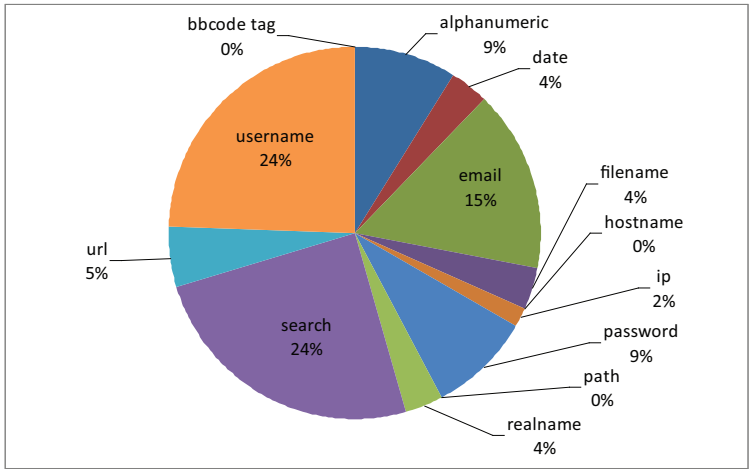


(b) SQL injection vulnerabilities.

Figure 5.3: Data types corresponding to vulnerable input parameters.



(a) cross-site scripting vulnerabilities.



(b) SQL injection vulnerabilities.

Figure 5.4: Structured string corresponding to vulnerable input parameters.

Chapter 6

Automated Prevention of Input Validation Vulnerabilities in Web Applications

In the previous chapters we have analyzed a large number of input validation vulnerability reports to understand why they are still very prevalent and how those vulnerabilities can be prevented. Application developers often fail to implement any countermeasures against those vulnerabilities and many of them can be prevented if web programming languages and frameworks would enforce the validation of user-supplied input using common data types.

In this Chapter, we present IPAAS, a novel technique for preventing the exploitation of cross-site scripting and SQL injection vulnerabilities based on automated data type detection of input parameters. IPAAS automatically and transparently augments otherwise insecure web application development environments with input validators that result in significant and tangible security improvements for real systems. Specifically, IPAAS automatically (i) extracts the parameters for a web application; (ii) learns types for each parameter by applying a combination of machine learning over training data and a simple static analysis of the application; and (iii) automatically applies robust validators for each parameter to the web application with respect to the inferred types.

IPAAS is transparent to the developer and helps therefore developers that are unaware of web application security issues to write more secure applications than they otherwise would do. Furthermore, our technique is not dependent on any specific programming language or framework. This allows IPAAS to improve the security of legacy applications and/or applications written in insecure languages. Unfortunately, due to the inherent drawbacks of input validation, IPAAS is not able to protect against all kind

```
<div class="msg">
     <h1 style="${msg.style}">${msg.title}</h1>
     <p>${msg.body}</p>
</div>
```

Figure 6.1: HTML fragment output sanitization example.

of cross-site scripting and SQL injection attacks. However, our experiments show that IPAAS is a simple and effective solution that greatly improves the security of web applications.

6.1 Preventing input validation vulnerabilities

Input validation and sanitization are related techniques for helping to ensure correct web application behavior. However, while these techniques are related, they are nevertheless distinct concepts. Sanitization — in particular, output sanitization — is widely acknowledged as the preferred mechanism for preventing the exploitation of cross-site scripting and SQL injection vulnerabilities. In this section, we highlight the advantages of input validation, and thereby motivate our approach we present in following sections.

6.1.1 Output sanitization

One particularly promising approach to preventing the exploitation of input validation vulnerabilities is robust, automated *sanitization* of untrusted input. In this approach, sanitizers are automatically applied to data computed from untrusted data immediately prior to its use in document or query construction [90, 96, 118].

As an example of output sanitization, consider the web template fragment shown in Figure 6.1. Here, untrusted input is interpolated as both child nodes of the `h1` and `p` DOM elements, as well as in the `style` attribute of the `h1` element. At a minimum, a robust output sanitizer should ensure that dangerous characters such as '`<`' and '`&`' should not appear un-escaped in the values to be interpolated, though more complex element white-listing policies could also be applied. Additionally, the output sanitizer should be *context-aware*; for instance, it should automatically recognize that '`"`' characters should also be encoded prior to interpolating untrusted data into an element attribute. The output sanitizer described here would be able to prevent attacks that might bypass input validation. For instance, an input verified to be valid might nevertheless be concatenated with dangerous characters during processing before being interpolated into a document.

Output sanitization that is automated, context-aware, and robust with respect to real browsers and databases is an extremely attractive solution to preventing cross-site scripting and SQL injection attacks. This is because it provides a high degree of assurance that the protection system's view of untrusted data used to compute documents and queries is identical to the real system's view. That is, if an output sanitizer decides that a value computed from untrusted data is safe, then it is almost certainly the case that that data is actually safe to render to the user or submit to the database.

Unfortunately, output sanitization is not a panacea. In particular, in order to achieve correctness and complete coverage of all locations where untrusted data is used to build HTML documents and SQL queries, it is necessary to construct an abstract representation of these objects in order to track output contexts. This generally requires the direct specification of documents and queries in a domain-specific language [90, 96], or else the use of a language amenable to precise static analysis. While new web applications have the option of using a secure-by-construction development framework or templating language, legacy web applications do not have this luxury. Furthermore, many web developers continue to use insecure languages and frameworks for new applications.

6.1.2 Input validation

In contrast to output sanitization, another approach to preventing cross-site scripting and SQL injection injection vulnerabilities is the use of *input validation*.

As explained in Chapter 5, input validation is fundamentally the process of ensuring that program input respects a specification of legitimate values (e.g., a certain parameter should be an integer, or an email address, or a URL). Any program that accepts untrusted input should incorporate some form of input validation procedures, or *input validators*, to ensure that the values it computes are sensible. The validation should be performed prior to executing the main logic of the program, and can vary greatly in complexity. At one end of the spectrum, programs can apply what we term *implicit validation* due to, for instance, typecasting of inputs from strings to integers in a statically-typed language. On the other hand, programs can apply *explicit validation* procedures that check whether program input satisfies complex structural specifications, such as the Luhn check for credit card numbers.

In the context of web applications, input validation should be applied to all untrusted input; this includes input vectors such as HTTP request query strings, POST bodies, database queries, XHR calls, and HTML5 postMessage invocations.

6.1.3 Discussion

Input validation is more general than output sanitization in the sense that input validation is concerned with the broader goal of program correctness, while sanitization has the specific goal of removing dangerous constructs from values computed using untrusted data. Sanitation procedures, or *sanitizers*, focus on enforcing a particular security policy, such as preventing the injection of malicious JavaScript code into an HTML document. While rigorous input validation can provide a security benefit as a side-effect, sanitizers should provide strong assurance of protection against particular classes of attacks. Input validation in isolation, on the other hand, cannot guarantee that an input it considers safe will not be transformed during subsequent processing into a dangerous value prior to being output into a document or query. Hence, input validation provides less assurance that vulnerabilities will be prevented than output sanitization.

We note, however, that despite these drawbacks, input validation has significant benefits as well. First, even though input validation is not necessarily focused on enforcing security constraints, rigorous application of robust input validators has been shown to be remarkably effective at preventing cross-site scripting and SQL injection attacks in real, vulnerable web applications. For instance, in the previous chapter, we have demonstrated that robust input validation would have been able to prevent the majority of XSS and SQL injection attacks against a large corpus of known vulnerable web applications.

Second, it is comparatively simple to achieve complete coverage of untrusted input to web applications as opposed to the case of output sanitization. Web application inputs can be enumerated given *a priori* knowledge of the language and development framework, whereas context-aware output sanitization imposes strict language requirements that often conflict with developer preferences. Consequently, input validation can be applied even when insecure legacy languages and frameworks are used.

6.2 IPAAS

In this section, we present IPAAS (**I**nput **PA**rameter **A**nalysis **S**ystem), an approach to securing web applications against cross-site scripting and SQL injection attacks using input validation. The key insight behind IPAAS is to automatically and transparently augment otherwise insecure web application development environments with input validators that result in significant and tangible security improvements for real systems.

IPAAS can be decomposed into three phases: (i) parameter extraction, (ii) type learning, and (iii) runtime enforcement. An architectural overview of IPAAS is shown in Figure 6.2. In the remainder of this section, we describe each of these phases in detail.

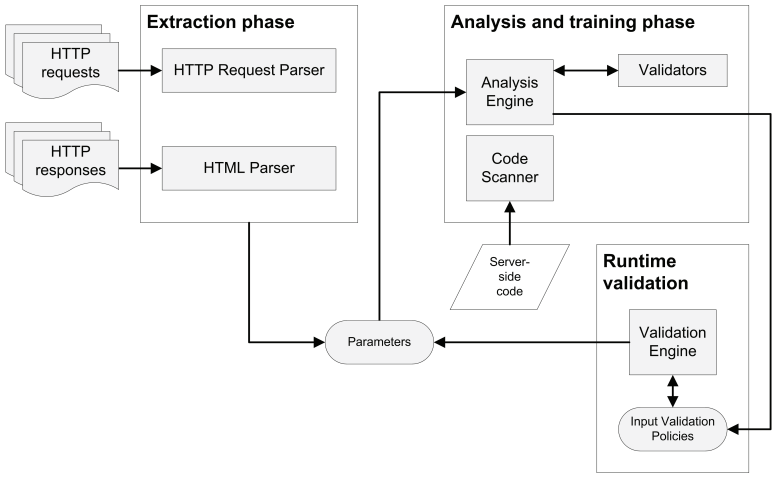


Figure 6.2: The IPAAS architecture. A proxy server intercepts HTTP messages generated during application testing. Input parameters are classified during an analysis phase according to one of a set of possible types. After sufficient data has been observed, IPAAS derives an input validation policy based on the types learned for each application input parameter. This policy is automatically enforced at runtime by rewriting the application.

6.2.1 Parameter Extraction

The first phase is essentially a data collection step. Here, a proxy server intercepts HTTP messages exchanged between a web client and the application during testing. For each request, all observed parameters are parsed into key-value pairs, associated with the requested resource, and stored in a database. Each response containing an HTML document is processed by an HTML parser that extracts links and forms that have targets associated with the application under test. For each link containing a query string, key-value pairs are extracted similarly to the case of requests. For each form, all input elements are extracted. In addition, those input elements that specify a set of possible values (e.g., `select` elements) are traversed to collect those values.

6.2.2 Parameter Analysis

The goal of the second phase is to label each parameter extracted during the first phase with a data type based on the values observed for that parameter. The labeling process is performed by applying a set of validators to the test inputs.

Type	Validator
`boolean`	`(0|1)|(true|false)|(yes|no)`
`integer`	`(+|-)?[0-9]+`
`float`	`(+|-)?[0-9]+(\+.[0-9]+)?`
`URL`	*RFC 2396, RFC 2732*
`token`	*static set of string literals*
`word`	[0-9a-zA-Z@_-]+
`words`	[0-9a-zA-Z@_- \r\n\t]+
`free-text`	*none*

Table 6.1: IPAAS types and their validators.

Validators

Validators are functions that check whether a value meets a particular set of constraints. In this phase, IPAAS applies a set of validators, each of which checks that an input belongs to one of a set of types. The set of types and regular expressions describing legitimate values are shown in Table 6.1. In addition to the types enumerated in Table 6.1, IPAAS recognizes lists of each of these types.

Analysis Engine

IPAAS determines the type of a parameter in two sub-phases. In the first, types are learned based on values that have been recorded for each parameter. In the second, the learned types are augmented using values extracted from HTML documents.

Learning In the first sub-phase, the analysis begins by retrieving all the resource paths that were visited during application testing. For each path, the algorithm retrieves the unique set of parameters and the complete set of values for each of those parameters observed during the extraction phase. Each parameter is assigned an integer score vector of length equal to the number of possible validators.

The actual type learning phase begins by passing each value of a given parameter to every possible type validator. If a validator accepts a value, the corresponding entry in that parameter's score vector is incremented by one. In the case that no validator accepts a value, then the analysis engine assigns the `free-text` type to the parameter and stops processing its values.

After all values for a parameter have been processed, the score vector is used to select a type and, therefore, a validator. Specifically, the type with the highest score in the vector is selected. If there is a tie, then the most restrictive type is assigned; this corresponds to the ordering given in

Table 6.1.

The second sub-phase uses the information extracted from HTML documents. First, a check is performed to determine whether the parameter is associated with an HTML `textarea` element. If so, the parameter is immediately assigned the `free-text` type. Otherwise, the algorithm checks whether the parameter corresponds to an `input` element that is one of a checkbox, radiobutton, or `select` list. In this case, the observed set of possible values are assigned to the parameter. Moreover, if the associated element is a checkbox, a multi-valued `select`, or the name of the parameter ends with the string `[]`, the parameter is flagged as a list.

The analysis engine then derives input validation policies for each parameter. For each resource, the path is linked to the physical location of the corresponding application source file. Then, the resource parameters are grouped by input type (e.g., query string, request body, cookie) and serialized as part of an input validation policy. Finally, the policy is written to disk.

Static Analysis The learning sub-phases described above can be augmented by static analysis. In particular, IPAAS can use a simple static analysis to find parameters and application resources that were missed during the learning phase due to insufficient training data. This analysis is, of course, specific to a particular language and framework. We describe our prototype implementation of the static analysis component in Section 6.2.4.

6.2.3 Runtime Enforcement

The result of the first two phases is a set of input validation policies for each input parameter to the web application under test. The third phase occurs during deployment. At runtime, IPAAS intercepts incoming requests and checks each request against the validation policy for that resource's parameters. If a parameter value contained in a request does not meet the constraints specified by the policy, then IPAAS drops the request. Otherwise, the application continues execution.

A request may contain parameters that were not observed during the previous phases, either in the learning sub-phases or static analysis. In this case, there are two possible options. First, the request can simply be dropped. This is a conservative approach that might, on the other hand, lead to program misbehavior. Alternatively, the request can be accepted and the new parameter marked as valid. This fact could be used in a subsequent learning phase to refresh the application's input validation policies.

6.2.4 Prototype Implementation

Parameter extraction We have implemented a prototype of the IPAAS approach for PHP. Parameter extraction is performed by a custom OWASP WebScarab extension, and HTML parsing performed by jsoup. WebScarab is a client-side interceptor proxy, but this implementation choice is of course not a restriction of IPAAS. The extractor could have easily been implemented as a server-side component as well, for instance as an Apache filter.

Type learning The parameter analyzer was developed as a collection of plugins for Eclipse and makes use of standard APIs exposed by the platform, including JFace and SWT. The Java DOM API was used to read and write the XML-based input validation policy files.

Static analyzer We implemented a simple PHP static analyzer using the Eclipse PHP Development Tools (PDT). The analyzer scans PHP source code to extract the set of possible input parameters. There are many ways in which a PHP script can access input parameters. In simple PHP applications, the value of an input parameter is retrieved by accessing one of the following global arrays: `$_GET`, `$_POST`, `$_COOKIE`, or `$_REQUEST`. However, in more complex applications, these global arrays are wrapped by special library functions that are specific to each web application.

In order to collect input parameters for PHP, our static analyzer performs pattern matching against source code and records the name of input parameters. The location of the name of an input parameter can be specified in a pattern. A pattern can be specified as a piece of PHP code and is attached to one or more input vectors (e.g., `$_GET`). For example, the pattern `optional_param('$', '*')` specifies a pattern that we used to extract input parameters from the source code of the Moodle web application. The analyzer makes a best-effort attempt to find all occurrences of function invocations of `optional_param` having two parameters. The value in the first argument is recorded, and the second argument is a "don't care" that is ignored. The analyzer can capture the names of input parameters in a similar way when the input parameter is accessed via an array.

To perform the pattern matching itself, the analyzer transforms the pattern and the PHP script to be analyzed into an abstract syntax tree (AST). Then, the static analyzer tries to match the pattern AST against the AST for the PHP script. For each match found in the source code, the analyzer then traverses the script's control flow graph (CFG) to check whether the match is reachable from the entry point of the script. For example, when an `optional_param` function invocation is observed, the analyzer checks whether a potential call chain exists from the invocation site to the script entry point. CFG traversal is recursive, including inclusions of other PHP files using the `require` and `include` statements.

Application	PHP Files	Lines of Code
Joomla 1.5	450	128930
Moodle 1.6.1	1352	365357
Mybb 1.0	152	42989
PunBB 1.2.11	70	17374
Wordpress 1.5	125	29957

Table 6.2: PHP applications used in our experiments.

Runtime enforcement The runtime component is implemented as a PHP wrapper that is executed prior to invoking a PHP script using PHP's `autoprepend` mechanism. The PHP XMLReader library is used to parse input validation policies. The validation script checks the contents of all possible input vectors using the validation routines corresponding to each parameter's learned type.

6.2.5 Discussion

The IPAAS approach has the desirable property that, as opposed to automated output sanitization, it can be applied to virtually any language or development framework. IPAAS is can be deployed in an automated and transparent way such that the developer need not be aware that their application has been augmented with more rigorous input validation. While the potential for false positives does exist, our evaluation results in Section 6.3 suggest that this would not be a major problem in practice.

However, our current implementation of IPAAS has a number of limitations. First, type learning can fail in the presence of custom query string formats. In this case, the IPAAS parameter extractor might not be able to reliably parse parameter key-value pairs.

Second, the prototype implementation of the static analyzer is fairly rudimentary. For instance, it cannot infer parameter names from variables or function invocations. Therefore, if an AST pattern is matched and the argument that is to be recorded is a non-terminal (e.g., a variable or function invocation), then the parameter name cannot be identified. In these cases, the location of the function invocation is stored along with a flag indicating that an input parameter was accessed in a dynamic way. This allows the developer the opportunity to identify the names of the input parameters manually after the analyzer has terminated, if desired.

Parameter Type	Joomla		Moodle		MyBB		PunBB		Wordpress		Total	
	xss	sqli	xss	sqli	xss	sqli	xss	sqli	xss	sqli	xss	sqli
`word`	2	4	5	10	11	14	16	2	5	0	39 (36%)	30 (25%)
`integer`	1	7	0	28	6	23	6	3	4	2	17 (16%)	63 (53%)
`free-text`	3	2	4	0	5	1	4	0	13	0	29 (27%)	3 (3%)
`boolean`	1	0	0	1	1	4	5	0	0	0	7 (6%)	5 (4%)
`enumeration`	1	2	0	0	3	8	1	2	0	1	5 (5%)	13 (11%)
`words`	2	1	0	1	0	0	2	0	1	0	5 (5%)	2 (2%)
`URL`	0	0	0	0	1	0	1	0	3	0	5 (5%)	0 (0%)
`list`	0	0	0	1	1	2	1	1	0	0	2 (2%)	4 (5%)
Total	10	16	9	41	28	52	36	8	26	3	109	120

Table 6.3: Manually identified data types of vulnerable parameters in five large web applications.

6.3 Evaluation

To assess the effectiveness of our approach in preventing input validation vulnerabilities, we tested our IPAAS prototype on five real-world web applications shown in Table 6.2. Each application is written in PHP, and the versions we tested contain many known, previously-reported cross-site scripting and SQL injection vulnerabilities.

To run our prototype, we created a development environment by importing each application as a project in Eclipse version 3.7 (Indigo) with PHP Development Tools (PDT) version 3.0 installed.

6.3.1 Vulnerabilities

Before starting our evaluation, we extracted the list of vulnerable parameters for reach application by analyzing the vulnerability reports stored in the Common Vulnerabilities and Exposures (CVE) database hosted by NIST [78]. For each extracted parameter, we manually verified the existence of the vulnerability in the corresponding application. In addition, we manually determined the data type of the vulnerable parameter.

Table 6.3 summarizes the results of the manual analysis, and shows, for each web application, the number of vulnerable parameters having a particular data type. The dataset resulting from this analysis contains 109 cross-site scripting and 120 SQL injection vulnerable parameters.

According to Table 6.3, more than half of the SQL injections are associated with integer parameters, while the majority of the cross-site scripting vulnerabilities are exploited through the use of parameters of type `word`. Interestingly, only a relatively small number of vulnerabilities are caused by

Parameter Type	Joomla		Moodle		MyBB		PunBB		Wordpress		Total	
	xss	sqli	xss	sqli	xss	sqli	xss	sqli	xss	sqli	xss	sqli
`word`	2	4	1	0	5	7	12	1	1	0	21 (19%)	12 (10%)
`integer`	1	6	0	2	2	8	5	1	3	0	11 (10%)	17 (14%)
`free-text`	3	2	1	0	4	0	2	0	10	0	20 (18%)	2 (2%)
`boolean`	1	0	0	0	1^1	3^3	4	0	1^1	1^1	7 (6%)	4 (3%)
`enumeration`	1	2	0	0	1	3	1	2	0	1	3 (3%)	8 (6%)
`words`	2	0	0	0	0	0	1	0	0	0	3 (3%)	0 (0%)
`URL`	0	0	0	0	1	0	0	0	1	0	2 (2%)	0 (0%)
`list`	0	0	0	0	0	1	0	0	0	0	0 (0%)	1 (1%)
`unknown`	0	0	2	0	2	1	1	0	1	0	6 (6%)	1 (1%)
Correctly Identified	10	14	4	2	15	20	26	4	16	1	71 (65%)	41 (34%)
Wrongly Identified	-	-	-	-	1	3	-	-	1	1	2 (1.8%)	4 (3.3%)

(*) number reported as superscript indicate the parameters identified with an incorrect type.

Table 6.4: Typing of vulnerable parameters in five large web applications before static analysis.

`free-text` or similarly unconstrained parameters. This supports our hypothesis that IPAAS can be used in practice to automatically prevent the majority of input validation vulnerabilities.

6.3.2 Automated Parameter Analysis

In order to automatically label parameters with types, IPAAS requires a training set containing examples of benign requests submitted to the web application. We collected this input data by manually exercising the web application and providing valid data for each parameter.

The results of our automated analysis are summarized in Table 6.4. For each application, the table reports the number of vulnerable parameters having a particular type. The results show that less than half of the parameters could be identified automatically. For most, our system was able to assign the correct type. However, in a few cases, the parameter was part of a request or serialized in a response, but had no value assigned to it. Hence, the type could not be identified. These parameters are reported as having type `unknown`.

Finally, IPAAS wrongly assigned the type `boolean` instead of `integer` to two cross-site scripting and four SQL injection vulnerable parameters. These misclassifications are caused by the overlap between `boolean` and `integer` validators. In fact, parameters having values of "0" and "1" can be considered of type `boolean` as well as `integer` (i.e., if only the values "0"

Type	Joomla		Moodle		MyBB		PunBB		Wordpress		Total	
	xss	sqli	xss	sqli	xss	sqli	xss	sqli	xss	sqli	xss	sqli
Detected by static analysis	3	9	6	40	28	46	24	8	23	1	94 (86%)	104 (87
Missed during type analysis	0	2	2	37	10	18	10	4	6	0	28 (26%)	61(51

Table 6.5: Results of analyzing the code.

Type	Joomla		Moodle		MyBB		PunBB		Wordpress		Total	
	xss	sqli	xss	sqli	xss	sqli	xss	sqli	xss	sqli	xss	sqli
`word`	2	4	4	7	10	10	15	1	5	0	36 (33%)	22 (18%)
`integer`	1	7	0	25	6	21	5	3	4	2	16 (15%)	58 (48%)
`free-text`	3	2	3	0	4	1	2	0	10	0	22 (20%)	3 (3%)
`boolean`	1	0	0	1	1	3	4	0	0	0	6 (6%)	4 (3%)
`enumeration`	1	2	0	0	3	8	1	2	0	1	5 (5%)	13 (11%)
`words`	2	1	0	1	0	0	2	0	1	0	5 (5%)	2 (2%)
`URL`	0	0	0	0	1	0	0	0	2	0	3 (3%)	0 (0%)
`list`	0	0	0	0	0	1	0	0	0	0	0 (0%)	1 (1%)
`unknown`	0	0	0	0	1	0	1	0	0	0	2 (2%)	0 (0%)
Total	10	16	7	34	26	44	30	6	22	3	95 (87%)	103 (86%)

Table 6.6: Typing of vulnerable parameters in five large web applications after static analysis.

and "1" are observed during training, the analysis engine gives priority to the type `boolean`). Collecting more data for each parameter by exercising the same functionality of a web application multiple times can result in different values for the same parameter. Hence, collecting more training data would increase the probability that our algorithm makes the correct classification.

6.3.3 Static Analyzer

To improve the detection ratio of the vulnerable parameters, we ran our static analyzer on the source code of each application.

Table 6.5 shows the number of vulnerable parameters that were identified with the help of the static analyzer. The tool was able to find 86% of the cross-site scripting and 87% of the SQL injection affected parameters. By comparing these input parameters with the ones that were detected by the analysis engine, we see that 26% of the cross-site scripting and 51% of the SQL injection affected parameters were missed by the analysis engine, but were found by the static analyzer. Hence, the static analyzer component can help in achieving a larger coverage of the type analysis, and, thus, help

Application	Vulnerabilities		Prevented Vulnerabilities	
	xss	sql	xss	sqli
Joomla	10	16	7 (70%)	14 (88%)
Moodle	9	41	4 (44%)	34 (83%)
MyBB	28	52	21 (75%)	43 (83%)
PunBB	36	8	27 (75%)	6 (75%)
Wordpress	26	3	12 (46%)	3 (100%)
Total	109	120	71 (65%)	100 (83%)

Table 6.7: The number of prevented vulnerabilities in various large web applications.

prevent a larger number of vulnerabilities.

Based on these results, we collected more input data by testing the functionality of each web application using the data from the static analyzer. Then, we ran IPAAS again to determine the data types of the newly discovered parameters, and we manually verified whether the types were correctly identified. The results are shown in Table 6.6. In this case, we obtained better coverage, with 87% of cross-site scripting and 86% of SQL injection affected parameters being properly identified. In addition, none of the parameters were misclassified.

Although the static analyzer helps significantly in achieving a higher coverage, a few parameters were still missed during analysis. This problem could be improved by employing a more precise static analysis. Also, we believe that unit testing might serve as an additional source of test input data to help improve IPAAS' coverage.

6.3.4 Impact

To assess the extent to which IPAAS is effective in preventing input validation vulnerabilities in practice, we manually tested whether it was still possible to exploit the aforementioned vulnerabilities while IPAAS was enabled. During our tests, we explored different ways to perform the attacks, and to evade possible sanitization and validation routines as reported by XSS and SQL cheatsheets available on the Internet.

Table 6.7 shows the number of cross-site scripting and SQL injection vulnerabilities that are prevented by IPAAS. We observe that most of the SQL injection vulnerabilities and a large fraction of cross-site scripting vulnerabilities became impossible to exploit with the input validation policies that were automatically extracted in our last experiment in place.

The results of this analysis are consistent with our observation that the majority of input validation vulnerabilities on the web can be prevented

by labeling the parameter with a data type that properly constrains the range of legitimate values. If a parameter is assigned to an unknown or unrestricted type such as `free-text`, our system will still accept arbitrary input. In these cases, the vulnerability is not prevented by our system.

The difference in the number of prevented cross-site scripting and SQL injection vulnerabilities is mainly due to the relatively large number of `integer` parameters that are vulnerable to SQL injection, while many cross-site scripting vulnerabilities are due to injections in `free-text` parameters. We believe that the large number of parameters vulnerable to SQL injection that correspond to the type `integer` is caused by the phenomenon that web applications frequently use integers to identify records.

6.4 Summary

Web applications are popular targets on the Internet, and well-known vulnerabilities such as cross-site scripting and SQL injection are, unfortunately, still prevalent. Current mitigation techniques for cross-site scripting and SQL injection vulnerabilities mainly focus on some aspect of automated output sanitization. In many cases, these techniques come with a large runtime overhead, lack precision, or require invasive modifications to the client or server infrastructure.

In this Chapter, we identified automated input validation as an effective alternative to output sanitization for preventing cross-site scripting and SQL injection vulnerabilities in legacy applications, or where developers choose to use insecure legacy languages and frameworks. We presented the IPAAS approach, which improves the secure development of web applications by transparently learning types for web application parameters during testing, and automatically applying robust validators for these parameters at runtime.

The evaluation of our implementation for PHP demonstrates that IPAAS can automatically protect real-world applications against the majority of cross-site scripting and SQL injection vulnerabilities with a low false positive rate. As IPAAS ensures the complete and correct validation of all input to the web application, IPAAS can in principle prevent other classes of input validation vulnerabilities as well. These classes include Directory Traversal, HTTP Response Splitting and HTTP Parameter Pollution.

Chapter 7

Conclusion and Future Work

7.1 Summary of Contributions

To understand how to improve the security of web applications, insights into how vulnerabilities in real web applications have evolved and how they can be prevented become of interest. In this dissertation we claim that, in order to improve the security of web applications, common web vulnerabilities such as cross-site scripting and SQL injection have to be automatically prevented using input validation techniques.

Our study on the evolution of input validation vulnerabilities in web applications in Chapter 4 demonstrates that these classes of vulnerabilities are still very prevalent. Measurements on a large number of input validation vulnerabilities provided insights into whether developers are better at writing secure web applications today than they used to be in the past. The results of this study suggest that web developers are still failing to implement existing defense mechanisms against input validation vulnerabilities and that there is a need for techniques that secure web applications *by design*. That is, techniques that prevent these classes of vulnerabilities automatically and work transparently to developers.

The tools, web frameworks and programming languages that are used to develop a web application have a direct impact on how the system is to be created and, as such, also determines the means that are necessary to secure the resulting application. To understand how to secure web applications, we explore in Chapter 5 the relationship between the web programming language used to develop a web application and vulnerabilities that are commonly reported. In particular, we identify and quantify the importance of typing mechanisms in web programming languages and frameworks for the security of web applications.

We used the insights from the empirical studies presented in Chapters 4 and 5, to build a system that improves the secure development of web applications by automatically and transparently augmenting web development

environments with robust input validators. We show that this approach improves the security of real web applications significantly.

This thesis made the following contributions:

7.1.1 Evolution of Web Vulnerabilities

Covering more than 10.000 vulnerabilities published between 2004 and 2009, we perform an automated analysis to understand whether an increased developers' attention has improved the security of web applications over the years. We measured the complexity of vulnerabilities and exploits, the popularity of vulnerable applications, the lifetimes of vulnerabilities and we identified trends. Measurements on the complexity of exploits show no significant changes which suggests that vulnerabilities are caused by an absence of sanitization routines rather than insufficiently implemented sanitization routines. We show that more vulnerabilities are reported about popular applications. We observe that the number of web vulnerability reports is not decreasing because many more applications written by different developers are vulnerable. Furthermore, we show that more vulnerabilities are reported about applications that are popular than about applications that are not popular. Finally, the lifetimes of vulnerabilities suggest that it is harder to find cross-site scripting vulnerabilities than SQL injection vulnerabilities.

The empirical data we collected and analyzed suggest that developers consistently fail to implement existing countermeasures against input validation vulnerabilities. We believe that this can be attributed mainly to informational (*developers are not aware*) or motivational (*developers do not care*) factors. Because of these reasons, we believe that the traditional approach of writing code and then testing for security does not work. Hence, we believe that there is a need for novel techniques that secure web applications *by design*. Application developers should not have to take care of implementing sanitization functionality while writing secure web applications; rather the programming language or platform should automatically do this.

7.1.2 Input Validation as Defense Mechanism

Measurements on the source code of 28 applications prone to 809 vulnerabilities provided insights into whether input validation can be an effective defense mechanism against common web vulnerabilities such as cross-site scripting and SQL injection. Input validation may prevent an attacker from injecting scripting code (i.e. a string) into a parameter which is normally used by an application to store - for example - an integer. Hence, input validation can potentially prevent many vulnerabilities. We determined the data type of more than 500 vulnerable parameters and we identified that

only 20 % of the vulnerable input parameter have to accept arbitrary text input. While these vulnerabilities can only be prevented using output sanitization, the remaining 80 % of vulnerabilities can be prevented using validators that enforce data types such as numeric, enumeration, boolean types or specific types of strings. Furthermore, an analysis based on 78 web application frameworks showed us that there is a mismatch between the set of validation functions normally provided by these frameworks and commonly reported attack vectors. Hence, the use of validation functionality provided by existing frameworks is not sufficient to develop secure web applications.

To conclude, our findings suggest that the majority of SQL injection vulnerabilities and a significant fraction of cross-site scripting vulnerabilities can be prevented if web applications would automatically perform input validation based on common data types such as integer, boolean, and specific types of strings such as e-mails and URLs.

7.1.3 Input Parameter Analysis System

We proposed IPAAS, a novel technique that prevents the exploitation of cross-site scripting and SQL injection vulnerabilities based on input validation. IPAAS automatically detects types of input parameters and transparently augments web platforms with input validators during the development of web applications.

We identify that input validation is not a panacea for preventing all cross-site scripting and SQL injection vulnerabilities for two reasons. First, web applications may have to accept arbitrary input in certain cases. Second, input validation in isolation cannot guarantee that input, which is considered as safe, will be not transformed to a dangerous value by application processing before outputting it in a web document or database query.

Although IPAAS is based on input validation, IPAAS has significant advantages. As the technique is transparent to developers, it helps developers that are unware of security issues to write more secure web applications than they otherwise would do. The IPAAS approach does not require the use of specific programming languages or frameworks. Consequently, the approach can be used to secure legacy and novel applications written in insecure languages.

We implemented IPAAS for PHP and tested it on five real web applications. Our evaluation using real attack data shows that IPAAS - despite the drawbacks inherent to input validation - is remarkably effective in preventing cross-site scripting and SQL injection vulnerabilities in real web applications. Therefore, we can conclude that during the implementation phase of a web application, IPAAS improves the security of the application significantly. Thus, by using IPAAS, web applications become more secure *by design*.

7.2 Critical Assessment

Despite the automated and manual analysis of a large number of vulnerabilities, our studies have some limitations:

- *Static lexical matching.* The vulnerability classification is partially based based on static lexical matching. Furthermore, this technique is also used for our analysis on the complexity of vulnerabilities. Identifying and analyzing vulnerabilities by searching for combinations of keywords may result into false positives or false negatives due to two factors. First, there is the possibility of having vulnerability reports containing semantically equivalent but lexical divergent expressions for the same concepts. Second, languages do change over time: neologisms may be accepted and after a period of time these are succeeded by other terms. Both factors skew data and may impact the results of our analysis.

  To get an understanding of the impact on our analysis, we measured the number of cross-site scripting and SQL injection vulnerabilities that are incorrectly classified as cross-site scripting or SQL injection vulnerabilities (e.g. the false positive rate). The false positive rate was measured by manually analyzing a sample of 50 vulnerabilities for each year between 2002 and 2009. In total, we manually analyzed 358 cross-site scripting vulnerabilities and 324 SQL injection vulnerabilities, representing 6.6 % and 6.7 % of the cross-site scripting and SQL injection classified vulnerabilities, respectively.

  In the sample, we found 10 false positives for cross-site scripting vulnerabilities (2,8 %) and 7 false positives for SQL injection (2,2 %). When manually analyzing the vulnerabilities, we did not find any evidence that the keywords we searched for during the vulnerability complexity analysis were used in other contexts than in the contexts we were looking for. Thus, our vulnerability classification and complexity analysis did not result in a significant number of false positives.

  Since the NVD dataset is a large dataset, we were unable to perform a manual analysis to measure the false negative rate – that is, the number of cross-site scripting or SQL injection vulnerabilities that were not classified as such. We expect that false positives are likely to occur more often in earlier years than in recent years as the language to describe cross-site scripting and SQL injection vulnerabilities becomes more mature over the years.

- *Availability of information.* Depending on the disclosure model that is used, vulnerabilities may or may not be disclosed. Even if the vulnerability is disclosed, not all information concerning the vulnerability is published. For example, with *responsible disclosure*, a software vendor

may decide not to publish the information that is needed for an attacker to exploit a vulnerability. In these cases, we were neither able to measure the exploit complexity nor able to analyze the data type of the affected input parameters.

Furthermore, there is more data openly available than what we used for our measurements on exploit complexity. For an analysis of this scale, we could only consider exploit data that we could process automatically. We collected this data from different Security Information Providers who did not always provide the data in a parsable format.

To summarize, the sometimes limited availability of information and the use of non-standard formats for publishing vulnerability and exploit information skew data and may impact the results of our analysis.

7.3 Future Work

In this dissertation, we have looked at vulnerabilities that occur on the server-side of the web application. However, client-side code may also contain vulnerabilities. Examples of client-side scripting vulnerabilities include DOM-based cross-site scripting, code injection and open-redirect vulnerabilities. With the advent of Web 2.0, an increasing number of web applications critically depend on client-side code written in JavaScript, Flash, Flex or Silverlight to provide a richer user experience.

These client-side technologies are under active development. For example, web browsers have started to implement the HTML5 standard, the next major version of HTML. HTML5 provides new language features such as `<video>`, `<audio>` and `<canvas>` elements as well as functionality to integrate Scalable Vector Graphics. These new features provide new attack vectors for traditional attacks such as script injection. Furthermore, HTML5 offers new APIs to allow, among other things, cross-document/channel communication, cross-origin resource sharing and client-side persistent storage. The insecure usage of these APIs introduce new security issues that can be exploited by hackers. Examples include as data leakage through cross-directory attacks on the Web Storage API and the execution of malicious code from foreign domains via the Cross-Document Messaging API.

We believe that due to the growing popularity of rich client-side functionality, web applications will become increasingly vulnerable to client-side scripting vulnerabilities. As client-side code process sensitive data, vulnerabilities in client-side code can be critical and attacks may have a serious impact. Therefore, we believe that future work should address this class of vulnerabilities.

Understanding the details of client-side vulnerabilities and attacks is a prerequisite for the design and implementation of secure web applications.

To gain deeper insights into these vulnerabilities and to create awareness, we believe that large scale measurements need to be performed. To improve the security of web applications, we believe that future work should focus on automated detection techniques to find client-side vulnerabilities. In addition to these techniques, we are also interested in if and how languages and frameworks can be improved to prevent client-side vulnerabilities *by design*.

Chapter 8

French Summary

8.1 Résumé

L'internet est devenu un environnement omniprésent dans le monde du travail et du loisir. Des millions de personnes utilisent l'Internet et les applications web tous les jours pour s'informer, effectuer des achats, se divertir, socialiser et communiquer. La popularité sans cesse croissante des applications web ainsi que des services associés entraînent l'exécution de nombreuses transactions critiques, qui soulèvent des questions de sécurité. Du fait de cette croissance, des efforts ont été entrepris durant cette derni'ere décennie pour rendre les applications web plus sûres. Des organisations ont travaillé sur l'importance de faire comprendre aux utilisateurs les enjeux de sécurité. Les chercheurs en sécurité se sont concentrés sur l'élaboration d'outils et de techniques pour améliorer la sécurité des applications web. Malgré ces efforts, de récents rapports provenant de l'institut SANS estiment que plus de 60 % des attaques commises sur l'Internet ciblent les applications web en se concentrant sur les vulnérabilités inhérentes aux problèmes de validation, comme le Cross-Site Scripting (XSS) ou les injections SQL (SQLi).

Dans cette thèse, nous avons conduit deux études de recherche empirique, analysant un grand nombre d'application web vulnérables. Le but est de comprendre les mécanismes complexes entraînant des vulnérabilités introduites par une mauvaise ou une absence de validation, et ainsi de retracer l'évolution de ces dernières lors de la dernière décennie écoulée. Nous avons assemblé une base de données contenant plus de 10.000 rapports de vulnérabilités depuis l'an 2000. Ensuite, nous avons analysé ces donnèes pour déterminer si les développeurs ont pris conscience des problématiques de sécurité web de nos jours, comparé à la période où ces applications émergeaient. Puis nous avons analysé l'étroit lien entre le langage de programmation utilisé pour développer l'application web et le nombre de vulnérabilité reporté. Nos analyses montrent que la complexité des attaques n'a pas drastiquement évolué, et que de nombreuses attaques simples subsistent. Nos études

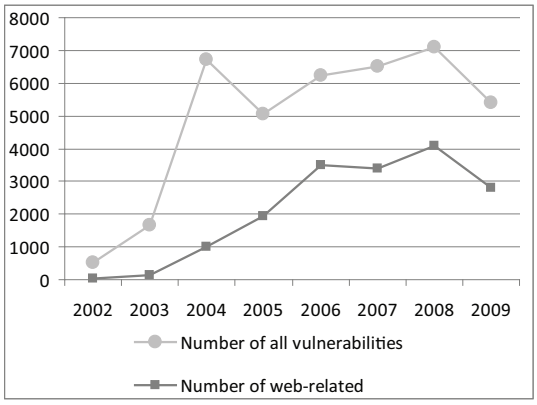


FIGURE 8.1: Nombre de vulnérabilités web au fil du temps, les données obtenues de NVD CVE [78].

montrent aussi que la plupart des injections SQL et des XSS peuvent être évitées en utilisant un mécanisme de validation basé sur un 'common data type'.

Avec ces résultats empiriques comme base, nous présentons notre solution IPAAS qui aide les développeurs novice en termes de sécurité à écrire des applications sécurisées par défaut. La solution propose une technique novatrice qui prévient l'exploitation des injections SQL et des XSS en se basant sur la détection automatique du typage des paramètres d'entrée. Nous montrons par ailleurs que cette technique améliore de manière probante la sécurité des applications web.

8.2 Introduction

Au fil des années, le World Wide Web a attiré de nombreux utilisateurs malveillants et les attaques contre les applications Web sont devenues prévalentes. Récents rapports provenant de l'institut SANS estiment que plus de 60 % des attaques commises sur l'Internet ciblent les applications web [23]. La situation précaire sur le Web peut être attribuée à plusieurs facteurs.

Tout d'abord, le nombre de vulnérabilités dans les applications Web a augmenté au fil des ans. La figure 8.1 montre le nombre de toutes les vulnérabilités par rapport au nombre de vulnérabilités liées au web qui ont été publiés entre 2000 et 2009 dans les Common Vulnerabilities and Exposures (CVE) Liste [78]. Nous observons que, de 2006, plus de la moitié des vulnérabilités signalées sont vulnérabilités liées au web. La situation ne s'est pas améliorée ces dernières années. Basé sur une analyse de 3000 sites web réalisés en 2010, un site Web contenait en moyenne 230 vulnérabilités selon un rapport de sécurité WhiteHat [105]. Bien que pas tous les vulnérabilités

Records	Date	Organizations
130.000.000	2009-01-20	Heartland Payment Systems, Tower Federal Credit Union, Beverly National Bank
94.000.000	2007-01-17	TJX Companies Inc.
90.000.000	1984-06-01	TRW, Sears Roebuck
77.000.000	2011-04-26	Sony Corporation
40.000.000	2005-06-19	CardSystems, Visa, MasterCard, American Express
40.000.000	2011-12-26	Tianya
35.000.000	2011-07-28	SK Communications, Nate, Cyworld
35.000.000	2011-11-10	Steam (Valve, Inc.)
32.000.000	2009-12-14	RockYou Inc.
26.500.000	2006-05-22	U.S. Department of Veterans Affairs

Table 8.1: Les plus grandes fuites de données en termes de documents divulgués selon [31].

web posent un risque pour la sécurité, de nombreuses vulnérabilités sont exploitées par des attaquants afin de compromettre l'intégrité, la disponibilité ou la confidentialité d'une application web.

Deuxièmement, les attaquants ont une large gamme d'outils à leur disposition pour trouver des vulnérabilités Web et lancer des attaques contre les applications Web. Les fonctionnalités avancées de Google Search permet à des attaquants de trouver des failles de sécurité dans le code de configuration et la programmation de sites Web. Ceci est également connu comme Google Hacking [12, 72]. De plus, il existe un large éventail d'outils et de cadres disponibles qui permettent aux attaquants de lancer des attaques contre les applications Web. Très connu dans ce contexte est le framework Metasploit [85]. Ce cadre s'appuie sur le plus grand modulaire base de données du monde de la qualité des exploits assurée, y compris des centaines d'exploits à distance, modules auxiliaires, et des charges utiles.

Finalement, les attaquants ont des motivations pour effectuer des attaques contre les applications Web. Ces attaques peuvent entraîner dans, entre autres choses, les fuites de données, usurper l'identité d'utilisateurs innocents et les infections de logiciels malveillants à grande échelle.

De plus en plus applications Web stockent et traitent des données sensibles telles que les informations d'identification utilisateur, les dossiers de compte et numéros de carte de crédit. Des vulnérabilités dans applications Web peuvent se produire sous la forme de violations de données qui permettent attaquants pour recueillir cette information sensible. L'attaquant peut utiliser cette information pour le vol d'identité ou il peut le vendre sur le marché clandestin. Voler de grandes quantités d'informations d'identification et de les vendre sur le marché clandestin peut être rentable pour

un attaquant comme le montre par plusieurs études [124, 13, 39, 104]. Les chercheurs en sécurité estiment que vol de numéros de cartes de crédit peut être vendu pour un prix compris entre $ 2 à $ 20 chacun [13, 39]. Pour les comptes bancaires de la gamme de prix par article situe entre $ 10 et $ 1000 tandis que pour les mots de passe e-mail l'intervalle est $ 4 à $ 30 [39] par article.

Applications Web légitimes qui sont les plus vulnérables peuvent être compromis par des attaquants afin d'installer des logiciels malveillants sur l'hôte de la victime dans le cadre d'un *drive-by-download* [84]. Le malware installé peut prendre le contrôle complet de la machine de la victime et l'agresseur utilise le logiciel malveillant de réaliser des profits financiers. En règle générale, les logiciels malveillants est utilisé à des fins telles que agissant comme un nœud botnet, la récolte des informations sensibles de la machine de la victime, ou d'effectuer d'autres actions malveillantes qui peuventêtre monétisés. Web malveillants est activement négociées sur le marché clandestin [124]. Bien qu'aucune évaluation de certaines existent sur le montant total de l'argent des attaquants gagnent avec la négociation d'actifs virtuels, tels que les logiciels malveillants sur le marché clandestin, certaines activités ont été analysées. Une étude réalisée par McAfee [64] montre que les machines compromises sont vendus comme des serveurs proxy anonymes sur le marché souterrain pour un prix compris entre $ 35 et $ 550 par mois en fonction des caractéristiques de la procuration.

Les attaques contre les applications Web sur la disponibilité, l'intégrité et la confidentialité des applications Web et les données qu'ils traitent. Parce que notre société dépend en grande partie sur les applications web, les attaques contre les applications Web constituent une menace sérieuse. Le nombre croissant d'applications web qui traitent des données de plus en plus sensibles a rendu la situation encore pire. Tableau 8.1 montre des rapports sur les grandes catastrophes en termes d'enregistrements de données exposées dans les dernières années. Alors que le Web n'est pas la principale source de violations de données, il représente encore 11 % des violations de données qui est la deuxième place. Bien qu'aucun chiffres globaux existent sur la perte annuelle causée par les fuites de données sur le Web, les coûts de certaines violations de données ont été estimées. En 2011, environ 77 millions de comptes utilisateurs sur le réseau Sony Playstation ont été compromis par un attaque injection de SQL sur deux propriétés de Sony. Sony a estimé qu'il serait passé 171.1 millions de dollars pour faire face à la violation de données [102].

Pour résumer, l'état actuel d'insécurité du Web peut être attribuée à la prévalence de vulnérabilités web, les outils sont disponibles pour les exploiter et les motivations (financières) des attaquants. Malheureusement, la popularité croissante du Web rendra la situation encore pire. Il va motiver les attaquants plus que les attaques peuvent potentiellement affecter un plus

grand nombre d'utilisateurs innocents qui en résultent dans plus de profits pour les attaquants. La situation doit être améliorée, car les conséquences des attentats sont dramatiques en termes de pertes financières et les efforts nécessaires pour réparer les dégâts.

Pour améliorer la sécurité sur le Web, beaucoup d'efforts ont été dépensés dans la dernière décennie pour rendre les applications Web plus sécurisé. Des organisations comme MITRE [62], SANS Institute [23] et OWASP [79] ont souligné l'importance de l'amélioration de la formation à la sécurité et la sensibilisation parmi les programmeurs, les clients logiciels, les gestionnaires et les agents logiciels principaux de l'information. En outre, la communauté de recherche en sécurité a travaillé sur les outils et techniques pour améliorer la sécurité des applications Web. Ces outils et techniques se concentrent principalement sur les deux réduire le nombre de vulnérabilités dans les applications ou sur la prévention de l'exploitation des vulnérabilités.

Bien qu'une quantité considérable d'efforts ont été dépensés par de nombreux intervenants différents pour rendre les applications Web plus sécurisé, nous manquons de preuves quantitatives de savoir si cette attention a amélioré la sécurité des applications web. Dans cette thèse, nous étudions comment les vulnérabilités web ont évolué dans la dernière décennie. Nous nous concentrons dans cette thèse sur SQL injection et les vulnérabilités cross-site scripting comme ces classes de vulnérabilités des applications Web ont la même cause : la validation des données fourni par l'utilisateur. De plus, ces classes de vulnérabilités sont répandus, bien connu et ont été bien étudiée dans la dernière décennie.

Nous observons que, malgré les programmes de sensibilisation à la sécurité et les outils, les développeurs web échouent systématiquement à mettre en œuvre des contre-mesures existantes qui se traduit dans les applications web vulnérables. De plus, l'approche traditionnelle de l'écriture de code et tester ensuite pour la sécurité ne semble pas bien fonctionner. Par conséquent, nous croyons qu'il ya un besoin de techniques qui assurent des applications web par la conception. Ce sont des techniques qui rendent les applications Web sécurisé automatiquement sans compter sur le développeur web. L'application de ces techniques à grande échelle devrait améliorer sensiblement la situation sécuritaire sur le Web.

8.2.1 Problématiques de recherche

Les sections précédentes montre que les applications Web sont fréquemment la cible par des attaquants et donc des solutions sont nécessaires qui aident à améliorer la situation sécuritaire sur le Web. Comprendre comment les vulnérabilités web commun peuvent être automatiquement empêché, c'est le défi principal de recherche dans ce travail.

Dans cette thèse, nous abordons les problèmes de recherche suivantes en ce qui concerne la sécurité des applications Web :

- *Les développeurs créent des applications Web plus sécurisées aujourd'hui qu'ilshabitude de faire dans le passé ?*

  Dans la dernière décennie, beaucoup d'efforts ont été dépensés par de nombreux intervenants différents pour rendre les applications Web plus sécurisé. À ce jour, il n'existe aucune preuve empirique disponible si cette attention a amélioré la sécurité des applications web. Pour obtenir un aperçu plus approfondi, nous effectuons une analyse automatisée sur un grand nombre de rapports de vulnérabilité cross-site scripting et SQL injection. En particulier, nous sommes intéressés à en savoir si les développeurs sont plus conscients des problèmes de sécurité web aujourd'hui qu'auparavant d'être dans le passé.

- *Le langage de programmation utilisé pour développer une application web influence sur l'exposition de ces applications à des vulnérabilités ?*

  Les langages de programmation contiennent souvent des caractéristiques qui aident les programmeurs pour empêcher des bogues ou des vulnérabilités liées à la sécurité. Ces caractéristiques comprennent, entre autres, les systèmes de type statique, des espaces de noms restreints et de la programmation modulaire. Il n'existe aucune preuve, aujourd'hui, si certaines caractéristiques des langages de programmation Web aider à atténuer les vulnérabilités de validation d'entrée. Dans notre travail, nous effectuons une analyse quantitative dans le but de comprendre si certains langages de programmation sont intrinsèquement plus robuste contre l'exploitation des vulnérabilités de validation d'entrée que les autres.

- *La validation d'entrée d'un mécanisme de défense efficace contre les vulnérabilités web courantes ?*

  Pour concevoir et implémenter des applications Web sécurisées, une bonne compréhension des vulnérabilités et des attaques est une condition préalable. Nous étudions un grand nombre de rapports de vulnérabilité et les référentiels de code source d'un nombre important d'applications web vulnérables, dans le but d'acquérir une compréhension plus profonde de la façon dont les vulnérabilités web commun peuvent être évités. Nous allons analyser si les mécanismes de typage dans un langage et des fonctions de validation d'entrée dans un cadre d'application web peut potentiellement prévenir les vulnérabilités web de nombreux. Aucune preuve empirique est disponible dès aujourd'hui ce spectacle dans lequel étendre ces mécanismes sont en mesure de prévenir les vulnérabilités web.

- *Comment pouvons-nous aider les développeurs d'applications, qui ne connaissent pas de problèmes de sécurité des applications web, à écrire des applications Web plus sécurisées ?*

Les résultats de nos études empiriques donnent à penser que de nombreux développeurs d'applications web ne sont pas conscients des questions de sécurité et qu'un nombre significatif de vulnérabilités web peut être évitée en utilisant de simples mécanismes de validation directe. Nous présentons un système qui apprend que les types de données de paramètres d'entrée lorsque les développeurs d'écrire des applications Web. Ce système est en mesure de prévenir de nombreuses vulnérabilités web communs en augmentant automatiquement autrement vulnérables environnements de développement Web avec les validateurs d'entrée robustes.

8.3 L'évolution des vulnérabilités de validation d'entrée dans les applications Web

Des efforts considérables ont été consacrés par de nombreux intervenants différents pour rendre les applications Web plus sécurisé. Cependant, nous manquons de preuves quantitatives que cette attention a amélioré la sécurité des applications Web au fil du temps. Dans cette section, nous étudions comment les classes ordinaires de vulnérabilités web ont évolué dans la dernière décennie. En particulier, nous sommes intéressés à en savoir si les développeurs sont mieux à la création d'applications Web sécurisées aujourd'hui qu'ils habitude d'être dans le passé. Nous mesurons la complexité exploit pour comprendre si des vulnérabilités exigent des scénarios d'attaque aujourd'hui plus complexes à exploiter les vulnérabilités qui sont le résultat de la contre-mesures insuffisantes. Par ailleurs, nous étudions comment les applications individuelles sont exposées à des vulnérabilités. Nous examinons si les applications web les plus populaires sont les plus exposés à des vulnérabilités que les applications non-populaires. Par mesurer la durée de vie des vulnérabilités dans les applications, nous essayons d'obtenir une compréhension de la difficulté de trouver des vulnérabilités.

Notre étude se concentre sur les vulnérabilités cross-site scripting et SQL injection comme ces classes de vulnérabilités des applications Web ont la même cause : désinfection incorrecte des données fourni par l'utilisateur qui résulte de hypothèses invalides par le développeur. De plus, ces classes de vulnérabilités sont répandus, bien connu et ont été bien étudiée dans la dernière décennie. Ainsi, il est probable qu'il y ait un nombre suffisant de rapports de vulnérabilité disponibles pour permettre une analyse empirique.

8.3.1 Méthodologie

Pour pourra répondre comment vulnérabilités cross-site scripting et SQL injection ont évolué au fil du temps, il est nécessaire d'avoir accès à d'importantes quantités de données sur les vulnérabilités. Par conséquent, nous

avons eu à recueillir et à classer un grand nombre de rapports de vulnérabilité. Par ailleurs, il est nécessaire d'extraire les descriptions d'exploits dans les rapports automatisé. Dans les sections suivantes, nous expliquons le processus que nous s'applique à collecter et classer les rapports de vulnérabilité et d'exploiter descriptions.

Collecter des données

Une source majeure d'information pour les failles de sécurité est le jeu de données CVE, qui est hébergé par MITRE [67]. Selon FAQ MITRE [69], CVE n'est pas une base de données de vulnérabilité, mais une identification de la vulnérabilité système qui 'vise à fournir des noms communs pour les problèmes connus du public qui permet' et qui permet 'bases de données de vulnérabilité et des capacités d'autres d'être reliés entre eux'. Chaque entrée possède un identifiant CVE unique, un statut ('entry' où 'candidate'), une description générale, et un certain nombre de références à une ou sources d'information les plus externes de la vulnérabilité. Ces références comprennent un identificateur de source et un identifiant de bien défini pour la recherche sur les la source de site Web.

Pour notre étude, nous avons utilisé les données de la National Vulnerability Database (NVD) [78] qui est fourni par le National Institute of Standards and Technology (NIST). Le NIST publie la base de données NVD comme un ensemble de fichiers XML, sous la forme :`nvdcve-2.0-year.xml`, lorsque l'année est un nombre à partir de 2002 jusqu'en 2010. Le premier fichier, `nvdcve-2.0-2002.xml` contient des entrées CVE à partir de 1998 jusqu'en 2002. Afin de construire des délais lors de l'analyse, nous avons besoin de connaître la date de la découverte, date de la divulgation, ou la date de publication d'une entrée de CVE. Depuis entrées CVE proviennent de différentes sources externes, les informations de synchronisation fourni dans les données CVE et NVD nourrit se sont avérées insuffisantes. Pour cette raison, nous chercher ces informations en utilisant la date de divulgation de l'entrée correspondante dans la Open Source Vulnerability Database (OSVDB) [55].

Pour chaque entrée CVE, nous avons extrait et stocké l'identifiant, la description, la date de divulgation de OSVDB, la vulnérabilité de classification (CWE), la notation CVSS, vendeur / produit / informations sur la version, et les références au sources extèrnes. Puis, nous avons utilisé les références de chaque entrée CVE pour récupérer l'information sur la vulnérabilité provenant de l'externe diverses sources. Nous avons stocké ces données du site avec les informations CVE pour une analyse ultérieure.

Classification des Vulnérabilités

Depuis notre étude se concentre particulièrement sur des vulnérabilités de cross-site scripting et SQL injection, il est essentiel de classer le les rapports de vulnérabilité. Comme mentionné dans la section précédente, les entrées CVE dans la base de données NVD sont classés selon le système de classification Faiblesse énumération commune. CWE vise à être un dictionnaire des faiblesses des logiciels, il ya des entrées pour cross-site scripting (CWE-79) et pour SQL injection (CWE-89).

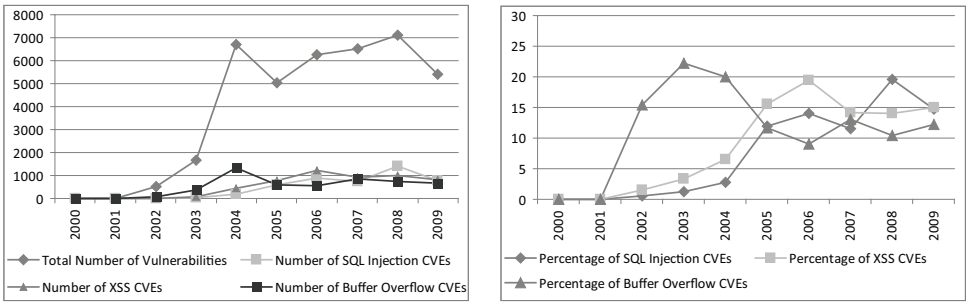
Bien que NVD fournit un mappage entre CVEs et CWES, cette cartographie est pas complète et de nombreuses entrées CVE n'ont pas de classification du tout. Pour cette raison, nous avons choisi d'effectuer une classification qui est basée à la fois sur la classification CWE et sur la description de l'entrée CVE. En général, nous avons observé que la description CVE est formaté selon le schéma suivant : {description de la vulnérabilité} {description de l'emplacement de la vulnérabilité} *allows* {description de l'attaquant} {description des impacts}. Ainsi, la description CVE comprend le type de vulnérabilité.

Pour obtenir les CVEs cross-site scripting à des données CVE, nous avons sélectionné les CVEs associé à l'identifiant CWE 'CWE-79'. Puis, nous avons ajouté les CVEs avec le texte cross-site scripting dans leur description en effectuant une requête insensible à la casse. De même, nous avons classé les CVEs SQL injection connexes en utilisant l'identifiant CWE 'CWE-89' et le mot-clé 'SQL injection'.

Les données des Exploits

Pour acquérir une vision générale sur la sécurité des applications Web, nous ne sommes pas seulement intéressés par l'information la vulnérabilité, mais aussi dans la façon dont chaque vulnérabilité peut être exploitée. Certaines sources externes de CVEs qui fournissent des informations concernant cross-site scripting ou SQL injection liées vulnérabilités fournissent également des détails exploit. Souvent, cette information est représentée par un script ou un *attack string*. An 'attack string' est une référence bien défini à un emplacement dans le application web vulnérable où le code peut être injecté. La référence est souvent une URL complète qui inclut le nom du script vulnérable, les paramètres HTTP, et certains caractères pour représenter les espaces réservés pour le code injecté. En plus d'utiliser des espaces réservés, parfois, des exemples réels de SQL ou du code Javascript peut également être utilisé. Deux exemples 'attack strings' sont les suivants :

```
http://[victim]/index.php?act=delete&dir=&file=[XSS]
http://[victim]/index.php?module=subjects&func=viewpage&pageid=[SQL]
```



(a) Les tendances des vulnérabilités dans des nombres

(b) Les tendances des vulnérabilités dans des pourcentages

FIGURE 8.2: Les Buffer overflow, cross-site scripting and SQL injection vulnérabilités au fil du temps.

A la fin de chaque ligne, notez les espaces réservés qui peuvent être substitués avec un code arbitraire par l'attaquant.

La structure similaire des 'attack strings' permet à notre outil pour extraire automatiquement, stocker et analyser le format exploit. Par conséquent, nous extraites et stockées toutes les 'attack strings' associés aux CVEs cross-site scripting et SQL injection.

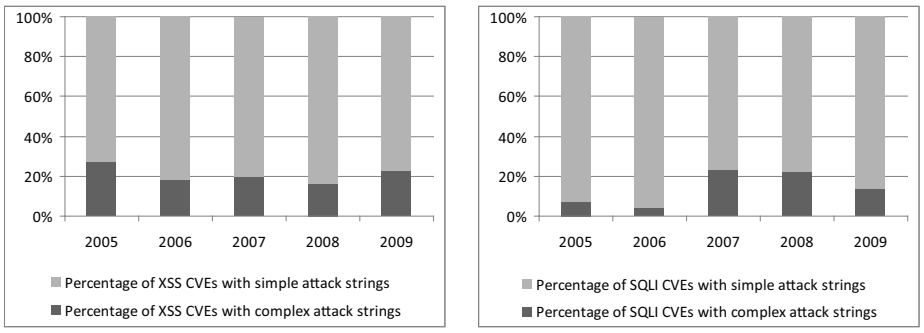
8.3.2 L'analyse des tendances vulnérabilités

La première question que nous souhaitons aborder dans cette étude est de savoir si le nombre des vulnérabilités de SQL injection et cross-site scripting signalés dans les applications web a été diminué ces dernières années. Pour répondre à cette question, nous avons automatiquement analysé les 39,081 entrées dans la base de données NVD 1998 à 2009[1]. Nous avons dû exclure 1,301 entrées CVE parce qu'ils n'ont pas de résultat correspondant de la base de données OSVDB et, en conséquence, fait pas de date de divulgation qui leur sont associés. Pour cette raison, ces entrées CVE ne sont pas pris en compte pour le reste de notre étude. Parmi les autres rapports de vulnérabilité, nous avons identifié un total de 5349 vulnérabilités de 'buffer overflow', 5413 vulnérabilités de 'cross-site scripting' et 4825 vulnérabilités de 'SQL injection'.

Figure 8.2a indique le nombre de rapports de vulnérabilité au fil du temps et de la figure 8.2b montre le pourcentage de vulnérabilités signalées au cours des entrées totales CVE.

Notre première espérance basée sur l'intuition a été de constater que le

[1] Au moment de notre étude, un ensemble de données de vulnérabilité complète de 2010 n'était pas disponible. Par conséquent, notre étude ne couvre pas 2010.



(a) Cross-site scripting (b) SQL injection

FIGURE 8.3: La complexité des exploits au fil du temps.

nombre de vulnérabilités signalées suivre une forme de cloche classique, en commençant par un démarrage lent lorsque les vulnérabilités sont encore relativement inconnue, puis une augmentation abrupte correspondant à la période dans laquelle les attaques sont décrits et étudiés, et enfin une phase diminue lorsque les développeurs commencent à adopter des contre-mesures nécessaires. En fait, les graphiques montrent une phase initiale (2002-2004) avec très peu de rapports sur les vulnérabilités de cross-site scripting et SQL injection et de nombreux rapports sur les vulnérabilités de buffer overflow. Notez que cette tendance est cohérente avec l'évolution historique. La sécurité Web a commencé à augmenter en importance après 2004, et le premier 'worm' basé sur cross-site scripting a été découvert en 2005 (Samy 'worm' [71]). Par conséquent, les menaces de sécurité Web, tels que cross-site scripting et SQL injection a commencé à recevoir plus d'attention après 2004 et, dans l'intervalle, ces menaces ont dépassé les problèmes de buffer overflow. Malheureusement, le nombre des vulnérabilités de SQL injection et cross-site scripting rapporté n'a pas sensiblement diminué depuis 2006. En d'autres termes, le nombre des vulnérabilités cross-site scripting et SQL injection trouvé en 2009 est comparable avec le nombre rapporté en 2006. Dans le reste de cette section, nous allons formuler et de vérifier un certain nombre d'hypothèses pour expliquer les raisons qui pourraient expliquer ce phénomène.

Sophistication attaque

Hypothèse 1 *Simple, facile à trouver vulnérabilités ont maintenant été remplacé par des vulnérabilités complexes qui exigent plus sophistiqué attaques.*

La première hypothèse que nous souhaitons vérifier est de savoir si le

nombre total de vulnérabilités ne diminue pas parce que les vulnérabilités découvertes simples dans les premières années ont été remplacés par de nouveaux qui impliquent des scénarios d'attaque plus complexes.

En particulier, nous sommes intéressés à trouver si la complexité des exploits a augmenté. Notre but en faisant cela est d'identifier les cas dans lesquels les développeurs d'applications étaient au courant des menaces, mais mis en œuvre insuffisantes, faciles à échapper à des routines de désinfection. Dans ces cas, un attaquant doit élaborer l'entrée malveillante avec plus de soin ou doit effectuer des transformations d'entrée de certains (par exemple, en majuscules ou en substitution de caractère).

Une façon de déterminer la complexité d'un exploit est d'analyser le 'attack string' et de chercher des preuves de techniques d'évasion possibles. Comme mentionné dans la section 8.3.1, nous extraire automatiquement le code d'exploitation à partir des données fournies par des sources externes d'information de vulnérabilité. Parfois, ces sources externes ne fournissent pas d'exploiter l'information pour tous rapportés vulnérabilité cross-site scripting ou SQL injection, ne fournissent pas d'exploiter l'information dans un format analysable, ou ne fournissent pas toute l'information exploit du tout. En conséquence, toutes les entrées ne peuvent être CVE associée à une *'attack string'*. D'autre part, dans certains cas, il existe plusieurs façons d'exploiter une vulnérabilité, et, par conséquent, de nombreux *'attack strings'* peuvent être associé à un rapport sur la vulnérabilité unique. Dans nos expériences, nous avons recueilli des 'attack strings' pour un total de 2632 vulnérabilités distinctes.

Pour déterminer la complexité exploit, nous avons examiné plusieurs caractéristiques qui peuvent indiquer une tentative de l'attaquant de se soustraire à une certaine forme de désinfection. La sélection des caractéristiques est inspiré par ce qu'on appelle des feuilles de triche injection qui sont disponibles sur l'Internet [63][95].

En particulier, nous classons une cross-site scripting 'attack string' aussi complexe (en la différence de simples) si elle contient une ou plusieurs de ce qui suit caractéristiques :

- Différents cas sont utilisés dans les balises de script (`ScRiPt`)
- Les balises de script contiennent un ou plusieurs espaces (`< script>`)
- La 'attack string' contient 'landingspace-code', qui est l'ensemble des attributs de HTML-tags (`onmouseover` ou `onclick` par exemple)
- La chaîne contient des caractères codés (`)`)
- La chaîne est répartie sur plusieurs lignes

Pour les 'attack strings' de SQL injection, nous avons examiné les caractéristiques suivantes :

- L'utilisation de prescripteurs commentaire (`/**/`) pour briser une mot-clé.
- L'utilisation de guillemets simples codées ('`%27`', '`&#x27`' ; '`&#39`', '`Jw==`')
- L'utilisation de guillemets doubles codées ('`%22`', '`"`', '`&#34`', '`Ig==`')

Si aucune des caractéristiques précédentes est présente, nous classons l'exploitation comme 'simple'.

Les figures 8.3a et 8.3b montrent le pourcentage des CVEs ayant une ou plusieurs 'attack string' complexes[2]. Les graphiques montrent que la majorité des exploits disponibles sont, selon notre définition, pas sophistiqué. En fait, dans la plupart des cas, les attaques ont été effectuées en injectant la 'attack string' la plus simple possible, sans nécessiter de trucs pour échapper à la validation des entrées.

On observe une légère augmentation du nombre de vulnérabilités SQL injection avec des 'attack strings' sophistiqués, mais nous n'observons pas la même pour les 'attack strings' cross-site scripting. C'est peut-être une première indication que les développeurs sont en train d'adopter (malheureusement insuffisante) mécanismes de défense pour empêcher SQL injection, mais que n'ils sont pas arrive toujours à adopter des mécanismes de défense contre les vulnérabilités cross-site scripting.

Application et la vie de vulnérabilité

Dans cette étude, nous avons déterminé que une constante, un grand nombre de simples, faciles à exploiter les vulnérabilités se trouvent encore dans beaucoup d'applications web d'aujourd'hui. Selon ces résultats, nous formulons une seconde hypothèse :

Hypothèse 2 *Même si le nombre de cas déclarés est de plus en plus les applications vulnérables, chaque application est de plus en plus sûr au fil du temps.*

Cette hypothèse est importante, parce que, si elle est vraie, cela signifierait que les applications Web (les produits bien connus en particulier) sont de plus en plus sécurisé. Pour vérifier cette hypothèse, nous avons étudié la durée ou la durée de vie des vulnérabilités cross-site scripting et SQL injection. Nous avons étudié les durées de vie des vulnérabilités cross-site scripting et SQL injection dans les dix plus affectés applications open source en fonction de la base de données du NIST NVD. Pendant une analyse des journaux de modifications pour chaque application, nous avons extrait la version dans

[2]Le graphe commence à partir de 2005 parce qu'il y avait moins de 100 vulnérabilités ayant des échantillons d'exploits disponibles avant cette année. Donc, les résultats avant 2005 sont statistiquement moins significative.

	Fondamentale	Non-Fondamentale
bugzilla	4	7
drupal	0	22
joomla	5	4
mediawiki	3	21
mybb	9	2
phorum	3	5
phpbb	4	2
phpmyadmin	14	13
squirrelmail	10	4
wordpress	6	9
Total	58	89

(a) Cross-site scripting

	Fondamentale	Non-Fondamentale
bugzilla	1	8
coppermine	1	3
e107	0	3
joomla	4	0
moodle	0	3
mybb	9	3
phorum	0	4
phpbb	3	0
punbb	4	2
wordpress	0	4
Total	22	30

(b) SQL injection

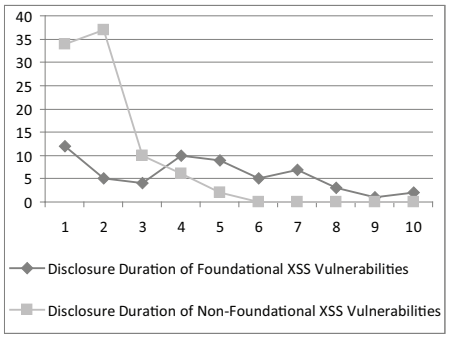
TABLE 8.2: Vulnérabilités fondamentales et non-fondamentales dans les dix les plus touchés ouverts applications Web source.

laquelle une vulnérabilité a été introduit et la version dans laquelle une vulnérabilité a été fixé. Afin d'obtenir des données fiables sur la durée de vie de la vulnérabilité, nous exclus les rapports de vulnérabilité qui n'ont pas été confirmées par le fournisseur concerné. Pour notre analyse, nous avons utilisé les identifiants CPE dans la base de données NVD, les sources de vulnérabilité externe, les informations vulnérabilité fournie par le vendeur. Nous avons également extrait les informations des systèmes de contrôle de version (CVS, SVN ou) des différents produits.

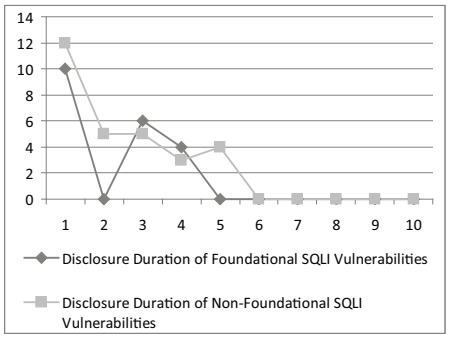
Tableau 8.2a et le tableau 8.2b montrent un total de 147 vulnérabilités de cross-site scripting et 52 vulnérabilités de SQL injection dans les applications les plus affectées. Les tableaux distinguent entre les vulnérabilités *fondamentale* et *non-fondamentale*. Foundational vulnerabilities are vulnerabilities that were present in the first version of an application, tandis que les non-fondamentaux vulnérabilités ont été introduites après la version initiale.

Nous avons observé que 39% des vulnérabilités cross-site scripting sont fondamentale et 61% sont non-fondatmentale. Pour SQL injection, ces pourcentages sont respectivement de 42% et 58%. Ces résultats suggèrent que la plupart des vulnérabilités sont introduites par une nouvelle fonctionnalité qui est intégrée dans les nouvelles versions d'une application web.

Enfin, nous avons étudié pendant combien de temps qu'il a fallu pour découvrir les vulnérabilités. Figure 8.4a et la figure 8.4b parcelle le nombre de vulnérabilités qui ont été divulgués après un certain laps de temps s'est écoulé après la publication initiale des applications. Les graphiques montrent que la plupart des vulnérabilités SQL injection sont généralement découvertes dans les premières années après la sortie du produit. Pour les vulnérabilités cross-site scripting, le résultat est différent. De nombreux vulnéra-



(a) Cross-site scripting



(b) SQL injection

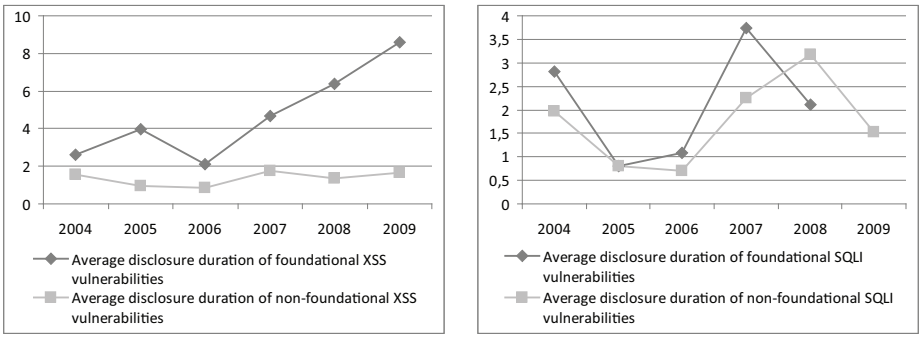
FIGURE 8.4: Temps écoulé entre la version du logiciel et à la divulgation de vulnérabilité au cours des années.

bilités fondamentales sont communiqués même 10 ans après le code a été initialement publié. Cette observation suggère que c'est très problématique pour trouver des vulnérabilités cross-site scripting fondamentale par rapport à des vulnérabilités SQL injection. Cette hypothèse est étayée par le fait que le temps moyen écoulé entre la version du logiciel et de la divulgation des vulnérabilités fondamentales est de 2 ans pour les vulnérabilités SQL injection, tandis que pour les vulnérabilités cross-site scripting cette valeur est de 4,33 ans.

8.3.3 Discussion

Nos résultats de cette étude montrent que la complexité des attaques cross-site scripting et SQL injection liées à des vulnérabilités dans la base de données NVD n'a pas cessé d'augmenter. Ni les conditions préalables à des attaques, ni la complexité des exploits ont changé de manière significative. Par conséquent, ce résultat suggère que la majorité des vulnérabilités ne sont pas dus à l'échec de désinfection, mais plutôt en raison de l'absence de validation. Malgré les programmes de sensibilisation fournis par MITRE [67], SANS Institute [23] et OWASP [79], les développeurs d'applications ne sont pas encore mise en œuvre de contre-mesures efficaces.

Les données empiriques que nous avons recueillies et analysées pour cette étude prend en charge l'intuition générale que les développeurs web échouent systématiquement à sécuriser leurs applications. La pratique traditionnelle de l'écriture d'applications, puis de les tester pour les problèmes de sécurité (par exemple, l'analyse statique, les tests blackbox, etc) ne semble pas bien fonctionner dans la pratique. Par conséquent, nous croyons que davantage de recherches sont nécessaires dans la sécurisation des applications de par leur



(a) Cross-site scripting (b) SQL injection

FIGURE 8.5: La durée moyenne de divulgation des vulnérabilités dans les années au fil du temps.

conception. Autrement dit, les développeurs ne devraient pas être concernés par des problèmes tels que xss ou sql. Au contraire, le langage de programmation ou de la plate-forme devrait veiller à ce que les problèmes ne se produisent pas lorsque les développeurs produire du code (par exemple, semblable à des solutions telles que dans [90] ou gérés langages tels que C# ou Java qui empêchent les problèmes comme 'buffer overflow'.

8.4 Les mécanismes pour valider l'entrée des données dans les applications Web et des Langues

Pour développer une application Web, des outils et des langages de programmation sont nécessaires. Le langage de programmation choisi pour développer une application a un effet direct sur la façon dont un système doit être créé et les moyens qui sont nécessaires pour s'assurer que l'application résultante se comporte comme prévu et est sécurisé. Dans cette section, nous étudions la relation entre le langage de programmation et des vulnérabilités Web qui sont fréquemment rapportés.

Une propriété importante d'un langage de programmation est le système de typage qui est utilisé. Un système de typage classe des instructions de programme et des expressions selon les valeurs qu'ils peuvent calculer, et il est utile pour statiquement raisonnement sur les comportements possibles du programme. Certains langages du web populaires tels que PHP et Perl sont faiblement typés, ce qui signifie que la langue convertit implicitement les valeurs quand on opère avec des expressions d'un type différent.

L'avantage des langues faiblement typés du point un développeur web de vue, c'est qu'ils sont souvent faciles à apprendre et à utiliser. De plus, ils

permettent aux développeurs de créer des applications plus rapidement qu'ils n'ont pas à se soucier de déclarer les types de données pour les paramètres d'entrée d'une application web. Par conséquent, la plupart des paramètres sont considérés comme génériques "strings", même si elles pourraient en fait représenter une valeur entière, un booléen, ou un ensemble de caractères spécifiques (par exemple, une adresse e-mail). En conséquence, les attaques sont souvent possible que si la validation est mauvaise. Par exemple, un attaquant pourrait injecter code de script (c'est à dire un string) dans la valeur d'un paramètre qui est normalement utilisé par l'application pour stocker un nombre entier.

Afin d'obtenir une compréhension plus profonde des raisons de vulnérabilités dans les applications Web, nous analysons dans ce chapitre autour de 3933 cross-site scripting et 3,758 SQL injection vulnérabilités affectant des applications écrites en langues populaires tels que PHP, Python, ASP et Java. Pour plus de 800 de ces vulnérabilités, nous avons manuellement extraire et d'analyser le code responsable de la gestion de l'entrée, et déterminé le type du paramètre affecté (par exemple, booléen, entier, ou une 'string').

8.4.1 Méthodologie

Nous avons analysé le code source d'un nombre important de web vulnérables applications dans le but de comprendre dans quelle mesure à typage et mécanismes de validation pourrait aider à prévenir vulnérabilités de cross-site scripting et de SQL injection. Afin d'obtenir un ensemble de test d'applications avec un nombre élevé de paramètres d'entrée vulnérables, nous avons choisi de concentrer notre étude sur 20 applications open source populaires web PHP qui contenaient la plus forte incidence des vulnérabilités cross-site scripting, et le 20 avec la plus forte incidence de vulnérabilités SQL injection. Les 28 applications appartenant à deux, en grande partie recouvrement, les ensembles sont les suivants : claroline, coppermine, deluxebb, drupal, e107, horde, jetbox, joomla, mambo, mantis, mediawiki, moodle, mybb, myblog-gie, papoo, phorum, phpbb, phpfusion, phpmyadmin, pligg, punbb, runcms, serendipity, squirrelmail, typo3, webspell, wordpress, et xoops.

Pour chacune de ces applications, nous avons manuellement examiné le correspondant la vulnérabilité rapports pour identifier la version de l'application spécifique et toute par exemple des entrées d'attaque. Compte tenu de cette information, nous avons téléchargé le code source de chaque application et reliés les vecteurs d'entrée en code source de la demande de pour déterminer une typage de données approprié. Nous avons répété ce processus pour un total de 809 rapports de vulnérabilité.

Au établit un lien entre rapports de vulnérabilité au code source, nous avons d'abord utilisé la version du code source qui a été connu pour être vulnérable. Puis, nous avons répété le processus et établit le lien entre les

```
POST /payment/submit HTTP/1.1
Host: example.com
Cookie: SESSION=cbb8587c63971b8e
[...]

cc=1234567812345678&month=8&year=2012&
save=false&token=006bf047a6c97356
```

FIGURE 8.6: Exemple de requête HTTP.

rapports de vulnérabilité du code source dans laquelle les vulnérabilités ont été corrigées. Pour déterminer le type de données du paramètre d'entrée vulnérables, nous avons manuellement analysé la façon dont chaque vulnérabilité a été corrigé et comment la valeur du paramètre d'entrée a été utilisé tout au long de l'application Web.

8.4.2 Analyse

Un mécanisme de défense qui est essentiel pour le bon fonctionnement des applications est la *validation des entrées*. Dans l'abstrait, de validation d'entrée est le processus d'attribution de la signification sémantique aux intrants non structurées et non fiable à une application, et veiller à ce que ces entrées de respecter un ensemble de contraintes décrivant une entrée bien formé. Pour les applications Web, les entrées prennent la forme de paires clé-valeur des 'strings'. La validation de ces entrées peut être effectuée soit en le navigateur en utilisant Javascript, ou sur le serveur. Comme il n'existe actuellement aucune garantie de l'intégrité de calcul dans le navigateur, la validation des entrées relevant de la sécurité doit être effectuée sur le serveur, et, par conséquent, nous limitons notre discussion sur la validation des entrées dans ce contexte.

Pour élucider le processus de validation d'entrée pour les applications web côté serveur, examiner la demande HTTP pédagogique illustré à la figure 8.6. Cette figure montre une structure typique pour une demande de soumission de paiement à une fiction application e-commerce. Dans le cadre de cette demande, il ya plusieurs paramètres d'entrée que la logique de commande pour `/payment/submit` doit gérer : `cc`, un numéro de carte de crédit ; `month`, un mois numérique ; `year`, une année numérique ; `save`, un indicateur indiquant si les informations de paiement doivent être enregistrés pour une utilisation future ; `token`, une token anti-CSRF ; et `SESSION`, un identificateur de session. Chacun de ces paramètres de la requête nécessite un autre type de validation d'entrée. Par exemple, le numéro de carte de crédit devrait être un certain nombre de caractères et de passer un test de Luhn. Le paramètre mois devrait être une valeur entière entre 1 et 12 in-

clus. Le paramètre année devrait aussi être une valeur entière, mais peut varier de l'année en cours à une année d'arbitraire dans un proche avenir. L'indicateur de sauvegarde doit être une valeur booléenne, mais comme il ya différentes représentations de la vraie logique et fausse, (par exemple, $\{\text{true}, \text{false}\}$, $\{1, 0\}$, $\{\text{yes}, \text{no}\}$), la demande doit toujours reconnaître un ensemble fixe de valeurs possibles.

La validation des entrées, en plus de son rôle dans la facilitation de la correction du programme, est un outil utile pour empêcher l'introduction de vulnérabilités dans les applications web. Y at-il un attaquant de fournir la valeur :

```
year=2012'; INSERT INTO admins(user, passwd)
 VALUES('foo', 'bar');--
```

à notre fiction application e-commerce dans le cadre d'une vulnérabilité SQL injection pour élever ses privilèges, de validation d'entrée appropriée serait de reconnaître que la valeur malveillant n'était pas une année valide, avec le résultat que la demande serait refuser de traiter la demande.

La validation des entrées peut se produire de multiples façons. La validation peut être effectuée implicitement — par exemple, par typecasting une 'string' à une type primitif, comme un booléen ou nombre entier. Pour l'attaque par exemple montré ci-dessus, un casting de la 'input string' à un nombre entier se traduirait par une erreur de cast de l'exécution, puisque la valeur malveillant n'est pas un entier bien formé. D'autre part, de validation d'entrée peut être exécutée de façon explicite, en invoquant des routines de validation fournis par le cadre. La validation explicite est généralement effectuée pour les valeurs d'entrée présentant la structure complexe, comme les adresses e-mail, URL, ou numéros de carte de crédit. à cet égard, le choix du langage de programmation et un cadre pour développer des applications Web joue un rôle important dans la sécurité de ces applications. Tout d'abord, si une langue dispose d'un système de type fort de telle sorte que transtypages de valeurs d'entrée mal formées à certains types primitifs se traduira par des erreurs d'exécution, la langue peut constituer une défense implicite contre l'introduction de vulnérabilités comme cross-site scripting et SQL injection. Deuxièmement, si un cadre pour les langues fournit un ensemble complet de routines de validation d'entrée pour les données complexes telles que les adresses e-mail ou numéros de carte de crédit, l'invocation de ces routines peuvent en outre améliorer la résilience d'une application web à l'introduction de vulnérabilités courantes.

Typecasting comme une défense implicite

Afin de quantifier la mesure dans laquelle transtypage de valeurs d'entrée à des types primitifs pourraient servir comme une couche de défense contre les vulnérabilités cross-site scripting et SQL injection, nous avons effectué

une analyse sur les rapports de vulnérabilité pour un ensemble de test d'applications web. Particulièrement, nous avons examiné le code source de ces applications afin de déterminer si l'entrée de la vulnérabilité a un type primitif. Lorsque nous ne pouvions identifier directement le type, nous avons examiné les modifications apportées au code source pour résoudre la vulnérabilité.

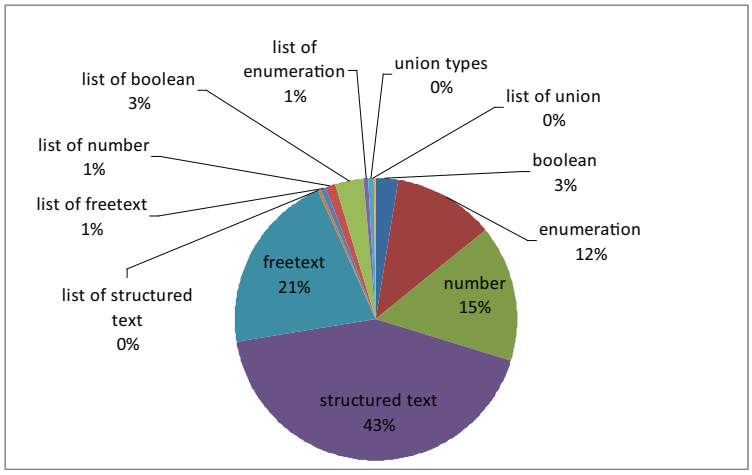
Nous avons extrait tous les paramètres d'entrée des applications grâce à l'approche décrit dans la section 8.4.1. We extracted all the input parameters of the applications through the approach described in Section 8.4.1. Following this approach, we were able to link 270 parameters corresponding to cross-site scripting attack vectors, and 248 parameters corresponding to SQL injection vectors to the source of of the test set of applications.[3]

Figures 8.7a et 8.7b montrent un aperçu des types correspondant à des paramètres d'entrée vulnérables à cross-site scripting et SQL injection. La plupart des paramètres vulnérables était l'un des types suivants : boolean, numeric, structured text, free text, enumeration, ou union. Les Booleans peuvent prendre soit logiques valeurs vraies ou fausses. Des exemples de types numériques sont des nombres entiers ou à virgule flottante. Par "structured text", nous entendons que le paramètre est une chaîne et, en outre, il ya une structure attendue à la 'string'. Un vrai nom, URL, adresse e-mail, ou un nom d'utilisateur dans un formulaire d'inscription sont des exemples de ce type. En revanche, le type "free text" texte désigne arbitraires, les 'strings' non structurées. Les paramètres d'entrée correspondant au type de recensement ne devrait accepter un ensemble fini de valeurs qui sont connus à l'avance. Les exemples sont les genres, les noms de pays, ou un champ de formulaire de sélection. Finalement, un type d'union indique une variable qui combine plusieurs types (par exemple, une valeur qui devrait être soit une valeur numérique ou une valeur booléenne).

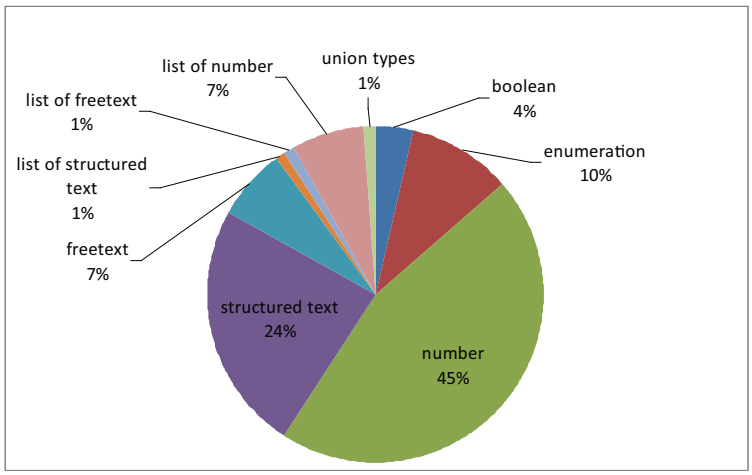
Seulement environ 20% des vulnérabilités de validation d'entrée sont associés avec le type de texte libre. Cela signifie que dans ces cas, la demande devrait accepter une entrée de texte arbitraire. Ainsi, une vulnérabilité de validation d'entrée de ce type ne peut être empêchée par la désinfection de l'fourni par la validation d'entrée.

Intéressant de noter que 35% des paramètres d'entrée vulnérables aux attaques cross-site scripting sont en réalité numérique, le dénombrement, ou les types booléens (y compris les listes de valeurs de ces types), tandis que 68% des paramètres d'entrée vulnérables aux SQL injection correspondent à ces types de données simples. Ainsi, la majorité des vulnérabilités de validation d'entrée pour ces applications auraient pu être évités par l'application de la validation de l'utilisateur fourni par l'entrée en fonction des types de données simples. Nous croyons que le grand nombre de paramètres vulné-

[3]Many CVE reports do not mention any attack vectors. Hence, we excluded them for this analysis.



(a) Vulnerabilités cross-site scripting



(b) Vulnerabilités SQL injection

FIGURE 8.7: Les types de données correspondant à des paramètres d'entrée vulnérables.

rables à SQL injection qui correspondent au type numérique est causée par le phénomène que beaucoup d'applications web utilisent des nombres entiers pour identifier les objets et utiliser ces valeurs dans la logique de l'application qui interagit avec la base de données back-end (par exemple pour identifier les utilisateurs, des messages ou blogs).

8.4.3 Discussion

Les résultats de notre étude suggèrent que la majorité des vulnérabilités cross-site scripting et SQL injection peut être prévenue par la validation implicite résultant d'incantation réduit à des types primitifs observés dans les applications écrites dans des langages fortement typés. En particulier, pour les vulnérabilités SQL injection, la distribution simple des paramètres à des nombres entiers et les booléens permettrait d'éliminer plus de la moitié des vulnérabilités existantes. Cela peut expliquer pourquoi les applications écrites en Java sont moins vulnérables à ces classes d'attaques.

Paramètres d'entrée vulnérables qui doivent accepter d'entrée arbitraire, et donc le cas doit être désinfecté par cas par le développeur, sont dans la pratique une large minorité. Par conséquent, nous préconisons que, analogue à l'affaire de cadre de soutien pour l'auto-assainissement, langages de développement web ou de cadres devraient soutenir la validation automatique des entrées de l'application Web en tant que mesure de sécurité. Une politique d'alimentation automatique de validation peut prendre plusieurs formes concrètes, telles que l'intégration dans le système de type de la langue, le cadre soutenu par l'annotation de paramètres d'entrée, ou une description de la politique centralisée appliquées au niveau du cadre.

Les résultats de notre étude empirique suggèrent que de nombreuses vulnérabilités SQL injection et cross-site scripting pourraient facilement être évités si les langages du web et des cadres serait en mesure de valider automatiquement l'entrée en fonction des types de données courants tels que les numéros entiers, booléen, et certains types de 'strings' telles que e-mails et les URL.

8.5 Prévention automatique des vulnérabilités de validation d'entrée dans les applications Web

Dans cette section, nous présentons IPAAS (**I**nput **PA**rameter **A**nalysis **S**ystem), une approche pour la sécurisation des applications Web contre les cross-site scripting et SQL injection attaque à l'aide de validation d'entrée. L'insight de IPAAS est de façon automatique et transparente d'augmenter autrement vulnérables environnements de développement d'applications Web avec des validateurs d'entrée qui se traduisent par des améliorations de sécurité importantes et tangibles pour les systèmes réels.

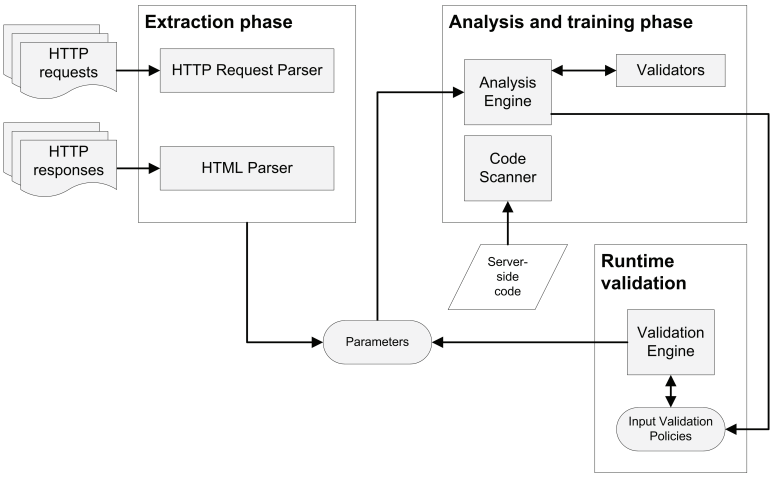


FIGURE 8.8: L'architecture IPAAS. The IPAAS architecture. A proxy intercepte HTTP messages générés au cours des essais d'application. Les paramètres d'entrée sont classés au cours d'une phase d'analyse selon l'une quelconque d'un ensemble de types possibles. Après suffisamment de données a été observée, IPAAS dérive d'une politique de validation d'entrée en fonction des types tirés pour chaque paramètre d'entrée l'application. Cette politique est automatiquement appliquée à l'exécution par la réécriture de l'application.

IPAAS peut être décomposée en trois phases : (i) l'extraction de paramètres, (ii) de type l'apprentissage, et (iii) l'application d'exécution. Un aperçu de l'architecture IPAAS est illustré à la figure 8.8. Dans le reste de cette section, nous décrivons chacune de ces phases en détail.

8.5.1 Extraction des paramètres

La première phase est essentiellement une étape de collecte de données. Ici, un proxy intercepte les messages du serveur HTTP échangées entre un client web et l'application au cours des essais. Pour chaque demande, tous les paramètres observés sont analysés en paires clé-valeur, associée à la ressource demandée, et stockées dans une base de données. Chaque réponse contenant un document HTML est traitée par un parseur HTML qui extrait des liens et des formes qui ont des objectifs liés à l'application testée. Pour chaque lien contenant une chaîne de requête, paires clé-valeur sont extraits comme dans le cas des demandes. Pour chaque formulaire, tous les éléments d'entrée sont extraites. De plus, ces éléments d'entrée qui spécifient un ensemble de valeurs possibles (par exemple, `select` éléments) sont traversées de collecter ces valeurs.

Type	Validator
`boolean`	`(0|1)|(true|false)|(yes|no)`
`integer`	`(+|-)?[0-9]+`
`float`	`(+|-)?[0-9]+(\+.[0-9]+)?`
`URL`	*RFC 2396, RFC 2732*
`token`	*static set of string literals*
`word`	[0-9a-zA-Z@_-]+
`words`	[0-9a-zA-Z@_- \r\n\t]+
`free-text`	*none*

Table 8.3: IPAAS types et leurs validateurs.

8.5.2 Analyse des paramètres

L'objectif de la deuxième phase consiste à étiqueter chaque paramètre extraite lors de la première phase avec un type de données sur la base des valeurs observées pour ce paramètre. Le processus de marquage est réalisé en appliquant un ensemble de validation à des entrées de test.

Les valideurs

Validateurs sont des fonctions qui vérifient si une valeur correspond à un ensemble particulier de contraintes. Dans cette phase, IPAAS applique un ensemble de validation, dont chacun vérifie qu'une entrée appartient à l'un d'un ensemble de types. L'ensemble des types et des expressions régulières décrivant les valeurs légitimes sont présentés dans le tableau 8.3. En plus des types énumérés dans le tableau 8.3, IPAAS reconnaît les listes de chacun de ces types.

Système d'analyse

IPAAS détermine le type d'un paramètre dans deux sous-phases. Dans la première, les types sont tirés fondée sur des valeurs qui ont été enregistrés pour chaque paramètre. Dans le deuxième, les types sont tirés augmenté en utilisant les valeurs extraites des documents HTML.

L'apprentissage Dans la première sous-étape, l'analyse commence par la récupération de tous les chemins des ressources qui ont été visitées pendant le test d'application. Pour chaque chemin, l'algorithme récupère l'ensemble unique de paramètres et l'ensemble complet de valeurs pour chacune de ces paramètres observés au cours de la phase d'extraction. Chaque paramètre est attribué un vecteur résultat entier de longueur égale au nombre de validation possibles.

La phase d'apprentissage type réel commence par passage de chaque valeur d'un paramètre donné à chaque type possible de validation. Si un validateur accepte une valeur, l'entrée correspondante dans le vecteur score de ce paramètre est incrémenté par un. Dans le cas où aucun validateur accepte une valeur, puis le moteur d'analyse assigne la `free-text` de type au paramètre et arrête le traitement des valeurs.

Après toutes les valeurs d'un paramètre ont été traitées, le vecteur score est utilisé pour sélectionner un type et, par conséquent, un validateur. Plus précisément, le type avec le score le plus élevé dans le vecteur est choisi. Si il ya une égalité, puis le type le plus restrictif est affecté, ce qui correspond à l'ordre donné dans le tableau 8.3.

La seconde sous-phase utilise les informations extraites de documents HTML. Tout d'abord, une vérification est effectuée afin de déterminer si le paramètre est associé à un fichier HTML, `textarea` élément. Si c'est le cas, le paramètre est immédiatement attribué le `free-text` type. Sinon, l'algorithme vérifie si le paramètre correspond à une `input` élément qui fait partie d'une case à cocher, bouton radio, ou `select` liste. Dans ce cas, l'ensemble de valeurs observées sont possibles assignée au paramètre. Ailleurs, si l'élément associé est une case à cocher, un multi-valeurs `select`, ou le nom du paramètre se termine par la 'string' `[]`, le paramètre est marqué comme une liste.

Le système d'analyse en déduit alors la politique de validation d'entrée pour chaque paramètre. Pour chaque ressource, le chemin est lié à l'emplacement physique du fichier source de l'application correspondante. Puis, les paramètres de ressources sont regroupées par type d'entrée (par exemple, query string, request body, cookie) et sérialisé dans le cadre d'une politique de validation d'entrée. Enfin, la politique est écrite sur le disque.

Analyse statique Les sous-phases de formation décrites ci-dessus peuvent être complétées par une analyse statique. En particulier, IPAAS pouvez utiliser une simple analyse statique de trouver des paramètres et des ressources d'application qui ont été manquées au cours de la phase d'apprentissage en raison de données insuffisantes de formation. Cette analyse est, bien sûr, spécifique à un langage de programmation particulier et le cadre.

8.5.3 Runtime Environment

Le résultat des deux premières phases est un ensemble de politiques de validation d'entrée pour chaque paramètre d'entrée à l'application Web en cours de test. La troisième phase se produit au cours du déploiement. à l'exécution, IPAAS intercepte les demandes entrantes et vérifie chaque requête contre la politique de validation des paramètres qui ressource. Si une valeur de paramètre contenue dans une demande ne répond pas aux contraintes spécifiées

par la politique, puis IPAAS chute de la demande. Sinon, l'application continue d'exécution.

Une demande peut contenir des paramètres qui n'ont pas été observées pendant les phases précédentes, soit dans le sous-phases d'apprentissage ou de l'analyse statique. Dans ce cas, il ya deux options possibles. Il s'agit d'une approche conservatrice qui pourrait, d'autre part, conduire à débordements programme. Sinon, la demande peut être acceptée et le nouveau paramètre marquée comme valide. Ce fait pourrait être utilisé dans une phase d'apprentissage à la suite de rafraîchir les politiques de l'application de validation d'entrée.

8.6 Conclusion

Pour comprendre comment améliorer la sécurité des applications Web, des idées sur la façon dont les vulnérabilités des applications Web réelles ont évolué et comment elles peuvent être évitées devenue d'intérêt. Dans cette thèse, nous affirmons que, en vue d'améliorer la sécurité des applications Web, les vulnérabilités web courantes telles que cross-site scripting et SQL injection doivent être automatiquement empêchée en utilisant des techniques de validation d'entrée.

Notre étude sur l'évolution des vulnérabilités de validation d'entrée dans les applications web dans le chapitre 4 démontre que ces classes de vulnérabilités sont encore très répandues. Mesures sur un grand nombre de vulnérabilités de validation d'entrée fourni des indications pour savoir si les développeurs sont mieux à l'écriture d'applications Web sécurisées aujourd'hui qu'elles ne l'habitude d'être dans le passé. Les résultats de cette étude suggèrent que les développeurs web sont toujours pas à mettre en œuvre les mécanismes de défense contre les vulnérabilités existantes de validation d'entrée et que il ya un besoin de techniques que les applications Web sécurisées *par la conception*. C'est, techniques qui empêchent ces classes de vulnérabilités automatiquement et fonctionne de façon transparente pour les développeurs.

Les outils, de cadres Web et langages de programmation qui sont utilisés pour développer une application web ont un impact direct sur la façon dont le système doit être créé et, en tant que telle, détermine également les moyens qui sont nécessaires pour assurer l'application résultante. Pour comprendre comment sécuriser des applications Web, nous explorons dans le chapitre 5 la relation entre le langage de programmation Web utilisé pour développer une application web et les vulnérabilités qui sont fréquemment rapportés. En particulier, nous identifier et de quantifier l'importance des mécanismes de typage des langages de programmation web et des cadres pour la sécurité des applications Web.

Nous avons utilisé les connaissances à partir des études empiriques pré-

sentées dans les chapitres 4 et 5, pour construire un système qui améliore le développement sécurisé des applications Web en automatique et transparente des environnements de développement Web augmentant avec validateurs d'entrée robustes. Nous montrons que cette approche améliore la sécurité des applications Web réelles de manière significative.

Appendix A

Web Application Frameworks

Framework	Version	Programming Language / Platform
flow3	1.0.0-alpha14-build46	php
php fat-free	1.4.4	php
alloy	0.6.0	php
php on trax	0.16.0	php
jelix	1.2.1.1	php
quickframework	1.4.1	php
jasper	1.9.2	php
ez components	2009.2.1	php
caffeine	1.0	php
drupal	7.0	php
xajax	0.5	php
wasp	1.2	php
cakephp	1.3.7	php
joomla	1.6.1	php
fusebox	5	php
modx	2.0.8pl	php
smarty	3.0.7	php
solarphp	1.1.2	php
runcms	2.2.2	php
codeigniter	2.0.0	php
symfony	1.4.9	php
adventure	1.13	php
nanoajax	0.0.2	php
qcodo	0.4.20	php
lamplighter	2.0.1	php
php.mvc	1.0	php

fastframe	3.4	php
yii	1.1.6	php
e-xoops	2.5.0	php
lithium	0.9.9	php
studs	0.9.8	php
kohana	3.1.1.1	php
horde	3.3.11	php
recess	0.2	php
lightvc	1.0.4	php
seagull	0.6.8	php
tekuna	0.2.202	php
kumbiaphp	1.0b1	php
nubuilder	11.02.18	php
catalyst	5.80032	perl
dancer	1.3	perl
cgi::application	4.31	perl
Rose-HTML-Objects	NA	perl
Reaction	NA	perl
interchange	5.6.3	perl
mason	1.45	perl
continuity	1.3	perl
mojo	1.12	perl
web.py	0.34	python
zope	3.4.0	python
web2py	1.93.2	python
django	1.2.5	python
rails	3.0.5	ruby
sinatra	1.2.1	ruby
iowa	0.99.2.19	ruby
camping	2.1	ruby
merb	1.1.2	ruby
castle	2.5	.net
dotnetnuke	5.6.1	.net
spring.net	1.3.1	.net
maverick.net	1.2.0.1	.net
clockwork	3.0.4.2	.net
bistro	0.9.2	.net
click	2.2.0	java
struts 2	2.2.1.1	java
myfaces	2.0.4	java
mojarra	2.1.0	java

play	1.1.1	java
google web toolkit	2.2.0	java
echo2	2.1.1	java
spring mvc	3	java
grails	1.3.7	java
jboss seam	2.2.1	java
wicket	1.4.16	java
tapestry	5.2.4	java
dwr	2.0.6	java
jvx webui	0.8	java
webflow	2.3.0	java

Table A.1: Web frameworks analyzed

Bibliography

[1] Acunetix. Acunetix web vulnerability scanner. http://www.acunetix.com/vulnerability-scanner/, 2012.

[2] Omar H. Alhazmi, Yashwant K. Malaiya, and Indrajit Ray. Security vulnerabilities in software systems: A quantitative perspective. In Sushil Jajodia and Duminda Wijesekera, editors, *DBSec*, volume 3654 of *Lecture Notes in Computer Science*, pages 281–294. Springer, 2005.

[3] Magnus Almgren, Hervé Debar, and Marc Dacier. *A Lightweight Tool for Detecting Web Server Attacks*, pages 157–170. 2000.

[4] AnantaSec. Ananta security blog. http://anantasec.blogspot.com/, 2009.

[5] William A. Arbaugh, William L. Fithen, and John McHugh. Windows of vulnerability: A case study analysis. *Computer*, 33:52–59, December 2000.

[6] Ashish Arora, Ramayya Krishnan, Rahul Telang, and Yubao Yang. Impact of vulnerability disclosure and patch availability - an empirical analysis. In *In Third Workshop on the Economics of Information Security*, 2004.

[7] Marco Balduzzi, Carmen Torrano Gimenez, Davide Balzarotti, and Engin Kirda. Automated discovery of parameter pollution vulnerabilities in web applications. In *NDSS'11, 8th Annual Network and Distributed System Security Symposium, 6-9 February 2011, San Diego, California, USA*, 02 2011.

[8] Davide Balzarotti, Marco Cova, Vika Felmetsger, Nenad Jovanovic, Engin Kirda, Christopher Krügel, and Giovanni Vigna. Saner: composing static and dynamic analysis to validate sanitization in web applications. In *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, CA, USA, May 2008.

[9] Inc. Barracuda Networks. Barracuda web application firewall. http://www.barracudanetworks.com/ns/products/web-site-firewall-overview.php, 2011.

[10] Daniel Bates, Adam Barth, and Collin Jackson. Regular expressions considered harmful in client-side xss filters. In *WWW '10: Proceedings of the 19th international conference on World wide web*, pages 91–100, New York, NY, USA, 2010. ACM.

[11] Jason Bau, Elie Bursztein, Divij Gupta, and John C. Mitchell. State of the art: Automated black-box web application vulnerability testing. In *IEEE Symposium on Security and Privacy*, pages 332–345. IEEE Computer Society, 2010.

[12] Justin Billig, Yuri Danilchenko, and Charles E. Frank. Evaluation of google hacking. In *Proceedings of the 5th annual conference on Information security curriculum development*, InfoSecCD '08, pages 27–32, New York, NY, USA, 2008. ACM.

[13] Nick Bilton. How credit card data is stolen and sold. http://bits.blogs.nytimes.com/2011/05/03/card-data-is-stolen-and-sold/, May 2011.

[14] Stephen W. Boyd and Angelos D. Keromytis. Sqlrand: Preventing sql injection attacks. In Markus Jakobsson, Moti Yung, and Jianying Zhou, editors, *Applied Cryptography and Network Security, Second International Conference, ACNS 2004, Yellow Mountain, China, June 8-11, 2004, Proceedings*, volume 3089 of *Lecture Notes in Computer Science*, pages 292–302. Springer, 2004.

[15] Jonathan Burket, Patrick Mutchler, Michael Weaver, Muzzammil Zaveri, and David Evans. Guardrails: a data-centric web application security framework. In *Proceedings of the 2nd USENIX conference on Web application development*, WebApps'11, pages 1–1, Berkeley, CA, USA, 2011. USENIX Association.

[16] Cristian Cadar, Daniel Dunbar, and Dawson Engler. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, OSDI'08, pages 209–224, Berkeley, CA, USA, 2008. USENIX Association.

[17] Bryan Casey, Carsten Hagemann, David Merrill, Jens Thamm, Ashok Kallarakkal, Jason Kravitz, John Kuhn, John C. Pierce, Jon Larimer, Leslie Horacek, Marc Noske, Marc van Zadelhoff, Mark E. Wallis, Michelle Alvarez, Mike Warfield, Ory Segal, Patrick Vandenberg, Pete

Allor, Phil Neray, Ralf Iffert, Randy Stone, Ryan McNulty, Scott Moore, Scott Van Valkenburgh, Tom Cross, and Vidhi Desai. Ibm x-force 2011 mid-year trend and risk report. Technical report, IBM, 2011.

[18] Hasan Cavusoglu, Huseyin Cavusoglu, and S. Raghunathan. Emerging issues in responsible vulnerability disclosure. In *Proceedings of WITS 2004*, 2004.

[19] US CERT. Cyber security bulletins. http://www.us-cert.gov/cas/bulletins/, 2012.

[20] S. M. Christey and R. A. Martin. Vulnerability type distributions in cve. http://cwe.mitre.org/documents/vuln-trends/index.html, 2007.

[21] S. Clark, S. Frei, M. Blaze, and J. Smith. Familiarity breeds contempt: The honeymoon effect and the role of legacy code in zero-day vulnerabilities. In *Annual Computer Security Applications Conference*, 2010.

[22] Mike Dausin, Adam Hils, Dan Holden, Prajakta Jagdale, Jason Jones, Rohan Kotian, Jennifer Lake, Mark Painter, Taylor Anderson McKinley, Alen Puzic, and Bob Schiermann. The 2011 mid-year top cyber security risks report. Technical report, Hewlett-Packard, 2011.

[23] Rohit Dhamankar, Mike Dausin, Marc Eisenbarth, and James King. The top cyber security risks. http://www.sans.org/top-cyber-security-risks/, 2009.

[24] A. Doupe, M. Cova, and G. Vigna. Why Johnny Can't Pentest: An Analysis of Black-box Web Vulnerability Scanners. In *Proceedings of the Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA)*, Bonn, Germany, July 2010.

[25] Maureen Doyle and James Walden. An empirical study of the evolution of php web application security. In *Proceedings of the 7th International Workshop on Security Measurements And Metrics*, Banff, Canada, September 2011.

[26] Inc. F5 Networks. F5 big-ip application security manager (asm). http://www.f5.com/solutions/security/web-application/, 2011.

[27] Joe Faulhaber, David Felstead, Paul Henry, Jeff Jones, Ellen Cram Kowalczyk, Jimmy Kuo, John Lambert, Marc Lauricella, Aaron Margosis, Michelle Meyer, Anurag Pandit, Anthony Penta, Dave Probert, Tim Rains, Mark E. Russinovich, Weijuan Shi, Adam Shostack, Frank Simorjay, Hemanth Srinivasan, Holly Stewart, Matt Thomlinson, Jeff Williams, Scott Wu, and Terry Zink. Microsoft security intelligence report volume 11. Technical report, Microsoft, 2011.

[28] Matthew Finifter and David Wagner. Exploring the Relationship Between Web Application Development Tools and Security. In *USENIX Conference on Web Application Development (WebApps)*. USENIX Association, June 2011.

[29] J. Fonseca and M. Vieira. Mapping software faults with web security vulnerabilities. In *Proceedings of the IEEE Internation Conference on Dependable Systems and Networks*, pages 257–266, June 2008.

[30] Django Software Foundation. Django. https://www.djangoproject.com/.

[31] Open Security Foundation. Osf datalossdb. http://www.datalossdb.org/, 2012.

[32] Stefan Frei. *Security Econometrics - The Dynamics of (In)Security*. PhD thesis, ETH Zurich, sep 2009.

[33] Stefan Frei, Martin May, Ulrich Fiedler, and Bernhard Plattner. Large-scale vulnerability analysis. In *LSAD '06: Proceedings of the 2006 SIGCOMM workshop on Large-scale attack defense*, pages 131–138, New York, NY, USA, 2006. ACM.

[34] Matthew Van Gundy and Hao Chen. Noncespaces: Using randomization to enforce information flow tracking and thwart cross-site scripting attacks. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2009, San Diego, California, USA, 8th February - 11th February 2009*, 2009.

[35] Vivek Haldar, Deepak Chandra, and Michael Franz. Dynamic taint propagation for java. In *Proceedings of the 21st Annual Computer Security Applications Conference*, pages 303–311, Washington, DC, USA, 2005. IEEE Computer Society.

[36] W. Halfond, S. Anand, and A. Orso. Precise Interface Identification to Improve Testing and Analysis of Web Applications. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2009)*, Chicago, Illinois, USA, July 2009.

[37] William G.J. Halfond and Alessandro Orso. AMNESIA: Analysis and Monitoring for NEutralizing SQL-Injection Attacks. In *Proceedings of the IEEE and ACM International Conference on Automated Software Engineering (ASE 2005)*, Long Beach, CA, USA, Nov 2005.

[38] Hewlett-Packard. Hp webinspect. https://www.fortify.com/products/web_inspect.html, 2012.

[39] Thorsten Holz, Markus Engelberth, and Felix Freiling. Learning more about the underground economy: a case-study of keyloggers and drop-zones. In *Proceedings of the 14th European conference on Research in computer security*, ESORICS'09, pages 1–18, Berlin, Heidelberg, 2009. Springer-Verlag.

[40] Pieter Hooimeijer, Benjamin Livshits, David Molnar, Prateek Saxena, and Margus Veanes. Fast and precise sanitizer analysis with bek. In *Proceedings of the 20th USENIX conference on Security*, SEC'11, pages 1–1, Berkeley, CA, USA, 2011. USENIX Association.

[41] Yao-Wen Huang, Fang Yu, Christian Hang, Chung-Hung Tsai, Der-Tsai Lee, and Sy-Yen Kuo. Securing web application code by static analysis and runtime protection. In *Proceedings of the 13th international conference on World Wide Web*, WWW '04, pages 40–52, New York, NY, USA, 2004. ACM.

[42] IBM. Rational appscan. http://www-01.ibm.com/software/awdtools/appscan/, 2012.

[43] Google Inc. ctemplate. http://code.google.com/p/ctemplate/.

[44] Trevor Jim, Nikhil Swamy, and Michael Hicks. Defeating script injection attacks with browser-enforced embedded policies. In *WWW '07: Proceedings of the 16th international conference on World Wide Web*, pages 601–610, New York, NY, USA, 2007. ACM.

[45] Martin Johns, Christian Beyerlein, Rosemaria Giesecke, and Joachim Posegga. Secure code generation for web applications. In Fabio Massacci, Dan S. Wallach, and Nicola Zannone, editors, *ESSoS*, volume 5965 of *Lecture Notes in Computer Science*, pages 96–113. Springer, 2010.

[46] Martin Johns, Björn Engelmann, and Joachim Posegga. Xssds: Server-side detection of cross-site scripting attacks. In *Proceedings of the 2008 Annual Computer Security Applications Conference*, ACSAC '08, pages 335–344, Washington, DC, USA, 2008. IEEE Computer Society.

[47] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. Pixy: A static analysis tool for detecting web application vulnerabilities (short paper). In *SP '06: Proceedings of the 2006 IEEE Symposium on Security and Privacy*, pages 258–263, Washington, DC, USA, 2006. IEEE Computer Society.

[48] Klaus Julisch. Clustering intrusion detection alarms to support root cause analysis. *ACM Trans. Inf. Syst. Secur.*, 6:443–471, November 2003.

[49] Auguste Kerckhoffs. La cryptographie militaire, January 1883.

[50] Adam Kieżun, Vijay Ganesh, Philip J. Guo, Pieter Hooimeijer, and Michael D. Ernst. HAMPI: A solver for string constraints. In *ISSTA 2009, Proceedings of the 2009 International Symposium on Software Testing and Analysis*, pages 105–116, Chicago, IL, USA, July 21–23, 2009.

[51] Adam Kieżun, Philip J. Guo, Karthick Jayaraman, and Michael D. Ernst. Automatic creation of SQL injection and cross-site scripting attacks. In *ICSE'09, Proceedings of the 31st International Conference on Software Engineering*, Vancouver, BC, Canada, May 20–22, 2009.

[52] Engin Kirda, Christopher Kruegel, Giovanni Vigna, and Nenad Jovanovic. Noxes: a client-side solution for mitigating cross-site scripting attacks. In *SAC '06: Proceedings of the 2006 ACM symposium on Applied computing*, pages 330–337, New York, NY, USA, 2006. ACM.

[53] D.V. Klein. Defending against the wily surfer - web-based attacks and defenses. In *Proceedings of the 1st USENIX Workshop on Detection Symposium and Network Monitoring*, Santa Clara CA, April 1999.

[54] Yuji Kosuga, Kenji Kono, Miyuki Hanaoka, Miho Hishiyama, and Yu Takahama. Sania: Syntactic and semantic analysis for automated testing against sql injection. In *ACSAC*, pages 107–117. IEEE Computer Society, 2007.

[55] Jake Kouns, Kelly Todd, Brian Martin, David Shettler, Steve Tornio, Craig Ingram, and Patrick McDonald. The open source vulnerability database. http://osvdb.org/, 2010.

[56] C. Kruegel and G. Vigna. Anomaly Detection of Web-based Attacks. In *Proceedings of the* 10th *ACM Conference on Computer and Communication Security (CCS '03)*, pages 251–261, Washington, DC, October 2003. ACM Press.

[57] C. Kruegel, G. Vigna, and W. Robertson. A Multi-model Approach to the Detection of Web-based Attacks. *Computer Networks*, 48(5):717–738, August 2005.

[58] V. Benjamin Livshits and Monica S. Lam. Finding security errors in Java programs with static analysis. In *Proceedings of the 14th Usenix Security Symposium*, pages 271–286, August 2005.

[59] Mike Ter Louw and V. N. Venkatakrishnan. Blueprint: Robust prevention of cross-site scripting attacks for existing browsers. In *Proceedings of the 2009 30th IEEE Symposium on Security and Privacy*, pages 331–346, Washington, DC, USA, 2009. IEEE Computer Society.

[60] PortSwigger Ltd. Burp scanner. http://portswigger.net/burp/scanner.html, 2011.

[61] Giorgio Maone. Noscript. http://www.noscript.net.

[62] Bob Martin, Mason Brown, Alan Paller, and Dennis Kirby. 2010 cwe/sans top 25 most dangerous software errors. http://cwe.mitre.org/top25/, 2010.

[63] Ferruh Mavituna. Sql injection cheat sheet. http://ferruh.mavituna.com/sql-injection-cheatsheet-oku/, 2009.

[64] McAfee. Mcafee threats report: Third quarter 2011. Technical report, McAfee Labs, 2011.

[65] Peter Mell, Karen Scarfone, and Sasha Romanosky. A complete guide to the common vulnerability scoring system version 2.0. http://www.first.org/cvss/cvss-guide.html, 2007.

[66] MITRE. Common platform enumeration (cpe). http://cpe.mitre.org/, 2010.

[67] MITRE. Common vulnerabilities and exposures (cve). http://cve.mitre.org/, 2010.

[68] MITRE. Common weakness enumeration (cwe). http://cwe.mitre.org/, 2010.

[69] MITRE. Mitre faqs. http://cve.mitre.org/about/faqs.html, 2010.

[70] M.J. Mondro. Approximation of mean time between failure when a system has periodic maintenance. *Reliability, IEEE Transactions on*, 51(2):166 –167, jun 2002.

[71] Nate Mook. Cross-site scripting worm hits myspace. http://betanews.com/2005/10/13/cross-site-scripting-worm-hits-myspace/, October 2005.

[72] Tyler Moore and Richard Clayton. Evil searching: Compromise and recompromise of internet hosts for phishing. In Roger Dingledine and Philippe Golle, editors, *Financial Cryptography and Data Security*, volume 5628 of *Lecture Notes in Computer Science*, pages 256–272. Springer Berlin / Heidelberg, 2009. 10.1007/978-3-642-03549-4_16.

[73] J.D. Musa, A. Ianino, and Okumuto K. *Software Reliability Measurement Prediction Application*. McGraw-Hill, 1987.

[74] Yacin Nadji, Prateek Saxena, and Dawn Song. Document structure integrity: A robust basis for cross-site scripting defense. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2009, San Diego, California, USA, 8th February - 11th February 2009*, 2009.

[75] Stephan Neuhaus and Thomas Zimmermann. Security trend analysis with cve topic models. In *Proceedings of the 21st IEEE International Symposium on Software Reliability Engineering*, November 2010.

[76] Stephan Neuhaus, Thomas Zimmermann, Christian Holler, and Andreas Zeller. Predicting vulnerable software components. In Peng Ning, Sabrina De Capitani di Vimercati, and Paul F. Syverson, editors, *ACM Conference on Computer and Communications Security*, pages 529–540. ACM, 2007.

[77] Anh Nguyen-Tuong, Salvatore Guarnieri, Doug Greene, Jeff Shirley, and David Evans. Automatically hardening web applications using precise tainting. In Ryôichi Sasaki, Sihan Qing, Eiji Okamoto, and Hiroshi Yoshiura, editors, *SEC*, pages 295–308. Springer, 2005.

[78] NIST. National vulnerability database version 2.2. http://nvd.nist.gov/, 2010.

[79] OWASP. Owasp top 10 - 2010, the ten most critical web application security risks. https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project, 2010.

[80] Andy Ozment and Stuart E. Schechter. Milk or wine: does software security improve with age? In *USENIX-SS'06: Proceedings of the 15th conference on USENIX Security Symposium*, Berkeley, CA, USA, 2006. USENIX Association.

[81] H. Peine. Security test tools for web applications. Technical Report 048.06, Fraunhofer IESE, January 2006.

[82] Tadeusz Pietraszek and Chris Vanden Berghe. Defending against injection attacks through context-sensitive string evaluation. In Alfonso Valdes and Diego Zamboni, editors, *RAID*, volume 3858 of *Lecture Notes in Computer Science*, pages 124–145. Springer, 2005.

[83] 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*, Chicago, Illinois, USA, 2010.

[84] Niels Provos, Panayiotis Mavrommatis, Moheeb Abu Rajab, and Fabian Monrose. All your iframes point to us. In *Proceedings of the 17th conference on Security symposium*, pages 1–15, Berkeley, CA, USA, 2008. USENIX Association.

[85] Rapid7. Metasploit. http://www.metasploit.com, 2012.

[86] Jeremias Reith. Internals of noxss. http://www.noxss.org/wiki/Internals, October 2008.

[87] Eric Rescorla. Security holes... who cares? In *Proceedings of the 12th conference on USENIX Security Symposium - Volume 12*, pages 6–6, Berkeley, CA, USA, 2003. USENIX Association.

[88] Eric Rescorla. Is finding security holes a good idea? *IEEE Security and Privacy*, 3:14–19, January 2005.

[89] Andres Riancho. w3af - web application attack and audit framework. http://w3af.sourceforge.net/, 2011.

[90] W. Robertson and G. Vigna. Static enforcement of web application integrity through strong typing. In *Proceedings of the 18th conference on USENIX security symposium*, pages 283–298. USENIX Association, 2009.

[91] W. Robertson, G. Vigna, C. Kruegel, and R. Kemmerer. Using Generalization and Characterization Techniques in the Anomaly-based Detection of Web Attacks. In *Proceeding of the Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2006.

[92] Martin Roesch. Snort - lightweight intrusion detection for networks. In *Proceedings of the 13th USENIX conference on System administration*, LISA '99, pages 229–238, Berkeley, CA, USA, 1999. USENIX Association.

[93] David Ross. Ie 8 xss filter. http://blogs.technet.com/srd/archive/2008/08/18/ie-8-xss-filter-architecture-implementation.aspx, August 2008.

[94] RSA. Cyber security awareness month fails to deter phishers. Technical report, RSA, 2011.

[95] RSnake. Xss (cross site scripting) cheat sheet esp: for filter evasion. http://ha.ckers.org/xss.html, 2009.

[96] Mike Samuel, Prateek Saxena, and Dawn Song. Context-sensitive auto-sanitization in web templating languages using type qualifiers. In *Proceedings of the 18th ACM conference on Computer and communications security*, CCS '11, pages 587–600, New York, NY, USA, 2011. ACM.

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

[98] Prateek Saxena, David Molnar, and Benjamin Livshits. Scriptgard: Automatic context-sensitive sanitization for large-scale legacy web applications. In *Proceedings of the Conference on Computer and Communications Security*, October 2011.

[99] Theodoor Scholte, Davide Balzarotti, and Engin Kirda. Quo Vadis? A Study of the Evolution of Input Validation Vulnerabilities in Web Applications. In *Proceedings of the International Conference on Financial Cryptography and Data Security*, Bay Gardens Beach Resort, Saint Lucia, 2011.

[100] Theodoor Scholte, Davide Balzarotti, and Engin Kirda. Have things changed now? a study of the evolution of input validation vulnerabilities in web applications. *Elsevier Computers & Security*, 2012.

[101] Theodoor Scholte, Davide Balzarotti, William Robertson, and Engin Kirda. An Empirical Analysis of Input Validation Mechanisms in Web Applications and Languages. In *Proceedings of the 27th ACM Symposium On Applied Computing (SAC 2012)*, Riva del Garda, Italy., March 2012.

[102] Mathew J. Schwartz. Sony data breach cleanup to cost $171 million. *InformationWeek*, May 2011.

[103] David Scott and Richard Sharp. Abstracting application-level web security. In *Proceedings of the 11th international conference on World Wide Web*, WWW '02, pages 396–407, New York, NY, USA, 2002. ACM.

[104] Panda Security. Panda security report the cyber-crime black market: Uncovered. Technical report, Panda Security, 2011.

[105] WhiteHat Security. Whitehat website security statistic report. Technical report, WhiteHat Security, 2011.

[106] SecurityFocus. Bugtraq. http://www.securityfocus.com, 2012.

[107] Nuno Seixas, Jose Fonseca, Marco Vieira, and Henrique Madeira. Looking at Web Security Vulnerabilities from the Programming Language Perspective: A Field Study. In *Proceedings of the International Symposium on Software Reliability Engineering*, pages 129–135. IEEE Computer Society, 2009.

[108] R. Shirey. Request for Comments: 2828. Technical report, The Internet Society, May 2000.

[109] R. Shirey. Request for comments: 4949. Technical report, The Internet Society, 2007.

[110] Zhendong Su and Gary Wassermann. The essence of command injection attacks in web applications. In *Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '06, pages 372–382, New York, NY, USA, 2006. ACM.

[111] Trustwave. Modsecurity: Open source web application firewall. http://www.modsecurity.org/, 2011.

[112] G. Vigna, W. Robertson, V. Kher, and R.A. Kemmerer. A Stateful Intrusion Detection System for World-Wide Web Servers. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC 2003)*, pages 34–43, Las Vegas, NV, December 2003.

[113] W3Counter. Web browser market share trends. http://www.w3counter.com/trends, 2011.

[114] J. Walden, M. Doyle, R. Lenhof, and J. Murray. Java vs. php: Security implications of language choice for web applications. In *International Symposium on Engineering Secure Software and Systems (ESSoS)*, February 2010.

[115] James Walden, Maureen Doyle, Grant A. Welch, and Michael Whelan. Security of open source web applications. In *Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement*, ESEM '09, pages 545–553, Washington, DC, USA, October 2009. IEEE Computer Society.

[116] Gary Wassermann and Zhendong Su. Sound and Precise Analysis of Web Applications for Injection Vulnerabilities. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation*, San Diego, CA, June 2007. ACM Press New York, NY, USA.

[117] Gary Wassermann and Zhendong Su. Static Detection of Cross-Site Scripting Vulnerabilities. In *Proceedings of the 30th International Conference on Software Engineering*, Leipzig, Germany, May 2008. ACM Press New York, NY, USA.

[118] Joel Weinberger, Prateek Saxena, Devdatta Akhawe, Matthew Finifter, Richard Shin, and Dawn Song. An Empirical Analysis of

XSS Sanitization in Web Application Frameworks. Technical report, UC Berkeley, 2011.

[119] A. Wiegenstein, F. Weidemann, M. Schumacher, and S. Schinzel. Web application vulnerability scanners - a benchmark. Technical report, Virtual Forge GmbH, October 2006.

[120] S.W. Woo, O.H. Alhazmi, and Y. K. Malaiya. An analysis of the vulnerability discovery process in web browsers. In *Proceedings of the 10th International Conference on Software Engineering and Applications*, Dallas, TX, Nov 2006.

[121] Yichen Xie and Alex Aiken. Static detection of security vulnerabilities in scripting languages. In *USENIX-SS'06: Proceedings of the 15th conference on USENIX Security Symposium*, Berkeley, CA, USA, 2006. USENIX Association.

[122] Fabian Yamaguchi, Felix Lindner, and Konrad Rieck. Vulnerability extrapolation: Assisted discovery of vulnerabilities using machine learning. In *Proc. of 5th USENIX Workshop on Offensive Technologies (WOOT)*, August 2011.

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

[124] Jianwei Zhuge, Thorsten Holz, Chengyu Song, Jinpeng Guo, Xinhui Han, and Wei Zou. Studying malicious websites and the underground economy on the chinese web. In *Workshop on the Economics of Information Security (WEIS'08)*, 2008.