A Survey on Server-Side Approaches to Securing Web Applications

XIAOWEI LI and YUAN XUE, Vanderbilt University

Web applications are one of the most prevalent platforms for information and service delivery over the Internet today. As they are increasingly used for critical services, web applications have become a popular and valuable target for security attacks. Although a large body of techniques have been developed to fortify web applications and mitigate attacks launched against them, there has been little effort devoted to drawing connections among these techniques and building the big picture of web application security research.

This article surveys the area of securing web applications from the server side, with the aim of systematizing the existing techniques into a big picture that promotes future research. We first present the unique aspects of the web application development that cause inherent challenges in building secure web applications. We then discuss three commonly seen security vulnerabilities within web applications: *input validation vulnerabilities*, *session management vulnerabilities,* and *application logic vulnerabilities*, along with attacks that exploit these vulnerabilities. We organize the existing techniques along two dimensions: (1) the security vulnerabilities and attacks that they address and (2) the design objective and the phases of a web application during which they can be carried out. These phases are *secure construction of new web applications*, *security analysis/testing of legacy web applications,* and *runtime protection of legacy web applications*. Finally, we summarize the lessons learned and discuss future research opportunities in this area.

Categories and Subject Descriptors: K.6.5 [**Management of Computing and Information Systems**]: Security and Protection

General Terms: Security

Additional Key Words and Phrases: Web application security, input validation vulnerability, session management vulnerability, application logic vulnerability

ACM Reference Format:
Xiaowei Li and Yuan Xue. 2014. A survey on server-side approaches to securing web applications. ACM Comput. Surv. 46, 4, Article 54 (March 2014), 29 pages.
DOI: http://dx.doi.org/10.1145/2541315

1. INTRODUCTION

The World Wide Web has evolved from a system that delivers static pages to a platform that supports distributed applications, known as web applications, and has become one of the most prevalent technologies for information and service delivery. The increasing popularity of web application can be attributed to several factors, such as remote accessibility, cross-platform compatibility, and fast development. Asynchronous JavaScript and XML (AJAX) technology also enhances the user experience of web applications with better interactiveness and responsiveness.

Authors' addresses: X. Li, Department of Electrical Engineering and Computer Science, Vanderbilt University, 400 24th Ave. South, Nashville , TN 37203; Y. Xue, Department of Electrical Engineering and Computer Science, Vanderbilt University, 400 24th Ave. South, Nashville, TN 37203. This work was supported by NSF TRUST (the Team for Research in Ubiquitous Secure Technology) Science and Technology Center (CCF-0424422).

DOI: http://dx.doi.org/10.1145/2541315

As web applications are increasingly used to deliver critical services, they become a valuable target for security attacks. Many web applications interact with back-end database systems, which may store sensitive information (e.g., financial, health). A compromised web application could result in an enormous information breach, severe financial losses, and ethical and legal consequences. A breach report from Verizon [2010] shows that web applications are at the top in both the number of breaches and the amount of data compromised.

The Web platform is a complex ecosystem composed of a large number of components and technologies, including the HTTP protocol, web server and server-side application development technologies (e.g., CGI, PHP, and ASP), and web browser and client-side technologies (e.g., JavaScript and Flash). Web applications built and hosted upon such a complex infrastructure enjoy a rapid and wide user adoption, thanks to the rich features of these components. However, the inconsistencies among different technologies migrate the challenge of ensuring security to web application development. Meanwhile, current web application development and testing frameworks offer limited security support. As a result, web application development is an error-prone process, and implementation of security measures requires substantial efforts. These efforts could be unrealistic under time-to-market pressure, especially for people with insufficient security skills or awareness. As a result, a high percentage of web applications deployed on the Internet are exposed to security vulnerabilities. According to a report by the Web Application Security Consortium, 49% of the web applications reviewed contain vulnerabilities considered high risk and more than 13% of the websites can be compromised completely [WASS 2007]. A recent report [WhiteHat 2010] revealed that more than 80% of the websites on the Internet have had at least one serious vulnerability.

Motivated by the urgent need to secure web applications, a substantial amount of research efforts have been devoted to this problem, developing a large number of techniques for hardening web applications and mitigating attacks. Many of these techniques make assumptions about the web technologies used in the application development and only address one particular type of security flaw. Additionally, the prototypes are usually implemented and evaluated on limited platforms. Practitioners often face a dilemma in selecting suitable techniques that meet their development needs—whether a technique can be directly applied; and if not, whether several techniques can be extended and/or combined. Thus, there is an urgent need to provide a systematic framework for uncovering the connection between the existing techniques. This survey marks an initial attempt toward such a framework.

In this article, we survey the state of the art in securing web applications, with a focus on approaches that are deployed on the server side. In particular, this survey covers the techniques that consider the following threat model: (1) *the web application itself is benign (i.e., not hosted or owned for malicious purposes) and hosted on a trusted and hardened infrastructure (i.e., a trusted computing base, including OS, web server, interpreter, etc.)*; and (2) *the attacker is able to manipulate either the contents or the sequence of web requests sent to the web application but cannot directly compromise the infrastructure or the application code*. Although this survey focuses on server-side techniques for securing web applications, it also covers techniques requiring collaboration between client and server. We note here that although browser security Wang et al. [2009] and Tang et al. [2010] is also an essential component in end-to-end web application security, research works on this topic usually have a different threat model, where web applications are considered as potentially malicious. This survey does not include the research works on browser security so that it can focus on the problem of building secure web applications and protecting vulnerable ones.

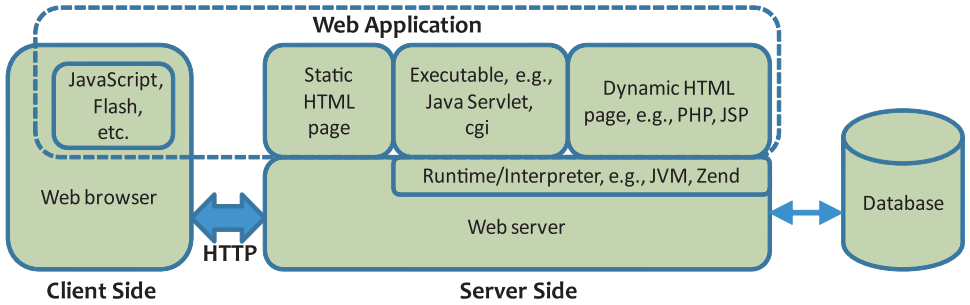

Fig. 1. Overview of web application.

Although there are several existing surveys in the web application security area, they either focus on one particular type of attack (e.g., XSS [García-Alfaro and Navarro-Arribas 2009], SQL injection [Halfond et al. 2006]) or summarize one class of techniques that can be applied (e.g., vulnerability analysis [Cova et al. 2007b], detection [Garcia-Alfaro and Navarro-Arribas 2008]). To the best of our knowledge, this is the first comprehensive survey covering a wide range of attacks—from input validation attacks to application logic attacks—and three categories of server-side defense techniques, including secure construction, vulnerability analysis and testing, and runtime protection.

The structure of this survey is as follows. Section 2 describes how a web application works and the unique characteristics of web application development. Section 3 describes three types of commonly seen security vulnerabilities within web applications, as well as the attack vectors that exploit these vulnerabilities. Section 4 classifies existing research works into three categories: *secure construction of new applications*, *security analysis and testing of legacy applications,* and *runtime protection of legacy applications*, and present the state of the art systematically. Finally, Section 5 summarizes this article by identifying several evolving trends in the area of web application security and the new challenges that are expected ahead.

2. UNDERSTANDING UNIQUE CHARACTERISTICS OF WEB APPLICATIONS

Web application is a client-server application that is executed over the Web platform. It is an integral part of today's Web ecosystem that enables dynamic information and service delivery. As shown in Figure 1, a web application consists of code on both the server side and the client side. On the server side, a web application receives user inputs via HTTP requests from the client (i.e., browser) and interacts with local file systems, back-end databases, or other components for data access and information retrieval. Its outputs (i.e., HTML pages) are sent to the client through HTTP responses. On the client side, HTML pages are rendered, and the client-side code (i.e., JavaScript) embedded in the HTTP responses is executed by the web browser. The client-side code can also communicate with the server-side code asynchronously without interfering with the display of the existing HTML page via AJAX and dynamically update the page. Currently, some programming libraries and frameworks (e.g., Rails and Django) are developed to facilitate rapid application development. For example, most web development frameworks provide session management functions, which developers can directly leverage to manage the web sessions.

The execution model of web applications is radically different from traditional applications. The development of web applications also faces challenges that are not ordinarily encountered in the development of traditional applications. First, in the open web environment, user inputs are potentially dangerous and can never be trusted. *Input*

validation is an important part of a web application to identify and sanitize untrusted user inputs. Although input validation is an essential function required by all types of applications, its implementation in web applications is much more challenging due to the unique features of web development technologies. Second, the communication between the client and the web application is carried out through the stateless HTTP protocol. As a result, multiple inputs from the same user will appear to be independent users to the web application. The web application has to employ *session management* in order to correlate the web requests from the same user into a web session. Finally, a web application is usually implemented as a number of modules with URLs as their entry points. This allows a user to directly access the application modules in an arbitrary order. Recently, AJAX applications provide users with a richer and more responsive experience by moving part of the computation to the client side, which avoids unnecessary network round trips. Since the computing results from the client side can never be trusted, additional validation of these results is required on the server side. These unique features significantly complicate the *logic implementation* of a web application, which needs to enforce the application's control flow between the client and the server and across different modules. These three aspects of web application development are described in the following sections.

2.1. Input Validation

User inputs can never be trusted and need to be validated or sanitized before they can be utilized by web applications. Usually, web developers employ sanitization routines (i.e., *sanitizers*) to transform user inputs into trusted data by filtering or escaping suspicious characters or constructs. Web applications allow developers to blend several types of constructs in one file for runtime interpretation. For instance, a PHP file may contain both static HTML tags and PHP statements, and an HTML page may embed executable JavaScript code. The representation of application data and code via an unstructured sequence of bytes is a unique feature of web application. While enhancing the development efficiency, this unique feature complicates the input validation of web applications, since developers have to anticipate the contexts where and how user inputs are utilized that pose different sanitization requirements (i.e., context-sensitive sanitization). For instance, applying the default HTML escaping sanitizer is recommended for sanitizing the values inside HTML tags; however, this sanitizer is insufficient for sanitizing values within URL attributes (such as src or href), since the URI attribute of the script tag can embed malicious code [Samuel et al. 2011].

2.2. Session Management

Web applications adopt an abstraction of a web session to identify and correlate a series of web requests from the same user during a certain period of time. A set of session variables (or session data) is associated with a web session and can be used by the application to record the conditions from the historical web requests that affect the future execution of the web application (i.e., application session state). The session variables can be maintained either at the client side (via a cookie, a hidden form, or URL rewriting) or at the server side (in a file or using a database). In the latter case, a unique identifier (session ID) is defined to index the explicit session variables stored at the server side and issued to the client. Most web programming languages (e.g., PHP and JSP) and frameworks offer developers a collection of functions for managing the web session. For example, in PHP, *session_start()* can be called to initialize a web session, and a predefined global array $\_SESSION$ can be employed to contain the session variables. In either case, the client plays a vital role in maintaining the session information.

2.3. Logic Implementation

A web application's logic is usually implemented through enforcing the control flow of the application and protecting sensitive information and operations, which can be achieved explicitly through security checks in the source code or implicitly through the navigation paths presented to users (i.e., interface hiding). Explicit security checks examine the application state, which is maintained by session variables and persistent objects in the database, before sensitive information and operations can be accessed. However, interface hiding only allows accessible resources and operations to be presented as web links exposed to users. Many web applications share common components in their business logic. For instance, authentication and authorization are the most common part of the control flow in many web applications through which a web application restricts its sensitive information and privileged operations from unauthorized users. A common practice is for developers to place authorization checks on certain security critical variables before all sensitive information and operations. These variables can include client-side cookies, session variables, and persistent objects in the database. In AJAX applications, the application functionality needs to be split in such a way that all required security checks are performed on the server.

3. UNDERSTANDING WEB APPLICATION VULNERABILITIES AND ATTACKS

In general, there are three types of security vulnerabilities within web applications at different levels: input validation vulnerability at the single request level, session management vulnerability at the session level, and application logic vulnerability at the level of the whole application. In what follows, we describe the three types of vulnerabilities and the common attacks that exploit them.

3.1. Input Validation Vulnerabilities

A common security practice is input data validation, since user input data cannot be trusted. Data validation is the process of ensuring that a program operates on clean, correct, and useful input data. When inputs are not sufficiently or correctly validated, attackers are able to craft malformed inputs, which can alter program executions and gain unauthorized access to resources. Input validation vulnerability is a long-lived problem in software security. Incorrect or insufficient input validation could invite a variety of attacks, such as buffer overflow attacks and code injection attacks.

Web applications may contain a wide range of input validation vulnerabilities. Since the entire web request, including request headers and payload data, is under the complete control of users, a web application has to ensure that user inputs are processed and utilized in a secure way during the execution. Web applications with input validation vulnerabilities are susceptible to a class of attacks usually referred to as script injections, dataflow attacks, or input validation attacks. This type of attack usually embeds malicious scripts within web requests with the goal of injecting them into trusted web contents composed by the web application. As a result, the structural integrity of the web application output is violated. Thus, input validation attacks can manifest as both malformed input and output of the application.

These attacks can be generally categorized by the locations where malicious scripts get executed. For example, directory traversal attacks aim to access unauthorized directories within the local file system by embedding "../" (dot dot slash) characters. OS/command injections happen when the operating system executes malicious shell commands injected through web requests. Currently, input validation attacks are seen as the most dangerous and popular attacks against web application security, which manifest as the top 2 security risks (i.e., injections, cross-site scripting [XSS]) from OWASP top-10 security risks [OWASP Top 10 2013].

In what follows, we review two most popular injection attacks, namely SQL injection and XSS. Most defense techniques against input validation attacks focus on these two attacks.

3.1.1. SQL Injection. A web application is vulnerable to SQL injection attacks when malicious content can flow into SQL queries without being fully sanitized, which allows the attacker to trigger malicious SQL operations by injecting SQL keywords or operators. For example, the attacker can append a separate SQL query to the existing query, causing the application to drop the entire table or manipulate the return result. Malicious SQL statements can be introduced into a vulnerable application using many different input mechanisms [Halfond et al. 2006], including user inputs, cookies, and server variables. A special case of SQL injection is second-order SQL injection, where the attacker stores the malicious content into the database and triggers its execution at a later time. Second-order SQL injection is much more difficult to identify and can bypass insufficient sanitization functions. SQL injections can lead to authentication bypass, information disclosure, and other problems. More details can be found in the survey entirely dedicated to SQL injections [Halfond et al. 2006].

3.1.2. Cross-Site Scripting. A web application is vulnerable to XSS attacks when malicious contents can flow into web responses without being fully sanitized, which allows the attacker to execute malicious scripts in victims' browsers, since the web browser trusts the contents returned by the web application under the same-origin policy. Common consequences of XSS attacks include disclosure of users' sensitive information, such as cookie details and credit card information. XSS also frequently serves as the first step that enables more sophisticated attacks (e.g., the notorious MySpace Samy worm [MySpace 2005]). There are several variants of XSS attacks based on how the malicious scripts are injected. Reflected XSS is launched when the victim clicks a crafted web link, which echoes back the XSS payload through the web application and enables its execution. Persistent (second-order) XSS happens when the malicious scripts are sent to the application back-end database as, for example, forum posts and comments, and stored for a period of time. The malicious scripts are triggered by the victim later when he visits a web page that contains the scripts. DOM-based XSS occurs when the malicious scripts are injected into the client-side JavaScript code for execution, even without sending to the server side. It is worth noting that DOM-based XSS is extremely difficult to handle using only server-side defenses. This does not cover all types of XSS, as there are others left unmentioned (e.g., content-sniffing XSS [Barth et al. 2009] and CSS-based XSS).

3.2. Session Management Vulnerabilities

Session management is essential for a web application to keep track of user inputs and maintain application states. Within the OWASP top-10 security risks [OWASP Top 10 2013], three are related to session management vulnerabilities: (1) Broken Authentication and Session Management, (2) Cross-Site Request Forgery (CSRF), and (3) Insufficient Transport Layer Protection.

In web application development, session management is accomplished through the collaboration between the client and the server. A common approach is that the server sends the client a unique identifier (i.e., a session ID) upon successful user authentication, through which the server recognizes the client on subsequent requests and indexes his session variables stored at the server side. Since session ID is the only proof of the client's identity, its confidentiality, integrity, and authenticity need to be ensured to avoid session hijacking. First, the session ID should be random for each client's visit and expire after a short period of inactivity. Weak session identifier generation allows attackers to hijack the victim's web sessions by predicting his session ID. Second,

transmission of the session ID should always be protected by a secure transport layer protocol (i.e., over SSL). Otherwise, attackers are able to sniff the session ID and hijack the session. Third, the client needs to make sure that his session ID is provided by the server and is unique. Adopting a session ID from an external source opens up a vulnerability to session fixation, where attackers can set the session ID to a value that is known to them.

Securing the session ID alone is not sufficient for secure session management. Session hijacking can also be achieved through malicious web requests that are associated with a valid session ID. CSRF is a popular attack of this type, where attackers trick the victim into sending crafted web requests on their behalf. The vulnerable web application cannot differentiate if the incoming web requests are malicious, since they are associated with valid session information. For example, attackers may forge a web request that instructs a vulnerable banking website to transfer the victim's money to his account. Login CSRF [Gmail CSRF Security Flaw 2007], on the other hand, tricks the victim into logging in to a target website using the attacker's credential through a forged request. This attack allows the attacker to harvest the information about the victim's activities under the attacker's account.

3.3. Application Logic Vulnerabilities

The decentralized structure of web applications poses significant challenges to the implementation of business logic. First, since web application modules can be accessed directly through their URLs, *interface hiding* mechanism has been commonly used as a measure for access control in web applications. However, this mechanism alone, which follows the principle of "security by obscurity," is not sufficient to enforce the control flow of a web application. Attackers can easily exploit hidden links to access unauthorized information or operations. Second, explicit security checks have to be placed by developers manually before all sensitive operations. Since it is difficult to anticipate all possible execution paths leading to sensitive operations, which are dispersed throughout the web application, it is very likely that security checks are missing on certain paths, allowing attackers to access sensitive operations. Third, developers usually send redirection headers to users when security checks fail. However, under certain circumstances, redirection headers do not stop the application execution, which allows the attacker to trigger sensitive operations even though the check fails. This vulnerability is also referred to as Execution After Redirect (EAR) [Doupé et al. 2011]. In AJAX applications, the application functionality is split between the client and the server. Since the computation results at the client side can be manipulated by malicious users, security checks need to be replicated at the server side to validate the results from the client side. Inconsistent or missing validation at the server side may allow attackers to tamper with the application business logic.

Application logic vulnerabilities highly depend on the intended functionality of a web application. For example, a vulnerable e-commerce website may have a specific logic vulnerability that allows attackers to apply the same coupon multiple times to reduce prices. Despite the heterogeneous application functionalities, there are several types of logic flaws that correspond to common business logic patterns in many applications. One common type is access control vulnerability, which allows attackers to access unauthorized sensitive information or operations. Another type is workflow violation, which allows attackers to violate the intended steps within business workflows. For example, a vulnerable e-commerce website may allow attackers to bypass the tax calculation step during the checkout procedure.

Attacks that target application logic vulnerabilities are generally referred to as logic attacks or state violation attacks. Depending on how attacks are launched, they can be given several other terms. Forceful browsing [Sun et al. 2011] is one attack vector,

where attackers directly point to hidden but predictable web links to access sensitive information. Parameter tampering [Bisht et al. 2010a] is launched by manipulating certain values in web requests to exploit application logic.

Since logic attacks are idiosyncratic to their specific targets and manifest as syntactically valid web requests, it is desirable to understand the application logic in order to identify the malicious intentions behind those requests, which makes them extremely challenging to be detected and mitigated. Three of the top-10 security risks for web applications [OWASP Top 10 2013] can be attributed to application logic vulnerabilities (i.e., Missing Functional Access Control, Insecure Direct Object Reference, and Unvalidated Redirects and Forwards).

4. CATEGORIZING EXISTING TECHNIQUES

A large number of techniques have been proposed to secure web applications. These techniques can be carried out during different phases of a web application's life cycle from development, review, and auditing to deployment. They usually have different design objectives depending on the phase of defense. Based on these objectives, we categorize these techniques into the following three classes.

(1) *Secure Construction of New Web Applications*: This class of techniques aims to construct secure web applications, ensuring that no potential vulnerabilities are introduced during the web application development. These techniques are usually carried out through the design of new web programming languages or frameworks that are built with security mechanisms, which root out certain types of vulnerabilities. Although applicable to the development of new web applications, these techniques are not suitable for fixing vulnerable legacy applications due to the huge amount of redevelopment effort that is required.

(2) *Security Analysis/Testing of Legacy Web Applications*: This class of techniques aims to identify vulnerabilities within web applications through program analysis (usually referred to as vulnerability analysis[1]) and testing techniques. Additional efforts have to be devoted to fix the identified vulnerabilities and retrofit the applications, either manually or automatically. They are usually designed to handle a specific programming language or framework and therefore cannot be easily extended to another. A key challenge for this class of techniques is the trade-off between completeness and correctness of vulnerability discovery. Static analysis, which examines application source code without execution, tends to be more complete in vulnerability discovery than dynamic analysis (including testing), which observes the application behavior through execution. On the other hand, static analysis is also more likely to introduce more false alerts, whereas dynamic analysis typically can guarantee the correctness of the identified vulnerabilities through its capability of generating concrete attack vectors.

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

In the following sections, we survey a number of techniques that have been proposed recent years. We organize these techniques along two dimensions. The first dimension

[1]A previous survey [Cova et al. 2007b] specifically focuses on vulnerability analysis.

	Input Validation Vulnerabilities		Session Management Vulnerabilities	Application Logic Vulnerabilities
Secure Construction of New Web Applications		Fisk [2004] McClure and Kruger [2005] Chong et al. [2007a, 2007b] Robertson and Vigna [2009] Samuel et al. [2011] Hooimeijer et al. [2011]	Johnston [2004] Jovanovic et al. [2006a] Barth et al. [2008] Sheridan [2008] Palmer [2008]	Chong et al. [2007a, 2007b] Corcoran et al. [2009] Yip et al. [2009] Krishnamurthy et al. [2010]
Security Analysis/Testing of Legacy Web Applications	Taint analysis	Huang et al. [2004] Minamide [2005] Livshits and Lam [2005] Xie and Aiken [2006] Jovanovic et al. [2006b, 2006c] Wassermann and Su [2007, 2008] Balzarotti et al. [2008] Lam et al. [2008]		Balzarotti et al. [2007] Chaudhuri and Foster [2010] Chlipala [2010] Felmetsger et al. [2010] Sun et al. [2011] Son et al. [2011] Doupé et al. [2011] Bisht et al. [2010a] Bisht et al. [2011] Li and Xue [2013]
	Testing	Huang et al. [2003] Mcallister et al. [2008] Martin and Lam [2008] Kiezun et al. [2009] Saxena et al. [2010a, 2010b] Doupé et al. [2012]		
Runtime Protection of Legacy Web Applications	Taint-based protection	Boyd and Keromytis [2004] Haldar et al. [2005] Nguyen-tuong et al. [2005] Pietraszek et al. [2005] Su and Wassermann [2006] Halfond et al. [2006b] Jim et al. [2007] Chin and Wagner [2009] Sekar [2009] Gundy and Chen [2009] Nadji et al. [2009] Ter Louw and Venkatakrishnan [2009] Stamm et al. [2010] Saxena et al. [2011]		Cova et al. [2007a] Dalton et al. [2009] Parno et al. [2009] Guha et al. [2009] Vikram et al. [2009] Li and Xue [2011] Li et al. [2012]
	Taint-free protection	Scott and Sharp [2002] Kruegel and Vigna [2003] Kruegel et al. [2005] Valeur et al. [2005] Halfond and Orso [2005] Bandhakavi et al. [2007] Ingham et al. [2007] Johns et al. [2008] Bisht and Venkatakrishnan [2008] Song et al. [2009] Scholte et al. [2012]		

Fig. 2. Summary of existing techniques.

is the security vulnerabilities that they address and the attacks that they aim to defeat; the second dimension is the design objectives that they bear and the phases during which they can be carried out. For each technique, we briefly review its design and comment on its unique strengths and limitations. We also discuss open issues that remain underaddressed in each area. Figure 2 shows a summary of techniques covered in this survey.

4.1. Addressing Input Validation Vulnerabilities

The root cause for input validation vulnerabilities is that untrusted user inputs flow into trusted web contents without being sufficiently and correctly sanitized, resulting

in an instance of insecure information flow. The *information flow specification* can be applied to address input validation vulnerabilities. This specification is instantiated for web applications as follows. First, all user inputs are marked as tainted at entry points (i.e., sources) of a web application. Then, the tainted inputs are propagated throughout the application through program statements (e.g., assignment) or functions (i.e., propagators). Before the tainted inputs can reach security-sensitive operation points (i.e., sinks), where they are utilized by the application (e.g., for composing SQL queries or Web responses), they have to be sanitized correctly to become untainted and safe for use.

Two techniques have to be carried out to enforce information flow specification. First, in *untrusted data separation*, untrusted user inputs have to be reliably separated and identified from the trusted web contents before they can be utilized by the web application. In practice, this can be achieved through a number of techniques, such as security-typed languages, strong typing, and taint tracking. Second, in *untrusted data handling*, untrusted user inputs have to be handled and processed by the web application in a secure way, ensuring that the structural integrity of the web contents is preserved. In practice, there are two general approaches to handling user inputs. One is to apply sanitization routines (i.e., *sanitizers*) over user inputs so that they can be trusted and utilized by the web application. Some techniques regard sanitizers as a black box and focus on data separation and tracking, whereas others specifically look into the sanitization functions and focus on the design of context-sensitive sanitization. The other approach is to quarantine or directly drop suspicious user inputs based on certain security policies. This approach is free of sanitization and therefore circumvents all of the complexities and challenges imposed by context-sensitive sanitization.

4.1.1. Secure Construction of New Web Applications. Security-typed language annotates information flows with specific labels and enforces security policies associated with different flows at both compile time (i.e., static checking) and runtime (i.e., dynamic checking). Servlet Information Flow (SIF) [Chong et al. 2007a] is a web application framework based on the security-typed language Jif [Myers et al. n.d.], which extends Java with information flow control and access control. SIF is able to label user input, track the information flow, and enforce the annotated security policies on user inputs at both compile time and runtime. Whereas SIF is applicable to web applications whose functionality is mainly implemented as server-side code, its parallel work Swift [Chong et al. 2007b] considers web applications with client-side code. Swift framework automatically and securely partitions Jif source code into server-side and client-side code and enforces end-to-end information flow policies over code at both sides. SIF and Swift can be used for building secure web applications that are free of input validation vulnerabilities, as long as the security policies associated with the information flow of untrusted user inputs are specified correctly. These two frameworks can also be used to enforce other security policies that are relevant with application logic (e.g., authorization), which is covered in Section 4.3.1.

Robertson and Vigna [2009] propose a strongly typed web development framework to build robust web applications against XSS and SQL injections. Developers specify the intended web document and SQL structure in a strongly typed language Haskell, in order to reliably separate untrusted user inputs from trusted static web contents. User inputs are passed through specific sanitization routines, depending on their types/contexts (e.g., a html tag or an attribute), to ensure the structure integrity of the web documents and SQL queries. As a purely server-side approach, this framework cannot handle client-side DOM-based XSS.

Besides newly developed frameworks, existing frameworks are appended with additional security mechanisms to assist developers in building secure web applications.

To defend against SQL injections, Prepared Statement [Fisk 2004] (or Parameterized Queries) and SQL DOM [McClure and Krüger 2005] are recommended for writing SQL query statements, where untrusted user inputs are filled into the place holders within the explicitly specified structure of SQL queries so that their structural integrity is enforced by interpreters.

Similarly, to eliminate XSS opportunities, HTML template systems (e.g., Google CTemplate) and some development frameworks (e.g., Django) force developers to separate user data from HTML structure explicitly and apply sanitization functions automatically when user inputs are embedded into web responses. These greatly facilitate the secure development of web applications by freeing developers from writing complex and error-prone sanitizers on their own.

All of the preceding techniques aim at identifying and separating untrusted user inputs so that sanitization routines can be applied. In order to root out input validation vulnerabilities, developers also need to ensure the correctness of each individual sanitization routine and apply them in a context-sensitive manner. Currently, the adoption of web development frameworks, which provide automatic context-sensitive sanitization functions, greatly reduces this challenging and error-prone task. However, they are not perfect, as a recent study [Weinberger et al. 2011] shows that input validation vulnerabilities can still arise even after those auto-sanitization functions are utilized. The reason for this is that there exists a large gap between the sanitization requirements for web application development and the actual capabilities provided by those frameworks. For example, web application frameworks have limited expressiveness of contexts and therefore cannot provide accurate context-sensitive sanitization. To address this challenge, Samuel et al. [2011] build a reliable context-sensitive autosanitization engine into web template systems based on type qualifiers. Hooimeijer et al. [2011] present BEK, a new language that enables the development of sanitizer functions for web applications and, more importantly, precise reasoning about the correctness of sanitizers.

4.1.2. Security Analysis/Testing of Legacy Web Applications. Since legacy web applications require developers to identify user inputs and write sanitization functions manually, two types of defects can be introduced: (1) missing sanitization and (2) faulty sanitization. Both taint analysis and testing can be employed to identify input validation vulnerabilities resulting from the preceding error-prone procedure.

Taint analysis aims to identify insecure information flows where user inputs are propagated and flow into sinks within web applications. To perform taint analysis, the set of sources, propagators, sinks, and sanitizers have to be manually specified and modeled, requiring human expertise. In order to identify missing sanitization, sanitizers are usually modeled as a black-box function, which takes the untrusted user inputs and outputs the trusted data. As a result, this technique is inadequate for examining faulty sanitization. Taint analysis usually involves complex and language-specific program analysis techniques to reliably taint and track user inputs. The analysis accuracy is greatly impacted by their capability of handling specific language features, especially when it comes to dynamic scripting languages.

Static taint analysis examines the application source code without execution and is usually employed for identifying missing sanitization. Several program analysis techniques, such as dataflow analysis, pointer analysis, and string analysis, can be applied. Static taint analysis can conservatively identify all possible insecure information flow but has limited capability of modeling the dynamic features of scripting languages (e.g., code inclusion, object-oriented code). Complex alias analysis has to be employed, causing static taint analysis to be inherently inaccurate, and possibly introducing false positives.

Huang et al. [2004] propose WebSSARI, a tool that applies static analysis into identifying vulnerabilities within web applications. The tool employs flow-sensitive, intraprocedural analysis based on a lattice model. They extend the PHP language with two type-states, namely tainted and untainted, and track each variable's type-state. In addition, runtime sanitization functions are inserted into the locations where the tainted data reach the sinks to automatically harden the vulnerable web application. WebSSARI does not support a number of language features, such as recursive functions and array elements.

Xie and Aiken [2006] perform a bottom-up analysis of basic blocks, procedures, and the whole program to find SQL injection vulnerabilities. Their technique employs symbolic execution to automatically derive the set of variables that need to be sanitized before utilized by functions. Their static analysis is also limited to a certain set of language features.

Pixy [Jovanovic et al. 2006b; 2006c] is an open-source tool that performs interprocedural flow-sensitive data flow analysis over PHP web applications. Pixy first constructs a control-flow graph for each function. Then, it performs precise alias and literal analysis on the intermediate nodes. Pixy is the first to apply alias analysis to scripting languages, which greatly improves the analysis precision.

Wassermann and Su [2007, 2008] propose string-taint analysis, which enhances Minamide's string analysis [Minamide 2005] with taint support. Their technique labels and tracks untrusted substrings from user inputs, ensuring that no untrusted scripts can be included in SQL queries and generated HTML pages. Their technique not only addresses missing sanitization but also faulty sanitization performed over user inputs.

Instead of analyzing PHP web applications, Livshits and Lam [2005] apply precise context-sensitive (but flow-insensitive) points-to analysis into analyzing the bytecode of Java web applications based on binary decision diagrams. In particular, they use a high-level declarative language Program Query Language (PQL) for specifying the information flow policy and automating the information flow analysis. This is different from traditional techniques based on type declaration or program assertions.

Dynamic taint analysis tracks the information flow of user inputs during runtime execution through instrumentation and can be used for identifying both missing and faulty sanitization. Compared to static taint analysis, dynamic taint analysis does not require complex code analysis and can better handle dynamic features of scripting languages, thus improving the analysis precision (i.e., fewer false positives). However, it inherits the limitation of dynamic analysis, which cannot guarantee the completeness of the analysis. Subtle vulnerabilities might be missed when the application execution space is not fully explored. Since dynamic tainting is usually employed for runtime monitoring, we will review the existing works using this technique in Section 4.1.3.

Hybrid taint analysis combines the strengths of static and dynamic analysis to improve the analysis precision. Saner [Balzarotti et al. 2008] first analyzes the correctness of both built-in and custom sanitization routines in PHP web applications and demonstrates that faulty sanitization can introduce numerous subtle flaws. Saner applies conservative static string analysis to model how user inputs are sanitized, then feeds a large set of malicious inputs into suspicious sanitization routines to identify weak or incorrect sanitization functions.

Lam et al. [2008] present a holistic technique, which combines static taint analysis, model checking, dynamic taint tracking, and runtime detection. In particular, they developed a model checker Query-based Event Director (QED) for J2EE web applications to improve the accuracy of static taint analysis by systematically exploring the application space and verifying the information flow specification. QED can also generate

concrete attack vectors to prove the correctness of analysis without false positives and facilitate the correction of application flaws.

Testing tries to construct input vectors that expose input validation vulnerabilities within web applications. Usually both benign and malformed user inputs are fed into web applications to check whether the structure of the outputs can be successfully tampered by the intentionally malicious inputs. Since testing is a dynamic analysis technique, it is challenging to generate test inputs that can completely explore the application space and discover all potential vulnerabilities. Some subtle vulnerabilities, such as faulty sanitization, are extremely difficult to uncover without the knowledge of application internal design details (e.g., how the user input is encoded).

Black-box testing/scanning tools, including both open-source (e.g., Spike, Burp) and commercial (e.g., IBM AppScan) products, are particularly attractive and useful when the application source code is unavailable. These tools usually first crawl the web application under test to identify all possible entry points for injection, then feed the application with test inputs that are randomly generated from a library of known attack patterns, and at last evaluate web responses to determine whether vulnerabilities exist. WAVES [Huang et al. 2003] is among the earliest testing frameworks to assess the security of a web application by injecting XSS and SQL injection attack vectors. Mcallister et al. [2008] utilize recorded user sessions to demonstrate that guided and stateful fuzzing can improve the scanner's performance.

Two recent surveys [Doupé et al. 2010; Bau et al. 2010] on black-box scanners demonstrate that black-box scanners are struggling with the issue of deep crawling, which is the practice of exploring the web application completely to reach more vulnerability injection points for fuzzing. Deep crawling is very important for the discovery of certain subtle flaws, such as second-order input validation vulnerabilities.

To improve the performance of current black-box scanners, Doupé et al. [2012] propose to incrementally build an internal state machine during crawling. This inferred state machine is utilized to drive and fuzz the web application. Their technique takes into account the application state changes and is therefore able to discover more subtle vulnerabilities that are hidden behind specific application states.

Traditional fuzzing can be enhanced with program analysis techniques to achieve better coverage and efficiency. For example, Martin and Lam [2008] apply model checking to explore the web application and generating attack vectors. Static analysis is used to prune infeasible execution paths in order to avoid state explosion and enhance input generation efficiency.

ARDILLA [Kiezun et al. 2009] improves input generation efficiency by symbolically tracking sample inputs through execution and only mutating those whose parameters flow into sensitive sinks. In particular, it is capable of tracking tainted data through a database, allowing it to precisely identify second-order XSS vulnerabilities.

FLAX [Saxena et al. 2010a] is a taint-enhanced black-box fuzzing technique that targets at client-side input validation vulnerabilities within JavaScript code and DOM-based XSS attacks. In particular, it applies dynamic taint analysis to extract knowledge about the sinks within JavaScript code and then uses it to prune the input mutation space and direct effective fuzzing.

Kudzu [Saxena et al. 2010b] is another automated vulnerability analysis/testing tool for discovering client-side input validation vulnerabilities. Different from FLAX, which relies on an external manually developed test harness to explore the path space, Kudzu automatically generates a test suite that explores the execution space. Specifically, Kudzu uses dynamic symbolic execution of JavaScript to explore the value space of a program and employs automatic Graphical User Interface (GUI) exploration to cover its event space. The symbolic execution engine used in Kudzu is based on a newly

defined constraint language and its constraint solver in order to provide rich support for reasoning about the operations of JavaScript applications.

4.1.3. Runtime Protection of Legacy Web Applications. In general, there are two approaches to protecting vulnerable web applications from input validation attacks at runtime. One approach requires examining the internals of a web application by tracking and processing (e.g., sanitizing or quarantining) untrusted user inputs, which we refer to as taint-based protection. The other approach observes the external behavior of a web application by evaluating incoming web requests and outgoing web responses, to detect and stop input validation attacks, which we refer to as taint-free protection.

Taint-based protection usually requires instrumentation of either the application source code or the infrastructure components (e.g., interpreter, system library). As a result, this approach may negatively affect an application's runtime performance and stability. Taint mode is first introduced into Perl by extending its interpreter to support dynamic taint tracking and ensure that no external data can be used by critical functions. Nguyen-tuong et al. [2005] modify the PHP interpreter to precisely taint and track user inputs at the granularity of characters. At runtime, untrusted user inputs are sanitized by explicitly calling a new function instrumented into the application. Haldar et al. [2005] instrument the Java system class bytecode to extend Java with taint tracking support. Chin and Wagner [2009] implement efficient character-level tainting via instrumentation of Java library classes and Java Servlet.

Taint tracking can also be achieved without the instrumentation of language runtime or libraries. SQLRand [Boyd and Keromytis 2004] follows the idea of instruction-set randomization [Kc et al. 2003] and separates untrusted user inputs from the SQL static structure by randomizing SQL keywords with secret keys. In this way, attackers cannot inject SQL keywords to tamper the SQL query structure. This method requires an additional SQL proxy to manage the randomization keys, translate "encrypted" SQL queries, and drop malformed ones at runtime.

CSSE [Pietraszek et al. 2005] assigns metadata to user inputs and modifies original operations and functions to preserve the metadata while processing user inputs. CSSE performs context-aware string evaluation to ensure no tainted user inputs can be injected into static web contents, including literals, SQL keywords, and operators.

SQLCheck [Su and Wassermann 2006] taints untrusted user inputs with surrounding special brackets (e.g., "[" and "]") and propagates bracketed user inputs throughout the application. SQL queries are dropped if any bracketed user data spans an SQL keyword, indicating an SQL injection attack.

Halfond et al. [2006b] propose "positive tainting", which taints and tracks trusted strings generated by the application instead of untrusted user inputs (i.e., negative tainting). Positive tainting is more conservative and accurate, since the size of the trusted dataset tends to be much smaller than that of untrusted data.

ScriptGuard [Saxena et al. 2011] also employs positive taint tracking for context-sensitive sanitization. It does this through instrumenting the contexts into the web application and tracking the trusted web contents. ScriptGuard detects and repairs the incorrect placement of sanitizers, which are linked to two instances of input validation flaws for ASP.NET applications: (1) context-mismatched sanitization and (2) inconsistent multiple sanitization. It also leverages a training phase for inferring the correct sanitizer of different contexts at runtime and automatically repairs those broken sanitization functions. One feature of ScriptGard is that it requires no changes to web browsers or to server-side source code. Instead, it uses binary rewriting of server code to embed a browser model that determines the appropriate browser parsing

context when HTML is output by the web application. ScriptGard can serve both as a testing aid to developers as well as a runtime mitigation technique.

Sekar [2009] proposes a black-box taint inference technique to avoid the overhead introduced by deep instrumentation. First, events that traverse across different components/libraries are intercepted, from which dataflows are identified via approximate string matching. Then, dataflows that contain untrusted user data are evaluated over a set of syntax-aware and taint-aware policies. This technique faces challenges when complex operations are performed over user input, in which case dataflows may not be identifiable.

To defend against XSS attacks, pure server-side protection is not adequate. First, the malicious scripts are executed in the browser, and the runtime behavior of the application in the browser cannot be fully anticipated from the server side. Second, subtle differences exist between browsers, which can be exploited to evade server-side defense. Finally, in the client-side XSS, such as DOM-based XSS, the malicious scripts are never sent to the server side, making the server-side approach inapplicable. Therefore, the collaboration between the server and the client is desirable to effectively mitigate XSS. The server side has the complete knowledge of what contents are legitimate and allowed and can help the client identify untrusted data more accurately. The untrusted data can then be handled within the browser. This collaborative approach usually requires changes to the web infrastructure, especially modification of the browser, hindering its adoption.

BEEP [Jim et al. 2007] embeds a whitelist of known-safe scripts into each web page and instructs the instrumented web browser to filter the suspicious scripts. The whitelist approach is similar to positive tainting, where untrusted data can be easily identified. BEEP also protects the whitelist from tampering using script key [Markham 2006]. Obviously, the effectiveness of the static whitelist is limited, especially for handling dynamically constructed scripts. BEEP cannot handle client-side XSS.

Nonespaces [Gundy and Chen 2009] associates elements and attributes of HTML documents with different permissions through a modified web template engine. The permissions are specified in a policy file and protected by different randomized namespaces. HTML documents are verified against the policy file at a proxy to determine whether they should be forwarded to browsers or get dropped. Noncespaces encodes the structure of web documents at a much finer granularity than BEEP [Jim et al. 2007]. Similar to BEEP, Noncespaces cannot handle client-side XSS.

DSI [Nadji et al. 2009] enforces the structure integrity of web documents through parser-level isolation of untrusted data in the browser based on a server-specified policy. Specifically, at the server side, web pages are instrumented so that all sections that may potentially contain user inputs are surrounded with randomized delimiters. At the client side, the static document structure can be robustly interpreted by the modified browser, where suspicious user data is tracked and monitored during execution. This technique can handle client-side XSS effectively.

BLUEPRINT [Ter Louw and Venkatakrishnan 2009] is designed to address the problem of browser inconsistencies in parsing web contents, which can be exploited to launch XSS attacks. At the server side, context representations of user inputs are embedded within web pages. At the client side, web pages are parsed by an external script library, which strips the parsing functionality from the browser and moves it to the server side.

Content Security Policy (CSP) [Stamm et al. 2010] requires developers to specify a special HTTP header (X-Content-Security-Policy) within web responses, which instructs the browser only to load and execute scripts from a whitelist of sources. To date, CSP has been adopted by a number of websites (e.g., Twitter) and mainstream browsers (e.g., Chrome, Firefox) for mitigating injection attacks. There are two major

reasons behind its wide adoption. First, CSP provides fine-granularity control over the web contents so that a variety of attacks, including DOM-based XSS and JSON-based XSS, can be defeated. Second, CSP is backward compatible and enables easy and smooth transition in the infrastructure level.

Taint-free protection examines either incoming web requests or outgoing web responses to detect injection attacks. Two types of security models, negative and positive, can be employed. A negative security model encodes attack patterns as signatures and is usually employed by web application firewalls (e.g., ModSecurity, Imperva), which monitor HTTP traffic between web applications and users, identifying known attacks. This model is accurate and efficient within its detection range but cannot handle zero-day attacks. It also requires expertise to develop and update attack signatures constantly.

On the other hand, a positive model characterizes patterns of normal behaviors for a web application, including web requests, responses, and SQL queries, identifying the deviations from normal patterns as potential attacks. This model can handle unknown attacks. Its detection performance closely depends on the accuracy of normal patterns.

Positive security policy can be manually specified by experts. For example, Scott and Sharp [2002] propose a security gateway that examines HTTP requests in terms of features such as parameter length and special characters, based upon a developer-specified security policy. Their system can be viewed as an additional layer of input validation.

Normal patterns can also be extracted from the application source code. AMNESIA [Halfond and Orso 2005] extracts the structure of legitimate SQL queries from PHP code, based on a Non-Deterministic Finite Automata (NDFA) model. However, the model accuracy is bounded by the flow-insensitive static analysis. It might miss certain attacks when the tampered SQL queries match a legitimate query on a different path.

CANDID [Bandhakavi et al. 2007] employs dynamic analysis to extract an accurate structure of SQL queries by feeding benign candidate inputs into the application. The application is instrumented at each query generation point with a shadow query, which captures the legitimate structure and is compared with the runtime generated ones. Dynamic analysis can monitor the execution paths, allowing for more complete and accurate modeling. Vulnerable query generation statements can also be retrofitted by prepared statements automatically [Bisht et al. 2010b].

Using similar techniques in CANDID, XSS-Guard [Bisht and Venkatakrishnan 2008] generates a shadow page to capture a web application's intent for each web response, containing only the authorized and expected scripts. Any differences between the real constructed page and the shadow page indicate potential script injections.

Normal patterns can be automatically inferred by observing HTTP traffic during a training phase. This technique is usually referred to as anomaly detection. The assumption for anomaly detection is that the resulting behaviors of attacks would deviate from the web application's normal attack-free behaviors sufficiently. The key challenge is how to establish accurate and sensitive normal models so that false positives and false negatives can be minimized.

Kruegel and Vigna [2003] and Krugel et al. [2005] are among the first that apply anomaly detection into detecting web-based attacks. They derive multiple statistical models to characterize different features of normal web requests, such as attribute length, character distribution, and attribute order. In the detection phase, a decision is made by taking into account all of the features of incoming web requests based on the statistical models. To further reduce false positives, anomalies are grouped into attack categories [Robertson et al. 2006]. Further research [Maggi et al. 2009] addressed the concept drift phenomenon in real-world applications. Valeur et al. [2005] extract a similar set of features from normal SQL queries for detecting SQL injections.

Several other works characterize normal web requests by transforming them into a set of tokens and extracting features of web requests at the token level. Ingham et al. [2007] employ deterministic finite automata, whereas Song et al. [2009] use a mixture of Markov chains based on n-gram transitions. A comparative study [Ingham and Inoue 2007] shows that token-based algorithms tend to be more accurate than character-based algorithms, since they are able to capture higher-level structure of web requests than individual characters.

XSSDS [Johns et al. 2008] proposes two techniques for detecting two types of XSS attacks. Reflected XSS attacks can be detected by matching incoming data with outgoing scripts within web responses. Stored XSS attacks can be detected by establishing a set of legitimate scripts during the training phase and identifying unobserved suspicious outgoing scripts.

Scholte et al. [2012] add an additional layer of external validation over HTTP requests to stop XSS and SQL injection attacks. They combine machine learning and static analysis to deduce the type (e.g., URL, integer, token) of HTTP request parameters and apply type-specific validators to identify potential attacks. The performance of their technique is largely limited by the expressiveness of their types.

We note here there are also a number of pure client-side defenses against XSS attacks, such as IE8 XSS filter [Ross 2008], Firefox NoScript plugin [NoScript], Noxes [Kirda et al. 2006], BrowserShield [Reis et al. 2006], CoreScript [Yu et al. 2007], and NoMoXSS [Nentwich et al. 2007]. These focus on client-side approaches and are therefore beyond the scope of this survey.

4.1.4. Summary. Although a substantial amount of efforts have been devoted to addressing input validation vulnerabilities and attacks, several open issues are still not sufficiently addressed, and XSS remains the most popular web attack nowadays. First, web application development frameworks have been increasingly adopted for developing new applications. These often provide autosanitization features and force developers to follow certain defensive programming practices (e.g., using Prepared Statements), eliminating a large portion of potential vulnerabilities and escalating the bar for attackers. However, as a recent study [Weinberger et al. 2011] shows, they still cannot meet all of the requirements posed by modern web applications. Designing and reasoning context-sensitive sanitization routines still require substantial work.

Second, the identification of input validation vulnerabilities from legacy web applications is still challenging. Although taint-based techniques have been demonstrated to be very effective, they cannot be directly applied to a large number of newly developed web applications. In particular, they cannot handle the dynamic and complex features of scripting languages (e.g., object-oriented code) very well and tend to result in false positives. In addition, web applications usually involve several technologies, languages, or components, which makes it even harder to track user information flow and identify subtle second-order attacks.

Black-box testing is a promising alternative, since it is independent of application languages and platforms. However, recent studies [Doupé et al. 2010; Bau et al. 2010] show that most current black-box scanners are still far from perfect. They have limited capabilities in several areas, such as detecting second-order vulnerabilities, handling active contents (e.g., flash, Java Applet), and deep crawling of the application for high coverage.

To address these issues, one single technique tends to be insufficient. We have seen an increasing number of works that combine two or more techniques to achieve better performance, such as hybrid taint analysis [Balzarotti et al. 2008], string taint analysis [Wassermann and Su 2007, 2008], and taint-enhanced fuzzing [Kiezun et al. 2009].

Another alternative is to apply one technique in a novel way, such as positive tainting [Halfond et al. 2006b] or black-box inference [Sekar 2009]. The question of how to combine existing techniques in a creative way to address the limitations of single techniques is an interesting research direction.

4.2. Addressing Session Management Vulnerabilities

Compared to input validation vulnerabilities and application logic vulnerabilities, which are very challenging to tackle and remain a subject of active research, the techniques that address session management vulnerabilities are relatively mature. Many defense mechanisms for session management vulnerabilities have been successfully adopted by web application frameworks for constructing applications. Here, we briefly review some key techniques and tools.

4.2.1. Secure Construction of New Web Applications. To ensure the confidentiality, integrity, and authenticity of session ID and thus avoid session hijacking, several programming practices are recommended [Johnston 2004; Palmer 2008]. First, a cryptographically strong random number generation algorithm needs to be used to generate a unique session ID for an authenticated user in order to keep the session ID random and unpredictable. Second, an automatic logout should be forced after a reasonably short period of inactivity. Third, transmission of the session ID should always be protected by a secure transport layer protocol to avoid traffic sniffing and information disclosure. When cookies are used to carry the session ID, their SECURE attribute should be set as a defensive practice to avoid accidental transmission of sensitive information over non-SSL protocols. Note that cookies may also be stolen through XSS, which is originated from the input validation vulnerabilities within the web application. The security measures against XSS have been extensively discussed in Section 4.1. To foil session fixation attacks, a web application should never trust the value of a session ID that is provided by a client. Instead, it should always create a unique session ID after a user is successfully authenticated and overwrite the session ID presented by the client with its own. Many web application development frameworks (e.g., Django, Rails 2) provide security features for session management. For example, Rails provides several functions to defend against session fixation attacks, including strong session ID generation function, session ID autoexpiration, and reissuance for each visit. It also supports convenient configuration of the SSL connection [Rail].

To defend against CSRF attacks, a secret validation token is commonly used (e.g., Jovanovic et al. [2006a]). Each HTTP request contains a "secret validation token" for the server to determine whether the request comes from an authorized user. This secret validation token should only be known to the server and the client, and hard to guess by attackers. If the validation token is missing from a request or it does not match the expected value, the server should reject the request. There are multiple methods for generating this token. For instance, the server can generate a random nonce for the user upon its first visit. For every subsequent request, the server validates the token. This nonce can be either independent of the session ID or bound to the session ID [Sheridan 2008; Jovanovic et al. 2006a]. This binding can be maintained either through a server-side state table or using the HMAC of the session identifier as a CSRF token, as implemented in the Ruby on Rails framework. Using secret validation tokens to protect against login CSRF [Gmail CSRF Security Flaw 2007], a "pre-session" is first created to bind the secret token before user login. Then, it transitions to a real session after successful user authentication. Other CSRF defense techniques include the use of HTTP Referrer header, which can be used to differentiate the requests coming from the same-site from cross-site requests. For AJAX applications, setting and validating custom HTTP headers using XMLHttpRequest can be applied to defend

against CSRF. To address the privacy concerns that are associated with the usage of Referrer header, Barth et al. [2008] propose to include an "Origin" header with POST requests, which carries the domain value of the URL of the requests. By checking this value, the application ensures that the received requests are not forged by attackers. This technique requires the modification of browsers, which may limit its adoption. There are also several client-side defenses against CSRF, such as RequestRodeo [Johns and Winter 2006] and BEAP [Mao et al. 2009], which are beyond the scope of this survey.

4.2.2. Summary. Session management is fundamental to the authentication and access control of a web application, as it identifies the same user across multiple web requests. Although the techniques to address session management vulnerabilities are relatively mature compared with other vulnerabilities, such vulnerabilities are still prevalent and remain one of the top security threats [OWASP Top 10 2013], largely due to the wide existence of legacy systems and the integration with those systems. Even for the construction of new secure web applications, it still requires consistent efforts from developers to follow secure coding practices to build robust session management mechanisms.

4.3. Addressing Logic Vulnerabilities

As opposed to input validation vulnerabilities that originate from insecure information flow, logic vulnerabilities are multifaceted without a single root cause. Existing works that address logic vulnerabilities have followed two directions: (1) targeting special types of vulnerabilities that are associated with common application functionalities, such as authentication or access control, and (2) aiming at general logic vulnerabilities that can depend on the functionalities of each application (referred to as application-specific logic vulnerabilities hereinafter). In this case, the application's intended functionality (i.e., specification) is required to tackle the logic vulnerabilities. Such a specification can be explicitly specified by developers during software development. In the absence of such a specification, which is commonly seen in practice, it must be inferred from the application implementation. Application specification inference is challenging, since a general method must be able to handle a number of heterogeneous web applications and platforms to minimize manual effort involved.

4.3.1. Secure Construction of New Web Applications. To enforce authorization policies in web applications, existing works have adopted an information flow model to prohibit sensitive information from flowing to unauthorized principals. Consider how the information flow model has been applied to prevent untrusted user inputs from flowing into trusted web contents to address input validation vulnerabilities. SIF [Chong et al. 2007a] and Swift [Chong et al. 2007b], which use security-typed language Jif to track the information flow, provide a unified framework, which can enforce both input validation policies and authorization policies.

SELinks [Corcoran et al. 2009] is a programming framework extending the LINKS web programming language with FABLE [Swamy et al. 2008], a type system for defining and enforcing custom, label-based security policies. Similar to Jif, each type of sensitive data in Fable is annotated with a security label. However, unlike Jif, the semantics of this label are user defined, and programmers can define the interpretation of labels in special enforcement functions that are separated from the rest of the program. FABLE can be used to define and enforce a wide range of policies, including access control, data provenance, and information flow policies, whereas Jif only supports information flow policy.

RESIN [Yip et al. 2009] is a system that allows programmers to specify application-level dataflow assertions using policy objects and define dataflow boundaries using filter objects. RESIN operates within a language runtime (e.g, Python or PHP interpreter).

It tracks application data as it flows through the application, checks dataflow assertions on every executed path, and invokes filter objects when data cross a dataflow boundary, such as writing to a network or a file. A variety of vulnerabilities can be mitigated using RESIN, including script injections or missing access control checks. Compared to SIF [Chong et al. 2007a] and SELinks [Corcoran et al. 2009], RESIN allows programmers to reuse the application's existing code and thus avoids the large amount of annotations and instrumentations required by security-typed languages. However, RESIN cannot track implicit dataflow, such as program control flow or data structure layout, which is the capability available through security-typed languages.

In addition to information flow models, the principle of least privilege and privilege separation have also been applied to facilitate constructions of web applications that minimize the side effects of an attack. Capsules [Krishnamurthy et al. 2010] is a web development framework based on an object-capability language Joe-E [Mettler et al. 2010] for enforcing isolation and facilitating the practice of the principle of least privilege. A web application is partitioned into isolated components, each of which is given a limited set of explicitly specified privileges. This technique minimizes the damages caused by vulnerable components, especially from third-party programs, and facilitates security reviews and verification. However, it cannot guarantee that each application component is free of vulnerabilities.

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

UrFlow [Chlipala 2010] is able to statically verify a variety of security policies, including both information flow and access control policies, within database-backed web applications. UrFlow requires developers to specify policies in the form of SQL queries and employs symbolic execution and theorem proving to automatically verify whether the program behaviors conform to those policies.

Rubyx [Chaudhuri and Foster 2010] is a symbolic execution framework for Ruby-on-Rails web applications. Rubyx allows developers to specify security policies using a set of programming interfaces and verifies those policies automatically. Rubyx is able to identify input validation vulnerabilities, CSRF, insufficient authentication, and application-specific logic flaws, depending on the security policies defined.

When application specifications are not provided by the developers, they have to be first inferred from the implementations. There are two general approaches for specification inference: (1) static analysis, which extracts the logic specification from the application's source code, and (2) dynamic analysis, which generates the specification by observing the application's runtime behavior.

Static Analysis

Sun et al. [2011] perform role-specific analysis of PHP web applications for identifying access control vulnerabilities. They first specify a set of roles with total order. Next, sitemaps are constructed for different roles in a web application based on explicit navigation links. By comparing per-role sitemaps, privileged pages can be identified. Finally, they analyze whether direct access to privileged pages from unauthorized roles is allowed, indicating missing access control checks.

RoleCast [Son et al. 2011] also tries to identify missing access control checks in PHP web applications. In RoleCast, roles are defined based on the common functionality and security logic. The set of user roles are inferred from the partitions of the file contexts on which security sensitive events are control dependent. Within a role, RoleCast identifies

security-critical variables and performs role-specific consistency analysis over security-critical variables to find the missing security checks.

Doupé et al. [2011] address EAR, a particular type of logic vulnerability where an application continues execution after developer-intended redirection, resulting in violation of intended control flow and unauthorized execution. In particular, they present a static analysis technique to identify such vulnerabilities within Ruby-on-Rails web applications. They extract the control flow graph from the application source code and identify the control flow paths, which lead to privileged code after redirection routines, as potential vulnerabilities.

MiMoSA [Balzarotti et al. 2007] identifies workflow violations that are introduced by unintended navigation paths among multiple modules, in addition to dataflow attacks that exploit the input validation vulnerabilities. MiMoSA infers a workflow graph of an application as a specification through two steps. The first step is intramodule analysis (a module is a PHP file in their analysis), which extracts a "state view" of each module by determining its sinks and state conditions before and after the sinks. The second step is intermodule analysis, which links the individual state views to construct the workflow graph of the entire application. Then, they apply model checking over the workflow graph to identify unintended navigation paths.

Dynamic/Hybrid Analysis

Waler [Felmetsger et al. 2010] can automatically discover application-specific logic flaws. First, Waler extracts value-based invariants for session variables and function parameters by observing normal executions and uses them as the logic specification. Model checking techniques are then applied to identify possible violations of inferred invariants. They also filter spurious invariants by analyzing the program control paths and capturing the relationship between session variables and database objects.

Whereas Waler focuses on the analysis of server-side code, NoTamper [Bisht et al. 2010a] aims at detecting parameter tampering opportunities behind web forms within AJAX applications. Such vulnerabilities are caused by the inconsistencies between the client-side and server-side validation. NoTamper extracts the constraints over parameters within forms from the client-side JavaScript code and generates malicious input vectors by negating those constraints. The web responses triggered by both benign and malicious inputs are examined to determine whether forms are vulnerable to parameter tampering. The black-box technique used by NoTamper may produce both false positives and false negatives. WAPTEC [Bisht et al. 2011] enhances NoTamper by applying white-box analysis to the server-side code to reduce false positives and identify the vulnerabilities that NoTamper fails to discover.

LogicScope [Li and Xue 2013] presents a source-code free approach for discovering logic vulnerabilities through examining web responses. It formulates the logic vulnerabilities as the discrepancies between the intended state machine and the actual implementation state machine. However, LogicScope cannot identify logic flaws (e.g., EAR) that tamper with the integrity of the database, since the effects of the malicious inputs are not reflected through web responses.

4.3.3. Runtime Protection of Legacy Web Applications. CLAMP [Parno et al. 2009] addresses access control vulnerabilities that can be exploited by a variety of attacks, including logic attacks, SQL injections, and even web server compromises, and protects sensitive user data by isolating application components running on behalf of different users through virtualization. CLAMP assigns a virtual web server instance to each user's web session so that the user can only access his own data. However, CLAMP cannot be applied to applications with shared data among users. In addition, CLAMP requires a small amount of changes to the application code.

Nemesis [Dalton et al. 2009] addresses a wide range of authentication bypass and access control vulnerabilities in legacy web applications. It uses a shadow authentication system to infer successful user authentication without depending on the potentially vulnerable authentication mechanism in the application. It employs dynamic information flow tracking techniques to track the flow of user credentials through the application's language runtime. The authentication information is further combined with programmer-supplied access control rules to ensure that only properly authenticated users are granted access to any privileged resources or data. Nemesis does not require changes to the application code but requests that the application developers provide annotation information for verifying the authentication credentials and explicitly specify access control policies.

Swaddler [Cova et al. 2007a] applies anomaly detection techniques for the discovery of state violation attacks. In particular, Swaddler establishes statistical models of session variables for each program block during its normal execution, which indicate the application state when that program block is executed. At runtime, this set of statistical models are evaluated to determine whether the application state is legitimate when the current program block is executed. As opposed to Nemesis and CLAMP, which focus on certain types of common logic vulnerabilities (e.g., authentication, access control), Swaddler provides a unified approach for a wide range of application-specific logic vulnerabilities.

BLOCK [Li and Xue 2011] is a black-box approach for inferring the application specification and detecting state violation attacks. It observes the interactions between the clients and the application, and extracts a set of invariants from web requests, responses, and session variables. Then, web requests and responses are evaluated at runtime against the inferred invariants to detect state violation attacks. Similar to Swaddler, BLOCK can address a wide range of application-specific logic vulnerabilities. Whereas Swaddler requires source code instrumentation, BLOCK is independent of the application's source code and programming platform.

SENTINEL [Li et al. 2012] is also a black-box system for detecting logic attacks. It focuses on detecting malicious SQL queries that are triggered at inappropriate application states toward the database. In particular, SENTINEL captures the relationship between the SQL queries and the session variables by extracting a set of invariants during normal executions. SQL queries are then evaluated at runtime against these inferred invariants.

Besides the vulnerabilities that are embedded in the server-side code, recent works [Guha et al. 2009; Vikram et al. 2009] also tackle the logic vulnerabilities within the client-side code for AJAX applications. Guha et al. [2009] extracts a control-flow graph of URLs from the client-side HTML and JavaScript code as the client specification using static analysis. This graph is then used in a reverse proxy to monitor client behaviors and detect malicious activities against server-side web applications.

Ripley [Vikram et al. 2009] detects malicious user behaviors within AJAX applications by leveraging replicated execution. Essentially, the client-side computation is emulated on a trusted server, where each client-side event is transferred to and executed as the replica of the client. The discrepancies between the execution results are flagged as exploits.

4.3.4. Summary. Securing web applications from logic flaws and attacks still remains an underexplored area. Only a limited number of techniques are proposed. Most of them only address a specific type of application logic vulnerabilities, such as authentication and access control vulnerabilities [Doupé et al. 2011; Sun et al. 2011; Son et al. 2011; Parno et al. 2009; Dalton et al. 2009], or inconsistencies between client and server

validations [Bisht et al. 2010a, 2011]. The fundamental difficulty in tackling general logic flaws is the absence of application logic specification. The absence of a general and automatic mechanism for characterizing the application logic is one of the inherent reasons for the inability of most application scanners and firewalls to handle logic flaws and attacks [Doupé et al. 2010; Bau et al. 2010].

Several recent works try to develop a general and systematic method for automatically inferring the specifications for web applications, which in turn facilitates automatic and sound verification of application logic. One of the key observations of these works [Balzarotti et al. 2007; Felmetsger et al. 2010; Cova et al. 2007a; Li and Xue 2011; Li et al. 2012] is that the application's intended behavior is usually revealed under its normal execution, when users follow the navigation paths. In Guha et al. [2009], similar assumption is made for well-behaved clients, where they are expected by the server to invoke the URLs in a particular sequence with particular arguments.

In order to infer the application logic, one class of methods leverages the program source code [Cova et al. 2007a; Felmetsger et al. 2010]. As a result, the inferred specification highly depends on how the application is structured and implemented (e.g., the definition of a program function or block). The accuracy of the inferred specification is also affected by its capability of handling language details. Another class of methods infers the application specification by observing and characterizing the application's external behavior [Li and Xue 2011; Li et al. 2012]. The noisy information observed from the external behaviors may lead to an inaccurate specification through these methods.

5. CONCLUSION

This article provides a comprehensive survey of recent research results in server-side approaches to securing web applications, first describing unique characteristics of web application development, then illustrating three types of vulnerabilities and attacks and focus on discussing three major classes of existing server-side approaches. The article also points out open issues that still need to be addressed.

Web applications have been evolving extraordinarily fast with new programming models and technologies. This results in an ever-changing landscape for web application security with new challenges, which requires substantial and sustained efforts from security researchers. Several major companies [Google; Facebook] have introduced bounty programs to reward finding new vulnerabilities within their websites. Next, we outline several evolving trends and also point out some pioneering works in this area. First, an increasing amount of application code and logic is moving to the client side, which brings new security challenges. Since the client-side code is exposed, attackers are able to gain more knowledge about the application and therefore are more likely to compromise the server-side application state. Several works have been trying to address this problem [Chong et al. 2007a; Saxena et al. 2010a; Guha et al. 2009; Vikram et al. 2009; Bisht et al. 2010a, 2011]. Second, the business logic of web applications is becoming more and more complex, further complicating the task of building secure web applications without logic vulnerabilities. For example, when multiple web applications are integrated through APIs, their interactions may expose logic vulnerabilities [Wang et al. 2011]. Third, an increasing number of web applications are embedding third-party programs or extensions (e.g., iGoogle gadgets, Facebook games). To automatically verify the security of third-party applications and securely integrate them is nontrivial [Krishnamurthy et al. 2010]. Last but not least, new types of attacks are always emerging, such as the HTTP parameter pollution attack [Balduzzi et al. 2011], requiring security professionals to quickly react, otherwise putting a huge number of web applications at risk.

ACKNOWLEDGMENTS

This work was supported by NSF TRUST (the Team for Research in Ubiquitous Secure Technology) Science and Technology Center (CCF-0424422). We specially thank Professor Dinghao Wu, Trey Reece, and anonymous reviewers for providing valuable suggestions and comments in improving the article.

REFERENCES

MySpace. 2005. MySpace Samy Worm. http://namb.la/popular/tech.html.

Marco Balduzzi, Carmen Torrano Gimenez, Davide Balzarotti, and Engin Kirda. 2011. Automated discovery of parameter pollution vulnerabilities in web applications. In *NDSS'11: Proceedings of the 8th Annual Network and Distributed System Security Symposium*.

Davide Balzarotti, Marco Cova, Vika Felmetsger, Nenad Jovanovic, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. 2008. Saner: Composing static and dynamic analysis to validate sanitization in web applications. In *Oakland'08: Proceedings of the 29th IEEE Symposium on Security and Privacy*. 387–401.

Davide Balzarotti, Marco Cova, Viktoria V. Felmetsger, and Giovanni Vigna. 2007. Multi-module vulnerability analysis of web-based applications. In *CCS'07: Proceedings of the 14th ACM Conference on Computer and Communications Security*. 25–35.

Sruthi Bandhakavi, Prithvi Bisht, P. Madhusudan, and V. N. Venkatakrishnan. 2007. CANDID: Preventing SQL injection attacks using dynamic candidate evaluations. In *CCS'07: Proceedings of the 14th ACM Conference on Computer and Communications Security*. 12–24.

Adam Barth, Juan Caballero, and Dawn Song. 2009. Secure content sniffing for web browsers, or how to stop papers from reviewing themselves. In *Oakland'09: Proceedings of the 30th IEEE Symposium on Security and Privacy*. 360–371.

Adam Barth, Collin Jackson, and John C. Mitchell. 2008. Robust defenses for cross-site request forgery. In *CCS'08: Proceedings of the 15th ACM Conference on Computer and Communications Security*. 75–88.

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

Prithvi Bisht, Timothy Hinrichs, Nazari Skrupsky, Radoslaw Bobrowicz, and V. N. Venkatakrishnan. 2010a. 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*.

Prithvi Bisht, A. Prasad Sistla, and V. N. Venkatakrishnan. 2010b. Automatically Preparing Safe SQL Queries. In *FC'10: Proceedings of the 14th International Conference on Financial Cryptography and Data Security*.

Prithvi Bisht, Timothy Hinrichs, Nazari Skrupsky, and V. N. Venkatakrishnan. 2011. WAPTEC: Whitebox analysis of web applications for parameter tampering exploit construction. In *CCS'11: Proceedings of the 18th ACM Conference on Computer and Communications Security*. 575–586.

Prithvi Bisht and V. N. Venkatakrishnan. 2008. XSS-GUARD: Precise dynamic prevention of cross-site scripting attacks. In *DIMVA'08: Proceedings of the 5th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*.

Stephen W. Boyd and Angelos D. Keromytis. 2004. SQLrand: Preventing SQL injection attacks. In *ACNS'04: Proceedings of the 2nd Applied Cryptography and Network Security Conference*. 292–302.

Avik Chaudhuri and Jeffrey S. Foster. 2010. Symbolic security analysis of ruby-on-rails web applications. In *CCS'10: Proceedings of the 17th ACM Conference on Computer and Communications Security*.

Erika Chin and David Wagner. 2009. Efficient character-level taint tracking for Java. In *Proceedings of the 2009 ACM Workshop on Secure Web Services (SWS'09)*. 3–12.

Adam Chlipala. 2010. Static checking of dynamically-varying security policies in database-backed applications. In *OSDI'10: Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*.

Stephen Chong, K. Vikram, and Andrew C. Myers. 2007a. SIF: Enforcing confidentiality and integrity in web applications. In *USENIX'07: Proceedings of the 16th Conference on USENIX Security Symposium*.

Stephen Chong, Jed Liu, Andrew C. Myers, Xin Qi, K. Vikram, Lantian Zheng, and Xin Zheng. 2007b. Secure web applications via automatic partitioning. In *SOSP'07: Proceedings of the 21st ACM SIGOPS Symposium on Operating Systems Principles*. 31–44.

Brian J. Corcoran, Nikhil Swamy, and Michael Hicks. 2009. Cross-tier, label-based security enforcement for web applications. In *SIGMOD'09: Proceedings of the 35th SIGMOD International Conference on Management of Data*. 269–282.

Marco Cova, Davide Balzarotti, Viktoria Felmetsger, and Giovanni Vigna. 2007a. Swaddler: An approach for the anomaly-based detection of state violations in web applications. In *RAID'07: Proceedings of the 10th International Symposium on Recent Advances in Intrusion Detection*. 63–86.

Marco Cova, Viktoria Felmetsger, and Giovanni Vigna. 2007b. Vulnerability analysis of web applications. In *Testing and Analysis of Web Services*, L. Baresi and E. Dinitto (Eds.). Springer.

Michael Dalton, Christos Kozyrakis, and Nickolai Zeldovich. 2009. Nemesis: Preventing authentication and access control vulnerabilities in web applications. In *USENIX'09: Proceedings of the 18th Conference on USENIX Security Symposium*. 267–282.

Adam Doupé, Bryce Boe, Christopher Kruegel, and Giovanni Vigna. 2011. Fear the EAR: Discovering and mitigating execution after redirect vulnerabilities. In *CCS'11: Proceedings of the 18th ACM Conference on Computer and Communications Security*.

Adam Doupé, Ludovico Cavedon, Christopher Kruegel, and Giovanni Vigna. 2012. Enemy of the state: A state-aware black-box vulnerability scanner. In *USENIX'12: Proceedings of the USENIX Security Symposium*. Bellevue, WA.

Adam Doupé, Marco Cova, and Giovanni Vigna. 2010. Why Johnny can't pentest: An analysis of black-box web vulnerability scanners. In *DIMVA'10: Proceedings of the Conference on Detection of Intrusions and Malware and Vulnerability Assessment*.

Facebook. Facebook Bounty Program. https://www.facebook.com/whitehat.

Viktoria Felmetsger, Ludovico Cavedon, Christopher Kruegel, and Giovanni Vigna. 2010. Toward automated detection of logic vulnerabilities in web applications. In *USENIX'10: Proceedings of the 19th USENIX Security Symposium*.

Harrison Fisk. 2004. Prepared Statements. http://en.wikipedia.org/wiki/Prepared_statement.

Joaquin Garcia-Alfaro and Guillermo Navarro-Arribas. 2008. A survey on detection techniques to prevent cross-site scripting attacks on current web applications. In *CRITIS'07: Proceedings of the Second International Conference on Critical Information Infrastructures Security*. 287–298.

Joaquín García-Alfaro and Guillermo Navarro-Arribas. 2009. A survey on cross-site scripting attacks. *CoRR: Computing Research Repository.* http://arxiv.org/abs/0905.4850.

Gmail CSRF Security Flaw. 2007. http://ajaxian.com/archives/gmail-csrf-security-flaw.

Google. Google Bounty Program. http://www.google.com/about/appsecurity/reward-program/.

Arjun Guha, Shriram Krishnamurthi, and Trevor Jim. 2009. Using static analysis for Ajax intrusion detection. In *WWW'09: Proceedings of the 18th International Conference on World Wide Web*. 561–570.

Matthew Van Gundy and Hao Chen. 2009. Noncespaces: Using randomization to enforce information flow tracking and thwart XSS attacks. In *NDSS'09: Proceedings of the 16th Annual Network and Distributed System Security Symposium*.

Vivek Haldar, Deepak Chandra, and Michael Franz. 2005. Dynamic taint propagation for Java. In *ACSAC'05: Proceedings of the 21st Annual Computer Security Applications Conference*. 303–311.

William G. J. Halfond and Alessandro Orso. 2005. AMNESIA: Analysis and Monitoring for NEutralizing SQL-Injection Attacks. In *ASE'05: Proceedings of the 20th IEEE and ACM International Conference on Automated Software Engineering*.

William G. J. Halfond, Jeremy Viegas, and Alessandro Orso. 2006a. A cassification of SQL-injection attacks and countermeasures. In *Proceedings of the International Symposium on Secure Software Engineering*.

William G. J. Halfond, Alessandro Orso, and Panagiotis Manolios. 2006b. Using positive tainting and syntax-aware evaluation to counter SQL injection attacks. In *SIGSOFT'06/FSE-14: Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 175–185.

Pieter Hooimeijer, Benjamin Livshits, David Molnar, Prateek Saxena, and Margus Veanes. 2011. Fast and precise sanitizer analysis with BEK. In *Proceedings of the 20th USENIX Conference on Security (SEC'11)*.

Yao-Wen Huang, Shih-Kun Huang, Tsung-Po Lin, and Chung-Hung Tsai. 2003. Web application security assessment by fault injection and behavior monitoring. In *WWW'03: Proceedings of the 12th International Conference on World Wide Web*. 148–159.

Yao-Wen Huang, Fang Yu, Christian Hang, Chung-Hung Tsai, Der-Tsai Lee, and Sy-Yen Kuo. 2004. Securing web application code by static analysis and runtime protection. In *WWW'04: Proceedings of the 13th International Conference on World Wide Web*. 40–52.

Kenneth L. Ingham and Hajime Inoue. 2007. Comparing anomaly detection techniques for HTTP. In *RAID'07: Proceedings of the 10th International Conference on Recent Advances in Intrusion Detection*. 42–62.

Kenneth L. Ingham, Anil Somayaji, John Burge, and Stephanie Forrest. 2007. Learning DFA representations of HTTP for protecting web applications. *Computer Networks* 51, 1239–1255.

Trevor Jim, Nikhil Swamy, and Michael Hicks. 2007. Defeating script injection attacks with browser-enforced embedded policies. In *WWW'07: Proceedings of the 16th International Conference on World Wide Web*. 601–610.

Martin Johns, Bjorn Engelmann, and Joachim Posegga. 2008. XSSDS: Server-side detection of cross-site scripting attacks. In *ACSAC'08: Proceedings of the 24th Annual Computer Security Applications Conference.* 335–344.

Paul Johnston. 2004. Authentication and Session Management on the Web. http://www.sans.org/reading_room/whitepapers/webservers/authent ication-session-management-web_1545.

Martin Johns and Justus Winter. 2006. RequestRodeo: Client-side protection against session riding. In *OWASP AppSec Europe*.

Nenad Jovanovic, Engin Kirda, and Christopher Kruegel. 2006a. Preventing Cross Site Request Forgery Attacks. In *SecureComm'06: 2nd International Conference on Security and Privacy in Communication Networks*. 1–10.

Nenad Jovanovic, Engin Kirda, and Christopher Kruegel. 2006b. Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities. In *Oakland'06: Proceedings of the 27th IEEE Symposium on Security and Privacy*. 258–263.

Nenad Jovanovic, Engin Kirda, and Christopher Kruegel. 2006c. Precise Alias Analysis for Syntactic Detection of Web Application Vulnerabilities. *ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*.

Gaurav S. Kc, Angelos D. Keromytis, and Vassilis Prevelakis. 2003. Countering code-injection attacks with instruction-set randomization. In *CCS'03: Proceedings of the 10th ACM Conference on Computer and Communications Security*. 272–280.

Adam Kiezun, Philip J. Guo, Karthick Jayaraman, and Michael D. Ernst. 2009. Automatic creation of SQL injection and cross-site scripting attacks. In *ICSE'09: Proceedings of the 31st International Conference on Software Engineering*. 199–209.

Engin Kirda, Christopher Kruegel, Giovanni Vigna, and Nenad Jovanovic. 2006. Noxes: A client-side solution for mitigating cross-site scripting attacks. In *SAC'06: Proceedings of the 2006 ACM Symposium on Applied Computing*. 330–337.

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

Christopher Kruegel and Giovanni Vigna. 2003. Anomaly detection of web-based attacks. In *CCS'03: Proceedings of the 10th ACM Conference on Computer and Communication Security*. 251–261.

Christopher Kruegel, Giovanni Vigna, and William Robertson. 2005. A multi-model approach to the detection of web-based attacks. *Computer Networks* 48, 5 (August 2005), 717–738.

Monica S. Lam, Michael Martin, Benjamin Livshits, and John Whaley. 2008. Securing web applications with static and dynamic information flow tracking. In *PEPM'08: Proceedings of the 2008 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*. 3–12.

Xiaowei Li and Yuan Xue. 2011. BLOCK: A black-box approach for detection of state violation attacks towards web applications. In *ACSAC'11: Proceedings of the 27th Annual Computer Security Applications Conference*.

Xiaowei Li and Yuan Xue. 2013. LogicScope: Automatic discovery of logic vulnerabilities within web applications. In *ASIACCS'13: Proceedings of the 8th ACM Symposium on Information, Computer and Communications Security*.

Xiaowei Li, Wei Yan, and Yuan Xue. 2012. SENTINEL: Securing database from logic flaws in web applications. In *CODASPY'12: Proceedings of the 2nd ACM Conference on Data and Application Security and Privacy*. 25–36.

V. Benjamin Livshits and Monica S. Lam. 2005. Finding security vulnerabilities in Java applications with static analysis. In *USENIX'05: Proceedings of the 14th Conference on USENIX Security Symposium*. 18.

Federico Maggi, William Robertson, Christopher Kruegel, and Giovanni Vigna. 2009. Protecting a moving target: Addressing web application concept drift. In *RAID'09: Proceedings of the 12th International Symposium on Recent Advances in Intrusion Detection*. 21–40.

Ziqing Mao, Ninghui Li, and Ian Molloy. 2009. Defeating cross-site request forgery attacks with browser-enforced authenticity protection. In *FC'09: Proceedings of the 13th International Conference on Financial Cryptography and Data Security*. 238–255.

Gervase Markham. 2006. Content Restrictions. http://www.gerv.net/security/content-restrictions/.

Michael Martin and Monica S. Lam. 2008. Automatic generation of XSS and SQL injection attacks with goal-directed model checking. In *USENIX'08: Proceedings of the 17th Conference on USENIX Security Symposium*. 31–43.

Sean Mcallister, Engin Kirda, and Christopher Kruegel. 2008. Leveraging user interactions for in-depth testing of web applications. In *RAID'08: Proceedings of the 11th International Symposium on Recent Advances in Intrusion Detection*. 191–210.

Russell A. McClure and Ingolf H. Krüger. 2005. SQL DOM: Compile time checking of dynamic SQL statements. In *ICSE'05: Proceedings of the 27th International Conference on Software Engineering*. 88–96.

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

Yasuhiko Minamide. 2005. Static approximation of dynamically generated web pages. In *WWW'05: Proceedings of the 14th International Conference on World Wide Web*. 432–441.

Andrew C. Myers, Lantian Zheng, Steve Zdancewic, Stephen Chong, and Nathaniel Nystrom. n.d. Jif: Java Information Flow. http://www.cs.cornell.edu/jif.

Yacin Nadji, Prateek Saxena, and Dawn Song. 2009. Document structure integrity: A robust basis for cross-site scripting defense. In *NDSS'09: Proceedings of the 16th Annual Network and Distributed System Security Symposium*.

Florian Nentwich, Nenad Jovanovic, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. 2007. Cross-site scripting prevention with dynamic data tainting and static analysis. In *NDSS'07: Proceedings of the 14th Network and Distributed System Security Symposium*.

Anh Nguyen-tuong, Salvatore Guarnieri, Doug Greene, Jeff Shirley, and David Evans. 2005. Automatically hardening web applications using precise tainting. In *Proceedings of the 20th IFIP International Information Security Conference*. 372–382.

NoScript. NoScript Features: Anti-XSS Protection. http://noscript.net/.

OWASP Top 10. 2013. Open Web Application Security Project Top Ten Security Risk (Feburary 2013). http://www.owasp.org/index.php/Top_10_2013

Chris Palmer. 2008. Secure Session Management with Cookies for Web Applications. https://www.isecpartners.com/media/12009/web-session-management.pdf.

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

Tadeusz Pietraszek and Chris Vanden Berghe. 2005. Defending against injection attacks through context-sensitive string evaluation. In *RAID'05: Proceedings of the 8th International Symposium on Recent Advances in Intrusion Detection*.

Rails. Ruby-on-Rails Security Guide. http://guides.rubyonrails.org/security.html.

Charles Reis, John Dunagan, Helen J. Wang, Opher Dubrovsky, and Saher Esmeir. 2006. BrowserShield: Vulnerability-driven filtering of dynamic HTML. In *OSDI'06: Proceedings of the 7th Symposium on Operating Systems Design and Implementation*. 61–74.

William Robertson and Giovanni Vigna. 2009. Static enforcement of web application integrity through strong typing. In *USENIX'09: Proceedings of the 18th Conference on USENIX Security Symposium*. 283–298.

William Robertson, Giovanni Vigna, Christopher Kruegel, and Richard Kemmerer. 2006. Using generalization and characterization techniques in the anomaly-based detection of web attacks. In *NDSS'06: Proceedings of the 13th Network and Distributed System Security Symposium*.

David Ross. 2008. IE 8 XSS Filter Architecture. http://blogs.technet.com/swi/archive/2008/08/19/ie-8-xss-filter-architecture-implementation.aspx.

Mike Samuel, Prateek Saxena, and Dawn Song. 2011. Context-sensitive auto-sanitization in web templating languages using type qualifiers. In *CCS'11: Proceedings of the 18th ACM Conference on Computer and Communications Security*. 587–600.

Prateek Saxena, Steve Hanna, Pongsin Poosankam, and Dawn Song. 2010a. FLAX: Systematic discovery of client-side validation vulnerabilities in rich web applications. In *NDSS'10: Proceedings of the 17th Annual Network and Distributed System Security Symposium*.

Prateek Saxena, Devdatta Akhawe, Steve Hanna, Feng Mao, Stephen McCamant, and Dawn Song. 2010b. A Symbolic Execution Framework for JavaScript. In *SP'10: Proceedings of the 2010 IEEE Symposium on Security and Privacy*. 513–528.

Prateek Saxena, David Molnar, and Benjamin Livshits. 2011. SCRIPTGUARD: Automatic context-sensitive sanitization for large-scale legacy web applications. In *CCS'11: Proceedings of the 18th ACM Conference on Computer and Communications Security*. 601–614.

Theodoor Scholte, William Robertson, Davide Balzarotti, and Engin Kirda. 2012. Preventing input validation vulnerabilities in web applications through automated type analysis. In *COMPSAC'12: Proceedings of the IEEE 36th Annual Computer Software and Applications Conference*.

David Scott and Richard Sharp. 2002. Abstracting application-level web security. In *WWW'02: Proceedings of the 11th International Conference on World Wide Web*. 396–407.

R. Sekar. 2009. An efficient black-box technique for defeating web application attacks. In *NDSS'09: Proceedings of the 16th Annual Network and Distributed System Security Symposium*.

Eric Sheridan. 2008. OWASP CSRFGuard Project. http://www.owasp.org/index.php/CSRF_Guard.

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

Yingbo Song, Angelos D. Keromytis, and Salvatore J. Stolfo. 2009. Spectrogram: A mixture-of-Markov-chains model for anomaly detection in web traffic. In *NDSS'09: Proceedings of the 16th Annual Network and Distributed System Security Symposium*.

Sid Stamm, Brandon Sterne, and Gervase Markham. 2010. Reining in the web with content security policy. In *Proceedings of the 19th International Conference on World Wide Web(WWW'10)*. 921–930.

Zhendong Su and Gary Wassermann. 2006. The essence of command injection attacks in web applications. In *POPL'06: Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 372–382.

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

Nikhil Swamy, Brian J. Corcoran, and Michael Hicks. 2008. Fable: A language for enforcing user-defined security policies. In *Oakland'08: Proceedings of the 29th IEEE Symposium on Security and Privacy*. 369–383.

Shuo Tang, Haohui Mai, and Samuel T. King. 2010. Trust and protection in the Illinois browser operating system. In *OSDI'10: Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*. 1–8.

Mike Ter Louw and V. N. Venkatakrishnan. 2009. Blueprint: Precise browser-neutral prevention of cross-site scripting attacks. In *Oakland'09: Proceedings of the 30th IEEE Symposium on Security and Privacy*.

Fredrik Valeur, Darren Mutz, and Giovanni Vigna. 2005. A learning-based approach to the detection of SQL attacks. In *DIMVA'05: Proceedings of the Conference on Detection of Intrusions and Malware and Vulnerability Assessment*. 123–140.

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

K. Vikram, Abhishek Prateek, and Benjamin Livshits. 2009. Ripley: Automatically securing web 2.0 applications through replicated execution. In *CCS'09: Proceedings of the 16th ACM Conference on Computer and Communications Security*. 173–186.

Helen J. Wang, Chris Grier, Alexander Moshchuk, Samuel T. King, Piali Choudhury, and Herman Venter. 2009. The multi-principal OS construction of the gazelle web browser. In *USENIX'09: Proceedings of the 18th Conference on USENIX Security Symposium*. 417–432.

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

WASS. 2007. 2007 Web Application Security Statistics. http://projects.webappsec.org/w/page/13246989/WebApplication/SecurityStatistics.

Gary Wassermann and Zhendong Su. 2007. Sound and precise analysis of web applications for injection vulnerabilities. In *PLDI'07: Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*. 32–41.

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

Joel Weinberger, Prateek Saxena, Devdatta Akhawe, Matthew Finifter, Richard Shin, and Dawn Song. 2011. A systematic analysis of XSS sanitization in web application frameworks. In *ESORICS'11: Proceedings of the 16th European Symposium on Research in Computer Security*.

WhiteHat. 2010. WhiteHat Website Security Statistic Report 2010. https://www.whitehatsec.com/resource/stats.html.

Yichen Xie and Alex Aiken. 2006. Static detection of security vulnerabilities in scripting languages. In *USENIX'06: Proceedings of the 15th Conference on USENIX Security Symposium*.

Alexander Yip, Xi Wang, Nickolai Zeldovich, and M. Frans Kaashoek. 2009. Improving application security with data flow assertions. In *SOSP'09: Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*. 291–304.

Dachuan Yu, Ajay Chander, Nayeem Islam, and Igor Serikov. 2007. JavaScript instrumentation for browser security. In *POPL'07: Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 237–249.

Received March 2012; revised June 2013; accepted October 2013