A Survey on Web Application Security

Xiaowei Li and Yuan Xue
Department of Electrical Engineering and Computer Science
Vanderbilt University
xiaowei.li, yuan.xue@vanderbilt.edu

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

This paper surveys the area of web application security, with the aim of systematizing the existing techniques into a big picture that promotes future research. We first present the unique aspects in the web application development which bring inherent challenges for building secure web applications. Then we identify three essential security properties that a web application should preserve: *input validity*, *state integrity* and *logic correctness*, and describe the corresponding vulnerabilities that violate these properties along with the attack vectors that exploit these vulnerabilities. We organize the existing research works on securing web applications into three categories based on their design philosophy: *security by construction*, *security by verification* and *security by protection*. Finally, we summarize the lessons learnt and discuss future research opportunities in this area.

I. Introduction

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

As web applications are increasingly used to deliver security 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), the compromise of web applications would result in breaching an enormous amount of information, leading to severe economical losses, ethical and legal consequences. A breach report from Verizon [1] shows that web applications now reign supreme 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 HTTP protocol, web server and server-side application development technologies (e.g., CGI, PHP, ASP), web browser and client-side technologies (e.g., JavaScript, Flash). Web application built and hosted upon such a complex infrastructure faces inherent challenges posed by the features of those components and technologies and the inconsistencies among them. Current widely-used web application development and testing frameworks, on the other hand, offer limited security support. Thus secure web application development is an error-prone process and requires substantial efforts, which could be unrealistic under time-to-market pressure and 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, about 49% of the web applications being reviewed contain vulnerabilities of high risk level and more than 13% of the websites can be compromised completely automatically [2]. A recent report [3] reveals that over 80% of the websites on the Internet have had at least one serious vulnerability.

Motivated by the urgent need for securing web applications, a substantial amount of research efforts have been devoted into this problem with a number of techniques developed for hardening web applications and mitigating the attacks. Many of these techniques make assumptions on the web technologies used in the application development and only address one particular type of security flaws; their prototypes are often implemented and evaluated on limited platforms. A practitioner may wonder whether these techniques are suitable for their scenarios. And if they can not be directly applied, whether these techniques can be extended and/or combined. Thus, it is desirable and urgent to provide a systematic framework for exploring the root causes of web application vulnerabilities, uncovering the connection between the existing techniques, and sketching a big picture of current research frontier in this area. Such a framework would help both new and experienced researcher to better understand web application security challenges and assess existing defenses, and inspire them with new ideas and trends.

In this paper, we survey the state of the art in web application security, with the aim of systematizing the existing techniques into a big picture that promotes future research. Based on the conceptual security framework by Bau and Mitchell [4], we organize our survey using three components for assessing the security of a web application (or equipped with a defense mechanism): system model, threat model and security property. System model describes how a web application works and its unique characteristics; threat model

describes the power and resources attackers possess; security property defines the aspect of the web application behavior intended by the developers. Given a threat model, if one web application fails to preserve certain security property under all scenarios, this application is insecure or vulnerable to corresponding attacks.

This survey covers the techniques which 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., the trust computing base, including OS, web server, interpreter, etc.)*; 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*. We note here that although browser security ([5], [6]) 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. The contributions of this paper are:

(1) We present three aspects in web application development, which poses inherent challenges for building secure web applications, and identify three levels of security properties that a secure web application should hold: *input validity*, *state integrity* and *logic correctness*. Failure of web applications to fulfill the above security properties is the root cause of corresponding vulnerabilities, which allow for successful exploits.

(2) We classify existing research works into three categories: *security by construction*, *security by verification* and *security by protection*, based on their design principle (i.e., constructing vulnerability-free web applications, identifying and fixing vulnerabilities, or protecting vulnerable web applications against exploits at runtime, respectively) and how security properties are assured at different phases in the life cycle of web application. We are not trying to enumerate all the existing works but have covered most of the represented works.

(3) We identify several open issues that are insufficiently addressed in the existing literature. We also discuss future research opportunities in the area of web application security and the new challenges that are expected ahead.

We structure the rest of this paper as follows. We first describe how a web application works and its unique characteristics in Section II. Then, we illustrate three essential security properties that a secure web application should hold, as well as corresponding vulnerabilities and attack vectors in Section III. In Section IV, we categorize and illustrate the state-of-the-art of proposed techniques systematically. Then, in Section V, we discuss future directions for web application security. We conclude our survey paper in Section VI.

II. Understand How A Web Application Works

Web application is a distributed 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 Fig. 1, a web application may consist of code on both the server side and the client side. The server-side code will generate dynamic HTML pages either through execution (e.g., Java servlet, CGI) or interpretation (e.g., PHP, JSP). During the execution of the server-side code, the web application may interact with local file system or back-end database for storing and retrieving data. The client-side code (e.g., in JavaScript) are embedded in the HTML pages, which is executed within the browser. It can communicate with the server-side code (i.e., AJAX) and dynamically updates the HTML pages. In what follows, we describe three unique aspects of the web application development, which differentiate web applications from traditional applications.

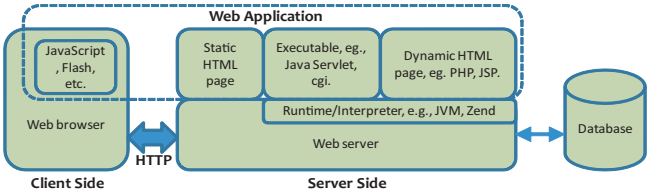

Fig. 1. Overview of Web Application

A. Programming Language

Web application development relies on web programming languages. These languages include scripting languages that are designed specifically for web (e.g., PHP, JavaScript) and extended traditional general-purpose programming languages (e.g., JSP). A distinguishing feature of many web programming languages is their *type systems*. For example, some scripting languages (e.g., PHP) are *dynamically typed*, which means that the type of a variable is determined at runtime, instead of compile time. Some languages (e.g., JavaScript) are *weakly typed*, which means that a statement or a function can be performed on a variety of data types via implicit type casting. Such type systems 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 functions and a web page may embed executable JavaScript code. The representation of application data and code by an unstructured sequence of bytes is a unique feature of web application that helps enhance the development efficiency.

B. State Maintenance

HTTP protocol is stateless, where each web request is independent of each other. However, to implement non-trivial functionalities, "stateful" web applications need to be built on top of this stateless infrastructure. Thus, the abstraction of web session is adopted to help the web application to identify and correlate a series of web requests from the same user during a certain period of time. The state of a web session records the conditions from the historical web requests that will affect the future execution of the web application. The session state can be maintained either at the client side (via cookie, hidden form or URL rewriting) or at the server side.

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. For example, most of web programming languages (e.g., PHP, JSP) 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 pre-defined global array *$_SESSION* is employed to contain the session state. In either case, the client plays a vital role in maintaining the states of a web application.

C. Logic Implementation

The business logic defines the functionality of a web application, which is specific to each application. Such a functionality is manifested as an intended application control flow and is usually integrated with the navigation links of a web application. For example, authentication and authorization are a common part of the control flow in many web applications, through which an web application restricts its sensitive information and privileged operations from unauthorized users. As another example, e-commerce websites usually manage the sequence of operations that the customers need perform during shopping and checkout.

A web application is usually implemented as a number of independent modules, each of which can be directly accessed in any order by a user. This unique feature of web applications significantly complicates the enforcement of the application's control flow across different modules. This task needs to be performed through a tight collaboration of two approaches. The first approach, which is practiced by most web applications, is interface hiding, where only accessible resources and actions of the web application are presented as web links and exposed to users. The second approach requires explicit checks of the application state, which is maintained by session variables (or persistent objects in the database), before sensitive information and operations can be accessed.

III. Understand Web Application Security Properties, Vulnerabilities and Attack Vectors

A secure web application has to satisfy desired security properties under the given threat model. In the area of web application security, the following threat model is usually considered: 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., the trust computing base, including OS, web server, interpreter, etc.)*; 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*. The vulnerabilities within web application implementations may violate the intended security properties and allow for corresponding successful exploits.

In particular, a secure web application should preserve the following stack of security properties, as shown in Fig. 2. *Input validity* means the user input should be validated before it can be utilized by the web application; *state integrity* means the application state should be kept untampered; *logic correctness* means the application logic should be executed correctly as intended by the developers. The above three security properties are related in a way that failure in preserving a security property at the lower level will affect the assurance of the security property at a higher level. For instance, if the web application fails to hold the input validity property, a cross-site scripting attack can be launched by the attacker to steal the victim's session cookie. Then, the attacker can hijack and tamper the victim's web session, resulting in the violation of state integrity property. In the following sections, we describe the three security properties and show how the unique features of web application development complicate the security design for web applications.

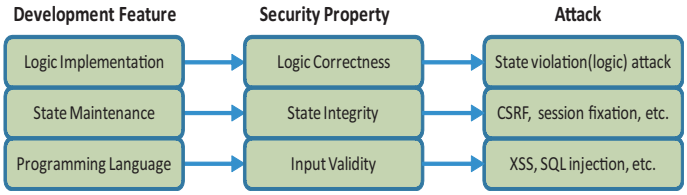

Fig. 2. Web Application Security Properties

A. Input Validity

Given the threat model, user input data cannot be trusted. However, for the untrusted user data to be used in the application (e.g., composing web response or SQL queries), they have to be first validated. Thus, we refer to this security property as **input validity property**:

All the user input should be validated correctly to ensure it is utilized by the web application in the intended way.

The user input validation is often performed via sanitization routines, which transform untrusted user input into trusted data by filtering suspicious characters or constructs within user input. While simple in principle, it is non-trivial to achieve the completeness and correctness of user input sanitization, especially when the web application is programmed using scripting languages. First, since user input data is propagated throughout the application, it has to be tracked all the way to identify all the sanitization points. However, the dynamic features of scripting languages have to be handled appropriately to ensure the correct tracking of user input data. Second, correct sanitization has to take into account the context, which specifies how the user input is utilized by the application and interpreted later either by the web browser or the SQL interpreter. Thus different contexts require distinct sanitization functions. However, the weak typing feature of programming languages makes context-sensitive sanitization challenging and error-prone.

In current web development practices, sanitization routines are usually placed by developers manually in an ad-hoc way, which can be either incomplete or erroneous, and thus introduce vulnerabilities into the web application. Missing sanitization allows malicious user input to flow into trusted

web contents without validation; faulty sanitization allows malicious user input to bypass the validation procedure. A web application with the above vulnerabilities fails to achieve the input validity property, thus is vulnerable to a class of attacks, which are referred to as script injections, data-flow attacks or input validation attacks. This type of attacks embed malicious contents within web requests, which are utilized by the web application and executed later. Examples of input validation attacks include cross-site scripting (XSS), SQL injection, directory traversal, filename inclusion, response splitting, etc. They are distinguished by the locations where malicious contents get executed. In the following, we illustrate the most two popular input validation attacks.

1) SQL Injection: A SQL injection attack is successfully launched when malicious contents within user input flow into SQL queries without correct validation. The database trusts the web application and executes all the queries issued by the application. Using this attack, the attacker is able to embed SQL keywords or operators within user input to manipulate the SQL query structure and result in unintended execution. Consequences of SQL injections include authentication bypass, information disclosure and even the destruction of the entire database. Interested reader can refer to [7] for more details about SQL injection.

2) Cross-Site Scripting: A cross-site scripting (XSS) attack is successfully launched when malicious contents within user input flow into web responses without correct validation. The web browser interprets all the web responses returned by the trusted web application (according to the same-origin policy). Using this attack, the attacker is able to inject malicious scripts into web responses, which get executed within the victim's web browser. The most common consequence of XSS is the disclosure of sensitive information, e.g., session cookie theft. XSS usually serves as the first step that enables further sophisticated attacks (e.g., the notorious MySpace Samy worm [8]). There are several variants of XSS, according to how the malicious scripts are injected, including stored/persistent XSS (malicious scripts are injected into persistent storage), reflected XSS, DOM-based XSS, content-sniffing XSS [9], etc.

B. State Integrity

State maintenance is the basis for building stateful web applications, which requires a secure web application to preserve the integrity of application states. However, The *involvement of an untrusted party (client) in the application state maintenance* makes the assurance of state integrity a challenging issue for web applications.

A number of attack vectors target the vulnerabilities within session management and state maintenance mechanisms of web applications, including cookie poisoning (tampering the cookie information), session fixation (when the session identifier is predictable), session hijacking (when the session identifier is stolen), etc. Cross-site request forgery (i.e., session riding) is a popular attack that falls in this category. In this attack, the attacker tricks the victim into sending crafted web requests with the victim's valid session identifier, however, on the attacker's behalf. This could result in the victim's session being tampered, sensitive information disclosed (e.g., [10]), financial losses (e.g., an attacker may forge a web request that instructs a vulnerable banking website to transfer the victim's money to his account), etc.

To preserve state integrity, a number of effective techniques have been proposed [11]. Client-side state information can be protected by integrity verification through MAC (Message Authentication Code). Session identifiers need to be generated with high randomness (to defend against session fixation) and transmitted over secure SSL protocol (against session hijacking). To mitigate CSRF attacks, web requests can be validated by checking headers (Referrer header, or Origin header [12]) or associated unique secret tokens (e.g., NoForge [13], RequestRodeo [14], BEAP [15]). Since the methods of preserving state integrity are relatively mature, thus falling beyond the scope of this survey.

C. Logic Correctness

Ensuring logic correctness is key to the functioning of web applications. Since the application logic is specific to each web application, it is impossible to cover all the aspects by one description. Instead, a general description that covers most common application functionalities is given as follows, which we refer to as **logic correctness property**:

Users can only access authorized information and operations and are enforced to follow the intended workflow provided by the web application.

To implement and enforce application logic correctly can be challenging due to its state maintenance mechanism and "decentralized" structure of web applications. First, interface hiding technique, which follows the principle of "security by obscurity", is obviously deficient in nature, which allows the attacker to uncover hidden links and directly access unauthorized information or operations or violate the intended workflow. Second, explicit checking of the application state is performed by developers manually and in an ad-hoc way. Thus, it is very likely that certain state checks are missing on unexpected control flow paths, due to those many entry points of the web application. Moreover, writing correct state checks can be error-prone, since not only static security policies but also dynamic state information should be considered. Both missing and faulty state checks introduce logic vulnerabilities into web applications.

A web application with logic flaws is vulnerable to a class of attacks, which are usually referred to as logic attacks or state violation attacks. Since the application logic is specific to each web application, logic attacks are also idiosyncratic to their specific targets. Several attack vectors that fall (or partly) within this category include authentication bypass, parameter tampering, forceful browsing, etc. There are also application-specific logic attack vectors. For example, a vulnerable e-commerce website may allow a same coupon to be applied multiple times, which can be exploited by the attacker to reduce his payment amount.

IV. Categorize Existing Countermeasures

A large number of countermeasures have been developed to secure web applications and defend against the attacks towards web applications. These methods address one or more security properties and instantiate them into concrete security specifications/policies (either explicitly or implicitly) that are to be enforced at different phases in the lifecycle of web applications. We organize existing countermeasures along two dimensions. The first dimension is the security property that these techniques address. The second dimension is their design principle, which we outline as the following three classes:

(1) **Security by construction**: this class of techniques aim to construct secure web applications, ensuring that no potential vulnerabilities exist within the applications. Thus, the desired security property is preserved and all corresponding exploits would fail. They usually design new web programming languages or frameworks that are built with security mechanisms, which automatically enforce the desired security properties. These techniques solve security problems from the root and thus are most robust. However, they are most suitable for new web application development. Rewriting the huge number of legacy applications can be unrealistic.

(2) **Security by verification**: this class of techniques aim to verify if the desired security properties hold for a web application and identify potential vulnerabilities within the application. This procedure is also referred to as vulnerability analysis. Efforts have to be then spent to harden the vulnerable web application by fixing the vulnerabilities and retrofitting the application either manually or automatically. Techniques within this class can be applied to both new and legacy web applications.

Existing program analysis and testing techniques are usually adopted by the works from this class. They have to be overcome a number of technical difficulties in order to achieve the completeness and correctness of vulnerability analysis. In particular, program analysis involves static analysis (i.e., code auditing/review performed on the source code without execution) and dynamic analysis (i.e., observing runtime behavior through execution). Static analysis tends to be complete at identifying all potential vulnerabilities, however, with the price of introducing false alerts. On the other hand, dynamic analysis guarantees the correctness of identified vulnerabilities within explored space, but cannot assure the completeness. Program testing focuses on generating concrete attack vectors that expose expected vulnerabilities within the web application. Similar to dynamic analysis, it also faces the inherent challenge of addressing completeness.

(3) **Security by protection**: this class of techniques aim to protect a potentially vulnerable web application against exploits by building a runtime environment that supports its secure execution. They usually either 1) place safeguards (i.e., proxy) that separate the web application from other components in the Web ecosystem, or 2) instrument the infrastructure components (i.e., language runtime, web browser, etc.) to monitor its runtime behavior and identify/quarantine potential exploits. These techniques can be independent of programming languages or platforms, thus scale well. However, runtime performance overhead is inevitably introduced.

Compared to a previous survey [16], which only focuses on vulnerability analysis, this survey is more comprehensive and covers the complete lifecyle of a web application, from development, auditing/testing to deployment. For each individual technique, we identify its unique strengthes and limitations, compared with other techniques. We also discuss open issues that remain insufficiently addressed. Fig. 3 shows a summary of existing techniques we have covered.

A. Input Validity

We first recall the input validity property:

All the user input should be validated correctly to ensure it is utilized by the web application in the intended way.

The root cause for input validation vulnerabilities is that untrusted user input data flows into trusted web contents without sufficient and correct validation, which is an instance of insecure information flow. Thus, the information flow security model can be naturally applied into addressing the input validity property, which we refer to as *information flow (taint propagation) specification*. This specification is modeled as follows in web applications. First, user input data is marked as tainted at entry points (i.e., *sources*) of the web application. Then, the tainted data flows in the application through certain statements/functions (i.e., *propagators*, such as assignment). Before the tainted data can reach security-sensitive operation points (i.e, *sinks*), where it is utilized by the application (e.g., for composing SQL queries or web responses), it has to be validated and becomes untainted. If the above specification is not enforced, the web application has input validation vulnerability.

To enforce the information flow specification, three tasks have to be performed: (1) *user input identification*, which requires all the untrusted user data to be reliably identified and separated from the trusted web contents; (2) *user input tracking*, which requires the user data to be reliably recognized throughout its flow within the application at a certain granularity; (3) *user input handling*, which requires the user data to be correctly handled, and thus utilized by the application in a secure way. In practice, user input identification and tracking can be achieved through strong typing, variable/byte tainting and tracking, etc. There are two general approaches for handling untrusted user input. One is to transform it into trusted data by sanitization routines (i.e., *sanitizers*), which are usually regarded as a black-box; the other is to quarantine it based on certain predefined security policies, so that potentially malicious user input cannot be executed and the structure integrity of web contents (e.g., web pages or SQL queries) is preserved. Although the latter approach requires certain manual intervention for specifying security policy, it circumvents reasoning about the correctness of sanitization routine, which can be challenging due to its context-sensitiveness.

Property/Technique	Input Validity Property		Logic Correctness Property	
Security by Construction		[17],[19],[20],[24],[81]		[17],[81],[83],[84], [85]
Security by Verification	Program analysis	static:[25],[26],[27],[28], [29],[30],[31],[32],[33] dynamic: [34],[35] hybrid: [36],[37]	Static analysis	[87],[88],[89],[90]
	Program testing	[38],[39],[40],[41],[42],[43]	Dynamic analysis	[91],[92],[93]
Security by Protection	Taint-based protection	[44],[45],[47],[48],[49],[58], [60],[59],[61]		[94],[95],[96],[97],[98],[99]
	Taint-free protection	[62],[63],[64],[67],[68], [69],[71],[72],[75]		

Fig. 3. Summary of existing techniques

1) Security by construction: Chong et al. [17] develop a web application framework SIF (Servlet Information Flow), based on a security-typed language Jif [18], 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 at both compile time and runtime. In addition, their parallel work Swift [19] is a unifying framework to enforce end-to-end information flow policies for distributed web applications. Jif source code can be automatically and securely partitioned into server-side and client-side code. SIF and Swift can be used for building secure web applications free of input validation vulnerabilities, as long as the security policies associated with the information flow of untrusted user data are specified correctly. We note that they can also be used to enforce other security policies that are relevant with application logic (e.g., authorization), which we will explain later.

Robertson et al. [20] propose a strong typing development framework to build robust web applications against XSS and SQL injection. This framework leverages Haskell, a strong typing language, to remedy the weak typing feature of scripting languages. Untrusted user data is reliably distinguished from trusted static web contents via static types and passed through type-specific sanitization routines. Identifying all different user input types and performing correct and accurate sanitization for each type still involve non-trivial manual efforts.

There are also other security mechanisms and defensive programming practices that are proposed to assist developers in building web applications free of input validation vulnerabilities. For example, Prepared Statement [21] (or SQL DOM [22]) are recommended for defending against SQL injections, where the structure of SQL query is explicitly specified by developers and enforced. HTML template systems (e.g., Google CTemplate) force developers to separate user data from HTML structure explicitly, so that auto-sanitization functions are performed before user data can be embedded in web responses. This feature addresses the completeness of user input sanitization, as long as the developers identify and mark all of them. However, the correctness of auto-sanitization routines is overlooked for a long time. A recent study [23] shows that there still exists a large gap between the correctness requirements and the actual capability of sanitization routines provided by template systems and web development frameworks. A very recent work by Samuel et al. [24] builds a reliable context-sensitive auto-sanitization engine into web template systems based on type qualifiers to address this problem.

2) Security by verification: There are broadly two approaches employed by the works in this class: program analysis and program testing. Program analysis techniques aim to identify insecure information flow within the web application. To do so, the set of sources, propagators, sinks and sanitizers have to be first manually specified by the developers, which obviously has great impacts on the analysis precision. In contrast, program testing aims to construct input validation attack vectors to expose vulnerabilities within web applications. Both benign and malformed user input into web applications and their responses are examined to see if there exists structural differences. There are two key challenges for employing testing techniques: (1) it is difficult to generate test cases to completely explore the paths through which user input can reach sinks; (2) it is difficult to generate specially crafted user input to expose subtle vulnerabilities within web applications, such as insufficient sanitization.

Program analysis. This class of techniques include static, dynamic and hybrid analysis.

Static analysis includes several techniques, such as dataflow analysis, pointer analysis, string analysis, etc. Static analysis can conservatively identify all possible insecure information flows, but is limited by its capability of modeling dynamic features of scripting languages, such as code inclusion, object-oriented code. Complex alias analysis has to be employed, which makes static taint analysis inherently inaccurate, leading to a number of false positives.

Huang et al. [25] propose a tool WebSSARI that applies static analysis into identifying vulnerabilities within web applications. The tool employs flow-sensitive, intra-procedural 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 where the tainted data reaches the sinks to automatically harden the vulnerable web application. However, a number of language features are not supported, such as recursive functions, array elements, etc.

Xie and Aiken [26] perform a bottom-up analysis of basic blocks, procedures and the whole program to find SQL injection vulnerabilities. Their technique is able to automatically derive the set of variables that have to be sanitized before function invocation using symbolic execution. However, their static analysis is also limited to a certain set of language features.

Pixy [27], [28] is an open source tool that performs interprocedural flow-sensitive data flow analysis on PHP web applications. Pixy first constructs a parse tree for PHP codes, which is represented as 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 over scripting languages, which greatly improves the analysis precision.

Wassermann et al. [29], [30] propose string-taint analysis, which enhances Minamide's string analysis [31] with taint support. Their technique labels and tracks untrusted substrings from user input and ensures no untrusted scripts can be included in SQL queries and generated HTML pages. Their technique not only addresses the missing sanitization but also the weak sanitization performed over user input.

Instead of analyzing PHP web applications, Livshits and Lam [32] apply precise context-sensitive (but flow-insensitive) points-to analysis into analyzing bytecode of Java web applications based on binary decision diagrams. In particular, they use a high-level language Program Query Language (PQL) for specifying the information flow policy and automate the information flow analysis, distinct from traditional techniques based on type declaration or program assertions.

Similarly, Rubyx [33] requires developers to specify security polices as constraints using the notions of principal, secrecy and trust level and verify those policies for Ruby-on-Rails web applications based on symbolic execution. It is able to identify a number of vulnerabilities, including XSS, CSRF, insufficient access control, as well as application-specific flaws. The completeness of Rubyx is the same as bounded model checking.

Dynamic analysis tracks the information flow of user input during runtime execution by instrumentation. Compared to static analysis, dynamic analysis doesn't require complex code analysis, thus improving the analysis precision. On the other hand, the deep instrumentation may negatively affect the application's performance and stability. Also, the completeness cannot be guaranteed.

Taint mode, which supports dynamic taint tracking, is first introduced into Perl, whose interpreter is extended to ensure that no external data can be used by critical functions. Nguyen-Tuong et al. [34] modify PHP interpreter to precisely taint user data at the granularity of characters and tracks tainted user data at runtime. However, the sanitization of user data requires retrofitting the application source code to explicitly call a newly-defined function, which can be error-prone and affect the analysis precision. Haldar et al. [35] instrument Java system class bytecode to extend Java with taint tracking support.

Hybrid analysis combines strengths of both static and dynamic analysis to further improve the analysis precision. Balzarotti et al. [36] argue that faulty sanitization can introduce numerous subtle flaws into the web application, which cannot be identified by the above techniques. They present Saner, which employs hybrid analysis to validate the correctness of both built-in and custom sanitization routines. Saner first applies conservative static string analysis to model how user input is sanitized, then feeds a large set of malicious inputs into suspicious sanitization routines to identify weak or incorrect sanitization.

Based on [32], Monica et al. [37] present a holistic technique, that combines static analysis, model checking, dynamic checking and runtime detection. In particular, they employ model checking to improve the accuracy of static analysis. Model checking can systematically explores the space of a finite-state system and verifies the correctness of the system with respect to a given property or specification. Model checking is also able to automatically generate concrete attack vectors and exploit paths and produce no false positives.

Program testing. A number of black-box testing tools, also referred to web application scanners, generate input vectors from a library of known attack patterns, including both open-source (e.g., Spike, Burp) and commercial (e.g., IBM AppScan) products. From the research community, WAVES [38] first applies penetration testing into identifying injection vulnerabilities within web applications and leverages machine learning to guide its test case generation. Secubat [39] is another black-box scanner targeting at SQL injection and XSS attacks. McAllister et al. [40] focus on utilizing recorded user sessions for more comprehensive exploration. Black-box testing techniques are essential when the application code is not available, which is a common scenario. They are increasingly deployed as remote web services.

Traditional fuzzing method feeds random inputs into web applications. To improve testing effectiveness, random fuzzing can be enhanced with program analysis techniques. On one hand, fuzzing can generate concrete attack vectors that confirm the presence of vulnerability, thus reducing false alerts. On the other hand, program analysis can guide test case generation for achieving better efficiency and coverage.

Martin et al. [41] apply model checking for systematically generating attacks vectors. Similar to [32], the target vulnerability is specified as PQL queries and instrumented into the application. They leverage Java PathFinder to systematically explore the application via concrete execution. In particular, to address the state explosion problem, the inherent challenge of model checking, they apply static analysis to prune infeasible paths and generate more promising input vectors.

ARDILLA [42] first generates sample inputs, then symbolically tracks tainted inputs through the execution and mutates the inputs, whose parameters flow into sensitive sinks. In particular, it is capable of tracking tainted data through database accesses, which enable it to precisely identify second-order XSS vulnerabilities.

FLAX [43] is a taint-enhanced black-box fuzzing technique that aims to discover CSV (client-side input validation) vul-

nerabilities within JavaScript code. Dynamic taint analysis extracts knowledge of the sinks, which is used to significantly prune the mutation space and direct more effective sink-aware fuzzing.

3) Security by protection: There are two approaches employed by the works in this class. One approach is to follow *information flow specification*, in which untrusted user inputs are identified and tracked, so that the trustworthiness of web contents can be evaluated. We refer to this approach as *taint-based protection*. In this approach, the corrupted web content (e.g., SQL queries) can be directly dropped. Alternatively, suspicious user input can be sanitized, filtered or quarantined, without dropping the entire web content (e.g., web responses). Instead of tracking user input, another approach aims to directly detect input validation attacks before it even reaches the web application or after it triggers a vulnerability in the application (i.e., the structure of web contents has been tampered), which we refer to as *taint-free protection*.

Taint-based protection. *To defend against XSS attacks*, ScriptGuard [44] addresses subtle sanitization errors (i.e., context-mismatched sanitization and inconsistent multiple sanitization) in large-scale and complex web applications. ScriptGuard instruments the web application with an inferred browser model (i.e., the context) when HTML is output and employs positive tainting to conservatively identify sanitization errors. At runtime, ScriptGuard leverages a training phase to learn correct sanitizers for different program paths and achieves auto-repairing of sanitization without incurring significant performance overhead.

However, pure server-side protections are susceptible to the browser inconsistencies and cannot effectively handle client-side XSS attacks (e.g., DOM-based XSS), which are launched during web pages dynamically get updated within the browser. Thus, several techniques require the client-server collaboration, in which the web application conveys certain security policies to be enforced at the browser side.

BEEP [45] embeds a whitelist of known-good scripts into each web page and enforces the policy by filtering suspicious scripts at the instrumented web browser. The whitelist works like tainting the trusted scripts so that untrusted ones can be identified. BEEP also protects the whitelist from being tampered using script key [46]. Although BEEP works efficiently, the whitelist is static and may not accurately differentiate injected scripts from trusted ones.

Matthew et al. [47] propose Noncespace, which annotates the elements and attributes within HTML document into different trust classes using randomized XML namespaces through a modified web template engine. Different trust classes are associated with distinct permissions, specified in a policy file. They set up a proxy to verify the HTML document with the policy file before forwarding it to the web browser. Injected documents will be identified and dropped. Although Noncespace encodes the structure of web documents at a much finer-granularity than BEEP [45], it still cannot detect sophisticated attacks, which are dynamically launched within the web browser due to the static policy.

Yacin et al. [48] propose to enforce the document structure integrity (DSI) of web pages via parser-level isolation (PLI). At the server side, web pages are instrumented (the authors refer to as serialization), where all sections that contain user input data are surrounded with randomized delimiters, before they are sent to the clients. Then, at the client side, the static document structure can be robustly interpreted by the modified web browser, while the suspicious user contents are tracked and monitored during dynamic evaluation and code execution. This technique is robust to a large categories of XSS attacks, including DOM-based XSS, etc. However, it relies on the instrumentation of web browsers.

Louw et al. [49] argue that the trust on the web browser for parsing web pages should be minimized, since the inconsistencies between web browser implementations may allow for server-side defenses to be circumvented by the attackers. Thus, their proposed system Blueprint embeds context representations (i.e., models) of user input into the original web pages and parses the web pages by linking a reliable script library. Their method moves the functions of browser parsers to the server side to ensure that no malicious contents can be executed. However, context-dependent embedding faces the same challenge as context-sensitive sanitization.

There are also a number of pure client-side defenses against XSS attacks, including IE8 XSS filter [50], Firefox NoScript plugin [51], XSSDS [52], Noxes [53], BrowserShield [54], CoreScript [55], NoMoXSS [56], etc. However, as described in the introduction section, this line of works assume a different threat model, thus beyond the scope of this survey paper.

To defend against SQL injections, dynamically generated SQL queries are evaluated to see if user input data has changed the query structure. Following the idea of instruction-set randomization [57], Boyd et al. [58] propose SQLrand to preserve the intended structure of SQL queries and defend against SQL injections. SQLrand separates untrusted user input from SQL structures by randomizing SQL keywords with secret keys, so that the attackers cannot inject SQL keywords to tamper the structure. It uses a SQL proxy to dynamically translate "encrypted" SQL queries and drop injected ones. However, managing randomization keys requires additional efforts.

Su et. al give a formalization of SQL injection and propose SQLCheck [59], which taints untrusted user input with surrounding special brackets and propagates bracketed user input throughout the application. A SQL injection attack is detected if any bracketed data spans a SQL keyword. However, this technique may break some internal functions (e.g., loop, conditional statement, etc.) when the bracketed user input traverses the web application.

Similar to dynamic analysis techniques, Tadeusz et al. [60] propose CSSE to detect injection attacks by tracking user input through meta-data assignment and metadata preserving operations. CSSE performs context-aware string evaluation to ensure no tainted user data can be used as literals, SQL keywords or operators.

Instead of tracking untrusted user input, Halfond et al. [61] propose a novel technique "positive tainting", which taints

and tracks trusted strings generated by the web application and performs syntax-aware string evaluation to detect SQL injections. The advantage of positive tainting is that it is conservative, since the set of trusted data easily converges to be complete, thus tends to be more accurate.

Taint-free protection. This class of techniques usually require an additional phase to establish detection models. To do so, one way is to directly encode the malicious user input patterns (i.e., attack signature), which is referred to as *misuse detection*. Another way is to characterize the benign user input pattern or the structure of web documents and SQL queries intended by the web application and identify the deviation from established models as potential attacks, which is referred to as *anomaly detection*.

Misuse detection employs a set of pre-defined attack signatures to identify known attacks toward web applications. Usually a proxy, also referred to as web application firewall, is set up for monitoring the HTTP interactions between the clients and the application and stopping the attacks from reaching the application. A number of application firewalls, both open source (e.g., ModSecurity) and commercial (e.g., Imperva, Barracuda), are on the market. From the academia, David et al. [62] first propose a security proxy, which examines HTTP requests in terms of parameter lengths, special characters, etc. Signature-based detection is accurate and efficient within its capability range. However, it cannot detect zero-day attacks and requires expertise to develop and update attack signatures.

Anomaly detection assumes that the attacks would cause the web application behavior to deviate sufficiently from that under attack-free circumstances. The key is to establish a model that characterizes the application's normal behavior. Such behavior model needs to be accurate and sensitive. Otherwise, it suffers from false positives and false negatives, respectively. Depending on the target attack, different features of the web application behavior can be examined, such as web request, response, SQL query, etc. and different modeling techniques can be applied.

Kruegel et al. [63], [64] are among the first that apply anomaly detection into detecting web-based attacks. They derive multiple statistical models for normal web requests, in terms of attribute length, character distribution, attribute order, etc. In the detection phase, a web request is blocked if any anomaly score given by those models exceeds the trained threshold. They further reduce false positives by grouping anomalies into specific attack categories based on heuristic [65] and addressing concept drift phenomenon in web applications [66]. Valeur et al. [67] also extract a similar set of features from normal SQL queries, especially for detecting SQL injections.

Instead of examining web requests at the character level, several other works characterize normal web requests by first transforming web requests into a set of tokens. For example, while Ingham et al. [68] employ deterministic finite automata, Song et al. [69] use a mixture of Markov chains based on n-gram transitions. A comparative study [70] shows that token-based algorithms tend to be more accurate, since they are able to capture higher-level structure of web requests than individual characters.

To detect SQL injections, AMNESIA [71] models the structure of legitimate SQL queries. In particular, it builds non-deterministic finite automata (NDFA) models for SQL queries by analyzing the application source code. SQL injections can be detected if the runtime generated query violates its intended structure. However, the model accuracy is bounded by their flow-insensitive static analysis. It may miss certain attacks if the resulting SQL query matches to a legitimate one on a different path.

CANDID [72] uses dynamic techniques to extract more accurate structure of SQL queries by feeding benign candidate inputs into the application. Then, the application is instrumented at each query generation point with a shadow query, which captures its legitimate structure and is compared with runtime generated queries. It also monitors the executed control path during dynamic execution, thus is more complete in modeling SQL queries than learning based technique [67]. The same technique is used for a different application, which automatically retrofits vulnerable SQL query generation into prepared statements [73]. Their technique is also extended to model web responses and detect XSS attacks. XSS-Guard [74] generates a shadow page to capture the web application's intent for each web response, which contains only the authorized and expected scripts. Any differences between the real constructed page and the shadow page indicate potential script injections.

A black-box taint-inference technique is proposed by Sekar [75] for detecting a range of injection attacks, which doesn't require source code and avoids the negative effects introduced by deep instrumentation (e.g., taint tracking). First, the events that traverse across different components/libraries are intercepted, from which data flows are identified through approximate string matching. Then, data flows that contain untrusted user data are evaluated over a set of language-neutral syntax- and taint-aware uniform policies. Policy-based evaluation makes this technique much more accurate than anomaly detection techniques. However, it faces challenges when complex operations are performed over user input, in which case such data flow may not be identifiable.

4) Open Issues: Though a substantial amount of efforts that have been devoted to input validation, several open issues are still not or insufficiently addressed for securing web applications from its related attacks. First although taint-based techniques (i.e., program analysis, taint-based protection) have been demonstrated to be very effective, tracking user input by program annotations still faces technical challenges. For static analysis, it is inherently difficult to handle dynamic and complex features of scripting languages (e.g., object-oriented code). Inaccurate approximation of the application behavior leads to a large number of false positives. Moreover, taint tracking is mostly limited to the application itself. Inability of tracking user input across multiple applications, external libraries, databases [76], etc, will miss certain subtle vulnerabilities and result in stored or second-order attacks.

In terms of handling user input, sanitization, as the most common approach, surprisingly fails to achieve its desired functionality in many web development frameworks [23]. Thus, reasoning the correctness of sanitization routines still requires substantial work ([36], [77], [44], [24]). Policy-based techniques, as another way of handling user input, become promising, since the abstraction of security policies from applications enables security mechanisms to be easily deployed for a number of applications and facilitates security review and verification [78]. However the development of the policy needs non-trivial human involvement.

Black-box application testing is independent of the application source code and platforms and provides a promising scalable method for web application security. However, recent comparative studies [79] [80] show that most of current black-box scanners can only offer security assurance at a certain level and has limited capabilities in several aspects, such as detecting "stored" vulnerabilities (e.g., stored XSS), handling active contents (e.g., flash, Java Applet), deep crawling of the application state and identifying application-specific flaws.

To address the above open issues, only relying on one single technique tends to be insufficient. We have seen an increasing number of works that combine two or several techniques and achieve better performances, such as hybrid taint analysis [36], string-taint analysis [29], [30], taint-enhanced fuzzing [42], etc. Another alternative is to apply one technique in a novel way, such as positive tainting [61], black-box inference [75], etc. How to combine existing techniques in a creative way to address the limits of single techniques is an interesting research direction.

B. Logic Correctness

We first recall the logic correctness property:

Users can only access authorized information and operations and are enforced to follow the intended workflow provided by the web application.

Different from input validation vulnerabilities that originate from insecure information flow, logic vulnerabilities are multi-faceted and specific to web applications. Due to the fact that logic is application specific, there are two scenarios for addressing this property: 1) security policies to be enforced are explicitly specified by developers; 2) security policies are not specified, in which case the specification has to be inferred from the application implementation. In the latter case, the specification inference is the key challenge, especially due to the heterogeneity of logic implementation.

1) Security by construction: Both information flow and access control models can be applied to construct secure web applications that enforce authorization policies. Different from the information flow specification applied for input validation, which prevents untrusted user data from flowing into trusted web contents, the application of information flow model into authorization prohibits sensitive information from flowing to unauthorized principals.

Security typed language, which usually implements a lattice-based type system, annotates data 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). For example, SIF [17] can also be used to enforce authorization policies, in addition to addressing input validity property. Similarly, SELinks [81] is a cross-tier programming framework for building secure and efficient multi-tier web applications, where security policies (e.g., access control, data provenance, information flow, etc.) are specified as customizable labels and a type system Fable [82] is employed to ensure that labeled data/function can only be accessed after checking policies. In particular, SELinks compiler translates customized access control checks into executable SQL queries by the database engine, which greatly improves the efficiency of cross-tier policy enforcement.

Security typed language provides strong security assurance, since it guards both explicit and implicit flow channels. However, it requires a lot of annotations, instrumentation, and even restructuring the application to handle complex and dynamic security policies. Resin [83] is a much lighter-weight approach to ensuring application-specific data-flow security policies at runtime for mitigating both script injections and missing access control checks. Based on a modified language runtime, it attaches *policy objects* to variables, tracks the policy objects flowing through the web application, including persistent storage, and enforces policies through *filter objects*, which guards the boundary between the web application and the external environment. In particular, Resin reuses the original programming language and structure, which greatly facilitates the adoption of Resin for developers. As expected, Resin cannot track implicit flow, such as program control flow, data structure layout, etc., which may miss subtle bugs within applications.

Static checking adds no runtime overhead, while dynamic checking is able to handle complex and dynamic security policies. UrFlow [84] is designed to combine their strengths. In particular, since security policies usually co-locate with application data in the database, e.g., access control matrix, it requires developers to specify security policies in the form of SQL queries. UrFlow is able to perform sound static checking of logic correctness of the application and verify dynamic policies via a *known* predicate. However, it only supports a limited range of authorization policies.

Access control model can be implemented through capability-based system to enforce authorization policies. Capsules [85] is a web development framework based on an object-capability language Joe-E [86] for enforcing privilege separation. The web application is automatically partitioned into isolated components, each of which only exposes limited and explicitly-specified privileges to others. Privilege separation can contain the damages caused by vulnerable components, especially third-party code, and facilitate security reviews and verifications. However, it cannot guarantee each application component free of vulnerabilities.

2) Security by verification: To verify if a web application follows a logic specification, such specification has to be first inferred from its implementation. Static analysis extracts the

specification by analyzing source code, while dynamic analysis observes the application behavior under normal execution. Then, the discrepancies between the inferred specification and the actual implementation are identified as logic vulnerabilities. Obviously, the quality of the inferred specification greatly affects the correctness and accuracy of logic verification.

Static analysis. MiMoSA [87] aims to identify vulnerabilities that are introduced by unintended navigation paths among multiple modules. First, each module (a PHP file in their case) is analyzed to extract the "state view", which represent the influences on state variables by this module. Then, separate state views are concatenated to derive the intended workflow graph. They apply model checking on the workflow graph to identify possible violations of graph traversal, which indicate workflow violation vulnerabilities. However, MiMoSA cannot discover missing or faulty checks within each module.

Similar in purpose as MiMoSA, Sun et al. [88] perform role-specific analysis on PHP web application for identifying access control vulnerabilities. They first specify a set of roles and infer the implicit access control policies by collecting the set of allowed pages for each role, which are exposed through explicit links. Then, they try to directly access other unprivileged pages for each role to identify missing or incorrect access checks.

RoleCast [89] aims to identify missing access control checks at a finer granularity. It first automatically infers the set of user roles for the application by partitioning program files based on a statistical measure. Then, it extracts the set of critical variables that need to be checked for each role. The inconsistencies of checking critical variables at different contexts are reported as vulnerabilities. However, it only models queries that affect the database state (i.e., INSERT, DELETE, UPDATE) as security-sensitive operations and cannot identify faulty checks.

Doupe et al. [90] address a particular type of vulnerability called Execution After Redirect (EAR), where the application continues execution after developer-intended redirection, thus resulting in violation of intended control flow and unauthorized execution. They extract the control flow graph from application source code and identify control paths that lead to privileged code after calling redirection routines.

Dynamic analysis. Waler [91] aims to automatically discover application-specific logic flaws. First, they infer the application specification by deriving value-based likely invariants for session variables and function parameters at each program function via dynamic execution. Then, they perform model checking combined with symbolic execution over the application source code to identify violations of inferred invariants. In particular, they only make use of "reliable" invariants, which are supported by explicit checks along the control path within the code and captures the relationship between session variables and database objects.

Bisht et al. propose a black-box fuzzing approach NoTamper [92] to detect a particular logic vulnerability within form processing functionalities of web applications, which is caused by inconsistent validation of form parameters between the client-side and server-side code. They extract the constraints over form parameters from client-side JavaScript code to generate benign inputs. They also construct malicious inputs by solving negated constraints and feed both into the web application. If their web responses are the same, one vulnerability is found. Their follow-up work WAPTEC [93] enhances the analysis precision by employing white-box analysis and automatically constructs concrete exploits.

3) Security by protection: Nemesis [94] implements dynamic information flow tracking through modifying language runtime to enforce the authentication mechanism and authorization policies in legacy web applications. In particular, it provides reliable evidences for successful authentication when user input meets "known" credentials via a shadow authentication system, thus bypassing the potentially vulnerable authentication mechanisms in the application. It also keeps track of users' credentials to enforce predefined access control policies over resources, including files, database objects, etc.

To provide robust user data segregation, CLAMP [95] employs virtualization technology to isolate the application components running on behalf of different users. CLAMP assigns a virtual web server instance to each user's web session and ensures that the current user can only access his/her own data. Session-level separation provides a certain level of access control assurance. However, it cannot stop the attacks within a single web session, especially in a shared-resource scenario.

Swaddler [96] applies anomaly detection into detection of state violation attacks. It establishes statistical models of session variables for each program block through runtime execution, which indicate the application state when the block is executed. At runtime, the set of models, i.e., the specification, are evaluated to determine if the execution of current program block is an instance of state violation attack.

Arjun et al. [97] extract a control-flow graph from client-side JavaScript code as the specification for well-behaved clients and then set up a proxy for monitoring client behavior and detecting malicious activities against server-side web applications. Ripley [98] is another technique for detecting malicious user behaviors within distributed Ajax web applications by leveraging replicated execution. The client-side computation is exactly emulated on the trusted server side and the discrepancies between computation results are flagged as exploits.

BLOCK [99] 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, which characterize the relationship between web requests, responses and session variables. Then, web requests and responses are evaluated at runtime with the inferred invariants. Compared to Swaddler, BLOCK is independent of the application source code.

4) Open Issues: Securing web applications from logic flaws and attacks still remain an under-explored area. Only a limited number of techniques are proposed. Most of them only address one specific part of application logic flaws [90], [88], [92]. The fundamental difficulty for ensuring application logic correct-

ness property is the absence of application logic specification. As logic is application specific, there is no general model of application logic that is applicable for all applications. The absence of a general and automatic mechanism for characterizing the application logic may be the inherent reason of the inability of application scanners and firewalls at handling logic flaws and attacks [79], [80].

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 class of methods leverage the program source code [96], [91]. As a result, the inferred specification is highly dependent on how the application is structured and implemented (e.g., the definition of a program function or block). Implementation flaws may result in an inaccurate specification. Other method infers the application specification by observing and characterizing the application's external behavior. The noisy information observed from external behaviors may lead to inaccurate specification in this method. Moreover, web application maintains both a large number of persistent states in the database. Correctly identifying these states to accurately characterize the application logic is extremely hard.

V. Conclusion and Future Directions

This paper provided a comprehensive survey of recent research results in the area of web application security. We described unique characteristics of web application development, identified important security properties that secure web applications should preserve and categorized existing works into three major classes. We also pointed out several open issues that still need to be addressed.

Web applications have been evolving extraordinarily fast with new programming models and technologies emerging, resulting in an ever-changing landscape for web application security with new challenges, which requires substantial and sustained efforts from security researchers. We outline several evolving trends and point out several pioneering works as follows. 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, the attacker is able to gain more knowledge about the application, thus more likely to compromise the server-side application state. Several works have been trying to address this problem [19], [43], [97], [98], [92], [93]. Second, the business logic of web applications is becoming more and more complex, which further exacerbates the absence of formal verification and robust protection mechanisms for application logic. For example, when multiple web applications are integrated through APIs, their interactions may expose logic vulnerabilities [100]. Third, an increasing number of web applications are embedding third-party programs or extensions, e.g., iGoogle gadgets, Facebook games etc. To automatically verify the security of third-party applications and securely integrate them is non-trivial [85]. Last but not least, new types of attacks are always emerging, e.g., HTTP parameter pollution attack [101], which requires security professionals to quickly react without putting a huge number of web applications at risk.

References

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

[2] Web Application Security Statistics, "http://projects.webappsec.org/w/page/13246989/WebApplicationSecurityStatistics."

[3] WhiteHat Security, "WhiteHat website security statistic report 2010."

[4] J. Bau and J. C. Mitchell, "Security modeling and analysis," *IEEE Security & Privacy*, vol. 9, no. 3, pp. 18–25, 2011.

[5] H. J. Wang, C. Grier, A. Moshchuk, S. T. King, P. Choudhury, and H. Venter, "The multi-principal os construction of the gazelle web browser," in *USENIX'09: Proceedings of the 18th conference on USENIX security symposium*, 2009, pp. 417–432.

[6] S. Tang, H. Mai, and S. T. King, "Trust and protection in the illinois browser operating system," in *OSDI'10: Proceedings of the 9th USENIX conference on Operating systems design and implementation*, 2010, pp. 1–8.

[7] W. G. Halfond, J. Viegas, and A. Orso, "A Classification of SQL-Injection Attacks and Countermeasures," in *Proc. of the International Symposium on Secure Software Engineering*, March 2006.

[8] MySpace Samy Worm, "http://namb.la/popular/tech.html," 2005.

[9] A. Barth, J. Caballero, and D. Song, "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*, 2009, pp. 360–371.

[10] Gmail CSRF Security Flaw, "http://ajaxian.com/archives/gmail-csrf-security-flaw," 2007.

[11] M. Johns, "Sessionsafe: Implementing xss immune session handling," in *ESORICS'06: Proceedings of the 11th European Symposium On Research In Computer Security*, 2006.

[12] A. Barth, C. Jackson, and J. C. Mitchell, "Robust defenses for cross-site request forgery," in *CCS'08: Proceedings of the 15th ACM conference on Computer and communications security*, 2008, pp. 75–88.

[13] N. Jovanovic, E. Kirda, and C. Kruegel, "Preventing cross site request forgery attacks," in *SecureComm'06: 2nd International Conference on Security and Privacy in Communication Networks*, 2006, pp. 1 –10.

[14] M. Johons and J. Winter, "Requestrodeo: Client-side protection against session riding," in *OWASP AppSec Europe*, 2006.

[15] Z. Mao, N. Li, and I. Molloy, "Defeating cross-site request forgery attacks with browser-enforced authenticity protection," in *FC'09: 13 th International Conference on Financial Cryptography and Data Security*, 2009, pp. 238–255.

[16] M. Cova, V. Felmetsger, and G. Vigna, "Vulnerability Analysis of Web Applications," in *Testing and Analysis of Web Services*, L. Baresi and E. Dinitto, Eds. Springer, 2007.

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

[18] L. Z. Andrew C. Myers, "Jif: Java information flow." [Online]. Available: http://www.cs.cornell.edu/jif

[19] S. Chong, J. Liu, A. C. Myers, X. Qi, K. Vikram, L. Zheng, and X. Zheng, "Secure web applications via automatic partitioning," in *SOSP '07: Proceedings of the 21st ACM SIGOPS symposium on Operating systems principles*, 2007, pp. 31–44.

[20] W. Robertson and G. Vigna, "Static enforcement of web application integrity through strong typing," in *USENIX'09: Proceedings of the 18th conference on USENIX security symposium*, 2009, pp. 283–298.

[21] H. Fisk., "Prepared Statements," 2004. [Online]. Available: http://dev.mysql.com/tech-resources/articles/4.1/prepared-statements.html

[22] R. A. McClure and I. H. Krüger, "Sql dom: compile time checking of dynamic sql statements," in *ICSE'05: Proceedings of the 27th international conference on Software engineering*, 2005, pp. 88–96.

[23] J. Weinberger, P. Saxena, D. Akhawe, M. Finifter, R. Shin, and D. Song, "A Systematic Analysis of XSS Sanitization in Web Application Frameworks," in *ESORICS'11: Proc. of 16th European Symposium on Research in Computer Security*, 2011.

[24] M. Samuel, P. Saxena, and D. Song, "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*, 2011, pp. 587–600.

[25] Y.-W. Huang, F. Yu, C. Hang, C.-H. Tsai, D.-T. Lee, and S.-Y. Kuo, "Securing web application code by static analysis and runtime protection," in *WWW'04: Proceedings of the 13th international conference on World Wide Web*, 2004, pp. 40–52.

[26] Y. Xie and A. Aiken, "Static detection of security vulnerabilities in scripting languages," in *USENIX'06: Proceedings of the 15th conference on USENIX Security Symposium*, 2006.

[27] N. Jovanovic, C. Kruegel, and E. Kirda, "Pixy: A static analysis tool for detecting web application vulnerabilities (short paper)," in *Oakland'06: Proceedings of the 27th IEEE Symposium on Security and Privacy*, 2006, pp. 258–263.

[28] ——, "Precise alias analysis for syntactic detection of web application vulnerabilities," in *ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, 2006.

[29] G. Wassermann and Z. Su, "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*, 2007, pp. 32–41.

[30] ——, "Static detection of cross-site scripting vulnerabilities," in *ICSE'08: ACM/IEEE 30th International Conference on Software Engineering*, 2008.

[31] Y. Minamide, "Static approximation of dynamically generated web pages," in *WWW'05: Proceedings of the 14th international conference on World Wide Web*, 2005, pp. 432–441.

[32] V. B. Livshits and M. S. Lam, "Finding security vulnerabilities in java applications with static analysis," in *USENIX'05: Proceedings of the 14th conference on USENIX Security Symposium*, 2005, p. 18.

[33] A. Chaudhuri and J. S. Foster, "Symbolic security analysis of ruby-on-rails web applications," in *CCS '10: Proceedings of the 17th ACM conference on Computer and communications security*, 2010.

[34] A. Nguyen-tuong, S. Guarnieri, D. Greene, J. Shirley, and D. Evans, "Automatically hardening web applications using precise tainting," in *Proc. of the 20th IFIP International Information Security Conference*, 2005, pp. 372–382.

[35] V. Haldar, D. Chandra, and M. Franz, "Dynamic taint propagation for java," in *ACSAC '05: Proceedings of the 21st Annual Computer Security Applications Conference*, 2005, pp. 303–311.

[36] D. Balzarotti, M. Cova, V. Felmetsger, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna, "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*, 2008, pp. 387–401.

[37] M. S. Lam, M. Martin, B. Livshits, and J. Whaley, "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*, 2008, pp. 3–12.

[38] Y.-W. Huang, S.-K. Huang, T.-P. Lin, and C.-H. Tsai, "Web application security assessment by fault injection and behavior monitoring," in *WWW'03: Proceedings of the 12th international conference on World Wide Web*, 2003, pp. 148–159.

[39] S. Kals, E. Kirda, C. Kruegel, and N. Jovanovic, "Secubat: a web vulnerability scanner," in *WWW'06: Proceedings of the 15th international conference on World Wide Web*, 2006, pp. 247–256.

[40] S. Mcallister, E. Kirda, and C. Kruegel, "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*, 2008, pp. 191–210.

[41] M. Martin and M. S. Lam, "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*, 2008, pp. 31–43.

[42] A. Kieyzun, P. J. Guo, K. Jayaraman, and M. D. Ernst, "Automatic creation of sql injection and cross-site scripting attacks," in *ICSE '09: Proceedings of the 31st International Conference on Software Engineering*, 2009, pp. 199–209.

[43] P. P. Prateek Saxena, Steve Hanna and D. Song, "Flax: Systematic discovery of client-side validation vulnerabilities in rich web applications." in *NDSS'10: Proceedings of the 17th Annual Network and Distributed System Security Symposium*, 2010.

[44] P. Saxena, D. Molnar, and B. Livshits, "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*, 2011, pp. 601–614.

[45] T. Jim, N. Swamy, and M. Hicks, "Defeating script injection attacks with browser-enforced embedded policies," in *WWW '07: Proceedings of the 16th international conference on World Wide Web*, 2007, pp. 601–610.

[46] G. Markham, "Content restrictions." 2006. [Online]. Available: http://www.gerv.net/security/content-restrictions/

[47] M. V. Gundy and H. Chen, "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*, 2009.

[48] Y. Nadji, P. Saxena, and D. Song, "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*, 2009.

[49] M. Ter Louw and V. Venkatakrishnan, "Blueprint: Precise browser-neutral prevention of cross-site scripting attacks," in *Oakland'09: Proceedings of the 30th IEEE Symposium on Security and Privacy*, 2009.

[50] D. Ross, "IE 8 XSS filter architecture." [Online]. Available: http://blogs.technet.com/swi/archive/2008/08/19/ie-8-xss-filter-architecture-implementation.aspx

[51] G. Maone, "NoScript features: Anti-XSS protection." [Online]. Available: http://noscript.net/feature-xss

[52] M. Johns, B. Engelmann, and J. Posegga, "Xssds: Server-side detection of cross-site scripting attacks," 2008, pp. 335–344.

[53] E. Kirda, C. Kruegel, G. Vigna, and N. Jovanovic, "Noxes: a client-side solution for mitigating cross-site scripting attacks," in *SAC '06: Proceedings of the 2006 ACM symposium on Applied computing*, 2006, pp. 330–337.

[54] C. Reis, J. Dunagan, H. J. Wang, O. Dubrovsky, and S. Esmeir, "Browsershield: vulnerability-driven filtering of dynamic html," in *OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation*, 2006, pp. 61–74.

[55] D. Yu, A. Chander, N. Islam, and I. Serikov, "Javascript instrumentation for browser security," in *POPL '07: Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 2007, pp. 237–249.

[56] F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna, "Cross-site scripting prevention with dynamic data tainting and static analysis," in *NDSS'07: Proceeding of the 14th Network and Distributed System Security Symposium*, 2007.

[57] G. S. Kc, A. D. Keromytis, and V. Prevelakis, "Countering code-injection attacks with instruction-set randomization," in *CCS '03: Proceedings of the 10th ACM conference on Computer and communications security*, 2003, pp. 272–280.

[58] S. W. Boyd and A. D. Keromytis, "Sqlrand: Preventing sql injection attacks," in *ACNS'04: Proceedings of the 2nd Applied Cryptography and Network Security Conference*, 2004, pp. 292–302.

[59] Z. Su and G. Wassermann, "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*, 2006, pp. 372–382.

[60] T. Pietraszek, C. V. Berghe, C. V, and E. Berghe, "Defending against injection attacks through context-sensitive string evaluation," in *RAID'05: Proceedings of the 8th International Symposium on Recent Advances in Intrusion Detection*, 2005.

[61] W. G. J. Halfond, A. Orso, and P. Manolios, "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*, 2006, pp. 175–185.

[62] D. Scott and R. Sharp, "Abstracting application-level web security," in *WWW '02: Proceedings of the 11th international conference on World Wide Web*, 2002, pp. 396–407.

[63] C. Kruegel and G. Vigna, "Anomaly Detection of Web-based Attacks," in *CCS'03: Proceedings of the 10th ACM Conference on Computer and Communication Security*, 2003, pp. 251–261.

[64] C. Kruegel, G. Vigna, and W. Robertson, "A Multi-model Approach to the Detection of Web-based Attacks," *Computer Networks*, vol. 48, no. 5, pp. 717–738, August 2005.

[65] W. Robertson, G. Vigna, C. Kruegel, and R. Kemmerer, "Using Generalization and Characterization Techniques in the Anomaly-based Detection of Web Attacks," in *NDSS'06: Proceeding of the 13th Network and Distributed System Security Symposium*, 2006.

[66] F. Maggi, W. Robertson, C. Kruegel, and G. Vigna, "Protecting a moving target: Addressing web application concept drift," in *RAID'09: Proceedings of the 12th International Symposium on Recent Advances in Intrusion Detection*, 2009, pp. 21–40.

[67] F. Valeur, D. Mutz, and G. Vigna, "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*, 2005, pp. 123–140.

[68] K. L. Ingham, A. Somayaji, J. Burge, and S. F. A. C, "Learning dfa representations of http for protecting web applications," *Computer Networks*, vol. 51, pp. 1239–1255, 2007.

[69] A. D. K. Yingbo Song and S. J. Stolfo, "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*, 2009.

[70] K. L. Ingham and H. Inoue, "Comparing anomaly detection techniques for http," in *RAID'07: Proceedings of the 10th international conference on Recent advances in intrusion detection*, 2007, pp. 42–62.

[71] W. G. Halfond and A. Orso, "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*, 2005.

[72] S. Bandhakavi, P. Bisht, P. Madhusudan, and V. N. Venkatakrishnan, "Candid: preventing sql injection attacks using dynamic candidate evaluations," in *CCS '07: Proceedings of the 14th ACM conference on Computer and communications security*, 2007, pp. 12–24.

[73] P. Bisht, A. P. Sistla, and V. Venkatakrishnan, "Automatically preparing safe sql queries," in *FC'10: Proceedings of the 14th International Conference on Financial Cryptography and Data Security*, 2010.

[74] P. Bisht and V. Venkatakrishnan, "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 Assesment*, 2008.

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

[76] B. Davis and H. Chen, "Dbtaint: cross-application information flow tracking via databases," in *WebApps'10: Proceedings of the 2010 USENIX conference on Web application development*, 2010.

[77] P. Hooimeijer, B. Livshits, D. Molnar, P. Saxena, and M. Veanes, "Fast and precise sanitizer analysis with bek," in *USENIX'11: Proceedings of the 20th USENIX Security symposium*, 2011.

[78] J. Weinberger, A. Barth, and D. Song, "Towards client-side html security policies," in *HotSec'11: Proc. of 6th USENIX Workshop on Hot Topics in Security*, 2011.

[79] A. Doupe, M. Cova, and G. Vigna, "Why Johnny Cant 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*, 2010.

[80] J. Bau, E. Bursztein, D. Gupta, and J. Mitchell, "State of the art: Automated black-box web application vulnerability testing," *Oakland'10: Proceedings of the 31st IEEE Symposium on Security and Privacy*, pp. 332–345, 2010.

[81] B. J. Corcoran, N. Swamy, and M. Hicks, "Cross-tier, label-based security enforcement for web applications," in *SIGMOD '09: Proceedings of the 35th SIGMOD international conference on Management of data*, 2009, pp. 269–282.

[82] N. Swamy, B. J. Corcoran, and M. Hicks, "Fable: A language for enforcing user-defined security policies," in *Oakland '08: Proceedings of the 29th IEEE Symposium on Security and Privacy*.

[83] A. Yip, X. Wang, N. Zeldovich, and M. F. Kaashoek, "Improving application security with data flow assertions," in *SOSP'09: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, 2009, pp. 291–304.

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

[85] A. Krishnamurthy, A. Mettler, and D. Wagner, "Fine-grained privilege separation for web applications," in *WWW'10: Proceedings of the 19th international conference on World Wide Web*, 2010, pp. 551–560.

[86] D. W. A. Mettler and T. Close, "Joe-e: A security-oriented subset of java," in *NDSS'10: Proceedings of the 17th Annual Network and Distributed System Security Symposium*, 2010, pp. 357–374.

[87] D. Balzarotti, M. Cova, V. V. Felmetsger, and G. Vigna, "Multi-module vulnerability analysis of web-based applications," in *CCS '07: Proceedings of the 14th ACM conference on Computer and communications security*, 2007, pp. 25–35.

[88] F. Sun, L. Xu, and Z. Su, "Static detection of access control vulnerabilities in web applications," in *USENIX'11: Proceedings of the 20th USENIX Security Symposium*, 2011.

[89] S. Son, K. S. McKinley, and V. Shmatikov, "Rolecast: finding missing security checks when you do not know what checks are," in *OOPSLA '11: Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2011, pp. 1069–1084.

[90] C. K. Adam Doupé, Bryce Boe and G. Vigna, "Fear the EAR: Discovering and Mitigating Execution After Redirect Vulnerabilities," in *CCS'11: Proceeding of the 18th ACM Conference on Computer and Communications Security*, 2011.

[91] V. Felmetsger, L. Cavedon, C. Kruegel, and G. Vigna, "Toward Automated Detection of Logic Vulnerabilities in Web Applications," in *USENIX'10: Proceedings of the 19th USENIX Security Symposium*, 2010.

[92] P. Bisht, T. Hinrichs, N. Skrupsky, R. Bobrowicz, and V. N. Venkatakrishnan, "Notamper: automatic blackbox detection of parameter tampering opportunities in web applications," in *CCS '10: Proceedings of the 17th ACM conference on Computer and communications security*, 2010.

[93] P. Bisht, T. Hinrichs, N. Skrupsky, and V. N. Venkatakrishnan, "Waptec: whitebox analysis of web applications for parameter tampering exploit construction," in *CCS'11: Proceedings of the 18th ACM conference on Computer and communications security*, 2011, pp. 575–586.

[94] M. Dalton, C. Kozyrakis, and N. Zeldovich, "Nemesis: preventing authentication & access control vulnerabilities in web applications," in *USENIX'09: Proceedings of the 18th conference on USENIX security symposium*, 2009, pp. 267–282.

[95] B. Parno, J. M. McCune, D. Wendlandt, D. G. Andersen, and A. Perrig, "CLAMP: Practical prevention of large-scale data leaks," in *Oakland'09: Proceedings of the 30th IEEE Symposium on Security and Privacy*, 2009.

[96] M. Cova, D. Balzarotti, V. Felmetsger, and G. Vigna, "Swaddler: An Approach for the Anomaly-based Detection of State Violations in Web Applications," in *RAID'07: Proceedings of the 10th International Symposium on Recent Advances in Intrusion Detection*, 2007, pp. 63–86.

[97] A. Guha, S. Krishnamurthi, and T. Jim, "Using static analysis for ajax intrusion detection," in *WWW'09: Proceedings of the 18th international conference on World Wide Web*, 2009, pp. 561–570.

[98] K. Vikram, A. Prateek, and B. Livshits, "Ripley: automatically securing web 2.0 applications through replicated execution," in *CCS '09: Proceedings of the 16th ACM conference on Computer and communications security*, 2009, pp. 173–186.

[99] X. Li and Y. Xue, "BLOCK: A Black-box Approach for Detection of State Violation Attacks Towards Web Applications," in *ACSAC'11: Proceedings of 27th Annual Computer Security Applications Conference*, 2011.

[100] R. Wang, S. Chen, X. Wang, and S. Qadeer, "How to shop for free online - security analysis of cashier-as-a-service based web stores," in *Oakland'11: Proceedings of the 32nd IEEE Symposium on Security and Privacy*, 2011.

[101] M. Balduzzi, C. T. Gimenez, D. Balzarotti, and E. Kirda, "Automated discovery of parameter pollution vulnerabilities in web applications." in *NDSS'11: Proceedings of the 8th Annual Network and Distributed System Security Symposium*, 2011.