Journal of Network and Computer Applications 109 (2018) 89–109

Contents lists available at ScienceDirect

Journal of Network and Computer Applications

journal homepage: www.elsevier.com/locate/jnca

DetLogic: A black-box approach for detecting logic vulnerabilities in web applications

G. Deepa *, P. Santhi Thilagam, Amit Praseed, Alwyn R. Pais

Department of Computer Science and Engineering, National Institute of Technology Karnataka, Surathkal, India

ARTICLE INFO

Keywords:
Application logic vulnerabilities
Logic attacks
Web application security
Parameter tampering
State violation
Workflow violation
Sequence violation
Authorization bypass
Authentication bypass

ABSTRACT

Web applications are subject to attacks by malicious users owing to the fact that the applications are implemented by software developers with insufficient knowledge about secure programming. The implementation flaws arising due to insecure coding practices allow attackers to exploit the application in order to perform adverse actions leading to undesirable consequences. These flaws can be categorized into injection and logic flaws. As large number of tools and solutions are available for addressing injection flaws, the focus of the attackers is shifting towards exploitation of logic flaws. The logic flaws allow attackers to compromise the application-specific functionality against the expectations of the stakeholders, and hence it is important to identify these flaws in order to avoid exploitation. Therefore, a prototype called **DetLogic** is developed for detecting different types of logic vulnerabilities such as parameter manipulation, access-control, and workflow bypass vulnerabilities in web applications. DetLogic employs black-box approach, and models the intended behavior of the application as an annotated finite state machine, which is subsequently used for deriving constraints related to input parameters, access-control, and workflows. The derived constraints are violated for simulating attack vectors to identify the vulnerabilities. DetLogic is evaluated against benchmark applications and is found to work effectively.

1. Introduction

Web applications play a pivotal role in every individual's day-to-day activities because they are widespread, and provide the platform for performing various activities such as communicating, banking, shopping, and social networking. Unfortunately, the increasing popularity, availability, ease of use, and the user base make web applications an appealing target for cybercriminals. Cybercriminals exploit vulnerabilities in the applications for monetary benefits, stealing confidential information, performing undesirable operations, and so on. According to Symantec Internet Security Threat report (Symantec, April 2016), 75% of legitimate applications had unpatched vulnerabilities, and one million attacks were reported on web applications in 2015. A report from Trustwave (2016) states that 97% of the applications tested by Trustwave had security vulnerabilities, and a median of 14 number of vulnerabilities is discovered per application.

Vulnerabilities in web applications can be majorly classified into two classes: Injection Vulnerabilities and Business Logic Vulnerabilities (Cova et al., 2007; Felmetsger et al., 2010; Li and Xue, 2013, 2014). The former allows the attackers to inject malicious data as part of a command/query/script that tamper the designated structure, while the latter allows the attackers to inject data that induce a web application software to behave differently and exhibit unforeseen behavior against the intention of the programmer. Large number of solutions are available for mitigating injection vulnerabilities, and hence the focus of the attackers has shifted towards the exploitation of logic vulnerabilities. Logic flaws are identified as the second topmost vulnerability that need the attention of the researchers (Trustwave, 2014). A report by Trustwave (2016) states that 64% of the tested applications had session management vulnerabilities, 41% had server-side vulnerabilities, and 39% of them had authentication and authorization vulnerabilities.

Business logic vulnerabilities are defects that normally permit malicious users to circumvent the expected functionality of an application. The attacks exploiting these defects are legitimate application transactions used to carry out an undesirable operation that is not part of usual business practice (Grossman, 2007). Since logic attacks are directly tied towards financial loss, the impact of logical vulnerabilities is often more severe. For instance, consider an online trading application that sets up an upper limit for a transaction for every user of the application, and controls the value through a parameter in the HTTP request. If the value

* Corresponding author.
E-mail addresses: gdeepabalu@gmail.com (G. Deepa), santhi@nitk.ac.in (P.S. Thilagam), amitpraseed@gmail.com (A. Praseed), alwyn@nitk.ac.in (A.R. Pais).

https://doi.org/10.1016/j.jnca.2018.01.008
Received 7 September 2017; Received in revised form 25 December 2017; Accepted 19 January 2018
Available online 22 February 2018

for the upper limit of the transaction is not validated properly at the server-side, then any knowledgeable user can modify the limit on their own in the request, consequently violating the anticipated behavior of the application.

Logic vulnerabilities can be classified into three major types: parameter manipulation, access-control, and workflow bypass vulnerabilities (Cova et al., 2007). These vulnerabilities arise due to the errors in management of data flowing between different web pages and the control flow of the application (Pellegrino and Balzarotti, 2014; Monshizadeh et al., 2016). Logic vulnerabilities are specific to the functionality of the web application, making them extremely challenging to identify and deal with.

State-of-the-art: The existing corpus of literature has solutions to secure web applications from logic flaws at different phases of the software development life cycle. (i) For applications under construction, secure web frameworks are proposed to take care of security aspects during the development of the application itself (Chong et al., 2007b,a; Jia et al., 2008; Swamy et al., 2008; Vikram et al., 2009; Yip et al., 2009; Corcoran et al., 2009; Swamy et al., 2010; Morgenstern and Licata, 2010; Krishnamurthy et al., 2010). (ii) For legacy applications (i.e., deployed applications), the existing works make use of white-box and black-box approaches to mitigate logic vulnerabilities/attacks. Some works concentrate on detecting logic vulnerabilities in the application, but put the burden of eliminating them on the developer (Bisht et al., 2011; Monshizadeh et al., 2014; Sun et al., 2011; Son et al., 2011; Li and Xue, 2013; Li et al., 2014). On the other hand, some research works focus on preventing logic attacks in the applications during runtime (Li and Xue, 2011; Li et al., 2012; Skrupsky et al., 2013). Vulnerability detection systems assist the developers in identifying the exploitable weaknesses so that they can be fixed, while attack prevention systems block malicious requests circumventing the anticipated behavior of the application.

Research Gaps: The approaches/solutions put forward have their own advantages and disadvantages which are described as follows. Secure frameworks cannot protect legacy applications from attacks, and secure coding practices are labor-intensive. The vulnerability detection systems that employ white-box approach demand the source-code of the application and are technology dependent. Sharing the source-code of the application to a third-party for identification of the vulnerabilities compromises the privacy of the stakeholders of the application. The problems with attack prevention systems are generation of false positives that block legitimate transactions from being executed, and instrumentation of the source code for analyzing the HTTP packets. The vulnerability scanners Acunetix, AppScan, AppScaner, WebInspect, and QualysGuard generate traditional attack vectors such as SQL injection, XSS, and Cross-site request forgery (CSRF), and do not address identification of logic vulnerabilities. Therefore, there is a demand for a system that is capable of detecting logic vulnerabilities in web applications independent of the functionality of the application. Hence, we propose a vulnerability detection system that employs black-box approach for detection of logic vulnerabilities.

Logic vulnerabilities allow the attackers to compromise the intended behavior of the application, and hence logic vulnerability identification requires extraction of the intended behavior of the application. The intended behavior is utilized for generating malicious requests that assist in detecting the vulnerabilities. The existing approaches take into account the parameters flowing to a web page, session variables associated with a web page, and the session variables used for maintaining the sequence of operations for modeling the intended behavior of the application to identify the different types of logic vulnerabilities. The existing approaches have their own limitations which are discussed as follows. (i) The black-box approaches for detecting parameter manipulation consider parameters that appear in a request for a web page and do not take into account the interaction between multiple web pages (i.e., data flow), and therefore fail to identify few vulnerabilities (Bisht et al., 2010). (ii) Literature on workflow bypass vulnerabilities identification considers session variables used for maintaining the sequence of operations, and does not take into account other parameters such as CSRF tokens (Li and Xue, 2013). The sequence of operations within the application is referred as workflow or control flow. (iii) The works that identify access-control vulnerabilities arising due to inappropriate definition and validation of session variables, are not flexible enough to identify other types of session related vulnerabilities leading to session puzzling and session fixation attacks. (iv) The works by Bisht et al. (2010), Li and Xue (2013), and Li et al. (2014) are ad-hoc approaches, and hence are unable to cover a wider range of logic vulnerabilities. Considering the pros and cons of the state-of-the-art systems, the proposed work constructs a web application model reflecting the intended behavior in terms of data flow and control flow that could be used for generating attack requests/vectors to detect all the three types of logic vulnerabilities.

Challenges: The difficulty involved in identifying logic vulnerabilities lies in extraction of intended behavior of the application without using the source code. The major challenges involved and the mechanisms employed in this paper to address the challenges are as follows:

- Understanding the business requirements imposed on user-input of the application to identify inconsistencies existing between client-side and server-side validations
  - The requirements placed on user-input are inferred by analyzing the HTML/JavaScript code available at the client-side
- Deriving the intended access-control policies of the application
  - The access-control policies related to the application are derived from the session variables defined for maintaining state of the application
- Deducing the business workflows of the application
  - The intended workflows in the application are derived from a model which is constructed out of the navigations done manually in the application

Contributions: In this work, a prototype called DetLogic is developed to identify business logic vulnerabilities existing in web applications. The prototype works in three phases: (i) extraction of the intended behavior of the web application under test, (ii) construction of concrete attack vectors based on the information gathered, and (iii) comparison of the responses obtained during normal and attack executions, and reporting vulnerabilities accordingly. The prototype detects three types of logic vulnerabilities: parameter manipulation, access-control, and workflow bypass vulnerabilities.

The major contributions towards this work are as follows:

- Development of a black-box approach for extracting the data flow and control flow of the web application to identify three different types of logic vulnerabilities.
- Construction of a model that reflects the intended behavior of the application using the extracted information.
- Derivation of constraints from the model, and generation of attack vectors that violate the derived constraints for identification of different types of logic vulnerabilities.
- Identification of vulnerabilities leading to session puzzling attacks. To the best of our knowledge, this is the first work to identify session puzzling vulnerabilities.
- Implementation of a prototype based on the proposed approach, and evaluation of effectiveness of the prototype using benchmark web applications.

The remainder of the paper is organized as follows. Section 2 provides an overview of the different types of application logic vulner-

abilities and their root causes. Section 3 illustrates the problem with an example. Section 4 presents an overview of the proposed approach and the proposed web application model. Sections 5 and 6 elaborate the design and implementation of the proposed approach, respectively. Section 7 discusses the evaluation results. Section 8 presents the related work and the paper is concluded in the last section.

2. Logic vulnerabilities

Logic vulnerabilities allow malicious users to inject data that induce a web application software to behave differently and exhibit unforeseen behavior against the intention of the programmer resulting in attacks.

2.1. Types of logic vulnerabilities

The three major types of logic vulnerabilities are described as follows.

Parameter manipulation: These vulnerabilities allow modification of values of critical variables, and favor the attackers to cause serious damage to the application by bypassing the client-side validation. For example, in an eCommerce application, failing to validate the price of an item at the server-side while placing an order allows an attacker to modify the price of the item to a negative value, and therefore allows the attacker to purchase the item for free.

Access-control vulnerabilities: These vulnerabilities allow attackers to acquire access to a restricted resource, which is exclusively intended for a highly privileged user of the application. According to OWASP (Top10, 2013) risks and SANS errors (SANS, 2011), four out of the ten risks,[1] and five out of the top 25 dangerous errors[2] are related to incorrect implementation of access-control checks.

Workflow bypass: These vulnerabilities allow the attackers to disturb the intended workflow of the application, consequently breaking the business-specific functionality of the application. Intended workflow is the sequence of steps to be followed for completing a certain task in the application. For example, an online banking application requires the user to select a beneficiary for transferring funds, enter the amount to be transferred, and then confirm the transfer. If the “transfer amount” web page does not check whether the user has selected a beneficiary by visiting the “select beneficiary” web page only, then there is a possibility that the “select beneficiary” page can be passed over by the user by just tampering the account number in the request to “transfer amount” page, resulting in a workflow bypass attack (Cova et al., 2007; Skrupsky et al., 2013; Li and Xue, 2011). This allows the attacker to transfer funds to their own account.

2.2. Root causes

Logic vulnerabilities originate due to the following implementation flaws: (i) missing server-side validation, (ii) missing and incomplete access checks, (iii) overloading of session variables, and (iv) missing sequence checks, which are discussed in detail below.

Missing server-side validation: A web application uses client-side scripting to process and validate the user-supplied input for quick processing and for bringing down the server-side loads. However, malicious users can circumvent the client-side validation either by disabling the JavaScript execution or by submitting malicious requests which tamper the parameters in such a way that the restrictions placed on the user-input at the client-side are violated. If the application has enforced the same set of restrictions at the server-side, then the tampered parameters would be rejected by the application. Otherwise, the parameters cause the application to behave in a fashion different from the requirement specification of the application. Thus, failing to enforce user-input validation at the server-side results in *parameter manipulation attacks* (Bisht et al., 2010, 2011; Skrupsky et al., 2013).

Missing/Incomplete access check: Web applications use HTTP, a stateless protocol, that treats each web request and response in an independent fashion. Hence, to maintain the state, applications use sessions to denote the logged in status of a user, the privilege level (say, admin or normal user), the sequence of operations, and a number of other things. Sessions can be maintained in two ways - purely at the client-side by means of cookies, or a combination of server and client-side. To maintain sessions, it is expected that the application performs some checks on the session variables before permitting access to its privileged resources. For example, a basic access check in an application verifies whether the session variables are set or not for allowing users to access the home page of the application only after logging in. However, some pages may not perform these checks, and consequently a malicious user can gain access to these pages. Thus, missing access checks lead to *authentication/authorization bypass attacks* (Sun et al., 2011; Son et al., 2011, 2013; Li and Xue, 2011).

Similarly, for a user to access the application, the session variables should be properly validated against the role of a user and the HTTP parameters. There may exist scenarios in which the application verifies whether the session variables or the HTTP parameters are set or not, but do not validate the values of the session variables against the role of the user and the HTTP parameters. The improper validation of session variables against the role and the HTTP parameters is termed as incomplete/improper access checks that lead to *vertical/horizontal privilege escalation attacks* (Li and Xue, 2011; Monshizadeh et al., 2014).

Overloading session variables: Uncontrolled creation/population of session objects or usage of identical session variables at various application entry points is called overloading of session variables, and may lead to *session puzzling attacks* (Chen, 2011). These attacks do not contain any malicious input. They are legal actions allowed by the web application, but when performed in a particular order compromise the intended functionality of the application. While exploiting session puzzles, the creation of session objects can be indirectly initiated, and later exploited by accessing a sequence of entry points (web pages, web services, remote procedure calls, etc.) in a certain order. Session puzzles enable adversaries to perform a variety of malicious actions such as bypassing authentication/authorization and elevation of privilege for users, and upset the normal execution of the application.

Missing sequence check: Web applications use sessions to maintain the sequence of operations within the application (Li and Xue, 2011). In addition to session variables, CSRF tokens generated at sensitive/critical pages of the application for preventing CSRF attacks indirectly assist in maintaining the sequence of operations (Jovanovic et al., 2006). CSRF tokens are stored either in session or as a cookie, and are validated against the HTTP parameters to prevent attacks (CSRF). A CSRF token is generated at the server-side of the application in a critical page, and is either stored in the session or issued to the client and set as a cookie. The CSRF token stored in the cookie/session is supposed to be validated against the CSRF parameter in the HTTP request of the web page that follows the critical page. Thus, CSRF token enables to maintain the sequence of operations within the application. If a web page that follows the critical page fails to validate the value of the CSRF token stored in the session/cookie against the

[1] (1) Broken Authentication and session management, (2) Insecure Direct Object References, (3) Missing Function Level Access-Control, and (4) Unvalidated Redirects and Forwards.

[2] CWE-306, CWE-862, CWE-250, CWE-863, and CWE-732.

HTTP parameter, then there is a possibility that the critical page can be bypassed by the attacker by directly placing request for the pages that follow the critical page. Thus, lack of validation of tokens issued for maintaining the sequence at the server-side results in *workflow bypass attacks*.

3. Problem illustration

This section describes the problem with a running example which details the various types of vulnerabilities that lead to a variety of logic attacks on web applications.

3.1. Motivating example

Listings 1 to 8 illustrate a sample web application intended for maintaining medical records of patients registered with the application. The application has six web pages: *Login, Forgot Password,* and *Index* pages which are common to all the users of the application, and *View, Create,* and *Delete* pages intended for users with specific roles. Three user roles exist in the application: *Patient, Physician,* and *Administrator*. The business requirements of the application specify the following rules: (i) a patient can view their own record, (ii) a physician can view records of their own patients, and can create new medical records, and (iii) administrator can view and delete record of any patient registered within the application, and can create new records.

The user is first presented with the *Login.php* page. After the user has provided valid credentials, the application redirects the user to *Index.php* page. The *Index.php* page is presented with three hyperlinks to *View.php, Create.php,* and *Delete.php* depending on the role of the logged-in user retrieved from the database. *View.php* is available for all the three roles, *Create.php* is meant for both *Physician* and *Administrator*, and *Delete.php* is intended for *Administrator* only. Listing 3 is a JavaScript file that validates the input supplied by the user at *Create.php* page.

The web application maintains two session variables – *role* and *userid*. The variable *role* keeps track of the privilege of a user within the application, and the variable *userid* keeps track of individual users by means of their user ID. The presented application inherits several vulnerabilities which are discussed in detail.

```
<?php
  if(isset($_GET["userid"])) {
    $user = $_GET["userid"];
    //Code to create hyperlink to View Record
    page
    .....
    if($_SESSION["userid"]==$user && ($_SESSION["
    role"]=="physician" || $_SESSION["role"]=="
    admin")) {
      //Code to create hyperlink for Create
    Record page
      .....
    }
    if($_SESSION["role"]=="admin" ) {
      //Code to create hyperlink to Delete Record
     page
      .....
    }
  } else {
    throw new Exception("Please login to the
    application");
  }
?>
```

Listing 1. Index.php.

Bug 1: *Index.php* page does not verify whether the session variable *userid* is set or not. As a result, any malicious user can access *Index.php* page without logging into the application by providing a valid value for the parameter *userid* in the HTTP request, and hence, the user would be provided with hyperlink to view the medical record. Thus, the exploitation of this missing access check results in authentication bypass attack.

```
<?php
  if(isset($_GET["userid"])) {
    $user = $_GET["userid"];
    if(isset($_SESSION["userid"]) {
      if($_SESSION["role"]=="patient") {
      //Code to Retrieve and Display Record
      .....
      } else if($_SESSION["role"]=="physician" ||
     $_SESSION["role"]=="admin") {
        if(isset($_POST["patientid"])) {
          $patientid = $_POST["patientid"];
          //Code to Retrive and Display Record
          .....
        } else {
?>
          //HTML Code to get the Patient ID
          .....
<?      }
      }
    }
  } else {
    die("Please login to the application");
  }
?>
```

Listing 2. View.php.

Bug 2: *View.php* page checks if $_*SESSION*["*userid*"] is set or not, but do not validate the value of the parameter *userid* in the request with the session variable. As a result, a user can view the medical record of another user by modifying the value of *userid* in the request after logging into the application. This improper access check in the page would result in horizontal privilege escalation attack. The page should have checked if the *userid* passed is the same as the session variable *userid* which is set.

```
function validateInput() {
  var age = document.getElementById("age").value;
  if(document.getElementById("fname").value=="")
    {
    alert("User name can not be blank");
    return false;
  } else if (age < 0 || age > 150) {
    alert("Age must be greater than zero and less
     than 150.");
    return false;
  }
  return true;
}
```

Listing 3. Validate.js.

```
<script type='text/javascript' src='Validate.js'>
</script>
<?php
  if(isset($_GET["userid"])) {
    $user = $_GET["userid"];
    if($_SESSION["userid"]==$user) {
      if($_POST["mode"]=="insert") {
        //Code to Insert Record
        .....
      } else {
?>
        //HTML Code to get inputs for creating record for a user
        .....
<?    }
    }
  } else {
    die("Please login to the application");
  }
?>
```

Listing 4. Create.php.

Bug 3: *Create.php* page checks whether the *userid* set in the session and passed in the request match, but fails to verify the role of the user having access to the web page. As a result, a user with lower privilege can create medical records by issuing a direct request to *Create.php* page. This incomplete access check in the web page would result in vertical privilege escalation/authorization bypass attack. This could be eliminated by checking whether the value of the $_*SESSION*["*role*"] variable is equal to *admin* or *physician*.

Bug 4: The web page *Create.php* gets input from the physician or admin to create medical records. There exist a field *Age* in the form which confines its value between 0 and 150 using client-side scripting presented in Listing 3. A malicious user can bypass the client-side validation and submit a request for *Create.php* page, with an invalid value, say, −5 for the parameter *Age*. The parameter *Age* is not validated at the server-side before insertion of the value into the database, and hence the intended behavior of the application is violated. Thus, missing server-side validation is exploited for launching a parameter manipulation attack.

```
<?php
  if(isset($_GET["userid"])) {
    $user = $_GET["userid"];
    if($_SESSION["userid"]==$user && $_SESSION["role"]=="admin") {
      if(isset($_POST["patientid"])) {
        HTTPRedirect("Confirm.php");
      } else {
        if(!isset($_COOKIE["csrf_token"])) {
          $token = bin2hex(random_bytes(32));
          setcookie("csrf_token",$token);
        }
?>
        //HTML Code to get Patient ID
        .....
<?    }
    }
  } else {
    die("Please login to the application");
  }
?>
```

Listing 5. Delete.php.

```
<?php
  if(isset($_GET["userid"])) {
    $user = $_GET["userid"];
    if($_POST["confirm"]=="yes") {
      if(isset($_POST["token"])) {
        //Code to Delete Record
        .....
      }
    } else {
?>
        //HTML Code to get Confirmation from user
        .....
<?    }
  } else {
    die("Please login to the application");
  }
?>
```

Listing 6. Confirm.php.

Bug 5: *Confirm.php* page gets the *userid* from the request, but fails to verify the role of the user and the value of the session variables. As a consequence, any user with lower privilege can delete existing medical records by issuing a direct request to *Confirm.php* page. This page misses the access check, and hence would result in vertical privilege escalation/authorization bypass attack.

Bug 6: The web page *Confirm.php* validates whether the CSRF token is present in the HTTP request or not, but fails to verify the value of the CSRF token set in the cookie against the HTTP parameter *token*. Even though CSRF token is used for preventing CSRF attacks, it indirectly helps in maintaining the sequence of operations within the application. As a consequence, any user can delete existing medical records by issuing a direct request to *Confirm.php* page by just appending the parameter *token* in the request. Thus, this web page misses the CSRF token validation, and hence would result in a workflow bypass attack.

```
<?php
  if(isset($_POST["userid"])) {
    $_SESSION["userid"] = $_POST["userid"];
    header("Location:PwdRecovery.php?userid=".$_SESSION["userid"]);
  } else {
?>
    <form action="ForgotPwd.php" id="Pwd" method="post">
      UserID: <input type="text" name="userid"><br>
      <input type="submit" value="Next">
    </form>
<? } ?>
```

Listing 7. ForgotPwd.php.

Bug 7: The application defines a session variable *userid* when a user logs in. Some pages in the application check whether the session variable *userid* is set and is equal to the *userid* parameter in the request. Now, there is a password recover facility in the web application. If a user has forgotten their password, the web application prompts them for their userid. Once the userid is entered, the application proceeds to display the subsequent pages, and stores the value of the userid in the session variable *userid*, which is the same as the session variable stored after logging in. Setting up the session variable in the *ForgotPwd.php* page is a simple mistake, but can have unforeseen consequences. For instance, after entering the userid, the session variable is set. Hence, a malicious user can instantly browse all the pages in the web application which checks only for the session variable *userid*. This is an example for

session variable overloading. This error can easily be avoided if session variable names are not duplicated.

```
1  <?php
2    if(isset($_POST["userid"]) && isset($_POST["
       password"])) {
3      $login = validateUser($_POST["userid"],
       $_POST["password"]);
4      if(!$login) {
5        HTTPRedirect("Login.php");
6      }
7      $_SESSION["userid"] = $_POST["userid"];
8      //Code to retrieve role of the user from the
       database
9      .....
10     $_SESSION["role"] = $role;
11     header("Location:index.php?userid=".$_SESSION
       ["userid"]);
12   } else {
13 ?>
14     //HTML form to get login credentials from a
       user
15     .....
16     .....
17 <? } ?>
```

Listing 8. Login.php.

Bug 8: *Login.php* gets input from the user, validates the input and redirects the user to *Index.php*. In case the user-input supplied is not valid, then line 5 of Listing 8 redirects the user to the same page i.e., *Login.php*, but the server executes the rest of the code (from line 7) as it is not embedded within the else clause. In other words, even after supplying invalid credentials, the web page sets the session variables because of which all the web pages in the application could be accessed. This kind of vulnerability is called Execution After Redirection (EAR) vulnerability (Doupé et al., 2011; Payet et al., 2013; Li and Xue, 2013). This error can be avoided by properly embedding the source code in appropriate blocks.

4. Proposed approach

Given a web application, the objective of this work is to develop a systematic approach for modeling the intended behavior of the application to detect business logic vulnerabilities prevailing in the application, independent of the functionality of the application. The formulated web application model should reflect the intended behavior in terms of data flow and control flow which are subsequently utilized for generation of attack vectors to identify the vulnerabilities. In the context of this work, the data flow refers to the parameters and session variables flowing between different web pages, and control flow refers to the sequence of operations between web pages of the application.

A prototype is implemented based on the proposed approach, and it constructs a web application behavioral model for identification of the logic vulnerabilities. This section describes the architecture of the prototype constructed for identifying the vulnerabilities, and the conceptual model proposed for reflecting the intended behavior of the application.

4.1. Overview

A prototype called DetLogic is developed as part of this work for identification of three different types of business logic vulnerabilities, namely, parameter manipulation, access-control, and workflow bypass vulnerabilities in web applications using a black-box approach. The prototype is a penetration testing tool that is employed between the client and server of the web application under test as shown in Fig. 1.

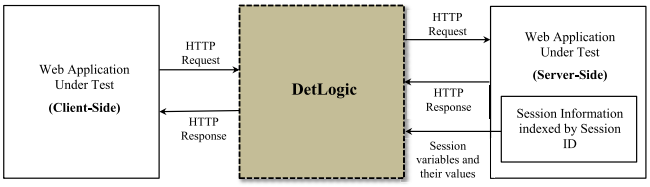


Fig. 1. DetLogic overview.

DetLogic acts a proxy intercepting the HTTP requests and responses to the application under test during learning phase, and places attack requests to the web server of the application under test during discovery of vulnerabilities. The working principle of DetLogic is explained in Section 5.

4.2. Web application model

The behavior of the web application is modeled conceptually as a finite state machine which is described in detail in this section.

A Finite State Machine (FSM) is utilized for modeling the behavior of the application, as FSM is well suited for modeling the behavior of any system. The information needed for constructing the FSM are extracted from execution traces collected using a proxy server. The proxy intercepts the HTTP requests to the web application and provides details regarding the parameters and session variables required for constructing the FSM.

4.2.1. FSM construction

To identify the three types of logic vulnerabilities, the web application is modeled as an annotated FSM (W) with hextuple (Q, Σ, A, δ, q_0, F), where

- Q denotes finite set of states, and the states are represented as DOM structure of the web pages with unique URL in the application.
- Σ indicates finite set of inputs. The input symbol which marks the transition to the next state in the FSM is represented as either the URL of the hyperlink clicked or the URL of the redirected web page of the application.
- A is a set of annotations, which are used for providing additional conditions that assist the transition from current state to the next state. The annotation is represented as a three tuple given by [R, P, S], where
  - R represents the set of roles (i.e., privilege level of users) having access to the page,
  - P represents the set of HTTP parameters and their respective values that get passed along the request, and
  - S represents the set of session variables and their corresponding values
- δ is the transition function, which is defined as a mapping from $Q \times \Sigma \times A$ into Q. For example, $\delta(q_s, i, a) = q_d$ indicates that if the current state is q_s with input symbol i ($i \in \Sigma$) under the annotation a ($a \in A$), then there will be a transition from the current state q_s to the next state q_d. We call this FSM as an annotated FSM.
- q_0 marks the initial state, where $q_0 \in Q$. Initial state is the home page of the application from where the navigation starts.
- F represents the set of final states, where $F \subseteq Q$. Final states refer to the pages that mark the end of navigation within the application (i.e., usually the *logout* page).

The DOM structure of the web page with unique URL in the application is represented as a state in the annotated FSM. The edges are labeled using the URL, and the HTTP parameters and session variables that are input/set by the users and application. The information regarding construction of the state machine is explained in detail in

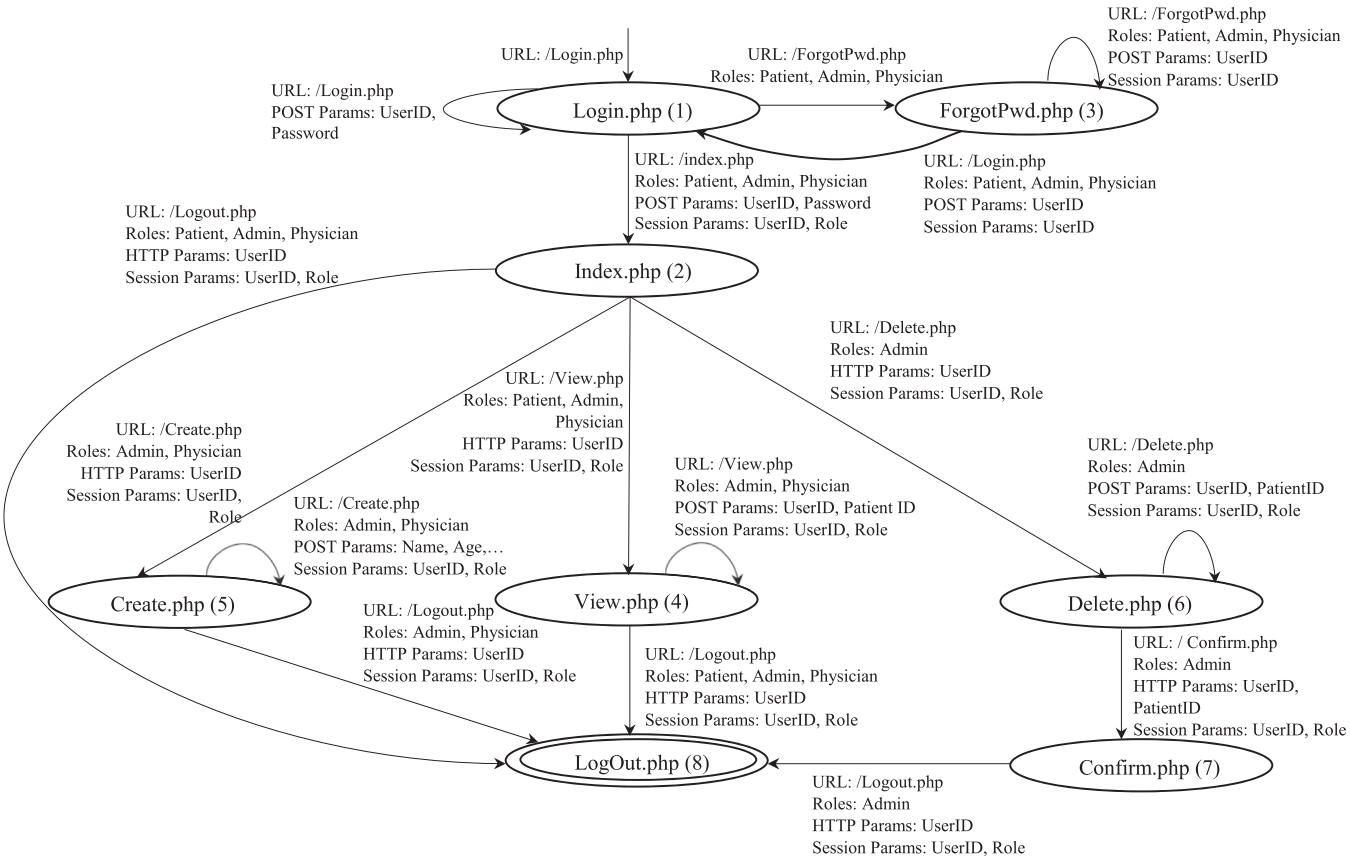


Fig. 2. Web application model.

Section 6.2.1. The URL on the labeled edge is treated as the input symbol, and the parameters and session variables are treated as annotation functions. The input symbols and the annotation functions at a particular web page decide the transition to the subsequent web pages. The parameters labeled as annotations on the edges of the FSM represent the data flow, the session variables and their values are used for inferring access-control policies, and the transitions from one state to another state represent the control flow of the application. The data flow and control flow are used for identifying parameter manipulation and workflow bypass vulnerabilities respectively, and the session variables are used for identifying access-control vulnerabilities.

In an ideal scenario, if all the web pages in the application are browsed by the tester for all the roles and for all the possible workflows, then the motivating example discussed in Section 3 can be modeled as shown in Fig. 2. The annotated FSM has eight states, and the input symbols represented on the edges in the state diagram mark the transitions within the application. *Login.php* and *Logout.php* mark the initial and final states of the application, respectively. The model reflects the intended business behavior of the application. *Login, Index, View, ForgotPwd,* and *Logout* pages are accessible by all users, while *Create*, and *Delete* pages are intended for admin, physician, and admin users, respectively. As seen in Fig. 2, in web page *Index.php*, if the user clicks on the hyperlink for creating new medical records, then a transition to the next state, *Create.php*, occurs if the session variables *userid* and *role* are set, and the parameter *userid* exists in the HTTP request. The input symbol *role* marked on the edge represents the role for which *Create.php* is available to the users.

4.2.2. Inference of constraints

The generated web application model (Annotated FSM) is analyzed for inferring constraints related to parameters (i.e., data), access-control (i.e., role), and control flow of the application, which are subsequently used to generate concrete attack vectors for identifying the three types of logic vulnerabilities. The details regarding extraction of constraints are described in detail in the following subsections.

4.2.2.1. Access-control constraints. The constructed state machine is used for generalizing a set of access-control constraints which are used for deriving attack vectors. Three types of access-control constraints are inferred from the session variables and HTTP parameters and their respective values at each state which are explained as follows. Firstly, the basic constraint inferred at each state from the constructed model is the null check on session variables and parameters that have caused transition to that state. The second type of constraint inferred is an equality constraint derived from the session variable and HTTP parameter whose values are identical. The third constraint derived is based on the role of the user having access to the state. Some of the constraints derived from Fig. 2 are as follows:

The constraints inferred at *Create.php* are:

(i) $\$\_SESSION[userid] \neq null$,
(ii) $\$\_POST[userid] \neq null$,
(iii) $\$\_SESSION[userid] == \$\_POST[userid]$, and
(iv) $\$\_SESSION[role] == Admin \parallel Physician$

Similarly, *Delete.php* infers constraints such as

(i) $\$\_SESSION[userid] \neq null$,

(ii) $\$\_POST[userid] \neq null$,
(iii) $\$\_SESSION[userid] == \$\_POST[userid]$, and
(iv) $\$\_SESSION[role] == Admin$

The session variables and the parameters that check for null values (Constraints (i) & (ii)) help in identifying missing access check vulnerabilities. Constraint (iii) that checks the value of the session variable against the value of the HTTP parameters is useful in identifying horizontal privilege escalation vulnerabilities. The session variables that check their values against the roles (Constraint (iv)) registered with the application aid in the identification of authorization bypass vulnerabilities. Attack vectors are generated by violating each of the extracted constraints to identify vulnerabilities. To check if Constraint (i) is implemented, the page *Create.php* is forcefully browsed with null value for the session variable *userid*. For Constraint (iii), the value of the parameter *userid* in the HTTP request is modified to a value different from the session variable *userid*. In the case of Constraint (iv), requests for web pages *Create.php*, and *Delete.php* are submitted with users having lower privileges (i.e., patient). In case the response obtained for the attack vectors being the same as that of normal requests, then vulnerability would be reported for these web pages.

4.2.2.2. Parameter constraints. The restrictions placed on the user-input at the client-side are stored as parameter-related constraints (Constraint (ii)). The client-side code is analyzed using an analyzer discussed in Section 6.2.2 to extract these constraints which are mapped onto the model. The JavaScript analyzer provides three different types of constraints on the parameters flowing between web pages: (i) Data type constraint, (ii) Value constraint, and (iii) Length constraint. Data type constraint states the type of value that should be provided as input by the user. Value constraint refers to the value that an input field can hold. Length constraint refers to the number of characters that an input can hold. For instance, for the parameter *Age* in *Create.php*, the value constraint inferred is $0 < age < 150$. Similarly, for parameter *name*, the length constraint inferred is $Length(fname) > 0$. The inferred constraints are violated during attack generation for detection of parameter manipulation vulnerabilities. In case the response obtained for the attack request is similar to that of the normal request, then the web page is missing validation for the parameters at the server-side.

4.2.2.3. Workflow constraints. With respect to detection of workflow bypass vulnerabilities, the sensitive pages in the application are to be identified to verify whether they can be bypassed. In the context of this work, the sensitive/critical pages are defined as the web pages which are developed with the intention that they should not be skipped while following a workflow. The identified sensitive/critical pages are marked as workflow constraints. As already described in Section 2.2, sequence of operations within the application is either maintained using sessions or CSRF tokens. Identifying critical pages in the application assists in unveiling pages that fail to maintain the sequence due to lack of validation of either session variables or CSRF tokens. While lack of validation of session variables against the HTTP parameter could be identified using access-control constraints as well, pages that lack validation of CSRF token stored in the cookie could be identified only from the discovered critical pages in the application.

The critical web pages can be identified by modeling the web application as a directed graph. To achieve this, an underlying directed graph G is constructed from the annotated FSM. The states of the FSM (i.e., set of all web pages) represent the vertex set of G, and the edge set of G is obtained by joining two vertices (i.e., web pages), say u and v, by an arc if there exists a workflow from the vertex u to vertex v. The self-loops in the directed graph are ignored, and the digraph so obtained is referred to as a control flow/workflow graph. The workflow graph will be either strongly or weakly connected. Perhaps, in most of the situations, the workflow graph is weakly connected. The critical pages in the workflow graph are identified by obtaining cut-vertices of G. A cut-vertex of G is a vertex in G whose removal either increases the number of components of G or disconnects G. In the context of this work, a cut-vertex refers to a web page which when skipped disconnects/disturbs the workflow of the application. In other words, a cut-vertex of the workflow graph corresponds to the web page developed with the intention that it should not be skipped while following the workflow. Here, the set of cut-vertices in the workflow graph is obtained by performing a depth-first search on the graph. Additionally, if removal of a vertex does not disconnect the directed graph, but creates unvisited vertices from the root vertex during the depth-first search, then it is also considered as a critical page. Thus, the web pages corresponding to the cut-vertices and the aforementioned condition are stored as workflow constraints/critical pages. The workflows involving these critical web pages are considered for identifying the workflow bypass vulnerabilities. The request for the critical web page is skipped in a workflow to detect whether the critical page can be bypassed.

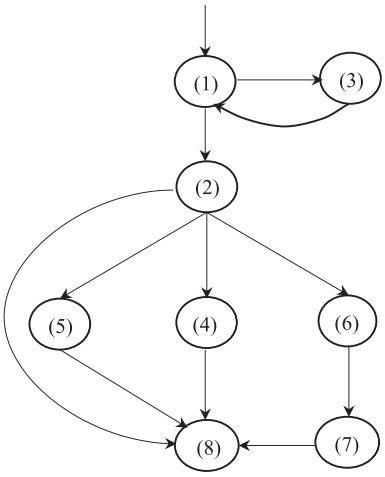


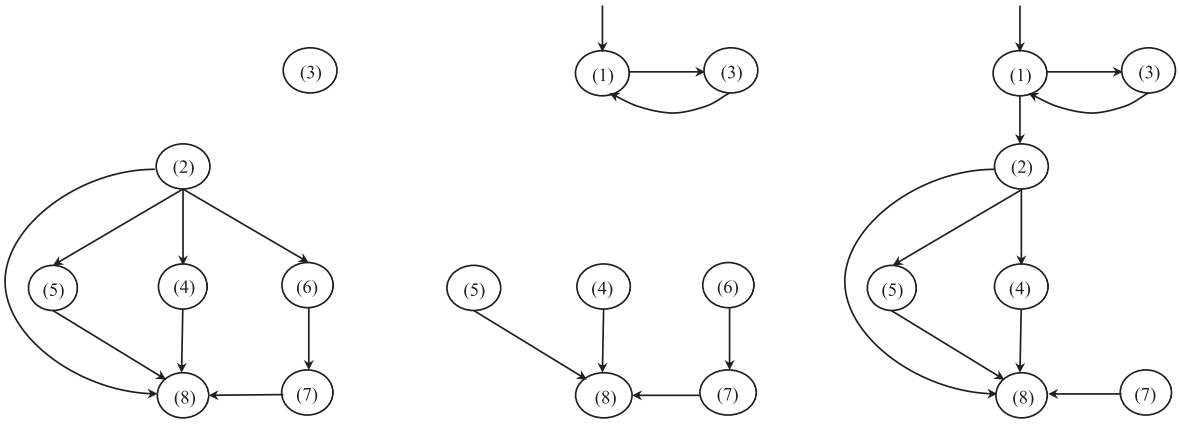
Fig. 3. Control flow graph.

Fig. 3 represents the control flow graph obtained from the annotated FSM shown in Fig. 2. The cut-vertices from the graph are nodes (1) and (2). Additionally, removal of Vertex (6) makes Vertex (7) unvisited from Vertex (1) during the depth-first search. Thus, the critical pages inferred from the application are *Login* (1) , *Index* (2) , and *Delete.php*(6) pages. Fig. 4a, b and c represent the graphs obtained after removal of critical pages.

Thus, the annotated FSM is effectively used for extraction of parameter constraints, access-control constraints and workflow constraints, which are subsequently violated for identification of parameter manipulation, access-control, and workflow bypass vulnerabilities respectively. Table 1 describes the defects existing in the application discussed in Section 3.1 and the inferred constraints.

Effectiveness of the model: The effectiveness of the proposed model is illustrated as follows:

- The annotations on the edges of the FSM offer the flexibility to identify the three different types of logic vulnerabilities.
- The session variables marked on the edges of the annotated FSM make the model capable and adaptable for identifying new types of vulnerabilities related to session management as compared to LogicScope (Li and Xue, 2013). Vulnerabilities such as improper handling of session identifiers (e.g., pages that issue session identifiers to users before logging in, but fail to modify them after logging in) can be identified by mapping the session identifier onto the model and triggering appropriate attack vec-



(a) Vertex 1 removed (b) Vertex 2 removed (c) Vertex 6 removed

Fig. 4. Graphs after removal of critical pages.

Table 1
Bugs and constraints.

Listing	Bug	Constraint	Location
Listing 1	Missing access check	$\$\_SESSION[userid] \neq null$	Line 4
Listing 1	Incomplete access check	$\$\_SESSION[\text{“}userid\text{”}] == \$\_POST[\text{“}userid\text{”}]$	Line 10
Listing 2	Improper access check	$\$\_SESSION[\text{“}userid\text{”}] == \$\_POST[\text{“}userid\text{”}]$	Line 4
Listing 4	Incomplete access check	$\$\_SESSION\,[\text{“}role\text{”}] == \text{“}physician\text{”} \parallel \text{“admin”}$	Line 6
Listing 6	Missing access check	$\$\_SESSION[\text{“}role\text{”}] == \text{“}admin\text{”}$, $\$\_SESSION[\text{“}userid\text{”}] == \$\_POST[\text{“userid”}]$	Line 4
Listing 7	Session variable overloading		Line 3
Listing 4	Missing server-side validation	$Age > 0$ && $Age < 150$	Line 9
Listing 6	Missing sequence check	Delete.php (Critical Page)	Line 5

tors.

- The annotated FSM model is capable of identifying workflow bypass vulnerabilities arising due to missing CSRF token validation at the server-side which is not addressed in BLOCK (Li and Xue, 2011) and LogicScope.
- The model offers flexibility in identifying the severity of vulnerabilities existing in the application. The control flow graph can be used to mark the nodes that are more critical by calculating the out-degree for the nodes. The identified critical nodes can be ranked based on their out-degree which could be subsequently used for marking the criticality of the web pages.
- Even though the number of states of the annotated FSM is more as compared to LogicScope, we tend to minimize the number of states by representing the states of the FSM using the web page template (i.e., DOM structure) after ignoring the data content in the web page. For any large application such as an eCommerce application, the size of the FSM will not increase with the number of web pages displaying millions of products being sold. All these web pages used for displaying the products will be treated as a single page while framing the state of the FSM due to similarity in their DOM structure.

5. Proposed system architecture

The prototype DetLogic operates in three phases: (i) learning phase, (ii) attack generation phase, and (iii) discovery phase. Learning phase is meant for extraction of the intended behavior of the application. It extracts constraints imposed on parameters and constructs a model reflecting the intended workflow of the application. Attack generation phase is meant for generating concrete attack vectors that help in identifying the vulnerabilities in the application. In the discovery phase, the prototype attempts to detect vulnerabilities in the web application by comparing the responses obtained during learning and attack generation phases. Fig. 5 shows the high-level system design of DetLogic. The working principle of DetLogic is elaborated below.

Input: Seed URL and valid user credentials are given as input to identify the points of injection for parameter manipulation. Manual traces are generated by the tester for identifying access-control and workflow bypass vulnerabilities.

Output: The output is a vulnerability report that lists the logic vulnerabilities, the types of attacks which are possible by exploiting the vulnerabilities, and the location of the vulnerabilities existing in the web application under test.

Trace Collection: Traces are the HTTP requests generated during navigation of the application under attack-free sessions. These traces can be generated either manually or automatically. The traces are analyzed for extraction of the intended behavior of the application. This work generates traces both manually and automatically to infer the specification of the application. Manual traces are used for inferring the control flow of the application. The HTTP requests generated manually are intercepted using a proxy to extract session-related information which is useful for identifying access-control vulnerabilities. Automated traces are generated using a crawler that explores all the web pages in the application on a depth-first search basis.

Learning: The learning phase is responsible for inferring the intended behavior of the web application. An HTML/JavaScript analyzer is employed for deriving the constraints on the parameters, and a model generator is utilized for extracting the intended behavior of the application in the form of an annotated FSM. The HTTP responses generated from the HTTP requests submitted to the application server during trace collection are collected, stored, and fed to the JavaScript analyzer and model generator. The analyzer and the model generator process the requests and responses, and represent the restrictions on parameters and navigation within the application in the form of parameter constraints and an annotated FSM respectively. The annotated FSM provides the data flow and control flow of the application.

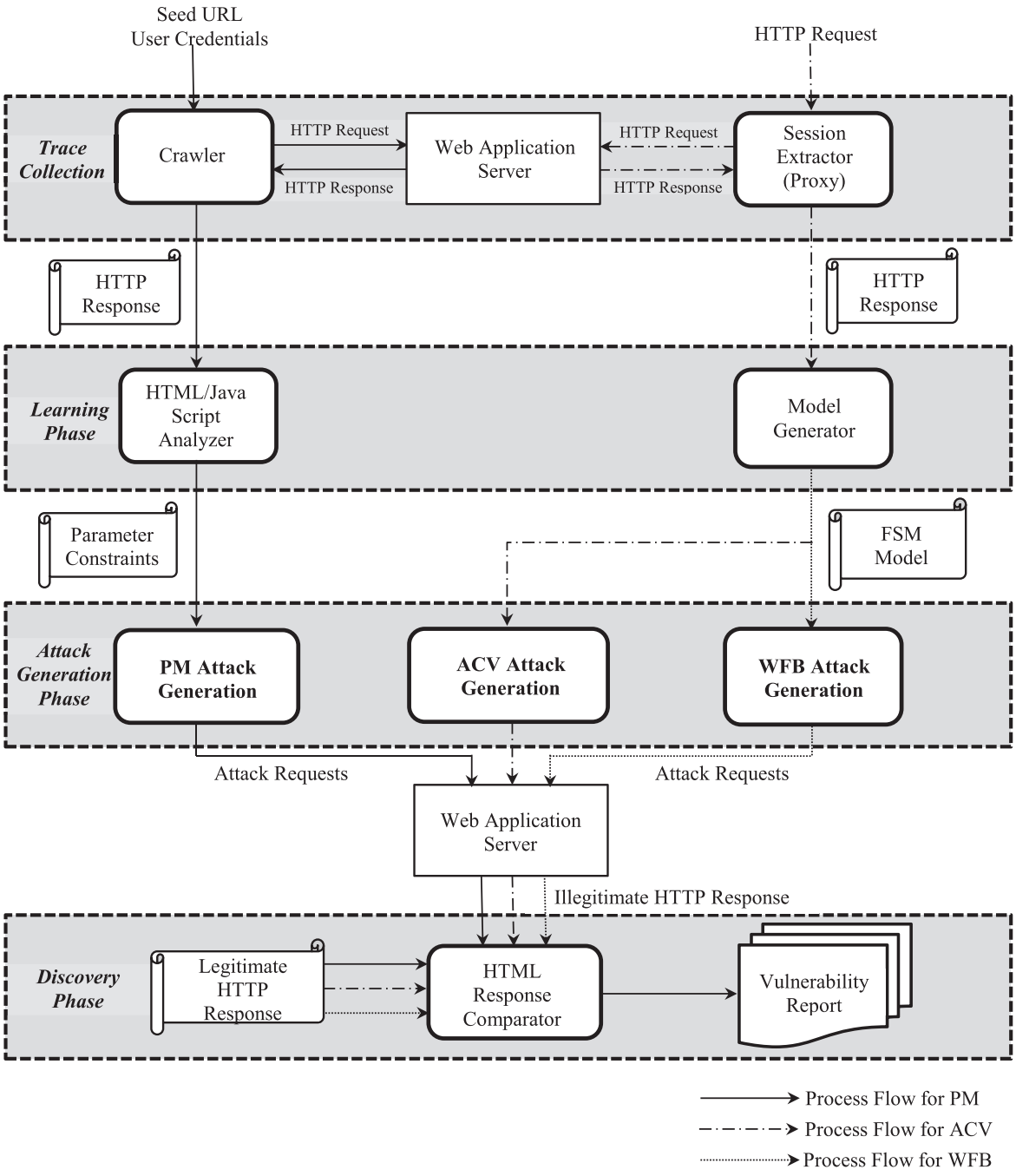


Fig. 5. DetLogic architecture.

Attack Generation: Three different types of attacks parameter manipulation, access-control violation and workflow bypass attacks are generated for identifying business logic vulnerabilities. Attack vectors are generated on injection points by either modifying values of parameters in the HTTP request in such a way that parameter constraints are violated, or appending parameters which are not supposed to be present in the request. For identifying access-control violation attack, access-control constraints are used to identify whether highly privileged pages can be accessed by users with low privileges. To identify workflow bypass attack, the critical web pages in the web application are identified from the control flow, and are skipped from being visited.

Discovery: HTML response comparator is developed for comparing the response generated after each attack request against the corresponding response stored in the database during learning. Vulnerability is reported when the response generated during an attack is similar to that of the response obtained for a valid request. A report is finally generated with details of vulnerabilities, the types of attacks that exploited the vulnerabilities, and the flaws which led to the attacks.

6. Implementation

This section elaborates the implementation details of the prototype.

6.1. Trace collection

Traces are essential for exploring the web application under test. Traces are generated in an automated fashion using a crawler as well as manually by a tester. Manual trace generation is preferred to automated crawling in the case of detection of access-control and workflow bypass vulnerabilities due to the following reasons:

(i) There is a possibility of inferring false behavior of the application as intended behavior by the crawler. For instance, consider a web page in an application which is vulnerable to authorization bypass attack. In the case of using an automated crawler for exploration of the application during learning, the crawler is able to visit the vulnerable page as a normal user itself, and hence would result in inference of a false condition stating the page can be visited by a normal user rather than a highly privileged user. As a result of this false learning, the vulnerability in the application would be missed from being identified. Thus, if the application is vulnerable, then automated crawling would result in inference of false behavior resulting in false negatives.

(ii) There is a possibility of missing few vulnerable pages from being identified by the crawler as the crawler may not navigate through the application in a fashion intended by the programmer. For example, consider an eCommerce website which has the following web pages namely *Purchase, Add New Item, Edit Item, Remove Item*, followed by *Confirm Order,* and *Payment* pages. Assuming *Purchase* page has hyperlinks to *Add New Item, Edit Item,* and *Remove Item* pages, if the crawler visits *Edit Item* or *Remove Item* web page with no items in the cart, then there is a possibility that the subsequent pages *Confirm Order* and *Payment* web pages might not be processed completely. As these web pages are visited during *Edit Item* or *Remove Item* workflows, even when a new item is added through the *Add New Item* page, the workflow will not be completed as the crawler marks *Confirm Order* and *Payment* pages as already visited. This may result in false inference of behavior of the application.

(iii) For identifying workflow bypass vulnerabilities, the major challenge involved is the extraction of all valid workflows in the web application. This is because, the web applications today change the state not merely due to click action on hyperlinks, but also due to events that are triggered on elements embedded within the web pages. It is difficult to crawl such applications as a crawler cannot initiate a JavaScript event. Therefore, this work makes use of traces generated manually.

Manual traces are generated by allowing a tester to navigate through the application in a browser configured to use a proxy server. The proxy intercepts the HTTP requests/responses which are stored for further analysis during construction of the model and discovery of vulnerabilities. In order to achieve better coverage, the application is explored for each role of the application. In addition to collection of the web traffic requests, session variables which are essential for maintaining the state of the application are extracted. These variables assist in the detection of access-control vulnerabilities. In order to achieve this, the proxy is configured with an extension module that is responsible for extraction of session information, and the same is discussed in Section 6.1.2.

6.1.1. Crawler

The crawler component developed as part of this work starts crawling the application starting from the seed URL and fetches the HTTP response. The crawler looks for hyperlinks in the HTML response of the web pages. It looks for the attributes such as *src, href,* and *action* in the HTML content, and JavaScript events such as *window.location, window.open, .location.assign, .href, .load, .action,* and *.src* as well for identifying the URLs in a web page. The URLs captured are stored in a list, and the crawler starts exploring the web pages of the application in a depth-first search fashion until all the web pages are visited. The working principle and algorithm of the crawler component can be found in Palsetia et al. (2016). The parameters that are flowing between different web pages are observed and noted. The response obtained is stored in the database for further processing. The coverage property of the crawler cannot be assured due to the following reasons: (i) The crawler does not consider the links embedded in active contents like Flash and Silverlight, and does not take into account AJAX requests, documents, images, etc. (ii) The crawler does not support type-enhanced form submission i.e., the crawler does not consider the constraints placed on the input fields in the application. As a result, the request for that particular web page may be ignored by the application server, and hence few web pages might be missed from exploration.

6.1.2. Session extractor

Sessions can be maintained in two ways - purely at the client-side by means of cookies, or a combination of server and client-side. In the latter approach, a cookie containing a unique session identifier (i.e., session id) is maintained at the client-side. For each request, this cookie is sent to the server, which then matches the session information stored on the server using the session id. The session id is used to extract the actual session variable name and its value, and then mapped with the URL. Session information extraction for PHP applications is described as follows. Session information is stored at the server-side indexed by the session id. The session id is usually passed with the HTTP response headers. The exception to this is Ruby, which passes the entire session data through cookies. PHP stores session information in a temporary folder. On a Linux installation, it is usually stored in */opt/lamp/temp/* folder. The session information is serialized and stored in the session file as variable name-value pairs. This information is deserialized and extracted for each request-response pair.

6.2. Learning phase

6.2.1. Model generator

Algorithm 1 presents the pseudocode for the construction of the annotated FSM from the execution traces. The algorithm observes the HTTP requests and responses while the tester is navigating through the application. Whenever a request is made, it is added to the input symbol set (Σ). The routine *ExtractParams()* retrieves the HTTP parameters from the request and stores them in the parameter set (P). On receiving a response, the algorithm verifies if the web page has been already visited. If not visited, then a new state is created and a transition is marked from the source state with the session variables and parameters associated with the transition added to the annotation set. The initial state and source state are initialized to null values before execution of the algorithm. The first state created during execution of the algorithm is assigned to the variable q_0. If the web page is already visited, then the state corresponding to that web page is retrieved using the function *getExistingState()*, and a new transition is created between the source state and existing state. If a transition already exists between the two states with different session variables, then the new session values are appended to the corresponding transition function. The new state created or the current state (i.e., the web page in which the user is currently in) becomes the source state for the next transition.

The session variables are extracted using the routine *ExtractSession()*, and are added to the set of session variables (S) in the annotation set (A). The routine identifies the server-side technology before extracting the session. As mentioned in Section 6.1.2, the module 'session extractor' maps web page URLs with session variables, and stores the following pieces of information: login information, HTTP parameters and their values, session variables and their values, and HTML responses.

After construction of the model, the transition functions leading to a state with same input symbol, and different values for HTTP parameters and session variables are analyzed, and the values of the session variables (S) representing the role of the user are extracted and stored in the variable *SetofAccessibleRoles*. This gives information regarding the roles having access to a particular state (i.e., web page).

Algorithm 1: Model construction

Data: Web Requests and Responses
Result: Model of the Web Application

```
1  Q, P, S, Σ, δ = []; q_0 = null; q_s = null; //q_s is Source state
2  while User is Browsing do
3      Read data;
4      if request then
5          Add request to Σ;
6          if request contains parameters then
7              P = ExtractParams(request);
8          end
9      else
10         //Code to handle HTTP Response
11         if Web Page has not been visited before then
12             q_d = CreateNewState(); //q_d is Destination state
13             if q_0 is null then
14                 q_0 = q_d; //Initial state of the FSM is initialized
15             end
16             t = CreateNewTransition(q_s, q_d, S, P, request);
17             Add q_0, q_d to Q;
18             Add t to δ;
19         else
20             //Web page already visited, get the state from Q
21             q_d=getExistingstate(Q);
22             t = CreateNewTransition(q_s, q_d, S, P, request);
23             if t exists in δ with different Session Values then
24                 t[session].add(S); //Append new Session values to existing record;
25             end
26         end
27         q_s = q_d;
28         S = ExtractSession();
29     end
30 end
```

Fig. 6 shows the annotated FSM generated for one of the test applications, Scarf. It is not the complete FSM, and it shows only a few pages that were explored by the tester. The obtained FSM is visualized using Graphviz,[3] a graph visualization software.

6.2.1.1. Inference of access-control and workflow constraints. Once the model of the web application is obtained, we attempt to learn the constraints imposed on the request-response pairs, the session variables, and the parameters by navigating through the model.

For identifying the different types of access-control vulnerabilities, the following types of web pages are to be identified by the prototype.

- Web pages that can only be accessed after logging in,
- Web pages that can only be accessed by certain roles, and
- Web pages that are customized for individual users and should not be accessible to other users with the same privilege.

The first set of web pages help in identifying pages which are missing access checks leading to authentication/authorization bypass attacks. The second and third types of conditions help in identifying pages with incomplete and improper access checks leading to vertical and horizontal privilege escalation attacks respectively. The aforementioned types of web pages can be identified from the observation of session variables and HTTP parameters and their respective values at each state of the constructed model.

For identifying workflow bypass vulnerabilities, the sensitive pages in the application are identified from the FSM. The sensitive pages are discovered by finding cut-vertices in a directed graph extracted from the FSM, and stored as workflow constraints for further processing to verify whether they can be bypassed. The algorithm developed for inference of constraints navigates through the FSM to extract the access-control and workflow constraints. The details regarding extraction of access-control and workflow constraints are found in Sections 4.2.2.1 and 4.2.2.3.

6.2.2. HTML/JavaScript analyzer

The objective of the learning phase for parameter manipulation is to extract restrictions placed on the user-supplied input so as to verify whether the same restrictions are implemented at the server-side. The restrictions placed on the value of the parameters at the client-side are stored as parameter-related constraints. To identify the client-side restrictions, we develop an HTML/JavaScript analyzer as suggested in NoTamper (Bisht et al., 2010). The analyzer extracts constraints imposed (if any) on each field of the web form. The 'constraints' provide information about the possible values that a field in the form can hold, form fields which are mandatory, etc. The validation code in the JavaScript is also taken into account for building the constraints. The analyzer takes the URL of the web page, HTML content of the web page, and inline JavaScript code as input, analyzes the source code, and provides the set of constraints imposed on the input parameters as output. The working principle of HTML/JavaScript analyzer can be found in Deepa et al. (2017).

The analyzer extracts three different types of constraints on the parameters flowing between web pages: (i) Data type constraint, (ii) Value constraint, and (iii) Length constraint. Data type constraint states the type of value that should be provided as input by the user. For example, an input field may prompt the user to enter only alphabets. Value constraint refers to the value that an input field can hold. This constraint is represented in the form of a regular expression against the input field. Length constraint refers to the number of characters that an input can hold. The type and values of the constraints are stored in the database of the prototype for further processing.

As explained in Section 2.2, the proposed approach for identifying parameter manipulation vulnerabilities concentrates on identifying the inconsistencies existing between client-side and server-side validations, and therefore inference of constraints from the JavaScript code helps in identifying validations missing at the server-side. Thus, the proposed approach ensures completeness in identifying the vulnerabilities with respect to the defined scope.

6.3. Attack generation phase

This section describes in detail the algorithms used for generating attack vectors from the inferred constraints to identify the three types of logic vulnerabilities.

[3] http://www.graphviz.org/.

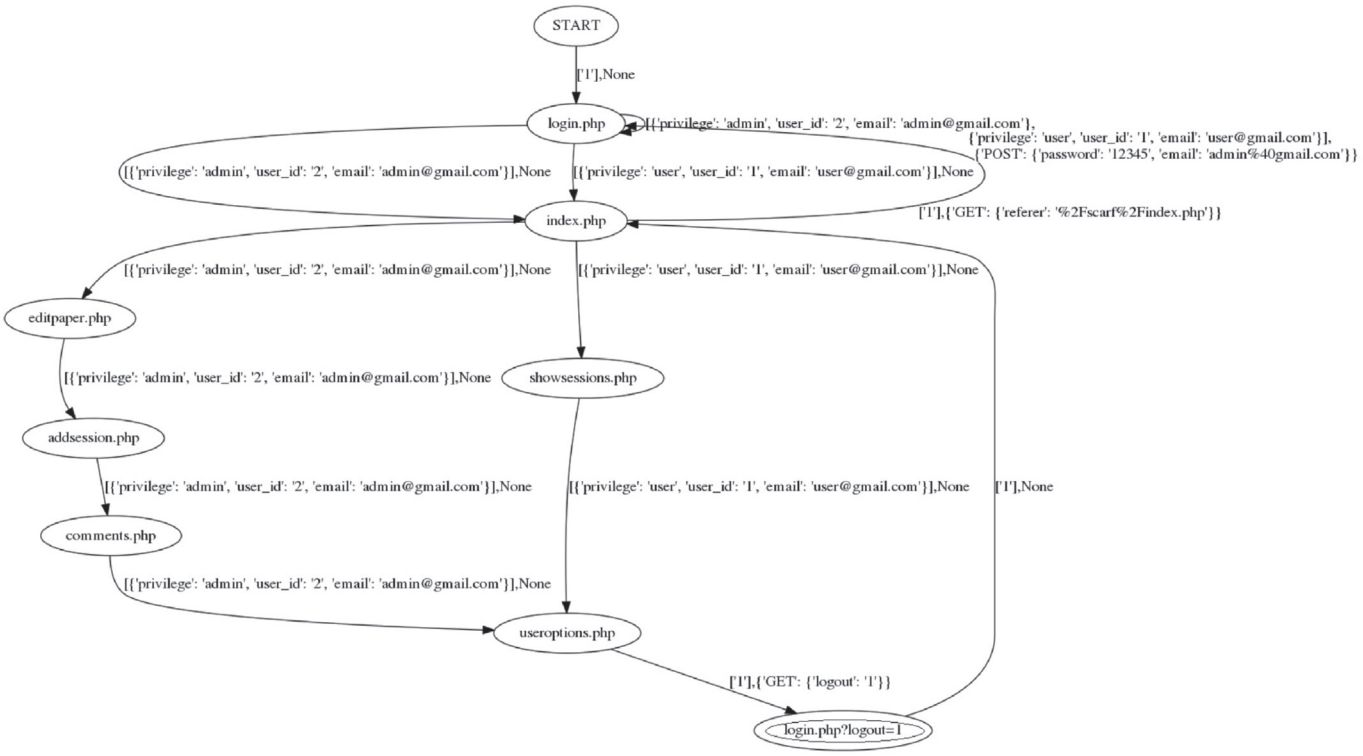


Fig. 6. Annotated finite state machine for application Scarf.

6.3.1. Parameter manipulation

The constraints extracted from the HTML/JavaScript analyzer are stored in a database and are used for generating attack vectors to identify the vulnerabilities. The attack vectors generated are as follows:

- Providing values for the parameters in such a fashion that the constraints imposed on the parameter are violated. Each type of constraints imposed on the input field such as data type constraint, value constraint, and length constraint are violated for identifying vulnerabilities. The constraints against each form field stored in the database of the prototype provides information regarding the type of constraint and the value, which are retrieved and violated to generate attack vectors. For example, consider a form field *quantity* which is supposed to have an integer value within a range of $0-10$. In such cases, attack requests are submitted with the following malicious inputs: (i) a value less than zero ($quantity = -5$), and (ii) a value greater than ten ($quantity = 20$). Thus, each extracted constraint is violated for identifying the vulnerabilities.
- Appending a parameter which is not supposed to be present in the HTTP request for a web page. The constraints extracted from the JavaScript analyzer have information regarding the parameters that should flow from one page to another page, as well as the parameter that goes along a request only for a user with particular privilege. The constraint is violated by adding the parameter in the request of a user with lower privilege. From the motivating example discussed in Section 3, the parameter *mode* with value as *insert* in *Create.php* is supposed to be present only for *physicians* and *administrator*. If the application is vulnerable, then the parameter *mode* can be included in the HTTP request by a guest user of the application consequently permitting the guest user to create new records.

The response obtained during attack generation is stored and compared with the response obtained during the normal execution of the application. When a success response is received for the attack vector, then the form fields submitted with invalid values are reported as vulnerable injection points.

6.3.2. Access-control violation

After extraction of access-control constraints from the model, the next step is to violate the constraints. The attack vectors generated for identifying access-control vulnerabilities are as follows:

Authentication Bypass: This attack focuses on identifying web pages that need to be accessed by privileged users, but fail to impose an access check in the source code. Such web pages are accessed as a third-party/non-privileged user by submitting an HTTP request for the web page without submitting valid credentials. In other words, a web page in an application is accessed by any user without logging in. This is called authentication bypass attack. The set of states (Q) and transition functions (δ) are fed as inputs for generation of attack vectors. The state set Q is utilized for identifying states that are accessed with session variables set in the transition function leading to that state. Requests are placed for those web pages without defining session variables, and the responses obtained are stored. The responses obtained for those requests are compared with the responses obtained during normal execution for reporting vulnerabilities.

For the application Scarf shown in Fig. 6, the web pages *addsession.php* and *useroptions.php* can be accessed by users after logging in. The prototype places requests for these web pages without setting proper user credentials to identify if the page is vulnerable to authentication bypass attack.

Vertical Privilege Escalation: This attack focuses on identifying the web pages with incomplete access checks. In such cases, highly privileged web pages are accessed by putting forward an HTTP request for the web page with a valid username in a parameter, but without proper session variables set for the privilege. This type of attack is called vertical privilege escalation attack. Algorithm 2 explains the

steps followed for generation of the attack. The first step is to identify privileged pages and the roles having access to the respective pages. To achieve this, transitions leading to same destination state in the FSM and the set of roles having access to that state are identified. The roles having access to the state are extracted from the session variables and stored in a list named *SetofAccessibleRoles*. Exclusion of *SetofAccessibleRoles* from the total set of roles intended for the application gives the set of non-accessible roles *(AttackRoles)* for the state. The list *AttackSet* is updated with the URL of the destination state and the set of roles not having the access to the state. The process is repeated until all privileged pages in the application are identified. After identifying the privileged pages, attack requests are submitted for those privileged web pages with non-privileged roles. The responses obtained for those attack requests are compared with the responses obtained during normal execution for reporting vulnerabilities.

Algorithm 2: Vertical privilege escalation attack generation

```
Data: Q and δ
Result: Vulnerability Report
AttackSet=[];
forall t in δ with same t.DestinationUrl do
    AttackRoles = SetofRoles - SetofAccessibleRoles;
    AttackSet.add(t.DestinationUrl,AttackRoles);
end
forall t in AttackSet do
    forall role in AttackRoles do
        Login(role);
        attack_response = SendRequest(destinationUrl);
        learnt_response = destinationUrl.dom;
        if attack_response = learnt_response then
            Report a vulnerability;
        end
    end
end
```

For the application Scarf shown in Fig. 6, the web pages *editpaper.php* and *comments.php* can be accessed exclusively by an *admin* user. The attack generator module places HTTP requests for these web pages with the parameter *username* set as a normal user to identify if the pages are vulnerable to vertical privilege escalation attack.

Horizontal Privilege Escalation: This attack focuses on identifying web pages with improper access checks. Such kind of vulnerabilities lead to horizontal privilege escalation attacks which mean a web page intended for a particular user can be accessed by any other user with the same privilege. In such cases, the web pages are accessed with the *username* parameter set to any valid user registered with the application with the same role, and the responses are compared for reporting vulnerabilities. Algorithm 3 explains the steps followed for generation of the attack. The first step is to identify transition functions having identical values for any of the parameters and session variables. The second step is to place requests for the destination state of those transition functions with value set for the parameter different from the value of the session variable. The responses obtained for those attack requests are compared with the responses obtained during normal execution for reporting vulnerabilities.

Algorithm 3: Horizontal privilege escalation attack generation

```
Data: Q and δ
Result: Vulnerability Report
AttackSet=[];
forall t in δ do
    forall p in t.Params do
        forall s in t.Session do
            if p.value = s.value then
                AttackSet.add(t.DestinationUrl, p, s);
            end
        end
    end
end
forall t in AttackSet do
    Login(Session);
    p = ModifyParameter(Params);
    attack_response = SendRequest(destinationUrl,p);
    learnt_response = destinationUrl.dom;
    if learnt_response = attack_response then
        Report a vulnerability;
    end
end
```

For the application Wackopicko (Doupé et al., 2010), the web page *view.php* intended for *user*1 is accessed by *user*2 by modifying the value of the parameter *userid* to *user*2 in the HTTP request. If the response obtained for *user*2 is the same as the response received for the *user*1, then vulnerability will be reported.

Session Puzzling: In addition to the aforementioned cases, the prototype identifies session puzzling also known as session variable overloading (Chen, 2011), a kind of attack which targets the vulnerabilities in session variables. To the best of our knowledge, no prior work has been reported to capture the session variable overloading. To identify session puzzling, the prototype identifies transitions where the same session variables are set along different paths in the constructed FSM. This does not pose a problem, if the parameters passed to set the session variables are same in each case. A potential problem arises when the same session variables are set, but lesser information is applied to define it. Referring to Bug 5 discussed in Section 3, the session variable *username* is set at *Login* page using the user-input parameters *username* and *password*. However, in *ForgotPwd* page, the same session variable is set by accepting just the *username*. Again, this cannot be regarded as a vulnerability. This is because, other session variables may be set to maintain a state such that the user is not perceived as fully logged in. If requests are sent to pages where the *username* parameter is necessarily set, and the response obtained is similar to the original responses, then this becomes a vulnerability.

6.3.3. Workflow bypass

The algorithm for initiating workflow bypass attacks firstly extracts the critical web pages in the application. The critical web pages are identified from the obtained web application model as mentioned in Section 4.2.2.3. For example, the critical web pages identified in the applications Scarf and Wackopicko are *Login* and *Index*, and *Login*, *Guestbook*, *View* and *Preview_comment* pages, respectively. The workflows (i.e., paths) containing these critical web pages are only considered for launching attacks. For each valid workflow with critical web pages, the prototype places HTTP requests sequentially for the pages

in the workflow. However, it fails to place request for the critical web page and instead places request for the next web page (i.e., the page that follows the critical web page). Thus, skipping a web page from a valid workflow is called a workflow bypass attack. The prototype observes the response obtained for the invalid workflow followed. If the response obtained for the attack is the same as that of the response obtained during normal execution, then vulnerability is reported. The process is repeated for the other identified workflows and vulnerabilities are reported accordingly. Algorithm 4 presents the pseudocode for generating workflow bypass attacks to identify vulnerable web pages.

Algorithm 4: Workflow bypass attack generation

```
Data: Q and δ
Result: Vulnerability Report
//Function to retrieve workflows
workflow_list = extractWorkflows(Q, δ);
//Function to identify critical pages
criticalNodes_list = critical page(Q, δ);
foreach workflow in workflow_list do
    foreach node in workflow do
        if node not in criticalNodes_list then
            attack_response = sendRequest();
        end
    end
    learnt_response = getValidResponse(workflow);
    if learnt_response = attack_response then
        Report a vulnerability;
    end
end
```

6.4. Discovery phase

The discovery phase is common for all three types of vulnerabilities. In the discovery phase, the HTML responses obtained during learning and attack generation phases are compared to identify vulnerabilities. The comparison is performed using an HTML response comparator.

HTML response comparator: The responses obtained during learning and attack generation phases are passed as input to the comparator, which compares the templates of the web pages. If the comparison yields a positive response, then vulnerability is reported. The vulnerabilities identified are recorded in a report which gives information about the web page that is vulnerable, the vulnerability that is exploited for launching the attack, and the type of attack launched.

Vulnerability report generation: A consolidated report is generated listing different types of business logic vulnerabilities in each web page, the vulnerable point that is exploited, the type of attack which exploited the vulnerability, the attack string, etc., which aids the developers to fix the flaws so as to guarantee security of the application.

Table 3
Number of constraints extracted.

Application	#Length Constraints	#Data Constraints	#Value Constraints	#Total Constraints
BookStore	16	48	49	113
Classifieds	12	18	14	44
Events	0	12	8	20
Employee Directory	0	8	16	24

7. Evaluation

This section presents the applications utilized for testing and a proper discussion on the results obtained during testing. The prototype DetLogic is implemented using Django web framework with Python. Redis, a data structure server, is used to store the information regarding the web application under test for further processing by the prototype. DetLogic is evaluated along two dimensions: (i) effectiveness of the prototype from the identified vulnerabilities, and (ii) performance of the prototype.

7.1. Experimental effectiveness

7.1.1. Parameter manipulation vulnerabilities

The applications of Halfond and Orso (2005) such as BookStore, Classifieds, Employee Directory, and Events are instrumented with parameter manipulation vulnerabilities for the purpose of evaluating our prototype. Table 2 shows the results obtained while testing the prototype against the four open-source web applications. The table gives the total number of forms existing in the application, the number of forms vulnerable to parameter manipulation vulnerabilities, number of vulnerabilities existing in the application and detected by the prototype. It can be inferred from the table that the prototype did not report any false positives. Table 3 gives the different types of constraints extracted for the test applications.

7.1.2. Access-control vulnerabilities

Table 4 presents the applications that are used for testing access-control and workflow bypass vulnerabilities. The benchmark applications Scarf, Wackopicko (Doupé et al., 2010), OpenIT, and Puzzlemall (Chen, 2011) are utilized for testing access-control vulnerabilities. Three of these web applications have been used extensively in the literature (Li and Xue, 2011, 2013; Li et al., 2012; Li et al., 2014). The fourth one is a relatively new open-source web application that exhibits session puzzling. Table 5 gives the results obtained while testing the application for access-control vulnerabilities. As seen from the results, the approach works relatively well on most of the web applications. Table 6 shows the different types of access-control vulnerabilities reported by DetLogic.

Discussion: One of the drawbacks of this approach is that the model is constructed using only a single trace generated by the user. Even though this is sufficient for identifying some kinds of attacks, this may

Table 2
Parameter manipulation vulnerabilities detected.

Application	#URLs	#Forms	#Vulnerable Forms		#Vulnerabilities		#FP	#FN
			Existing	Detected	Existing	Detected		
BookStore	477	32	5	5	64	64	0	0
Classifieds	3,465	20	16	16	86	86	0	0
Events	67	12	11	11	39	39	0	0
Employee Directory	141	9	8	8	42	42	0	0

FP – False Positives, FN – False Negatives.

Table 4
Applications for evaluation.

Application	Description	Vulnerabilities existing in the application	References
Scarf	Conference management system	Authentication bypass	CVE-2006-5909
Wackopicko	Image management system	Access-control vulnerabilities	Li and Xue (2011); Li et al. (2012); Li and Xue (2013); Li et al. (2014)
OpenIT	IT management	Access-control vulnerabilities	Bisht et al. (2010)
Puzzlemall		Session variable overloading	Chen (2011)
OsCommerce	eCommerce application	Workflow bypass (Instrumented)	Li and Xue (2011)

Table 5
Access-control vulnerabilities detected.

Application	#LOC	#States	#Attack Requests	# Vulnerabilities		#TP	#FP	#FN
				Existing	Detected			
Wackopicko	4,037	12	9	4	3	3	0	1
Scarf	1,913	12	27	8	9	8	1	0
OpenIT	26,035	14	2	2	2	2	0	0
Puzzlemall	1,979	19	1	1	1	1	0	0

LOC – Lines of Code, TP – True Positives, FP – False Positives, FN – False Negatives

result in false positives while detecting horizontal privilege escalation. This is because our technique assumes the constraints based on the available traces. If only one trace is presented, then it can erroneously assume some constraints that are actually not constraints. This results in false positives in the case of the application Scarf. As an example, the *addsession.php* page in Scarf application uses a parameter *month* which is an integer value ranging from 1 to 12. Assuming the input provided by the tester for the parameter *month* is 5, and by chance the user logged in also has a session variable *userid* set to 5, then the prototype erroneously infers a constraint stating that the value of the parameter *month* and the session variable *userid* should be same. Unfortunately, this is not true. This drawback can be removed by using multiple traces and stricter constraint extraction.

DetLogic reports one false negative in Wackopicko. The reason behind this is that *highquality.php* page in the application allows the attacker to view high-quality pictures without purchasing them by manipulating the *picid* parameter in the request. This relationship between users and pictures is not incorporated as part of session variables and exists in the database table "own". As a result, the proposed approach cannot capture the constraint when constructing the FSM model. Therefore, no attack vectors will be generated to violate such constraint, and hence is missed from being identified.

Comparison with LogicScope: The results are comparable to existing work LogicScope (Li and Xue, 2013). Table 7 gives the comparison of the results obtained for DetLogic with the results as stated in LogicScope. Puzzlemall is not used in LogicScope, and hence the number of attack requests launched is marked as NA. The number of states that would be inferred is two, since it has only two session variables. LogicScope is incapable of detecting the session puzzling vulnerability existing in Puzzlemall.

The model generated by LogicScope treats the cartesian product of the session variables and their values as states, while the proposed model considers the web pages as states of the FSM. In addition, the model proposed by LogicScope is not flexible enough to address vulnerabilities leading to session puzzling attack while DetLogic captures session puzzling. The proposed model is also adaptable to identify other types of vulnerabilities such as EAR vulnerability, and vulnerabilities arising due to improper handling of session identifiers.

7.1.3. Workflow bypass vulnerabilities

The applications Scarf, Wackopicko, OpenIT, and OsCommerce are utilized for testing workflow bypass vulnerabilities, and Table 8 gives the results obtained. Scarf and Wackopicko have *Login* pages that can by bypassed by *comments.php* and *Generaloptions.php*, and *View.php* and *Upload.php*, respectively. OsCommerce, a commercially used eCommerce application, is instrumented with a coding flaw in *checkout_payment* page, where the CSRF token set in the cookie at *checkout_shipping.php* is not validated against the HTTP parameter leading to a workflow bypass attack. As a result, a user could pay for the purchased items without inclusion of the shipping charges. The state machine inferred from OsCommerce website with role as customer has 31 states. Thus, the number of states inferred for a real time application is less than 50 due to consideration of the web page templates.

Discussion: While BLOCK (Li and Xue, 2011) and LogicScope (Li and Xue, 2013) detect workflow bypass resulting due to improper session management, the proposed approach is capable of detecting workflow bypass attacks resulting due to improper management of CSRF tokens as well. The correctness of results obtained using the approach depends on the model which is generated using manual traces. Therefore, to achieve better accuracy the tester need to explore all possible workflows.

7.2. Performance evaluation

The performance of the prototype is evaluated using the following metrics: true positive rate (TPR), false positive rate (FPR), false negative

Table 6
Types of access-control vulnerabilities detected.

Application	#Missing access check	#Incomplete access check	#Improper access check	#Session variable overloading
Wackopicko	2	0	1	0
Scarf	3	4	2	0
OpenIT	0	0	2	0
Puzzlemall	0	0	0	1

Table 7
Comparison of the vulnerabilities detected by DetLogic with LogicScope (Li and Xue, 2013).

Application	DetLogic						LogicScope					
	# States	# Attacks	# Vulnerabilities (E)	# Vulnerabilities (D)	# False Positives	# False Negatives	# States	# Attacks	# Vulnerabilities (E)	# Vulnerabilities (D)	# False Positives	# False Negatives
Wackopicko	12	9	4	3	0	1	2	21	4	2	0	1
Scarf	12	27	8	9	1	0	3	49	8	10	2	0
OpenIT	14	2	2	2	0	0	5	65	2	3	1	0
Puzzlemall	19	1	1	1	0	0	2	NA	1	0	0	0

E – Existing, D – Detected.

Table 8
Workflow bypass vulnerabilities detected.

Application	#LOC	#States	#Critical Nodes	# Vulnerabilities		#FP	#FN
				Existing	Detected		
Scarf	1,913	12	2	2	2	0	0
Wackopicko	4,037	12	4	2	2	0	0
OpenIT	26,035	14	1	0	0	0	0
OsCommerce	22,642	31	17	1	1	0	0

LOC – Lines of Code, FP – False Positives, FN – False Negatives.

rate (FNR), precision, and recall. The FPR and FNR of the prototype for both parameter manipulation and workflow bypass vulnerabilities are 0%. With regard to detection of access-control violations, the FPR and FNR are 5% and 6.25% respectively. The precision and recall percentage values are 99.1% and 97.9% respectively. The overall TPR of the prototype is found to be 97.9%.

The time required for generating the application model depends on the time taken by the tester to navigate through the application. This is because, the prototype is a proxy and the model is constructed during navigation itself. Additionally, the time complexity for finding critical pages from the control flow graph is $O(V(V+E))$, where V is the number of vertices and E is the number of edges in the graph. The adjacency list representation of graph is used for performing the depth-first search.

7.3. Limitations and extensions

The advantages of the proposed approach are as follows:

- The generated FSM represents every aspect of the interactions between the web application and the user.
- The model is extensible and adaptable for identifying improper handling of session tokens and variables. The flexibility offered by the model renders a way to detect session puzzling vulnerabilities.
- The generated model can be used for detection of EAR vulnerabilities as well. The first type of access-control constraint inferred from the FSM (i.e, web pages at which session variables are defined and set) can be used for identifying EAR vulnerabilities. During attack generation, a request for the web page that defines the session variables is placed with invalid values for the input parameters. If the application is not vulnerable, then the request will be redirected to an error page. Consecutively, a second attack request is placed for the web page that follows the previous web page. If a success response is received, then EAR vulnerability is reported. Thus, the inferred constraint can be effectively used for identifying EAR vulnerabilities by placing two attack requests consecutively.
- The cut-vertices identified from the control flow graph of FSM can be used to identify the severity of vulnerabilities. The out-degree of the cut-vertex node can be used for ranking the nodes which in turn would help in marking the severity of web pages in the application.

Even though the prototype works based on a black-box approach in an automated fashion, few limitations exist which are listed as follows.

- The prototype can detect parameter manipulation vulnerabilities in applications that involve JavaScript in the client-side code validation. Other technologies such as JQuery, VBScript are not taken care of by the prototype.
- The approach followed for detection of access-control vulnerabilities requires the session information at the server-side of the application.
- The crawler component employed for exploring the application does not address AJAX applications, and does not support type enhanced form submission.
- Manual trace generation may not explore all possible workflows in the application. The correctness and completeness of the results depend on the model generated which in turn depends on the user navigating through the web application under test. When the user does not explore all the possible navigations within the application, the vulnerabilities in unvisited pages may be missed from being identified.
- The proposed prototype is not tested on applications developed using recent technologies such as JQuery, AngularJS, and ReactJS which are more dynamic in nature. However, as the proposed model considers the DOM structure of the web pages in addition to URL as the states, the proposed prototype should be able to handle modern applications involving these recent technologies.
- The proposed prototype can be suitably tested on large web applications. The term ‘large’ is subjective in nature, and it can refer to various factors such as the number of lines of code, number of web pages, number of users of the application, and the amount of data handled by the application. However, the proposed prototype works based on the available workflows within the application, and hence the approach should be scalable for large applications.

The proposed prototype can be enhanced by using a simulated crawler to overcome the usage of two components: crawler and man-

Table 9
Comparison of the prototype with existing literature.

Research article	System type	Analysis	Type of flaws							Remarks
			MAC	IAC	PO	MSV	SeV	StV	SO	
Swaddler (Cova et al., 2007)	AP	B						✓		Involves code instrumentation
Nemesis (Dalton et al., 2009)	AP	B	✓	✓						Imparts performance overhead
BLOCK (Li and Xue, 2011)	AP	B					✓	✓		Offline-learning and runtime protection, Imparts performance overhead
SENTINEL (Li et al., 2012)	AP	B						✓		Offline-learning and uses proxy to capture queries, Blocks malicious queries
Bypass-Shield (Mouelhi et al., 2011)	AP	B				✓				Detects MSV related to SQL injection and XSS
TamperProof (Skrupsky et al., 2013)	AP	B				✓	✓			Involves code instrumentation, Solution deployed in proxy
Flowwatcher (Muthukumaran et al., 2015)	AP	B	✓	✓						Proxy-based solution
MiMoSA (Balzarotti et al., 2007)	VD	W						✓		
Waler (Felmetsger et al., 2010)	VD	W						✓		Uses Symbolic model checking to identify logic flaws
Sun et al. (2011)	VD	W	✓							Requires developer to specify set of roles and their entry points
RoleCast (Son et al., 2011)	VD	W	✓							
SaferPHP (Son and Shmatikov, 2011)	VD	W	✓							
WAPTEC (Bisht et al., 2011)	VD	W				✓				Employs symbolic evaluation
ViewPoints (Alkhalaf et al., 2012)	VD	W				✓				Detects MSV related to SQL injection and XSS
FixMeUp (Son et al., 2013)	VD	W	✓							Repairs the access-control flaws
MACE (Monshizadeh et al., 2014)	VD	W	✓	✓						Requires annotations from the developers
ASIDE (Zhu et al., 2015)	VD	W	✓	✓						Employs interactive static analysis
LogicPatcher (Monshizadeh et al., 2016)	VD	W	✓	✓		✓				Repairs the logic flaws
NoTamper (Bisht et al., 2010)	VD	B				✓				Given a web page, identifies client-server inconsistencies
PAPAS (Balduzzi et al., 2011)	VD	B			✓					Identifies parameter overriding vulnerabilities
LogicScope (Li and Xue, 2013)	VD	B	✓	✓			✓			Identifies sequence violation due to missing validation of session variables
BATMAN (Li et al., 2014)	VD	B	✓	✓						Uses HTTP proxy to intercept HTTP packets and SQL proxy to intercept SQL queries
Scout (Wen et al., 2016)	VD	B	✓	✓						Addresses MongoDB applications
DetLogic	VD	B	✓	✓	×	✓	✓	✓	✓	

AP – Attack Prevention, VD – Vulnerability Detection, W – White-Box, B – Black-Box, MAC – Missing Access Check, IAC – Incomplete/Improper Access Check, PO – Parameter Overriding, MSV – Missing Server-Side Validation, SeV – Sequence Violation, StV – State Violation, SO – Session Variable Overloading.

ual trace collection. The simulated crawler requires a user to navigate through the application, and fill in all the forms in the application with valid values. The sequence of operations in the application and the possible values for form fields are recorded which can be later replayed during the learning phase.

8. Related work

This section discusses the programming frameworks that can be used for preventing logic vulnerabilities, and the works that target towards identifying the vulnerabilities using white-box and black-box approaches. A comprehensive review on logic vulnerabilities can be found in Li and Xue (2014), Prokhorenko et al. (2016), and Deepa and Thilagam (2016).

8.1. Secure programming frameworks

Swift (Chong et al., 2007b,a) and Ripley (Vikram et al., 2009) are frameworks that avoid client-side and server-side inconsistencies by placing sanitization code at appropriate locations (i.e., client-side/server-side). Resin (Yip et al., 2009) prevents attacks due to SQL injection, XSS, and missing access-control checks during runtime, and requires the developers to specify the data flow assertions that define security policies for the application. The frameworks Aura (Jia et al., 2008), Fable (Swamy et al., 2008), SELinks (Corcoran et al., 2009), FINE (Swamy et al., 2010), Aglet (Morgenstern and Licata, 2010), and Capsules (Krishnamurthy et al., 2010) require participation of the programmers to specify security policies related to information flow and access-control of web applications, and verify if the policies are properly applied. All of the aforementioned frameworks implicitly provide necessary correction to ensure security of the applications in the process of development. The limitation with these frameworks is that they cannot ensure security of deployed applications.

8.2. White-box detection of logic flaws

WAPTEC (Bisht et al., 2011) offers a multi-tiered analysis for detection of parameter tampering vulnerabilities. ViewPoints (Alkhalaf et al., 2012) employs differential string analysis to discover client and server-side inconsistencies existing in the input sanitization functions.

MiMoSA (Balzarotti et al., 2007) characterizes the expected behavior of the application considering the interactions across intra and inter-modules of the application for identifying logic violations. Waler (Felmetsger et al., 2010), an extension of MiMoSA, extracts the application-specific properties in the form of invariants, analyzes the invariants using symbolic model checking, and identifies violations. While the prototypes developed by Sun et al. (2011), Son and Shmatikov (2011), and Son et al. (2011) detect access checks that are omitted in the application, MACE (Monshizadeh et al., 2014) detects access-control vulnerabilities arising due to both incomplete and omission of access checks in

the application. Zhu et al. (2015) developed a prototype for discovering access-control vulnerabilities using static analysis approach; the prototype requires intervention of the developers to specify the application-specific policies.

8.3. Black-box detection of logic flaws

NoTamper (Bisht et al., 2010) infers the specification of the application from web page form processing and validation code available at the client-side, and extracts the constraints on user-supplied input. The tool submits user-input to the application in such a way that the constraints placed on the user-input are ignored, and the response obtained for these invalid input are observed for disclosing vulnerabilities. While NoTamper, WAPTEC, ViewPoints, and Bypass-Shield (Mouelhi et al., 2011) detect vulnerabilities that stem from client-side and server-side inconsistencies, PAPAS (Balduzzi et al., 2011) discovers vulnerabilities that allow an attacker to override the existing value of the parameter with an injected value.

LogicScope (Li and Xue, 2013) identifies logic vulnerabilities by modeling the behavior the web application using an FSM. Vulnerabilities are reported if there exists dissimilarity between the expected and actual FSMs. The prototype BATMAN (Li et al., 2014) discloses access-control vulnerabilities in web applications by constructing attack vectors that violate the user-level and role-level policies inferred during learning of the application. Pellegrino and Balzarotti (2014) generate test cases that are capable of identifying vulnerabilities in applications involving third-party API integrations. Scout (Wen et al., 2016) identifies access-control vulnerabilities in applications which use MongoDB at the backend.

8.4. Repairing of logic flaws

FixMeUp (Son et al., 2013) eliminates access-control vulnerabilities in web applications by repairing the code using existing statements, and in addition, validates the code repair. LogicPatcher (Monshizadeh et al., 2016) identifies incomplete access checks existing in applications with minimal guidelines as compared to FixMeUp, and patches them with fixes.

8.5. Black-box prevention of logic attacks

Swaddler (Cova et al., 2007) detects state-violation attacks by inferring the normal behavior of the application from the session variables. The prototype then monitors the state variables at runtime for identifying deviations from the normal behavior. Mouelhi et al. (2011) devised a prototype for detecting attacks that bypass the client-side validation in web application using black-box approach. TamperProof (Skrupsky et al., 2013) acts similar to a web proxy, and patches an identifier referred as patchID to each web form presented from the server. These patchIDs are used for verifying whether the submitted requests are legitimate requests. TamperProof can be employed to protect applications that are susceptible to parameter tampering attacks.

BLOCK (Li and Xue, 2011) is a proxy deployed between the client and server, and is capable of detecting state violation attacks from a set of invariants inferred from the intercepted HTTP conversations. SENTINEL (Li et al., 2012) intercepts each SQL query, and derives a set of invariants associated with each SQL query. The tool blocks queries which violate the identified invariants, and imposes performance overhead as it intercepts each SQL query for evaluation. Nemesis (Dalton et al., 2009) and FlowWatcher (Muthukumaran et al., 2015) prevent leakage of sensitive data to anonymous and unprivileged users of the application. Table 9 presents the comparison of the proposed prototype with existing literature.

9. Conclusions

This paper presents a prototype to detect three different types of logic vulnerabilities existing in web applications. The prototype employs black-box approach, and forceful browsing for identifying the vulnerabilities. Black-box approach is utilized for inferring the intended specification of the application in the form of constraints, and forceful browsing is used for launching concrete attack vectors that assist in identifying the potential vulnerabilities in the application. The prototype extracts the data flow and control flow of the application using an HTML/JavaScript analyzer and session extractor. The data flow and control flow are used for constructing an annotated FSM which is subsequently used for deriving parameter-related, access-related, and workflow constraints. The constraints are utilized in identifying the three types of logic vulnerabilities. In addition to the three types of vulnerabilities, the prototype is capable of detecting vulnerabilities leading to session puzzling attacks. The prototype has been exhaustively tested on benchmark PHP and Java web applications, and is found to work effectively with a precision, and a true positive rate of 99.1% and 97.9% respectively. Therefore, DetLogic can become an essential tool for ensuring the security of applications against logic attacks irrespective of the functionality of the application under test.

Acknowledgments

This work was a part of the R&D project entitled “Development of tool for detection of XML based injection vulnerabilities in web applications”, and was supported by the Ministry of Communications and Information Technology (MCIT) (currently known as Ministry of Electronics and Information Technology (MeiTY)), Government of India.

References

Acunetix. Acunetix wvs. https://www.acunetix.com/vulnerability-scanner/.
Alkhalaf, M., Choudhary, S.R., Fazzini, M., Bultan, T., Orso, A., Kruegel, C., 2012. Viewpoints: differential string analysis for discovering client- and server-side input validation inconsistencies. In: Proceedings of the 2012 International Symposium on Software Testing and Analysis. ACM, New York, NY, USA, pp. 56–66.
AppScan. Ibm appscan. http://www-03.ibm.com/software/products/en/appscan.
AppScanner. Trustwave app scanner. https://www.trustwave.com/Products/Application-Security/App-Scanner-Family/.
Balduzzi, M., Gimenez, C.T., Balzarotti, D., Kirda, E., 2011. Automated discovery of parameter pollution vulnerabilities in web applications. In: Proceedings of the 18th Network and Distributed System Security Symposium, San Diego, CA, USA.
Balzarotti, D., Cova, M., Felmetsger, V.V., Vigna, G., 2007. Multi-module vulnerability analysis of web-based applications. In: Proceedings of the 14th ACM Conference on Computer and Communications Security, ACM, New York, NY, USA, pp. 25–35.
Bisht, P., Hinrichs, T., Skrupsky, N., Bobrowicz, R., Venkatakrishnan, V.N., 2010. Notamper: automatic blackbox detection of parameter tampering opportunities in web applications. In: Proceedings of the 17th ACM Conference on Computer and Communications Security. ACM, New York, NY, USA, pp. 607–618.
Bisht, P., Hinrichs, T., Skrupsky, N., Venkatakrishnan, V.N., 2011. Waptec: whitebox analysis of web applications for parameter tampering exploit construction. In: Proceedings of the 18th ACM Conference on Computer and Communications Security. ACM, New York, NY, USA, pp. 575–586.
Chen, S., 2011. Session Puzzles - Indirect Application Attack Vectors Technical Report Ernst & Young, http://puzzlemall.googlecode.com/files/Session%20Puzzles%20-%20Indirect%20Application%20Attack%20Vectors%20-%20May%202011%20-%20Whitepaper.pdf.
Chong, S., Liu, J., Myers, A.C., Qi, X., Vikram, K., Zheng, L., Zheng, X., 2007a. Secure web applications via automatic partitioning. In: Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles. ACM, New York, NY, USA, pp. 31–44.
Chong, S., Vikram, K., Myers, A.C., 2007b. SIF: enforcing confidentiality and integrity in web applications. In: Proceedings of the 16th USENIX Security Symposium. USENIX Association, Berkeley, CA, USA. pp. 1:1–1:16.
Corcoran, B.J., Swamy, N., Hicks, M., 2009. Cross-tier, label-based security enforcement for web applications. In: Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data. ACM, New York, NY, USA, pp. 269–282.
Cova, M., Balzarotti, D., Felmetsger, V., Vigna, G., 2007. Swaddler: an approach for the anomaly-based detection of state violations in web applications. In: Recent Advances in Intrusion Detection. Springer Berlin Heidelberg, pp. 63–86. volume 4637 of Lecture Notes in Computer Science.

CSRF. Cross-site request forgery (CSRF) prevention cheat sheet. https://www.owasp.org/index.php/Cross-Site_Request_Forgery_%28CSRF%29_Prevention_Cheat_Sheet.
Dalton, M., Kozyrakis, C., Zeldovich, N., 2009. Nemesis: preventing authentication & access control vulnerabilities in web applications. In: Proceedings of the 18th USENIX Security Symposium. USENIX Association, Berkeley, CA, USA, pp. 267–282.
Deepa, G., Thilagam, P.S., 2016. Securing web applications from injection and logic vulnerabilities: approaches and challenges. Inf. Softw. Technol. 74, 160–180.
Deepa, G., Thilagam, P.S., Khan, F.A., Praseed, A., Pais, A.R., Palsetia, N., 2017. Black-box detection of xquery injection and parameter tampering vulnerabilities in web applications. Int. J. Inf. Secur. 1–16.
Doupé, A., Boe, B., Kruegel, C., Vigna, G., 2011. Fear the EAR: discovering and mitigating execution after redirect vulnerabilities. In: Proceedings of the 18th ACM Conference on Computer and Communications Security. ACM, New York, NY, USA, pp. 251–262.
Doupé, A., Cova, M., Vigna, G., 2010. Why johnny can't pentest: an analysis of black-box web vulnerability scanners. In: Detection of Intrusions and Malware, and Vulnerability Assessment. Springer Berlin Heidelberg, pp. 111–131. volume 6201 of Lecture Notes in Computer Science.
Felmetsger, V., Cavedon, L., Kruegel, C., Vigna, G., 2010. Toward automated detection of logic vulnerabilities in web applications. In: Proceedings of the 19th USENIX Conference on Security. USENIX Association, Berkeley, CA, USA, p. 10.
Grossman, J., 2007. Seven Business Logic Flaws that Put Your Website at Risk. WhiteHat Security. October.
Halfond, W.G.J., Orso, A., 2005. Amnesia: analysis and monitoring for neutralizing SQL-injection attacks. In: Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering. ACM, New York, NY, USA, pp. 174–183.
Jia, L., Vaughan, J.A., Mazurak, K., Zhao, J., Zarko, L., Schorr, J., Zdancewic, S., 2008. Aura: a programming language for authorization and audit. In: Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming. ACM, New York, NY, USA, pp. 27–38.
Jovanovic, N., Kirda, E., Kruegel, C., 2006. Preventing cross site request forgery attacks. In: Securecomm and Workshops, pp. 1–10.
Krishnamurthy, A., Mettler, A., Wagner, D., 2010. Fine-grained privilege separation for web applications. In: Proceedings of the 19th International Conference on World Wide Web. ACM, New York, NY, USA, pp. 551–560.
Li, X., Si, X., Xue, Y., 2014. Automated black-box detection of access control vulnerabilities in web applications. In: Proceedings of the 4th ACM Conference on Data and Application Security and Privacy. ACM, New York, NY, USA, pp. 49–60.
Li, X., Xue, Y., 2011. Block: a black-box approach for detection of state violation attacks towards web applications. In: Proceedings of the 27th Annual Computer Security Applications Conference. ACM, New York, NY, USA, pp. 247–256.
Li, X., Xue, Y., 2013. Logicscope: automatic discovery of logic vulnerabilities within web applications. In: Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security. ACM, New York, NY, USA, pp. 481–486.
Li, X., Xue, Y., 2014. A survey on server-side approaches to securing web applications. ACM Comput. Surv. 46 (54), 1–54 29.
Li, X., Yan, W., Xue, Y., 2012. Sentinel: securing database from logic flaws in web applications. In: Proceedings of the Second ACM Conference on Data and Application Security and Privacy. ACM, New York, NY, USA, pp. 25–36.
Monshizadeh, M., Naldurg, P., Venkatakrishnan, V., 2016. Patching logic vulnerabilities for web applications using LogicPatcher. In: Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy. ACM, New York, NY, USA, pp. 73–84.
Monshizadeh, M., Naldurg, P., Venkatakrishnan, V.N., 2014. Mace: Detecting privilege escalation vulnerabilities in web applications. In: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security. ACM, New York, NY, USA, pp. 690–701.
Morgenstern, J., Licata, D.R., 2010. Security-typed programming within dependently typed programming. In: Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming. ACM, New York, NY, USA, pp. 169–180.
Mouelhi, T., Le Traon, Y., Abgrall, E., Baudry, B., Gombault, S., 2011. Tailored shielding and bypass testing of web applications. In: 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation (ICST), pp. 210–219.
Muthukumaran, D., O'Keeffe, D., Priebe, C., Eyers, D., Shand, B., Pietzuch, P., 2015. FlowWatcher: defending against data disclosure vulnerabilities in web applications. In: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security. ACM, New York, NY, USA, pp. 603–615.
Palsetia, N., Deepa, G., Khan, F.A., Thilagam, P.S., Pais, A.R., 2016. Securing native xml database-driven web applications from xquery injection vulnerabilities. J. Syst. Softw. 122, 93–109, http://www.sciencedirect.com/science/article/pii/S0164121216301571, https://doi.org/10.1016/j.jss.2016.08.094.
Payet, P., Doupé, A., Kruegel, C., Vigna, G., 2013. EARs in the wild: large-scale analysis of execution after redirect vulnerabilities. In: Proceedings of the 28th Annual ACM Symposium on Applied Computing. ACM, New York, NY, USA, pp. 1792–1799.
Pellegrino, G., Balzarotti, D., 2014. Toward black-box detection of logic flaws in web applications. In: Proceedings of 21st Network and Distributed System Security Symposium, San Diego, CA, USA.
Prokhorenko, V., Choo, K.K.R., Ashman, H., 2016. Web application protection techniques: a taxonomy. J. Netw. Comput. Appl. 60, 95–112.
QualysGuard,. Qualysguard. https://www.qualys.com/enterprises/qualysguard/.
SANS, 2011. Cwe/sans top 25 most dangerous software errors. http://www.sans.org/top25-software-errors/.
Skrupsky, N., Bisht, P., Hinrichs, T., Venkatakrishnan, V.N., Zuck, L., 2013. Tamperproof: a server-agnostic defense for parameter tampering attacks on web applications. In: Proceedings of the Third ACM Conference on Data and Application Security and Privacy. ACM, New York, NY, USA, pp. 129–140.
Son, S., McKinley, K.S., Shmatikov, V., 2011. Rolecast: finding missing security checks when you do not know what checks are. In: Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications. ACM, New York, NY, USA, pp. 1069–1084.
Son, S., McKinley, K.S., Shmatikov, V., 2013. Fix me up: repairing access-control bugs in web applications. In: Proceedings of 20th Annual Network and Distributed System Security Symposium, San Diego, CA, USA.
Son, S., Shmatikov, V., 2011. SAFERPHP: finding semantic vulnerabilities in PHP applications. In: Proceedings of the ACM SIGPLAN 6th Workshop on Programming Languages and Analysis for Security. ACM, New York, NY, USA. pp. 8:1–8:13.
Sun, F., Xu, L., Su, Z., 2011. Static detection of access control vulnerabilities in web applications. In: Proceedings of the 20th USENIX Conference on Security. USENIX Association, Berkeley, CA, USA, p. 11.
Swamy, N., Chen, J., Chugh, R., 2010. Enforcing stateful authorization and information flow policies in fine. In: Programming Languages and Systems. Springer Berlin Heidelberg, pp. 529–549. volume 6012 of Lecture Notes in Computer Science.
Swamy, N., Corcoran, B., Hicks, M., 2008. Fable: a language for enforcing user-defined security policies. In: 2008 IEEE Symposium on Security and Privacy, SP 2008, pp. 369–383.
Symantec, April 2016. Symantec Internet Security Threat Report Technical Report https://www.symantec.com/security-center/threat-report.
Top10, 2013. OWASP Top Ten 2013 Technical Report, https://www.owasp.org/index.php/OWASP_Top_10#tab=OWASP_Top_10_for_2013.
Trustwave, 2014. 2014 Trustwave Global Security Report Technical Report, https://www.trustwave.com/Resources/Trustwave-Blog/The-2014-Trustwave-Global-Security-Report-Is-Here/.
Trustwave, 2016. 2016 Trustwave Global Security Report Technical Report, https://www.trustwave.com/Resources/Library/Documents/2016-Trustwave-Global-Security-Report/.
Vikram, K., Prateek, A., Livshits, B., 2009. Ripley: automatically securing web 2.0 applications through replicated execution. In: Proceedings of the 16th ACM Conference on Computer and Communications Security. ACM, New York, NY, USA, pp. 173–186.
WebInspect,. Hp WebInspect, http://www8.hp.com/in/en/software-solutions/webinspect-dynamic-analysis-dast/.
Wen, S., Xue, Y., Xu, J., Yang, H., Li, X., Song, W., Si, G., 2016. Toward exploiting access control vulnerabilities within MongoDB backend web applications. In: 2016 IEEE 40th Annual Computer Software and Applications Conference (COMPSAC), pp. 143–153, https://doi.org/10.1109/COMPSAC.2016.207.
Yip, A., Wang, X., Zeldovich, N., Kaashoek, M.F., 2009. Improving application security with data flow assertions. In: Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles. ACM, New York, NY, USA, pp. 291–304.
Zhu, J., Chu, B., Lipford, H., Thomas, T., 2015. Mitigating access control vulnerabilities through interactive static analysis. In: Proceedings of the 20th ACM Symposium on Access Control Models and Technologies. ACM, New York, NY, USA, pp. 199–209.

G Deepa is currently a Ph.D. Scholar with the Department of Computer Science and Engineering, National Institute of Technology Karnataka, Surathkal, India. She received her M.E. degree in Computer Science and Engineering from Anna University, Chennai in 2012. Her research interests include Web application security, Data mining and Software testing.

P Santhi Thilagam is currently an Associate Professor in the Department of Computer Science and Engineering, National Institute of Technology Karnataka, Surathkal (NITK), India. She received her Ph.D. in Computer Science and Engineering from NITK in 2008. Her research interests include Distributed data management, Graph mining and Data security.

Amit Praseed is currently a Ph.D. Scholar with the Department of Computer Science and Engineering, National Institute of Technology Karnataka (NITK), Surathkal, India. He received his B.Tech degree in Computer Science and Engineering in 2014 from CET (College of Engineering Trivandrum), Kerala, India and M.Tech degree in Computer Science and Engineering - Information Security in 2016 from NITK, Surathkal, India. His research interests include web security and information security.

Alwyn R Pais is currently an Assistant Professor in the Department of Computer Science and Engineering, National Institute of Technology Karnataka (NITK), Surathkal, India. He completed his B.Tech. (CSE) from Mangalore University, India, M.Tech. (CSE) from IIT Bombay, India and Ph.D. from NITK, Surathkal, India. His area of interests include Information Security, Image Processing, Computer Vision, Wireless Sensor Networks, and Internet of Things (IoT).