The ISC Int'l Journal of Information Security

July 2021, Volume 13, Number 2 (pp. 117–129)

http://www.isecure-journal.org

IDOT: Black-Box Detection of Access Control Violations in Web Applications

Mohammad Ali Hadavi [1,*], Arash Bagherdaei [1], and Simin Ghasemi [2]

[1] *Faculty of Electrical and Computer Engineering, Malek-Ashtar University of Technology, Iran.*

[2] *Department of Computer Engineering, Payame Noor University (PNU), Iran.*

ARTICLE INFO.

Article history:

Received: October 25, 2020

Revised: March 14, 2020

Accepted: May 26, 2021

Published Online: June 26, 2021

Keywords:

Access Control, Insecure Direct Object Reference (IDOR), Parameter Manipulation, Security, Vulnerability, Web Application

Type: Research Article

doi: 10.22042/isecure.2021.254089.580

dor: 20.1001.1.20082045.2021.13.2.3.8

ABSTRACT

Automatic detection of access control violations in software applications is a challenging problem. Insecure direct object reference (IDOR) is among top-ranked vulnerabilities, which violates access control policies and cannot be yet detected by automated vulnerability scanners. While such tools may detect the absence of access control by static or dynamic testing, they cannot verify if it is properly functioning when it is present. When a tool detects requesting access to an object, it is not aware of access control policies to infer whether the request is permitted. This completely depends on the access control logic and there is no automatic way to fully and precisely capture it from software behavior. Taking this challenge into consideration, this article proposes a black-box method to detect IDOR vulnerabilities in web applications without knowing access control logic. To this purpose, we first, gather information from the web application by a semi-automatic crawling process. Then, we tricksily manipulate legal requests to create effective attacks on the web application. Finally, we analyze received responses to check whether the requests are vulnerable to IDOR. The detection process in the analysis phase is supported by our set theory based formal modeling of such vulnerabilities. The proposed method has been implemented as an IDOR detection tool (IDOT) and evaluated on a couple of vulnerable web applications. Evaluation results show that the method can effectively detect IDOR vulnerabilities provided that enough information is gathered in the crawling phase.

1 Introduction

Web applications have changed from static and read-only systems to dynamic, complex, and interactive systems, which provide plenty of information and services for users. Their widespread use and high availability on the one hand, and the need for prohibiting unauthorized access to classified and confidential information on the other hand have led to the condition in which the existence of even a slight vulnerability could cause irreparable damages.

Web application vulnerabilities can be divided into three classes including injection, session management, and business logic related vulnerabilities [1]. Injection vulnerabilities usually occur due to a weakness in the validation of user inputs. Session vulnerabilities result from a weakness in managing session identifiers. Both

* Corresponding author.

Email addresses: hadavi@mut.ac.ir, bagherdaei@mut.ac.ir, s.ghasemi@pnu.ac.ir

ISSN: 2008-2045

of these vulnerabilities can be properly detected using some methods such as static analysis [2] or fuzzing [3]. The third class of vulnerabilities is related to access control violations and improper authorizations. The main challenge of this type of vulnerabilities is their dependence on the access control logic. While automated testing tools may detect the absence of access control by static or dynamic testing, they cannot verify if it is properly functioning when it is present. According to the latest report of OWASP top-ten vulnerabilities, *broken access control* is the fifth-ranked vulnerability [4].

Insecure direct object reference (IDOR) is a vulnerability in the third aforementioned class of vulnerabilities, results in access control violation. In IDOR, request parameters are manipulated in such a way that direct access to an internal object is illegally allowed. Such direct references can be to a file, a database record, or a key sent via a URL or a form field. Conducted researches also confirm that automated IDOR discovery requires a detection tool empowered with the knowledge of access control policies for the application under evaluation, which in general is a very sophisticated task [5].

1.1 Problem Statement

Increasing services and functionalities of web applications, variety and volume of users, and more granular and complex access policies, make IDOR a prevalent vulnerability. Unfortunately, automated solutions are not yet able to detect IDOR vulnerabilities [6–8]. This is because vulnerability scanners, despite their plurality, cannot determine whether a response received from an application contains an object to which the user must not access.

Considering the above limitations for automated IDOR detection, this paper proposes a black-box method to identify potentially vulnerable requests against IDOR without knowing the access control policies of the application under evaluation. The potentially vulnerable requests are then knowingly manipulated and their responses are examined to detect whether they are actually vulnerable. We limited the implementation of our method to GET and POST HTTP methods. Also, the current implementation acts as a proof-of-concept for our method without focusing on the performance issues.

1.2 Contributions

The contributions of this paper are as follow:

- A black-box method to detect IDOR vulnerabilities without knowing access control policies is presented.

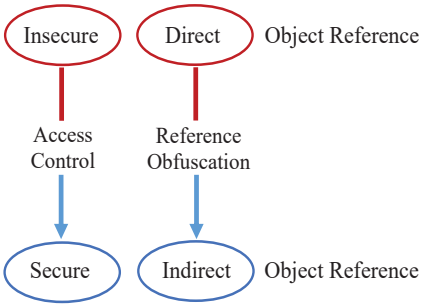

Figure 1. General approaches for IDOR prevention

- The method is formalized based on the set theory to show whether the attack on the application is successful.
- The method is implemented as an IDOR detection tool (IDOT) and is evaluated on vulnerable web applications to show its effectiveness in finding IDOR vulnerabilities.

The rest of the paper is organized as follows. Section 2 details the related work focusing on IDOR prevention and detection approaches. Section 3 presents the proposed method in three subsections namely information gathering, attack phase, and analysis phase, according to three steps of the method. Section 4 reports the implementation issues and evaluation results. Finally, Section 5 concludes the paper and gives some suggestions for future work.

2 IDOR Prevention and Detection Approaches

IDOR vulnerabilities occur when an unauthorized user or process can directly refer to an internal object or a function by manipulating parameters or values in a request. The prevention methods can be classified into two general categories; obfuscating references to objects, and controlling access to objects (Figure 1). In the reference obfuscation, the possibility of a direct reference to an internal object is limited. The concept of Random Access Map can be used to convert direct references to indirect ones or define internal references separately for each user or session [9]. In the access control, access requests to internal objects are checked and denied for unauthorized requests. For the white-box detection of this vulnerability, the same two approaches can be used, as well. That is, the program code is analyzed to check if there are any direct references to internal objects in a system, and in such cases, it is checked if a sound access control routine mediates access requests.

The main challenge to detect this vulnerability is when the application code is missing. In this case, the behavior of the program should be analyzed in response to the requests sent to the program by a tester. The tester must actively test every parameter (whose

value is used to refer to objects) in every request of the application under evaluation. This is a very time-consuming task and the result completely depends on the expertise of the tester in the sense that how to manipulate each parameter to access private objects of another user. Therefore, IDOR cannot be detected in general by traditional security testing tools. Exceptionally, a small subset of IDOR, known as path traversal, which leads to insecure access to files in the host system, e.g., `passwd`, can be detected by existing tools [10]. Few research works have been reported for the black-box detection of IDOR [6, 8, 10, 11]. Vithanage and Jeyamohan [8] presented a method to detect insecure configurations, insecure direct references, and injection vulnerabilities. However, the method's efficacy completely depends on a human agent who identifies suspicious requests and analyzes corresponding responses. Also, the method suffers from false negatives since it confines IDOR vulnerabilities only to the parameters in the URL address of web pages.

Dolati *et al.* [6] presents a method to detect IDOR vulnerabilities. The general approach is to extract exchanged parameters from the traffic between client and server and to apply some rules to the parameters to identify immune or susceptible parameters. The key point is to derive appropriate rules to correctly identify susceptible parameters. The evaluation results of their work confirm its inefficiency in practice, which is mainly due to the poor performance of the defined rules. The rules in their work only pick out four percent of parameters (on average) as non-vulnerable parameters. That is, the remaining 96% of parameters must be manually checked by an expert human. Moreover, this method is just limited to extracting susceptible parameters, and the actual software behavior after manipulating those parameters is not analyzed. Therefore, IDOR vulnerabilities are not actually identified in their work.

Discovery of access control vulnerabilities, which are closely linked to IDOR, has been also the focus of some research, either on white-box [12–16] or on black-box [17–21] approach. The closest work to the aim of this paper is the research conducted by Li *et al.* [17] . They try to extract access control policies, and then to identify violations of the policies by examining the traffic between client and server. One problem with this research is that the authors assume that IDOR occurs for users with different roles while it is possible to have users with the same role having their private objects. Besides, it puts the basis of extracting access control policies on queries sent between the application and the database, which reflects only a part of access control policies. Furthermore, it cannot be considered as a black-box method since it requires the interaction between the program and its database.

Noseevich and Petukhov [21] used state preserving differential analysis to discover access control vulnerabilities in a program. In their method, HTTP requests are first divided into some classes based on their behavior similarity. Then, use-case graphs and their dependencies are obtained regarding the classes of requests and user roles. The main problem with this study, as mentioned by the authors, is its weakness in extracting use-case graphs and their dependencies, which reflect access control policies.

How to manipulate request parameters plays an important role in exploiting IDOR vulnerabilities. The manipulation of parameters has been investigated in some studies [9, 22–26]. For example, Bisht *et al.* [25] propose a black-box method to detect parameters, fit for parameter manipulation. Their method through which the input parameters are automatically generated and injected into the program suffers from both false positives and negatives. This is because their method is not able to generate all possible input values to test the parameter. Moreover, there is not a reliable way in their method to determine whether the program has accepted the input. More importantly, their focus is on considering syntactic limitations to generate parameter values whereas semantic limitations are of more importance for IDOR vulnerabilities.

Table 1 has compared four related works, which are closer to the aim of this paper. As we can see in the table, Li *et al.*'s work [19], needs application-dataVithanage and Jeyamohan's work base interactions in addition to browser-application traffic. This is in contrast with the black-box assumption of this work. Although Dolati *et al.* [10] present a black-box noninvasive method, they only focus on detecting parameters susceptible to IDOR. Moreover, their method is not effective in practice due to lots of false positives. Noseevich and Petokhov's method [23] has false negatives in its detection, related to hidden functionalities that are not in the graphical user interface. It also suffers limitations in the method implementation. Vithanage and Jeyamohan's work [11] efficiently detects IDOR vulnerabilities but it is limited to parameters in URL addresses, which makes lots of false positives in some web applications. More importantly, its effectiveness is directly related to the human agent who has a great responsibility in the detection process (both in attack generation and response analysis). Our approach, compared to the existing works, formalizes the solution, tries to reduce the human intervention in the detection process, and increases the accuracy of detection by reducing false negatives and positives.

3 The Proposed Method

In this section, the proposed solution to detect IDOR vulnerabilities is described.

Table 1. Comparison of black-box IDOR detection methods

	key idea	Implementation/ tool	advantages	disadvantages
[23]	state-based differential analysis, utilizing use case graph	a proof-of-concept aided with human agent to build use case graph	method accuracy	false negatives, implementation limitations
[19]	virtual SQL query concept, role-level and user-level policy inferences	yes	efficient implementation	false negatives, needs Application-Database interactions
[10]	rule-based parameter extraction	no tool	not invasive	considerable false positives, limited effectiveness
[11]	human-aided URL address investigation	yes but highly dependent on human agent	low false positives	limited in URL addresses, high dependency on human agent
our method	targeted manipulation and affected page investigation	yes (aided with human agent in crawling)	method formalization, low rates of false positives/negatives	performance issues in implementation

3.1 Solution Overview

IDOR vulnerability detection in our method is the result of performing the three following phases, performed iteratively:

(1) Information gathering: We improve our understanding of the program by interacting with it as a normal user.
(2) Attack: An illegal request is built and sent to the program
(3) Response analysis: The program response to the attack is evaluated to find out whether it is vulnerable.

Figure 2 depicts the general process of IDOR detection in our method. As illustrated in the figure, the steps are performed iteratively. After each iteration, the information is updated based on the previous attack, and its response is used for triggering the next attack. We elaborate more on the aforementioned phases in the following subsections.

3.2 Information Gathering

In the information gathering phase, the information extracted from user interactions with the application is collected. We assume that in this phase, a non-malicious user normally interacts with the application interface through which access control policies have been correctly implemented. Moreover, we assume that the user performs a complete crawl to generate all possible requests. The more comprehensive we crawl, the more precision we get in our results. In Section 4, we show the effect of the comprehensiveness of the crawling process on the precision of our detection method. It is worth to mention that in this phase, two users for each distinct role should do the crawling process. The result of crawling is a set of sextuples (*user*, *session*, *url*, *request*, *response*, *state*) where:

- *user* identifies the crawling user account.

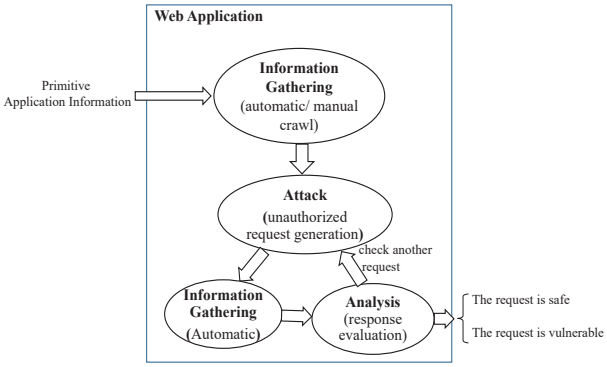

Figure 2. The general process of the proposed method

- *session* identifies the session in which the crawling process is performed. Obviously, when a user logs out from the application the session is terminated and a new session is assigned if he re-logs into the application.
- *url* is the address of the visited page.
- *request* is the request sent by the crawler to the application.
- *response* is the application response to the received request.
- *state* identifies the system state in which the system is when crawling a page.

More precisely, *state* is specified by the database state, which in turn is specified by the data stored on it. That is unless the data stored on the database is not modified (inserted, deleted, or updated), the system state does not change. When a request modifies data stored on the database, the application enters a new state. We model *state* as an incremental integer initialized by zero when we start the crawling process. A database change results in a new state whose value is obtained by adding one to the current state value. Database change is recognized by permanent changes in the pages/forms of the web application.

3.3 Attack Phase

To detect vulnerabilities, it is required to examine the application with unauthorized requests. The attack is performed for pairs of users -an attacker and a victim- who have different access rights according to access control policies. The attacker submits an illegal request to access a victim's private object. The problem is how to tricksily manipulate each parameter to access private objects of another user without knowing the access control logic of the application under evaluation. In this section, we show how to build targeted illegal requests for IDOR detection. The detection process is to examine the application behavior when confronting illegal requests.

Consider A as the attacker and B as the victim. The request is called vulnerable against IDOR if A gets access to the requested resource, which is a B's private object. A request is modeled as a set of parameters and their values.

Let define $Req^u_{pg} = \{req_1, req_2, \ldots, req_n\}$ as the set of authorized requests of user u from page pg. Assume that B visits a page pg and creates a request req^B to the server where $req^B \in Req^B_{pg}$. Let req^A, submitted by A, is the corresponding request to req^B from page pg. Performing the attack is to manipulate req^A to have an illegal request, which may lead to unauthorized access to a B's private object. Manipulating req^A is done by modifying any parameters in req^A. It may also be possible to submit the same request req^B by A.

According to the above explanations, there are two main steps for performing an attack; finding an attacker's request corresponding to a victim's request, and subtly manipulating the request. In the following subsections, we elaborate on the two main steps of performing an attack.

3.3.1 Finding Corresponding Requests

To find corresponding requests between the attacker and the victim users, we first need to find the corresponding pages from which the requests are sent. For this purpose, we use the URL address of pages.

Based on RFC 1738 [27], a URL can be defined as a triplet $url = (scheme, host, path)$. For example, in `http://idor-vul.com/path/to/directory/index.html`, "`http`" is the *scheme*, "`path/to/directory/index.html`" is the *path*, and "`idor-vul.com`" is the *host*. Formally, the *path* in a URL is a set of directories, ended up with a file name as following:

$$path = (dir_1, \ldots, dir_n, file)\ ;\ n \geq 0$$

To find corresponding pages it is required to find the corresponding paths. Let us define corresponding paths, corresponding URLs, and corresponding pages.

Definition 1 (Corresponding paths). *We say $path_i$ and $path_j$ are corresponding and show it by $path_i \equiv path_j$ if and only if the two paths are different in only one of their constituent directories.*

Definition 2 (Corresponding URLs). *We say two URLs url_i and url_j are corresponding and show it by $url_i \equiv url_j$ if and only if the host parts of the URLs are equal and their path parts are corresponding:*

$$url_i \equiv url_j \iff url_i.host = url_j.host \ \wedge \ url_i.path \equiv url_j.path$$

Definition 3 (Corresponding pages). *We say two HTML pages pg_i and pg_j are corresponding and show it by $pg_i \equiv pg_j$ if and only if $url_i \equiv url_j$ where url_i is the URL address of pg_i and url_j is the URL address of pg_j.*

Remark 1. When two users visit the same page, it does not mean that they see the same contents. For example, suppose that each user has access to his setting page. The URL address of the setting page may be the same for all users but the content is different for each user.

Using the information collected in the first phase, i.e., data gathering, and based on the above definitions, we can find the corresponding requests. The next step is to manipulate an attacker's request for its corresponding victim's request.

3.3.2 Manipulating Parameters

Parameter manipulation can be performed by substituting parameter values with random values, which makes it a time-consuming and hard-to-analysis task. Instead, we perform parameter manipulation in such a way that illegal parameter values are knowingly chosen such that the attack is more likely to be successful.

Each request sent to the server consists of several parameters (user-defined or predefined). Formally, a request is defined as follows.

Definition 4 (Request). *A request generated by a user u is a triple of the form req^u = (method, url, P) where method is a constant from the set Method = {GET, POST, DELETE, HEAD, PUT, OPTIONS, CONNECT, TRACE, PATCH} to show the request method, url is the URL address from which the request is sent, and P is the set of request parameters {p_1, p_2, …, p_n} where each p_i ($1 \leq i \leq n$) is a pair of (name, value).*

Based on the above definition, $PV_p^{req^u}$={v_1, …, v_n} is the set of permitted values for parameter p in the request req sent by the user u. In this notation, it is assumed that req is sent from a page pg with

an address *url*. The set of permitted values for each parameter can be extracted from the traffic based on the information collected during the information gathering phase. Specifically, for each parameter in a request *req* from a page with the URL address *url*, the sextuples are extracted for which username is u, address is *url*, and request is *req*.

Definition 5 (Corresponding requests). *Two requests req from user u and req′ from user u′ are corresponding if and only if their method parts be the same and their URLs be corresponding:*

$$req \equiv req' \iff req.url \equiv req'.url \ \wedge \ req.method = req'.method$$

When A attacks B on parameter p in request *req*, a targeted manipulation of A's request is to find an illegal value v for a parameter p where p is a parameter among the set of parameters of *Req*. We show $IV_{A\to B}^{p,req^A}$ as the set of illegal values for p in request *req* when A attacks B. $IV_{A\to B}^{p,req^A}$ is computed by Equation 1. That is, all victim's permitted values can be assigned to p by the attacker.

$$IV_{A\to B}^{p,req^A} = PV_p^{req^B} \tag{1}$$

Example 1. Consider a "send-message" request with the following URL address in which an attacker A with $id = 6$ sends a message to a victim user B with $id = 1$:

```
GET http://www.idor-vul.com/sendmessage?sender_id=6&
receiver_id=1&message_body=some+text+here HTTP/1.1
```

Now, we calculate illegal values for `receiver_id` in this request. If the attacker has already sent messages to users with $id = 1$ and $id = 2$, and the victim has previously sent messages to users with $id = 1$, $id = 2$, and $id = 7$, we have:

$$IV_{A\to B}^{receiver\_id,req^A} = PV_{receiver\_id}^{req^B} = \{1, 2, 7\}.$$

Illegal values can be found for all parameters of all requests using the information gathered in the first phase. If $P = \{p_1, \ldots, p_m\}$ is the set of parameters in req^A, we must find $IV_{A\to B}^{p_i,req^A}$ for $i = \{1, \ldots, m\}$ where B is the victim user. Any subset of parameters can be chosen for manipulation. For example, consider $m = 3$, $x = |IV_{A\to B}^{p_1,req^A}|$, $y = |IV_{A\to B}^{p_2,req^A}|$, and $z = |IV_{A\to B}^{p_3,req^A}|$. There are $x+y+z+xy+xz+yz+xyz = (1+x)(1+y)(1+z) - 1$ different manipulations.

Generalizing this simple example, all possible parameter manipulation of a request can be enumerated by Equation 2:

$$NPM = \prod_{i=1}^{m}(1 + |IV_{A\to B}^{p_i,req^A}|) - 1 \tag{2}$$

NPM in Equation 2 is the number of possible manipulations. Equation Equation 2 says that there are $\binom{m}{i}$ possible parameter selection and for each parameter p selected for manipulation, there are $|IV_{A\to B}^{p,req^A}|$ different manipulations.

3.4 Analysis Phase

In the analysis phase, the application response to the manipulated request is analyzed to find whether the attack is successful. The input to the analysis phase is the illegal request to the server together with its response. Depending on the type of request that can be passive or active, two types of analysis are performed. Active requests change the application state by making a permanent change in data stored on the database, e.g., password change, posting a comment, or removing a file. In contrast, passive requests do not change the application state, e.g., request to view a page (reporting), search a keyword, or view users' comments. An active request triggers an active attack for unauthorized data modification while a passive request triggers a passive attack for unauthorized data observation.

The first step in the analysis phase is to distinguish whether the attack is active or passive. Assume that a user u sends a request *req* from a page *pg* in state α. $AP_\alpha^{req^u}$ is defined as a set of pages affected by *req*:

$$AP_\alpha^{req^u} = \{page | page_\alpha \neq page_\beta, \beta = \alpha + 1\} \tag{3}$$

Equation 3 specifies those pages, which are not equal in the two subsequent states α (before sending the request) and β (after sending the request). $page_\alpha$ and $page_\beta$ in Equation 3 denote the same page *page* in two states α and β, respectively. Therefore, *req* is a passive request if $AP_\alpha^{req^u} = \emptyset$ and vice versa, it is active if $AP_\alpha^{req^u} \neq \emptyset$.

3.4.1 Analysis of Passive Attacks

The key point for the analysis of response to a passive attack is that the attack is successful if the attacker receives the private information of the victim user. The required information for the analysis of passive attacks includes:

(1) Attacker (A) and victim (B) users
(2) req^A, which is the authorized request of A
(3) The illegal request req_t^A (the tampered form of req^A) and its response
(4) req^B, which is the B's request, corresponding to req^A

To analyze a passive attack based on the above inputs, all the victim's requests corresponding to req_t^A are extracted. For example, if the victim user has five "`View Comments`" requests with different parameters,

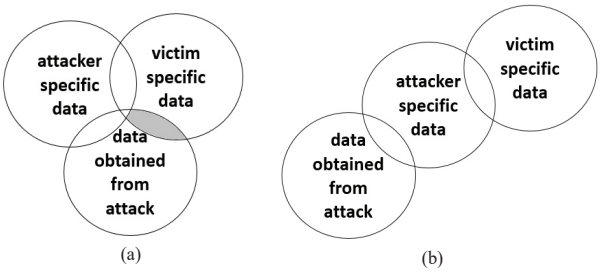

Figure 3. Venn diagram for (a) a successful passive attack (b) an unsuccessful passive attack

all the five requests are extracted and the analysis is performed on them. We show the response of a request req by $resp(req)$. A response has a body containing HTML codes. For simplicity, the response body is referred to by "$page$", so we have:
$page^A = resp(req^A).body$,
$page^B = resp(req^B).body$, and
$page_t^A = resp(req_t^A).body$

Consider the Venn diagram in Figure 3 (a). Each set in the figure represents the data that the application sends in response to a user's request. Specifically, the upper left circle represents the set of data sent by the application to the attacker in response to his legal requests. The upper right circle represents the set of data sent by the application to the victim user in response to his authorized requests. But the lower circle represents the set of data, which the attacker receives in response to his manipulated requests, e.g., a request to see the victim's comments. Considering Figure 3 (a), $page^A$ is the "attacker specific" area, $page^B$ is the victim specific area, and $page_t^A$ is the area specified by "data obtained from attack". The attack is successful when the gray area in Figure 3 (a) is not empty. It means that the attacker, by generating an illegal request and sending it to the application, has obtained data from the victim user that he could not access under normal circumstances. Equation 4 formulates the vulnerable area (VA), which is the gray area in Figure 3:

$$VA = (page^B \cap page_t^A) - (page^B \cap page^A) \quad (4)$$

Figure 3(b) shows a situation where an attacker has sent an illegal request, which has not been led to a successful attack. req^B is not vulnerable against IDOR, if for all attacker's requests corresponding to req^B, the analysis results show that the vulnerable area is empty.

3.4.2 Analysis of Active Attacks

We assume active attacks cause changing the application state and the change is reflected on the content of at least one page in the application. It is also possible for active attacks to have more than one affected pages, which are not necessarily among the response page of the active request. Therefore, active requests, which do not make permanent changes to the application pages are subject to false negatives.

The key point in active attack analysis is to find out to what extent a request of a victim user has been emulated by an attacker. To this purpose, we need to analyze affected pages as well as resulting changes due to the illegal request.

Example 2. Consider the following request in which a message from a user with `id=u6` is sent to another user with `id=u1`:

```
GET http://www.idor-vul.com/sendmessage?sender_id=u6&
receiver_id=u1&message_body=some+text+here HTTP/1.1
```

Now, suppose that the attacker (`id=u6`) generates the following unauthorized request in order to attempt to send a message from the user with `id=7` to the user with `id=u1`:

```
GET http://www.idor-vul.com/sendmessage?sender_id=u7&
receiver_id=u1&message_body=msg+from+7+to+1 HTTP/1.1
```

After sending the above unauthorized request, if the sent message (from `u7` to `u1`) is in the `u1`'s "inbox" page and also in the `u7`'s "sent messages" page, the attack is successful and the "send message" request in the application is vulnerable to IDOR. In this example, we assume that the "inbox" and "sent messages" pages are the only pages affected by a "send message" request.

The general procedure of analyzing active attacks after submitting an illegal request req_t^A is described in five steps:

(1) checking error response to req_t^A
(2) checking affected pages of req_t^A
(3) extracting attacker's requests corresponding to req_t^A as well as victim's requests corresponding to req_t^A
(4) checking affected pages by the above sets of corresponding requests
(5) find similarities between the set of affected pages

Now, each step is explained in the following.

Step 1. After submitting the illegal request req_t^A, its response is examined. If the response contains a web server error (400 and 500 series), the attack is recognized as unsuccessful, so the application is not vulnerable to the request req_t^A. Otherwise, the next step is carried out.

Step 2. The affected pages of the request req_t^A, i.e., $AP^{req_t^A}$, is analyzed. For simplicity, in the notation of $AP^{req_t^A}$, we ignore writing the current state α. By definition, $AP^{req_t^A}$ contains pages, changed after submitting req_t^A. $AP^{req_t^A} = \emptyset$ means that the application has not accepted the unauthorized request req_t^A, so

it is not vulnerable for this request. Otherwise, the analysis must be continued to the next step.

Step 3. The next step is to extract requests, corresponding to req_t^A from the set of attacker's requests. $CReq^A$ in Equation 5 is the set of attacker's requests corresponding to req_t^A. In Example 2, all authorized "send message" requests sent by the attacker to different users are extracted.

$$CReq^A = \{req \mid req \in Req^A \ , \ req \equiv req_t^A\} \quad (5)$$

Subsequently, the requests corresponding to req_t^A from the set of victim's requests are extracted. $CReq^B$ in Equation 6 is the set of such requests. In Example 2, all "send message" requests sent by the victim user to different users are extracted.

$$CReq^B = \{req \mid req \in Req^B \ , \ req \equiv req_t^A\} \quad (6)$$

Step 4. The next step is to obtain the set of affected pages by $CReq^A$ and $CReq^B$, denoted by AP^{CReq^A} and AP^{CReq^B}, respectively. These sets show which pages of the application change after the attacker or victim send requests corresponding to req_t^A.

Step 5. The final step is to find the similarity between $AP^{req_t^A}$ and any member of AP^{CReq^A} and AP^{CReq^B}. $Sim^{req_t^A}_{AP^{CReq^A}}$ is the similarity ratio of req_t^A and AP^{CReq^A}. We calculate the similarity by dividing the number of pages shared between the sets by the total number of pages. Then we have two sets of similarity ratios $Sim^{req_t^A}_{AP^{CReq^A}}$ and $Sim^{req_t^A}_{AP^{CReq^B}}$. We choose the maximum value of each set, namely $MaxSim^A$ and $MaxSim^B$. $MaxSim^A$ shows the maximum similarity between the response of the attacker's manipulated request and the response of the attacker's authorized requests. Similarly, $MaxSim^B$ shows the maximum similarity between the response of the attacker's manipulated request and the response of the victim's requests. $MaxSim^A > MaxSim^B$ implies that the application behavior when confronting the unauthorized request req_t^A is more similar to its behavior when confronting the authorized request of the attacker, which in turn implies that the application is less likely to be vulnerable. On the other hand, $MaxSim^A < MaxSim^B$ means that the application, confronting an illegal request req_t^A, behaves more similar to a request originated from a victim user, so it is more likely to be vulnerable against IDOR. If the two values are equal, we could not infer about the vulnerability and analytical evidence should be increased. For example, by manipulating the parameters in req^A, a new req_t^A can be made (Different from the previous one), and then by performing the above analysis, reliable results may be inferred.

Examples 3 and 4 exemplify the process of active attack analysis. For the sample "send message" request, Example 3 shows the concept of affected pages and Example 4 shows their analysis.

Example 3. Considering Example 2, assume that the victim `u7` sends messages through the "send message" request to three users `u1`, `u2`, and `u3`, and the attacker `u6` sends messages to `u1` and `u5`. Assuming *inbox* and *sent* pages are the only pages changed after a "send message" request, then AP^{CReq^A} and AP^{CReq^B} are as follows:

$$AP^{CReq^A} = \{\{inbox_{u1}, sent_{u6}\}, \{inbox_{u5}, sent_{u6}\}\}$$
$$AP^{CReq^B} = \{\{inbox_{u1}, sent_{u7}\}, \{inbox_{u2}, sent_{u7}\}, \{inbox_{u3}, sent_{u7}\}\}$$

Example 4. Consider Example 2 and Example 3. According to Example 2, $AP^{req_t^A} = \{inbox_{u1}, sent_{u7}\}$. Then, the similarity ratios $Sim^{req_t^A}_{AP^{CReq^A}}$ and $Sim^{req_t^A}_{AP^{CReq^B}}$ are calculated as follows:

$$Sim^{req_t^A}_{AP^{CReq^A}} = \{\frac{1}{2}, \frac{0}{2}\} \ , \quad Sim^{req_t^A}_{AP^{CReq^B}} = \{\frac{2}{2}, \frac{1}{2}, \frac{1}{2}\}$$

For the maximum similarity ratios, we have $MaxSim^A = 0.5$ and $MaxSim^B = 1$. That it, $MaxSim^B > MaxSim^A$. It implies that the application, confronting an illegal "sent message" request by A, behaves more similar to the response of a "sent message" request originated from the victim user (B). Therefore, the "send message" request is supposed to be vulnerable against IDOR.

4 Implementation and Evaluation

We have implemented our method as an IDOR detection tool, called IDOT [1]. In this section, we briefly review some implementation issues we encountered while developing IDOT. Then, we report the evaluation results of applying our method on two web applications. For the sake of simplicity, in the current version of IDOT, we focus only on GET and POST methods in requests. The tool can be easily extended to adopt other method types of a request, mentioned in Section 3.3.2.

4.1 Implementation Issues

IDOT architecturally consists of three main modules, as depicted in Figure 4. The crawler module crawls the application and gathers information. The attack module tries to attack the application by generating illegal requests. The analysis module analyzes the results of an attack to find out whether the application is vulnerable. IDOT outputs a report of vulnerable

[1] IDOR detection tool

requests together with their attack payloads. IDOT has been implemented in about 4000 lines of code in Python programming language (version 3) and SQLite as the database management system. A human agent can interact with IDOT in a command-line environment.

4.1.1 The Crawler Module

To do the automatic part of crawling, we use a framework called Scrapy. Scrapy provides APIs to help programmers for using the framework and sending their requests. We use some libraries such as *requests* for sending requests and *beautiful soup* for parsing HTML pages.

To perform crawling, at first, a text file containing some primary settings to run Scrapy, e.g., usernames and passwords to login to the application, is inputted to IDOT. IDOT then starts a full scan of the web application by Scrapy using the provided input. A human agent can help the crawler module and manually crawls those pages, which cannot be crawled automatically. The information obtained from the crawling by the human agent is added to the information of the automatic crawling. After completing this step, the crawling process is performed again (for the second time) for the same user in another session. That is, the crawling process is performed for each user in two distinct sessions. We also crawl the web application as the guest user (without username and password), so he is usually limited in using most of the application capabilities.

The output of the information gathering phase consists of the sextuples, described in Section 3.2 and stored on a database table.

4.1.2 The Attack Module

The attack module uses the information previously gathered by the crawler module. The information is processed to subtly manipulate data in a user's request such that the request leads to unauthorized access to a private object of another user. In short, this module is responsible to generate deliberate illegal requests by manipulating their parameters.

4.1.3 The Analysis Module

The main part of the analysis module is to find the affected pages of an illegal request. Affected pages are those pages whose contents have changed after sending illegal requests. To find affected pages, an exhaustive search is done within all pages of all users using the information gathered in the crawling process. Any changes in a page compared to the last version of the page, stored in the database, indicates that the page has been affected by the request. There are also some false positives in detecting affected pages due to some constantly changing parameters in a page such as anti-CSRF tokens and the current time. We considerably reduce such false positives in our implementation by retrieving a page at two different times without change to the application state.

In the analysis of passive attacks, it is required to do intersection and subtraction on two HTML pages. For this purpose, we use the tree-based DOM structure of HTML pages. In such a structure, those parts of an HTML page that are visible to a user are generally placed in leaf nodes. Therefore, leaves contain page content and internal nodes contain HTML tags. By using subtraction and intersection operations, based on Equation 4, the vulnerable area can be obtained. Let us exemplify the intersection and subtraction operations over two HTML pages.

Example 5. Consider a simple page like the one in Figure 5 for two users `Arash` and `Ramin`. In these pages, the first paragraph is a shared text between the two users. The second paragraph of each page is a dedicated text specific to each user. There is also a form on the page with three input elements: a submit button, a text input, and a drop-down list having two common options for `Arash` and `Ramin` and a private option specific to each user. Figure 6 is the tree structure of HTML code of Figure 5. IDOT constructs a similar tree structure except adding some metadata for the nodes of the tree. For the sake of simplicity, the metadata is not shown in Figure 6. The colored part is the difference between the content of the two pages of `Arash` and `Ramin`. To subtract Figure 6 (b) from Figure 6 (a), that is $a - b$, all identical leaves are removed from a together with their identical parents until the root. Figure 7 (a) shows the subtraction result.

To intersect between Figure 6 (b) and Figure 6 (a), that is $a \cap b$, all leaves in a without an equivalent leaf in b (the colored parts in Figure 6) are removed together with their parents until the root. The result of the intersection is shown in Figure 7 (b).

4.2 Experimental Evaluation

In this section, we report the evaluation results of our method to detect IDOR vulnerabilities in two web applications, as case studies. Since IDOR is a logical vulnerability, it can only be detected by manual testing and existing vulnerability scanners are not yet able to detect it. Therefore, we cannot compare IDOT with exiting vulnerability scanners. Instead, we compare the results of IDOT with expected results obtained by manual tests of a software security expert. We examine two web applications, namely RailsGoat and

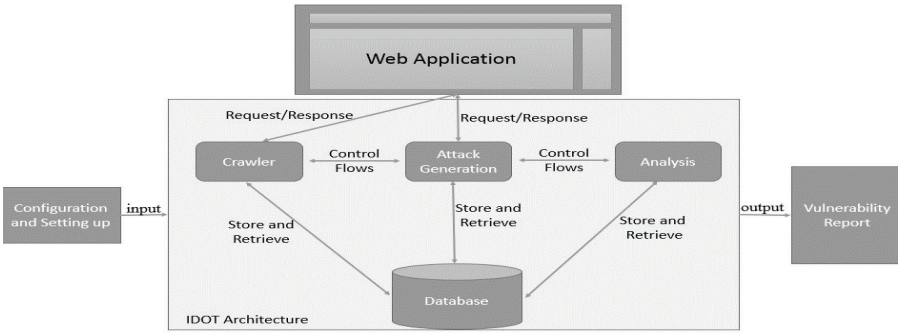

Figure 4. The architecture of IDOT

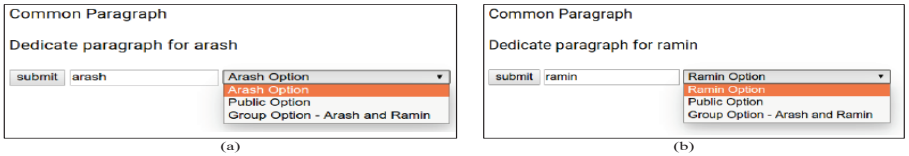

Figure 5. The pages for user `Arash` (a) and user `Ramin` (b)

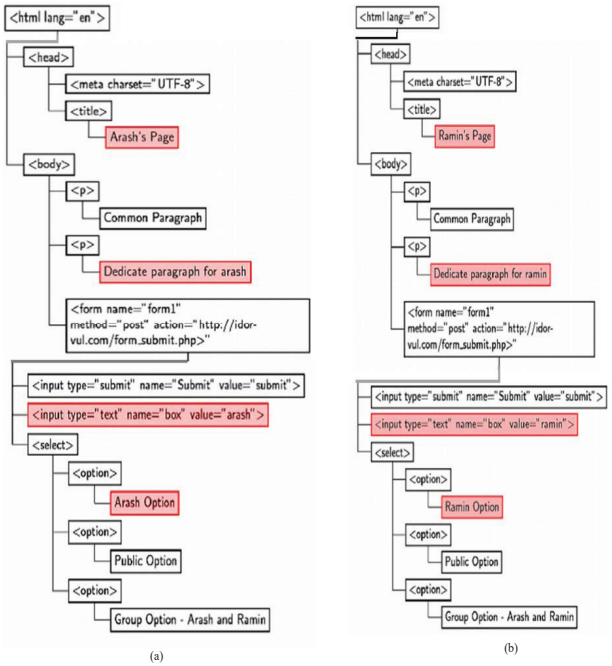

Figure 6. Trees corresponding to (a) `Arash`'s page and (b) `Ramin`'s page

GetBoo. They are publicly available and chosen from OWASP *Broken Web Applications (BWA)*[2] project. OWASP BWA consists of vulnerable web applications offered as virtual machines. We chose RailsGoat and GetBoo since they are vulnerable to IDOR and the vulnerabilities are publicly known and verifiable. For certainty, the IDOR vulnerabilities were detected and verified by an expert software security tester.

[2] `https://www.owasp.org/index.php/OWASP_Broken_Web_Applications_Project`, [visited: August 2019]

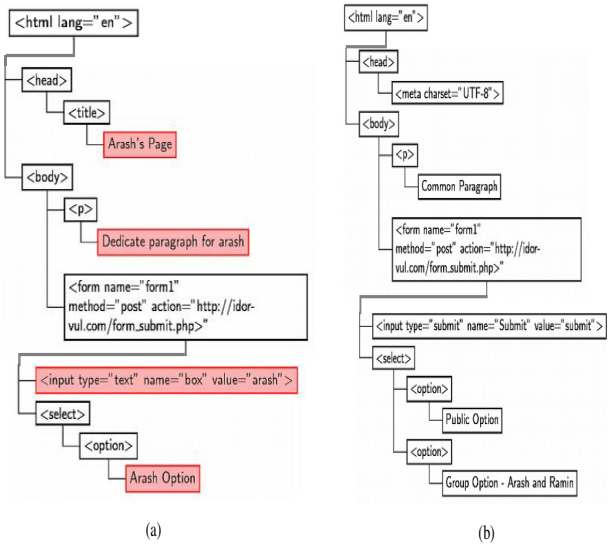

Figure 7. The tree obtained by (a) a - b and (b) $a \cap b$

For the valuation of GetBoo and RailsGoat, IDOT together with XAMPP was installed on a virtual machine having Ubuntu (version 16.04) operating system with about 3GB RAM and two processing cores. we first perform a crawling process in separate sessions for each role using the Scrapy framework as described in Section 4.1.1. Then, the information gathered in the crawling phase is used for the attack and analysis phases by IDOT. The evaluation time for each of our case studies takes about 30 minutes, which is not short for such small applications. However, the attack and analysis time can be reduced by manual intervention in choosing a subset of parameters for manipulation (instead of trying all possible parameter manipulations). Also, since the bottleneck is pro-

cessing, the time decreases in case there exist more processing cores.

RailsGoat was evaluated as our first case study. In this web application, users can view and manage their incomes and send messages to other users. We crawled RailsGoat using three registered users as well as a guest user. Our investigation shows that there are 29 passive and eight active requests in this application. IDOT, after attack and analysis, detected three vulnerable passive requests and two vulnerable active requests. The test performed by the software security expert showed that two passive requests among 29 and two active ones among eight requests are vulnerable. Comparing IDOT results with the expected results obtained by manual tests indicates that for the case of RailGoat IDOT has one false positive (in passive requests) and there is no false negative. Table 2 summarizes the results of testing RailsGoat by IDOT.

Table 2. Test results of RailGoat

	Passive Requests	Active Requests	Total
Number of requests	29	8	37
Number of *Expected* vulnerable requests	2	2	4
Number of *Detected* vulnerable requests	3	2	5

GetBoo is our second case study. It is a web application for managing user bookmarks including 35 passive and 15 active requests. Running IDOT on Getboo resulted in detecting only a vulnerable passive request. The vulnerable request opens the related link in a stored bookmark. For example, assume u_1 has stored a bookmark b with an access link `http://getboo.com/redirect.php?id=9`. Now, if u_2 uses this link in his request, he can get access to u_1's bookmark. No other vulnerable request is detected by IDOT. The result obtained through manual tests performed by the human expert also confirms that no other IDOR vulnerability exists in this application. Table 3 summarizes the results of testing GetBoo by IDOT.

Table 3. Test results of GetBoo

	Passive Requests	Active Requests	Total
Number of requests	35	15	50
Number of *Expected* vulnerable requests	1	0	1
Number of *Detected* vulnerable requests	1	0	1

5 Conclusion and Future Work

IDOR is a logical vulnerability exploited by manipulating request parameters to get unauthorized direct access to an object and violating access control policies. The logical nature of IDOR makes its black-box detection a challenging problem. Since existing vulnerability scanners cannot understand access control policies of web applications and the policies are different for each application, they are not able to detect such vulnerabilities. Considering this gap, we proposed a black-box method to detect IDOR without being aware of the access control logic of web applications. We implemented our method as an IDOR detection tool, called IDOT, and experimentally evaluated its effectiveness using a couple of case studies. The evaluation results show that IDOT detects IDOR vulnerabilities with low rates of false-positive and negative.

The current version of IDOT has some limitations. For example, it neglects file upload requests, which may lead to false-negative if there exist such requests in an application. Moreover, it only supports HTML and CSS codes. While our method is not conceptually limited to these technologies, we plan to improve IDOT implementation to support technologies such as Ajax and JavaScript to decrease the false-negative rate of detection.

We also plan to extend IDOT in terms of time efficiency. Now, it takes a long time to evaluate an application, most of it pertains to generating different forms of manipulated requests and also finding affected pages of a request.

References

[1] G. Deepa and P. Santhi Thilagam. Securing Web Applications From Injection and Logic Vulnerabilities: Approaches and Challenges. *Information and Software Technology*, 74(C):160–180, jun 2016.

[2] Melina Kulenovic and Dzenana Donko. A Survey of Static Code Analysis Methods for Security Vulnerabilities Detection. In *37th International Convention on Information and Communication Technology, Electronics and Microelectronics*, pages 1381–1386, July 2014.

[3] Jun Li, Bodong Zhao, and Chao Zhang. Fuzzing: a Survey. *CyberSecurity*, 1(6):1–13, 2018.

[4] OWASP top 10 - 2017: The ten most critical web application security risks, version 4.2. available at: https://owasp.org/www-project-top-ten/ visited: March 2021.

[5] Pontus Thulin. Evaluation of the Applicability of Security Testing Techniques in Continuous Integration Environments. Master's thesis, Jan 2015.

[6] Vahid Dolati, Mohammad Ali Hadavi, and Hasan Mokhtari Sangchi. A Method for Black-box Detection of Insecure Direct Object Reference Vulnerability. In *21th Annual National Conference of Computer Society of Iran*, Mar 2016. in Persian.

[7] Elie Saad and Rick Mitchell. OWASP web security testing guide, version 4.2, 2020.

[8] Nisal Madhushan Vithanage and Neera Jeyamohan. WebGuardia - An Integrated Penetration Testing System to Detect Web Application Vulnerabilities. In *International Conference on Wireless Communications, Signal Processing and Networking (WiSPNET)*, pages 221–227. IEEE, March 2016.

[9] Hasty Atashzar, Atefeh Torkaman, Marjan Bahrololum, and Mohammad H. Tadayon. A Survey on Web Application Vulnerabilities and Countermeasures. In *6th International Conference on Computer Sciences and Convergence Information Technology*, pages 647–652, Nov 2011.

[10] Vanja Suhina. Exploiting and Automated Detection of Vulnerabilities in Web Applications. Technical report, Department of Electronics, Microelectronics, Computer and Intelligent Systems, Faculty of Electrical Engineering and Computing, University of Zagreb, 2007.

[11] Ajay Kumar. Shrestha, Pradip Singh Maharjan, and Santosh Paudel. Identification and Illustration of Insecure Direct Object References and Their Countermeasures. *International Journal of Computer Applications*, 114(18):39–44, 2015.

[12] Francois Gauthier and Ettore Merlo. Fast Detection of Access Control Vulnerabilities in PHP Applications. In *19th Working Conference on Reverse Engineering*, pages 247–256. IEEE, Oct 2012.

[13] Fangqi Sun, Liang Xu, and Zhendong Su. Static Detection of Access Control Vulnerabilities in Web Applications. In *Proceedings of the 20th USENIX Conference on Security*, SEC'11, pages 11–11. USENIX Association, Aug 2011.

[14] Sooel Son, Kathryn S. McKinley, and Vitaly Shmatikov. Fix Me Up: Repairing Access-Control Bugs in Web Applications. In *20th Annual Network and Distributed System Security Symposium, NDSS 2013*, pages 1–16. IEEE, Feb 2013.

[15] Viktoria Felmetsger, Ludovico Cavedon, Christopher Kruegel, and Giovanni Vigna. Toward Automated Detection of Logic Vulnerabilities in Web Applications. In *Proceedings of the 19th USENIX Conference on Security*, pages 10–10. USENIX Association, Aug 2010.

[16] Michael Dalton, Christos Kozyrakis, and Nickolai Zeldovich. Nemesis: Preventing Authentication & Access Control Vulnerabilities in Web Applications. In *18th USENIX Security Symposium*, USENIX Security'00, pages 267–282. USENIX Association, Aug 2009.

[17] Xiaowei Li, Xujie Si, and Yuan Xue. 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*, CODASPY '14, pages 49–60. ACM, 2014.

[18] Maliheh Monshizadeh, Prasad Naldurg, and V.N. Venkatakrishnan. Mace: Detecting Privilege Escalation Vulnerabilities in Web Applications. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS 2014, pages 690–701. ACM, 2014.

[19] Xiaowei Li, Wei Yan, and Yuan Xue. SENTINEL: Securing Database from Logic Flaws in Web Applications. In *Proceedings of the Second ACM Conference on Data and Application Security and Privacy*, CODASPY '12, pages 25–36. ACM, Feb 2012.

[20] Xiaowei Li and Yuan Xue. BLOCK: A Black-box Approach for Detection of State Violation Attacks Towards Web Applications. In *Proceedings of the 27th Annual Computer Security Applications Conference*, ACSAC '11, pages 247–256. ACM, 2011.

[21] George Noseevich and Andrew Petukhov. Detecting Insufficient Access Control in Web Applications. In *First SysSec Workshop*, pages 11–18, jul 2011.

[22] Muath Alkhalaf, Shauvik Roy Choudhary, Mattia Fazzini, Tevfik Ultan, Alessandro Orso, and Christopher Kruegel. 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*, ISSTA 2012, pages 56–66. ACM, 2012.

[23] Marco Balduzzi, Carmen Torrano Gimenez, Davide Balzarotti, and Engin Kirda. Automated Discovery of Parameter Pollution Vulnerabilities in Web Applications. In *Proceedings of the 18th Network and Distributed System Security Symposium*. IEEE, feb 2011.

[24] Nazari Skrupsky, Prithvi Bisht, Timothy Hinrichs, VN Venkatakrishnan, and Lenore Zuck. 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*, CODASPY '13, pages 129–140. ACM, 2013.

[25] Prithvi Bisht, Timothy Hinrichs, Nazari Skrupsky, Radoslaw Bobrowicz, and VN Venkatakrishnan. Notamper: Automatic blackbox detection of parameter tampering opportunities in web applications. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, CCS '10, pages 607–618. ACM, 2010.

[26] Xiaowei Li and Yuan Xue. Logicscope: Automatic discovery of logic vulnerabilities within web applications. In *Proceedings of the 8th ACM*

SIGSAC Symposium on Information, Computer and Communications Security, ASIA CCS '13, pages 481–486. ACM, 2013.

[27] Berners T. Lee, L. Masinter, and M. Mccahill. RFC 1738: Uniform resource locator (URL), 1994.

Mohammad Ali Hadavi received his Ph.D. degree in Computer Engineering from Sharif University of Technology, Tehran, Iran, in 2015. He received his M.Sc. and B.Sc. degrees in Software Engineering from Amirkabir University of Technology, Tehran, Iran, in 2004, and from Ferdowsi University of Mashhad, Iran, in 2002, respectively. Now, he is an assistant professor at Malek Ashtar University of Technology. Focusing on information security, he has published more than 30 papers in national and international journals and conference proceedings. His research interests include software security, database security, and security aspects of data outsourcing.

Arash Bagherdaei was born in 1991. He received his B.Sc. degree in Information Technology from Hamedan University of Technology, Hamedan, Iran. He then received his M.Sc. degree in Secure Computing from Malek Ashtar University of Technology, Tehran, Iran in 2017. Now, he is working on developing embedded systems in the private sector.

Simin Ghasemi was born in Zanjan, Iran in 1987. She received her B.Sc. in 2010 from Institute for Advanced Studies in Basic Sciences (IASBS), and M.Sc. from Sharif University of Technology on data security in 2012. She is already a faculty member of Payam Noor University of Zanjan. Her research interests include data and software security, database outsourcing, and trust based models.