Journal of Computing and Security

Research Article, 2021, Volume 8, Number 1 (pp. 19–32)

http://www.jcomsec.org

Automatic Black-Box Detection of Resistance Against CSRF Vulnerabilities in Web Applications

Samira Sadeghi [a], Mohammad Ali Hadavi [a,*]

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

ARTICLE INFO.

Article history:
Received: 2 February 2020
Revised: 21 April 2021
Accepted: 18 April 2021
Published Online: 9 July 2021

Keywords:
Web Security, Vulnerability Detection, Cross-Site Request Forgery (CSRF), Anti-CSRF Token, Traffic Analysis.

ABSTRACT

Cross-Site Request Forgery (CSRF) is an attack in which an infected website causes a victim's browser to perform an unwanted operation on a trusted website. The main solution to tackle this attack is to use random tokens in requests, sent by the browser. Since such tokens cannot be guessed or rebuilt by the attacker, he is not able to forge the requests. The tokens can be specific to a request, a page, or a session. Existing methods for detecting CSRF vulnerabilities mainly rely on simulating an attack by manipulating a request, submitting it to the server, and analysis of the response to the forged request. This kind of test must be repeated for each request in a web application to identify whether the application is vulnerable. Moreover, it may lead to undesired changes to the application database by submitting fake requests. This paper presents a method to passively detect CSRF-resistant requests by analyzing the traffic to the target website. To this end, we formulate a set of rules to analyze the possible existence of anti-CSRF tokens. Traffic analysis based on the proposed rules outputs resistant requests due to the use of random tokens. Consequently, the requests without such tokens are deduced to be potentially vulnerable. The proposed method is implemented and evaluated by the traffic extracted from several websites. The results confirm that the method can effectively detect anti-CSRF tokens in requests and the more complete the website traffic, the more accurate the results.

1 Introduction

Vulnerability is a defect and weakness in the design, implementation, or operation of an application that can violate security policies [1]. One of the most significant web vulnerabilities is Cross-Site Request Forgery (CSRF.(Since many websites cannot protect themselves against the attack, CSRF is called "sleeping ghosts" among the vulnerabilities in the web [2]. The attack exploits the trust that a server has upon its user/browser. That is, a malicious website sends a request to a web application through the browser of the victim user, who has been previously authenticated by the application. In this way, the attacker performs malicious actions using the victim's user access level.

The existing methods to detect this vulnerability are mainly white-box in which the application code is checked to see if preventive measures have been implemented during programming [3] [2] [4]. Some

* Corresponding author.
Email addresses: s.sadeghi@mut.ac.ir (S. Sadeghi), hadavi@mut.ac.ir (M. A. Hadavi)
https://dx.doi.org/10.22108/JCS.2021.127261.1064
ISSN: 2322-4460

other methods detect CSRF by adding or changing parameters in the requests sent to the server [2, 4–7]. These methods examine the reaction of the application to the forged requests. The existing methods have at least one of the following limitations:

(1) The detection method is white-box. That is, it needs source code or at least some information about the page structure, the content of the tags, etc.
(2) Some methods are not automatic. That is, the methods should be manually carried out for each request of the web application, separately.
(3) The detection method behaves actively. That is, it makes unauthorized changes to the application database to check the application behavior against an intentional forged request.

Considering the above limitations and extending the idea in [8], we propose a method to detect CSRF vulnerabilities in web applications without access to the source code and any changes in requests (no changes in the application database). The non-invasive nature of our methods results in finding *potential points of vulnerability* by focusing on the most prevalent CSRF countermeasure, i.e., anti-CSRF random tokens. For this purpose, we define a set of rules to explore the existence of anti-CSRF tokens in traffic. Traffic analysis concerning the suggested rules determines whether a request is potentially safe or vulnerable. We have implemented our method and evaluated it on several websites.

This paper is structured into five sections. In Section 2, we briefly review the CSRF attack and generic known methods to address it. Related research on CSRF detection is also reviewed in this section. Section 3 describes the proposed method by describing four sets of rules. In Section 4, we report the results of our implementation and evaluation on several websites as case studies. Finally, we conclude the paper along with some directions for future work in Section 5.

2 CSRF Attack and Countermeasures

The general structure of HTTP messages includes requests sent from a client to a server and the server responses to the client. In the HTTP protocol, when a user logs into a web application using his credentials, e.g., his username and password, the application authenticates the user. In case of successful authentication, the server allows the user to login into the application and sends cookies to the client browser to maintain the user status. The client browser sends the cookies back to the server in subsequent user requests so that the server identifies the client and handles the requests without user re-authentication. CSRF attacks misuse this functionality to perform a forged request on behalf of an authenticated user. In the attack, an attacker creates a forged request and submits the request through the browser of an authenticated user, who is the victim in this scenario. Since the browser is not able to detect that the request is forged, it is accompanied by appropriate cookies and executed at the server-side [2]. The implementation scenario for a CSRF attack is based on six steps, presented in Figure 1:

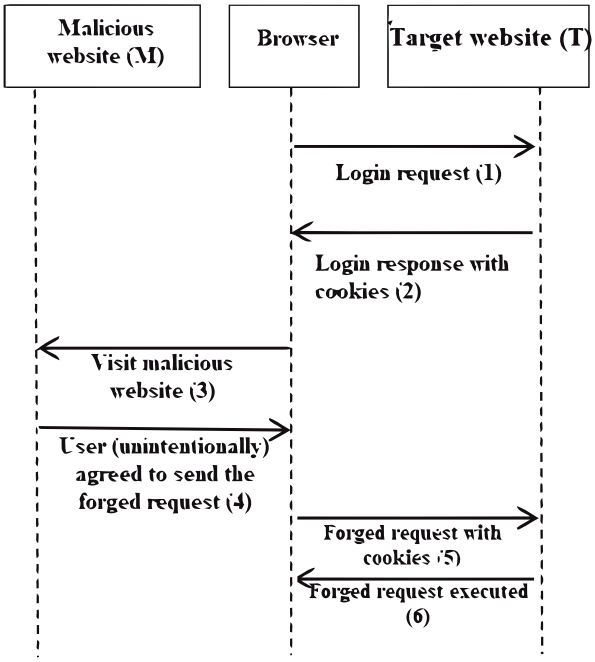

Figure 1. CSRF Attack Scenario [9].

(1) The victim user opens the target website (T) in a browser and sends a login request to the website.
(2) The target website (T) sends the response to the login request along with a cookie.
(3) The victim user visits a malicious website (M).
(4) On the website (M), there is malicious content (such as JavaScript code, an img tag, ...) that includes the forged request. The victim user is tempted to send the forged request from his browser.
(5) As a result, the victim browser puts the cookie, received from the website (T) in the request header.
(6) The website (T), after receiving the request, extracts the cookie, verifies it, and executes the request on the victim's account without the user being aware of it.

In the above scenario, the attacker is not able to forge a request if the request includes a parameter whose value is randomly (unpredictably) assigned by

the server and is known by the client. The random value is further verified by the server when the client submits the request. Now, the attacker cannot build a valid request without having the random value for the request. Such random valued parameters in requests, called anti-CSRF tokens, are the most effective way to prevent CSRF attacks. There are different ways to implement anti-CSRF tokens in a web application.

In addition to anti-CSRF tokens, other countermeasures have been proposed for CSRF attacks such as checking the HTTP referer and limiting cookies' lifetime:

(1) Checking HTTP referer: The referer field in HTTP requests indicates the URL of the website, where the request has been submitted from. This is used to check the domain of the source website before executing the request on the server. If the domain of the source website is not compatible with the specified domain, the request is detected as a suspected request [1]. This countermeasure is useful only if there are limited, and more importantly, previously known web URLs, from which the requests to the target website can be originated.
(2) Limiting cookies' lifetime: By limiting the lifetime of cookies, we decrease the risk of CSRF attacks by reducing the attack success probability [10], [1]. In cases where a fake request is made a long time after the second step in Figure 1, the cookie has been expired and the attack is stopped at its fifth step. Indeed, this countermeasure can only prevent the attacks triggered after the specified cookies' lifetime.

Since random tokens are considered the most common and effective mechanism to prevent CSRF attacks, we focus on this mechanism and detect CSRF vulnerability by analyzing web traffic to find out if such tokens are used in requests/responses.

2.1 Research on CSRF Detection

CSRF prevention has been the aim of some research, which presents solutions such as CSRFGuard [5], ESCUDO [6], Mutual Approval [11], App Isolation [12], and SessInt [13], to impede the vulnerability from occurring. However, since the focus of this paper is to detect CSRF, in the following, we only review research related to CSRF detection. A survey of CSRF prevention methods can be found in Calzavara et al. [14].

Shahriar and Zolkernine [3] implemented a plug-in to detect CSRF vulnerability based on the two concepts of visibility and content type. Visibility is related to parameters and their values in requests. It means that a request is from an open page in the browser, and parameters in the request are from the page forms and tags. The content-type points to the fact that the content of HTML tags contained in a request must be the same as the expected content type of the tags. This plug-in executes CSRF detection at the client-side. Therefore, it has a negative impact on the speed and performance of the browser. Moreover, this method requires comparing the content type of the response with the expected content type. That is, it needs to be aware of the form on the page, which is a kind of dependency on the page structure. In other words, the detection method follows a white-box approach.

OWASP has developed a tool, namely CSRFTester, to detect CSRF vulnerabilities [7]. With CSRFTester, we can save a transaction performed by an authenticated user. Then, manipulate the request parameters in the transaction, and finally resend the transaction to execute at the server-side. If the request is successfully executed, we conclude that the request is vulnerable to CSRF. This approach may cause some undesirable changes to the system database. Also, CSRFTester only uses two sessions to detect vulnerabilities, and having multiple sessions does not improve the detection rate. Moreover, with CSRFTester we must manually check each request separately, which is a time-consuming task for a large web application.

Rocchetto et al. [4] proposed a model-based method using *ASLan++* language to detect CSRF vulnerabilities. Their method requires design phase specifications of client-server interactions to model a web application. Their method outputs whether the specification is safe under several assumptions and known protection mechanisms. If it detects a CSRF vulnerability, an abstract attack trace to exploit the vulnerability is reported. This means that their method is not black-box, and is more useful to check the effectiveness of design phase decisions. Moreover, the detection accuracy entirely depends on the level of details in specifications of client-server interactions.

Deemon [15] is another tool designed to detect CSRF vulnerabilities with dynamic analysis, using program status changes, data flow patterns, and graph properties. Deemon cannot be used to test a web application in an operational environment since it results in unauthorized database changes.

Srokosz et al. [16] attempted to detect CSRF attacks using the user's history of actions. An action requested by a user is valid if it is similar to the existing user's history of actions. Otherwise, the request is considered suspicious and for that, additional authorization is required. The authors propose an algorithm to compute the similarity between an action and a history of actions. The accuracy of this approach depends on the

history of actions. New authorized actions, which are not in history, are supposed as suspicious actions and require additional qualifications. On the other hand, one previously performed action can be misused by an attacker without being detected. This approach has more senses to be used as a prevention rather than a detection method.

Calzavara et al. [17] proposed a machine learning-based solution to detect CSRF vulnerabilities. Their solution to this purpose is called Mitch [17] [18]. Using Mitch, they simulate CSRF attacks on sensitive requests, which may lead to inappropriate changes to the application database due to its invasive nature. As another limitation, this tool assumes to know sensitive HTTP requests for which CSRF vulnerability is examined. To collect sensitive requests, a browser extension has been developed to manually label the requests as sensitive or insensitive. Moreover, the detection results suffer from false negatives.

Requiring access to the program source code and the active behavior of the detection method, which leads to unauthorized changes to the program database, are among the main limitations of the existing methods. Regarding these limitations, we propose a black-box CSRF detection method in the next section.

3 The Proposed Method

The most common defense mechanism against CSRF is using anti-CSRF tokens in requests. This mechanism is effective to prevent the attack since such tokens cannot be guessed or regenerated by the attacker, who does not have access to the client machine through which the request is submitted. In this paper, we use this feature and try to find anti-CSRF tokens in the traffic. We explore traffic to find some evidence, showing whether anti-CSRF tokens have been used in requests. Those requests, for which no token is detected, are identified as vulnerable requests.

Figure 2 shows an architectural view of our method, consisting of three main components:

(1) The parameter extraction component to extract all parameters sent in requests in the input traffic
(2) A rule base consisting of a set of rules to detect anti-CSRF tokens from the extracted parameters
(3) The detection engine that outputs two sets; a set of requests empowered with anti-CSRF tokens and a set of potentially vulnerable requests

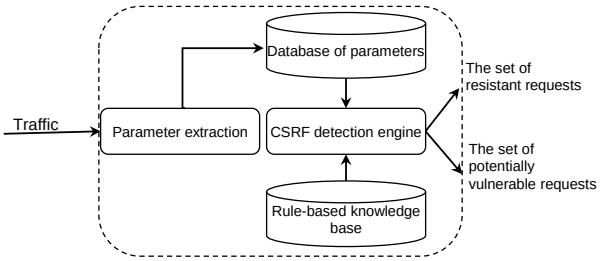

Figure 2. The Architecture of the Proposed CSRF Detection System.

3.1 Definitions

We first define the terms used in this paper and then, describe the proposed method based on the definitions.

Definition 1 (Parameter). Parameter p is a value in a request body sent from a client to a server. We model the parameter by a triple (n, v, u) in which:

(1) n is the parameter name,
(2) v is the value of the parameter, and
(3) u is a Boolean value indicating whether v is assigned by a user. $u = 0$ if v is assigned by a user, otherwise $u = 1$.

Given Definition 1, we use $p.v$ for the value and $p.n$ for the name of parameter p. Similarly, $p.u$ shows whether $p.v$ has been assigned by a user.

Definition 2 (Request). Each request r sent from a client to a server is modeled with $r = (url, P)$ in which:

(1) url is the address of the page, where the request has been sent from.
(2) $P = \{p_1, p_2, p_3, \ldots, p_n\}$ is the set of parameters in r, where each parameter $p_i \in P$ is defined as in Definition 1.

Based on Definition 2, we use $r.url$ to show the url of request r, and $r.P$ to show the set of parameters P in r. Also, $Req = \{r_1, r_2, r_3, \ldots, r_m\}$ represents the set of requests sent from the client to the server.

Definition 3 (Request). The referrer in a request r is the URL address of the page from which r has been sent. We say $ref\ (r) = url$ when r is sent from url.

Definition 4 (Response). Response res contains a set of elements sent from the server to a client in response to a request r. We say $res(r) = (url, E)$ where

(1) url represents the address of page res and
(2) $E = \{e_1, e_2, \ldots, e_k\}$ is the set of elements e_i in the response body. Each entity $e_i \in E$ is a triple (n, v, t) in which:
   (a) n is the element name,
   (b) v is the element value, and

(c) t is a Boolean value. $t = 1$ if the element is hidden, otherwise $t = 0$.

In our notations, *res(r).e.n* denotes the name of the element e in the response page *res* corresponding to request r. Similarly, *res(r).e.t* specifies whether the element type is hidden, and *res(r).url* indicates the URL address of the response page *res*.

Definition 5 (Session). Each session s is a set of requests and corresponding responses as defined in (1):

$$s = \{(r_1, res(r_1)), (r_2, res(r_2)), \ldots, (r_l, res(r_l))\} \quad (1)$$

Definition 6 (Traffic). Traffic T is a set of sessions as defined in (2):

$$T = \{s_1, s_2, s_3, \ldots, s_p\} \quad (2)$$

Based on the above definitions, we define some notations used further in this paper:

(1) Req^s as defined in Equation 3 is the set of all requests in a session s.

$$Req^s = \bigcup_{(r,res(r))\in s} r \quad (3)$$

(2) $s.P$, as defined in (4), is the set of parameters in a session s.

$$s.P = \bigcup_{(r,res(r))\in s} r.P \quad (4)$$

(3) $T.P$, as defined in (5), is the set of parameters in traffic T.

$$T.P = \bigcup_{s\in T} s.P \quad (5)$$

3.2 Detection Rules

The basis of our method is to detect anti-CSRF tokens in request parameters. For this purpose, based on the characteristics of random tokens, we have formulated some rules as the basis for recognizing these parameters. The rules according to their function are divided into four categories:

(1) The primary rule: The primary rule involves those characteristics that an anti-CSRF token necessarily has.
(2) Separation rules: separation rules classify the parameters detected by the primary rule into three sets of tokens, namely request-level, page-level, and session-level tokens.
(3) Precision rules: Precision rules eliminate parameters mistakenly identified as anti-CSRF tokens.
(4) Confidence rules: Confidence rules are to show the confidence level towards detected tokens.

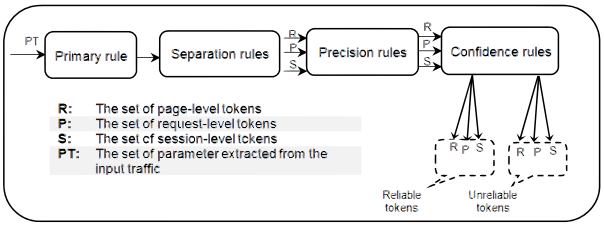

Figure 3. The Order of Applying Anti-CSRF Detection Rules.

The confidence rules divide detected tokens into two sets of reliable and unreliable tokens. The set of reliable tokens include parameters distinguished definitely as anti-CSRF tokens. Contrarily, the set of unreliable tokens includes random-valued parameters in requests for which there is not enough evidence to be anti-CSRF tokens due to reasons such as incomplete traffic.

Figure 3 shows the order in which the above rules are applied in our method. As shown in the figure, in the first step, the primary rule is applied to the parameters sent in the traffic. Then, using the separation rules, the extracted tokens from the previous step are divided into three sets of request-level, page-level, and session-level tokens. In the third step, the precision rules are applied to each set to remove false detections. Finally, the confidentiality rules are applied to the output sets of the precision rules to judge the reliability level of the results based on the completeness of the input traffic. The rules are explained below.

3.2.1 The Primary Rule

Rule 1- primary rule: Each parameter $p(n, v, u)$ in a request r can be an anti-CSRF token ($p \in TK$) if it has the following properties:

(1) $p.v$ and $p.n$ exist in the hidden elements of *ref* (r).
(2) $p.v$ has not been set by a user ($p.u = 1$).

Equation 6 formalizes the primary rule.

$$TK = \left\{ p \in T.P \;\middle|\; \begin{array}{c} \exists r, r' \in Req : (r \neq r'), \\ p \in r.P, p.u = 1, \\ res(r').url = ref(r), \\ res(r').e.n = p.n, \\ res(r').e.v = p.v, \\ res(r').e.t = 1 \end{array} \right\} \quad (6)$$

Equation 6 says that a parameter $p \in r.P$ is potentially an anti-CSRF token provided that there exists at least one request $r' \in Req \left(r \neq r'\right)$ such that:

(1) The referrer of r is the *url* address of the response page of $r^{'}$. That is, $res(r^{'}).url = ref\,(r)$.
(2) There is a hidden element in the response page of $r^{'}$ with a name and value equal to the name and value of p. That is, $res(r^{'}).e.type = 1$, $res(r^{'}).e.v = p.v$, and $res(r^{'}).e.n = p.n$.
(3) The value of p has not been set by a user. That is, $p.u = 1$.

The primary rule applies to all parameters extracted from the input traffic. Finally, a set of tokens TK is selected from the parameters.

3.2.2 The Separation Rules

The rules stated in this section, divide TK into three subsets, namely $TK_{Request}$, TK_{Page}, and $TK_{Session}$, which represent the set of parameters with an anti-CSRF token at the request-level, page-level, and session-level, respectively.

Rule 2- request-level separation rule: A parameter$p \in TK$ is a request-level token if its value is unique in the traffic as formulated in (7).

$$TK_{Request} = \{p_i \in TK \,|\, \forall p_j \in TK, j \neq i \Rightarrow p_i.v \neq p_j.v\} \tag{7}$$

Equation 7 states that p is considered as a request-level token if there is no parameter p' in the traffic whose value is equal to the value of p.

Rule 3- session-level separation rule: A parameter $p \in TK$ is a session-level token if its value is not seen in another session, as formulated in (8).

$$TK_{Session} = \left\{ p \in TK \,\middle|\, \begin{array}{c} \exists s, s^{'} \in T : s \neq s^{'}), \\ p \in s.P, p^{'} \in s^{'}.P, \\ p.n = p^{'} \\ \Rightarrow p.v \neq p^{'}.v \end{array} \right\} \tag{8}$$

Equation 8 says that p is a member of $TK_{Session}$ if there is no parameter p' with the same name in another session, having the same value as p.

Rule 4- page-level separation rule: A parameter p is a page-level token if it has the same value when it is sent from identical URLs and $p.v$ is seen in only one session. Equation 9 formalizes this rule.

$$TK_{Page} = \left\{ p \in TK_{Session} \,\middle|\, \begin{array}{c} \forall r, r^{'} \in Req : \\ r.url = r^{'}.url, \\ p \in r.P, \\ \exists p^{'} \in r^{'}.P, p.n = p^{'}.n \\ \Rightarrow p.v = p^{'}.v \end{array} \right\} \tag{9}$$

Equation 9 says that p is a page-level token in request r provided that p appears in any request r' of the same page having the same name and value. Since we assume that the value of a page-level anti-CSRF token is seen in only one session, page-level parameters are chosen from $TK_{Session}$.

After applying the separation rules, we obtain three sets $TK_{Request}$, TK_{Page}, and $TK_{Session}$ as the input sets to the precision rules.

3.2.3 The Precision Rules

After applying the separation rules, we observe that:

(1) A parameter, which is not an anti-CSRF token, is mistakenly detected as a token and falls in one of the sets.
(2) Some parameters are in common between $TK_{Request}$, TK_{Page}, and $TK_{Session}$ which must be deleted from the larger set.

The first precision rule is to remove the false detections, and the second precision rule is to remove identical members between the three sets. Precision rules prune $TK_{Request}$, TK_{Page}, and $TK_{Session}$ and produce new sets with the same name $TK_{Request}$, TK_{Page}, and $TK_{Session}$. To formulate this, we define $B \leftarrow A$ to say A is assigned to B when A and B are two sets. This notation is used for Equations (10) to (13).

In the remaining parts of this section, for the sake of simplicity, we do not take into account the page-level tokens. The rules described in this section can be easily generalized to the page-level tokens, as well.

The first precision rule: The former observation from the two above observations is about wrong detections and is usually due to the analysis of incomplete traffic. Let us clarify it with two examples.

Example 1 - Assume that in an e-shopping website, a request for the purchase of goods is sent from pages with different addresses. The request has a session-level anti-CSRF token p with $p.n = Token$. Assume that the traffic generated from visiting this site as the input of our method has two sessions. In the first session, the customer has purchased from url_1, and in the second session, two purchase requests are sent from url_2 and url_3. Therefore, in the first session, there is

Table 1. Output of the Separation Rules for the Sample Traffic in Example 1.

Output Tokens	URL of the request in which the token is sent
$TK_{Request}$= $\{p = (Token, v_1, 1)\}$	url_1
$TK_{Session}$ = $\{p = (Token, v_1, 1), p^{'} = (Token, v_2, 1)\}$	url_1, url_2, url_3

an anti-CSRF parameter p with $p.n = Token$ and $p.v = v_1$. In the second session, there are two parameters with $p.n = Token$ and $p.v = v_2$, sent in requests from url_2 and url_3. From this case, we conclude that:

(1) The parameter p with $p.n = Token$ and $p.v = v_1$ is a part of $TK_{Request}$ because v_1 is seen once in the traffic (there is no other parameter $p^{'}$ with $p'.v = v_1$). The first separation rule puts p in the set of request-level tokens.
(2) The parameter p with $p.n = Token$ and $p.v = v_1$ is in $TK_{Session}$ since v_1 is seen only in the first session. The second separation rule puts p in the session-level tokens.
(3) p with $p.n = Token$ and $p.v = v_2$ is placed in $TK_{Session}$ because there is no parameter in another session having the same value.

Table 1 shows anti-CSRF tokens derived in this example.

From this example, we conclude that $(Token, v_1, 1)$ is a parameter in two sets $TK_{Request}$ and $TK_{Session}$. However, the token used for this website is a session-level token. The reason that this parameter is also a member of $TK_{Request}$ is incomplete traffic. In other words, if there were two purchase requests in the first session, it was not concluded that $(Token, v_1, 1)$ is a request-level token. The first precision rule tries to remove such extra parameters.

Example 2 - Assume a request to add items to the shopping basket on the e-shopping website has a parameter p with $p.n = id$. This is the identifier of the product, added to the basket. Now, assume that from url_1 a user adds two products having the identifier id_1 and id_2 to his shopping basket and leaves the session (id_1 and id_2 are parameter values). Then, in another session, the user intends to remove id_1 from his basket, so sends the remove request from url_2. Our method, inputting such traffic and applying the primary and separation rules, outputs the following results:

(1) The parameter p with $p.n = id$ and $p.v = id_1$ is seen in two different sessions and it has none of the conditions to be a member of any of the token sets.
(2) The parameter p with $p.n = id$ and $p.v = id_2$ is seen only once in the traffic, so it is detected as a token in $TK_{Request}$ and also in $TK_{Session}$.

The sets obtained after applying the separation rules are shown in Table 2. The results in Table 2 are not precise since the parameter with the value id_2 is detected as an anti-CSRF token, while actually, it is not. This is because the number of requests with id parameter in the traffic is three, while according to the result obtained from the separation rules, resistant requests due to the presence of the request (or session)-level tokens for the parameter id is one. The first precision rule is applied to the above sets and tries to remove parameters such as $(id, id_2, 1)$ from the sets.

Table 2. Output of the Separation Rules for the Sample Traffic in Example 2.

Output Tokens	URL of the request in which the token is sent
$TK_{Request}$= $\{p = (id, id_2, 1)\}$	url_1
$TK_{Session}$ = $\{p = (id, id_2, 1)\}$	url_1

Based on the two above examples, the first precision rule is divided into request-level and session-level precision rules.

Rule 5- request-level precision rule: Parameter p with a specific name, observed k times in the traffic, is a request level token if there exist at least k resistant requests each of which has a parameter with the same name. Otherwise, p is deleted from $TK_{Request}$. Equation 10 formalizes this rule. In

$$TK_{Request} \leftarrow (TK_{Request} - \{p | p \in TK_{Request}, |A| > |B|\})$$

$$A = \bigcup_{r \in Req, p' \in r.P, p'.n = p.n} r \qquad (10)$$

$$B = \bigcup_{r \in Req, p' \in r.P, p'.n = p.n, p'.v = p.v} r$$

In (10), A is the set of requests in which there is a parameter $p^{'}$ with the same name as $p.n$. Similarly, B is the set of requests in which a parameter with the same name $p.n$ and value $p.v$ is sent. Equation 10 says that all parameters with a specific name must be removed from $TK_{Request}$ provided that the number of requests in traffic having a parameter with the same

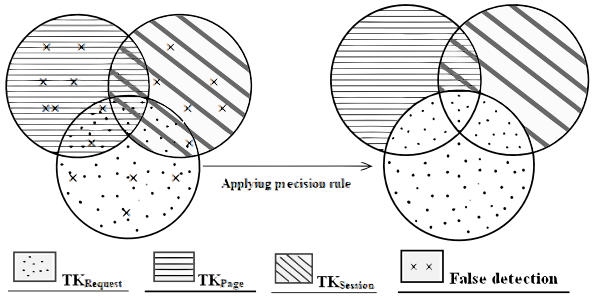

Figure 4. The Effect of the First Precision Rule.

name is greater than the number of requests, which are resistant due to having request-level anti-CSRF parameters with the same name.

In Example 1, the request sent from url_1 is the only request detected as a resistant request due to having a request-level anti-CSRF token. Also, three requests sent from url_1, url_2, and url_3 have been detected as resistant requests due to having session-level anti-CSRF tokens. The total number of requests in traffic having a parameter with the name *token* is three. Therefore, the result of applying this rule to the sets obtained from Example 1 is the removal of the parameter p with $p.n = token$ and $p.v = v_1$ from $TK_{Request}$.

Rule 6- session-level precision rule: The parameter p with a specific name, observed k times in traffic, is a session-level token if there exist at least k resistant requests due to having parameters as session-level tokens with the same name. Otherwise, p is deleted from $TK_{Session}$. Equation 11 formalizes this rule.

$$TK_{Session} \leftarrow (TK_{Session} - \{p | p \in TK_{Session}, |M| > |N|\})$$

$$M = \bigcup_{r \in Req, p' \in r.P, p'.n = p.n} r \qquad (11)$$

$$B = \bigcup_{r \in Req, p' \in r.P, p'.n = p.n, p'.v = p.v} r$$

In (11) M is the set of all requests in which a parameter p' with the same name $p.n$ has been sent. Also, N is the set of requests each of which has a parameter p' with the same name $p.n$ and the same value $p.v$. Figure 4 depicts the effect of the first precision rule on $TK_{Request}$, $TK_{Session}$, and TK_{Page}.

The second precision rule: Based on the separation rules, $TK_{Session}$ subsumes $TK_{Request}$ and TK_{Page}. That is, a request-level or page-level anti-CSRF token is also a session-level token, and a request-level token is also a page-level token. However, an anti-CSRF token is a member of one of the sets. Assume that $p(n, v, u)$ has been detected as a request-level anti-CSRF token. Therefore, v is sent only once in the traffic. It is clear that p has the condition of being in TK_{Page} as well as in $TK_{Session}$, but it must be removed from the two sets since actually, it is a request-level token. Similarly, session-level tokens, which are also seen in the set of page-level tokens, should be removed from the set of session-level tokens.

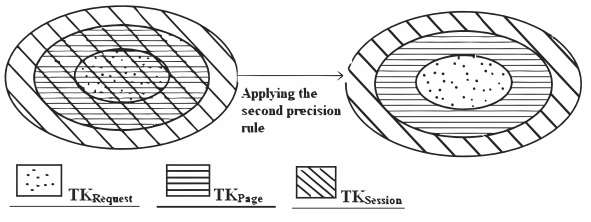

Figure 5. The Effect of the Second Precision Rule.

Rule 7- second precision rule: All request-level tokens are removed from the set of session-level tokens as shown in (12). Similarly, all request-level tokens are removed from the set of page-level tokens as shown in (13).

$$TK_{Session} \leftarrow (TK_{Session} - TK_{Request}) \qquad (12)$$

$$TK_{Page} \leftarrow (TK_{Page} - TK_{Request}) \qquad (13)$$

Figure 5 depicts the effect of the second precision rule on $TK_{Request}$, $TK_{Session}$, and TK_{Page}.

3.2.4 Confidence Rules

Since our method detects anti-CSRF tokens by the analysis of parameters in the traffic, more complete traffic leads to more reliable results. In this section, we express three confidence rules to show the conditions in which we can decide whether a detected parameter is an anti-CSRF token.

The first confidence rule: The first confidence rule targets request-level tokens. It examines the completeness condition of the traffic. The tokens for which the completeness condition is satisfied to constitute the set of reliable tokens. For reliable tokens, we are sure that the requests comprising those tokens are secure against CSRF. For the others, with respect to the existing traffic, there is not enough evidence for detected parameters to be anti-CSRF tokens.

Rule 8- request-level confidence rule: A parameter $p \in r.P$ is a reliable request-level token if

there is a session in which there exists a parameter $p' \in r'.P$ $(r \neq r')$ where $p'.n = p.n$. In (14), $TK_{Request_1}$ shows the set of reliable request-level tokens, and $TK_{Request_0}$ shows the set of remaining tokens, which are not reliable.

$$TK_{Request_1} = \left\{ p \in TK_{Request} \middle| \begin{array}{c} \exists s \in T : \\ p.p^{'} \in s.P, \\ \exists r, r^{'} \in Req, \\ r \neq r^{'}, \\ p \in r.P, p^{'} \in r^{'}.P, \\ p^{'}.n = p.n \end{array} \right\}$$

$$TK_{Request_0} = TK_{Request} - TK_{Request_1} \tag{14}$$

Equation 14 says that reliable request-level tokens are chosen from $TK_{Request}$, i.e., they have unique values. Moreover, it says that there should be at least two requests r_1 and r_2 in a session such that parameters with the same name (and different values) have been sent. Otherwise, it is possible in our method, due to incomplete traffic, that a request is detected as resistant while actually, there is not an anti-CSRF token in the request parameters.

The second confidence rule: The second confidence rule targets the set of session-level tokens and divides $TK_{Session}$ into two sets of reliable and unreliable tokens. The main condition for session-level tokens is their repetition (at least twice) in one session. Another condition is that a token must be seen at least in two sessions with the same name and different values. That is, the required evidence for $p \in TK_{Session}$ to be a session-level anti-CSRF token is that there must be two requests r_1 $(p \in r_1.P)$ and r_2 $(p \in r_2.P)$ in the same session and r_3 $(p \in r_3.P)$ in another session where $p.n$ is identical for the three requests and $p.v$ is different for requests in different sessions. Since parameter names belonging to $TK_{Session}$ have equal values in a session (see (8)), it is enough to check parameters in different sessions.

Rule 9- session-level confidence rule: A parameter $p \in r.P$ is a reliable session-level anti-CSRF token if there exists a request $r^{'}$ belonging to another session such that $r^{'}$ includes a parameter with the same name.

$$TK_{Session_1} = \left\{ p \in TK_{Session} \middle| \begin{array}{c} \exists s, s^{'} \in T : s \neq s^{'}, \\ p \in s.P, \exists p^{'} \in s^{'}.P, \\ p^{'}.n = p.n, \end{array} \right\}$$

$$TK_{Session_0} = TK_{Session} - TK_{Session_1} \tag{15}$$

Equation 15 says that p is reliably a session-level token if it appears with the same name in two distinct sessions s and $s^{'}$. In (15), $TK_{Session_1}$ includes parameters that are reliably deduced to be session-level tokens. Contrarily, $TK_{Session_0}$ includes the tokens, which may not be a session-level token when we augment the input traffic.

The third confidence rule: The conditions of page-level tokens are similar to the conditions of session-level tokens. The third confidence rule divides TK_{Page} into two sets TK_{Page_1} and TK_{Page_0} as the set of reliable and unreliable tokens, respectively.

Rule 10- page-level confidence rule: A parameter p is a reliable page-level token if a parameter with the same name $p.n$ is seen in two distinct requests r_1 and r_2 originating from one page in a session, and also in a different request r_3 originating from the same page but in another session.

$$TK_{Page_1} = \left\{ p \in TK_{Page} \middle| \begin{array}{c} \exists s, s^{'} \in T, (s \neq s^{'}), \\ r, r^{'} \in Req^{s}, (r \neq r^{'}), \\ r^{''} \in Req^{s^{'}} : \\ r.url = r^{'}.url = r^{''}.url, \\ p \in r.P, \\ \exists p^{'} \in r^{'}.P, p^{'}.n = p.n \\ \exists p^{''} \in r^{''}.P, p^{''}.n = p.n \end{array} \right\}$$

$$TK_{Page_0} = TK_{Page} - TK_{Page_1} \tag{16}$$

Equation 16 says that p is reliably a page-level token if it appears with the same name in two request r and $r^{'}$ of the same page and same session as well as in another request $r^{''}$ of the same page but another session. In (16), TK_{Page_1} includes the parameters which are confidently page-level anti-CSRF tokens. Contrarily, TK_{Page_0} includes remaining parameters for which we are not sure that they are page-level anti-CSRF tokens.

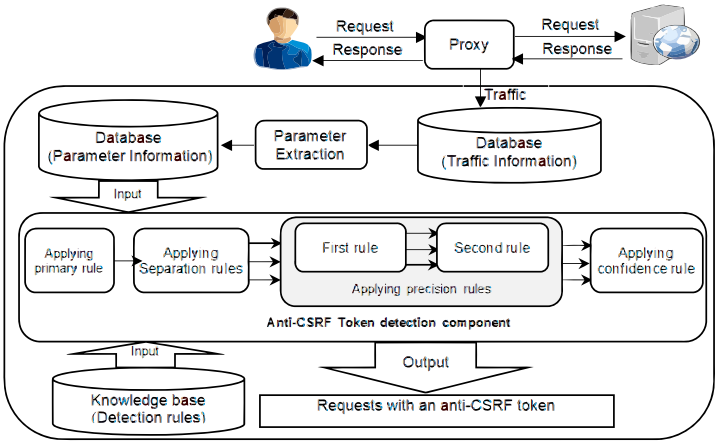

Figure 6. The Architecture of Our System to Detect Resistant Requests Against CSRF.

4 Implementation and Evaluation

We implemented our method with C# in Microsoft Visual Studio as a development environment. SQLite Studio 3.1.1 is the DBMS and OWASP ZAP is the proxy. We used the proxy for storing the traffic from/to the browser to be used as the input of our method.

Figure 6 depicts the architectural view of the proposed method. As shown in the figure, the Parameter Extraction component extracts the parameters along with their values from the input traffic and stores them on a database. Then, the primary, separation, and precision rules are applied step by step on the extracted parameters. At this step, we have three sets of request, page, and session-level anti-CSRF tokens. The confidence rules are finally applied to the sets to categorize the sets into reliable and unreliable detections with respect to the input traffic.

4.1 Evaluation Results

We carry out a set of experiments to evaluate our method. Our experiments are organized into four case studies. The first three case studies are performed on Getboo, Bamilo, and Chmail, respectively. In these case studies, we evaluate the effectiveness of our detection rules on the gathered traffic of the mentioned websites. The third case study also examines the effect of traffic volume on the accuracy of the results. In the fourth case study, we test two publicly accessible vulnerable web applications, namely BWAPP and OWASP-Mutillidae, and compare the results of our method with that of Acunetix, a well-known vulnerability scanner [1]

In all our experiments, we manually generate input traffic by simulating the behavior of a real user in different sessions; logging in, crawling, and doing interactions with the application. We use OWASP ZAP proxy to gather user interactions with the websites and form the input traffic. The results of our method on the input traffic are compared with the real state of the websites. The real state is obtained through testing of the website against CSRF performed by a human expert.

Figure 7. The Number of Parameters and Detected Tokens for Getboo.

Case Study 1: Getboo is one of OWASP BWA [2] projects. The obtained traffic of this application consists of 1261 requests containing 432 parameters. Figure 7 shows the number of parameters detected as anti-CSRF tokens after applying the primary, separation, and precision rules.

The results of tests performed by a human expert on Getboo indicate that our method has no false negative in CSRF token detection. Also, the false positives come from the incompleteness of the input traffic. Table 3 has summarized the outcome of precision and confidence rules for our first case study. As we see in Table 3, the number of expected tokens (obtained from manual tests performed by a human expert) is equal to the reliably detected tokens.

Case Study 2: Bamilo is a shopping and sales website whose traffic in our experiment consists of 898 requests with 162 parameters. Figure 8 illustrates the results of our method on Bamilo.

[1] https://www.acunetix.com/vulnerability-scanner/

[2] Broken Web Application

Table 3. The Number of Detected Anti-CSRF Tokens vs. the Expected Number of Anti-CSRF Tokens for Getboo.

	precision rules	reliable	unreliable	expected tokens obtained from manual tests
the number of request-level tokens	51	51	0	51
the number of page-level tokens	1	0	1	0
the number of session-level tokens	6	0	6	0

Table 4. The number of detected anti-CSRF tokens vs. the expected number of anti-CSRF tokens for Bamilo.

	precision rules	reliable	unreliable	expected tokens obtained from manual tests
the number of request-level tokens	2	0	2	0
the number of page-level tokens	0	0	0	0
the number of session-level tokens	7	7	0	7

Figure 8. The number of parameters and detected tokens for Bamilo.

Figure 9. The Number of Parameters and Detected Tokens for Chmail.

Table 4 shows the results of precision and confidence rules for Bamilo. As Table 4 shows, our method does not have false negatives in this case. Moreover, all its false positives result from inadequate traffic. As we can see in Table 4, applying the confidence rules in our method confirms that the two false detections have been detected as unreliable tokens.

Case Study 3: Chmail is an electronic mail system as the second case study. The gathered traffic consists of 86 requests and 150 parameters. Figure 9 shows the number of parameters in each step of the token detection process.

Table 5 has summarized the outcome of precision and confidence rules for Chmail. As we can see in Table 5, the expected number of request-level tokens is two, while the system has detected five tokens. Also, 38 session-level tokens have been detected by the system while there are eight tokens. The confidence rules have recognized 38 tokens as unreliable. It seems that the system is not performing well in this case. The reason for having inadequate results is incomplete traffic related to a large percentage of extracted parameters. This is evidenced by information in Table 5, which shows that 42 detected tokens are unreliable. This is about 93% of all detected tokens, i.e., 45 tokens (summation of request, page, and session-level tokens). That is, the traffic of about 93% of parameters is not complete enough to provide us with an accurate inference.

Now, we augment the traffic by increasing the number of requests-responses in the traffic to investigate its effect on the accuracy of our results. The augmented Chmail traffic consists of previous request-responses plus new requests-responses, inserted into the existing traffic. It includes 244 requests having 524 distinct parameter names. Figure 10 shows the number of parameters after applying the detection rules.

Table 6 shows the results of precision and confidence rules. As shown in Table 6, with increasing traffic volume, the result becomes more accurate. Before traffic augmentation, three out of five request-level detected tokens were false detections. However, with the new traffic, the method has detected seven request-level tokens, and none of them are false detections. Another observation in Table 6 is the significant increase in the number of page-level tokens, compared to the previous results. The method has detected 27 page-level tokens among which 18 are unreliable. This indicates that the traffic for these parameters is still incomplete. For the case of session-level tokens, traffic augmentation leads to get accurate results in which all 16 detected tokens are anti-CSRF tokens.

Case Study 4: Now, we compare the results of our method with Acunetix as a well-known vulnerability

Table 5. The Number of Detected Anti-CSRF Tokens vs. the Expected Number of Anti-CSRF Tokens for Chmail.

	precision rules	reliable	unreliable	expected tokens obtained from manual tests
the number of request-level tokens	5	3	2	2
the number of page-level tokens	2	0	2	1
the number of session-level tokens	38	0	38	8

Table 6. The Number of Detected Anti-CSRF Tokens vs. the Expected Number of Anti-CSRF Tokens for Chmail With the Augmented Traffic.

	precision rules	reliable	unreliable	expected tokens obtained from manual tests
the number of request-level tokens	7	7	0	7
the number of page-level tokens	27	18	9	12
the number of session-level tokens	16	16	0	16

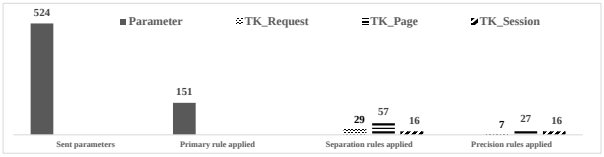

Figure 10. The Number of Parameters and Detected Tokens for the Augmented Traffic of Chmail.

scanner, whose claim is to detect CSRF vulnerabilities. We would like to emphasize that our method is not directly comparable with vulnerability scanners such as Acunetix since we detect resistance requests against CSRF and they detect vulnerable requests. However, since the ultimate goal is to detect CSRF vulnerabilities in web applications, by this comparison, we aim to show that our solution gives more reliable results compared to Acunetix. It helps security testers to have a better analysis of web applications.

OWASP BWAPP and OWASP-Mutillidae are two vulnerable web applications used in this experiment. BWAPP is a free and open-source deliberately insecure web application, having three vulnerable requests against CSRF. We ran Acunetix for BWAPP and it detects no vulnerable request among the total of 4,356 identified requests. Mutillidae is also a free and open-source vulnerable web application for training purposes. Among the total of 3,719 requests, it has three requests vulnerable to CSRF. Acunetix reported none of them in its results. For both these applications, our method detects no anti-CSRF tokens in the requests. That is, no CSRF resistant request is detected using our method. From the security point of view, this means that all the requests have the potentiality of being vulnerable against CSRF.

5 Summary and Conclusions

Our goal in this research was to provide a method for the black-box detection of resistant requests against CSRF attacks. The goal is intended to facilitate the security evaluation of web-based software applications. In the proposed method, a security evaluator or a white-hat hacker does not need to access the source code of the web application. It is only required to access browser-server traffic. Furthermore, the evaluation process is non-invasive; it does not lead to any changes to the application database.

We proposed 10 rules in four different categories, applied on the request parameters in the website traffic. The rules are aimed at extracting anti-CSRF tokens from the traffic to find out which requests are resistant to CSRF. Since the method tries to detect anti-CSRF tokens in the client-server traffic, more voluminous traffic pertaining to different sessions increases the precision and confidence level towards the results.

We plan to investigate more on the CSRF defense mechanisms and extend the proposed rules to reduce the number of false detections of the method. Our method is currently limited to detecting anti-CSRF tokens as a defensive mechanism. However, it cannot distinguish whether a detected token is properly checked on the server-side. This is due to the passive nature of our method. It can be extended to perform an active attack and send a request whose detected token has been manipulated. Then, the method can examine the application behavior in response to the manipulated request to identify whether the attack is successful.

Since our method is not a real-time detection

method, time efficiency is not a major issue. However, we also plan to work on the implementation issues of our method and improve the efficiency of the current implementation.

References

[1] B. Liu, L. Shi, Z. Cai, and M. Li. Software Vulnerability Discovery Techniques: A Survey. In *2012 fourth international conference on multimedia information networking and security*, pages 152–156. IEEE, 2012. ISBN 978-1-4673-3093-0. doi:10.1109/MINES.2012.202.

[2] R. D. Kombade and BB. Meshram. CSRF Vulnerabilities and Defensive Techniques. *International Journal of Computer Network and Information Security*, 4(1), 2012. doi:10.5815/ijcnis.2012.01.04.

[3] H. Shahriar and M. Zulkernine. Client-Side Detection of Cross-Site Request Forgery Attacks. In *2010 IEEE 21st International Symposium on Software Reliability Engineering*, pages 358–367. IEEE, 2010. ISBN 978-1-4244-9056-1. doi:10.1109/ISSRE.2010.12.

[4] M. Rocchetto, M. Ochoa, and M. T.Dashti. Model-based Detection of CSRF. In *IFIP International Information Security Conference*, pages 30–43. Springer, 2014. ISBN 978-3-642-55414-8. doi:10.1007/978-3-642-55415-5_3.

[5] M. Rocchetto, M. Ochoa, and M. T.Dashti. A Study of the Effectiveness of CSRF Guard. In *2011 IEEE Third International Conference on Privacy, Security, Risk and Trust and 2011 IEEE Third International Conference on Social Computing*, pages 1269–1272. IEEE, 2011. ISBN 978-1-4577-1931-8. doi:10.1109/PASSAT/SocialCom.2011.58.

[6] K. Jayaraman, W. Du, B. Rajagopalan, and S. J. Chapin. ESCUDO: A Fine-Grained Protection Model for Web Browsers. In *2011 IEEE Third International Conference on Privacy, Security, Risk and Trust and 2011 IEEE Third International Conference on Social Computing*, pages 231–240. IEEE, 2010. ISBN 978-1-4244-7261-1. doi:10.1109/ICDCS.2010.71.

[7] OWASP Foundation. Cross-Site Request Forgery (CSRF). `http://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF)`, Date Accessed: June 3, 2018.

[8] S. Sadeghi and M. A. Hadavi. Black-box Detection of Resistance against CSRF Attacks (in Persian). In *15th International ISC Conference on Information Security and Cryptology*, 2018.

[9] P. Khurana and P. Bindal. Vulnerabilities and Defensive Mechanism of CSRF. *International Journal of Computer Trends and Technology*, 13(4):2231–2803, 2014. ISSN 2231-2803. doi:10.14445/22312803/IJCTT-V13P135.

[10] M. S. Siddiqui and D. Verma. Cross-site request forgery: A common web application weakness. In *2011 IEEE 3rd International Conference on Communication Software and Networks*, pages 538–543. IEEE, 2010. ISBN 978-1-61284-485-5. doi:10.1109/ICCSN.2011.6014783.

[11] T. Oda, G. Wurster, P. C. Van Oorschot, and A. Somayaji. SOMA: mutual approval for included content in web pages. In *Proceedings of the 15th ACM conference on Computer and communications security*, page 89–98. ACM, 2008. doi:10.1145/1455770.1455783.

[12] E. Y. Chen, J. Bau, C. Reis, A. Barth, and C. Jackson. App isolation: get the security of multiple browsers with just one. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 227–238. ACM, 2011. doi:10.1145/2046707.2046734.

[13] M. Bugliesi, S. Calzavara, R. Focardi, W. Khan, and M. Tempesta. Provably Sound Browser-Based Enforcement of Web Session Integrity. In *2014 IEEE 27th Computer Security Foundations Symposium*, pages 366–380. IEEE, 2011. ISBN 978-1-4799-4290-9. doi:10.1109/CSF.2014.33.

[14] S. Calzavara, R. Focardi, M. Squarcina, and M. Tempesta. Surviving the Web: A Journey into Web Session Security. *ACM Computing Surveys (CSUR)*, 20(1):1–34, 2017. doi:10.1145/3038923.

[15] G. Pellegrino, M. Johns, S. Koch, M. Backes, and C. Rossow. Deemon: Detecting CSRF with Dynamic Analysis and Property Graphs. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1757–1771. ACM, 2017. doi:10.1145/3133956.3133959.

[16] M. Srokosz, D. Rusinek, and B. Ksiezopolski. A new WAF-based architecture for protecting web applications against CSRF attacks in malicious environment. In *2018 Federated Conference on Computer Science and Information Systems (FedCSIS)*, pages 391–395. IEEE, 2018. ISBN 978-83-949419-5-6. doi:10.15439/2018F208.

[17] S. Calzavara, M. Conti, R. Focardi, A. Rabitti, and G. Tolomei. Machine Learning for Web Vulnerability Detection: The Case of Cross-Site Request Forgery. *IEEE Security & Privacy*, 18(3):8–16, 2020. ISSN 1540-7993. doi:10.1109/MSEC.2019.2961649.

[18] S. Calzavara, M. Conti, R. Focardi, A. Rabitti, and G. Tolomei. Mitch: A machine learning approach to the black-box detection of CSRF vulnerabilities. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages

528–543. IEEE, 2019. ISBN 978-1-7281-1149-0. doi:10.1109/EuroSP.2019.00045.

Samira Sadeghi has been graduated in M.Sc. degree in Computer Engineering security orientation from Malek Ashtar University of Technology, Tehran, Iran, in 2017. she has been graduated in B.Sc. in degrees in Computer Engineering software orientation from Bu-Ali Sina University of Technology, Hamadan, Iran, in 2004. Her research interests including database security, network security, and network.

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.