Model-Based Detection of CSRF

Marco Rocchetto, Martín Ochoa, Mohammad Torabi Dashti

▶ **To cite this version:**

Marco Rocchetto, Martín Ochoa, Mohammad Torabi Dashti. Model-Based Detection of CSRF. 29th IFIP International Information Security Conference (SEC), Jun 2014, Marrakech, Morocco. pp.30-43, 10.1007/978-3-642-55415-5_3 . hal-01370351

HAL Id: hal-01370351

https://inria.hal.science/hal-01370351

Submitted on 22 Sep 2016

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License

Model-based Detection of CSRF

Marco Rocchetto[1], Martín Ochoa[2], and Mohammad Torabi Dashti[3]

[1] Università di Verona, Italy
[2] Technische Universität München, Germany
[3] ETH Zürich, Switzerland

Abstract. Cross-Site Request Forgery (CSRF) is listed in the top ten list of the Open Web Application Security Project (OWASP) as one of the most critical threats to web security. A number of protection mechanisms against CSRF exist, but an attacker can often exploit the complexity of modern web applications to bypass these protections by abusing other flaws. We present a formal model-based technique for automatic detection of CSRF. We describe how a web application should be specified in order to facilitate the exposition of CSRF-related vulnerabilities. We use an intruder model, à la Dolev-Yao, and discuss how CSRF attacks may result from the interactions between the intruder and the cryptographic protocols underlying the web application. We demonstrate the effectiveness and usability of our technique with three real-world case studies.

1 Introduction

HTTP and HTTPS, the dominant web access protocols, are stateless. Web servers therefore use Cookies, among other means, to keep track of their sessions with web clients. Cookies are stored by the client's browser; whenever the client sends an HTTP(S) request to the web server, the browser automatically attaches to the request the Cookie that is originated from the web server. This mechanism allows the clients to experience a seamless stateful web browsing, while in fact using an inherently stateless protocol such as HTTP.

Cross-site request forgery attacks (CSRF) exploit the aforementioned mechanism of automatically-attached Cookies. A typical CSRF occurs as explained in the following. The attacker tricks the client into accessing a sensitive web server by making a rogue URL link available to the client: the link instructs the web server to perform a transaction on behalf of the client (e.g. to transfer money). If the client accesses the web server through the rogue link, then in effect the client requests the web server to perform the transaction. The only missing part of the puzzle is a valid Cookie that needs to be attached to the request, so that the web server authenticates the client.

Now, if it happens that the client accesses, via the rogue link, the web server *while* a session between the client and the web server is active, then the client's browser automatically attaches the proper Cookie to the request. The web server would then accept the attacker-generated request as one genuinely sent by the client, and the attack is successful. The web server can deter the attack by checking that critical requests are in fact generated by the client: the requests may have to include an extra random value that is only known to the client and the web server passed as a POST parameter, the

web server might prompt the client to solve a CAPTCHA to demonstrate that he is aware of the transaction taking place, etc.

However, a number of related vulnerabilities and design flaws might render such countermeasures against CSRF useless. Due to the complexity of modern web applications[4], those vulnerabilities might be difficult to spot. For instance, if the web server uses poorly generated random values, the attacker may open simultaneous sessions with the web server, record the random values, and infer their pattern. It is also well known [7] that state-of-the-art vulnerability scanners do not detect vulnerabilities linked to logical flaws of applications. In general, one should proceed with care when assessing the security of productive servers for vulnerabilities with potential side-effects such as CSRF, since one might affect the integrity of data, making manual testing a challenging task.

Contributions. To address these problems, we propose a model-based technique in order to detect issues related to CSRF during the design-phase. The essence of the formal model is simple: the client acts as an oracle for the attacker. The attacker sends a URL link to the client and the client follows the link. The bulk of the model is therefore centered around the web server, which might have envisioned various protection mechanisms against CSRF vulnerability exploitation. The attacker, in our formal model, is allowed to interact with the web server and exhaustively search his possibilities to exploit a CSRF. The expected result of our technique is, when a CSRF is found, an abstract attack trace reporting a list of steps an attacker has to follow in order to exploit the vulnerability. Otherwise, the specification is safe (under a number of assumptions, as described in Sect. 2.3) with respect to CSRF. We demonstrate the effectiveness and the usefulness of our method through a made-up example (Sect. 2.4) and three real-world case studies (Sect. 3): DocumentRepository and EUBank (two anonymized real-life applications) and WebAuth [11].

More specifically, in this paper, we propose a model-based analysis technique that (i) extends the usage of state of the art model-checking technology for security to search for CSRF based on the ASLan++ language [14]. We also (ii) investigate the usage of the intruder, à la Dolev-Yao [6] (DY from now on), for detecting CSRF on web applications (while it is usually used for security protocols analysis) and, finally, we (iii) show how to concretely use the technique with real web applications.

Structure of the paper. Section 1 gives a general overview of CSRF. In Sect. 2 we describe how to model a web application in order to search for CSRF; there we also introduce the specification language used and our running example. In Sect. 3 we present three case studies, and discuss our findings. In Sect. 4 we discuss related work, and finally in Sect. 5 we conclude the paper proposing future research directions.

2 Modeling CSRF

In this section we describe a technique for modeling web applications in order to search for CSRF. We first give an overview of the CSRF and then we define general guidelines

[4] A web application is a software application hosted on one or more web servers

for writing a specification with the focus on CSRF detection. Finally, we introduce the ASLan++ language used in Sect. 2.3 in which we formally define our technique.

2.1 CSRF

As described in Sect. 1, in order to exploit a CSRF, and attack[5] a web application, mainly three parties have to get involved: an intruder, a client and a web server. The intruder is the entity that wants to find (and then to exploit) the vulnerability and attack the web application hosted on the web server. The web server is thus the entity that represents the web application host and, finally, the client entity is the honest agent who interacts with the web application (i.e. with the web server).

If the web application is vulnerable to CSRF, an attacker can trick the client to perform requests to the web server on his behalf. This attack scenario (depicted in Fig. 1-Left) can be summarized by the following steps:

1. the client logs in to the web application (authentication phase)
2. the web server sends a Cookie (Cookie exchange) to the client who will store it (within the web browser).
3. From this point on, the Cookie will be automatically attached by the web browser to every request sent by the client to the web server (in message 3. of Fig. 1-Left the client sends an honest request along with his Cookie)
4. the intruder sends to the client a malicious link containing a request (`Request'`) for the web application on the web server
5. if the client follows the link, the web browser will automatically attach the Cookie and will send the malicious request to the web server
6. the web application cannot distinguish a request made by the intruder and forwarded by the client from one made and sent by an honest agent; therefore, it accepts the request.

It is important to observe that, from the description of CSRF we have given, an intruder sees the client as an "oracle". The intruder does not see the communication between the client and the web server but it will send a request to the client and wait for it to be executed.

The state-of-the-art protections against CSRF attacks are mainly two (as reported in [10]) and can be used together:

- the web server asks the client for a confirmation at every request the client sends to the web server
- a secret, usually called *CSRF token* (e.g. a pseudo-random token), shared between the client and the web server, has to be attached to every request

In Fig. 1-Right we report the message sequence chart (MSC) of a web application that uses both these CSRF protection mechanisms. In this way, the intruder cannot simply send a request to the client and wait for its execution. In fact, the client will not confirm the request and the browser will not automatically add the secret to the request.

[5] In this context, with "attacking a web application" we mean that an intruder can perform requests to the web application that it should not be allowed to do.

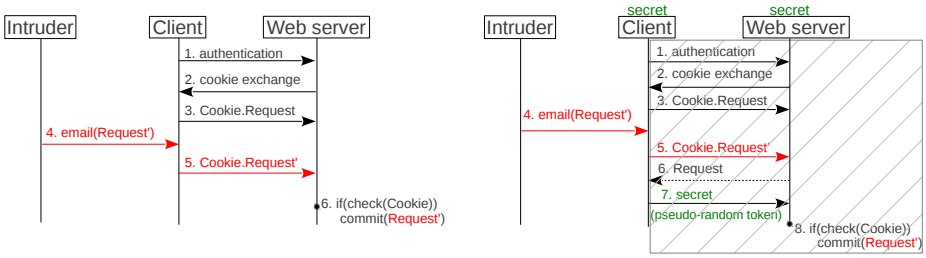

Fig. 1. (Left) CSRF Oracle Message Sequence Chart - (Right) CSRF from the intruder point of view and the barred part is not visible to the intruder

Our goal is to check if protections against CSRF, implemented in a web application, are strong enough; that is, to check if there is a way for the intruder to bypass protections and force the web server to commit a rogue request that it is not allowed to do. Before defining our technique for specifying a web application with the focus on detecting CSRF we show a set of rules (guidelines) that a modeler should be aware of for defining a web application model without trivial flaws that can lead to a CSRF:

- the CSRF token has to be unique for each client
- the CSRF token must be unique for each client-server interaction
- the CSRF token must not be sent with the query string (the part of URL containing data to be passed to the web application) in the URL
- a request has to fail if the CSRF token is missing

2.2 An introduction to ASLan++

In this section we give a brief presentation of the formal language ASLan++ [14], focusing on the aspects we use for modeling web applications to search for CSRF. This aspects have been used to model our case studies and the running example of Sect. 2.3.

The ASLan++ language is a typed language for formally specifying security-sensitive web servers, web applications and service-oriented architectures. ASLan++ can be used to specify a web system and its security goals, with a modeling language similar to programming languages. The specification will be automatically translated (using a translator [2]) into a more low-level ASLan specification that serves as input of the model checking tools of the AVANTSSAR platform [2]. These will return an abstract attack trace of an attack on the model, if found.

An ASLan++ specification consists in a hierarchy of entity declarations, that are similar to Java classes. The top-level entity is usually called Environment in which commonly the Session entity is defined. The Session entity is composed by sub-entities that are the main principal involved in the system (e.g., clients, servers). Each sub-entity defines the internal behavior of the component it models and the interaction with other entities. For example, the following ASLan++ code represents a simple communication between a client and a server in which the client creates a nonce [6] that he sends to the server. We first focus on the client perspective.

[6] Nonce is a freshly generated number, used only once in the execution of the system.

```
1  entity Environment{
2  ...
3   entity Session(U,S:agent){
4
5    entity Client(Actor, Server: agent){
6     symbols
7      Na: text;
8
9     body{
10     Na:=fresh();
11     Actor -> Server: Na;
12    }
13   }
14    entity Server(Actor, Client:
          agent){
15     Client -> Actor: Na;
16   }
17
18   body{ %of Session
19    new Client(U,S);
20    new Server(U,S);
21   }
22  }
23
24  body{ %of Environment
25   any U. Session(U, Server);
26  }}
```

The keyword **entity** defines a new component of the model (in this case we consider the `Client` entity). It accepts arguments, the first one is **Actor** that defines the name of the agent playing the role of the entity Client. The other parameter defines the name of the Server entity that the client wants to communicate with.

Inside the Client entity we have the keyword **symbols** that is used to define the type of all the variables, constants, functions used inside the entity. In our case there is only the `Na` variable with type `text` [7].

Inside the **body** section the behavior of the entity is defined. In this example we have the assignment `Na:=fresh();` that sets the variable `Na` to a new constant calculated by the function `fresh()`.

The Server entity is the dual of the Client one, so it will receive the message that the client has sent: `Client -> Actor:?Na;`. The **Actor** keyword in the server refers to the server entity itself and the `?` is used to assign a value to the variable it precedes.

There are several different types of channels in ASLan++ but we are interested in only four types. The plain communication channel `->` defines an insecure communication between two entities; that is, no authentication nor confidentiality will be guaranteed for the messages going through the channel. The other three types of communication channels are the authentic one, `*->`, that ensure messages come from the claimed sender, the confidential one (`->*`) in which messages can only be received from the intended receiver and the secure channel `*->*` that combines both authentication and confidentiality.

The Session entity gathers together the two entities Client and Server and in its body section creates the new instances of the two sub-entities (**new** `Client(U,S);` and **new** `Server(U,S);`).

The session entity is called by the Environment that instantiates the Session. We now show a shorthand useful for session instantiation that does not bound the client to a particular constant but expresses that each agent of the specification can impersonate the role: **any** `U. Session(U, Server);`.

The ASLan++ language allows us to also define conditionals. There are the usual if-then-else statements and also a **select**`{`**on**`(statement):{<positive branch>}}` that is semantically equivalent to the positive evaluation of if-then-else statement (i.e., the nega-

[7] There are several types: agent, text, message are the main ones. Agent is used to define roles while text and message are for variable and constant. The main difference between text and message is that the former cannot be decomposed by an intruder, while message can.

tive branch will not be considered by the model checker). There is also the `while(...)` loop that is used to define that a process (usually a server) is listening for incoming communication.

The last two points we want to briefly discuss are the intruder role and the goal section. The intruder/attacker (à la Dolev-Yao [6]) is intended to control the entire network but every cryptographic algorithm is treated as if cryptography were perfect; that is, the intruder can collect all the messages that are transmitted over the network but cannot break cryptography.

ASLan++ defines several ways to formalize security goals but the one we are interested in is reachability. In the rest of the paper we will use a predicate `commit` over a variable that if is reached with a particular assignment then the CSRF is used by the intruder to attack the web application. We will discuss this last point in details in Sect. 2.3. There are several other aspects of the language that are outside the scope of this paper, however the ones we have briefly defined are enough to understand the ASLan++ code that we have used in the reminder of the paper.

2.3 CSRF in ASLan++: modeling and detection

In this section we describe a technique for modeling web applications to check for CSRF. We will use the DocumentRepository specification as a running example. Even if no attack has been detected on this case study it is illustrative for several reasons: it uses the usual client-server paradigm, it models Cookies generation, storage and exchange (using a database) and the server handles login, commit, and malicious requests. Before going into the details of the modeling part we give the system description of the running example.

DocumentRepository description The DocumentRepository [8] system is a document repository that implements a document management system for the secure management and sharing of documents via web browser. Its main purpose is then to share and store different documents produced by various group of possibly different institutions. Suppose that both Alice and Bob (from two different institutions) are using the repository system. A typical scenario is the following:

- Alice logs in, via the login page, by providing her credentials (username and password)
- Alice is then forwarded to the system starting page where she can browse to the repositories list. She can now access to all public repositories and to private ones to which she has the permission
- Alice clicks on one of the private repositories she has access to (repository A) and uploads a new document
- Now Bob, who is the administrator of the repository A can download, edit or remove the file Alice has just uploaded and he can also edit Alice's permission on the private repository

[8] The DocumentRepository system is a non public industrial case study within the SPaCIoS project. We have then hided the real name of the system and omitted some of the details.

ASLan++ modeling for CSRF detection In order to check for CSRF we consider two entities: *Client/Oracle* entity and *Server* entity, as described in Sect. 2.1.

Client/Oracle. In the Client entity we model a first authentication phase to obtain the Cookie and logging in to the web application. First, in line 2, the client sends its credentials (username and password) and then the server, upon checking the received credentials are correct, sends back a new Cookie to the Client (lines 5, 6).

```
1  % sends his/her name and password to the server's login service
2  Actor ->* Server: Actor.UserName.Password;
3
4  % the server's login service responds to the login request with a Cookie
5  select { on (Server *->* Actor: ?Cookie &
6           ?Cookie=cookie(UserName,?,?)): {} }
```

After this phase, the Client can perform requests to the server asking for services. When a client wants to send a request to the DocumentRepository system, it first loads the web page (usually using a web browser). The server produces the web page and sends it together with a CSRF token (i.e., a fresh pseudo-random token linked to the session ID of the Client). At specification level, skipping line 7 for the moment, we can model this mechanism by creating a variable `Request` that the Client wants to submit. When the Client sends this `Request` to the server (line 10), the latter will generate and send the token, `CSRFToken`, back to the Client (line 11). Now the Client sends (line 12) the `Request` together with the Cookie and the CSRF token.

```
7  ? -> Actor: ?Request;
8  % load request page with the csrf token
9  % client asks for a web page; server sends it to him including a csrf token
10 Actor *->* Server: Cookie.Request;
11 Server *->* Actor: ?CSRFToken;
12 Actor *->* Server: Cookie.Request.CSRFToken;
```

Between the authentication and the request submission parts, in line 7, we have added a message containing a variable `Request`. This message is sent from an unknown entity in order to model the scenario in which the Client receives a malicious email from a third party; the email contains a link to submit a request to the web application.

Finally, in line 14, the Client will receive from the server the confirmation that the request has been executed by the web application.

```
13 % the server's frontend sends back to the client the answer
14 Server *->* Actor: Request.?Answer;
```

Server. The server entity accepts three different kinds of requests: authentication, request for a web page and request that it has to commit to the web application.

With *authentication request* a Client (if not already authenticated) sends to the server its username and password asking to log in (line 16). The server will check the received credentials (lines 17, 19) and, if they are correct, it will generate a Cookie (line 25) that will be sent back to the Client (line 30).

```
15 % 1) login service receives the client request and generate a new session Cookie
16 on((? ->* Actor: ?UserIP.?UserName.?Password
17     & !dishonest_usr(?UserName)) &
18     % checks if the data are available in the database
19     loginDB->contains((?UserName,?Password,?Role))): {
20     % we have checked, using the password, that the client is legitimate.
```

```
21      % With the query, we extract the role of the legitimate client.
22
23      % creates the Cookie and sends it back to the client
24      Nonce := fresh();
25      Cookie := cookie(UserName,Role,Nonce);
26      % adds the Cookie into the DB associated with the name of the client
27      cookiesDB->add(Cookie);
28
29      % uses the IP address to communicate the Cookie to the correct client
30      Actor *->* UserIP: Cookie;
31  }
```

The second type of request is a *web page request*. The Client asks for a web page before sending a request to the web application. The Client is already logged in and then it sends the request together with the Cookie (line 34). The server will check the Cookie (line 35) and generate a fresh token (line 37) that will send back to the Client (line 39).

```
32  % 2) with a Cookie, a client makes a request to the frontend
33  % without the CSRF token and receives the respective token from the repository
34  on(?UserIP *->* Actor: cookie(?UserName,?Role,?Nonce).?Request &
35  cookiesDB->contains(cookie(?UserName,?Role,?Nonce))): {
36
37          CSRFToken:=fresh();
38          csrfTokenDB->add((UserIP,Request,CSRFToken));
39          Actor *->* UserIP: CSRFToken;
40  }
```

The third case is when a Client sends a *request to the server* (line 43). The server checks both the token (line 46) and the Cookie (line 50) and then commits the request (line 54).

```
41  % 3) a client makes a request (along with a Cookie) to the frontend
42  %and receives the answer from the repository
43  on(?UserIP *->* Actor: cookie(?UserName,?Role,?Nonce).?Request.?CSRFToken &
44
45    % checks if the token is the right one
46    csrfTokenDB->contains((?UserIP,?Request,?CSRFToken)) &
47
48    % checks if the client is allowed to do this request and the link client-Cookie
49    checkPermissions(?UserName,?Request) &
50    cookiesDB->contains(cookie(?UserName,?Role,?Nonce))): {
51
52    % if the client has the right credential, then the request
53    % is executed and the answer is sent back to the Client
54    commit(Request);
55    Answer := answerOn(Request);
56    Actor *->* UserIP: Request.Answer;
57   }
58  }
```

Goal. The goal is to check if there is a way for the intruder to commit a request to the web application. We use a predicate `commit` over a constant `intruderRequest` that if true, the intruder has found a CSRF. `csrf_goal: [](!commit(intruderRequest));`

From the specification, the only way that the intruder has to commit a request is to bypass the CSRF protection (i.e., CSRF Token). To model that the intruder wants to submit a request that an honest agent does not, we have introduced a particular request (`intruderRequest`) within the Session entity as follows:

```
59  body { %% of the Environment entity
60    role1->can_exec(request1);
```

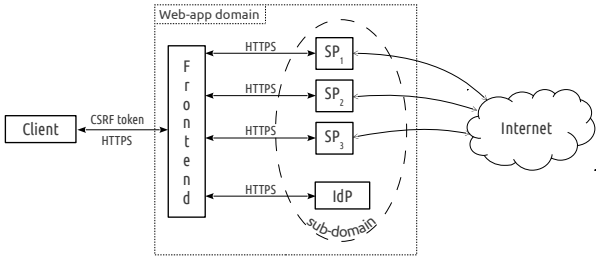

Fig. 2. Application scenario - Example

```
61    role1->can_exec(intruderRequest);
62    role2->can_exec(request2);
63    any UserIP. Session(UserIP, usr1, role1, request1) where !dishonest(UserIP);
64    new         Session(i     , usr2, role2, request1);
65  }
```

Validation. The AVANTSSAR platform has reported that the DocumentRepository specification is safe with respect to the modeled goal. This means that, given a bounded number of sessions, and considering a DY intruder, in this modeled scenario the three model-checkers on which the AVANTSSAR platform relies on have not detected a CSRF.

2.4 A more complex example

In this section we present an example, depicted in Fig. 2, to motivate the usage of our technique for more complex architectures. Our aim is to show that, after abstracting away unimportant implementation details, with our modeling technique it is possible to identify CSRF at design phase. We show a design schema of a Service Oriented Architecture (SOA) named Arch1. Arch1 (and a SOA in general) is a distributed system that offers functionalities that are not all hosted within the same server but are distributed on various hosts. Usually, a common interface (e.g., the AVANTSSAR platform web interface available from `www.avantssar.eu`) is offered to the client for communicating with the system. The architecture structure is then hidden to users who see the SOA as if it were a client-server architecture. It is also a common design choice to provide APIs for directly communicating with one (or a subset) of the services of the SOA so that expert users can develop their own client (or use a customized one if provided). Languages such as WSDL [4] are widely used in SOAs for this purpose.

Architecture description. Arch1 architecture is mainly composed by four parts: Client, Frontend, Service and Identity Providers (from now on SP and IdP respectively).

- the *client* entity represents the client browser. It can only communicate with the frontend via HTTPS, i.e., a secure channel: authenticated and encrypted
- the *frontend* entity provides a common interface to communicate with the system. To avoid CSRF, for each HTML page loaded by the client a CSRFToken (Sect. 2) is attached by the frontend; the client will attach it to its request for guaranteeing freshness. Upon the receipt of the correct message, composed by request and token, the frontend will forward the request to the correct SP

– *SPs* are the core of the system and provide the services of the SOA
– *IdP* is the entity that handle the client authentication

The frontend, SPs and IdP are all within the same web application domain, i.e., they are grouped in sub-domains that refer to the same domain. As a last remark on the architecture, each SP is also accessible from outside the SOA, and a client can directly communicate with a SP from the Internet.

The scenario we have modeled starts with an authentication phase in which the client logs in using SAML-SSO protocol. After that, it can submit requests to the system by communicating with the frontend that acts as a proxy between the client and SPs. We have also modeled, as motivated in Sect. 2.3, that if the client receives from another (dishonest) entity a request for the Arch1 system, his browser attaches the Cookie and sends the request. This behavior represents that the user clicks on a malicious email sent by a dishonest agent trying to exploit a CSRF on the system.

Authentication phase. As already stated in Sect. 1, HTTP(S) is a stateless protocol and then Cookies are used to ensure authentication. To store a Cookie the client has to authenticate with the SOA. We have chosen one of the state-of-the-art authentication protocols, SAML-SSO[9] but we could have used OpenID or OAuth obtaining the same behavior at this level of abstraction. SAML-SSO uses an IdP to authenticate the credentials (e.g., username and password) given by a client. Here the client can only communicate with the frontend and then the frontend will act as a Proxy between the client and the IdP. The client provides his credentials to the frontend that forwards them to the IdP. The IdP, after validating the client's credentials, creates a Cookie that is sent back to the Client via the frontend. Now the Cookie is stored within the client's browser used and it will be attached to every request he will send to the SOA (because every part of the SOA is inside the same web domain).

Honest client behavior. Once authenticated, the client has a Cookie stored in his web browser. He loads a web page in which a CSRFToken is provided and he sends the request together with the token and the Cookie to the frontend. The frontend forwards the message to the correct SP that, through the frontend, will communicates the result of the commitment of the client's request. An SP will not directly check the Cookie of the request but will ask the IdP to check if the Cookie is a valid one.

Arch1 ASLan++ model and validation. Due to page limit we cannot report the entire ASLan++ model but it follows the structure of the running example of Sect. 2.3.

We have used the AVANTSSAR platform for the verification of the specification obtaining the following attack trace.

```
MESSAGES:
1. frontend *->* <client> : n78(Csrftoken)
2. <?> ->* frontend : Client(80).Cookie(83).Req(83).Csrftoken(84)
3. frontend *->* <sp> : Client(80).Cookie(83).Req(83).Csrftoken(84)
4. <frontend> *->* sp : Client(80).Cookie(83).Req(83).Csrftoken(84)
5. <?> -> client : intruderreq.sp
6. client ->* <sp> : client.client_cookie.intruderreq
7. <?> ->* sp : client.client_cookie.intruderreq
```

We have assumed the Client has already logged in to the web application. In message 1, where the brackets $<\ldots>$ denote the intended communication partner of whose identity the honest communication agent cannot be sure, the frontend sends the CSRF Token to the Client (after having checked the Cookie of the Client). In message 2 the Client sends an honest request to the frontend and in messages 3 and 4 the frontend forwards the request to an SP. In message 5 the intruder sends an email containing a malicious link to the Client with a dishonest request for a SP. The Client clicks on it and in message 6 the request is sent directly to the SP. In message 7 the intruder request is committed and a CSRF is performed.

The attack trace shows that the modeled architecture Arch1 is vulnerable to CSRF. It is clear that, even if protections against CSRF have been (correctly) implemented, the manual detection of CSRF is a difficult and time consuming task. Our technique has permitted the automatic detection of CSRF in a complex architecture as Arch1 is. This extends the AVANTSSAR platform functionalities to check for CSRF.

3 Case studies and results

In this section we present results of applying our approach to three case studies. We have used the AVANTSSAR platform [2] to carry out the case studies and for the validation of the CSRF goal.

DocumentRepository. The AVANTSSAR platform model checkers conclude that the specification (described in Sect. 2.3) is safe with respect to the CSRF goal (i.e., no attack trace has been found). This means that the CSRF protection (i.e. CSRF token) cannot be bypassed, in the modeled scenario (i.e., with a bounded number of sessions), by the DY intruder [6]. This result, which has been confirmed by our industrial partner, shows that the combination of SSL and a CSRF token do not permit the intruder to attack the web application using a CSRF.

WebAuth. Authors in [1] have developed a methodology to detect web vulnerabilities using formal methods. In their most extensive case study, WebAuth [11], they show how they have found a CSRF. In order to compare [1] with our methodology we have then chosen to model the same case study.

WebAuth is an authentication system for web pages and web application developed by Stanford University. It consists of three main components: User Agent (UA), WebAuth-enabled Authentication Server (WAS) and WebAuth Key Distribution Center (WebKDC). The first time a UA attempt to access a web page he is redirected to the WebKDC that will ask to the UA for providing his authentication credential. An encryption identity is given to the UA. Now, the UA can use his new encrypted identity to obtain the web pages he wants to browse. The UA identity is stored in a Cookie that the browser will "show" to the WAS in a user transparent way so the UA will simply browse the pages while the browser will use the Cookie to retrieve them.

The result of the analysis shows a flaw in the authentication protocol, rather than a CSRF, in which the intruder convinces the UA to be communicating with the WAS while the UA is communicating with the intruder. In fact, in the attack reported in [1]

the token that has to be shown to the WAS in order to retrieve the service is the same used to start the authentication procedure and, due to the stateless property of HTTP, the WAS cannot detect if the two are different.

In order to detect a CSRF, we started from the protocol specification of WebAuth where the possibility of adding a CSRF token is not mandatory nor excluded. We have then modeled two versions: one with CSRF token exchange and the other one without. The model checkers return "NO_ATTACK_FOUND" (that means the specification is safe with respect to the CSRF goal defined) if the token is present, otherwise they report a CSRF as in the attack trace that follows.

```
MESSAGES:
[...]
18. UA(121) *->* <Actor(121)> : n119(Cookie).intruderRequest
19. <UA(121)> *->* Actor(121) : n119(Cookie).intruderRequest
20. Actor(121) *->* <UA(121)> : intruderRequest
```

From message 1 to message 17 there are the needed interactions between the UA and the system in order to obtain the Cookie. Message 18 shows that the intruder sends to the UA a malicious message. In the real case it would be an email with a link containing a request that can be executed only from UA that has access to the system. The intruder is not logged in to the WebAuth application but he wants the honest agent AU to execute the action. In messages 19 and 20 the UA follows the link and then his browser automatically adds the Cookie to the request hidden in the link. In message 21 the system replies with an acknowledge of the execution of the request. We can conclude that with our technique it has been possible to detect both an authentication flaw of the protocol underling the web application and CSRF, while the two were confused in [1].

EUBank. We have analyzed a web application of one of the major European bank searching for CSRF. We have manually analyzed the web application and in particular the money transfer part. The scenario we have modeled can be summarized by the following steps:

1. *Login phase:* a client logs in the web application by providing two numerical codes, a client id and a numerical password over an HTTPS communication
2. *Bank transfer set up:* the user fills in a form with all the necessary data for committing a bank transfer. Once committed, the web application asks for a confirmation. The user has to provide a numerical code that he can retrieve from his EUBank Passa hardware key that displays a numerical code freshly generated every sixty seconds.
3. *Bank transfer conclusion:* after checking the numerical code insert by the user, the web application sends to the client a confirmation page with all the details of the bank transfer

It is important to highlight that all the communication between the client and the server (bank) goes through a secure HTTPS channel, and even if no CSRF token is generated from the web application the EUBank Pass code is used also as a CSRF token

The first model we have implemented follows exactly the steps above, with the assumption that the client has his own EUBank Pass and no CSRF has been detected

from the AVANTSSAR platform model checkers. Modeling a scenario in which an intruder has obtained the EUBank Pass key (e.g., through social engineering [12]), we obtain an attack trace reporting a CSRF on the web application. We have not reported it for lack of space and because it is very similar to the abstract attack trace of CSRF of the WebAuth case study.

We have also manually tested it by transferring money from an EUBank account to another (of a different bank) simulating a CSRF exploitation. We have reported the attack and the bank has confirmed it.

4 Related work

There exist several works that aim to perform model-based testing of Web applications, e.g., [5,1,13]. In particular we want to compare our techniques with works that consider CSRF vulnerabilities.

In [1], authors have presented a (formal) model-based method for the verification of Web applications. They propose a methodology for modeling Web applications and the results of the exploitation of the technique on five case studies, modeled in the Alloy [8] modeling language. Even if the idea is similar, they have defined three different intruder models that should find Web attacks while we have used the standard DY intruder. Also, the detailed way they have used to define the Web application models results in attack traces which are difficult to interpret. In contrast, we have chosen to abstract away from implementation details creating a more abstract modeling technique to easily define a Web application scenario, thus more amenable to human interpretation. The ASLan++ language has permitted us to use the AVANTSAR platform [2] (a state-of-the-art formal analysis tool) and to obtain human-readable attack traces. As a final remark, we have also showed in Sect. 3 that authors in [1] have not found a CSRF on their most extensive case study, confusing an authentication flaw for a CSRF.

Another work close to ours is [3], in which authors have presented a tool named SPaCiTE that, relying on a model checker for the security analysis that uses ASLan++ specifications as input, generates potential attacks with regard to common Web vulnerabilities such as XSS, SQL-i an access control logic flaws. However, they have not explored CSRF.

5 Conclusions

In this work, we have shown that a model-based technique for detecting CSRF related vulnerabilities is feasible and can be of help in complex web applications, by leveraging existing symbolic techniques under the Dolev-Yao adversary models.

In future work, we plan to investigate how to model further web vulnerabilities for the detection of more complex attacks. This is not a trivial task, in fact, the required level of details needed for modeling a specification for the detection of other vulnerabilities, e.g. XSS (Cross-Site Scripting), has a strong impact on the performance of the model checking techniques available.

Acknowledgments

This work was partially supported by the FP7-ICT-2009-5 Project no. 257876, "SPaCIoS: Secure Provision and Consumption in the Internet of Services".

References

1. D. Akhawe, A. Barth, P. Lam, J. Mitchell, and D. Song. Towards a Formal Foundation of Web Security. In *Computer Security Foundations Symposium (CSF), 2010 23rd IEEE*, pages 290–304, 2010.
2. A. Armando, W. Arsac, T. Avanesov, M. Barletta, A. Calvi, A. Cappai, R. Carbone, Y. Chevalier, L. Compagna, J. Cuèllar, G. Erzse, S. Frau, M. Minea, S. Mödersheim, D. Oheimb, G. Pellegrino, S. Ponta, M. Rocchetto, M. Rusinowitch, M. Torabi Dashti, M. Turuani, and L. Viganò. The AVANTSSAR Platform for the Automated Validation of Trust and Security of Service-Oriented Architectures. In C. Flanagan and B. König, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 7214 of *Lecture Notes in Computer Science*, pages 267–282. Springer Berlin Heidelberg, 2012.
3. M. Büchler, J. Oudinet, and A. Pretschner. SPaCiTE – Web Application Testing Engine. In *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on*, pages 858–859, 2012.
4. E. Christensen, F. Curbera, G. Meredith, S. Weerawarana, et al. Web Services Description Language (WSDL) 1.1, 2001.
5. A. C. Dias Neto, R. Subramanyan, M. Vieira, and G. H. Travassos. A Survey on Model-based Testing Approaches: A Systematic Review. In *WEASELTech '07*, pages 31–36. ACM, 2007.
6. D. Dolev and A. Yao. On the Security of Public-Key Protocols. *IEEE Transactions on Information Theory*, 2(29), 1983.
7. A. Doupé, M. Cova, and G. Vigna. Why Johnny Can't Pentest: An Analysis of Black-Box Web Vulnerability Scanners. In C. Kreibich and M. Jahnke, editors, *Detection of Intrusions and Malware, and Vulnerability Assessment*, volume 6201 of *Lecture Notes in Computer Science*, pages 111–131. Springer Berlin Heidelberg, 2010.
8. D. Jackson. *Software Abstractions: Logic, Language, and Analysis*. MIT Press, 2012.
9. OASIS. Profiles for the OASIS Security Assertion Markup Language (SAML) V2.0. Available at `http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=security`, 2005.
10. OWASP. OWASP Cross Site Request Forgery. `https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF)`, 2013.
11. R. Schemers and R. Allbery. WebAuth v3 technical specification. `http://www.webauth.stanford.edu/protocol.html`, 2009.
12. T. Thornburgh. Social Engineering: The "Dark Art". In *Proceedings of the 1st Annual Conference on Information Security Curriculum Development*, InfoSecCD '04, pages 133–135, New York, NY, USA, 2004. ACM.
13. T. Tidwell, R. Larson, K. Fitch, and J. Hale. Modeling Internet Attacks. In *Proceedings of the 2001 IEEE Workshop on Information Assurance and security*, volume 59, 2001.
14. D. von Oheimb and S. Mödersheim. ASLan++ — a formal security specification language for distributed systems. In B. Aichernig, F. de Boer, and M. Bonsangue, editors, *Formal Methods for Components and Objects, FMCO 2010, Graz, Austria*, volume 6957 of *LNCS*, pages 1–22. Springer, 2010.