Computación y Sistemas
ISSN: 1405-5546
computacion-y-sistemas@cic.ipn.mx
Instituto Politécnico Nacional
México

Ferman, Victor; Hutter, Dieter; Monroy, Raúl
A Model Checker for the Verification of Browser Based Protocols
Computación y Sistemas, vol. 21, núm. 1, 2017, pp. 101-114
Instituto Politécnico Nacional
Distrito Federal, México

ISSN 2007-9737

A Model Checker for the Verification of Browser Based Protocols

Victor Ferman[1], Dieter Hutter[2], Raúl Monroy[1]

[1] Tecnologico de Monterrey, Escuela de Ingeniería y Ciencias,
Atizapán de Zaragoza, Estado de México,
Mexico

[2] German Research Center for Artificial Intelligence, Universität Bremen,
Germany

{vferman,raulm}@itesm.mx, hutter@dfki.de

Abstract. A browser based protocol is the chief security component of a safety critical web application, such as e-banking. Accordingly, browser based protocols need to be thoroughly verified in order to guarantee they are up to comply with key security properties. To this end, we have developed WebMC, a model checker especially designed to consider web standards, with the aim of analyzing browser based protocol execution, as encompassed by the interactions of a typical user, a browser, and active attacker playing the role of the network, and one or more servers. In this paper, we shall show how to use WebMC in the design and the development of browser based protocols. Our tool has been successfully validated: WebMC has been able to reproduce a number of the verification results found in the literature, but fully automatically.

Keywords. Model checking, browser based protocols, security protocols, formal methods.

1 Introduction

Web applications permeate our everyday life. We use them for a lot of different activities, ranging from financial services, like e-banking, to containerization in IaaS platforms, like Docker. Web applications have become of paramount importance, mostly because they are easy to use; easy to deploy to large numbers of users; involve a small cost; and above all, because they run on a browser, which yields a small footprint on the client side. Due to web penetration, it is of the utmost importance that we address the security of web applications; this includes guaranteeing that sending critical data through any of these kinds of applications is secure, even if the associated servers fail, or even if a server gets compromised by an intruder.

Proving security properties of *browser based protocols*, the key components of secure web applications, has been largely ignored. This is in contrast to its counterpart, proving correctness of security protocols, for which there exists several tools involving a large degree of automation (see [12], for a survey), and using either of various approaches, including formal methods [15]. While such tools offer a good starting point for the security analysis of browser based protocols, they are not enough because, in comparison to security protocols, browser based protocols involve more complex behavior, and more complex message structure. Further, tools for general software analysis require specific machinery and proof methods that are not suited to the verification of browser based protocols, because they often require the complete exploration of the state space to reach an output decision.

Security analysis of browser based protocols is complex, error-prone and difficult to automate. It has to take into account issues that are beyond the scope of tools for security protocol analysis. Among other things, this involves accounting for the effect of browser policies on information security guarantees; the implications of browser using of frames, running scripts, and administering cookies; the intricate interactions

that may arise amongst a user, the associated browser, the network, and the servers running the protocol under analysis; and the behavior of a complex protocol, which often leaves the user with incomplete knowledge of what is being executed and of the information being exchanged.

WebMC is based on Internet standards. The design of it has taken inspiration from OFMC [2]. WebMC is able to represent widespread attacks, such as cross site scripting (XSS), cross site request forgery (CSRF), and session fixation. It can also automate attacks that require the reuse of data and cookies contained in messages, as well as the behaviors that may arise from the interactions of the different participants in a browser based protocol.

In this paper, we discuss the way in which WebMC can be used to analyze the security guarantees provided by a given browser based protocol.

The rest of this paper is structured as follows. We start in Section 2 describing the main attacks to which browser based protocols are vulnerable to and which we are interested in automating. We continue reviewing related work in Section 3, and then giving an overview of WebMC's inner workings in Section 4. Then, we provide an overview of WebMC's implementation, Section 5. After that, we present, in Section 6, a full example of how WebMC can be used in order to specify and verify a browser based protocol. Then, in Section 7, we report on the results of a successful validation of WebMC; we shall see that our tool has been able to reproduce a number of the results found in the literature, but fully automatically. Finally, we conclude paper and pose ways in which WebMC can be extended in Section 8.

2 Attacks on Browser Based Protocols

In this section, we shall outline the attacks on browser based protocols that WebMC aims to automate. These attacks are all very relevant, in terms of prevalence, and have caused already in countless loses.

Cross Site Scripting (XSS) is an attack in which a malicious script is injected into a web page that is in control of an honest server. This attack is used to either make transactions in the name of a user, steal user data or both. By contrast, *Cross Site Request Forgery* (CSRF) is an attack in which the user accesses an attacker controlled web page, and in which the attacker tries to take advantage of the user's interactions and of the information inside the browser. CSRF tries to make requests in name of a user. However, notice that the attacker does not know if the user already has an active session with the honest service; nor is it able to know the information the browser has about said honest service. XSS and CRSF differ subtly: the main difference is that in CRSF one server is fully controlled by the attacker, while both the server and the service remain uncompromised, and thus the attacker cannot directly access any of the information under the browser's or the server's possession.

Cookies are a means in which a servers store critical information in a browser. They are used in order to keep a consistent state, given that a user may access the corresponding service via a multitude of browsers, from a range of different IP addresses, and using any of a number of devices. However, a major drawback of using cookies is that an attacker may tamper with the data inside them. Cookie tampering severely affects protocol and session management, and may lead to an insecure state, where a user impersonates other or even the attacker, or where a user sends unintended information across several sessions. There are two kinds of successful cookie attacks, depending on how far, fully or partially, an attacker has been able at making an honest user impersonate it. Upon full impersonation, the attacker is capable of successfully modifying part of an honest web page, leaving the user none the wiser. Hence, cookie attacks rely on two things: one is that the attacker has a means to retrieve information from an honest server (*e.g.* by having a valid account with the service), and the other is that the attacker is able to inject information (*e.g.* a cookie) to a browser without user intervention.

WebMC is able to account for the lot of interactions that occur in the execution of a browser based protocol, as compound by the actions of four nominated, individual components, namley: a user, the user browser, the network (in our

case, the attacker), and the server running the protocol. To automate the attacks presented in this section, WebMC models HTTPS request and responses, browser policies, cookie management, script execution, resource usage and resource visibility, and it models an attacker who owns the network and so is able to corrupt servers.

3 Related Work

Several mechanisms for the security analysis of browser based protocols can be found in the literature [8, 9, 4, 1, 14, 7, 5, 6, 11, 10]. Most of them present severe drawbacks. This is either because the mechanism ignores relevant details of a browser specification, hence yielding an oversimplification [8, 1], or because it does not abstract away intricate details of IETF or W3C specifications, hence becoming unsuitable for automation. In what follows, we outline those mechanisms that have been integrated into an automated tool.

Information flow analysis [1, 14, 7, 5] has been used in order to prove confidentiality of the information exchanged in a browser based protocol. In particular, Armando *et al.* [1] have used the SAT Model Checker (SATMC) to analyze a set of LTL formulas verifying the Google version of the SAML protocol. While SATMC has been able to find an attack on this version of SAML, protocol specification is not really browser based, for it does not consider the key characteristics of a browser. Accordingly, in subsequent papers, Armando *et al.* have attempted to overcome this limitation, improving on their formalism to include, for example, cookies, and so have been able to represent more kinds of attacks. However, doing so requires reworking the models for each new protocol and property to be verified.

In a different vein, Kumar [10, 11] has argued that belief logics, not needing an explicit attacker, are more suitable to analyze browser based protocols. Kumar has proposed an extension of BAN, which he has used, along with models encoded in Alloy, in order to prove security goals. While Kumar states that his approach simplifies the models of the protocol components: user, browser, and servers, he does not explain how this simplification or encoding should work. The simplification reduces in about 60% the complexity of the models, thus speeding up the Alloy analysis. In order to test his approach, Kumar analyses the SAML identity linking protocol.

WebMC can be used to automatically verify the security of a given browser based protocol. The inner workings of it rely on a state event system, which we outline below.

4 WebMC: Internal Workings

Analyzing whether or not a given browser based protocol satisfies a security property amounts to analyzing that the property holds in every state reached by all possible traces of the protocol. In our approach, a step of a protocol trace is a result of an attacker action or the interaction of two out of four types of participants: a user, a browser, the network, and servers. Each participant is modeled as a state event system; they are all composed together into yet, another state event system (see Figure 1), where it is possible to use a branching-type logic.

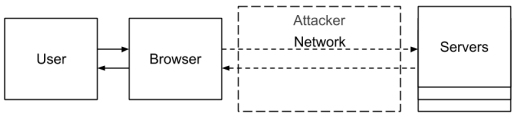

Fig. 1. The Models Used by WebMC

Our approach to verifying a browser based protocol takes three inputs, namely: a protocol specification, the initial knowledge of the participants, and the property the protocol is meant to satisfy. Both the protocol specification and the knowledge are used to instantiate the complete behavior of the server, the attacker, and the user, respectively. The underlying mechanism of WebMC uses the state event system to generate and then analyze the protocol, following an on the fly strategy; the analysis is driven so as to identify a state that violates the given security property. As expected, WebMC can take a property other than one of security; this is useful, for example, to

prove that a protocol description is feasible in that it accepts a complete run.

Four key components of WebMC's underlying mechanism are the behavioral models of the user, the browser, the network, and the server. In our approach, the user participant is key to protocol analysis: it is taken to drive the overall execution of the protocol, and, so, is modeled so as not to be vulnerable to a phishing attack. The browser participant models a simple browser, which is able to: communicate with the user via a display; construct a web page from server's responses; execute simple scripts; follow policies; and store information. The server participant is parametric, since it is instantiated by the input protocol specification; however, it contains general rules to act upon and keep track of protocol execution. Finally, the attacker participant, which controls the network. This model is also parametric, since, as shall be discussed later on in the text, we may like to conduct protocol analysis under the presence of an attacker that is able to hold its own sessions with one or more servers.

WebMC is hence able to account for the lot of interactions that occur in the execution of a browser based protocol, as compound by the actions of the four nominated, individual components: the user, the user browser, the network (in our case, the attacker), and the server running the protocol. To automate the attacks presented in Section 2, WebMC's internal machinery handles relevant issues of browser based protocols, including HTTPS request or response handling, browser policy application, cookie management, script execution, resource usage, and resource visibility; the analysis includes the intervention of an attacker who owns the network and is able to corrupt servers.

In what follows, we complete our discussion of WebMC's inner workings outlining protocol execution state, event transition rules, what counts as a protocol and a protocol goal, and the attacker.

4.1 The System's State in Our Tool

As we have said, our tool poses an state event system which consists of parametric models of a web user, its browser, the attacker, and a variety of servers. The state of the system is then the collection of the states of its components and the intended goal. We must remark that browser based protocols have one important difference with other protocols, that is, the participants excluding the attacker may not change its roles in subsequent runs of a protocol. The state of the user, the browser, the servers, and the attacker consist of data important for their function.

For example, the browser, as we know, takes inputs from users and servers, and transforms these inputs in things the other party can understand (*i.e.* transforms user inputs into requests and responses into a display and maybe some more requests). The current state of the browser is then given by a mix of the previous state, the user's input, and the responses received until this point in time. As we know files, web pages, and scripts are not static and thus are updated with each input and output in the browser. It is because of this constant change that the information that the user and the servers get is just a subset of all the information possessed by the browser at any given time.

4.2 Interactions among Participants

WebMC uses a deterministic state event system; however, several events are able to occur at any one time. As we have said, each component of our system (*i.e.* user, browser, servers, attacker) is an state event system by itself and since we are analyzing protocols, transitions cannot be seen as independent actions performed by a single participant in a protocol. An event in our method is the reception or sending of a message by a participant. Communication between participants is done by sharing the same events, that is, an input-output event can occur if there are two participants willing to perform the complementary actions.

We must remark that there are states in which a participant may be able to perform several synchronizations one after another with one or more participants. Further, as is common when dealing with parallel processes after two participants perform a synchronization the others can act independently without also synchronizing.

However, we just have mentioned that principals may not change roles as such the communication between processes is restricted by the following:

Definition 1. *Let $\mathcal{U}$ represent the user process, $\mathcal{B}$ the browser process, $\mathcal{A}$ the attacker process, $\mathcal{S}^j$ a server process, X, Y represent any two principals among the already mentioned and* sort *be a function that returns a set of events a given principal can perform then:*

$$\begin{aligned}
\mathrm{sort}(\mathcal{U}) \cap \mathrm{sort}(\mathcal{B}) &= \mathrm{sort}(\mathcal{U}),\\
\mathrm{sort}(\mathcal{B}) \cap \mathrm{sort}(\mathcal{A}) &= \mathrm{sort}(\mathcal{A}),\\
\mathrm{sort}(\mathcal{S}^j) \cap \mathrm{sort}(\mathcal{A}) &= \mathrm{sort}(\mathcal{S}^j),\\
\mathrm{sort}(\mathcal{U}) \cap \mathrm{sort}(X) &= \varnothing \; where X \in \{\mathcal{A}, \mathcal{S}^1, \ldots, \mathcal{S}^n\},\\
\mathrm{sort}(\mathcal{B}) \cap \mathrm{sort}(X) &= \varnothing \; where \; X \in \{\mathcal{S}^1, \ldots, \mathcal{S}^n\},\\
\mathrm{sort}(X) \cap \mathrm{sort}(Y) &= \varnothing \; where \; X \in \{\mathcal{S}^1, \ldots, \mathcal{S}^n\}\\
&\quad \wedge Y \in \{\mathcal{U}, \mathcal{B}, \mathcal{S}^1, \ldots, \mathcal{S}^n\}/X,
\end{aligned}$$

wherein an empty intersection denotes that communication cannot happen between the two principals.

WebMC is then in charge of taking all of the principal instances and advancing the execution of a protocol. This is done by synchronizing the principals using complementary actions (*i.e.* sending-receiving a message with certain characteristics), then updating the state of the principals and the system by applying the corresponding rules, and finally by interleaving these synchronizations among principals. This means, that the system interleaves the execution of asynchronous events generated by the principals in order to create execution traces of protocols.

In principle, parallel composition of the state event systems allows for the synchronization of more than two principals with a single action; however, our tool only takes into account one user, one browser, one attacker, and several servers. While there may be several servers, each message within events states the intended destination, which means that a server will and must ignore all messages not directed at it thus preventing multiway synchronization from occurring.

4.3 A Protocol in WebMC

Let us continue by defining how the different states the system can reach are related. Let e denote an event, $\mathcal{E}$ represent a set containing all such possible events; τ then represents a grounded execution trace in the system, and $\mathcal{T}$ represents a set containing all such execution traces. We define an execution trace of the system $\tau \in \mathcal{T}$ as a sequence $[e_0, e_1, \ldots, e_n]$ with $e_k \in \mathcal{E}$, $\forall k \in \{0, \ldots, n\}$.

Definition 2 (Protocol)**.** *A protocol P is a subset of system traces each of which is denoted with a p.*

A protocol run p is then a grounded trace that is an element of P. Traces are deemed to be left-complete, *i.e.* each left prefix of a trace is a trace. The main difference between τ and p traces is that τ traces may not conform to any given protocol as defined by its specification. We can say that, for a user in our system, a protocol is a set of all runs and each run is given by a p trace. We must note that this definition can be naturally extended to all of the components of our system.

Whether a given trace belongs to a protocol depends on the protocol specification; however, given a protocol specification WebMC is unable to generate events that lie outside said specification. We must also note that since WebMC calculates on the fly each and every trace of the system our rules and transitions can be said to behave in branching time, as such, the set P can also be visualized as a tree or a graph.

4.4 Goals in WebMC

In WebMC security is mapped to a goal. Goals are expressed as a list of events that should never happen and yield an state, if it concerns a reachability goal then the tool searches for the occurrence of an event and its resulting state within all protocol traces; however, if the goal concerns security the tool searches for the existence of such event and thus an state that should never happen within a protocol trace p. For example, if we want to assure that a protocol is able to terminate the goal would correspond to the last message of said protocol or if we want to assure that a

term guarantees authentication the goal would be represented as a list containing a message where such term is being reused by a party. As we have said messages are components of events and contain an origin, a destination, and a payload.

4.5 The Attacker in WebMC

Now that we have given a description of how the system works we will proceed by briefly discussing the attacker. That is, we will describe its state, its capabilities, and its relation to the other parties. As in any work of this kind, the network is an insecure channel through which information or more precisely messages flow from one participant to another. While using this channel the participants may encrypt some or all of their messages with either a symmetric or asymmetric key, and thus while the contents may remain hidden from an attacker, the attacker can still capture and may be able to reuse these messages.

In order to represent different kinds of attack we model two kinds of attacker and two kinds of server corruption. *Full corruption* in which the attacker has complete control over a server, and *Partial corruption*, where the attacker may add data and instructions to messages but cannot directly access or retrieve the information contained in the messages.

The attacker is parametric in that its state contains several data structures that need to be specified by the end-user. The attacker state contains a unique identifier; a couple of sets of server identifiers denoting the servers that are either fully or partially under the attacker's control; a set of public, private, and symmetric keys either known or obtained during a protocol's run; a set of fields which values will be generated on-demand (*e.g.* nonces); a pair of association lists that contain information the attacker owns and knowledge about all of the other principals; a set of files that the attacker has acquired; and a pair of lists representing request and response queues.

The first type of attacker in our tool is a somewhat simple attacker, this attacker can corrupt servers, analyze, separate, and concatenate messages as well as to encrypt, sign, and decrypt messages to which he possesses the adequate keys; however, it may not hold sessions in his name with different servers. In other words, the main goal of this attacker is to observe, gather, include pertinent data and instructions in the messages that it has access to, and see if the data it has gathered is enough to construct an attack on either the user or on one of the honest servers. The second type of attacker is a fully active attacker. This second type of attacker is able to do the same as the previous but it does not have the restriction on being able to have its own sessions, thus it is also able to create, send, and receive messages to and from both servers and browsers.

As we can see, the actions the attacker is able to perform closely follow the actions of the Dolev-Yao attacker. The actions of the Dolev-Yao are simple yet powerful enough to describe what both attackers need to do; however, in order to better understand and characterize the actions an attacker may perform in WebMC we give the attacker's actions new names that describe better their interactions with the system and bundle some of them together, when required, to make the process easier to understand as well as accessible for the automation of attacks.

The attackers are then able to perform at the following actions while still conforming to the expected messages of a protocol.

- Add cookies to messages it has access to.
- Add visible and invisible elements to messages it has access to.
- Add instructions to messages it has access to.
- Get knowledge from corrupted servers.
- Encrypt, sign, and decrypt information it has access to with known keys.
- Synthesize messages from knowledge.
- Act as the initiating party to any protocol.
- A combination thereof.

In order to limit the size of the search space, the Attacker is only able create messages that correspond to a protocol. Messages outside of the protocols are deemed to be useless since the servers just send error responses and just increase the size of an already infinite search space. Also, as can be seen from the actions we described, we removed the ability of the attacker to flush the

channel and thus will not model or analyze any kind of denial of service attacks.

In the simplest of terms the attacker is a buffer, which in the case of the second type is also able to send and receive its own messages as long as it has the necessary information to create and read them. An example of the attacker's behavior is the following: If the attacker receives a message whose intended destination is different from one of the fully corrupted servers the attacker will just add it to the corresponding queue. On the other hand, if the intended destination of the message is a fully corrupted server, the attacker will take all of the information and add it to its knowledge about the principal who originated the message.

5 The WebMC Implementation

The main use for WebMC is to try to solve the *Security Problem*, the *Intuder Deduction Problem*, and a third problem that arises from the characteristics of the browser (*i.e.* the attacker being able to use secrets that are not known to him by using the Browser as an intermediary). In other words, our tool is able to encode properties relating to secrecy and confidentiality in order to find counterexamples to the presented goals.

WebMC is thus an state exploration tool that takes a protocol specification, and a set of properties that should never hold (*i.e.* events representing attacks or flaws in authentication). The tool lets us to either interactively traverse all of the possible protocol execution paths or automatically find attacks using the two kinds of attacker. WebMC, tries to find counterexamples to the defined goals or properties (*i.e.* states where the attacks are feasible) by calculating, on the fly, all of the possible execution paths of the protocol or application.

5.1 Protocol Specification in WebMC

A server and protocol specification is a set of rules written as a Haskell program that define the behavior of the server. The specification must contain at least the name of the server, the expected URLs, and what messages are to be sent upon receiving a request to a given URL. The specification may also contain data and keys known at the start of a run (*e.g.* credentials and trusted server keys), which data is to be kept for persistent sessions, which information is to be freshly generated, which nonces and information should be tracked to avoid replays, and what messages are to be sent in case of receiving an unexpected or invalid message.

We must note that neither the method nor the tool are able to verify if a given server specification corresponds to an actual protocol but is able to verify that a protocol is able to finish successfully if the end user states explicitly that the goal is to search for protocol completion. WebMC is able to check for well-formedness according to data types; however this does not guarantee that the protocol is well specified. This means that WebMC may get stuck on a loop and never find an attack if not tested correctly. This is why protocol analysis should start with proving the feasibility of protocol execution successful.

Now that we have discussed protocol specification let us continue with the implementation of WebMC. WebMC is implemented in two ways the first of them, to be discussed in Section 5.2, is a module in charge of generating the search space on the fly and allows the end users to explore interactively the different ways in which a protocol's execution can take place while our second implementation is an automatic tool which will be discussed in Section 5.3.

5.2 Interactive Tool

The interactive implementation of WebMC was programmed so the end user of the tool is able to select the branch to be analyzed and go back to another branch if she chooses to do so. In other words, the interactive WebMC presents all of the possible actions to the end user, lets her choose which to execute, and is also able to present the current state of the different principals. Interactive WebMC has been tremendously useful since it has let us explore the execution of the protocols we are specifying, find ways in which the search for counter examples can be structured, and debug the features included in the implementation of our models.

We decided to model the interactions between the different agent as events that are fully executed when received, this in order to avoid having wait states in which nothing evident happened. However, for the sake of clarity our implementation we decided to include the changes the attacker makes to messages as its own set of actions, allowing us to see better what are the steps needed in order for attacks to be successful.

Now that we have explained the characteristics of our interactive tool we will proceed to explain the implementation detail of the automatic version of our tool.

5.3 Automatic Search

While proving security in WebMC is roughly equivalent to encoding security properties in a set LTL formulas and then proving these formulas hold; we took a more straightforward and slightly different path. As we have said, the properties WebMC looks for are encoded as attacks and, in order to avoid the complexities arising from more than one event being able to occur at any one time, we use an heuristic to select and analyze only one branch at a time. Our tool calculates the execution tree on the fly since calculating said tree at the start is not feasible due to its size.

The search in the automatic version of WebMC is implemented as a deterministic hybrid search with no hard maximum depth level. WebMC searches the state tree as long as it has not found any attacks and as soon as an attack is found the execution trace is returned to the end user. The evaluation function penalizes repeated events and paths that lead to useless states (*e.g.* the user and in turn the browser sending an empty request to a server that does not handle empty requests) while trying to lead the search in a way that resembles the normal protocol execution and giving preference in the following order: first the events of the attacker, then those of the servers, the browser, and finally the user. This means that we have a preference for the attacker's actions while taking into account the state of all the other participants and the characteristics marked as valid for the attacker, as such, the heuristic usually leads to a normal execution of a protocol but does not guarantee that if there is an attack it will find the shortest attack trace.

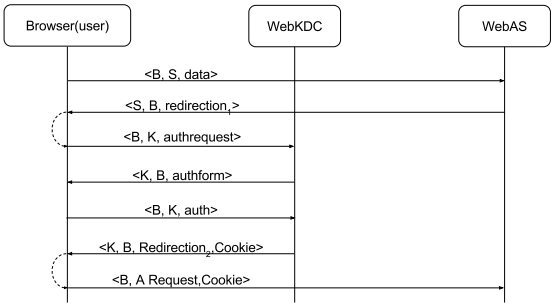

Fig. 2. The WebAuth protocol

Regarding the performance of the search we consider it to be good since it takes less than one second to find most of the attacks to the protocols we tested; however, it's performance for worst case scenarios could be made better by detecting and avoiding the exploration of states equivalent to those already explored (a technique used in Binary Decision Diagrams [3]).

6 Protocol Testing and Verification with WebMC

So far we have discussed protocol specification and how WebMC searches for attacks; however, this is not enough to give a full idea of the way in which it can be used. As such, we will now proceed to give a detailed example of how a protocol is specified and how the tool should be used. To do so we will discuss the WebAuth Protocol, what we consider to be a flaw in its specification, and how it was specified for WebMC.

WebAuth (see Figure 2) is an authentication protocol for web pages in which the first time a User attempts to access a service, they will be sent to an authentication server and prompted to authenticate. Once the user has logged in, the authentication server will send their encrypted identity back to the original web page and the identity will also be stored in a cookie set by the authentication server. The

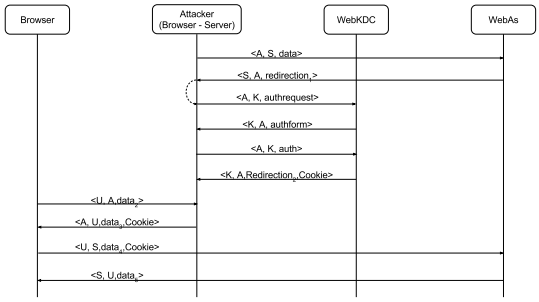

Fig. 3. Attack to the WebAuth protocol

user will not need to authenticate again until their credentials expire, even if they visit multiple protected services. The attack to WebAuth (see Figure 3) consists on reusing the cookie provided by the authentication server in order to make an honest user impersonate the attacker and use services in its name; as we can see, the *authentication* posed by the protocol compromised and thus an attack is possible.

As we can see the attack is pretty straightforward but it has been overlooked, thus highlighting the need for a tool that is able to automatically find these kinds of attacks. With that said, we will now proceed to give a detailed description about how we can model this and other protocols in our tool.

6.1 Protocol Specification in WebMC

As we have mentioned previously, specifying the servers is equivalent to specifying the protocol. In this section we will discuss how the WebAuth protocol is specified. For specifications to work we will use a header, like the one in Listing 1, containing the name of the file (usually corresponding to the name of the protocol to be specified), and the different libraries or modules we want to use. In this case we will use WebKereberos as the name for our file and protocol, the Haskell Map library, the Attacker and Server modules of WebMC that provide the implementations of the attacker and the server models respectively and the Types module that corresponds to the data type definitions of WebMC.

Listing 1. Header for specifications, including the name of the protocol to specify

```
module WebKereberos where
import qualified Data.Map as Map
import Server
import Attacker
import Types
```

After providing the header we need to proceed to specify the servers. As we can see in Figure 2, the WebAuth protocol requires three kinds of servers. The first kind of server corresponds to a Service Provider (webAS) in charge of providing a service to the user, the second kind corresponds to an Identity Provider (webKDC) in charge of asserting the identity of users so that the service knows who is using it, and a third server fully controlled by the attacker (aServer).

The code inside Listing 2 corresponds to the specification of the webAS server specification. In the case of our tool, the specifications are functions that receive a server name and return an instance of the corresponding server. For a server to be instantiated it receives the following information in order: its unique identifier, the values which will be generated automatically, a description of the persisted data to store, a list of fields which contain keys, a list of fields to keep track of so as to not accept repeated values, a list of known keys, an association list of known information, and an association list of recognized URLs and the corresponding actions that should take place. The full specification of each server comes after the where clause. In case the server being specified does not need to generate automatic values (*e.g.* nonces), keep track of persistent data, keys or previously used values, etcetera. We leave some of fields empty in order to instruct our tool not to use said characteristics. Since servers do know information and need to respond to certain messages we fill the known data and rule fields corresponding to the last two arguments of the instantiation code.

Usually, the known data fields correspond to things like known user names, passwords, URLs, and other data that needs to be verified upon arrival. On the other hand the rules field corresponds to an association list of URLs, the different fields they expect on requests, the requests and responses that should be made if

Listing 2. Specification for the Service Provider Server in WebAuth

```
webAS:: String -> String -> Server
webAS cName kdc =
    initEmptyServer cName [] ('''', []) [] [] [pkey] Map.empty ruleMap
    where kdcUrl = Url kdc ''one''
          pkey = Pub kdc
          url1 = Url cName ''one''
          cbUrl = Url cName ''two''
          inst1 = Instruction (Right True) Rule { rType = RuleType Normal Full, rMethod = Post,
                    rUrl = Left kdcUrl, rContents = Map.singleton ''cbUrl'' (show cbUrl) }
          response1 = Response {destinationIdentifier = ''', origin = url1, resNonce = ''',
                      csp = emptyCSP, componentList = [], fileList = Map.empty,
                      instructionList = PageInstructions { autoList = [inst1], conditionalList = [] }}
          inst2 = Instruction (Left 1) Rule { rType = RuleType Normal Full, rMethod = Post,
                    rUrl = Left url1, rContents = Map.singleton ''success!!'' ''?'' }
          component2 = Component { cOrigin = url1, cList = [inst2], cPos = 1, cVisible = True}
          response2 = Response {destinationIdentifier = ''', origin = url1, resNonce = '''',
                      csp = emptyCSP, componentList = [component2], fileList = Map.empty
                      instructionList = PageInstructions { autoList = [], conditionalList = [] } }
          inst3 = Instruction (Left 1) Rule { rType = RuleType Normal Full,
                    rMethod = Post, rUrl = Left cbUrl, rContents = Map.singleton ''success!!'' ''?'' }
          component3 = component2 {cOrigin = cbUrl, cList = [inst3]}
          response3 =  response2 {origin = cbUrl, componentList = [component3] }
          ruleMap = Map.fromList [ (url1, [(['']id_token''], [], response2, Just response1),
                                           ([], [], response1, Nothing)]),
                                   (cbUrl, [(['']id_token''], [], response3, Nothing) ]) ]
```

the server where to receive a valid request to said URL, and an error response in case the request is not valid. In the case of our servers these rules are located at the end of the definitions and are constructed by using all of the information (*i.e.* request, responses, components, instructions, and known information) declared previously.

6.2 Goal Specification in WebMC

After defining the behavior of the servers in the protocol, we need to provide the security goals and define the attacker to be used by our tool. To do so we use the $\mathrm{getServers}$ and $\mathrm{secondGoal}$ functions, as in Listing 3. The $\mathrm{getServers}$ function takes no argument and returns a list of servers, a security goal in the form of a list of requests and responses, and the instance of the attacker to be used by the tool when searching for the attacks. While the $\mathrm{secondGoal}$ function takes an initial state and based on that returns a new state under which the search is to continue in order to find the attack.

As we can see in Listing 3, we are instantiating one webKDC server, one webAS server, one aServer server, the attacker, and defining the goals. The servers are instantiated by calling the previously crafted server specifications, while we construct the goal with a request saying that the attacker should send a valid request to the webAS server. Finally, we instantiate a type two attacker (by telling it that it may create new messages with the True flag). The attacker also possesses a list of the servers it has fully corrupted (the aServer), a list of servers it has partially corrupted, a list of all servers participating in the protocol, a list of fields it can generate automatically (*e.g.* nonces), an association list of known or acquired files, and its knowledge (in this case a valid account at the webKDC server). After the $\mathrm{getServers}$ function we specify another function which will be called after the first goal is reached. The $\mathrm{secondGoal}$ function adds information to the user, changes the attacker to a type one attacker, and defines a new goal consisting of injecting the cookie to the user and the user being able to access the original webAS using the attacker identity.

6.3 Creating an Entry Point for WebMC

Once the servers, the goals, and the attacker have been specified we can continue by creating the entry point of our program. The explanation of this entry point will be divided in two parts presented in Listings 4 and 5. The first part, in Listing 4, corresponds to the header. In the header

Listing 3. Sever and Attacker instantiation, specification of the goals for the WebAuth protocol

```
getServers :: ([Server], [Either Request Response], Attacker)
getServers = ([kdc, was, aServ], goals, myAttacker)
    where kdc = webKDC ''kdc''
          was = webAS ''was'' ''kdc''
          aServ = aServer ''att''
          cbUrl = Url { server = ''was'', path = ''two'' }
          aKnown = Map.fromList [(''user'', ''uname''), (''pass'', ''pass''), (''cbUrl'', show cbUrl) ]
          rPayload1 = Map.fromList [(''id_token'', '''')]
          req1 = Request { originIdentifier = ''attacker'', destination = cbUrl, reqNonce = '''',
                   method = Post, payload = rPayload1 }
          goals = [Left req1]
          myAttacker = initAttacker ''attacker'' True [''att''] [] [kdc, was, aServ] [] Map.empty aKnown

secondGoal::State -> State
secondGoal cState = nState
    where cUser = user cState
          cAttacker = attacker cState
          cGoals = mGoals cState
          url1 = Url { server = ''was'', path = ''one'' }
          aUrl = Url { server = ''att'', path = '''' }
          kUrls = [aUrl]
          req2 = Request {originIdentifier = ''browser'', destination = aUrl, reqNonce = '''',
                   method = Get, payload = Map.empty }
          rPayload = Map.fromList [(''id_token'', '''')]
          req3 = Request {originIdentifier = ''browser'', destination = url1, reqNonce = '''',
                   method = Post, payload = rPayload }
          nUser = cUser {knownUrls = kUrls}
          nAttacker = cAttacker { asSessions = False }
          nGoals = Left req2:Left req3:cGoals
          nState = cState {user = nUser, attacker= nAttacker, mGoals = nGoals}
```

of our entry point we should include the search heuristic to be used or if the program is to be interactively executed (in this case we are using the hybrid search), the browser module to be used, the protocol specification to be used, the user model to be used, the data types of our method, and finally some libraries that help us measure the execution time of our program.

Listing 4. Header for entry point file

```
import          BFTest
import          Browser
import          Criterion.Measurement
import          Data.Functor
import          Data.Map                as Map
import          WebKereberos
import          Types
import          User
```

The second part of our entry point program, as presented in Listing 5, is composed of the user knowledge we want to use, the instantiation of the browser we want to use, and the call to the search heuristic (or the interactive version) we want to use. After writing the entry point and the protocol specification we just need to compile the entry point, and execute the resulting file in order for the search to start.

6.4 How Results are presented by WebMC

Finally, after compiling our code, we can execute the program which will output either a trace that leads to an attack (as presented in Figure 3) like the one in Listing 6 or a message saying that it reached its iteration limit and that no attack was found for the protocol being analyzed.

In the case of the WebAuth protocol we present in Listing 6 an excerpt of the output generated by our tool. This excerpt serves as a sample of what is to be expected from our program; however, we must mention that since we are using a heuristic in order to search all of the possible executions for an attack the tool does not usually return the shortest attack trace but first one it found. In this case, the trace presents the same steps needed by Figure 3 in order to reach the point in which the user is able to send the attack message.

Now that we have concluded our explanation of how our tool works, how protocols are specified,

Listing 5. Body of the program in charge of executing and defining User Knowledge

```
main:: IO ()
main = do
    putStrLn ''Welcome''
    (pFlag, pState) <- loopState iState
    if pFlag
       then do
            putStrLn ''''
            putStrLn ''Continuing with second goal''
            loop (secondGoal pState)
       else putStrLn '':(''
    secs <\$> getCPUTime >>= print
    putStrLn ''''
    putStrLn ''Bye!''
    where url1 = Url { server = ''was'', path = ''one'' }
          aUrl = Url { server = ''att'', path = '''' }
          kUrls = [aUrl, url1]
          myUser = initUser ''user'' Map.empty Map.empty kUrls
          myBrowser = initEmptyBrowser ''browser''
          (myServers, goals, myAttacker) = getServers
          iState = State myUser myBrowser myServers myAttacker goals []
```

and how protocols are instantiated in order to be analyzed; we will proceed to discuss some experimental results of using WebMC.

7 Experimental Results

So far we also have analyzed and reproduced the results of [1] in which is reported an attack to a version of the SAML protocol, found two attacks for the OAuth 1.0 protocol that had been reported by security researchers like the one in [13], and found what we consider to be a vulnerability on the WebAuth protocol (a version of the Kereberos protocol for the web). As we can see in table 1, our tool and method are able to reproduce several kinds of attacks previously reported by security researchers in both theoretical and practical scenarios. We also report the time it took for our tool to find each of the attacks, when the attacks require characteristics not present on one of the attackers we report NA and when the tool was not able to find an attack in less than a day either due to the characteristics of the attack or the size of the search space we simply report an NF.

The attack to SAML (see figure 4) consists on reusing the token provided by IdP in order to use a different service; as we can see, the *secrecy* of the assertion is compromised and thus an attack is possible. The attacks on the Oauth 1.0 protocol

Table 1. Experimental results

	Time to Find the Attack	
Protocol	Type Two Attacker	Type One Attacker
SAML	<1s	<1s
Oauth 1.0 (Attack 1)	<1s	NA
Oauth 1.0 (Attack 2)	<1s	NA
WebAuth		
full	NF	NA
Sub Goals	<2s	
Session Establishment	<1s	NA
Session Fixation	NF	<1s
SAML-Fix	NF	NF
OpenID 1.0	NF	NF

rely on the fact that tokens are not scoped by the specification. While the first Oauth attack obtains a new token and uses it to perform an arbitrary action, the second obtains an already valid token and uses it to perform an arbitrary action. As we can see, the problem with Oauth is that tokens can be freely obtained, and thus, do not offer any kind of security beyond a weak authentication also broken by the second attack. The vulnerability we found in the WebAuth protocol relies on the fact that cookies are used for user authentication, which means that if an attacker possesses valid credentials for the Kereberos/Authentication server then it could inject cookies with its identity to honest users. Finally, we found no attacks to the proposed fix to SAML protocol and OpenID 1.0 if

Listing 6. WebMC Sample Output for the WebAuth protocol attack

```
Attacker Send [] to Url {server = ''was'', path = ''one''} with attacker
was -> attacker: Response nonceattacker1
Attacker Pass Response: was -> attacker
Attacker Send [''cbUrl''] to Url {server = ''kdc'', path = ''one''} with was
kdc -> attacker: Response nonceattacker2
Attacker Send [''user'',''pass'',''cbUrl''] to Url {server = ''kdc'', path = ''two''} with attacker
Attacker Pass Response: kdc -> attacker
Attacker Pass Response: kdc -> attacker
Continuing with second goal
U -> B: Send_Url Url {server = ''att'', path = ''''}
B -> att: request
Attacker Pass Request: browser-> att
att -> browser: Response noncebrowser1
Attacker Add Instructions: att -> browser
Attacker Add Cookies: [Right (Url {server = ''kdc'', path = ''''}),
  Right (Url {server = ''was'', path = ''''})] to att -> browser
Attacker Pass Response: att -> browser
B -> was: request

''1.359 s''
Bye!
```

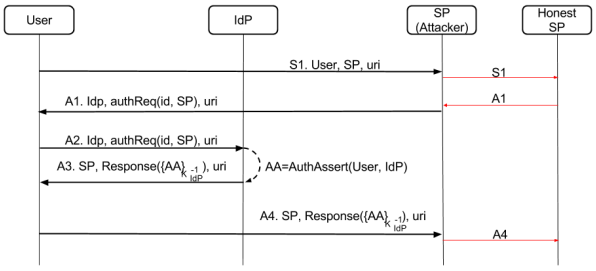

Fig. 4. The SAML protocol and an attack

formalized as described without adding or leaving out any of the required functionality described in the specifications.

We can see in Table 1 the results of our experiments and would like to point to an interesting result. While the tool is able to find an attack to the WebAuth protocol it would take too long to find it if it were to be specified with a single goal, this happens due to the characteristics of our heuristic and the great amount of messages the attacker can construct. In order to find this vulnerability we separated the search in two; the first, using a type two attacker, in order to prove that an attacker with valid credentials could get a valid identity cookie; and the second, using the type one attacker, in order to limit the interactions of the attacker and to prove that the cookie could be injected and then used by an unsuspecting victim.

8 Conclusions and Future Work

Browser based protocols, as expected, share countless similarities with other protocols and applications; however, in the case of browser based protocols the participants in them do not have a complete understanding of what is the correct information flow or know about all of the data being transfered from one participant to another. Because of these differences is that we need a model tailored specifically for them. A tool the one presented would not only lead to a better understanding of browser based protocol properties, allow us to analyze potential security vulnerabilities and shortcomings said protocols may have, but also aid in the development of new policies and capabilities for browsers and servers since testing their security would require little modification to the tool.

Tools encoding formal models of existing software tend to be somewhat incomplete since it would be impossible to formalize all of the existing characteristics and interactions that arise in our everyday world; however, we have identified some ways in which WebMC can be expanded, and enriched. We consider the following features the most important to be added:

— Accessing files within the host computer
— Modifying previous instructions
— Calculating values to be sent
— Inter-Frame communication
— A language for the specification of protocols, applications, and security properties

With this said, the results of using WebMC are promising, and we are working towards the formalization of other protocols and finding new attacks. We aim at making protocol verification an integral part of the design process in order to create better applications and protocols that take security as one of their foundations.

Acknowledgements

The research reported here was supported by CONACyT - DFG Grant 121596, Google Faculty Research Awards, and Tecnologico de Monterrey & CONACyT studentship 239453 to the first author.

References

1. **Armando, A., Carbone, R., Compagna, L., Cuellar, J., & Tobarra, L. (2008).** Formal analysis of saml 2.0 web browser single sign-on: breaking the saml-based single sign-on for google apps. *Proceedings of the 6th ACM workshop on Formal methods in security engineering*, FMSE '08, ACM, New York, NY, USA, pp. 1–10.
2. **Basin, D., Mödersheim, S., & Viganò, L. (2005).** Ofmc: A symbolic model checker for security protocols. *International Journal of Information Security*, Vol. 4, No. 3, pp. 181–208.
3. **Bryant, R. E. (1986).** Graph-based algorithms for boolean function manipulation. *IEEE Transactions on computers*, Vol. 100, No. 8, pp. 677–691.
4. **Fett, D., Kusters, R., & Schmitz, G. (2014).** An expressive model for the web infrastructure: Definition and application to the browser id sso system. *Security and Privacy (SP), 2014 IEEE Symposium on*, pp. 673–688.
5. **Gajek, S., Manulis, M., Sadeghi, A.-R., & Schwenk, J. (2008).** Provably secure browser-based user-aware mutual authentication over tls. *Proceedings of the 2008 ACM symposium on Information, computer and communications security*, ASIACCS '08, ACM, New York, NY, USA, pp. 300–311.
6. **Gordon, A. D. & Pucella, R. (2005).** Validating a web service security abstraction by typing. *Formal Aspects of Computing*, Vol. 17, No. 3, pp. 277–318.
7. **Groß, T. (2003).** Security analysis of the saml single sign-on browser/artifact profile. *Proceedings of the 19th Annual Computer Security Applications Conference*, ACSAC '03, IEEE Computer Society, Washington, DC, USA, pp. 298–307.
8. **Groß, T., Pfitzmann, B., & Sadeghi, A.-R. (2005).** Browser model for security analysis of browser-based protocols. *Proceedings of the 10th European conference on Research in Computer Security*, ESORICS'05, Springer-Verlag, Berlin, Heidelberg, pp. 489–508.
9. **Groß, T., Pfitzmann, B., & Sadeghi, A.-R. (2005).** Proving a ws-federation passive requestor profile with a browser model. *Proceedings of the 2005 Workshop on Secure Web Services*, SWS '05, ACM, New York, NY, USA, pp. 54–64.
10. **Kumar, A. (2012).** A belief logic for analyzing security of web protocols. **Katzenbeisser, S., Weippl, E., Camp, L. J., Volkamer, M., Reiter, M., & Zhang, X.**, editors, *Trust and Trustworthy Computing: 5th International Conference, TRUST 2012, Vienna, Austria, June 13-15, 2012. Proceedings*, Springer, pp. 239–254.
11. **Kumar, A. (2014).** A lightweight formal approach for analyzing security of web protocols. In **Stavrou, A., Bos, H., & Portokalidis, G.**, editors, *Research in Attacks, Intrusions and Defenses*, volume 8688 of *Lecture Notes in Computer Science*. Springer International Publishing, pp. 192–211.
12. **Monroy, R. & Lopez Pimentel, J. C. (2008).** Formal support to security protocol development: A survey. *Computación y Sistemas*, Vol. 12, No. 1, pp. 89–108.
13. **Muthiyah, L. (2015).** *How I Hacked Your Facebook Photos - Deleting any photo albums*. Blog.
14. **Pfitzmann, B. & Waidner, M. (2001).** A model for asynchronous reactive systems and its application to secure message transmission. *Proceedings of the 2001 IEEE Symposium on Security and Privacy*, number May in SP '01, IEEE Computer Society, Washington, DC, USA, pp. 184–200.
15. **Viganò, L. (2006).** Automated security protocol analysis with the avispa tool. *Electron. Notes Theor. Comput. Sci.*, Vol. 155, pp. 61–86.

Article received on 17/10/2016; accepted on 02/11/2016.
Corresponding author is Victor Ferman.