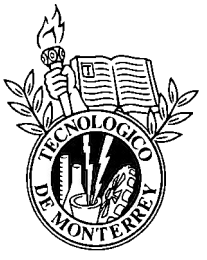
INSTITUTO TECNOLÓGICO Y DE ESTUDIOS SUPERIORES DE MONTERREY
CAMPUS ESTADO DE MÉXICO



TECNOLÓGICO DE MONTERREY®

WebMC: A Model Checker for the Web

A thesis submitted in fulfilment of the requirements
for the degree of Doctor in Computer Science

Author: Victor Ferman

Advisor:	Dr. Raúl Monroy	
Advisor:	Dr. Dieter Hutter	
Thesis Committee:	Dr. Guillermo Morales Luna	Examiner
	Dr. Juan Carlos Lopez Pimentel	Examiner
	Dr. Luis Ángel Trejo Rodríguez	Examiner
	Dr. Raúl Monroy	Examiner
	Dr. Dieter Hutter	Examiner

Atizapán de Zaragoza, Estado de México
3rd December, 2016

Declaration of Authorship

I, Victor Ferman, declare that this thesis proposal titled, 'WebMC: A Model Checker for the Web' and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed: _____

Date: _____

Abstract

WebMC: A Model Checker for the Web

by Victor Ferman

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, to deploy to large numbers of users, they are cheap, and above all, because they have a small footprint on the client side due to running above browsers. Due to the ubiquitous nature of the web, it is of the utmost importance that we address the security of web applications; this includes guaranteeing that sending critical data through these 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 [1], for a survey), and using either of various approaches, including formal methods [2]. 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.

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 human nature of the end-user; the effect of browser policies on information security guarantees; the implications of browser using of frames, running scripts, and managing cookies; the intricate interactions that may arise among all of the participants in 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 [3]. 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.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Information security is concerned with the way in which information and systems are protected. It is commonly said that information security has three main components namely: confidentiality, integrity, and availability. Modern information and computer systems are said to be secure if they assure these three conditions; however, this is definition gives an incomplete view of security since we have to take into account the sometimes conflicting needs of the different stakeholders of the systems as in multilateral security. According to [4] integrity is a requirement that information and programs are changed only in a specified and authorized manner or that a system performs its intended function in an unimpaired manner, free from deliberate or inadvertent unauthorized manipulation of the system. Confidentiality is defined as a requirement that private or confidential information not be disclosed to unauthorized individuals. Finally, availability is a requirement intended to assure that systems work promptly and service is not denied to authorized users.

The web is one of such system that requires information security to be held. Web applications are ubiquitous in our everyday life, we use them for everything from the common, like web-mail, to information sensitive activites, like banking, and even to communicate and manage instances of IaaS platforms like Docker and Amazon's AWS that hold information all kinds of information about its users. Web applications are becoming more and more important because of their ease of use, their ease of deployment to large numbers of users, their relatively small cost considering the amount of people they can be deployed to, and above all, their use of the browser as a platform and thus their small footprint on the end-users' systems. Due to the ubiquity of web applications it is of the utmost importance that we think about their security and that of the data we send through them by using the corresponding browser based protocols. That is, We need to ensure that data is secure in case a server is attacked or fails, or even if the browser is being attacked by an intruder that is able to fully control the network (Dolev-Yao) even when considering perfect cryptography.

Research on securing different systems, protocols and applications is vast and focuses on different aspects of what means for a system, program or protocol to be secure. Extensive testing is one of these ways; however, testing tends to be incomplete and it is not always able to deal with errors originating from the design of the applications and protocols. Other, more formal, approaches like theorem proving and model checking have been devised (for a survey see [1]). These formal approaches have led to the development of several tools, like [2], for the automated verification of software and protocols.

While security analysis of browser based protocols and web applications is complex, error prone, and difficult to automate; there are several ways in which securing the information in browsers, web applications and browser based protocols has been attempted. This automation requires taking into account several things like a user with incomplete knowledge of the protocol and information being exchanged by the other participants; the effects of browser policies on the behavior and security of information or protocols; the use of browser characteristics like frames, scripts, and cookies by the web applications; the interaction of the different participants in web applications (*i.e.* user, browser, the network, one or more servers) and the protocols or applications being run. Additionally the problem is made harder by the existence of sophisticated attackers that can corrupt participants in order to take advantage of the mentioned characteristics.

Existing tools for the verification of software and security protocols are a good starting point for the research on the security of browser based protocols and web applications; however, we find they are too limiting to represent browser based protocols and web applications since standard tools for model checking software tend to be limited in the amount and dynamics of behaviors they analyze, and in the structure and contents of the messages and events they are able to represent. Moreover, these tools require specific machinery and proof methods that are not always adequate to deal with the problem at hand (*e.g.* they may require the complete exploration of the state space).

It is because of the previously mentioned shortcomings that security verification methods for browser based protocols like [5] and [6] have been created. While these approaches are at almost opposite ends of exhaustiveness regarding the characteristics of the web included neither of them has been made into an automated tool, the latter method, as the authors admit, due to the complexity of dealing in an automated manner with all the characteristics included in the models used by the method they propose.

1.1 Goals And Scope of this Work

Due to the characteristics and limitations of existing approaches we consider that there is an opportunity and a need for a new approach that is detailed yet still amenable to automation. We also pose that due to the amount of characteristics shared between browser based protocols and web applications they can be thought to be equivalent and thus analyzing one is the same as analyzing the other. With this in mind, we decided to develop both a method and a tool for verifying browser based protocols and web applications, terms to be hereinafter used interchangeably with *protocols* and *web protocols*. The results of our research are compiled and presented in this work, specifically we present WebMC a method and a tool able to analyze protocols and detect the effects of widespread attacks like cross site scripting (XSS) and cross site request forgery (CSRF).

Our goal is to is to automatically verify the security of web protocols, as such, we designed WebMC, a method and a tool based on standards that formalizes and models the structures and behavior of the different participants in web applications. We must note that web protocols are designed to work above other protocols like those in layer 7 of the OSI model. To be more specific we want to analyze protocols that work over HTTPS (HTTP+TLS or SSL) and leave aside behaviors arising from usecured HTTP and DNS. With this said, our model and application also take into account request and responses, browser policies, cookies, simple scripts, frames, element visibility; while leaving aside things like headers, inter-frame communication, complex scripts, and storage. The chosen characteristics are useful in order to represent the previously mentioned attacks; however, these characteristics are not able to represent all of the possible behaviors as we will discuss in Chapter 4.

In order to achieve our goal, WebMC includes models for a user, a generic server, a web browser, the network, the attacker, and a message theory. We also propose verifying these protocols in a similar way to OFMC [3] due top the way in which it manages infinite state spaces and searches for a counter example to a given security property. In contrast to other methods designed to verify protocols, WebMC aims to be more detailed than [5], enough to represent a wide range of behaviors, while not being as exhaustive as [6] thus losing the ability to analyze certain interactions like inter-frame communication.

1.2 Structure of this Work

So far we have discussed the aim and the importance of our work. In order to better guide the reader we will now conclude this chapter by giving an account of the contents of this work.

Chapter 1 is this chapter, the introduction, in which we state the importance of our work. It presents and gives the basic assumptions, goals and structure of this work.

Chapter 2 Gives an overview of the formal methods area and discusses in detail the approaches taken by the community in order to secure browsers, web applications and browser based protocols.

Chapter 3 Presents and gives a detailed look at widespread attacks to protocols. The chapter also discusses the characteristics both a method and an automated tool should include in order to be able to represent and analyze these attacks.

Chapter 4 Takes a look at the general characteristics of our method, explains our design choices, and discusses formally what is a protocol and a security property in our method as well as what we mean by security.

Chapter 5 Presents the first half of our model, that is, we discuss in detail Users and Browsers. We start by explaining what we consider to be a User of web applications and browser based protocols. Then, we provide a model for a User and give a look to its capabilities and how it interacts with the other participants in order to create a cohesive whole. After presenting the user we continue by presenting the model and behavior of the browser to be used by our method. We define what are web pages to our model and how they are constructed, how scripts work, how messages and information storage are represented, the browser's characteristics, how security policies work, and how the browser communicates with the user and the servers.

Chapter 6 Discusses the second half of our model. We begin our discussion by presenting the characteristics and capabilities of servers, the difficulties in creating a model that is able to represent all of the possible behaviors, how we solved this problem, how servers are deeply related to protocols and how server instances are able to interact with the browser and other server instances in our method. We continue by giving and in-depth look at the characteristics of the network and the attacker. We do so by providing a detailed account of what the attacker capabilities are, and how these capabilities are related to the Dolev-Yao attacker; this discussion also explores how servers are compromised and what it entails.

Chapter 7 Presents the implementation of our model, gives an overview of the design considerations and the inner workings of the tool, and concludes with a discussion about the experimental results obtained from the use of WebMC.

Chapter 8 Is the final chapter of this work, in which we present and discuss conclusions reached from the work we did and how it may be continued and enhanced in the future.

Chapter 2

Related Work

Our aim is to verify browser based protocols and web applications. So far we have talked about the importance of our work. However, our work is not isolated, and as such we shall describe its context so as to allow us to highlight the relevance of working in the subject. In order to be as clear as possible in regards to the context; we will first explore what has been done in the area of security protocol verification, due to the large amount of characteristics and work that is related to our own, then we will continue by exploring the approaches to secure both browsers and the protocols that use them as platforms.

There are many different approaches to determine whether a security protocol ensures its goal or not. Among all of the approaches to security protocol verification some (e.g belief logics, state exploration, and information flow analysis) are more relevant to our interests as they have been used to approach the verification of browser based protocols.

The specific problem of web protocol verification has been studied in two ways. One of these ways studies how browsers and their behavior can be secured, while the other deals with the interactions that arise between the participants of a protocol as it is being executed and how these interactions affect security.

This chapter will be structured as follows, we will first explore how the browsers have been studied and analyzed in order to be secured from attacks in Section 2.1. Then, in Section 2.2, we start by giving a general overview of the security protocol verification area, discuss some of the advantages and disadvantages of some of the approaches related to protocol verification, and after that we take a detailed look at the different approaches taken to solve the problem of browser based protocol verification. Finally, in Section 2.3 we conclude by summarizing the difference between our approach and that of others.

2.1 Research on Browser Behavior Analysis

As we have said securing the information that goes through a browser is an important research area. The research in this area can be divided in two categories: the analysis of browser behavior Section 2.1, and the verification of browser based protocols Section 2.2. In both categories; the approaches taken and the results are of restricted in that they tend to either simplify too much [5, 7], be too detailed [6] or too specific in regards to a given web browser implementation [8, 9]. In the case of the formal analysis of browser based protocols researchers usually either ignore the browser behavior, thus simplifying too much; or include too many details from IETF and W3C specifications, making the model too complex for automatic protocol verification.

Protocol verification usually leaves aside the behavior of the clients since the clients are made specifically for one or just a few protocols and thus their behavior does not change depending on what the contents of messages being received. However, in the case of web protocols this is not posible since browsers are not like any of the other clients and as such there is a full research area about their behavior. We can think about this research area as being at the other end of the protocol verification approach; that is, it tries to understand and to secure the behavior of a browser while executing applications while disregarding the fact that an application may be insecure in some other ways (for example the lack of encryption or the fact that some messages can be misunderstood or re-sent by someone else). More specifically research on browser behavior leaves aside the way the browser interacts with the servers. This research area focuses only on analyzing web specifications, their intended properties and the information flow within the browser, or pays attention only to the bugs in a given browser implementation. Browser analysis usually lacks a formalism or even a rigorous mathematical method, and focuses instead on the exploitation of vulnerabilities. This means that the approach is not concerned with a systematic way in which errors and vulnerabilities may be found or fixed, nor is it concerned with the analysis of attacks that happen outside of the browser itself.

The research in this area is often spread widely and is usually very practical in nature. We can see examples of research in the area in every-year's BlackHat or CanSecWest conferences, and in contests like "Pwn2Own" where a person or a group try to exploit previously unknown vulnerabilities on popular software like browsers. Some of the other work in the area has a more formal approach; however, it still lacks an analysis about how the data behaves. We must remark that due to the characteristics and focus of this research some of their results may be seen in places like the CVE [10] and blogs instead of an academic journal.

2.1.1 Script Hardening and Dynamic Code Analysis

The first aspect to securing browsers we are going to analyze is that of securing the execution of scripts. As we know, the behavior of the web browsers is outlined by a set of several standards (e.g HTML [11], HTTP [12], CSS [13], JavaScript [14], URI [15], etcetera) some of which are presented as RFCs, while others are just specifications given by a working group. In [16, 17] the authors describe how JavaScript and other features of the browsers can be often used to compromise the security of the users; The authors outline that some of the approaches taken by the people researching this area are often not enough since they make accessing web pages a little bothersome (*e.g.* by being off-line, increasing load times or breaking existing benign web applications).

The approach in [16] is one in which Heiderich *et al.* take into account the characteristics of JavaScript and use them to develop a tool, written in JavaScript itself. The proposed tool, IceShield, is one that freezes the JavaScript prototypes (classes), then does a dynamic code analysis, after doing so the tool, based on heuristics, decides whether the site (or some elements of it) may be malicious or not. Finally, the tool modifies the elements in the web page to prevent the attack from being executed.

In other words, Iceshield can be used to prevent malicious websites from performing attacks. According to the data presented on the paper it has very few false positives and false negatives. Also, one of the most noteworthy characteristics of the tool is that it should be compatible across browsers (given that said browsers implement the ECMAScript specification) and computer architectures since it is written in JavaScript.

Another approach that tries to deal with the same problem is FlowFox, a browser implementation presented in [17] where De Groef *et al.* describe how they use secure multi-execution (*i.e.* executing the program once for each of the defined security labels, each of which possess specific rules for input and output) in order to avoid data being leaked from one security level to another. In other words, FlowFox tries to control the information flow within the browsers and scripts while still being compatible (*i.e.* not changing or removing functionality) with existing web applications. However, these kind of approaches are criticized by [16] since they tend to usually incur in large overheads, break functionality, or be constrained to small subsets of the JavaScript specification.

As we can see both of these approaches have advantages and disadvantages. The main advantage is that they successfully avoid a lot of the possible attacks to web applications and browser based protocols that depend on the browser being vulnerable while the main disadvantage is that the approaches do not take into account what happens inside the servers and how the data received by them may be used with malicious ends. There are other disadvantages to these methods, in the case of the approach by Heiderich *et al.* said disadvantages include the existence of false

positives and that the approach may end up breaking some web pages while in the approach by De Groef *et al.* the disadvantages include the large overhead caused by tagging the information and running the scripts multiple times each time a web page is loaded.

2.1.2 Formal Specification of Browser Behavior

In [8] Bohannon *et al.* argue that browsers are complex since they are in charge of interpreting inputs, presenting results, executing cryptographic algorithms and protocols, keeping track of data, among other things. The authors also explain that there are several aspects that can be secured in web browsers that go from their implementation to the use of policies that restrict the behavior of the web pages being presented and pose that in order to understand and fully analyze all of the different security policies that work in the browser we need a formal specification of said policies and a formal model of a web browser.

Due to the needs and constraints presented in the paper, the authors also note that using a browser implementation (*e.g.* Firefox or Chrome) as the model is not a feasible option due to the complexity and since implementations are buggy, difficult to modify, and already contains an specific version of the policies that restrict behaviors in web pages. With that said, the authors present a model written in OCaml that does not try to fully emulate a browser. The aim of the presented model is to formalize the core functionality of web browsers (*i.e.* models the asynchronous nature of web browsers and covers the basic aspects of windows, DOM trees, cookies, HTTP requests and responses, user input, and a minimal scripting language with first-class functions, dynamic evaluation, and AJAX requests) and thus the model allows for the analysis of the interaction of scripts with the data and the browser itself.

While the model presented in [8] takes into account that the browser interacts with the network, it still incomplete as it leaves important things like defining and testing the different security policies that may be implemented by browsers as future work. The model also leaves aside how the information that goes through the browser may be used by an attacker and just focuses on how an attacker might use the scripts to interact with the data within the browser itself thus having the same weakness as the works previously discussed (*i.e.* it does not take into account how the data may be used by another party once it leaves the browser).

2.1.3 Security Assessment of Browser Implementations

Finally, to conclude our discussion on how browsers have been secured, we will discuss the approach taken by the authors of [9]. This approach is quite different from the previous ones since the authors try to assess the security of different browser implementations. The research in [9] bases its analysis on the premise that all applications contain bugs and thus the security

depends on the amount and quality of the different anti-exploitation mechanisms implemented by the different vendors.

In this document the authors claim that the anti-exploitation mechanisms are the most important metric for browser security because these mechanisms can hinder the success of the attacks by making necessary the use of a string of vulnerabilities in order to compromise the browser. The analysis of the different behaviors is done by using several tools and web pages designed specifically to find how the different architectures executed code and accessed information. The results from these experiments is a comparison and a ranking of the browsers according to the perceived level of security they provide.

As we can see research done in this aspect of web application security can be widely different from one another; however, the research shares the goal of securing the interaction of the users with the browser and the browser themselves. While the research in browser behavior is important it overlooks how information may be used by an attacker in the network even if the information and messages it can access are encrypted.

2.2 Verification of Browser-based Protocols

We will start our discussion by exploring the area of security protocol verification, to do so we will use the survey by Monroy *et al.* [1] where the authors state that there are two analogous approaches: the computational complexity approach and the formal methods approach. The former approach tries to find flaws in the cryptography of a protocol while the latter one assumes perfect cryptography. Obviously these two approaches encompass a variety of techniques, each of which has trade-offs. The survey reviews three approaches to formal security protocol verification. The first is based on belief logics, the second is based on state exploration and the third is based on theorem proving. The authors also state that the interest in security protocol verification comes from the fact that security protocols tend to be faulty and their flaws may go unnoticed to the creators for years giving a false sense of security, thus we have the need to create rigorous verification tools to assure that the information is safe.

2.2.1 Belief Logics

The belief logic approach was the first attempt to automate the verification of security protocols. One of the most notable belief logics is BAN-logic which allows for short, abstract proofs to be made but is unable to identify some protocol flaws, like the ones in the original version of the Needham-Schroeder key exchange protocol; however, when the proof fails there is no obvious way to create a counter example.

BAN logic allows to check the trustworthiness of exchanged information and whether said information is secured against eavesdropping. BAN needs the description of a protocol, and uses a set of axioms and a set of inference rules that allow to prove whether the participants beliefs about the protocols are correct or not and thus lead to proofs of attacks being possible. However, even if BAN logic finds that the beliefs hold it does not mean that no attacks are possible and thus some attacks may go unnoticed if not tested further.

2.2.2 State Exploration Techniques

The state exploration approach tries to explore as many execution paths of a protocol as possible, checking at each state if some conditions hold. The search space for protocol verification can be infinite because of the number of participants and the role each one takes in a particular run of the protocol; however, some of the state exploration tools, like OFMC [3], do not search in the whole space because of previous theoretical results and symbolic execution techniques.

2.2.2.1 Model Checking

There are several approaches to model checking but they all try to solve the problem of testing automatically whether a description (model) of a protocol complies with a given set of characteristics. One example of this approach is the one used by Lowe in which a protocol is modeled using CSP and then checked with FDR, another good example of this approach is that of AVISPA [2] with its many model checkers like OFMC [3].

CSP stands for Communicating Sequential Processes, it is a process algebra among many others like the Calculus of communicating Systems (CCS) proposed by Robin Milner in the 1980s, and the π-calculus proposed by Milner, Joachim Parrow and David Walker in 1990s. CSP was proposed by Charles Hoare in a 1978 paper. Later on Hoare also published [18], a book, in which he describes how to use CSP and gives mathematical proofs of its characteristics.

These approaches are useful because there are several tools that have been developed to automate the model checking; however, the difficulty lies in creating a good model of the protocol or system to be tested and when creating this kind of model we are required to assume that the system is programmed correctly and that different runs will not interact.

2.2.2.2 Information Flow Analysis

Information flow analysis is an approach where the main objective is to find if there is any way in which the information can be tainted or declassified when following its path through the system. In our case it would be akin to analyzing how the data travels from the user through the browser

and the channel to finally reach the servers and back. The information flow analysis approach is usually done in different ways, there is one that uses dynamic or static code analysis of the protocol or application that wants to be verified, and another that does this analysis based on a model of the system being analyzed. There are several examples of using these methods not only in the area of security protocol verification but also on the area of application verification. the main inconveniences of this approach are related to the large overhead when doing dynamic code analysis, not being able to check for all the interactions with static code analysis, and the difficulty in automating the verification of complex models.

2.2.2.3 Strand Spaces

The strand spaces method was developed initially by Herzog, Thayer and Guttman in [19] and it provides a way to represent simply and graphically the information flow of a protocol. This method allows us to represent several runs in parallel; however, one of its main disadvantages is that it cannot deal with protocols that might change its behavior programmatically, that is, depending on the messages sent and received by the principals.

2.2.3 Theorem Proving

The theorem proving approach tries to produce a formal proof from the description of the protocol, a set of axioms and a set of inference rules. This approach lets us lets us reason about the results of a finite sequence of actions and works by using either Higher-Order Logic or First Order Logic (see [20] for an example). However, using Higher-Order logic requires interaction with the theorem proving framework, while the First Order logic approach is more automatic and viable for generating counter-examples.

2.2.3.1 The Inductive Approach

This approach was suggested by Lawrence Paulson [21] and combines characteristics from state exploration and belief logics methods for protocol verification. The approach uses Higher-Order Logic and thus to prove that a protocol is indeed correct we must use the inductive method to prove it. The proofs for this method are usually performed with aid from the interactive theorem prover Isabelle.

One of the main advantages of this approach is that, since it is based on Higher-Order Logic, it allows us to express things clearly and concisely; however, one of the main disadvantages is that it requires user interaction to guide proof search.

2.2.3.2 First Order Logic (FOL)

There are many methods for protocol verification that use FOL. These methods usually require the protocols to be represented in a predicate language so that a tool like a theorem prover can be used to find if the protocol is correct. These methods have the same advantages that the Inductive approach albeit with limited expressive power and their main disadvantages are that they are not guaranteed to terminate and that they usually cannot offer examples of the attacks that make the protocols insecure (*i.e.* a counter example).

Now that we have finished our discussion about the research on browser behavior analysis we will continue by exploring the work related to the verification of protocols. As we have said browser based protocol verification is the other approach to secure our information. This problem has been studied before in numerous papers like [5–7, 22–26]; however, we find the work in the area can still be improved in several ways. In other words, we take the research in this area as our starting point, and as such, we will continue by giving a detailed account of the work in the area.

2.2.4 Protocol Verification Using Browser Models

We have stated previously that our work uses a model of the participants in web protocols in order to verify their properties, as such we will start our discussion on browser based protocol verification by discussing the work that is most related to ours (*i.e.* research that uses browser models in order to verify protocols).

2.2.4.1 A simple browser model

The first approach we will discuss is that of Groß *et al.* [5, 22] where the authors' goal is to verify the security of browser based protocols by using a browser model. In [5] the authors discuss security protocols and how the browser behaves in a different way than most security protocol principals; that means that since the browser does not know about the protocol that is being executed, the browser cannot be assumed to do something to assure that the data is secure. Then the authors give a detailed account of how the browser model was created, the model's shortcomings, its main components and how the components are related to finally show why a user authentication protocol is secure given that every part of the model behaves the way they say it should.

In Figure 2.1 we can see the model proposed by [5]. This model is basically a flow diagram where each one of the boxes represents a process containing input and output ports for communications. Each process is then modeled as an automaton (see an example automaton in Figure 2.2).

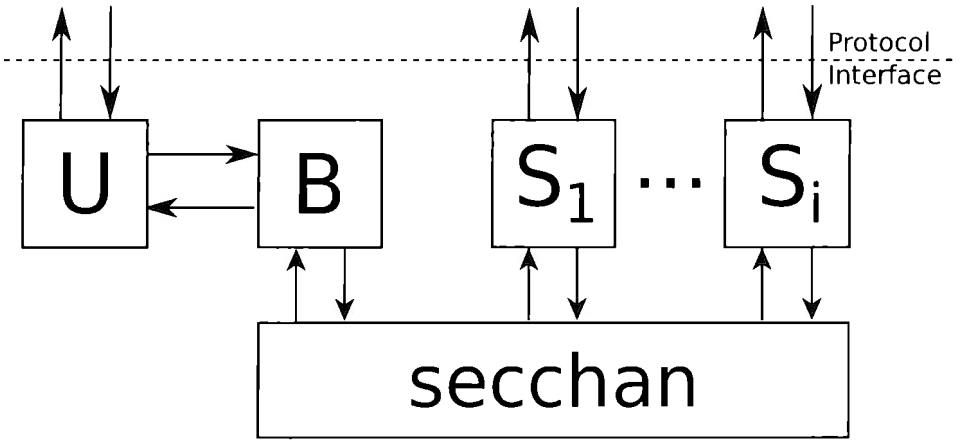


FIGURE 2.1: Model used in [5]

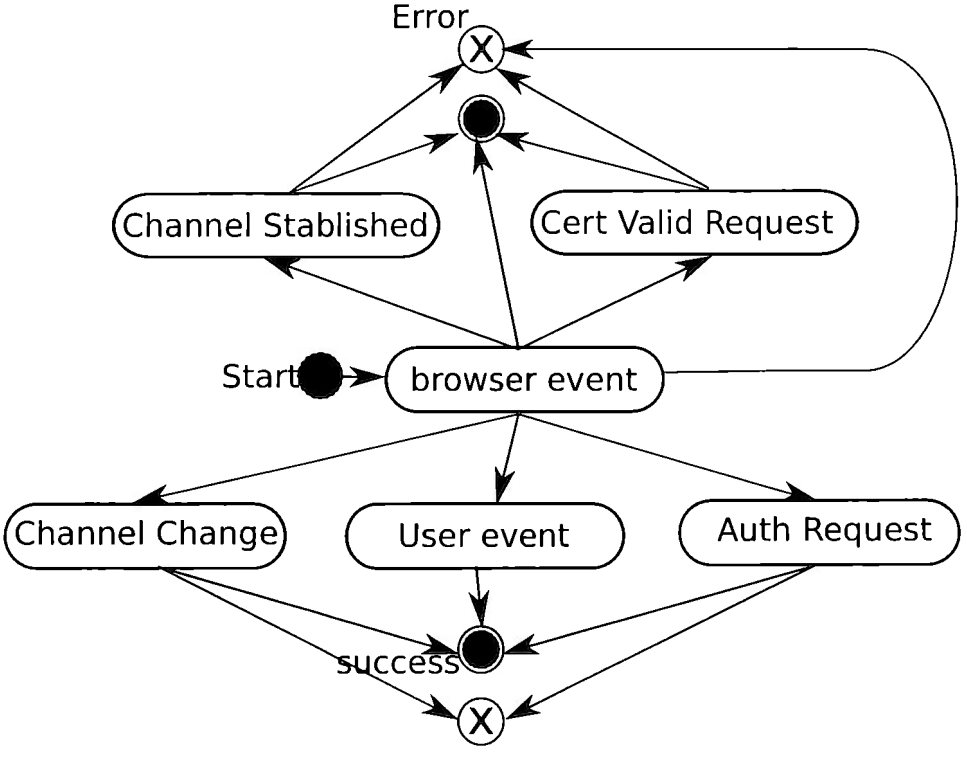


FIGURE 2.2: Automaton representing the model of the user, from [5]

In the model (Figure 2.1) the process **U** represents a user, **B** represents a browser, **Secchan** represents a channel and $\mathbf{S_1} \ldots \mathbf{S_i}$ represent the servers. As we can see the process of the attacker is nonexistent and even if we consider the attacker to be the channel the paper does not provide an automaton describing it. This lack of an attacker means that we cannot fully reproduce the way in which the process of the channel works and if the attacker capabilities they mention are enough or even equivalent to those of the attacker in the Dolev-Yao model.

The model is good since it allows us to see graphically how the different processes interact with each other; however, due to the lack of a more formal definition of how each of the processes work and the partial description allowed by the automata it is difficult to verify the properties of the protocol using model checkers. Further, only a few process algebra tools allow for message passing and said tools cannot trim down the search space that emerges in the context of security protocol verification due to the previously mentioned characteristics of the model.

With the discussion on [5] concluded, we will now proceed to briefly discuss [22]. The goal of this second paper by Groß *et al.* is to show that the WS-federation protocol specification is secure. In this second paper the authors reuse most of the components from the model in [5]. In both papers the authors resort to prove that given an ideal user behavior the information flow from the user to the server does not allow for the attacker to obtain a users credentials instead of proving that the properties of the protocols being modeled ensure the security of the data, due to the previously mentioned shortcomings of the formalization. The main difference in [22] is that the new model distinguishes between different kinds of servers that communicate through the user's browser which are referred as identity suppliers and consumers.

As we can see, these browser models focus on how the information flows from the browser to the sever and vice-versa while they leave aside most of the information stored by the browser and how the browser's behavior may change when interacting with itself via scripts, frames, etcetera. Although the models presented are successful in analyzing the information flow we find they are not able to find some other attacks that rely on the browser's behavior (*e.g.* those that rely on cookies) which might not be evident because of their shortcomings and level of detail.

2.2.4.2 A browser model based on standards

In contrast to the work by Groß *et al.*, Fett *et al.* present other model in [6]. The model created by Fett *et al.* has the same goal as the previous model but it can be seen as being at the other end of exhaustiveness concerning the amount of characteristics of the web being included. The model proposed by Fett *et al.* takes into account DNS, browsers, servers, as well as different kinds of messages and the attacker. The browser model includes things like cookies, local storage, scripts, policies, frames, and inter-frame communication. The main goal of this

work is to be a comprehensive model of the web; however, they note this completeness has its drawbacks due to the complexity and in turn how difficult it would be to create a tool for automatic analysis of protocols and applications.

The authors pose that an accurate formal model of the web is very valuable not only because it allows to precisely state security properties and perform formal analysis, but also because condenses important aspects in several specifications that would be otherwise spread across different documents (*e.g.* the HTML, DNS, HTTP, and JavaScript standards). The model proposed in [6] is thus a model intended to analyze the information flow of protocols and applications that takes inspiration from the Dolev-Yao communication model and as such it assumes perfect cryptography and takes into account an attacker that is in complete control of the network.

Web browsers in this model are assumed to be used by honest users but may be taken over by attackers. The user and its knowledge is modeled as being part of the browser. As such, the initial state of the browser contains all of the user's secrets, and user actions are taken as a set of non-deterministic actions that the browser can execute. The model also considers a flat DNS system; however, since the DNS requests and responses are not authenticated its behavior may be subsumed by an attacker. Messages, correspond to either requests or responses and are composed as sequences of sequences that contain all of the information required for an specific action to take place in the system. Additionally, the browser contains information regarding open and active windows (also known as documents or web pages), cookies, information storage, known keys, domains with STS, known and pending DNS addresses, nonces, pending requests, and whether it has been corrupted or not.

While the DNS and browser models are assumed to be independent of specific web applications scripts and web servers are not. Scripts are used to represent both static and dynamic web pages. Each time a script action is executed the browser is to interpret its meaning and carry out an action (*i.e.* send a message). However, since servers depend heavily on the specific web applications being run and their behavior is much broader than that of scripts, the authors do not provide a fixed model andcinstead argue that it should be provided on a case by case basis.

The method presented by the authors has been used in order to verify BrowserID protocol and was able to find two attacks which were corroborated and fixed by Mozilla. The analysis was made by manually proving theorems about the security of the BrowserID protocol. The proofs provided were achieved by using the models; that is, carefully analyzing the interactions that arise from both the data definitions and the algorithms that describe the behavior of the participants.

While this model is more comprehensive and accurate than the previous model we discussed. This model, as the authors admit, is also not suitable for automation. Moreover, the authors

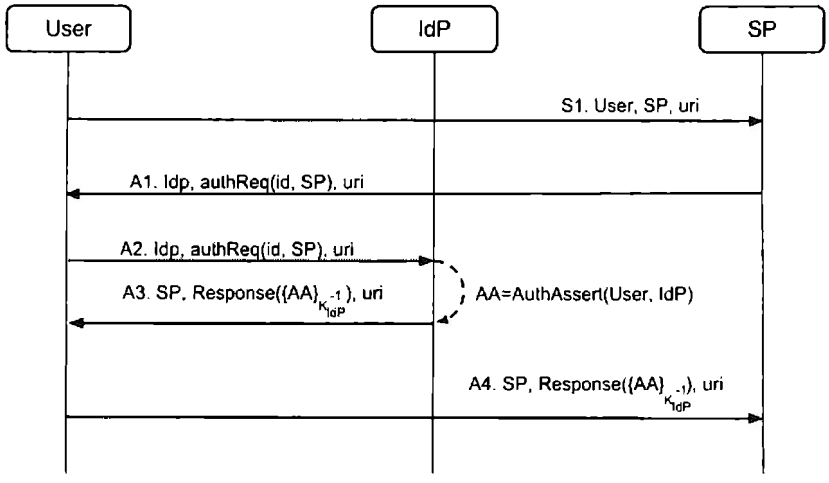


FIGURE 2.3: The SAML protocol as modeled by [7] using Google's specs

highlight that, as in any model, theirs has some limitations like not being able to reason about interactions of scripts, and the lack of a user interface model in order to represent elements overlapping or clickjacking attacks. These limitations lead us to conclude that while there are useful models for the verification of protocols we still need an approach that is both accurate to a degree and able to be automated.

2.2.5 Protocol Verification Using Model Cherckers

Now that we have seen the work done with browser models we will continue by discussing approaches that define web protocols as being related to security protocols and thus use model checkers used to verify the latter objects in order to verify the former ones. In [7] Alessandro Armando and Roberto Carbone show us how model checkers can help us in the verification of browser based protocols; however, there are some shortcomings as with any other approach. This paper takes into account the work done in the area of browser based protocol verification by Tomas Groß [5, 22] among others.

The approach taken by the authors of [7] is using SAT Model Checker (SATMC) to analyze a set of LTL formulas that model the Google version of the SAML protocol, the intruder and the intended security. As we can see in Figure 2.3 this is the way in which the authors model the information flow of the SAML protocol according to the specs and in Figure 2.4 the authors model the information flow of the Google version of the SAML protocol.

In Figures 2.3 and 2.4 we can see the intended information flow of the protocol. First the Consumer (User in the figures, corresponding to the user and its browser in the real world) sends a request to a Service Provider (SP in the figure). Then the Service Provider gives a

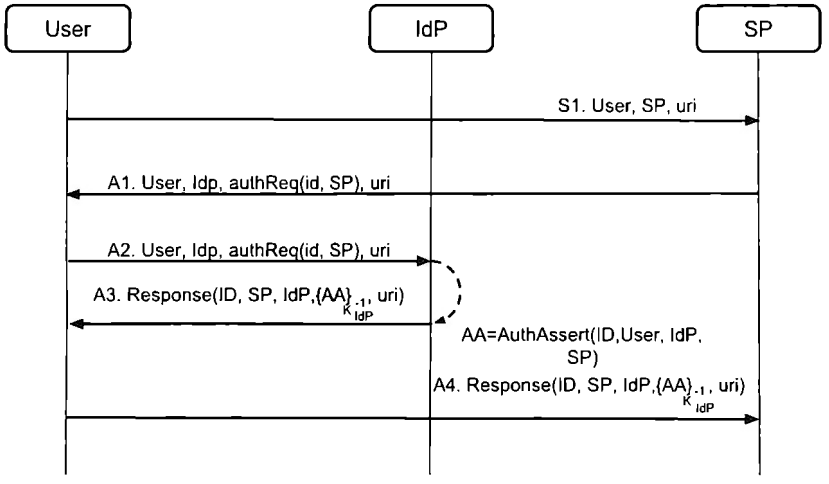


FIGURE 2.4: The SAML protocol as modeled by [7] using specs

response that requires the user to be authenticated by an Identity Provider (IdP in the figure). After that the Consumer sends its credentials to the Identity Provider and if the credentials are valid the Identity Provider responds with an authentication assertion that is sent first to the user and then to the Service Provider. Finally, the service provider, responds with the resource requested initially by the Consumer. The only difference between the two protocol versions is the contents of the A3 Messages.

After modeling the protocol the authors of [7] use SATMC to check for vulnerabilities that can be used to attack the protocol. The information flow used in the vulnerability that was discovered can be seen in Figure 2.5, this vulnerability relies on the fact that the protocol does not say explicitly who the intended recipient is, so the authentication message can be reused by a dishonest service provider to impersonate some user. This attack is made possible by the fact that on Google's version of the protocol the Authentication Assertion (the part of the Identity Provider response that is signed) and the "Response" did not include which service was the intended recipient of the message and thus can be used several times with different service providers.

As we can see, the approach in [7] works really well; however, it does not take into account the way in which the web page was requested or how scripts, which may be running in the background, interact with the protocol among other things that may put the protocol in a vulnerable state. In other words, authors do not take into account the possibility of mixing content from various sources or things like external scripts doing something with the information contained by the web page being presented (*e.g.* it does not take into account invisible content, animations that change the web page, sending and receiving extra data, etc.) and thus leaves

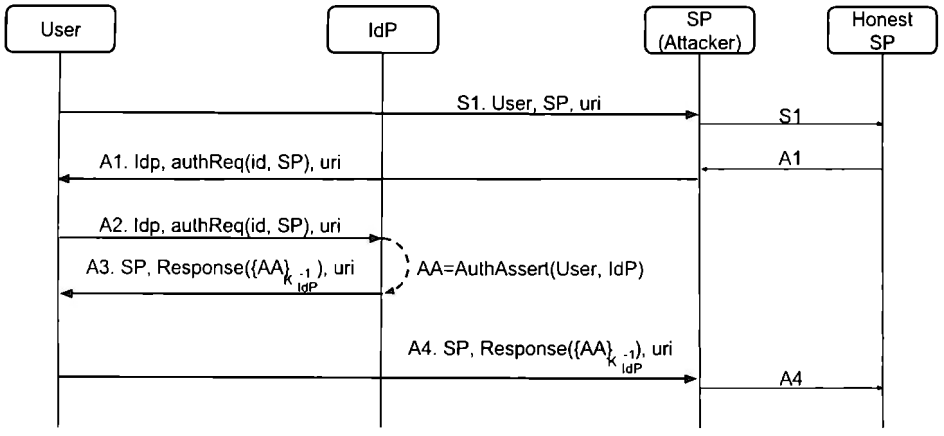


FIGURE 2.5: The vulnerability found in the Google version of the SAML protocol

aside some potential attacks that could modify the behavior of the protocol slightly to send some data to other parties like the attacker.

It is also noteworthy that this approach does not take into account attacks such as the ones presented in [27]. According to [27] it is possible for an XML message to pass validation and still be tampered or modified by an attacker. These attacks rely on the fact that usually the signature is validated by the ID of the elements while the actual logic of the applications rely on the position of the elements in the XML messages.

As we can see, the approach presented in [7] has been successful in finding some attacks to browser based protocols. However, we found that some of its own features restrict its usefulness like not including enough characteristics of the browsers and the approach requiring, for each protocol to be analyzed, an specification represented as a set of LTL formulas. We must note that representing protocol properties as LTL formulas can be difficult since there might be ambiguity in the protocol's specification or other problems that where not taken into account when the properties were being transformed to these LTL formulas.

2.2.6 Protocol Verification using Belief Logics and Model Checkers

So far we have outlined approaches that use browser models and others that use LTL formulas to encode protocols and model checkers in order to verify if the encoded specifications are correct. However, there is work that merges some of the approaches mentioned at the start of this section. One of the most notable works that make use of a mix of approaches is that of Kumar [28] where he extends some of the ideas behind his previous work [29].

In [29] the author argues that the analysis of browser based protocols poses new challenges like the inclusion of principals without identifying keys, a lack of global identities, the existence of

user interaction with the protocols, the characteristics of browsers that allow for them to be used as attack vectors, and the use of lower level protocol as primitives. It is due to these challenges and things like belief logics not needing an explicit attacker that the author suggests that the use of BAN (an inference based approach) is better suited to analyze browser based protocols.

The proposed BAN extension consist of 6 new operators and 7 new inference rules. To illustrate the usefulness of his BAN extension, Kumar analyses and finds attacks to the SAML Identity Linking Protocol and the Oauth protocol. To discover these attacks the author first uses his extension to BAN logic in order to find what are the main security properties or belief of the participants; then, when he finds one which he considers to be not strong enough he hardens said property and checks where its revision may fail. Finally, with this the revised security goal and his analysis of the protocol the author proposes an attack and a way in which it may be fixed.

The attack found for the SAML identity Linking Protocol was one in which both the attacker and user have valid account for the service but the attacker instead of finishing its authentication reuses the given token in a way that induces the victim into also login into the service thus linking both accounts. The attack found for the OAuth protocol is similar to the previous attack, it consists on the attacker beginning the protocol without finishing it and then inducing the victim to unknowingly finish its execution. The fix for both attacks consists on adding a cookie with information regarding the token provided by the identity providers, the attack is fixed in this way since the author considers the attacker to be unable to tamper with the information inside cookies. As we can see, Kumar's extension to BAN logic is useful. However, the analysis is done by hand at each and every step; further, it requires for the person doing the analysis to be highly familiar with the formalism, the security goals, attacks and the protocols themselves.

Kumar tells us in [28] that there are several shortcomings in the area (see[30]) like soundness issues, and the error prone work of idealization as well as the lack of work in making belief logics and model checkers able to deal with the especial characteristics of browser based protocols. It is due to these and other shortcomings that the author continues his research in the area. In this subsequent work Kumar proposes an hybrid approach that uses an extension to BAN logic in order to create simpler models to be used by Alloy.

The extension used in [28] is similar to that of [29]; however, there is a difference in the inference rules used by the two approaches. The Alloy framework proposed by the author contains the formalization of several constructs and principals like those of users, servers, cookies, keys, and messages; these formalizations also encode several rules that model the behavior of the principals as well as several constraints and assertions that should hold through the analysis of the protocols.

Kumar proposes the use of the logic in order to find the goals and to simplify the models to be encoded in Alloy by extending the proposed formalizations; however, he does not state clearly how this simplification or encoding should work and just states that the simplification reduces in about 60% the complexity of the models and thus the Alloy analysis is faster. In order to test his new approach Kumar analyses the same SAML identity linking protocol as in his previous work and finds the same attack.

2.3 Conclusions

As we can see in this brief account of the research done in order to secure both browsers and web protocols, the work is differs widely and most of the time too focused on certain aspects due to the complexity of the web. Due to the characteristics and shortcomings of the research in the area as well as the importance of the web in our everyday lives; We pose that there is a need for a method for the verification of web protocols that is amenable to automation while being able to represent the different behaviors of the web.

The main problem we found with using the current state exploration approaches was that they lacked the expressiveness we wanted and needed; this, in order to express the behavior of browsers and servers and how their behavior depends on the contents of the messages themselves and not on either the type or order of the messages. On the other hand the use of logics; be it either belief logics, first order logic, or higher order logic; demanded that users of the method should be experts, and completely familiar with both the problem and the logics. Requiring such familiarity is problematic if we want to take the formal verification from an afterthought to a reality when designing new protocols and applications.

Now that we have made clear our starting point, the relevance of our work in this area, and why we need expressiveness and compromise in the amount of characteristics included in order to get the verification of protocols from an afterthought to a reality when designing, understanding and analyzing web protocols. We will proceed by providing a general overview of our method as well as by defining its most important characteristics.

Chapter 3

Attacks on Web Applications

In order to analyze and verify security properties of protocols we need to be able to represent and automate the analysis of attacks. There are a great number of attacks to which protocols may be vulnerable. As such, the main objective of this chapter is to explore the difficulties that lie in representing the attacks we want to be able to analyze, and what is needed by an automated tool in order to be able to analyze them. We will focus on Cross Site Scripting, Cross Site Request Forgery and attacks based on cookies, since we consider these attacks to be the most widespread. We will discuss the attacks' information flow, and explore the different characteristics the attacks rely on. We conclude the discussion by summarizing the characteristics required in order for the attacks to be successful and thus required by an automated tool in order to represent them.

To depict the information flow of an attack we will use diagrams to exemplify the expected behaviors. The diagrams follow the usual conventions. An arrow denotes the submission, by the participant at the start of the line, and reception, by the participant the arrow points to, of a message. Messages are specified above each arrow. Messages hidden from the end user are denoted by dashed lines. Messages specify their sender and the intended recipient as well as their contents. In the information flow diagrams we shall use B to denote the browser, A to denote the attacker and S to denote a server.

3.1 Cross Site Scripting (XSS)

Cross Site Scripting, commonly known as 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 victim user, steal user data or both. The attack is carried out in a way that resembles the diagram in Figure 3.1 where "fake instructions" correspond to the malicious script, "data" corresponds to the normal information in the protocol and "fake

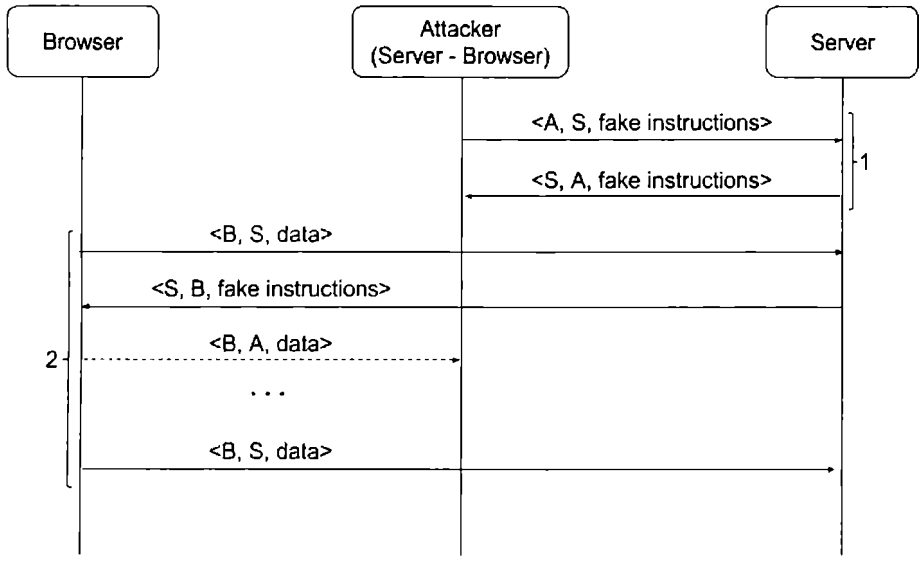


FIGURE 3.1: XSS attack

data" to information injected along with the script that is used to perform actions that should not happen.

In the diagram, an attacker has a session with an honest server, and uses a vulnerability or a characteristic in the server to inject scripts in a web page. Once a malicious script has been injected into a web page, the attacker has indirect access to all of the information the browser has access to in said web page.

For a XSS attack to take place we need at least two sessions, one with a normal protocol execution and the other which runs a protocol that has been modified by the attacker. The first session we need is one between the attacker and the server (denoted in Figure 3.1 as 1). In this session the attacker behaves like a browser would but sends malicious information along with correct information to corrupt the server and protocol. A second session now occurs between the browser and the server (marked in Figure 3.1 as 2). In this part the original protocol and server have been corrupted by the attacker (strictly speaking, the attacker is using the first session in order to modify the protocol to be executed in the second), as such the attacker may modify the information flow. When the server is corrupted in this way the attacker has no direct access to the stored information in the server. The goal for the second part is reached when the attacker gets information that belongs to the user or honest server, information that he should not receive (*e.g.* a cookie containing a session identifier), or when he performs actions in the name of the user (*e.g.* a money withdrawal from the user's account).

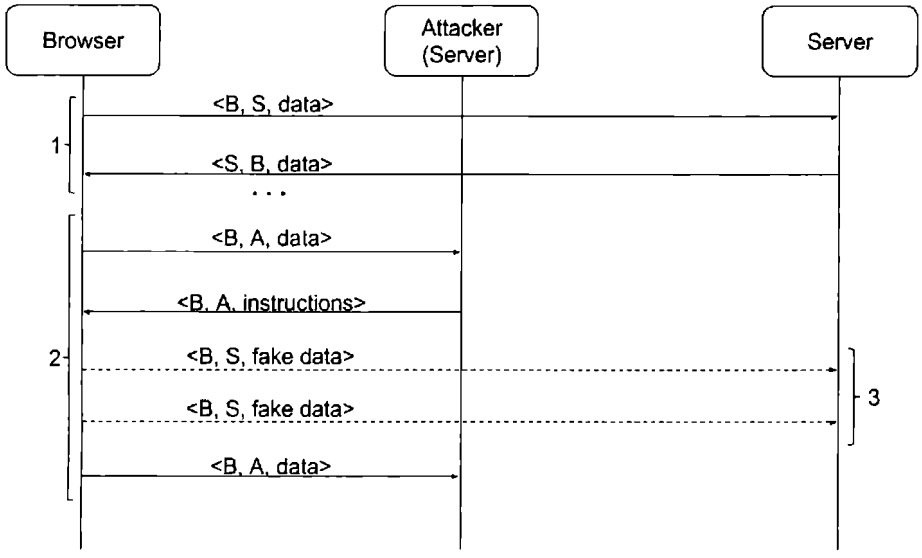


FIGURE 3.2: CSRF attack

A tool able to analyze the behavior arising from a XSS attack needs to model partial server corruption as well as scripts, and an attacker able to generate arbitrary scripts and information. These characteristics are missing from most of the previously mentioned approaches and thus are not able to deal with the attack.

3.2 Cross Site Request Forgery (CSRF)

Cross Site Request Forgery (CSRF) is an attack where 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. The attack consists on trying to make requests in name of the user; however, the attacker is neither able to know if the user already has an active session with the honest service nor is the attacker able to access or retrieve the information the browser has about said honest service. The information flow of a CSRF attack is depicted in Figure 3.2, it takes place specifically in the dashed lines that correspond to actions the user cannot see. The main difference with XSS is that in CSRF one server is fully controlled by the attacker while the server and service to be attacked is never compromised and thus the attacker cannot directly access any of the information held by the browser or the server.

CSRF attacks rely on the ability of an attacker to include invisible components and extra instructions on web pages. For this attack to work we need two sessions; one between the browser and an honest server (denoted as 1 on Figure 3.2); and one between the browser and a compromised server (*i.e.* the attacker). The session between the attacker and the browser

should happen after the browser already has a session with the honest server, and is used in a way in which the attacker can send any information by using the browser as a way to reflect the messages (labeled as 2 on Figure 3.2). The attack is carried out in the part marked with a 3 on Figure 3.2 and is deemed successful when the attacker notices changes in the data he has access to on the honest server (not seen on the figure).

A tool able to analyze the behavior arising from a CSRF attack needs to model a browser able to present partial information to users and servers, a user with an incomplete knowledge of both what is currently being presented and of the communication happening, servers fully *controlled* by the attacker as well as scripts, and an attacker able to generate arbitrary scripts and information. These characteristics are missing from most of the previously mentioned approaches and thus are not able to deal with the attack.

3.3 Cookie Attacks

Cookies are the usual way in which servers store information in browsers and are tremendously useful since users may access services via a multitude of browsers, IP addresses and devices, thus the servers require to store session identifiers and information in each of the browsers to be able to keep a consistent state. However, there is a major drawback to storing information in cookies. That is, an attacker may tamper with the data inside them. This tampering affects protocols and session management since servers may store session identifiers inside of cookies which may lead to some insecure states. Some of these insecure states consist in a user impersonating another user or even an attacker, while other of these insecure states can be identified by a browser that sends unintended information across user sessions. Examples of the information flow of attacks based of cookies can be seen in Figures 3.3 and 3.4 where "session_n" indicates a cookie to be set in the browser.

The attacks in Figures 3.3 and 3.4 are achieved when an attacker succeeds in making a user impersonate him fully or partially. The difference between the two attacks lies in that in the first one the attacker modifies part of an honest web page so that the user does not suspect anything. This is possible due to the fact that pages are not monolithic and get information, frames, media and instructions from several locations. Both of these attacks rely on two things, the attacker having a way to retrieve information from honest servers (*e.g.* by having a valid account with the service) and the attacker being able to inject information (*e.g.* cookies) to a browser without user interaction.

For the attack in Figure 3.3 to succeed three full sessions are required. We need one session between the browser and the honest server (marked as 1 on the figure); one session between the attacker and the honest server (labeled as 2 in the figure); finally one session between

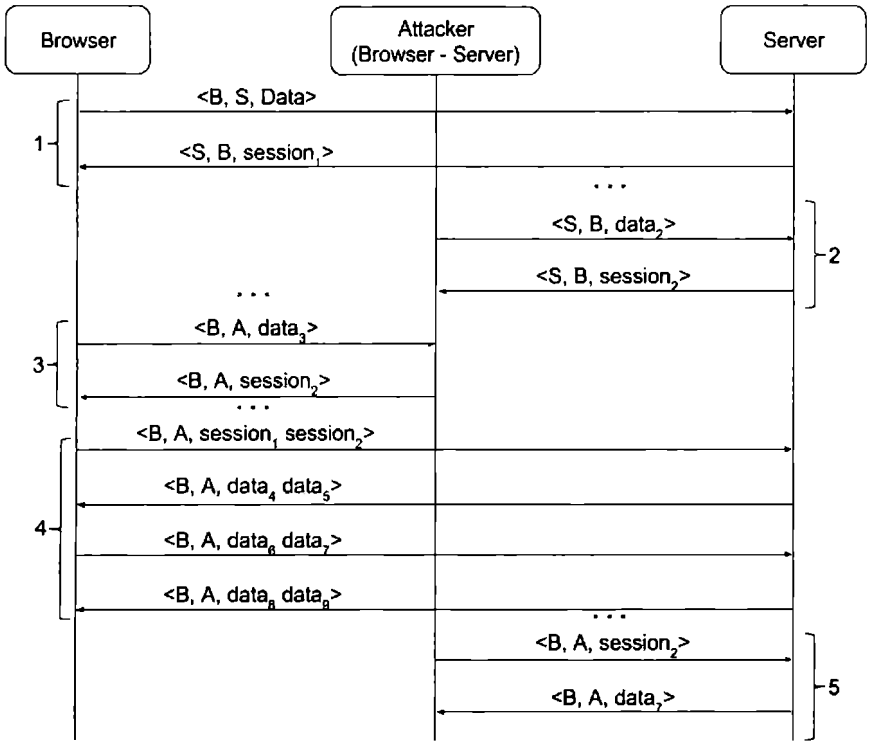


FIGURE 3.3: Cookie attack 1

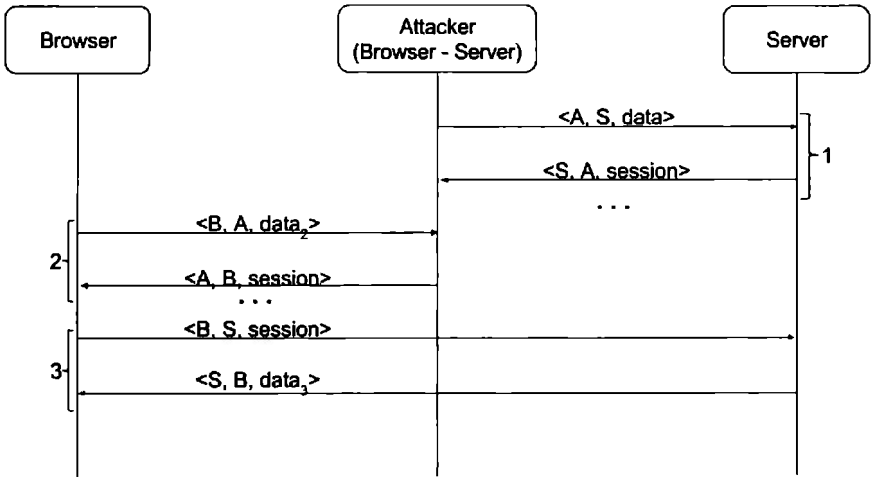


FIGURE 3.4: Cookie attack 2

the attacker and the browser, in this session the attacker overwrites and adds cookies to the information in the browser (denoted by 3 in the figure). After the third session takes place the user needs to interact again with the honest server, or in other words, continue the first session (marked as 4 in the figure); however, the session being continued has been modified by the attacker and in doing so, the information sent by the browser partially corresponds to both the attacker session with the honest server and the browser session with the honest server. Finally, if the attack was successful the attacker may retrieve all of the information stolen from the user (labeled as 5 in the figure).

For the attack in Figure 3.4 to succeed we also need three sessions. The first one between the attacker and an honest server (marked as 1 in the figure); the second one happens between the browser and the attacker, in this way the attacker is able to insert cookies to the browser (labeled as 2 in the figure); finally a third session that the server misinterprets as a continuation of the first session with the attacker but instead of the attacker it happens between the browser and the honest server (denoted by 3 in the figure).

The first cookie attack mentioned works by causing the server to interpret the attacker modified session in an inconsistent way, thus storing data sent by the browser to a session to which the attacker has access to. The second cookie attack mentioned works by first creating a information the server is willing to accept in the attacker-server session, injecting this information to the browser using the browser-attacker session, and thus starting and collecting data on a what should be a new or empty browser-server session.

A tool able to analyze the behavior arising from the cookie attacks needs to model cookies, a browser able to handle cookies, and an attacker able to have its own sessions and able to generate arbitrary scripts and information. Characteristics are missing from most of the previously mentioned approaches and thus are not able to deal with the attack.

3.4 Conclusions

As we said in the introduction, our automated tool and method work by trying to find a counter example to a given security property. Specifically, our goal is to analyze properties relating to confidentiality and integrity while leaving aside concerns about availability. The main confidentiality concept we want to assert is *secrecy* (*i.e.* that some information will never be known to an attacker even if it compromises one or more servers excluding those that are the intended recipients of the secrets). The integrity properties we want to assert are related to an attacker not being able to use either injected information or information any of the participants possess in order to modify the state of one or more honest servers against the will of a user.

With that said, we will now summarize what our method requires to model in order to be able to represent the presented attacks. Due to the possible behaviors presented, the method requires the existence of four different kinds of participant namely a user, a browser, an attacker and one or more servers. We model the user and browser as two different entities in order to be able to differentiate their actions and to keep separated their knowledge about what is happening during the application or protocol execution. To represent the actions of the user we need her to interact with the browser; however, the browser must present just parts of all the information it possesses in order to be able to send hidden messages. As such, our method requires the existence of hidden elements and a display. Since the browser needs to send and receive several messages for each web page loaded, the method needs a model of scripts and web pages that require resources from several servers. Since the servers and browsers need to keep track of sessions and other data the method also needs a way store said information. Finally the attacker must be able to control the network, and act as a browser or a server when required; thus the attacker must be able to corrupt servers and have all the capabilities of the Dolev-Yao attacker.

Chapter 4

Method Overview

As expected, the verification of security in browser based protocols requires a formalization of the protocol or application to be studied, a property of interest to be proven about the protocol, and a set of rules that allow to compose such proof. Due to the nature of our approach, an state exploration technique in charge of finding a counter example to a security goal, these three components are interwoven and divided in two parts. The first part is composed of the models and behaviors of four parametric participants; that is, it contains the rules under which our proof will be composed. The second part consists on the specification of a protocol, participant knowledge and a security goal. Both parts are to be used by our method in the following way: The given specification is used in order to instantiate the parametric components and thus follow the rules in order to find a counter example to the given goal if there is any.

In order to explain how our method works, in Section 4.1, we will start to explore the general characteristics of our proposal, After doing so we will discuss, in Section 4.2, the system in which protocols will be verified. Then, in Section 4.3, we will define the types of rules used by the participants in order to change their internal state. After that, in Section 4.4, we will define what a protocol is and give an overview of how protocols are analyzed in the system. Finally, in Section 4.5, we will provide a definition for the security properties, that is, we will conclude by providing the relation that exists between a protocol and a security property thus explaining what we mean by a security guarantee.

4.1 General Description of Our Method

As we have discussed in Chapter 2 there are different approaches for the verification of browser-based protocols. Further, in Chapter 3, we have analyzed the characteristics needed by a method and a tool in order to be able to represent different kinds of attacks. With this in mind, it can

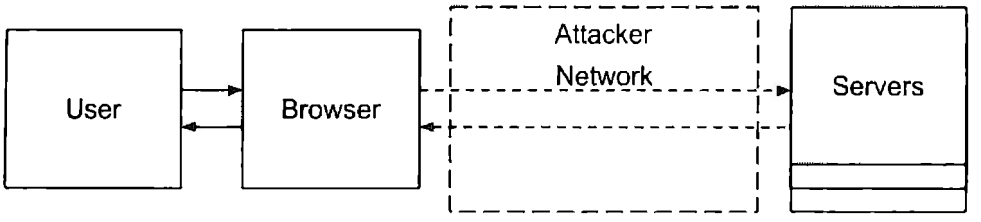


FIGURE 4.1: WebMC as presented in Chapters 5 and 6

be argued that we require a new way of approaching the problem of browser based protocol verification. We propose a method that not only deals with how the information flows but that also deals with at least part of the behavior of a normal browser (*e.g.* Multi-part pages, scripts, cookies, etcetera). Due to the limitations and lack of expressiveness of other approaches we decided to develop our own tool and method. Both, our tool and method, take ideas from the related work. Now that we have discussed the motivation and the previous work, we present our method, a state transition system that generates and analyzes on the fly the interactions among the different participants in browser based protocols, this in order to find counter-examples to given security properties. Our method uses the model shown in Figure 4.1. Each of the components of our method, that is, the corresponding models and behaviors will be presented in detail in Chapters 5 and 6.

Our method and tool consist of a user component that models the actions of a real world user with knowledge of the web and thus is not fooled by phishing attacks; a browser component which models a real world browser which consists of a display, is able to construct web pages from several request and responses, manages request and responses according to policies, executes simple scripts and is able to store information; a server component that models a parametric server to be instantiated by the different specifications and requirements of the protocols; and an attacker component that models attacker owned network. Each of these models can be extended or replaced to deal with other particularities of web communication; however, with the mentioned characteristics we are able to represent all of the previously presented attacks.

Due to size and level of detail already contained in our models, and in order to keep them from being too complex for automation and thus keep us from being able to implement an automated tool, we decided to leave outside the scope of the method some characteristics like the interaction of browsers with the operating and file systems, frame navigation, inter-frame communication, and complex scripts that modify the browser state. However, in Chapter 8 we discuss some possible additions to our model that would be useful in order to automate more behaviors and in turn the attacks that depend on them.

We must note that since the network is modeled as being owned by the attacker we also decided to ignore the behavior dependent on DNS and unencrypted HTTP and instead take into account an encrypted channel. We decided to ignore unencrypted channels since they allow the attacker to carry out a permanent man in the middle attack, thus preventing us from proving even aliveness in the Lowe hierarchy of authentication [31]. With this said, we pose that our tool is able to verify all of the levels of authentication posed by Lowe given the properties can be expressed within the characteristics of our model and said properties are presented as a list of events in the system as defined in Section 4.5.

A system in our method is composed of several participants or processes, its state is the union of the states of the participants it comprehends and its behavior is the result of the communication or interaction among them. Interactions in the system have, then, as a result the change in state of two participants and thus of the system as a whole. In order to represent each participant we use a model defined as a conditional sequence of discrete events the participant is able to perform (a set of state transition rules). Events in turn include messages with complex structures.

Interaction in our system is the result of sharing events; that is, two components which have the same sort may carryout events just as in process algebras. Each participant and thus the system can then be seen as a labeled state transition system in which events, and their contents, correspond to the labels used and to the information being exchanged. We take these events as being composed of ground terms; that is, events in the system possess no variable or symbolic content. If we take into account that events are grounded and different for each run we can then generate a trace describing the state transitions of the system, and in principle we can also generate all possible traces of said system. Finally, we must note that the communication within the system in restricted and only happens in the way that is depicted in Figure 4.1.

As we can see our verification method is one heavily influenced by process algebras. While using already defined process algebras, like CSP, CSS and the π-calculus for the specification of our models and the verification of security properties, is one of the most compelling approaches to try we decided against their use because of the limitations of the languages and the tools available. We needed to express some capabilities, like abstract data types and branching execution, inherent to browser based protocols in a simple way. That is, a different approach was more sensible to express our needs such as modeling the state of participants as abstract data types, modeling message passing, and modeling in a simple way branching execution paths based on the contents of messages.

4.2 The System and its State in our Method

So far we have talked about the general characteristics of the method, we are ready to explore it further by describing the way in which the participants behave:

- User
  - Gives an input to the browser in the form of an URL, a click on the back button, a click on the forward button, a click on a link or button presented by the web page or form submission
- Browser
  - Receives User inputs and if enabled transforms them in requests, otherwise it ignores the user input
  - Sends requests to the Servers via the Attacker
  - Receives responses via the Attacker and processes them in case it is expecting them to form a display for the user, otherwise the responses are ignored
- Attacker
  - Receives all requests from the Browser
  - Receives all requests and responses from the Servers
  - Analyzes all incoming messages
  - May modify outgoing messages
  - May create its own messages with its knowledge
- Server
  - Receives requests and responses via the Attacker
  - Processes responses
  - Sends zero or more requests and one response for each request it receives.

Now that we have given a high level explanation of how the participants work we must define the system. A system is composed by a user, her browser, the attacker and a variety of servers. More formally:

Definition 1 (System). *Let $\mathcal{U}$ denote a user process, $\mathcal{B}$ denote a browser process, $\mathcal{A}$ denote attacker, and $\mathcal{S}^1, \ldots, \mathcal{S}^n$ denote the different server processes that participate in a protocol. The system $\mathcal{G}$ is: $\mathcal{G} \stackrel{def}{=} \langle \mathcal{U}, \mathcal{B}, \mathcal{A}, \mathcal{S}^1, \ldots, \mathcal{S}^n \rangle$.*

We use an event-based model for capturing the interface and the behavior of each of the principals, as well as for the system of browser based protocols. As usual, events correspond to actions that cause a transition from one state to another.

Definition 2 (State Event System). *A state-event system is a 6-tuple,* $\rangle E, In, Out, St, st_0, T\langle$ *where* E *is a set of symbolic events,* $In \subseteq E$ *and* $Out \subseteq E$ *are the sets of input and output events, respectively,* St *is the set of symbolic states,* $st_0 \in St$ *is the initial state, and* $T \subset St \times E \times St$ *is the transition relation. We require* In *and* Out *to be disjoint and* T *to be a partial function from* $St \times E \rightarrow St$.

We consider E, In, Out to be sets containing an infinite number of ground events. Carrying out an event is further constrained by the knowledge and information contained in the state of each participant.

The global state of the system g is then composed by the individual states of the participants in a protocol, namely the states of the user, the browser, the attacker and the servers. We will use G to denote the set of all possible states g.

Definition 3 (State of the System). *Let* U *denote the state of the user,* B *denote the state of the browser,* A *denote the state of the attacker, and* $S^1, \ldots, S^n$ *denote the states of different servers that participate in a protocol. The state of the system* g *is:* $g \stackrel{def}{=} \langle U, B, A, S^1, \ldots, S^n \rangle$.

With this said, we can model each component as a state event system, and their composition, another state event system, corresponds to the system uses by our method. The states of the user U, the browser B, the servers S and the attacker A consist of data important for their function and will be defined in the corresponding chapters. We must now remark a characteristic that differentiates browser based protocols from security protocols, the participants excluding the attacker may not change their roles in subsequent runs of a protocol. In other words: A user will always behave like a user; a browser since its a program executed in a computer will always behave like a browser and may not subsume the behaviors of either users, servers or the attacker; the server may not change its specification and may not under any circumstance behave like another participant; Finally, the attacker may behave like any participant as long as it has the required knowledge.

4.3 Transition Rules and Interactions among Participants

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, for an input or output to occur there needs to be two participants willing to perform an event.

The state transformations that occur are calculated by following transition rules associated to each of the participants. The transition rules that control the transformations and behavior of each principal will be given in the chapters corresponding to each of them and are expressed in either one of the following two ways:

$$input.X\ conditions \rightarrow update\_functions\ X'$$
$$X.output\ conditions \rightarrow update\_functions\ X'$$

Where X denotes the present state of a *Principal* X, *input* and *output* represent an event. The event appearing before the "." indicates that the principal is able to receive said kind of input event while it appearing after the dot indicates that the principal is able to perform an output of said event. The preconditions for sending or receiving an event are stated by *conditions*, these conditions are predicates about the state of the principal and the event to be performed. We will use *update_functions* to denote the transformations or postconditions of executing an event, these functions model and specify how the state of the *Principal* is transformed. X' represents the state of the *Principal* after performing and processing an event. That is the $'$ denotes the next state, and we must note that we also use the $'$ as an homomorphism on the internal structure of the principals.

The first rule tells us that the state of principal X is able to change its state to X' if it receives an input event e and the conditions are met. This means that if a principal receives an input and the conditions for it to be processed are not met the principal will simply ignore the message (which may lead to a lossy queue in some cases). The second rule tells us that a principal will perform an output event if the conditions for its output are met; further, the rules allow for the existence of states in which a principal is able to perform several outputs one after the other. We must note that our system only allows for one event between any two states and that it does not include any internal or non-shared events.

The Interactions, and thus the state transitions for the parallel composition of all the elements of the system can be defined by the following:

Definition 4 (Interaction). *Let X and Y be two principals of the system and be contained in (G), e be an event principals are willing to perform, and* event *be a function on a principal that returns a set of events said principal is able to perform then:*

$$\frac{X \xrightarrow{e} X' \; Y \xrightarrow{e} Y'}{X|Y \xrightarrow{e} X'|Y'}$$
$$\frac{X \xrightarrow{e} X'}{X|Y \xrightarrow{e} X'|Y} \; e \notin \text{event}(Y)$$

The system 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 the synchronizations among the different principals. This means, that the system interleaves the execution of asynchronous atomic events generated by the principals in order to create execution traces of protocols. In principle, the rules allow for the synchronization of more than two principals with a single action; however, the system only takes into account one user, one browser, one attacker, and several servers. While there may be several servers, each of the messages states the intended destination, which means that a server will and must ignore all events with messages not directed at it thus preventing multiway synchronization from occurring. However; explaining and defining the state and transitions of each component is not enough since at any one time more than one event may enabled in the system. It is because of this that we model the choice of the transition to be carried out as an non-deterministic choice among all of the feasible transitions.

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 we must add the following restrictions on how events are shared among participants and thus which participants are able to communicate.

Definition 5 (Restrictions on Participant Interactions). *Let X and 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:*

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

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

4.4 What is a Protocol in Our Method

Usually, when a participant (*i.e.* User, Browser, or Server) views a protocol execution he is limited to its local view of the protocol's intended behavior. Due to this limited view is that

we want to formalize not only how each of the participants in browser based protocols behave, but also how participants see the protocol being executed and how do they determine if the protocol was completed successfully or not.

In order to define how the different states of both the principals and the system are related let e denote an event, and $\mathcal{E}$ represent a set containing all such possible events; An execution of a given state event system is represented by a trace τ and $\mathcal{T}$ represents a set containing all such execution traces. We define an execution trace $\tau \in \mathcal{T}$ as a sequence of grounded events $[e_0, e_1, \ldots, e_n]$ with $e_k \in \mathcal{E}$, $\forall k \in \{0, \ldots, n\}$. Traces are deemed to be left-complete, *i.e.* each left prefix of a trace is a trace.

Definition 6 (Protocol). *A protocol P is an interleaving of traces, each interleaving is denoted with a p*

These definitions are then naturally extended to the system and its global state g. The main difference between a system trace τ and a protocol trace p is that system traces are not constrained to the information and knowledge defined by any one protocol. Whether a given trace belongs to a protocol depends on the protocol specification; however, given a protocol specification our models are constrained by it a thus unable to perform events that lie outside said specification (as will be seen in Chapters 5 and 6). In other words, We can say that a protocol is a set of all runs and each run is given by a p trace.

We must also note that a state-event system is deterministic in the sense that for a any state g and a given event e, there is at most one successor state g. With this said, we can argue that the system behaves in CTL since different events may be enabled in a given state g and their occurrence results in different successor states. As such, the set P can also be visualized as a tree or a graph.

4.5 Proving Protocol Security in Our Method

In our method security is mapped to a safety property. A security goal is then expressed as a sequence of symbolic events, yielding an insecure state, that should never happen within a protocol trace p. For example, if we want to assure that a term guarantees authentication the goal would be represented as a sequence with one event representing a message where such term is being reused by a party. Thus, if we want to talk about security we must first define what it means for two events to be equivalent. In order to compare events we must first introduce an structure on their contents let then an event e contain a message of form $\langle origin, destination, payload \rangle$ where *origin* and *destination* refer to principals and *payload* is the information exchanged.

Definition 7 (Value Equivalence). *Let payload and payload' be the contents of a message, they are deemed to be equivalent iff there is a one to one mapping from payload to payload' that does not change.*

This notion is then naturally extended to events.

Definition 8 (Event Equivalence). *Let e, e' represent two events, they are said to be equivalent written as $e \approx e'$ in symbols iff they contain the same information (*i.e. *origin, destination, and payload) modulo nonces and other fresh values.*

This notion of equivalence is in turn extended to take into account sequences of events. This extension is necessary to ensure consistency both within a list of events and between the list of events and a trace p. Correspondence is defined as follows.

Definition 9 (Correspondence). *Given a trace $p = [e_0, \ldots, e_k]$ and a sequence of events $E' = [e'_0 \ldots e'_n]$, we say that* corresponds(E', p) *holds iff for each element of E' there exists an equivalent event in the trace p. That is, there is a list of events $E'' = [e''_1 \ldots e''_{k-n-1}]$ such that the sequence of events p ($[e_0, \ldots, e_{k-1}]$) is equivalent to an interleaving of E' and E''.*

We are now able to define what is an insecure state.

Definition 10 (Insecure State). *Given an event sequence $E' = [e'_0, \ldots, e'_n]$ pertaining a security goal (*i.e. *violates a security property when executed), a state g_k resulting from following a trace $p = [e_0, \ldots, e_m]$. g_k is* insecure *with respect to E' iff E' corresponds to p and e'_n is equivalent to e_j for any $j < k$.*

In order to continue with our exploration of what it means for a browser based protocol to be secure let insecure(g, p) be a predicate on whether a state g complies with the previous definition in a given trace p. Then proving that a protocol is secure is equivalent to proving the following lemma:

Lemma 1. *A protocol P is secure iff it contains no insecure states in any trace p. In symbols:* $\forall p \in P \nexists g;$ insecure(g, p)

Proof: We can safely assume that the protocol starts in a secure state, and we will consider that an insecure state is the result of an attack. Thus proving Lemma 1 is trivial, albeit time consuming, based on Definition 10. From Definition 10 we can see that an attack is denoted by an event allowed in a state belonging to a protocol trace, since a protocol contains all traces including those with attacker actions proving or refuting Lemma 1 consists on finding an insecure state. □

4.6 Conclusions

Now that we have explained how our method works, we will proceed to provide a detailed explanation of the characteristics as well as a model for the user, the browser, a model for a parametric server (*i.e.* it needs to be instantiated according to an specification) and that of the attacker. We will model in detail the parts, the state of each participant, the transition functions for each participant, define how the state-event systems are instantiated, and describe how the principals interact with one another in order to analyze protocols. We must note that, since the participants interact, the definitions are interdependent and some elements may be defined and introduced where they are needed to explain something rather than in the paragraphs that correspond to the participant said definition or element belongs to.

Chapter 5

User and Browser Models for the Verification of Web Protocols

Browser based protocols are somewhat peculiar because the honest participants have a defined role (user, browser and server) that cannot be changed either during or after a run, and because users are able to stop protocol execution, run other protocols before continuing with the original, and go back to previous steps of the protocol. This means that at all times, a user will only communicate with a browser; the browser will communicate with servers and the user; and a server may only communicate with either a browser or another server but not with a user. The attacker may behave as either a browser or a server depending on its goals.

User models are usually abstracted away and considered to be part of the protocol endpoints. However, as [32] poses, users usually a person in the real world, are one of the most important protocol participants, since users are fallible, attackers are aware of this fallibility, users own the information we are trying to protect, and while they may be unknowledgeable they are the participants that decide whether a protocol is being run correctly or not. In the case of browser based protocols the assumption of users not being fallible is not as valid since their interactions shape the execution of protocols even when the users have no idea about the actual protocol being run or how a proper execution should behave. In other words, a user interacts with the web via the browser which means that there might be invisible elements, that she does not really know which data is stored by the browser or even when and what data is being sent to servers most of the time. As such, we need a separate entity from the other components that models a user and her knowledge, to represent all the different kinds attacks, as well as rules governing user interactions, to represent how she guides and affects protocol execution.

Browsers, on the other hand, are the clients a user has in order to communicate with the external world and one of the most complex components of our method. In the real world, browsers, are in charge of interpreting the commands given by the user and the messages received from

servers as well as providing outputs for both the user and servers, all of this while interacting with the operating and file systems in the host computer. Additionally, browsers need to keep track of information and past interactions.

In this chapter we will discuss the first half of our system. We will begin our discussion by talking about the user in Section 5.1, and then we will proceed to talk about browsers in Section 5.2. Our discussion on the user starts in Section 5.1.1 where we will review the different elements with which a user interacts as well as provide a model for the knowledge or state of the user, then we will proceed to define the rules that model its behavior. Our discussion on the Browser begins in Section 5.2.1 where we give a detailed look at web pages, scripts, and by providing a model for the contents and the state of the browser; after doing so we define the internal state of the browser in Section 5.2.2; then, in Section 5.2.3, we discuss how the browser interacts with the servers, that is, we discuss the contents of requests and responses; after that, in Section 5.2.4, we will explore how the different policies shape and affect the behavior of the browser; we conclude our discussion on the Browser in Section 5.2.5 where we provide and explain the rules that allow the browser to interact with the user and servers in our method. Finally, we present our conclusions about the user and browser models in Section 5.3.

5.1 The User Model in WebMC

So far, we have discussed the method in general and have made clear the motivation behind our method having a user model. As such, we start discussion by giving a general overview of the user and its internal state.

5.1.1 The User Model

We shall start our discussion by defining the elements which the user makes use of to communicate with the servers through the browser. In order to make use of any kind of service on the web a user must interact with one or more servers, each of which is identified by an *URL*. We model an URL denoted by *url* as a tuple of the form $\langle server, path \rangle$, where *server* and *path* are strings used to denote a web server and an specific function on the server respectively. Clearly this representation of an URL is a simplification of the URL semantics as defined by RFC 3986 in [15]; however, while our URL definition will not be able to represent authentication via the URL itself or query strings it will be able to differentiate among IP addresses, ports, etcetera. We use $\mathcal{L}$ to denote the set of all URLs.

For a user, the state of a protocol is given by the information she knows and sees. The state of a user, or *User knowledge*, corresponds to the information which the user has acquired from

the start of the protocol execution until the current time. However, this knowledge will not be updated and thus, its actions depend only on its initial knowledge and what is being presented to him by the browser.

Definition 11. *Let U denote the state of the user and be defined as follows:*

$$U \stackrel{def}{=} \langle KUrls, KData \rangle \tag{5.1}$$

*Where $Kurls$ denotes a set of user known URLs and $KData$ denotes a set of associations that contains information known (*e.g. *user names and password) about different URLs.*

As we have said previously, a user makes use of a service (*i.e.* indirectly interacts with servers) by interacting with the browser. So before going further we provide both an abstraction of the browser and a definition of the inputs that the user issues to participate in a protocol.

5.1.1.1 A Browser From the User Perspective

For the user the browser is a *display*, something akin to an image containing a snapshot of the state of the browser. The display contains all the visible things that the user may interact with which we call *components.*

Components may use all the space they need in the display, and may even overlap one another; however, they may not interfere with any of the static elements of the display (*i.e.* the address bar, the back and forward buttons) or cover completely another component. Let $\mathcal{V}$ ranged over by $v_0, v_1, \ldots$ represent the set of all components, and let $V_0, V_1, \ldots$ denote subsets of $\mathcal{V}$. Then:

Definition 12 (Display). *A display d is a tuple $\langle url, V \rangle$ where $url \in \mathcal{U}$ is the associated URL with d, and V is a set of components $\{v_0, \ldots, v_n\}$ also for d. We shall use $\mathcal{D}$ to represent the set of all possible displays.*

5.1.1.2 User Commands

The user interacts with the browser issuing commands, we follow standard web nomenclature and call these commands *user inputs* to denote these are inputs to be interpreted by the browser and the web page being presented.

Definition 13 (User Input). *A* user input *i is either a position pos, an identifier in the display d, that usually corresponds to a click on a component (*e.g. *a link or button); an URL, $url \in \mathcal{L}$; a pair $\langle txt, pos \rangle$ involving some strings being input into a position, usually corresponding to a component being filled (*e.g. *a form being filled and then receiving a click on the send button);*

or either one of the atoms _Back and _Forward which represent a click on the corresponding button in the browser. In symbols we write:

$$i \in \{pos, url, \langle txt, pos\rangle, \_Back, \_Forward\}$$

We use $\mathcal{I}$ to denote the set that contains all possible user inputs, and let $i_0, i_1, \ldots$ range over $\mathcal{I}$. We must note that all of the user inputs will be interpreted by the browser.

Now that we have a few basic symbols and concepts defined we can go further and provide a definition of the user state, what the user takes to be the state of a protocol, and his behavior.

5.1.1.3 State Transitions of the User

As we know, g represents the state of the system at time j and g' the state of the system at time $j+1$. We will follow the same convention for the participants and their internal structure. With this said, let urls be a function on the state of the user that returns a set of all the URLs known by the user, links a function on the display that returns a set of all components a user can interact with but require no extra information, forms a function on the display and the user that returns a set of all of the elements a user can interact with that require extra information know by the user, and back, forward be functions that return either an empty set or a singleton containing the corresponding value for the event depending on if the buttons are enabled or not on the display. We must note that we consider the update of the browser's display to be immediate and that the functions work over the last available display and as such the user will not have any input events. The rules that manage the transitions given by the user outputs is the following:

$$\begin{aligned}
U._Back \in \text{back}(d) &\rightarrow U' \\
U._Forward \in \text{forward}(d) &\rightarrow U' \\
U.url\ url \in \text{urls}(U) &\rightarrow U' \\
U.pos\ pos \in \text{links}(d) &\rightarrow U' \\
U.form\ form \in \text{forms}(U, d) &\rightarrow U'
\end{aligned}$$

Where the first and second rules tell us that a user is able to send a click on the back or forward browser buttons as long as they are enabled in the display, and that after performing said event it continues to act as it did before. The third rule tells us that the user is always able to input an URL to the browser as long as she knows the URL. The fourth and fifth rules tell us that the user is able to click on links and buttons or submit forms as long as they exist in the display and the user knows the data being asked.

As is evident the event to be performed corresponds to what is immediately after the dot on the left side of the rule, and the conditions should hold otherwise the event cannot take place. U' is the same state as U since the knowledge and state of the user is not updated. As we can infer from the rules, the user poses an infinite transition system since there should always be at least one event it can perform. This in turn leads us to some properties of the user: a user may restart a protocol execution at any time, a user may start a new protocol at any time and a user may continue a partial execution of a protocol at any time.

With this said, the user state event system $\mathcal{U}$ be defined as

Definition 14. *let* $In = \varnothing$, $Out = \{pos, url, \langle txt, pos\rangle, \_Back, \_Forward\}$, $E = In \cup Out$, $KUrls \neq \varnothing$, $KData \neq \varnothing$, $U_0 = \langle KUrls, KData\rangle$, Ss *be a singleton with* U_0 *as its only element, and* T *be equal to the previously mentioned transition rules then:*

$$\mathcal{U} = \langle E, In, Out, Ss, U_0, T\rangle \tag{5.2}$$

The users in our model are probably the simplest of the participants; however, this model is the one that allows us to expand and continue our search for attacks beyond one simple session. The user is the participant that makes the system to be an infinite state transition system since it is the only one of the honest participants that does not depend on having an input to perform an event. We will now proceed to talk about the browser in our method.

5.2 The Browser in WebMC

We will now proceed to discuss the browser. As we know, browsers communicate with servers through messages. Server messages are usually presented to users in the form of web pages; however, in the browser they may take the form of HTML, scripts, plain text, binary data or a combination of thereof. When presenting web pages to the user the browser, in following the instructions posed in the messages it receives, may hide either elements, interactions, or both contained within these web pages from a user. The capability of hiding elements and interactions lead to an user that may end up interacting with something she did not intend or foresee to interact with; further, since the browser does not possess the protocol specification it is unable to determine the protocol or application is being executed at the server or whether if the protocol is being carried out according to its specification.

Browser interactions follow certain rules or policies regarding when and how information can be sent, received, updated and accessed. These policies were created by browser developers and the committees in charge of the web standards in order to protect users and secure applications. In following these policies, browsers may modify the behavior of protocols and applications by not

allowing a message to be sent or read by the web page that originated said message. In other words, the browser may restrict the execution of a protocol or application in case one or more of its policies are broken; which in turn means that browsers have Turing-complete interpreters with, few if any, restrictions (posed by policies like the Content Security Policy and Same Origin Policy) on making asynchronous requests.

Before going further we remark what we have said previously. Servers will be identified by an URL, and by design we only consider one display, one user and one browser.

5.2.1 Information Contained Inside The Browser

When we think of the web and web browsers we are immediately reminded of *web pages*. As such, the state of a browser involves information it has already received from servers, namely: a set of web pages, the current web page being presented to the user via the display, and some other pieces of information to be discussed later on in the text. We will start by defining a web page.

We define web pages as containing 3 things: an URL, a list of instructions, and browser components. The URL denotes the origin of the web page. The list of instructions represent scripts, links, and forms to be executed, clicked or submitted respectively. The components represent the Document Object Model (DOM). As can be seen from this informal definition of web pages, scripts have been constrained to represent messages and will not affect in other ways what is being presented to the user. Additionally, instructions are not be able to modify in any way either the components or other instructions.

5.2.1.1 Instructions

We shall describe each of the parts of web pages, one at a time, to then be able to give a formal definition. An instruction, represented by *ins*, is a conditional rule: $\langle trigger, rule \rangle$, where *trigger* is a predicate on whether an input has arrived to the browser, and *rule* an action to be carried out by the browser. Each rule includes a type, $t \in \{\text{normal}, \text{script}, \text{frame}, \text{resource}\}$, an HTTP method, $m \in \{\_\text{GET}, \_\text{POST}, \_\text{PUT}, \_\text{DELETE}\}$, an URL, and an association list of name and value strings, *Ldat*. Executing an instruction means to send the message specified by the rule. As expected, the type in the rule is used by the browser to trigger policies depending on the responses and the method may be useful for the server depending on the protocol it is implementing. More formally:

Definition 15 (Instruction). *Let trigger be a predicate on whether an input occurred, t represent the type of a request to the server, m represent the HTTP method, and let Ldat represent a list*

of name-value tuples. Then an instruction *ins is a conditional rule of the form: trigger* $:\Rightarrow \langle t, m, url, Ldat\rangle$.

We will say that an instruction is executed whenever the precondition trigger *evaluates to true. Further, an instruction is* mandatory *if it is always executed upon being received by the browser inside a response.*

As we have said, the instructions allow us to model scripts, redirections and to ask for extra resources, and the rules themselves allow us to check the policies if present.

5.2.1.2 Browser Components

A browser component is given by an URL, a list of instructions and whether or not it should be visible to the user via the display. More formally, let $\mathcal{C}$ be the set of all possible browser components, ranged over by $b_0, b_1, \ldots$, then:

Definition 16 (Component). *A* browser component b *is a 4-tuple of the form:* $\langle origin, pos, Linst, visible\rangle$ *where origin is an url* $\in \mathcal{U}$*; pos is the position it will have on the display; Linst is a set of instructions ins none of which is mandatory; and visible* $\in \{\top, \bot\}$ *represents whether the element is* visible to the user *on the display or not.*

Instructions will be executed whenever the trigger holds even if the predicate has nothing to do with the particular component it is contained in, allowing us to represent scripts and how their executions are chained. The URL in a web page and that of a component allows us to check for the origins of the components and the web page and decide whether something should be included in a web page or not.

As expected there is a close relation between browser components ($b \in \mathcal{C}$) and user components ($v \in \mathcal{V}$). This relation uses the transform function that maps one visible browser component b to a component v, this relation is further captured as follows. Let $\exists! c \in C.\, R$ be an abbreviation of $\exists c. \in C \wedge \forall c' \in C.\, (R(c) \rightarrow c' = c)$; that is, $\exists! c \in C.\, R(c)$ denotes that there exists one and only one element c in C such that $R(c)$ holds. Further, let visible(b) denote that $b \in \mathcal{C}$ is visible to the user, or visible for short. Then, $\forall b \in \mathcal{C}.\ \text{visible}(b) \Rightarrow \exists! v \in \mathcal{V}.\ v = \text{transform}(b)$, in other words, for each of the visible browser components there is one and only one user component v. This unique user component v is obtained from using the function transform on an element b of $\mathcal{C}$. Notice that transform produces at most one user component v; however, there is no function that would allow for the contrary, *i.e.* mapping a component v to a single browser component b.

Lemma 2. *The Browser cannot map a user input denoting the interaction with a user component v to an interaction with a single browser component b.*

Proof: Proving this lemma is trivial if we follow Definitions 12 and 16 and the previously explained relation. The user may click on any visible component in the display; however, by definition there is no function that maps from a single user component v to a single browser component b.

Given that there is no mapping from a single user component v to a single component b our browser model assumes that the user interacted with all of the browser components at the same position and executes the corresponding triggered instructions.

Browser Components are arranged in a tree, just like the DOM tree, and thus there are nodes with parent and children, and leaf nodes with only a parent node. The root node of the browser components is an invisible empty element with an position 0. A restriction on trees is that $\forall b \in \mathcal{C}.\ \neg root(b) \Rightarrow \exists par \in \mathcal{C}.\ (par \neq b \wedge \text{parent}(par, b))$ where $\text{parent}(par, b)$ is a predicate that holds whenever par is the parent of b, and $root(b)$ return true iff node b is root. When a click or form submission event arrives the browser will check the components which position corresponds to the position indicated by the event for instructions to be carried out. After checking the related components it will "bubble up" the search to its ancestors. Finally it will check other unrelated components that should have access to the information (*e.g.* inside the same frame) for instructions. All of the instructions will be queued in that order. When more than one trigger holds at the same time, all of the instructions will be queued for execution and they will be ordered in the same manner as in the previous explanation.

The inclusion of visible and invisible browser components helps us formalize user interaction with both visible and invisible things thus allowing us to represent click-jacking attacks where the user thinks she is interacting with something but does so with another things while organizing the components in a tree allows us to represent and manage the effects of both frames and scripts.

5.2.1.3 Web Pages

With this said, a web page is defined as follows:

Definition 17 (Web Page). *A* web page *denoted by w is a tuple: $\langle url, csp, Elem, inst\rangle$, where $url \in \mathcal{L}$, csp is a tuple $\langle S\_wlist, F\_wlist, R\_wlist\rangle$ containing three sets of strings denoting where scripts, frames and resources can be loaded from and if they can be executed, $Elem$ representing the DOM-tree, $inst$ is a tuple $\langle cinst, auto\rangle$ containing a two sets one of conditional instructions $cinst$ and another of mandatory instructions $auto$.*

The previously mentioned instruction lists *cinst* and *auto* are important because they allow the browser to send and receive several messages in order to compose a single web page (including

the display d for the user, as presented on Section 5.1), and to behave differently depending on the inputs received.

So far, we have defined web pages and how they are important for shaping the interaction with both a user and the servers. However, we still need to define some other important pieces of information that are contained inside the browser and thar further shape these interactions.

5.2.2 The Browser State

Apart from the stacks of web pages and the web page currently displayed, the state of a browser contains a number of files which allow us to represent things like cookies. Each file is a piece of information that servers have sent to the browser and are to be kept by the browser as long as the file has not expired. Let a file be denoted by fl, then fl is a triplet $\langle origin, ttl, ltxt \rangle$ where $origin$ is an $url \in \mathcal{L}$, ttl is a time j marking the expiration of fl and $ltxt$ is a list of name-value pairs.

The current state of the browser is defined as follows:

Definition 18 (Browser State). *Let $bkList$, $fwList$ be lists of the previous and forward web pages respectively, w be the current web page being displayed, $oweb$ be the received web page before the execution of any instruction, $Bfiles$ a set of files fl, rl a list of events or rules to be executed out, and Rq a set of expected messages, then the* state of the browser B *is given by:*

$$B \stackrel{def}{=} \langle bkList, fwList, w, oweb, Bfiles, rl, Rq \rangle$$

The browser will always be on an valid state since we do not model errors in the channel. The data currently known by the browser is given by the mix of the previously known data per Definition 18, the previous input by the user, and the response to the triggered instruction.

We note that files, web pages, and lists of instructions are not static and are updated with each input and output; this means that the list of files $lfiles$, and the web page w in the current state of the browser are composed by communicating several times with servers. It is because of this constant change that the information the user and the servers get is just a subset of all the information the browser possesses at any given time.

5.2.3 Messages for Browser Communication

So far we have defined the state of the browser, we will continue by defining the way the browser interacts with the users and the servers. To do so, we will start this discussion by defining the inputs to the browser and what they mean.

5.2.3.1 Browser Inputs

We can think of the browser as an interpreter for two kinds of input events, the kind of event depends on where the input comes from. In order to be clear on the type and contents of events, they will be represented by their type of payload and thus be identified by an i when they come from the users and by a *Response* denoted by res when they come from the servers.

Definition 19 (Response). *Let oID be the unique identifier of a participant, used by a server to communicate with the appropriate party, let csp be a tuple $\langle S\_wlist, F\_wlist, R\_wlist\rangle$ containing three sets of strings, let Elem be a set of elements b to be added to the DOM, inst be a tuple $\langle cinst, auto\rangle$ containing a list of conditional instructions cinst and a list of mandatory instructions auto to be executed upon retrieval, and let $Lfiles$ be a set of files fl. Then, the type of a* response *res is a 5-tuple of the form: $\langle oID, url, csp, Elem, inst, Lfiles\rangle$.*

5.2.3.2 Browser outputs

Similarly, a browser has two kinds of outputs, one for the users and another for the servers. The output to the user is a display $d = \text{display}(B_j)$ which is calculated instantaneously, where display is a function on the current state of the browser (*e.g.* it calls transform on all of the browser components to produce the set V included in d), and the output to the servers is a *request.*

The output to the servers is a little more complicated than the output for the users. This is because while the output to a user is basically a combination of either several responses or at least parts of them, the output for a server needs to be calculated from all of the past responses received by the browser. Formally:

Definition 20 (Request). *A* request *req is a 4-tuple: $\langle oID, url, m, ldata\rangle$, ldata is the result of applying the function* list2set$(Ltxt^\frown ldat^\frown \text{file}(B, url))$ *where the function* list2set *acts as the natural definition and converts a list to a set and works on the concatenation ($^\frown$) of ltxt from the user input, ldat from the triggered instruction and the results of* file(B, url)*, a relation that returns a list of name-value pairs representing the contents of files.*

Requests contain an origin identifier oID (its value is bID in the case of a request made by the browser), an URL $url \in \mathcal{L}$, an HTTP method m, and a list of name-value pairs $ltxt$. The identifier oID is required so that the server knows where to send its response, the URL $url \in \mathcal{L}$ so that we know which server the message should be sent to, the method m so that the server can manage the request and the list of pairs $ltxt$ is required to denote the message payoff.

As we have said before in Definition 15 instructions are executed when the condition in its trigger holds, and as such, most of the information in a request will be obtained from the instruction

which will be executed (*i.e.* the URL, the method, and part of the payload); however some other information is obtained from the user input (*i.e. ltxt*) and from the current state of the browser (*i.e.* the association list returned by file)

The browser will interpret all of the events and send most of the requests unless they go against one of its policies; however, interpreting an event and sending the corresponding messages does not mean that the results will be presented to the user or used in subsequent requests to servers.

We should note again that the browser does not make a decision on whether a protocol execution is correct or not and instead just sends some of the information it currently possesses via the display or when performing a request.

5.2.4 Security Policies and Browser Behavior

As we have said at the start of this section the policies shape part of the behavior of the browser and try to prevent attacks from happening by saying what requests and responses can be made and read. Roughly the policies behave in the following way.

The Content Security Policy (CSP) consists of various white lists that a web page provides in order to tell the browser where different requests can be made to. The policy requires, for every request going to be made, the browser to check the type of content that is being requested (media, frames, scripts, etcetera) and whether its location is in the corresponding white list for its type. In order to maintain backwards compatibility, if any of the white lists is empty it means that any request can be made for said content type.

The same origin policy (SOP) is a set of rules that define whether the browser can send a message or if the web page can access the data of a response. This policy basically checks whether a request is allowed by seeing if it is going to the same domain the web page is originated from or by sending a preflight request that asks for permission to the external domain. It also checks whether a response can be read or not. The latter rule is stricter and only allows the content of a response to be directly read if the response originates from the same domain that the current web page being displayed.

In order to explain in a clear way the policies and some of their effects we will define some functions that work over the different data types.

- url(*ins*) is a function that takes an instruction as an input and returns the URL to be called if said instruction were to be executed.
- server(*url*) is a function that takes an URL as an input and returns the string representing the server in the *url* tuple.

- type(ins) is a function that takes an instruction as an input and returns the type t of said instruction.
- $\text{wlist}_B(t)$ is a function that works over the current state of the browser per Definition 18, takes a type t as an input, and returns a set of servers.
- $\text{mandatory}_B(ins)$ is a function that works over the current state of the browser and takes an instruction and returns whether or not the instruction is mandatory.

Let CSP be defined as a predicate $\text{allowed}_B(ins)$ that denotes whether an instruction ins can be executed or not and be defined as follows:

Definition 21 (CSP).

$$allowed_B(ins) = \begin{cases} \top & if \begin{cases} \text{wlist}_B(\text{type}(ins)) = \emptyset \\ \vee\ (\text{mandatory}_B(ins) \\ \quad \vee\ t \neq \texttt{script}) \\ \wedge\ \text{server}(\text{url}(ins)) \\ \quad \in \text{wlist}_B(\text{type}(ins)) \\ \vee\ \neg\text{mandatory}_B(ins) \\ \wedge\ \text{type}(ins) = \texttt{script} \\ \wedge\ \text{server}(\text{url}(ins)) \\ \quad \in \text{wlist}_B(\text{type}(ins)) \\ \wedge\ \texttt{javascript://} \\ \quad \in \text{wlist}_B(\text{type}(ins)) \end{cases} \\ \bot & otherwise \end{cases}$$

Let SOP be defined as a predicate $\text{same\_origin}_B(ins)$, that denotes whether a triggered instruction will request something that complies with the policy and thus whether if the response can be fully read or not, be defined in the following way:

Definition 22 (SOP). *Let $server_B$ be the origin server of the current web page was loaded. Then SOP the predicate is defined as follows:*

$$\text{same\_origin}_B(ins) = \begin{cases} \top & if \begin{cases} \text{mandatory}_B(ins) \\ \vee\ \text{server}(\text{url}(ins)) \\ \quad = server_B \\ \vee\ \text{type}(ins) \neq \texttt{script} \end{cases} \\ \bot & otherwise \end{cases}$$

To continue our discussion on browser behavior we will proceed to define the state transition rules used by our method and describe how these rules affect the internal state of the browser.

5.2.5 State Transition Rules for the Browser

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). We will now present its transition rules, and describe its state update function, to better understand how the browser state changes through time.

We will start by discussion the transitions based on user inputs. Let enabled be a predicate on whether an *user_input* is feasible on the current state of the browser, head and tail be functions on lists that follow their natural definition, $\frown$ denotes the concatenation of two lists, and let $\uparrow$ take a tuple and denote the update from B to B' by modifying the elements in said tuple. The most simple examples are those of what happens when the user clicks on the back or forward buttons of the display:

$$
\begin{aligned}
\_Back.B\ \neg\text{enabled}(B, \_Back) &\rightarrow B \\
\_Back.B\ \text{enabled}(B, \_Back) &\rightarrow B' \uparrow \langle bkList' = \text{tail}(bklist), \\
&\qquad fwList' = [w] \frown fwlist, \\
&\qquad w' = \text{head}(bkList), \\
&\qquad oweb' = \text{head}(bkList), \\
&\qquad rl' = \text{wRules}(w') \frown [], Rq' = \varnothing\rangle \\
\_Forward.B\ \neg\text{enabled}(B, \_Forward) &\rightarrow B \\
\_Forward.B\ \text{enabled}(B, \_Forward) &\rightarrow B' \uparrow \langle fwList' = \text{tail}(fwlist), \\
&\qquad bkList' = [w] \frown bklist, \\
&\qquad w' = \text{head}(fwList), \\
&\qquad oweb' = \text{head}(bkList), \\
&\qquad rl' = \text{wRules}(w') \frown [],\ Rq' = \varnothing\rangle
\end{aligned}
$$

Where, for example if a user clicks on the back or forward buttons when they are disabled on the display the browser will just ignore the input and will not update its state in any way. However, if the user clicks on said buttons and they are enabled the browser state will be updated in the following way: the corresponding (back or forward) list of web pages will be updated to contain just the tail of said list, the list corresponding to the contrary action will be updated to now contain a new element in the head corresponding to the original web page in the state

of the browser, the current web page and the original web page will now contain the head of the corresponding list of the action, the list of pending rules will emptied and be updated to contain only the mandatory rules on the new current web page, and finally the response queue will be emptied.

Let us continue by providing the remaining rules for user input. Let newRl be a function on the state browser and an URL input that returns a new instruction to be queued for execution, rules be a function on the state of the browser and a position that returns all of the possible instructions to be carried out given an input in said display position.

$$\begin{aligned}
url.B &\rightarrow B' \uparrow \langle rl' = \text{newRl}(B, url) \frown [], Rq' = \varnothing \rangle \\
pos.B\ \neg\text{enabled}(B, pos) &\rightarrow B \\
pos.B\ \text{enabled}(B, pos) &\rightarrow B' \uparrow \langle rl' = \text{rules}(B, pos) \frown rl \rangle \\
\langle txt, pos \rangle.B\ \neg\text{enabled}(B, pos) &\rightarrow B \\
\langle txt, pos \rangle.B\ \text{enabled}(B, pos) &\rightarrow B' \uparrow \langle rl' = \text{rules}(B, pos) \frown rl \rangle
\end{aligned}$$

This means that if a user gives a new URL then the rule queue will be emptied and now contain just a new rule corresponding to a request of the received URL and the response queue will be emptied; that if the browser receives an input corresponding to a disabled element it will just ignore the input; and that if the element is enabled on the display it will search of the current web page all of the possible triggered rules and will queue them for execution.

Now let us present the rules triggered by requests and responses. Let expected be a predicate on whether a response is being expected at the current state of the browser, refresh be a predicate on whether a response is supposed to refresh the current web page or just update it, fullreq be a predicate on whether a response to a given request will refresh the contents of a web page or just update it, wRules be a function that takes a web page and returns a list of mandatory instructions, files be a function that takes a response and returns the list of files included in said response, webp a function on a response that returns a web page crafted from said response, rRules be a function that takes a response and returns a list with all the mandatory instructions included in said response, reqToRq a function that transforms an instruction into a singleton containing the corresponding structure that accepts responses, + be a function that takes two compatible types and returns the union or concatenation of the analogous components. Finally, let $\_Empty_w$ be an atom that represents an empty web page, then the rules the browser follows

for request and responses are:

$$
\begin{aligned}
res.B\ \neg\text{expected}(B, res) &\rightarrow B \\
res.B\ \text{expected}(B, res) \wedge \text{refresh}(B, res) &\rightarrow B' \uparrow \langle lfiles' = lfiles \cup \text{files}(res), \\
&\quad\ w' = \text{webp}(res), \\
&\quad\ oweb' = \text{webp}(res), \\
&\quad\ rl' = rules(res),\ Rq' = Rq/res\rangle \\
res.B\ \text{expected}(B, res) \wedge \neg\text{refresh}(B, res) &\rightarrow B' \uparrow \langle lfiles' = lfiles \cup \text{files}(res), \\
&\quad\ w' = w + \text{webp}(res), \\
&\quad\ rl' = \text{rRules}(res) \frown rl, \\
&\quad\ Rq' = Rq/res\rangle \\
B.req\ \neg\text{fullreq}(head(rl)) &\rightarrow B' \uparrow \langle rl' = \text{tail}(rl), \\
&\quad\ Rq' = \text{reqToRq}(request) \cup Rq\rangle \\
B.req\ \text{fullreq}(head(rl)) &\rightarrow B' \uparrow \langle bkList' = [oweb] \frown bklist, \\
&\quad\ oweb' = \_Empty\_w, \\
&\quad\ w' = \_Empty\_w\ rl' = [], \\
&\quad\ Rq' = \text{reqToRq}(req)\rangle
\end{aligned}
$$

Where the first three rules correspond to what happens when the browser receives a response and the last two correspond to what happens when the browser sends a request. In the case of the first if the browser receives a response it is not expecting then it just ignores it; the second and third rules manage what happens when the browser receives a response that is expected and are similar. The main difference between the second and third rules is that if the response is to refresh the presented web page, then it replaces the original and current web pages; otherwise, the response just adds the contents to the web page and to the rule queue; in both cases the files are appended and the response is removed from the queue. When the browser sends a response several things may happen depending on whether the request is to refresh the whole web page or instead it is to add information to the current web page. In the case of the fourth rule (adding information), the browser will just remove the request from the list of pending requests and move it to the set of pending responses; while in the other case the browser will add the original web page to the list of previously visited web pages, empty the current and original web pages, empty the list of pending requests and replace the pending responses with a singleton corresponding to the request being made.

Let us conclude our discussion by defining the browser as a state event system $\mathcal{B}$:

Definition 23. *let In be an infinite set of responses and user inputs, Out be an infinite set of requests,* $E = In \cup Out$ *is then the union of those two infinite sets,* $B_0 = \langle \varnothing, \varnothing, \_Empty\_w, [], [] \rangle$*, St be a singleton with* B_0 *as its only element, and T be equal to the previously mentioned transition rules then:*

$$\mathcal{B} = \langle E, In, Out, St, B_0, T \rangle \tag{5.3}$$

5.3 Conclusions

As we have said, the browser is probably the most important and complex of all the participants in web protocols. It was modeled after carefully reading the specifications on how it should work since there is a vast amount of information and behaviors encoded on real world browsers. However, if a browser model were to encode all of the real world browser characteristics the resulting model complexity would make it difficult to automatically analyze and reason about the browser's interactions. In other words, we decided to simplify and abstract away some of the browsers' capabilities to avoid creating a model that would be too complex as it would prevent us from reaching a meaningful conclusion about the security of browsers or that of the protocols being analyzed.

So far we have seen an overview of the system, and explained in detail the models, the properties, capabilities and the respective rules that shape the behavior of both the user and browser in our method; however, this discussion is not enough to have a full understanding of how all of the pieces are put together, as such, it is time to think about the participants that lie outside of the host computer. That is, in order to be thorough and for you, the reader, to understand how our models fit together and how our method works we need to present the other half of the system. We will continue our discussion in Chapter 6 by presenting both the server model, one of the most distinguishing characteristics of our method, and the attacker model in WebMC.

Chapter 6

The Server and The Attacker Models for the Verification of Web Protocols

The server is the most important component, as well as being one of the distinguishing components, of our approach to the analysis of web protocols. The server is the only participant who has full access to the rules modeling the part of the protocol being instantiated to be analyzed. In order to avoid creating a model for several servers, one set of severs for each specific protocols, we rather designed a parametric model for a server, which is given as a set of general rules that may be instantiated depending on the needs of a given protocol. This has enabled us to promptly capture the description of several kinds of servers and it allows the end user of our method to think more on the contents of messages and the server interactions instead of thinking on all the messages and interactions that may arise from the behavior of the other components.

As we know from Chapter 4, the system is modeled as having an insecure channel as a way to communicate the browser and the servers. Due to the nature of the channel, the attacker can be seen as being its owner. The attacker should then able to intercept, drop, analyze, replay any message that goes through said channel as well as to synthesize arbitrary messages (*i.e.* the olev-Yao attacker). Moreover, while using the web we are not always sure if the servers we are communicating with are honest or have been corrupted in any way and thus may want to steal our information; as such, we modeled the attacker as also being able to corrupt servers in two ways. With these characteristics in mind, and since we will ignore availability issues, we decided to model the behavior of the attacker as a buffer that is able to create, replay and modify messages. The modification of messages depends on the attacker having access to the encryption keys (*i.e.* a server has been corrupted) while the creation of messages depends on the attacker having the necessary knowledge to create a valid message for either a server or the browser.

This chapter discusses then the second half of the system, the half that depends on the network in order to perform any kind of action. We will begin our discussion by exploring the last of the honest participnts in our model, the server, in Section 6.1. The discussion on the server consists on a discussion where we describe the general characteristics of cryptographic primitives used and by giving a formal definition of the server state in Section 6.1.1. After that, we will discuss how the server specifications work in Section 6.1.2. we conclude our discussion on the server, in Section 6.1.3, where we will provide a set of general rules that servers follow in order to participate in protocols. Then, in Section 6.2, we will concern ourselves with the attacker, its capabilities and how these capabilities are reflected in our method. The section on the attacker is structured as follows: In Section 6.3, we will explore the information used and stored by the attacker. After that, in Section 6.4, we analyze the capabilities of the attacker and discuss how these capabilities relate to those of the eponymous Dolev-Yao attacker. Finally, in Section 6.5, we conclude the discussion on the attacker by providing the transition rules used by the model in order to update the attacker's state.

6.1 The Server in WebMC

As we have said, Servers are the ones in charge of defining the protocol to be executed and deciding whether or not a received message corresponds to a protocol. In other words, formalizing the server is equivalent to formalizing a protocol. Servers for browser based protocols are the participants that resemble the most to a principal in other kinds of protocols; however, there is a big difference since web servers do not really track which step of the protocol they are currently at and what the next step is even when they can do so (they are said to be "stateless" in the web development community). Web servers do keep and state and this notion of "statelessness" does not imply that they do not store information that has been sent and received previously which in turn enables the server to have the notion of session.

We will now provide a parametric model for the server. The model is parametric in that it needs for the user of the method to specify how the requests, responses and information therein are to be handled. That is, the model needs an specification in order to work, it may not use all of the characteristics provided, and while the rules for modeling the server are independent of other components the traces of the overall system are restricted by the necessary sharing of events with different principals.

6.1.1 State of The Server

Cryptography should be transparent for the user and the browser; however, since the servers need to verify and act based on the data they receive, the servers need to be aware of a few

cryptographic primitives. We will use K_{id} and K_{id}^{-1} to denote the public and private keys for a principal with id *id*, K_{id,id_2} to denote a shared key between principals with ids *id*, and id_2. This means that servers may receive encrypted or signed data as part of any message, but will only be able to check or modify the data if they know the required keys to do so.

Now that we have explained the cryptographic primitives to be used we will proceed by defining the state of the server. The state of the server S consists of all of the information the server currently possesses including its rules to process *requests* and *responses*.

The state of the server S consists of all of the information the server currently possesses including its rules to process *requests* and *responses*.

Definition 24 (Server State). *The state of the server is a tuple containing:*

$$\langle sID,\ key\_list,\ known\_data,\ server\_info,\ session,\ persistent\_session,\ seen,\ pending\_requests,\ EResponses,\ pending\_responses,\ specification\rangle$$

Where sID is a unique identifier, key_list is an association list of keys to be stored by the server; known_data, server_info, and seen are lists of name-value pairs; session, and persistent_session are association lists of name-value pairs and requests; pending_requests, and pending_responses are lists of feasible outputs; EResponses is a set of expected responses; and finally, specification a tuple containing msgs an association list of symbolic request to be accepted along with the symbolic requests and responses to be made, PData a set of field names to keep across sessions, KData a set of field names containing keys to be stored, TData a set of field names to track in order to avoid replays.

This parametric server enables us to represent a broad range of characteristics and behaviors of web servers like transient and persistent sessions (*i.e.* sessions that end the moment the server sends the response and sessions which require for certain information to be stored for long term usage), key exchange, and keeping track of nonces and other values.

Transient sessions are modeled by using *session* in order to store information until a response can be sent, *pending_request* in order to queue one or more requests to be sent before the response can be made, *expected_responses* in order to keep track of the responses that are to be received from other servers, and *pending_responses* in order to keep track of the responses that need to be eventually sent to the corresponding destinations. Persistent sessions are represented by using *persistent_session* to store data across transient sessions along with *session*, *pending_request*, *expected_responses*, *pending_responses*. To reproduce key exchange and already known keys we use *key_list*. Finally to check against replay attacks by keeping track of already seen values we use *seen*.

6.1.2 Protocol and Server Specification

As we have said, the server is the most simmilar to a principal in security protocols, and so, by itself it is not really complex; however, if we want our tool to be useful for a wide range of protocols and applications we needed to make a model that could be instantiated with the different rules and needs of said protocols and applications. Giving a server specification is equivalent to specifying a protocol; however, our method does not currently possess an specification language and as such WebMC requires for the server specification to be done in the same language it was programmed (*i.e.* Haskell) and by using the data types we provide.

A server and protocol specification is a set of rules 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 by default (*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 is able to finish successfully or if it corresponds to an actual protocol. WebMC is able to check for well-formedness according to data types; however this does not guarantee that the protocol reaches completion. This means that both the method and WebMC may get stuck on a loop and never find an attack. This is why protocol analysis should start with proving the feasibility of protocol execution successful.

6.1.3 State Transition Rules for the Server

Once we have a server specification, we can instantiate the server and have working rules. However, servers will always be able to receive requests directed at them. As such, we will start by presenting the rules governing how requests are handled. Let S represent the state of the sserver at time j and S' at time $j+1$,valid be a predicate on whether the server considers a request being received as valid according to its specification, data a function that returns all of the data included in a request, pData a function on the server and a request that returns a list with information that should be persistent across sessions, tData a function on the state of the server and a request that returns a list with information that should be tracked to avoid replays, requests be a function on the state of the server that returns zero or more requests to be made according to the specification and the received request, response be a function on the state of the server and the received request that returns a response according to the specification. We will follow the same notation as with the browser and $\frown$ will denote concatenation of two lists,

head and tail will act as their natural definition on lists, finally an $\uparrow$ will denote an update of the state of the server. The rules for received requests are:

$$\begin{aligned}
req.S\ \text{destination}(req) = sID_S \wedge \neg\text{valid}(S, req) \rightarrow S' \uparrow \langle & pending_responses' = \\
& [\text{response}(S, req)] \frown \\
& pending_responses\rangle \\
req.S\ \text{destination}(req) = sID_S \wedge \text{valid}(S, req) \rightarrow S' \uparrow \langle & session' = \\
& \text{data}(req) \frown session \\
& persistent_session' = \\
& \text{pData}(S, req) \frown \\
& persistent_session \\
& seen' = \text{tData}(S, req) \frown \\
& seen \\
& pending_requests' = \\
& \text{requests}(S, req) \frown \\
& pending_requests \\
& pending_responses' = \\
& [\text{response}(S, req)] \frown \\
& pending_responses\rangle
\end{aligned}$$

This rules mean that if a request is received by a server and it is invalid then an error response will be calculated according to the specification and will then be added to the response queue; On the other hand if the request is valid then the information contained in the current session will be added to the information on the server, if any information is to be kept across sessions it will be stored, the tracked fields will be updated with the information in the request, zero or more requests will be added to the request queue in the server according to the specification and one response will be added to the response queue.

Servers, just as browsers, will only take into account responses they expect. As such we will continue by discussing the rules used by the server in order to handle the received responses. Let expected a predicate on whether a response is expected at a given state of the server, queued a predicate on whether a request or response is in the queue at a given state of the server (*destination*) be a function that takes an event and returns the unique identifier of the intended recipient, kData a function that takes as an input the state of the browser and a response and returns a list of the included keys in said response, rsData a function that takes as an input a response and returns a list of data about the server that generated the response, rData a

function that takes as an input a response and returns all of the data in the response, prData a function on the state of the server and a response that returns a list with information that should be persistent across sessions, trData a function on the state of the server and a response that returns a list with information that should be tracked to avoid replays, remove a function that takes a list and an element and removes said element from the list if there is one. The rules for received responses are:

$$
\begin{aligned}
res.S \text{ destination}(res) = sID_S \wedge \neg\text{expected}(S, res) &\rightarrow S \\
res.S \text{ destination}(res) = sID_S \wedge \text{expected}(S, res) &\rightarrow S' \uparrow \langle key\_list' = \\
&\quad \text{kData}(S, res) \\
&\quad \frown key\_list \\
&\quad server\_info' = \\
&\quad \text{rsData}(res) \\
&\quad \frown server\_info \\
&\quad session' = \text{rData}(res) \\
&\quad \frown session \\
&\quad persistent\_session' = \\
&\quad \text{prData}(S, res) \frown \\
&\quad persistent\_session \\
&\quad seen' = \text{trData}(S, req) \\
&\quad \frown seen \\
&\quad EResponses' = \\
&\quad EResponses/res\rangle
\end{aligned}
$$

Where the first rule tells us that if the server is not expecting a response it will just ignore it, and the second tells us the following in case a response is expected: the server will add all of the received keys to the known keys, will add the response data to the session and to the persistent session (if any) that originated the request-response pair, the values of any tracked fields will be marked as seen and the response will be removed from the response queue.

Finally we will discuss how the server manages its outputs. To do so, let relQueued be a predicate on whether if given a state of the server and a queued response, either *pending_responses* or *expected_responses* contain an element related to said queued response. Then, the rules for

server outputs are:

$$S.req; \text{queued}(S, req) \rightarrow S' \uparrow \langle pending\_requests_{S'} = \text{remove}(pending\_requests_S, req), \; EResponses' = EResponses \cup req2Res(req) \rangle$$

$$S.res \; \neg\text{relQueued}(S, req) \rightarrow S' \uparrow \langle pending\_responses_{S'} = \text{remove}(pending\_responses_S, res) \rangle$$

Where the first rule tells us that the server will remove from its queue the request and add a new expected response to the set of expected responses. And the second rule tells us that a response is only able to be sent if all of the requests in the session it belongs to have been answered and if so the server will send the response and remove it from its pending response queue.

Now that we have defined the state and the transition rules for the server. We define it as a state event system $\mathcal{S}$:

Definition 25. *let In be an infinite set or request with the server as its destination, Out an infinite set of requests and responses with the server as its origin, $E = In \cup Out$, $key\_list \neq \varnothing$, $known\_data \neq \varnothing$, $specification \neq \varnothing$, $S_0 = \langle sID,\ key\_list,\ known\_data, \varnothing, \varnothing, \varnothing, \varnothing, \varnothing, \varnothing, \varnothing, specification \rangle$, S be a singleton with S_0 as its only element, and T be equal to the previously mentioned transition rules then:*

$$\mathcal{S} = \langle E, In, Out, St, B_0, T \rangle \tag{6.1}$$

As we can see from the server rules, a server does not end the execution of a protocol by ignoring a request; a server simply sends an error response that depends on the protocol specification, and its up to the user to decide whether to end its execution or to continue interacting in another way. We must stress what we said in §6.1.2, the method is not able to differentiate between complete, incomplete, functional or non-functional specifications and thus it may get stuck in an infinite loop or never find an attack.

With the discussion on the server we conclude our exploration of three things: what happens at both ends of the communication channel, how are protocols instantiated and defined, and how honest participants behave and are modeled. We will now proceed to discuss the channel through which the honest participants communicate; that is, we will explore the capabilities of the attacker and how the attacker is able to influence protocol execution in order to reach insecure states.

6.2 The Attacker in WebMC

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 attacks we will model two kinds of server corruption. *Full corruption* in which the attacker has complete control over a server, and *Soft corruption*, or partial corruption, in which the attacker may add data and instructions to messages but cannot directly access or retrieve the information contained in the messages.

6.3 The Attacker State

The attacker state contains several things like 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.

Definition 26 (Attacker State). *let aID be a unique identifier, $FsID$ a set of server identifiers sID denoting the servers that are fully under the attacker's control, $SsID$ another set of server identifiers sID denoting servers that are partially under the attackers' control, $Keys$ be a set of public, private and symmetric keys, gen a set of fields which values will to be generated on-demand (*e.g. *nonces), $aInfo$ an association list that maps from the unique identifiers of honest principals (*i.e. *uID, bID, sID) to knowledge about said principals, $aFiles$ a set of files that the attacker has acquired by participating in sessions with servers, $known$ an association list of known names to their values, $reqQ$ and $resQ$ are a pair of lists representing request and response queues respectively; then, the state of an attacker is a 10-tuple containing $\langle aID, FsID, SsID, Keys, gen, aInfo, aFiles, known, reqQ, resQ\rangle$ where $FsID$ and $SsID$ and disjunct sets of servers*

6.4 The Kinds of Attacker in WebMC

As we have said, the network is an insecure channel controlled by the attacker. However, Making the attacker have the standard Dolev-Yao capabilities is somewhat troublesome since its capabilities deal with messages at a low level of abstraction compared to what would be needed in order to modify messages like requests and responses that have a defined structure. With this and our model in mind, we devised two types of attackers which we will explain in the following paragraphs.

6.4.1 Type One Attacker

The first type of attacker is a somewhat simple attacker. This attacker can corrupt servers; however, it may not send messages of its own to the different servers. In other words, the main goal of this attacker is to observe, gather and to include pertinent data and instructions in the messages that it has access to. The main goal of this attacker is to see if the data it is able to gather, either through several user sessions or by corrupting some servers, is enough to construct an attack on the user or on one of the honest servers.

6.4.2 Type Two Attacker

The second type of attacker is a fully active attacker. The second attacker must be able to at least send and receive messages to and from both servers and browsers; he also must be able to concatenate, separate, and create messages; finally he must be able to encrypt, sign, and decrypt messages and terms to which he possesses the adequate keys. As we can see, the actions of the Dolev-Yao attacker are simple yet powerful enough to describe what an active attacker needs to do; as such, in order to better understand and characterize the actions an attacker may perform we will give the Dolev-Yao 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 and accessible for the automation of attacks.

We pose that the attacker should be able to perform at least 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.
- Check the content of its sessions at any server.
- Act as the initiating party to any protocol.
- A combination thereof.

We designed the Attacker to create messages that correspond to a protocol since, as we have said, messages outside of the protocols would be useless and create an even larger search space. Also, as can be seen from the actions we described, we removed the possibility of type flaw attacks as well as the ability of the attacker to flush the channel and thus do not model or analyze any kind of denial of service attacks.

6.5 State Transition Rules for the Attacker

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. FsID, be a selector on the set of fully corrupted servers of the attacker, SsID, be a selector on the set of partially corrupted servers of the attacker, origin be a function that takes an event and returns the unique id of the originating participant, destination be, as previously defined, a function that takes an event and returns the unique id of the intended recipient, aCookies be a function that retrieves a subset of all of the cookies possessed by the attacker, aInstructions be a function that retrieves a subset of all the valid instructions the attacker is able to generate, new be a function that takes the information known to the attacker and produces all of the valid messages that can be sent to servers that are not corrupted, $\uparrow$ be a function that takes a an structure and tuple and updates the values of the structure accordingly. The rules the attacker follows are:

$$
\begin{aligned}
req.A\ \text{destination}(req) \notin \text{FsIDs}(A)& \\
\text{origin}(req) \notin \text{FsIDs}(A) \quad &\rightarrow \quad A' \uparrow \langle reqQ' = [req] \frown reqQ \rangle \\
res.A\ \text{destination}(res) \notin \text{FsIDs}(A)& \\
\text{origin}(res) \notin \text{FsIDs}(A) \cup \text{SsIDs}(A) \quad &\rightarrow \quad A' \uparrow \langle resQ' = [res] \frown resQ \rangle \\
res.A\ \text{destination}(res) = aID \quad &\rightarrow \quad A' \uparrow \langle known' = data(res) \frown known \\
& \qquad aFiles' = aFiles \frown \text{aCookies}(A) \rangle \\
req.A\ \text{origin}(req) \in \text{FsIDs}(A) \quad &\rightarrow \quad A' \uparrow \langle aInfo' = data(req) \frown aInfo, \\
& \qquad reqQ' = [req] \frown reqQ \rangle \\
res.A\ \text{origin}(res) \in \text{FsIDs}(A) \quad &\rightarrow \quad res' \uparrow \langle instructions' = \\
& \qquad \text{aInstructions}(A) \frown instructions \rangle \\
& \qquad A' \uparrow \langle aInfo' = data(res) \cup \\
& \qquad aInfo, resQ' = [res'] \frown resQ \rangle \\
res.A\ \text{origin}(res) \in \text{SsIDs}(A) \quad &\rightarrow \quad res' \uparrow \langle instructions' = \\
& \qquad \text{aInstructions}(A) \frown instructions \rangle \\
& \qquad A' \uparrow \langle resQ' = [res'] \frown resQ \rangle \\
A.req\ req \in reqQ \cup new(A) \quad &\rightarrow \quad A \uparrow \langle reqQ' = tail(reqQ) \rangle \\
A.res\ res \in resQ \quad &\rightarrow \quad A \uparrow \langle resQ' = tail(resQ) \rangle
\end{aligned}
$$

These rules have an interpretation like the following: If the attacker receives a message whose intended destination is different from one of the fully corrupted servers the update function will just add it to the corresponding request or response queue ($reqQ$ or $resQ$). On the other hand, if the intended destination of the message is a fully corrupted server, the update function will take all of the information and add it to $aInfo$ and file said information under the origin of the message. Another example is what happens when the attacker receives a response with an origin corresponding to that of a fully corrupted server, in this case the attacker will take all of the information and add it to $aInfo$ and file said information under the destination of the message, and will then modify the response to contain a subset of all the cookies and instructions the attacker may add to a response.

With this we have defined the state and the transition rules for the attacker. Now we will define it as a state event system $\mathcal{A}$:

Definition 27. *let In be an infinite set or request and responses with either the attacker, the browser or one of the servers as its destination, Out an infinite set of requests and responses with either the attacker, the browser or one of the servers as its origin,* $E = In \cup Out$, $aID \neq \varnothing$,

$aInfo = \varnothing$, $aFiles = \varnothing$, $reqQ = varnothing$, $resQ = \varnothing$, $A_0 = \langle aID, FsID, SsID, Keys, gen, aInfo, aFiles, known, reqQ, resQ\rangle$, St *be a singleton with* A_0 *as its only element, and* T *be equal to the previously mentioned transition rules then:*

$$\mathcal{A} = \langle E, In, Out, St, A_0, T\rangle \tag{6.2}$$

6.6 Conclusions

The server in our systems has two main purposes, one is to guide the protocol executions by giving an adequate answer to all of the request it receives and the other is to be the participants in charge of interpreting and instantiating protocols. Without servers our system would be unable analyze protocols or be of any use. We must stress what we said in §6.1.2, the method is not able to differentiate between complete, incomplete, functional or non-functional specifications and thus it may get stuck in an infinite loop or never find an attack.

With this discussion of the attacker and the server we have now finished explaining our method, the models and behavior of the participants used in order to represent and analyze browser based protocols. To summarize, our method and models allow us to represent four different participants in browser based protocols. A user that owns information and can be misled into interacting with components, and servers it did not intend to. A parametric server that can be instantiated, and changes its behavior based on the specification provided. A Browser that behaves according to security policies, presents information to the user and interacts with servers through the network. Finally and attacker that owns the network, can modify and craft messages, corrupt servers and act as an initiator or as a server in protocols. In the next chapter we will proceed by explaining how the discussions in Chapters 4 to 6 fit together in order to create an automated tool for the verification of web protocols, and by verifying the usefulness of our method.

Chapter 7

WebMC

The method we propose, with what we have discussed so far, is able to represent and analyze the security of protocols; however, a method by itself falls short of an automated system. We will use this chapter to achieve our goal of creating an automated system. As we can see from Chapters 3 and 4 the main use for a tool implementing our method should be 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, in order to achieve our goal we will present and discuss our tool. A tool that is able to encode properties relating to secrecy and confidentiality in order to find counterexamples to Lemma 1.

Based in our model and method we have developed WebMC, 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 previously defined attacker model. 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. In order to select the branch most likely to contain a counter example we use heuristics (to be defined on the sections regarding the implementations).

This chapter is structured as follows. In Section 7.1 we talk about the ways in which our tool is implemented. Then, in Section 7.3, we give a full example of how our tool works by presenting our analysis of the WebAuth protocol (a version of the Kereberos protocol for the web). Finally, in Section 7.4, we present all of our experimental results.

7.1 The WebMC implementation

In order to test our tool and search for attacks on protocols we needed to create a way in which the system is animated so that each of the modules, corresponding to the different principals, interact with each other. The first result of this, to be discussed in Section 7.1.1 is a module in charge of generating the search space on the fly that allows the users to explore the different ways in which a protocol's execution can take place. Our second result is an automatic tool which will be discussed in Section 7.2.

7.1.1 Interactive Tool

The interactive implementation of WebMC was programmed so the user is able to elect 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.

As we have said in Chapter 4 we decided to model the interactions between the different agents as atomic events, this in order to avoid having wait states in which nothing evident happened. However, for the sake of clarity our implementation of the attacker's behavior is slightly different from that presented in Chapter 6. The main difference being that in our implementations we decided to include the changes the attacker makes to messages as its own set of atomic 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.

7.2 Automatic Tool

After completing the interactive tool and concluding we were actually able to find previously reported attacks with our tool, we decided to implement a module to find these attacks automatically. The approaches taken and how they were developed are presented in the following paragraphs in the order in which they were implemented.

7.2.1 Iterative Deepening

Our first attempt at automatically finding attacks to protocols was implemented as an iterative deepening search with a maximum depth level provided by users. When using iterative deepening the tool searches the tree as long as it has not found any attacks, as soon as an attack is found the attack trace is returned to the user.

For the iterative deepening version of the tool we used the type one attacker; and as we know, the order in which the branches are selected for search is important for the performance of this kind of search. Its because of this that we implemented the iterative deepening search in two ways. For the first implementation we used a naïve ordering that presents the options starting with the possible user actions, then the browser's, and finally the servers' actions; this implementation while giving us the expected results has a somewhat poor performance since it took more than 30 seconds to find an attack on a simple protocol even when it did not have to take into account most of the type two attacker's possible actions.

For our second implementation of iterative deepening, we ordered the possible actions in a way that leads to the normal execution of a protocol; we start by listing the servers' actions, then continue with the browser's and finally list the user's actions. This second approach to ordering had a much better performance, taking about one second to find the reported attack; however, it still lacked the information corresponding to the type two attacker's actions.

Both of the implementations of iterative deepening for the automatic search have poor performance in the worse case scenario (*i.e.* there are no attacks at the expected depth) and take almost 3 hours to fully search a tree with a maximum depth of 15.

7.2.2 Hybrid Search

Our second attempt at automatically finding attacks to protocols was implemented as an hybrid search with no hard maximum level. Just as with the iterative deepening version, the tool searches the 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. However, in this version of the tool we included the two types of attacker, letting the person in charge of the specifications choose the attacker best fitted for the task, and implemented an heuristic that reduced the search space almost in half by making the user component unable to tell the browser to request urls that require more information than that available and by forbidding the execution of the same action twice in a row.

The heuristic we implemented gives a weight between 0 and twice the depth level to the attacker actions, a weight of twice the depth level to server actions, a weight of 24 times the depth level

to the browser actions, and a weight of between 34 and 144 times the depth level. 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.

The performance for this version of the tool is much better 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 implementing a function that detects and avoids the exploration of states equivalent to those already explored (a technique used in Binary Decision Diagrams).

So far we have explained how WebMC searches for attacks but this is not enough to give a full idea of the way in which it works. As such, we will now proceed to give a detailed example of how a protocol is specified and how the tool should be used.

7.3 Case: The WebAuth Protocol

We have discovered what we consider to be an attacks to the WebAuth protocol. WebAuth (see Figure 7.1) 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 user will not need to authenticate again until their credentials expire, even if they visit multiple protected services. The attack to WebAuth (see Figure 7.2) consists on reusing the cookie provided by the authentication server in order for multiple an honest user use services in the name of the attacker; 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.

7.3.1 WebAuth 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 servers and the attacker for the WebAuth protocol were

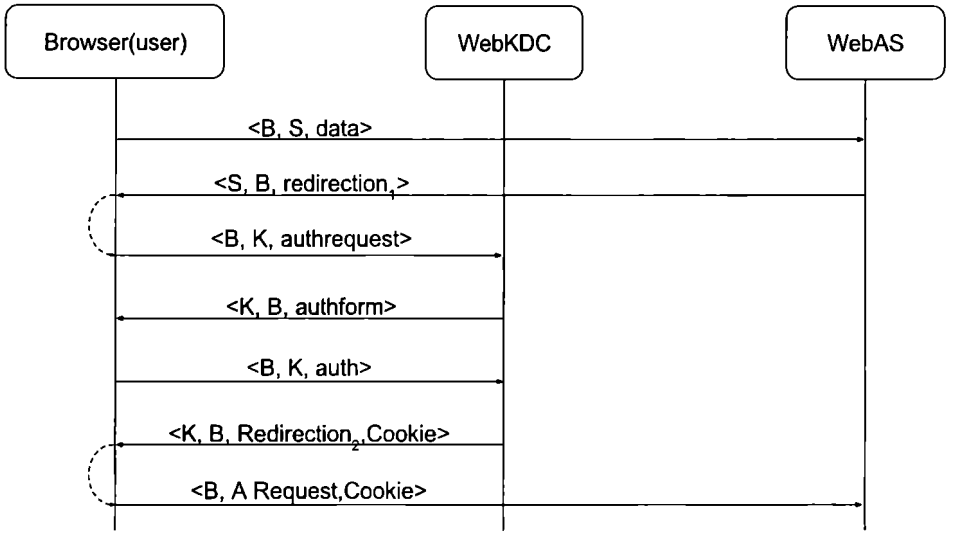


FIGURE 7.1: The WebAuth protocol

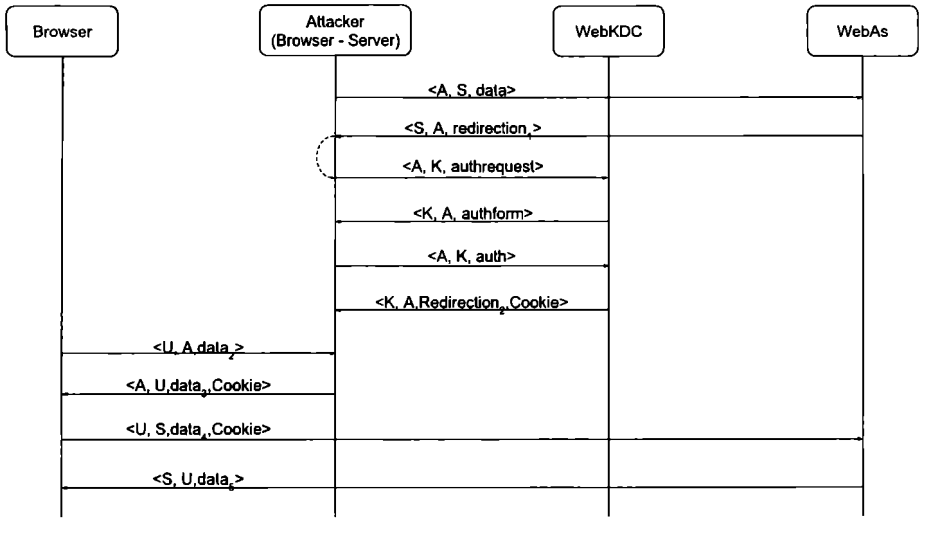


FIGURE 7.2: Attack to the WebAuth protocol

specified. In order for specifications to work we will use a header, like the one in Listing 7.1, 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.

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

LISTING 7.1: Header for specifications, including the name of the protocol to specify

After providing the header we need to proceed to specify the servers. As we can see in Figure 7.1, 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).

```
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 = [],
                      instructionList = PageInstructions { autoList = [inst1],
                                        conditionalList = [] },
                      fileList = Map.empty}
          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],
                        instructionList = PageInstructions { autoList = [],
                                          conditionalList = [] },
                        fileList = Map.empty }
          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) ]) ]
```

LISTING 7.2: Specification for the Service Provider Server in WebAuth

```
webKDC:: String -> Server
webKDC cName =
    initEmptyServer cName auto pdata [] [] [] known ruleMap
    where auto = ["credentials"]
          pdata = ("id_token", ["cbUrl"])
          known = Map.fromList [("user", ["uname"]), ("pass", ["pass"])]
          url1 = Url cName "one"
          url2 = Url cName "two"
          pKey = Pri cName
          inst1 = Instruction (Left 1) Rule { rType = RuleType Normal Full,
                    rMethod = Post, rUrl = Left url2,
                    rContents = Map.fromList [("user", "?"), ("pass", "?"),
                                ("cbUrl", "")] }
          component1 = Component { cOrigin = url1, cList = [inst1], cPos = 1,
                       cVisible = True}
          response11 = Response {destinationIdentifier = "", origin = url1,
                       resNonce = "", csp = emptyCSP,
                       componentList = [component1],
                       instructionList = PageInstructions { autoList = [],
                                         conditionalList = [] },
                       fileList = Map.empty }
          inst2 = Instruction (Right True) Rule { rType = RuleType Normal Full,
                    rMethod = Post, rUrl = Right "cbUrl",
                    rContents = Map.empty }
          f1 = WebFile 3600 $ Map.singleton "credentials" ""
          f2 = WebFile 3600 $ Map.singleton "id_token" ("Sig credentials "++ show pKey)
          fm = Map.fromList [(Left "cbUrl", f2), (Right (Url cName ""), f1)]
          response2 = Response {destinationIdentifier = "", origin = url2,
                       resNonce = "", csp = emptyCSP, componentList = [],
                       instructionList = PageInstructions { autoList = [inst2],
                                         conditionalList = [] },
                       fileList = fm }
          ruleMap = Map.fromList [
                (url1, [(["cbUrl"], [], response11, Nothing),
                        (["credentials", "cbUrl"], [], response2,
                          Just response11),
                        (["crendentials", "reauth", "cbUrl"], [], response11,
                          Nothing)]),
                (url2, [(["user", "pass", "cbUrl"], [], response2, Nothing)])]
```

LISTING 7.3: Specification for the Identity Provider Server in WebAuth

```
aServer::String -> Server
aServer cName = initEmptyServer cName ["id_token"] ("",[]) [] [] [] Map.empty ruleMap
    where url1 = Url cName ""
```

```
response1 = Response {destinationIdentifier = "", origin = url1,
        resNonce = "", csp = emptyCSP,
        componentList = [],
        instructionList = PageInstructions { autoList = [],
                          conditionalList = [] },
        fileList = Map.empty }
ruleMap = Map.fromList [ (url1, [([], [], response1,
                          Just response1)])]
```

LISTING 7.4: Specification for the Attacker Server in WebAuth

The code inside Listings 7.2 to 7.4 corresponds to the specification of the webAS server specification, that of the webKDC server, and that of the aServer specification respectively. 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, the 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 the corresponding fields empty to tell 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 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.

Finally, after defining the behavior of our servers we need to instantiate, provide the security goals and define the attacker to be used by our tool. To do so we will use the getServers and secondGoal functions, as in Listing 7.5. The 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 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.

```
getServers :: ([Server], [Either Request Response], Attacker)
getServers = ([kdc, was, aServ], goals, myAttacker)
```

```
    where kdc = webKDC "kdc"
          was = webAS "was" "kdc"
          aServ = aServer "att"
          --url1 = Url { server = "was", path = "one" }
          cbUrl = Url { server = "was", path = "two" }
          --aUrl = Url { server = "att", path = "" }
          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}
```

LISTING 7.5: Sever and Attacker instantiation, Specification of the goals for the WebAuth protocol

As we can see in Listing 7.5, we are instantiating one webKDC server, one webAS server, and an 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, the 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 getServers function we specify another function which will be called after the first goal is reached. The 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.

Once the servers, the goals, and the attacker have been specifies 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 7.6 and 7.7. The first part, in Listing 7.6, corresponds to the header. In the header of our entry point we should include the following: 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, and finally the data types of our method and language libraries that help us measure and execute our program.

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

LISTING 7.6: Header for the file to use our specification and execute the tool

The second part of our entrypoint program, as presented in Listing 7.7, 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 program.

```
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 []
```

LISTING 7.7: Body of the program in charge of executing and defining User Knowledge

7.3.2 The results of our WebAuth Test

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 7.2) like the one in Listing 7.8 or a message saying that it reached its iteration limit and that no attack was found for the protocol being analyzed.

```
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!
```

LISTING 7.8: WebMC Sample Output for the WebAuth protocol attack

In the case of the WebAuth protocol we present in Listing 7.8 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 7.2 in order to reach the point in which the user is able to send the attack message.

In the following section we will proceed to discuss our other experiments with the tools and our findings. We must note that there are some other examples of how protocols have been formalized in the WebMC github repository `https://github.com/vferman/webmc`

7.4 Experimental Results

As we can see, while proving security in our method is roughly equivalent to encoding security properties in a set LTL formulas and then proving these formulas hold; we took a more straight

TABLE 7.1: 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	NA	NA
Sub Goals	<2s	
Session Establishment	<1s	NA
Session Fixation	NA	<1s
SAML-Fix	NA	NA
OpenID 1.0	NA	NA

forward and slightly different path in order to create our tool. The properties our tool looks for are encoded as attacks, and we use something similar to branching time since at any moment more than one action may be carried out. In order to avoid the complexities of branching time, our tool uses an heuristic to select and analyze only one branch at a time. Since calculating on the fly all of the possible execution paths causes the tree to be infinite, we also encoded a soft limit on the depth of the tree which corresponds to a point were we consider its not useful to continue searching.

So far we also have analyzed and reproduced the results of [7] 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 [33], 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 7.1, 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 7.3) 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 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

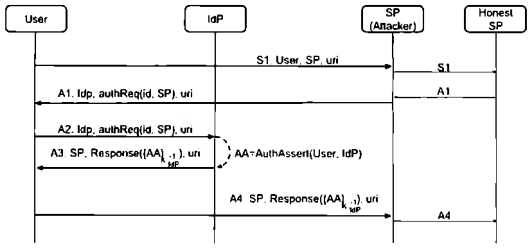


FIGURE 7.3: The SAML protocol and an attack

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

We can see in Table 7.1 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.

7.5 Conclusions

As we can see, WebMC is able to reproduce the results of several research groups. Moreover, our tool is not only able to find attacks reported as theoretical but also attacks found by security researchers that focus on existing exploits being used. With this chapter we conclude the discussion on our system and tool, thus leaving us with the task of discussing our conclusions and how our method and tool could be improved in order to be friendlier and find other kinds attacks.

Chapter 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 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 model like the one presented would not only lead to a better understanding of protocol properties and allow us to analyze potential security vulnerabilities and shortcomings they 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 method and tool.

This chapter is structured as follows: We will first, in Section 8.1, talk about how our method and tool can be extended in order to increase its usefulness. Then, in Section 8.2, we will proceed to present the conclusions that can be excepted from this work.

8.1 Future Work

As we know 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. As such, in here we present some characteristics and design choices we made for our browser model, how these design choices affect our analysis, and how some limitations due to complexity can be eliminated or mitigated.

Some complex characteristics of the browser model we have found so far are the following:

- Accessing files within the host computer.
- Modifying previous instructions.

- Calculating values to be sent.
- Inter-Frame communication.

and will be analyzed in more depth in the following paragraphs.

8.1.1 Accessing Files Within the Host Computer

As we know browsers work above an operating system with a file system in place. This means that a browser implementation is able communicate with the users, the operating system, the file system (*i.e.* read and write local files), and through the network with one or more servers.

In order to simplify the structure of our model we decided to ignore the way the browser is able to communicate with the operating system and the file system while keeping the ability to communicate with users and servers; however, doing this leaves a gap in our analysis (*i.e.* we cannot analyze attacks that rely on implementation bugs or browser features like XSLT transformations that may depend on where the document is being read from) that may be used by attackers [34] and thus could be important to secure protocols and applications, and also to ensure that our analysis correspond to reality.

The fact that our model cannot access the local file system means that we will not be able to analyze attacks relying on specially crafted files with a payload only executed when off-line, or how local and Internet content may be mixed, and thus the interaction of on-line content with the local system.

We believe that this limitation can be mitigated if we include a special kind of server with a different set of policies (also contained in the HTTP, and HTML standards). This local server would basically act as an oracle for the browser, represent the file system, and allow the browser to read and write files.

8.1.2 Modifying Previous Instructions

Web browsers can execute scripts. Scripts allow web pages to change dynamically and send different requests to different servers depending on user inputs, script interaction or the environment itself. The dynamic nature of the web is really useful; however, it also brings its share of problems specially when dealing with code and behavior analysis.

Scripts in the real world are able to modify themselves, and thus modify web pages' behavior although it may not be apparent from the users' perspective. This characteristic of scripts makes it difficult to completely analyze its execution since security conditions that held true at

one point may not do so in the future. As such, we decided that our model will not be able to accommodate scripts that can be altered somehow.

The decision to model a browser where a script cannot modify itself means that instructions may be added and increase without a limit until a new page is loaded, and that we might need to modify protocols in order to make them conform to how our model behaves (instructions must contain all the information they need) since we might need information included in other requests to complete an instruction.

We believe that although this design choice can be eliminated, by for example having an index or list of instructions and thus other instructions might be able to refer to them and thus to locate and modify other instructions, doing so would increase the complexity of the model and of our analysis, and would cause more problems than the ones it solves. Hence, our system and tool will not deal with this problem and will require the use of structure that holds instructions and will increase in size until a new web page is loaded.

8.1.3 Calculating Values

Scripts and instructions are not really static. Most of the time scripts need either information from the user or calculate things that may be needed by the server but that cannot be included on the original response. Right now, our model is unable to differentiate between values that come in the responses and values that should be calculated by the browser. This also means that we cannot detect implementation errors in which the inputs are not sanitized.

We decided to avoid calculating values for simplicity; however, modifying the way in which our model behaves does not increase its complexity by a lot and gives us enough flexibility to increase the scope of our analysis. We propose that this limitation can be mitigated by modifying our data structures to include name-value pairs that mix normal strings when values come from the server with special values that indicate that something should be calculated by the browser. This new feature would also require an special mechanism that compares the values to the list of keywords and then assigns them a new calculated value.

Basically we would need to use a mechanism that is aware of where the instruction is located and that would replace special strings representing functions (*e.g.* hashes, nonces, timestamps, numbers, cookies, concatenations, encryption or decryption) to the corresponding values that should be sent to the servers (*i.e.* we do not expect these functions to calculate actual values but structures related to our method to replace the values function strings).

8.1.4 Inter-Frame Communication

As we know, some web pages may incorporate content that comes from more than one server. This mixed content can sometimes be in the form of media components (*i.e.* images, audio, and video) while other times it may require further interaction and encapsulation and the content will be mixed by using frames or pop-up windows.

Since the content in one these frames may need to send some information or data to its parent, a children, or a sibling frame, we need to be able to express said messages to model the interaction that takes place in some paths of the protocols we want to analyze. Further this will let us analyze and secure more of the web applications and protocols that we use in our everyday lives.

The main difficulty of implementing this feature comes from the combination of frame-policies and the characteristics of web pages themselves. We may for example have a web page that does not know it is in a frame and thus will not interact with messages sent to it, or we might have a web page in a frame that cannot interact with messages due to the browser policies on frames, or fully working web page that tries to communicate with the its parent and its sibling frames.

8.2 Conclusions

The results of our model and application are promising, and we are working on formalizing 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.

To summarize, or contributions are the following:

- A method for the analysis of browser based protocols and web applications.
- 3 models allow us to represent the different participants in web protocols (namely a user model, a browser model and an attacker model).
- A generic server model that can be instantiated, and changes its behavior based on the specification provided.
- WebMC: a tool for the verification of browser based protocols and applications

We must note that we model a user that owns information and can be misled into interacting with components, and servers it did not intend to. A Browser that behaves according to security

policies, presents information to the user and interacts with servers through the network. Finally and attacker that owns the network, can modify and craft messages, corrupt servers and act as an initiator or as a server in protocols.

We must also remark the existing differences between the approach we propose and that of others. Our method allows its users to automatically find attacks to browser based protocols. Additionally, it does not require for its users to give a full specification of each participant every time a new protocol or application needs to be tested. Another advantage of our method is that we made it into an standalone application, which means its user does not have to extensively alter the behavior of existing tools, formalisms and specification languages in order to deal with the characteristics of browser based protocols and web applications.

Bibliography

[1] Raul Monroy and Juan C. Lopez Pimentel. Formal support to security protocol development: A survey. *Computación y Sistemas*, 12(1):89–108, 2008.

[2] Luca Viganò. Automated security protocol analysis with the avispa tool. *Electron. Notes Theor. Comput. Sci.*, 155:61–86, May 2006.

[3] David Basin, Sebastian Mdersheim, and Luca Vigan. Ofmc: A symbolic model checker for security protocols. *International Journal of Information Security*, 4(3):181–208, 2005.

[4] Barbara Guttman and Edward A. Roback. Sp 800-12. an introduction to computer security: The nist handbook. Technical report, Gaithersburg, MD, United States, 1995.

[5] Thomas Groß, Birgit Pfitzmann, and Ahmad-Reza Sadeghi. Browser model for security analysis of browser-based protocols. In *Proceedings of the 10th European conference on Research in Computer Security*, ESORICS'05, pages 489–508, Berlin, Heidelberg, 2005. Springer-Verlag.

[6] D. Fett, R. Kusters, and G. Schmitz. An expressive model for the web infrastructure: Definition and application to the browser id sso system. In *Security and Privacy (SP), 2014 IEEE Symposium on*, pages 673–688, May 2014.

[7] Alessandro Armando, Roberto Carbone, Luca Compagna, Jorge Cuellar, and Llanos Tobarra. Formal analysis of saml 2.0 web browser single sign-on: breaking the saml-based single sign-on for google apps. In *Proceedings of the 6th ACM workshop on Formal methods in security engineering*, FMSE '08, pages 1–10, New York, NY, USA, 2008. ACM.

[8] Aaron Bohannon and Benjamin C. Pierce. Featherweight firefox. In *2010 USENIX Conference on Web Application Development*, pages 123–134. The USENIX Association, The USENIX Association, June 2011.

[9] Joshua Drake, Paul Mehta, Charlie Miller Shawn Moyer, Ryan Smith, and Chris Valasek. Browser security comparison, December 2011.

[10] National Insitute of Standards and Technology. National vulnerability database version 2.2, 1 2012.

[11] W3C. Html 5.1 nightly, July 2013.

[12] Network Working Group. Hypertext transfer protocol – http/1.1, June 1999.

[13] W3C. Cascading style sheets (css) snapshot 2010, May 2011.

[14] Ecma International. Ecmascript language specification, June 2011.

[15] Network Working Group. Uniform resource identifier (uri): Generic syntax, January 2005.

[16] Mario Heiderich, Tilman Frosch, and Thorsten Holz. Iceshield: Detection and mitigation of malicious websites with a frozen dom. In *Recent Advances in Intrusion Detection*, pages 281–300. Springer, 2011.

[17] Willem De Groef, Dominique Devriese, Nick Nikiforakis, and Frank Piessens. Flowfox: a web browser with flexible and precise information flow control. In *Proceedings of the 2012 ACM conference on Computer and communications security*, number 19, pages 748–759, October 2012.

[18] Charles R. A. Hoare. *Communicating Sequential Processes.* Prentice Hall International, 4 edition, 1985.

[19] Jonathan C Herzog, Joshua D Guttman, and F. Javier Thayer. Strand spaces: proving security protocols correct. *Journal of Computer Security*, 7(2-3):191–230, March 1999.

[20] Christoph Weidenbach. Towards an automatic analysis of security protocols in first-order logic. In *Proceedings of the 16th International Conference on Automated Deduction: Automated Deduction*, CADE-16, pages 314–328, London, UK, UK, 1999. Springer-Verlag.

[21] L.C. Paulson. Proving properties of security protocols by induction. In *Proceedings 10th Computer Security Foundations Workshop.* Institute of Electrical and Electronics Engineers (IEEE).

[22] Thomas Groß, Birgit Pfitzmann, and Ahmad-Reza Sadeghi. Proving a ws-federation passive requestor profile with a browser model. In *Proceedings of the 2005 Workshop on Secure Web Services*, SWS '05, pages 54–64, New York, NY, USA, 2005. ACM.

[23] Birgit Pfitzmann and Michael Waidner. A model for asynchronous reactive systems and its application to secure message transmission. In *Proceedings of the 2001 IEEE Symposium on Security and Privacy*, number May in SP '01, pages 184–200, Washington, DC, USA, 2001. IEEE Computer Society.

[24] Thomas Groß. Security analysis of the saml single sign-on browser/artifact profile. In *Proceedings of the 19th Annual Computer Security Applications Conference*, ACSAC '03, pages 298–307, Washington, DC, USA, 2003. IEEE Computer Society.

[25] Sebastian Gajek, Mark Manulis, Ahmad-Reza Sadeghi, and Jörg Schwenk. Provably secure browser-based user-aware mutual authentication over tls. In *Proceedings of the 2008 ACM symposium on Information, computer and communications security*, ASIACCS '08, pages 300–311, New York, NY, USA, 2008. ACM.

[26] Andrew D Gordon and Riccardo Pucella. Validating a web service security abstraction by typing. *Formal Aspects of Computing*, 17(3):277–318, 2005.

[27] Michael McIntosh and Paula Austel. Xml signature element wrapping attacks and countermeasures. In *Proceedings of the 2005 workshop on Secure web services*, SWS '05, pages 20–27, New York, NY, USA, 2005. ACM.

[28] Apurva Kumar. A lightweight formal approach for analyzing security of web protocols. In Angelos Stavrou, Herbert Bos, and Georgios Portokalidis, editors, *Research in Attacks, Intrusions and Defenses*, volume 8688 of *Lecture Notes in Computer Science*, pages 192–211. Springer International Publishing, 2014.

[29] Apurva Kumar. *A Belief Logic for Analyzing Security of Web Protocols*, pages 239–254. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[30] Colin Boyd and Wenbo Mao. On a limitation of ban logic. In *Workshop on the Theory and Application of Cryptographic Techniques on Advances in Cryptology*, EUROCRYPT '93, pages 240–247, Secaucus, NJ, USA, 1994. Springer-Verlag New York, Inc.

[31] Gavin Lowe. A hierarchy of authentication specifications. pages 31–43. IEEE Computer Society Press, 1997.

[32] David Basin, Saa Radomirovic, and Lara Schmid. Modeling human errors in security protocols. In *2016 IEEE 29th Computer Security Foundations Symposium (CSF)*. Institute of Electrical and Electronics Engineers (IEEE), jun 2016.

[33] Laxman Muthiyah. How i hacked your facebook photos - deleting any photo albums. Blog, 2 2015.

[34] Mario Heiderich, Tilman Frosch, Meiko Jensen, and Thorsten Holz. Crouching tiger - hidden payload: Security risks of scalable vectors graphics. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, pages 239–250, New York, NY, USA, 2011. ACM.