Towards a Formal Foundation of Web Security

Devdatta Akhawe*, Adam Barth*, Peifung E. Lam†, John Mitchell† and Dawn Song*
*University of California, Berkeley
{devdatta,abarth,dawnsong}@cs.berkeley.edu
†Stanford University
{pflam,mitchell}@cs.stanford.edu

***Abstract*—We propose a formal model of web security based on an abstraction of the web platform and use this model to analyze the security of several sample web mechanisms and applications. We identify three distinct threat models that can be used to analyze web applications, ranging from a *web attacker* who controls malicious web sites and clients, to stronger attackers who can control the network and/or leverage sites designed to display user-supplied content. We propose two broadly applicable security goals and study five security mechanisms. In our case studies, which include HTML5 forms, Referer validation, and a single sign-on solution, we use a SAT-based model-checking tool to find two previously known vulnerabilities and three new vulnerabilities. Our case study of a Kerberos-based single sign-on system illustrates the differences between a secure network protocol using custom client software and a similar but vulnerable web protocol that uses cookies, redirects, and embedded links instead.**

I. Introduction

The web, indispensable in modern commerce, entertainment, and social interaction, is a complex delivery platform for sophisticated distributed applications with multifaceted security requirements. However, most web browsers, servers, network protocols, browser extensions, and their security mechanisms were designed without analytical foundations. Further complicating matters, the web continues to evolve with new browser features, protocols, and standards added at a rapid pace [1]–[6]. The specifications of new features are often complex, lack clear threat models, and involve unstated and unverified assumptions about other components of the web. As a result, new features can introduce new vulnerabilities and break security invariants assumed by web applications [7]–[9]. Just as formal models and tools have proven useful in evaluating the security of network protocols, we believe that abstract yet informed models of the web platform, web applications, and web security mechanisms will be amenable to automation, reveal practical attacks, and support useful evaluation of alternate designs.

In this paper, we take a first step towards building a comprehensive formal foundation for web security. In particular, we propose a formal model for the web platform, which includes a number of key web concepts, and demonstrate that our model is useful for finding bugs in real-world web security mechanisms. Our model is sufficiently abstract to be amenable to formal analysis and yet appears sufficiently detailed to express subtle attacks missed by expert human analysts in several cases. We provide an executable implementation of a subset of our model in Alloy [10] and demonstrate the utility of this subset (and, more generally, our model) via five case studies. Although we imagine our model being used for more than vulnerability discovery, we focus in this paper on analyzing existing protocols for design errors. We show that our model can capture two previously known and three previously unknown vulnerabilities.

The web security model consists of a selection of web concepts, precise threat models, and two broadly applicable security goals. These design choices are informed by previous experience designing and (informally) evaluating web security mechanisms, such as preventing cross-site request forgery [11], securing browser frame communication [12], preventing DNS rebinding [13], and protecting high-security web sites from network attacks [14]. These and other previous studies suggest that a few central modeling concepts will prove useful for evaluating a wide range of mechanisms.

The central web concepts we formalize in our model include browsers, servers, scripts, HTTP, and DNS, as well as ways they interact. For example, each script context, representing execution of JavaScript within a browser execution environment, is associated with a given "origin" and located in a browser. By making use of browser APIs, such as XMLHttpRequest, these script contexts can direct (restricted forms of) HTTP requests to various DNS names, which resolve to servers. These servers, in turn, respond to these requests and influence the browser's behavior. Although the web security model we describe in section II also contains other concepts such as frames, the location bar, and the lock icon, our executable implementation described in section III focuses on browsers, servers, scripts, HTTP, and DNS, which form the "backbone" of the model.

We propose three distinct and important threat models: a *web attacker*, an *active network attacker*, and a *gadget attacker*. The most important threat model, at least for mechanisms and studies we are familiar with, is the web attacker. The web attacker operates a malicious web site and may use a browser, but has no visibility into the network beyond requests or responses directed towards the hosts it operates. Many core web security mechanisms are designed to resist the threats we formalize as the web attacker but fail to provide protection against more powerful attackers. An active network attacker has all the abilities of a web attacker plus the ability to eavesdrop, block, and forge network messages. The active network attacker we define is slightly more powerful than the eponymous threat considered when analyzing a traditional network protocol because our active network attacker can make use of browser APIs. Finally, we also consider a web attacker with the ability to inject (limited kinds of) content into otherwise honest web sites, corresponding to the gadget attacker considered in [12]. This threat lets us consider the robustness of web security mechanisms in the presence of third-party content ranging from comments on a blog to gadgets in a mashup.

The third part of our model formulates two widely applicable security goals that can be evaluated for various mechanisms: (i) new mechanisms should not violate any of the invariants that web sites commonly rely upon for security and (ii) a "session integrity" condition, which states that the attacker is unable to cause honest servers to undertake potentially sensitive actions. There is a wide spectrum of security goals we could investigate, but we focus on these goals because they are generally applicable to many web security mechanism, including those in our case studies.

Concretely, we implement the core components of our model in Alloy [10], [15], an automated tool that translates a declarative object-modeling syntax into propositional input to a SAT solver. Using Alloy's declarative input language, we axiomatize the key concepts, threat models, and security goals of our model. Our axiomatization is incomplete (both because we do not implement all the concepts in our model and because browser implementations might contain bugs) but useful nonetheless. After modeling a specific web security mechanism, the Alloy satisfiability (SAT) solver attempts to find browser and site interactions that violate specified security goals. Typically, we ask not whether a particular web security mechanism is secure, but how powerful an attacker is required to defeat the security mechanism. In this way, we aim to quantify the security of the mechanism.

To demonstrate the utility of our model, we conduct five case studies. We use our model to analyze a proposed cross-site request forgery defense based on the Origin header, Cross-Origin Resource Sharing [3] (the security component of the new cross-origin XMLHttpRequest API in the latest browsers), a proposal [16] to use Referer validation to prevent cross-site scripting, new functionality in the HTML5 form element, and WebAuth, a Kerberos-based single sign-on system used at a number of universities. In each case, our model finds a vulnerability in the mechanism, two of which were previously known and three of which were previously unknown. The Referer validation example, in particular, demonstrates that our model is more sophisticated than previous approaches because [16] analyzed the mechanism with Alloy and concluded that the mechanism was secure. The WebAuth example shows that subtle security issues arise when embedding a well-understood network protocol (Kerberos) in a web security mechanism because of interactions between the assumptions made by the protocol and the behavior of the web platform.

Related Work: There is a large body of work on formally verifying security properties of network protocols, including model checking using a variety of tools [17]–[20], constraint-based methods [21], and formal and automated proof methods [22]–[25]. Some of our previous non-formalized work on web security (e.g., [26]) hints at formal analysis by showing the existence of a frame communication bug that is apparent as soon as the protocol is written down formally. There has also been some work on formal verification of web service security [27], [28]. A number of the notions we formalize in this paper have been used informally in the past. For example, MashupOS [29] contemplates web attacker-like threats and the gadget attacker makes an explicit appearance in [30]. The designers of the OP browser [31] use formal methods to verify some security properties of their design (whether or not the security indicators behave as expected). Formal methods have also been used to verify code-level properties of the status bar in Internet Explorer [32]. The most closely related work [16] uses Alloy to verify the security properties of a particular cross-site scripting defense (which we analyze further in this paper). However, none of these works attempt to formulate a general model of web security applicable beyond a single mechanism.

Organization: The remainder of this paper is organized as follows. Section II presents our formal model. Section III explains how we implement our model in Alloy. Section IV analyzes five example web security mechanisms using our model. Section V contains some operational statistics and advice about the model. Section VI concludes.

II. General Model

There are many threats associated with web browsing and web applications, including phishing, drive-by downloads, blog spam, account takeover, and click fraud. Although some of these threats revolve around exploiting implementation vulnerabilities (such as memory safety errors in browsers or tricking the user), we focus, in this paper, on ways in which an attacker can abuse web functionality that exists by design. For example, an HTML form element lets a malicious web site generate GET and POST requests to arbitrary web sites, leading to security risks like cross-site request forgery (CSRF). Web sites use a number of different strategies to defend themselves against CSRF [11], but we lack a scientifically rigorous methodology for studying these defenses. By formulating an accurate model of the web, we can evaluate the security of these defenses and determine how they interact with extensions to the web platform.

A core idea in our model is to describe what could occur if a user navigates the web and visits sites in the ways that the web is designed to be used. For example, the user could choose to type any web address into the address bar and visit any site, or click on a link provided by one site to visit another. Because browsers support the "back" button, returning the user to a previously visited page, many sites in effect allow a user to click on all of the links presented on a page, not just one. When the user visits a site, the site could serve a page with any number of characteristics, possibly setting a cookie, or redirecting the user to another site. The set of events that could occur, therefore, includes browser requests, responses, cookies, redirects, and so on, transmitted over HTTP or HTTPS.

We believe that examining the set of possible events accurately captures the way that web security mechanisms are designed. For example, the web is designed to allow a user to visit a good site in one window and a potentially malicious site in another. Because the back button is so popular, web security mechanisms are usually designed to be secure even if the user returns to a previously visited page and progresses differently the second (or third or fourth) time.

The model we propose has three main parts: web concepts, threat models, and security goals. The web concepts represent the conceptual universe defined by web standards. Our formalization of these concepts includes a set of browsers, operated by potential victims, each with its user and a browsing history, interacting with an arbitrary number of web servers that receive and send HTTP requests and responses, possibly encrypted using SSL/TLS. Our model considers a spectrum of threats, ranging from a web attacker to a network attacker to a gadget attacker. For example, a web attacker controls one or more web sites that the user visits, and may operate a browser to visit sites, but does not control the network used to visit other sites. Finally, we regard security goals as predicates that distinguish felicitous outcomes from attacks.

A. *Web Concepts*

The central concepts of the web are common to virtually every web security mechanism we wish to analyze. For example, web mechanisms involve a web browser that interacts with one or more web servers via a series of HTTP requests and responses. The browser, server, and network behavior form the "backbone" of the model, much in the same way that cryptographic primitives provide the backbone of network protocols. Many of the surprising behaviors of the web platform, which lead to attacks against security mechanisms, arise from the complex interaction between these concepts. By modeling these concepts precisely, we can check their interactions automatically.

Non-Linear Time: We use a branching notion of time because of the browser's "back" button. In other words, we are not concerned with the actual temporal order between unrelated actions. Instead, if a user could click on either of two links, then use the back button to click on the other, we represent this as two actions that are unordered in time. In effect, our temporal order represent necessary "happens before" relations between events (e.g., an HTTP request must happen before the browser can receive the corresponding HTTP response). Instead of regarding these branches as possible futures, we regard them as all having occurred, for example letting an attacker transport knowledge from one branch to another. In addition to conceptual economy, abstracting from the accidental linear order can reduce the number of possible states in need of exploration.

This notion of time leads to a model that is largely *monotonic*. If an attacker can undertake an action at one point in time, we assume that action is thereafter always available to the attacker (because the user can usually return to that state via the back button). In contrast, traditional models of network protocol security are non-monotonic: once the protocol state machine advances to step 3, the attacker can no longer cause the state machine to accept a message expected in step 2. Although we do not exploit this monotonicity directly in this paper, we believe this property bears further investigation.

Browser: The user's web browser, of course, plays a central role in our model of web security. However, the key question is what level of abstraction to use for the browser. If we model the browser at too low a level (say bytes received over the network being parsed by an HTML parser and rendered via the CSS box model into pixels on the screen), our model will quickly become unwieldy. The HTML5 specification alone is some $45{,}000$ lines of text. Instead, we abstract the browser into three key pieces:

- *Script Context.* A script context represents all the scripts running in the browser on behalf of a single web origin. The browser does not provide any isolation guarantees between content within an origin: all same-origin scripts "share fate." Correspondingly, we group the various scripts running in different web pages within an origin into a single script context and imagine them acting in unison.
- *Security UI.* Some parts of the browser's user interface have security properties. For example, the browser guarantees that the location bar accurately displays the URL of the top-level frame. We include these elements (notably, the location bar, the lock icon, and the extended validation indicator) in our model and imbue them with their security properties. In addition, we model a forest of frames, in which each frame is associated with a script context and each tree of frames is associated with a constellation of security indicators. We assume that each frame can overwrite or display (but not read) the pixels drawn by the frame below it in the hierarchy, modeling (at a high level) how web

pages are drawn.

- *State Storage.* Finally, the browser contains some amount of persistent storage, such as a cookie store and a password database. We assume that confidential information contained in these state stores is associated with an origin and can be read by a script context running on behalf of that origin. To keep the model monotonic, we model these state stores as "append-only," which is not entirely accurate but simplifies the model considerably.

Servers: We model web servers as existing at network locations (which are an abstraction of IP addresses). Each web server is owned by a single principal, who controls how the server responds to network messages. Servers controlled by "honest" principals follow the specification but a server controlled by a malicious principal might not. Servers have a many-to-many relation to DNS names (e.g., `www.example.com`), which themselves existing in a delegation hierarchy (e.g., `www` delegates to `example`, which delegates to `com`). Holding servers in a many-to-many relation with DNS names is essential for modeling various tricky situations, such as DNS rebinding [13], where the attacker points a malicious DNS name at an honest server.

Network: Finally, browsers and servers communicate by way of a network. In contrast to traditional models of network security, our model of the network has significant internal structure. Browsers issue HTTP requests to URLs, which are mapped to servers via DNS. The requests contain a method (e.g., GET, POST, DELETE, or PUT) and a set of HTTP headers. Individual headers carry semantics. For example, the Cookie header contains information retrieved from the browser's cookie store and the Referer header identifies the script context that initiated the request. It is an important part of security mechanisms such as CSRF defenses [11] that the Referer header, for example, is set by the browser and not controlled by content rendered in the browser.

Network requests can be generated by a number of web APIs, including HTML forms, XMLHttpRequest, and HTTP redirects, each of which imposes different security constraints on the network messages. For example, requests generated by XMLHttpRequest can be sent only to the same origin (in the absence of CORS [3]), whereas requests generated by HTML forms can be sent to any origin but can contain only certain methods and headers. These restrictions are essential for understanding the security of the web platform. For example, the Google Web Toolkit relies on the restrictions on custom HTTP headers imposed by the HTML form element to protect against CSRF [33].

B. Threat Models

When evaluating the security of web applications, we are concerned with a spectrum of threats. The weakest threat is that of a *web attacker*: a malicious principal who operates a web site visited by the user. Starting with the web attacker as a base, we can consider more advanced threats, such as an *active network attacker* and a *gadget attacker*.

Web attacker: Although the informal notion of a web attacker has appeared in our previous work [11], [12], [14], [34], [35], we articulate the web attacker's abilities precisely.

- *Web Server.* The web attacker controls at least one web server and can respond to HTTP requests with arbitrary content. Intuitively, we imagine the web attacker as having "root access" to these web servers. The web attacker controls some number of DNS names, which the attacker can point to any server. Canonically, we imagine the DNS name `attacker.com` referring to the attacker's main web server. The web attacker can obtain an HTTPS certificate for domains owned by the attacker from certificate authorities trusted by the user's browser. Using these certificates, the attacker can host malicious content at URLs like `https://attacker.com/`.
- *Network.* The web attacker has no special network privileges. The web attacker can respond only to HTTP requests directed at his or her own servers. However, the attacker can *send* HTTP requests to honest servers from attacker-controlled network endpoints. These HTTP requests need not comply with the HTTP specification, nor must the attacker process the responses in the usual way (although the attacker can simulate a browser locally if desired). For example, attacker can send an arbitrary value in the Referer header and need not follow HTTP redirects. Notice that the web attacker's abilities are decidedly *weaker* than the usual network attacker considered in studies of network security because the web attacker can neither eavesdrop on messages to other recipients nor forge messages from other senders.
- *Browser.* When the user visits the attacker's web site, the attacker is "introduced" to the user's browser. Once introduced, the attacker has access to the browser's web APIs. For example, the attacker can create new browser windows by calling the `window.open()` API. We assume the attacker's use of these APIs is constrained by the browser's security policy (colloquially known as the "same-origin policy" [36]), that is, the attacker uses only the privileges afforded to every web site. One of the most useful browser APIs, from the attacker's point of view, is the ability to generate cross-origin HTTPS requests via hyperlinks or the HTML form element. Attacks often use these APIs in preference to directly sending HTTP requests because (1) the requests contain the user's cookies and (2) the responses are interpreted by the user's browser.

A subtle consequence of these assumptions is that (once introduced) the attacker can maintain a persistent thread of control in the user's browser. This thread of control, easily achieved in practice using a widely known web application programming technique [37], can communicate freely with (and receive instructions from) the attacker's servers. We do not model this thread of control directly. Instead, we abstract these details by imagining a single coherent attacker operating at servers and able to generate specific kinds of events in the user's browser (accurately associated with the attacker's origin).

Network Attacker: An active network attacker has all the abilities of a web attacker as well as the ability to read, control, and block the contents of all *unencrypted* network traffic. In particular, the active network attacker need not be present at a network endpoint to send or receive messages at that endpoint. We assume the attacker cannot corrupt HTTPS traffic between honest principals because trusted certificate authorities are unwilling to issue the attacker certificates for honest DNS names, although these certificate authorities are willing to issue the attacker HTTPS certificates for malicious DNS names and the attacker can, of course, always self-sign a certificate for an honest DNS name. Without the appropriate certificates, we assume the attacker cannot read or modify the contents of HTTPS requests or responses.

Gadget Attacker: The gadget attacker [26] has all the abilities of a web attacker as well as the ability to inject some limited kinds of content into honest web sites. The exact kind of content the gadget attacker can inject depends on the web application. In many web applications, the attacker can inject a hyperlink (e.g., in email or in blog comments). In some applications, such as forums, the attacker can inject images. In more advanced applications, such as Facebook or iGoogle, the attacker can inject full-blown gadgets with extensive opportunity for misdeeds. We include the gadget attacker to analyze the robustness of security mechanisms to web sites hosting (sanitized) third-party content.

User Behavior: The most delicate part of our threat model is how to constrain user behavior. If we do not constrain user behavior at all, the user could simply send his or her password to the attacker, defeating most web security mechanisms. On the other hand, if we constrain the user too much, we risk missing practical attacks.

- *Introduction.* We assume the user might visit any web site, including the attacker's web site. We make this assumption because we believe that an honest user's interaction with an honest site should be secure even if the user separately visits a malicious site in a different browser window. A concerted attacker can always acquire traffic by placing advertisements. For example, in a previous study [13], we mounted a web attack by purchasing over 50,000 impressions for $30. In other words, we believe that this threat model is an accurate abstraction of normal web behavior, *not* an assumption that web users promiscuously visit all possible bad sites in order to tempt fate.
- *Not Confused.* Even though the user visits the attacker's web site, we assume the user does not confuse the attacker's web site with an honest web site. In particular, we assume the user correctly interprets the browser's security indicators, such as the location bar, and enters confidential information into a browser window only if the location bar displays the URL of the intended site. This assumption rules out *phishing* attacks [38], [39], in which the attacker attempts to fool the user by choosing a confusing domain name (e.g., `bankofthevvest.com`) or using other social engineering. In particular, we do *not* assume a user treats `attacker.com` as if it were another site. However, these assumptions could be relaxed or varied in order to study the effectiveness of specific mechanisms when users are presented with deceptive content.

C. *Security Goals*

Although different web security mechanisms have different security goals, there are two security goals that seem to be fairly common:

- *Security Invariants.* The web contains a large number of existing web applications that make assumptions about web security. For example, some applications assume that a user's browser will never generate a cross-origin HTTP request with DELETE method because that property is ensured by today's browsers (even though cross-origin GET and POST requests are possible). When analyzing the security of new elements of the web platform, it is essential to check that these elements respect these (implicit) security invariants and "don't break the web" (i.e., introduce vulnerabilities into existing applications). We formalize this goal as a set of invariants servers expect to remain true of the web platform. Although we focus at present on invariants relevant to the mechanisms at hand, we believe that future work on web security can fruitfully aim to identify more invariants.
- *Session Integrity.* When a server takes action based on receiving an HTTP request (e.g., transfers money from one bank account to another), the server often wishes to ensure that the request was generated by a trusted principal and not an attacker. For example, a traditional cross-site request forgery vulnerability results from failing to meet this goal. We formalize this goal by recording the "cause" of each HTTP request (be it an API invoked by a script or an HTTP redirect) and checking whether the attacker is in this casual chain.

We are unaware of previous work that recognizes the value of identifying clear web security invariants. However, because many web security mechanisms depend on complementary properties of the browser, web protocols, and user behavior, we believe that these invariants form the core of a comprehensive scientific understanding of web security.

III. Implementation in Alloy

We implement a subset of our formal model in the Alloy modeling language. Although incomplete, our implementation [40] contains the bulk of the networking and scripting concepts and is sufficiently powerful to find new and previously known vulnerabilities in our case studies. In this section, we summarize how we implement the key concepts from the model in this language. We first express the base model, containing the web concepts and threats, and then add details of the proposed web mechanism. Finally, we add a constraint that negates the security goal of the mechanism and ask Alloy for a satisfying instance. If Alloy can produce such an instance, that instance represents an attack because the security goal has failed.

Expressing our model in Alloy has several benefits. First, expressing our model in an executable form ensures that our model has precise, testable semantics. In creating the model, we found a number of errors by running simple "sanity checks." Second, Alloy lets us express a model of unbounded size and then later specify a size bound when checking properties. We plan to use this distinction in future work to prove a "small model" theorem bounding the necessary search size (similar to [41]). Finally, Alloy translates our high-level, declarative, relational expression of the model into a SAT instance that can be solved by state-of-the-art SAT solvers (e.g. [42]), letting us leverage recent advances in the SAT solving community.

A. An Introduction to Alloy

Alloy [10], [15], [43] is a declarative language based on first order relational logic. All data types are represented as relations and are defined by their *type signatures*; each type signature plays the role of a type or subtype in the type system. A type signature declaration consists of the type name, the declarations of *fields*, and an optional *signature fact* constraining elements of the signature. A *subsignature* is a type signature that extends another, and is represented as a subset of the base signature. The immediate subsignatures of a signature are disjoint. A *top-level* signature is a signature that does not extend any other signature. An *abstract signature*, marked *abstract*, represents a classification of elements that is intended to be refined further by more "concrete" subsignatures.

In Alloy, a *fact* is a constraint that must always hold. A *function* is a named expression with declaration parameters and a declaration expression as a result. A *predicate* is a named logical formula with declaration parameters. An *assertion* is a constraint that is intended to follow from the facts of a model.

The *union* $(+)$, *difference* $(-)$ and *intersection* $(\&)$ operators are the standard set operators. The *join* $(.)$ of two relations is the relation obtained by taking concatenations of a tuple from the first relation and another tuple from the second relation, with the constraint that the last element of the first tuple matches the first element of the second tuple, and omitting the matching elements. For example, the join of $\{(a,b),(b,d)\}$ and $\{(b,c),(a,d),(d,a)\}$ is $\{(a,c),(b,a)\}$. The *transitive closure* $(\hat{})$ of a relation is the smallest enclosing relation that is transitive. The *reflexive-transitive closure* $(*)$ of a relation is the smallest enclosing relation that is both transitive and reflexive.

Alloy Analyzer is a software tool that can be used to analyze models written in Alloy. The Alloy code is first translated into a satisfiability problem. SAT solvers are then invoked to exhaustively search for satisfying models or counterexamples to assertions within a bounded *scope*. The scope is determined jointly by the user and the Alloy Analyzer. More specifically, the user can specify a numeric bound for each type signature, and any type signature not bounded by the user is given a bound computed by the Alloy Analyzer. The bounds limit the number of elements in each set represented by a type signature, hence making the search finite.

B. Realization of Web Concepts

Many of the concepts in our general model have direct realizations in our implementation. For example, we define types representing `Principals`, `NetworkEndpoints`, and `NetworkEvents`. A `NetworkEvent` represents a type of network message that has a sender (i.e., it is `from a NetworkEndpoint`) and a recipient (i.e., it is `to a NetworkEndpoint`). It is a subsignature of `Event`, hence it also inherits all fields of `Event`. A `NetworkEvent` can be either an `HTTPRequest` or an `HTTPResponse`, which include HTTP-specific information such as a `Method` and a set of `HTTPRequestHeaders` or `HTTPResponseHeaders`, respectively:

```
abstract sig NetworkEvent extends Event {
  from: NetworkEndpoint,
  to: NetworkEndpoint
}
abstract sig HTTPEvent extends NetworkEvent {
  host: Origin
}
sig HTTPRequest extends HTTPEvent {
  method: Method,
  path: Path,
  headers: set HTTPRequestHeader
}
sig HTTPResponse extends HTTPEvent {
  statusCode: Status,
  headers: set HTTPResponseHeader
}
```

Figure 1 depicts some of the types used in our expression together with the relations between these types. For example, `HTTPRequest` is a subtype of `HTTPEvent`, and contains `path` and `headers` as some of its fields. This metamodel provides a conceptual map of our model. In the remainder of this section, we highlight parts of the model that lend intuition into its construction.

Principals: A `Principal` is an entity that controls a set of `NetworkEndpoints` and owns a set of `DNSLabels`, which represent fully qualified host names:

```
abstract sig Principal {
  servers: set NetworkEndpoint,
  dnslabels: set DNS
}
```

The model contains a hierarchy of subtypes of `Principal`. Each level of the hierarchy imposes more constraints on how the principal can interact with the other objects in the model by adding declarative invariants. For example, the servers owned by principals who obey the network geometry (i.e., every kind of principal other than active network attackers) must conform to the routing rules of HTTP:

```
abstract sig PassivePrincipal
  extends Principal{} {
    servers in HTTPConformist
}
```

Browsers: A `Browser` is an `HTTPClient` together with a set of trusted `CertificateAuthorites` and a set of `ScriptContexts`. (For technical convenience, we store the `Browser` as a property of a `ScriptContext`, but the effect is the same.)

```
abstract sig Browser
  extends HTTPClient {
  trustedCA: set CertificateAuthority
}
sig ScriptContext {
  owner: Origin,
  location: Browser,
  transactions: set HTTPTransaction
}
```

The browser uses its set of `trustedCAs` to validate HTTPS certificates used by `NetworkEndpoints` in responding to HTTPS requests.

In addition to being located in a particular `Browser`, a `ScriptContext` also has an `Origin` and a set of `HTTPTransactions`. The `Origin` is used by various browser APIs to implement the so-called "same-origin" policy. For example, the `XMLHTTPRequest` object (a subtype of `RequestAPI`) prevents the `ScriptContext` from initiating `HTTPRequests` to a foreign origin. The `transactions` property of `ScriptContext` is the set of `HTTPTransactions` (`HTTPRequest`, `HTTPResponse` pairs) generated by the `ScriptContext`.

C. *Facts and Assertions*

We find it convenient to model the browser's cookie store using *fact* statements about cookies, rather than declare a new *type signature* for it. We require that `HTTPRequests` from `Browsers` contain only appropriate cookies from previous `SetCookieHeaders`. Selecting the appropriate cookies uses a rule that has a number of cases reflecting the complexity of cookie policies in practice, part of which is shown below. More explicitly, a browser attaches a cookie to an `HTTPRequest` only if the cookie was set in a previous `HTTPResponse` and the servers of the `HTTPRequest` and `HTTPResponse` have the same DNS label.

```
fact {
  all areq:HTTPRequest | {
    areq.from in Browser
    hasCookie[areq]
  } implies all acookie: reqCookies[areq]|
    some aresp: getBrowserTrans[areq].resp | {
      aresp.host.dnslabel = areq.host.dnslabel
      acookie in respCookies[aresp]
      happensBeforeOrdering[aresp,areq]
  }
}
```

Causality: Every `HTTPTransaction` has a `cause`, which is either another `HTTPTransaction` (e.g., due to a redirect) or a `RequestAPI`, such as `XMLHttpRequest` or `FormElement`. Each `RequestAPI` imposes constraints on the kinds of `HTTPRequests` the API can generate. For example, this constraint limits the `FormElement` to producing GET and POST requests:

```
fact {
  all t:ScriptContext.transactions |
    t.cause in FormElement implies
      t.req.method in GET + POST
}
```

Session Integrity: Using the `cause` relation, we can construct the set of principals involved in generating a given `HTTPTransaction`. The predicate below is especially useful in checking assertions of session integrity properties because we can ask whether there exists an instantiation of our model in which the attacker caused a network request that induced an honest server to undertake some specific action.

```
fun involvedServers[
    t:HTTPTransaction
    ]:set NetworkEndpoint{
 (t.*cause & HTTPTransaction).resp.from
 + getTransactionOwner[t].servers
}
```

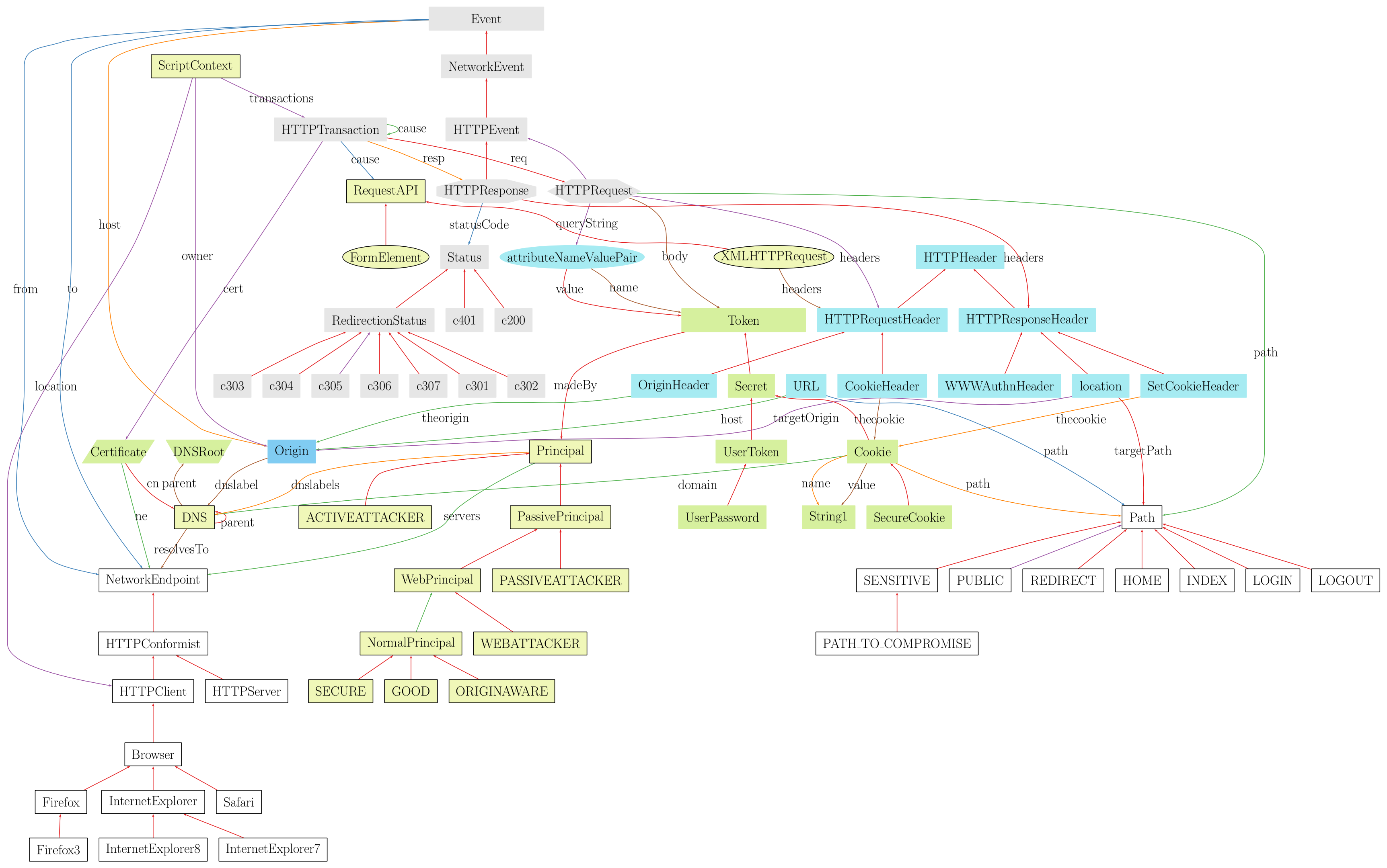

Figure 1. The metamodel of our formalization of web security. Red unmarked edges represent the ‘extends’ relationship.

```
pred webAttackerInCausalChain[
  t:HTTPTransaction]{
    some (WEBATTACKER.servers
    & involvedServers[t])
}
```

IV. Case Studies

In this section, we present a series of case studies of using our model to analyze web security mechanisms. We study five web security mechanisms. For the first two, the Origin header and Cross-Origin Resource Sharing, we show that our model is sufficiently expressive to rediscover known vulnerabilities in the mechanism. For the other three, Referer Validation, HTML5 forms, and WebAuth, we use our model to discover previously unknown vulnerabilities.

A. Origin Header

In a previous paper [11] that did not involve formal analysis, we proposed that browsers identify the origin of HTTP requests by including an Origin header and that web sites use that header to defend themselves against Cross-Site Request Forgery (CSRF).

Modeling: To model the Origin header, we added an `OriginHeader` subtype of `HTTPRequestHeader` to the base model and required that browsers identify the origin in the header:

```
fun getOrigin[r:HTTPRequest] {
  (r.headers & OriginHeader).theorigin
}
fact BrowsersSendOrigin{
  all t:HTTPTransaction,sc:ScriptContext | {
    t in sc.transactions
  } implies {
    getOrigin[t.req] = sc.owner
  }
}
```

To model the CSRF defense, we added a new type of honest web server that follows the recommendations in the paper (namely rejects "unsafe" methods that have an untrusted Origin header):

```
pred checkTrust[r:HTTPRequest,p:Principal]{
  getOrigin[r].dnslabel in p.dnslabels
}
fact {
  all aResp: HTTPResponse | {
    aResp.from in ORIGINAWARE.servers
    and aResp.statusCode = c200
  } implies {
    let theTrans = getTransaction[aResp] |
      theTrans.req.method in safeMethods or
      checkTrust[theTrans.req,ORIGINAWARE]
  }
}
```

Vulnerability: We then checked whether this mechanism satisfies session integrity. Alloy produces a counter example: if the honest server sends a POST request to the attacker's server, the attacker can redirect the request back to the honest server. Because the Origin header comes from the original `ScriptContext`, the honest server will accept the redirected request, violating session integrity. Although this vulnerability was known previously, the bug eluded both the authors and the reviewers of the original paper.

Solution: One potential solution is to update the Origin header after each redirect, but this approach fails to protect web sites that contain open redirectors (which are remarkably common). Instead, we recommend naming all the origins involved in the redirect chain in the Origin header. The current Internet-Draft describing the Origin header [44] includes this fix. We have verified that the fixed mechanism enjoys session integrity in our model (for the finite sizes used in our analysis runs).

B. Cross-Origin Resource Sharing

Cross-Origin Resource Sharing (CORS) lets web sites opt-out of some of the browser's security protections. In particular, by returning various HTTP headers, the site can instruct the browser to share the contents of an HTTP response with specific origins, or to let specific origins issue otherwise forbidden requests. CORS is somewhat complex because it distinguishes between two kinds of requests: simple requests and complex requests. Simple requests are supposedly safe to send cross-origin, whereas complex requests require a *pre-flight* request that asks the server for permission before sending the potentially dangerous request. CORS is a good case study for our model for two reasons: (1) maintaining the web security invariants is a key requirement in the design, driving the distinction between simple and complex requests; (2) complex requests can disrupt session integrity if the pre-flight request is not handled properly.

Modeling: We model the `PreflightRequest` as a subtype of `HTTPRequest` imbued with special semantics by the browser. Even though the implementations of CORS in browsers reuse the XMLHttpRequest JavaScript API, we model CORS using a new `RequestAPI`, which we call `XMLHttpRequest2`, making it easier to compare the new and old behavior. CORS involves a number of HTTP headers, which we model as subtypes of `CORSResponseHeader`. Finally, we model servers as `NetworkEndpoints` that never include any CORS headers in HTTP responses.

```
fact {
 all p:PreFlightRequest | {
  p.method = OPTIONS and
  some p.headers & AccessControlRequestMethod
  and some p.headers & OriginHeader and
  some p.headers & AccessControlRequestHeaders
}
```

```
fact {
  all t:HTTPTransaction,sc:ScriptContext |{
    t in sc.transactions and
    t.^cause in
      (XMLHTTPRequest2+HTTPTransaction)
  } implies {
    isPreFlightRequestTransaction[t]
    or isSimpleCORSTransaction[t]
    or isComplexCORSTransaction[t]
    or (not isCrossOriginRequest[t.req])
  }
}
```

Vulnerability: Alloy produced a (previously known) counter-example that breaks a key web security invariant because a legacy server might redirect the pre-flight request to the attacker's server. According to the current W3C Working Draft [3], browsers follow these redirects transparently, letting the attacker's server return a CORS header that opts the legacy server into receiving new kinds of requests (such as DELETE requests). This attack is fairly practical because many web sites contain open redirectors. Notice that we did not need to model open redirectors explicitly. Instead, a legacy server might redirect requests to arbitrary locations because the model does not forbid these responses.

Solution: A simple solution is to ignore redirects for pre-flight requests. The most recent Editor's Draft [45] (which is more up-to-date) has precisely this behavior. We verify the security of the updated protocol (up to a finite size) in our model by adding the following requirement:

```
fact {
  all first:HTTPTransaction | {
    first.req in PreFlightRequest and
    first.resp.statusCode in RedirectionStatus
  } implies no second:HTTPTransaction |
    second.cause = first
}
```

C. Referer Validation

A recent paper [16] proposes that web sites should defend against CSRF and Cross-Site Scripting (XSS) by validating the Referer header. The authors claim that these attacks occur "when a user requests a page from a target website s by following a link from another website s' with some input." To defend against these attacks, the web site should reject HTTP requests unless (1) the Referer header is from the site's origin or (2) the request is directed at a "gateway" page, that is carefully vetted for CSRF and XSS vulnerabilities; see Figure 2. What makes this security mechanism a particularly interesting case study is that the authors model-check their mechanism using Alloy. However, their model omits essential details like HTTP Redirects and cross-origin hyperlinks. As a result, it is unable to uncover a vulnerability in their mechanism.

Modeling: Because the Referer header is already part of the model, we only needed to add a new class of principal: `RefererProtected` that exhibits the required behavior. We then added a constraint that HTTP requests with external Referers are allowed only on the "`LOGIN`" page:

```
fact {
 all aReq:HTTPRequest | {
  (getTransaction[aReq].resp.from
      in RefererProtected.servers )
  and isCrossOrigin[aReq]
  } implies aReq.path = LOGIN
}
```

Vulnerability: Alloy produces a counterexample for the session integrity condition because the attacker can mount a CSRF attack against a `RefererProtected` server if the server sends a request to the attacker's server first (see the dashed lines in Figure 2). For example, if the attacker can inject a hyperlink into the honest site, the user might follow that hyperlink, generating an HTTP request to the attacker's server with the honest site's URL in the Referer header. The attacker can then redirect that request back to the honest server. This previously unknown vulnerability in Referer validation is remarkably similar to the vulnerability in the Origin header CSRF defense described above.

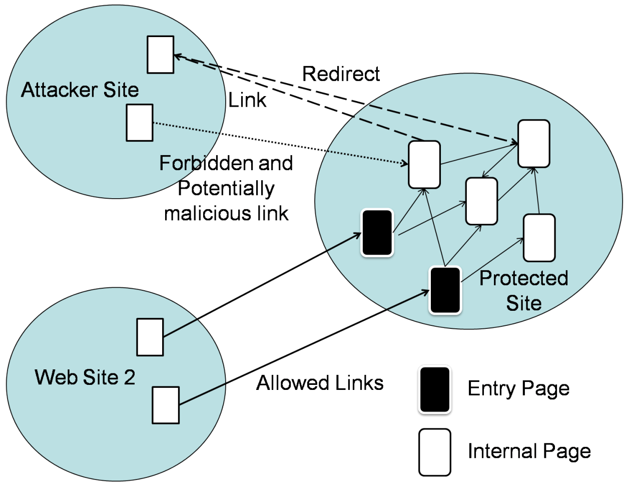

Figure 2. Vulnerability in Referer Validation. This figure is adapted from [16], with the attack (dashed line) added.

Solution: This vulnerability is difficult to correct on the current web because the Referer header is already widely deployed (and therefore, for all practical purposes, immutable). One possible solution is for the web site to suppress all outgoing Referer headers using, for example, the `noreferrer` relation attribute on hyperlinks.

D. HTML5 Forms

HTML5, the next iteration of the HyperText Markup Language, adds functionality to the `FormElement` API to generate HTTP requests with PUT and DELETE methods. To avoid introducing security vulnerabilities into existing web sites, the specification restricts these new methods to requests that are sent to a server in the same origin as the document containing the form.

Modeling: Modeling this extension to the web platform was trivial: we added PUT and DELETE to the whitelist of methods for `FormElement` and added a requirement that these requests be sent to the same origin:

```
t.req.method in PUT+DELETE implies
  not isCrossOriginRequest[t.req]
```

Vulnerability: Alloy produces a counterexample that breaks a web security invariant. An attacker can generate a PUT request to `attacker.com` and then redirect that request to an honest server, causing the server to receive an unexpected PUT request. Figure 3 depicts part of the counterexample produced by Alloy. Although apparently simple, this vulnerability had not been previously detected in HTML5 despite extensive review by hundreds of experts. (To be fair, HTML5 is enormous and difficult to review in its entirety.)

Solution: The easiest solution is to refuse to follow redirects of PUT or DELETE requests generated from HTML forms. We have verified this fix (up to a finite size) using our model. We have contributed our findings and recommendation to the working group [46] and the working group has adopted our solution.

E. WebAuth

In our most extensive case study, we analyze WebAuth [47], a web-based authentication protocol based on Kerberos. WebAuth is deployed at a number of universities, including Stanford University. WebAuth is similar to Central Authentication Service (CAS) [48], which was originally developed at Yale University and has been deployed at over eighty universities [49], including UC Berkeley. Although we analyze WebAuth specifically, we have verified that the same vulnerability exists in CAS.

Protocol: Of all our case studies, WebAuth most resembles a traditional network protocol. However, new security issues arise when embedding the protocol in web primitives because the web attacker can interact with the protocol in different ways than a traditional network protocol attacker can interact with, say, Kerberos. The WebAuth protocol involves three roles:

1) User-Agent (UA), the user's browser,
2) WebAuth-enabled Application Server (WAS), a web server that is integrated with WebAuth, and
3) WebKDC, the web login server.

Although WebAuth supports multiple authentication schemes, its use of tokens and keys closely resembles Kerberos. The WebKDC shares a private key with each WAS, authenticates the user, and passes the user's identity to the WAS via an encrypted token (i.e., ticket). WebAuth uses so-called "Secure" cookies to store its state and HTTPS to transmit its messages, ostensibly protecting the protocol from network attackers.

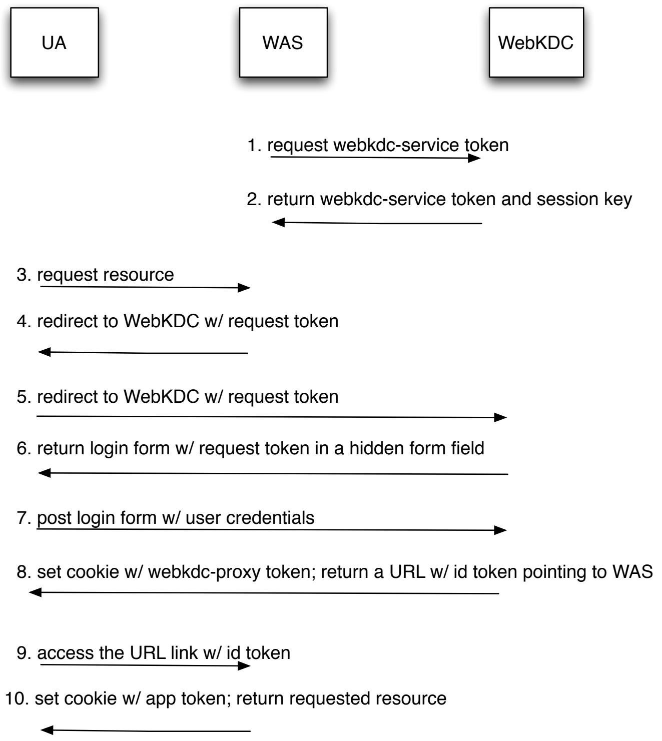
UA
WAS
WebKDC
1. request webkdc-service token
2. return webkdc-service token and session key
3. request resource
4. redirect to WebKDC w/ request token
5. redirect to WebKDC w/ request token
6. return login form w/ request token in a hidden form field
7. post login form w/ user credentials
8. set cookie w/ webkdc-proxy token; return a URL w/ id token pointing to WAS
9. access the URL link w/ id token
10. set cookie w/ app token; return requested resource

Figure 4. The WebAuth protocol

The main steps of the protocol are depicted in Figure 4. We describe the steps below:

- *WAS Initialization (Steps 1–2).* At startup, the WAS authenticates itself to the WebKDC using its private Kerberos key, and receives a *webkdc-service token* and a session key.
- *Login (Steps 3–10).* When the user wishes to authenticate to the WAS, the WAS creates a *request token* and redirects the UA to the WebKDC, passing the token in the URL. The WebKDC authenticates the user (e.g., via a user name and password), stores a cookie in the UA (to authenticate the user during subsequent interactions without requiring the user to type his or her password again), and redirects the user back to the WAS, passing an *id token* identifying the user in the URL. The WAS then verifies various cryptographic properties of the token to authenticate the user. Finally, the WAS stores a cookie in the UA to authenticate the user for the remainder of the session.

Modeling: To model WebAuth, we added a number of type signatures for the WebAuth tokens, and predicates for HTTP message validation. For example, the `WAPossessTokenViaLogin` predicate tests whether the WAS has received a proper *id token*:

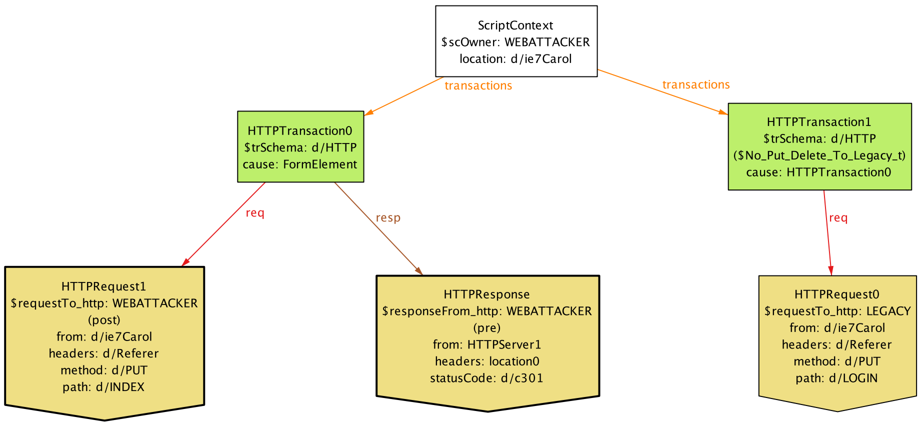

Figure 3. Counterexample generated by Alloy for the HTML5 form vulnerability.

```
pred WAPossessTokenViaLogin[httpClient:
HTTPClient, token:WAIdToken, usageEvent:Event]
{ some t1:HTTPTransaction|{
    happensBeforeOrdering[t1.req, usageEvent]
    and t1.req.path = LOGIN and
    t1.req.to in WAWebKDC and
    t1.req.from in httpClient and
    t1.resp.statusCode in RedirectionStatus and
    WAContainsIdToken[t1.resp, token] and
    token.creationTime = t1.resp.post and
    token.username in httpClient.owner
  }
}
```

The confidentiality of tokens is key to modeling the security properties of the WebAuth protocol. As with cookies, we require an HTTPClient to have received a token in a previous HTTP request before including the token in another HTTP request:

```
fact {
  all httpClient:HTTPClient,req:HTTPRequest,
  token:WAIdToken | {
    req.from in httpClient and
    WAContainsIdToken [req, token]
  } implies
    WAPossessToken[httpClient, token, req]
}
```

Notice that we permit the attacker to be registered as a user at the WebKDC and to send and receive HTTP requests and responses to the WAS and the WebKDC both directly and via the browser.

Vulnerability: Alloy produces a counterexample showing that the WebAuth protocol does not enjoy session integrity because the attacker can force the user's browser to complete the login procedure. Worse, the attacker can actually force the user's browser to complete the login procedure *with the attacker's credentials*. This previously unknown vulnerability is a variation of login CSRF [11], a vulnerability in which the attacker can confuse a web site into authenticating the honest user as the attacker. We confirmed this attack on the Stanford WebAuth implementation by embedding a link containing an *id token* of the attacker in an email, and verifying that a user who clicks on the link is logged in as the attacker to the system. The same attack works against the CAS deployment at U.C. Berkeley. Login CSRF vulnerabilities have a number of subtle security consequences [11]. One reason that Stanford or Berkeley might be concerned about these vulnerabilities is that if some information (such as a download) is available only to registered students, a registered student could log in and then export a link that allows others to access protected information, without revealing the password they used to authenticate.

The vulnerability arises because the WAS lacks sufficient context when deciding whether to send message 10. In particular, the WAS does not determine whether it receives message 3 and message 9 from the same UA. An attacker can run the first eight steps of the protocol with the WAS and WebKDC directly, but splice in the UA by forwarding the URL from message 8 to the user's browser. In a traditional network protocol, this attack might not succeed because the UA would not accept message 10 without previously sending message 3, but the attack succeeds in the web setting because (1) the UA is largely stateless, and (2) the attacker can induce the UA to send message 9 by showing the user a hyperlink on `attacker.com`.

Solution: We suggest repairing this vulnerability by binding messages 3 and 9 to the same UA via a cookie. Essentially, the WAS should store a nonce in a cookie at the UA with message 4 and should include this nonce in the *request token*. In message 8, the WebKDC includes this nonce in the *id token*, which the UA forwards to the WAS. The WAS, then, should complete the protocol only if the cookie received

Case Study	Lines of new code	No. of clauses	CNF gen. time (sec)	CNF solve time (sec)
Origin Header	25	977,829	26.45	19.47
CORS	80	584,158	24.07	82.76
Referer Validation	35	974,924	30.75	9.06
HTML5 Forms	20	976,174	27.67	73.54
WebAuth	214	355,093	602.4	35.44

Table I
VARIOUS STATISTICS FOR EACH CASE STUDY

along with message 9 matches the nonce in the *id token*. We have verified that the fixed mechanism enjoys session integrity in our model (up to a finite size).

The security of this scheme is somewhat subtle and relies on a number of security properties of cookies. In particular, this solution is not able to protect against active network attackers because cookies do not provide integrity: an active network attacker can overwrite the WAS cookie with his or her own nonce (even if the cookie is marked "Secure"). However, active network attackers can mount these sorts of attacks against virtually all web applications because an active network attacker can just overwrite the final session cookie used by the application, regardless of the WebAuth protocol. Nonetheless, our proposed solution improves the security of the protocol against web attackers.

V. MEASUREMENT

We implemented the model in the Alloy Analyzer 4.1.10. We wrote our security invariants as assertions and asked Alloy to search for a counterexample that violates these assertions, bounding the search to a finite size for each top-level signature. This bound is also called the *scope* of the search, and for our experiments was set at 8. Alloy also allows us to specify finer-grained bounds on each type, but we do not use this feature in our experiments.

For each case study, we counted the number of lines of new Alloy code we had to add to the base model (some 2,000 lines of Alloy code) to discover a vulnerability and measured the time taken by the analyzer to generate the conjunctive normal form (CNF) as well as the time taken by the SAT solver (minisat) to find a solution (see Table I). All tests were performed on an Intel Core 2 Duo CPU 3.16Ghz with 3.2GB of memory. As is common in other SAT solving applications, we observe no clear correlation between the number of lines of new code, the number of clauses generated, and the CNF generation and solving times.

The SAT solver is able to find a counterexample (if one exists) in a few minutes. In the absence of a counterexample, the time taken by the SAT solver increases exponentially as the scope is increased. To quantify this behavior, we measured the time taken to analyze the HTML5 form vulnerability before and after we implemented the fix in the

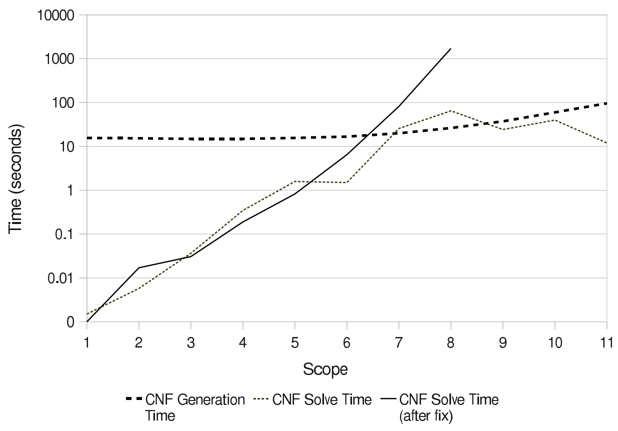

Figure 5. Log-scale graph of analysis time for increasing scopes. The SAT solver ran out of memory for scopes greater than eight after the fix.

model (see Figure 5). Recall that, after the fix, Alloy is not able to find a counterexample.

VI. CONCLUSION

In this paper, we present several steps toward a formal foundation for web security. The model described in this paper comprises key web security concepts, such as browsers, HTTP, cookies, and script contexts, as well as the security properties of these concepts. We have a clearly defined threat model, together with a spectrum of threats ranging from web attackers to network attackers to gadget attackers. In this model, we have also included two high-level security properties that appear to be commonly required of web security mechanisms.

We have implemented core portions of our model in Alloy, which lets us execute the model and check whether various web security mechanisms actually have the security properties their designers desire. We have used this implementation to study five examples, ranging in complexity from a small tweak to the behavior of the HTML form element in HTML5 to a full-blown web single sign-on protocol based on Kerberos. In each case, we found vulnerabilities, two previously known, three previously unknown. These vulnerabilities arise because of the complex interaction between different components of the web platform.

As the web platform continues to grow, automated tools for reasoning about the security of the platform will increase in importance. Already web security is sufficiently complex that a working group of experts miss "simple" vulnerabilities in web platform features. These vulnerabilities appear simple in retrospect because only a tiny subset of the platform is required to demonstrate the *insecurity* of a mechanism, whereas knowledge of the entire platform is required to demonstrate its *security*. Of course, our model (and our implementation) does not capture the entire web platform.

However, we believe our model is an important first step towards creating a formal foundation for web security.

VII. Acknowledgments

We thank David Wagner, Adrian Mettler, Matthew Finifter and William Robertson for helpful feedback on the paper and suggestions for improvements. This work is partially supported by the Air Force Office of Scientific Research under MURI Grant No. 22178970-4170, the National Science Foundation under Grant No. 0448452, the Office of Naval Research through contract N000140910681, and the National Science Foundation TRUST Center under Grant No. CCF-0424422. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the sponsors.

References

[1] E. Hammer-Lahav, "Oauth core 1.0 revision a," 2009. [Online]. Available: http://oauth.net/core/1.0a

[2] J. Hodges, C. Jackson, and A. Barth, "Strict transport security," 2009. [Online]. Available: http://lists.w3.org/Archives/Public/www-archive/2009Sep/att-0051/draft-hodges-strict-transport-sec-05.plain.html

[3] A. van Kesteren, "Cross-origin resource sharing," 2009. [Online]. Available: http://www.w3.org/TR/cors/

[4] S. Stamm, "Content security policy," 2009. [Online]. Available: https://wiki.mozilla.org/Security/CSP/Spec

[5] Microsoft Inc., "Xdomainrequest object," 2009. [Online]. Available: http://msdn.microsoft.com/en-us/library/cc288060%28VS.85%29.aspx

[6] A. Inc., "Cross-domain policy file specification," 2008. [Online]. Available: http://www.adobe.com/devnet/articles/crossdomain_policy_file_spec.html

[7] E. Hammer-Lahav, "Acknowledgement of the oauth security issue," 2009. [Online]. Available: http://blog.oauth.net/2009/04/22/acknowledgement-of-the-oauth-security-issue/

[8] T. Klose, "Confused deputy attack on cors," 2009. [Online]. Available: http://lists.w3.org/Archives/Public/public-webapps/2009AprJun/1324.html

[9] E. Nava and D. Lindsay, "Abusing internet explorer 8's XSS filters," in *BlackHat Europe*, 2010. [Online]. Available: http://p42.us/ie8xss/Abusing_IE8s_XSS_Filters.pdf

[10] Daniel Jackson, *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, 2006.

[11] A. Barth, C. Jackson, and J. C. Mitchell, "Robust defenses for cross-site request forgery," in *In Proc. of the 15th ACM Conf. on Computer and Communications Security (CCS 2008)*. ACM, 2008, pp. 75–88.

[12] A. Barth, C. Jackson, and J. Mitchell, "Securing frame communication in browsers," *Commun. ACM*, vol. 52, no. 6, pp. 83–91, 2009.

[13] C. Jackson, A. Barth, A. Bortz, W. Shao, and D. Boneh, "Protecting browsers from dns rebinding attacks," *ACM Trans. Web*, vol. 3, no. 1, pp. 1–26, 2009.

[14] C. Jackson and A. Barth, "Forcehttps: protecting high-security web sites from network attacks," in *WWW '08: Proceeding of the 17th international conference on World Wide Web*. New York, NY, USA: ACM, 2008, pp. 525–534.

[15] D. Jackson, "Alloy: a lightweight object modelling notation," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 11, no. 2, pp. 256–290, 2002.

[16] F. Kerschbaum, "Simple cross-site attack prevention," in *Proceedings of the Third international workshop on Security and Privacy in Communication networks*, 2007.

[17] J. Mitchell, M. Mitchell, and U. Stern, "Automated analysis of cryptographic protocols using Murφ," in *Proc. IEEE Symp. Security and Privacy*, 1997, pp. 141–151.

[18] J. C. Mitchell, V. Shmatikov, and U. Stern, "Finite-state analysis of ssl 3.0," in *Proceedings of the Seventh USENIX Security Symposium*, 1998, pp. 201–216.

[19] A. W. Roscoe, "Modelling and verifying key-exchange protocols using CSP and FDR," in *8th IEEE Computer Security Foundations Workshop*. IEEE Computer Soc Press, 1995, pp. 98–107.

[20] D. X. Song, "Athena: a new efficient automatic checker for security protocol analysis," in *Proceedings of the Twelfth IEEE Computer Security Foundations Workshop*, June 1999, pp. 192–202.

[21] J. Millen and V. Shmatikov, "Constraint solving for bounded-process cryptographic protocol analysis," in *CCS '01: Proceedings of the 8th ACM conference on Computer and Communications Security*. New York, NY, USA: ACM, 2001, pp. 166–175.

[22] M. Burrows, M. Abadi, and R. Needham, "A logic of authentication," *ACM Transactions on Computer Systems*, vol. 8, no. 1, pp. 18–36, 1990.

[23] G. Bella and L. C. Paulson, "Kerberos version IV: Inductive analysis of the secrecy goals," in *Proceedings of the 5th European Symposium on Research in Computer Security*, J.-J. Quisquater, Ed. Louvain-la-Neuve, Belgium: Springer-Verlag LNCS 1485, 1998, pp. 361–375.

[24] A. Datta, A. Derek, J. C. Mitchell, and A. Roy, "Protocol Composition Logic (PCL)," *Electronic Notes in Theoretical Computer Science*, vol. 172, pp. 311–358, 2007.

[25] A. Datta, A. Derek, J. C. Mitchell, V. Shmatikov, and M. Turuani, "Probabilistic polynomial-time semantics for a protocol security logic." in *Proceedings of the 32nd International Colloquium on Automata, Languages and Programming (ICALP '05)*, ser. Lecture Notes in Computer Science. Springer-Verlag, 2005, pp. 16–29.

[26] Adam Barth and Collin Jackson and John Mitchell, "Securing frame communication in browsers," in *SS'08: Proceedings of the 17th conference on Security symposium*. Berkeley, CA, USA: USENIX Association, 2008, pp. 17–30.

[27] K. Bhargavan, C. Fournet, and A. Gordon, "Verified reference implementations of ws-security protocols," *Lecture Notes in Computer Science*, vol. 4184, p. 88, 2006.

[28] A. Gordon and R. Pucella, "Validating a web service security abstraction by typing," *Formal Aspects of Computing*, vol. 17, no. 3, pp. 277–318, 2005.

[29] J. Howell, C. Jackson, H. J. Wang, and X. Fan, "Mashupos: operating system abstractions for client mashups," in *HOTOS'07: Proceedings of the 11th USENIX workshop on Hot topics in operating systems*. Berkeley, CA, USA: USENIX Association, 2007, pp. 1–7.

[30] J. Magazinius, A. Askarov, and A. Sabelfeld, "A lattice-based approach to mashup security," in *In Proc. of the 5th ACM Symposium on Information, Computer and Communications Security (ASIACCS 2010)*. ACM, 2010.

[31] C. Grier, S. Tang, and S. T. King, "Secure web browsing with the op web browser," in *SP '08: Proceedings of the 2008 IEEE Symposium on Security and Privacy*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 402–416.

[32] J. Meseguer, R. Sasse, H. J. Wang, and Y.-M. Wang, "A systematic approach to uncover security flaws in gui logic," in *SP '07: Proceedings of the 2007 IEEE Symposium on Security and Privacy*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 71–85.

[33] GWT Team, "Security for gwt applications," 2008. [Online]. Available: http://groups.google.com/group/Google-Web-Toolkit/web/security-for-gwt-applications

[34] A. Barth, J. Caballero, and D. Song, "Secure content sniffing for web browsers, or how to stop papers from reviewing themselves," in *SP '09: Proceedings of the 2009 30th IEEE Symposium on Security and Privacy*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 360–371.

[35] A. Barth and C. Jackson, "Beware of finer-grained origins," in *Proc. of Web 2.0 Security and Privacy 2008 (W2SP 2008)*. IEEE Computer Society, 2008.

[36] M. Zawelski, "Browser security handbook," 2009. [Online]. Available: http://code.google.com/p/browsersec/wiki/Main

[37] Apple Inc., "Remote scripting with IFRAME," 2010. [Online]. Available: http://developer.apple.com/internet/webcontent/iframe.html

[38] E. Felten, D. Balfanz, D. Dean, and D. Wallach, "Web spoofing: An internet con game," *Software World*, vol. 28, no. 2, pp. 6–8, 1997.

[39] R. Dhamija, J. Tygar, and M. Hearst, "Why phishing works," in *Proceedings of the SIGCHI conference on Human Factors in computing systems*. ACM, 2006, p. 590.

[40] D. Akhawe, A. Barth, P. E. Lam, J. C. Mitchell, and D. Song, "Web security model implementation," 2010. [Online]. Available: http://code.google.com/p/websecmodel

[41] L. Momtahan, "A simple small model theorem for Alloy," Oxford University Computing Laboratory, Tech. Rep. RR-04-11, June 2004.

[42] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: engineering an efficient sat solver," in *DAC '01: Proceedings of the 38th annual Design Automation Conference*. New York, NY, USA: ACM, 2001, pp. 530–535.

[43] Software Design Group, MIT, "Alloy analyzer 4," 2010. [Online]. Available: http://alloy.mit.edu/alloy4/

[44] A. Barth, C. Jackson, and I. Hickson, "The http origin header," 2009. [Online]. Available: http://tools.ietf.org/html/draft-abarth-origin

[45] A. van Kesteren, "Cross-origin resource sharing (editors draft)," 2009. [Online]. Available: http://dev.w3.org/2006/waf/access-control

[46] A. Barth, "<form method="delete"> and 307 redirects," 2009. [Online]. Available: http://www.mail-archive.com/whatwg@lists.whatwg.org/msg19379.html

[47] R. Schemers and R. Allbery, "Webauth v3 technical specification," 2009. [Online]. Available: http://webauth.stanford.edu/protocol.html

[48] D. Mazurek, "CAS protocol," 2005. [Online]. Available: http://www.jasig.org/cas/protocol

[49] JASIG, "CAS deployment," 2010. [Online]. Available: http://www.jasig.org/cas/deployments