Client Side Web Session Integrity as a Non-Interference Property

Wilayat Khan[1], Stefano Calzavara[1], Michele Bugliesi[1], Willem De Groef[2], and Frank Piessens[2]

[1] Ca' Foscari University of Venice, Italy
[2] iMinds-DistriNet, KU Leuven, Belgium.

Abstract. Sessions on the web are fragile. They have been attacked successfully in many ways, by network-level attacks, by direct attacks on session cookies (the main mechanism for implementing the session concept) and by application-level attacks where the integrity of sessions is violated by means of cross-site request forgery or malicious script inclusion. This paper defines a variant of non-interference – the classical security notion from information flow security – that can be used to formally define the notion of client-side application-level web session integrity. The paper also develops and proves correct an enforcement mechanism. Combined with state-of-the-art countermeasures for network-level and cookie-level attacks, this enforcement mechanism gives very strong assurance about the client-side preservation of session integrity for authenticated sessions.

Keywords: web security, information flow control

1 Introduction

Because of the stateless nature of the HTTP protocol, web applications that need to maintain state over multiple interactions with a client have to implement some form of *session management*: the server needs to know to which ongoing session (if any) incoming HTTP requests belong. Sessions are usually implemented by means of *session cookies*. The server generates an unpredictable random identifier at the start of a session, and sends it to the browser as a cookie. All subsequent requests from the same client will carry this cookie, and this tells the server which session incoming requests belong to. Session management is an important but vulnerable part of the modern web, in particular because client authentication is usually tied to sessions: the client is authenticated using either a password, a single-sign-on system or some multi-factor scheme, and if authentication is successful, the server marks the *session* as authenticated. Hence, attacks against session management can be used to impersonate clients to the server.

Sessions can be attacked at many layers. First, at the network layer, network sniffing or man-in-the-middle attacks can break the confidentiality or integrity of web sessions. This is a well-understood problem with well-understood solutions: by appropriate use of transport level security techniques such as SSL/TLS, these

attacks can be stopped. Second, at the session implementation layer, script injection or again network level attacks can be used to steal a session cookie and hijack the session, or to impose a session cookie on a client (a so-called session fixation attack [19, 15]). Again, this is a well-understood problem: ensuring that sessions only run over SSL/TLS, prohibiting script access to session cookies (by setting the HttpOnly and Secure attributes on session cookies), and enforcing renewal of a session on authentication, are appropriate countermeasures to such attacks. Third, sessions can be attacked at the application layer: since cookies are attached to HTTP requests by the browser automatically – without any web application involvement – any page in the browser can send malicious requests to any of the servers that the browser currently has a session with, and that request will automatically get the session cookie attached and hence will be considered as part of a (possibly authenticated) session by the server. If the page sending the malicious request is from a different origin, such attacks are called CSRF (cross-site request forgery) attacks [4]. But malicious requests can also be sent by scripts included in – or injected by an attacker into – a page from the same origin. Since both inclusions of third-party scripts [23] and script injection vulnerabilities are common [18], these are important attack vectors.

The focus of this paper is on *client-side protection against application-level attacks against sessions*. We assume that state-of-the-art countermeasures are in place for network-level and session management-level attacks, and our objective is to formally define the notion of *client-side session integrity* and to develop provably secure countermeasures for application-level attacks. While point solutions exist to protect against various forms of CSRF and script injection, the problem of application-level session integrity is not yet well-understood. There are two existing formalizations of the notion of web session integrity: Akhawe et al. [2] develop an Alloy model of the web platform and define session integrity as the property that *no attacker is in the causal chain of any HTTP request belonging to the session*. This is an excellent definition for the purpose of studying CSRF attacks and countermeasures, but the underlying model does not have a sufficiently detailed representation of scripts to study other application-level session integrity issues. In a very recent paper, Bugliesi et al. [10] are the first to provide a formal definition of session integrity that is browser-centric and amenable for client-side enforcement. They define how an attacker can influence execution traces of the browser, and then define session integrity as the property that the attacker has no effective way of interfering with an authenticated session. Based on this definition, they also design an access control/tainting mechanism that enforces session integrity at the client side.

The main objectives of this paper are (1) to refine the definition of Bugliesi et al. to a classical non-interference property [25], under the assumption that appropriate defenses against both network-level and cookie-level attacks are put in place, and (2) to design an information flow control technique that can enforce session integrity in a more permissive and fine-grained way than access control mechanisms can. This is crucial to foster the usability of the client-side protection mechanism and support collaborative web scenarios, like e-payment.

In summary, the main contributions of this paper are:

- the development of *login history dependent* non-interference for reactive systems, a variant of non-interference where the security labeling function is execution history dependent.
- the application of login history dependent non-interference to web session integrity: we show how this notion of non-interference captures the peculiarities and complexities of web session integrity.
- the development of a mechanism for enforcing login history dependent non-interference by means of secure multi-execution, with a formal proof of security.
- the design of additional improvements to this mechanism for the web context.
- a prototype implementation of the mechanism as an extension of the FlowFox information flow secure web browser.

The remainder of this paper is structured as follows. First, in Section 2, we give an informal overview of the problem of application-level session integrity and the idea of login history dependent non-interference. We formalize this in Section 3, where we define an enforcement mechanism and prove it secure. Then, in Section 4 we show how this applies to web session integrity, and in Section 5 we describe a few extensions to the formal model to make the enforcement mechanism more compatible with the web. In Section 6 we describe our prototype implementation. Sections 7 and 8 discuss related work and conclude.

2 Informal Overview

Consider a user using his web browser to interact with a number of web sites. With some of these web sites, the user has an ongoing authenticated session (for instance with his web mail provider M and with a social networking site S). Other sites have been opened in the browser by casually surfing the web, and the user has no authenticated session with them. Both pages from more trusted sites (like M or S) and less trusted sites (e.g., a web site O) might themselves consist of content retrieved from a variety of origins. A page served by M might include scripts, images and other resources from anywhere on the web.

The problem we consider in this paper is the following one: how can we make sure that the browser protects the integrity of the authenticated sessions that it has, for instance, with M, in the sense that no other web site than M itself can influence authenticated HTTP requests from the browser to M. Even if we assume (as we do in this paper) that network communication and session cookies are adequately protected, the following example attacks are still possible:

- CSRF: Pages from O can send HTTP requests to M or S, for instance by including an image or a script from these sites, or (in some cases) by sending an XHR request. The browser will automatically attach cookies to these requests, including the session cookie, and hence such requests are treated by the server as belonging to the authenticated session.

- Malicious resource inclusions: if M includes a script from some script provider (e.g. an advertisement network, a JavaScript library provider, or a web analytics company) then that script can send arbitrary authenticated HTTP requests to M.
- Client-side or reflected XSS: Pages from O can load pages from M and use a mal-formed fragment identifier or URL parameter that trigger a client-side (DOM-based) or reflected XSS vulnerability. The injected script can then send arbitrary authenticated requests to M.

A common way to formalize integrity properties such as the one above is based on concepts from information flow security. One defines a partially ordered set of security labels that represent integrity levels (in the simplest case, two labels $\top$ and $\bot$ for high, respectively low, integrity). All inputs and outputs from the program under consideration (in our case, the browser) are labeled. Inputs are labeled $\top$ if they come from a trustworthy source, and $\bot$ otherwise. Outputs are labeled $\top$ if their integrity is important and $\bot$ otherwise. A program is information flow secure (non-interferent) if low integrity inputs do not influence high integrity outputs (i.e. no information flows from low integrity sources to high integrity targets).

A complication in the case of web session integrity is that both the set of integrity labels, as well as the labeling function, evolve over time as the user logs into more sites. The same message sent by site O to site M (for instance if the page from O sends a request to load a resource from M) will be of low integrity level if the browser is currently not logged into M, and it will be of a higher integrity level if the browser *is* logged into M. This kind of login history dependent non-interference is exactly what we will formalize and then instantiate to the web context in the following sections.

3 Login History Dependent Non-Interference: Definition and Enforcement

Following Bohannon et al. [9, 8, 7], we model a browser as a *reactive system*. Then we introduce the property of *login history dependent reactive non-interference* and an enforcement mechanism for it.

3.1 Reactive system

A reactive system is a constrained labeled transition system that transforms input events into sequences of output events.

Definition 1 (Reactive System). *A* reactive system *is a tuple* $(\mathcal{C}, \mathcal{P}, \mathcal{I}, \mathcal{O}, \longrightarrow)$*, where* $\mathcal{C}$ *and* $\mathcal{P}$ *are disjoint sets of consumer and producer states respectively,* $\mathcal{I}$ *and* $\mathcal{O}$ *are disjoint sets of input and output events respectively. The last component,* $\longrightarrow$*, is a labeled transition relation over the set of states* $\mathcal{S} \triangleq \mathcal{C} \cup \mathcal{P}$ *and the set of labels* $\mathcal{A} \triangleq \mathcal{I} \cup \mathcal{O}$*, subject to the following constraints:*

1. $C \in \mathcal{C}$ *and* $C \xrightarrow{\alpha} Q$ *imply* $\alpha \in \mathcal{I}$ *and* $Q \in \mathcal{P}$*;*
2. $P \in \mathcal{P}$*,* $Q \in \mathcal{S}$ *and* $P \xrightarrow{\alpha} Q$ *imply* $\alpha \in \mathcal{O}$*;*
3. $C \in \mathcal{C}$ *and* $i \in \mathcal{I}$ *imply* $\exists P \in \mathcal{P} : C \xrightarrow{i} P$*;*
4. $P \in \mathcal{P}$ *implies* $\exists o \in \mathcal{O}, \exists Q \in \mathcal{S} : P \xrightarrow{o} Q$*.*

We limit our attention in this paper to deterministic reactive systems.

We assume given a set of web domains $\mathcal{D}$, and we stipulate that the set of input events $\mathcal{I}$ contains an event $\mathsf{login}(d)$ for all $d \in \mathcal{D}$. This event models a successful login of the browser into domain d. We assume the set of output events $\mathcal{O}$ contains an event $\cdot$ that represents a *silent* output, i.e. an internal computation step of the reactive system. A stream is defined by the coinductive interpretation of the grammar $S ::= [] \mid s :: S'$, where s ranges over individual stream elements. Bohannon et al. define the behaviour of a reactive system in a state Q as a relation between input and output streams. To handle login history dependence, we instead define it as a relation between input streams and event streams that contain both input and output events, appropriately interleaved:

Definition 2 (Reactive behaviour). *A reactive system state Q generates the event stream S from the input stream I if the judgment $Q(I) \rightsquigarrow S$ holds, where this judgment is coinductively defined by:*

$$\frac{}{C([]) \rightsquigarrow []} \qquad \frac{C \xrightarrow{i} P \qquad P(I) \rightsquigarrow S}{C(i :: I) \rightsquigarrow i :: S} \qquad \frac{P \xrightarrow{o} Q \qquad Q(I) \rightsquigarrow S}{P(I) \rightsquigarrow o :: S}$$

3.2 Login history dependent non-interference

The lattice of possible integrity levels $\mathcal{L}$ has elements $\top$ (highest integrity), $\bot$ (lowest integrity), and d for all $d \in \mathcal{D}$ (integrity level of authenticated communication with domain d). Since higher integrity information can flow to lower integrity levels but not vice-versa, we define the ordering relation on $\mathcal{L}$ as $\top \leq d \leq \bot$, and for different d and d', d and d' are incomparable.

The key idea of login history dependent non-interference (LHDNI) is to make the labeling function that assigns integrity levels to events dependent on the login events that have occurred. Initially, all network events are low integrity ($\bot$), but after a $\mathsf{login}(d)$ event, network communication with d will have level d. This models the behaviour of a web browser: because of the automatic attaching of cookies (including the session cookie), the integrity of network communication to domain d becomes more important after a login to d. It also models our assumption that the server will be more careful with HTTP responses for authenticated sessions (integrity level of these responses is higher).

The login history is represented as a finite sub-lattice L of $\mathcal{L}$, where L is initially $\{\top, \bot\}$, and L evolves with inputs processed as follows (where we write $L \oplus d$ for extending L with element d):

$$\frac{(\tau\text{-LOGIN}) \quad i = \mathsf{login}(d)}{L \xrightarrow{i} L \oplus d} \qquad \frac{(\tau\text{-NIL}) \quad i \neq \mathsf{login}(d)}{L \xrightarrow{i} L}$$

In words, whenever the user logs into a domain d, label d is added to the set of integrity labels L.

The function $lbl_L(e) : \mathcal{I} \uplus \mathcal{O} \to \mathcal{L}$ that labels events depends on the login history L. The intuition is that interactions that belong to a session with a domain d will get label d iff $d \in L$, otherwise they get label $\bot$, i.e. once logged in to d, we care about the integrity of messages to d. We stipulate that $lbl_L(\mathsf{login}(d)) = d$ for any d.

We use the notation $L^{|l}$ for the list of labels $l' \in L$ such that $l \leq l'$ and $L_{|l}$ for the list of labels $l' \in L$ such that $l' \leq l$. For an input i, for simplicity, we write $L^{|lbl_L(i)}$ as just $L^{|i}$.

LHDNI is defined in terms of the relation *LHD-similarity*, which defines when two streams look the same to an observer at level l while taking the login history into consideration.

Definition 3 (LHD-similarity). *Under login history L, two streams S and S' are LHD-similar at level l if the judgment $L \vdash S \approx_l S'$ holds, where this judgment is coinductively defined by:*

$$\frac{}{L \vdash [] \approx_l []}\ \text{(ID-NIL)} \qquad \frac{s = \mathsf{login}(d) \qquad d \leq l \qquad L \oplus d \vdash S \approx_l S'}{L \vdash s :: S \approx_l s :: S'}\ \text{(ID-LOGIN)}$$

$$\frac{s \neq \mathsf{login}(d) \qquad lbl_L(s) \leq l \qquad L \vdash S \approx_l S'}{L \vdash s :: S \approx_l s :: S'}\ \text{(ID-SIM)} \qquad \frac{lbl_L(s) \not\leq l \qquad L \vdash S \approx_l S'}{L \vdash s :: S \approx_l S'}\ \text{(ID-L)}$$

$$\frac{lbl_L(s) \not\leq l \qquad L \vdash S \approx_l S'}{L \vdash S \approx_l s :: S'}\ \text{(ID-R)}$$

Now, a state is LHDNI if l-similar inputs lead to l-similar outputs:

Definition 4 (LHDNI). *A state Q of a reactive system is LHDNI if $Q(I) \rightsquigarrow S$ and $Q(I') \rightsquigarrow S'$ imply that $\forall l \in \mathcal{L}, \emptyset \vdash I \approx_l I' \Rightarrow \emptyset \vdash S \approx_l S'$.*

Notice that it is important that we compare S and S', the event streams that contain interleaved input and output events, because of the history dependence of the definition of LHD-similarity. If we would only consider the output events, as classic non-interference definitions do, then there would be no login event present in the output streams; but we have to keep the login events there, because they influence the labeling function.

3.3 Enforcement

We now build an enforcement mechanism based on secure multi-execution (SME) [16, 6, 24]. The basic idea is to construct a new reactive system that is a wrapper

around multiple copies (sub-executions) of the original reactive system, one for each level in the login history L. When the wrapper consumes an input event, it is passed to the copies at or higher than the level of the input. When a sub-execution produces an output, if its level matches the level of the execution, the output is produced by the wrapper, otherwise it is suppressed.

A state of the wrapper is a triple (L, R, L_q), where

- L is the login history,
- R is a function mapping security labels in L to states, i.e. $R(l)$ is the sub-execution at level l, and
- L_q is a waiting queue of levels that still need to process the last input consumed. It is initially empty and when an input is consumed it is set to all levels that should process this input. We order these from low integrity to high integrity such that the sub-execution at level $\bot$ is always executed first.

States $(L, R, [])$ are consumer states, and states (L, R, L_q) with $L_q \neq []$ are producer states. The initial state of the wrapper is a state $(\{\top, \bot\}, R, [])$ with $R(\top)$ and $R(\bot)$ being the initial state of the original reactive system.

$$\text{(LOGIN)}\quad \frac{R(l) \xrightarrow{i} P_l \qquad i = \mathsf{login}(d) \qquad R'(d) = P_\top \qquad d \notin L \qquad L' = L \oplus d \qquad R'(l) = P_l \text{ for } l \in L_q \setminus \{d\} \qquad L_q = L'^{|d} \qquad R'(l) = R(l) \text{ for } l \notin L_q}{(L, R, []) \xrightarrow{i} (L', R', L_q)}$$

$$\text{(LOAD)}\quad \frac{R(l) \xrightarrow{i} P_l \qquad i \neq \mathsf{login}(d) \vee (i = \mathsf{login}(d) \text{ with } d \in L) \qquad L_q = L^{|i} \qquad R'(l) = P_l \text{ for } l \in L_q \qquad R'(l) = R(l) \text{ for } l \notin L_q}{(L, R, []) \xrightarrow{i} (L, R', L_q)}$$

$$\text{(OUT-P)}\quad \frac{R(l) \xrightarrow{o} P \qquad lbl_L(o) = l}{(L, R, l :: L_q) \xrightarrow{o} (L, R[l \mapsto P], l :: L_q)}$$

$$\text{(OUT-C)}\quad \frac{R(l) \xrightarrow{o} C \qquad lbl_L(o) = l}{(L, R, l :: L_q) \xrightarrow{o} (L, R[l \mapsto C], L_q)}$$

$$\text{(DROP-P)}\quad \frac{R(l) \xrightarrow{o} P \qquad lbl_L(o) \neq l}{(L, R, l :: L_q) \dashrightarrow (L, R[l \mapsto P], l :: L_q)}$$

$$\text{(DROP-C)}\quad \frac{R(l) \xrightarrow{o} C \qquad lbl_L(o) \neq l}{(L, R, l :: L_q) \dashrightarrow (L, R[l \mapsto C], L_q)}$$

Fig. 1. Basic semantics for secure multi-execution of a reactive system

The semantics is shown in Figure 1. The main extension with respect to standard SME for reactive systems [6] is the way in which login events are handled: these update the login history L, and hence also the number of sub-executions in the wrapper, and (implicitly) the labeling function lbl_L. Note how

the newly created sub-execution at level d is initialized: $P_\top$ is the resulting state after giving i to $R(\top)$, i.e. we essentially clone the sub-execution at level $\top$ and feed it i. This is the right thing to do, as we want the newly created sub-execution to have seen all the events of higher integrity than d. The (LOAD) rule handles other input events than initial login events. It essentially feeds the input to all sub-executions with a level $\leq lbl_L(i)$, by updating the appropriate sub-executions in R to a state where they have received i, and by setting the waiting queue to contain all levels that have to process this input event. The other four rules implement the SME output rules, making sure that output of level l is only performed by the execution at level l. They also make sure that, as sub-executions return to a producer state, the next sub-execution in the waiting queue gets a chance to run.

These rules effectively block *all* cross-origin requests to authenticated domains. For instance, if a page received from an unauthenticated domain (a $\bot$ event) loads an image from an authenticated domain d, the corresponding HTTP request (a d-level event) will be suppressed.

We can do substantially better: instead of dropping such requests, we can strip the session cookie from the request as in other client-side CSRF protection systems [10, 14]. We assume the existence of a function $strip^L(o)$ that for any o with $lbl_L(o) = d$ (for some d) strips the session cookies from o, and for all other o returns o.

We define the projection functions π_l^L as follows:

$$\pi_l^L(o) = \begin{cases} strip^L(o) & \text{if } l = \bot \\ o & \text{otherwise} \end{cases}$$

We assume that the event labeling function lbl_L checks for the presence of an authentication cookie to deem a network output as a high integrity event. Hence $lbl_L(strip^L(o))$ is always $\bot$.

$$\text{(OUT-P)}\quad \frac{R(l) \xrightarrow{o} P \qquad release_{L,l,L_q}(o)}{(L, R, l :: L_q) \overset{\pi_l^L(o)}{\rightarrow} (L, R[l \mapsto P], l :: L_q)}$$

$$\text{(OUT-C)}\quad \frac{R(l) \xrightarrow{o} C \qquad release_{L,l,L_q}(o)}{(L, R, l :: L_q) \overset{\pi_l^L(o)}{\rightarrow} (L, R[l \mapsto C], L_q)}$$

$$\text{(DROP-P)}\quad \frac{R(l) \xrightarrow{o} P \qquad \neg release_{L,l,L_q}(o)}{(L, R, l :: L_q) \dashrightarrow (L, R[l \mapsto P], l :: L_q)}$$

$$\text{(DROP-C)}\quad \frac{R(l) \xrightarrow{o} C \qquad \neg release_{L,l,L_q}(o)}{(L, R, l :: L_q) \dashrightarrow (L, R[l \mapsto C], L_q)}$$

Fig. 2. Semantics for secure multi-execution of a reactive system (updated)

The basic semantics (Figure 1) released an output o from a sub-execution at level l only if $lbl_L(o) = l$. We can now generalize this: a sub-execution at level l can release $\pi_l^L(o)$ if the following predicate holds:

$$release_{L,l,L_q}(o) \quad = \quad lbl_L(o) = l \vee (l = \bot \wedge lbl_L(o) \notin L_q)$$

That is, an output is *released* from a sub-execution if its label matches the label l of the sub-execution, or when $l = \bot$ and there is no sub-execution at the level of the output in the waiting queue. Since we process sub-executions in the order from low integrity to high integrity, this means that this output is being sent in response to an input that was *not* of level $lbl_L(o)$, and hence is a cross-domain request to an authenticated domain. We show the updated rules in Figure 2.

3.4 Security

We now show that the enforcement mechanism defined above guarantees LHDNI. All the proofs of lemmas and theorems are given in the full version [21].

Theorem 1 (Security). *All the initial states of the wrapper are LHDNI.*

We prove the theorem using Bohannon's ID-bisimulation proof technique [9]. It suffices to prove that there exists an ID-bisimulation $\approx_l$ such that for every state of the wrapper (L, R, L_q), we have $(L, R, L_q) \approx_l (L, R, L_q)$. The proof of security consists of two steps: first we have to define the relation $\approx_l$ and then we need to show that it is indeed an ID-bisimulation relation. Note the overloading of the $\approx_l$ notation. When used between streams, it is interpreted as LHD-similarity (Definition 3), when used between reactive system states, it refers to the definition below.

Definition 5 (*l-similarity* relation $\approx_l$). *The state (L_1, R_1, L_{q1}) is l-similar to the state (L_2, R_2, L_{q2}) (written $(L_1, R_1, L_{q1}) \approx_l (L_2, R_2, L_{q2})$) iff:*

- $L_{1|l} = L_{2|l}$, *and*
- $R_1 \approx_l R_2$, *meaning* $\forall l' \leq l$: $R_1(l') = R_2(l')$, *and*
- $L_{q1|l} = L_{q2|l}$.

Lemma 1. *The l-similarity relation is an ID-bisimulation relation.*

4 Instantiation to Web Session Integrity

In this section, we show by example how LHDNI protects browsers from typical attacks on session integrity. Recall that we assume that best practices for session security (i.e. the use of SSL/TLS and the use of the Secure and HttpOnly attributes on session cookies) are in place. We assume that login events are recognizable by the browser; they are triggered for instance by a bookmarklet or password manager, and the response page of the site that one is logging into is shown in a separate top-level frame (tab) in the browser. The browser should enforce that logins to these known and trusted domains *must* happen through these bookmarklets, to avoid attacks such as login CSRF [4].

We show by example how remaining attacks such as classic CSRF and malicious script inclusion are countered by our enforcement mechanism. A similar example can be constructed for client-side or reflected XSS.

Applying the enforcement mechanism described by the semantics in Figure 2 to web browsers requires us to define the sets of input and output events for a browser. We limit our attention to a simple set of events that can model the attacks we care about. These events are described in Table 1 (the first 4 events are input events, the last 4 are output events). The table also shows the value of the lbl_L function.

All these events are standard browser events and easy to recognize by the browser (for the $\mathsf{login}(d)$ event because of the assumptions we made above).

Table 1. User actions, input/output events and their labels

User actions	I/O events	lbl_L	
		$d \in L$	$d \notin L$
typing URL to domain d in the address bar	$\mathsf{ui\_load}(d)$	$\top$	$\top$
network response from domain d with header h	$\mathsf{net\_resp}(d, h)$	d	$\bot$
clicking link on the page from domain d	$\mathsf{ui\_link\_click}(d)$	d	$\bot$
entering password on the page from domain d	$\mathsf{login}(d)$	d	d
network request to domain d (incl. cookie)	$\mathsf{net\_req}(d)$	d	$\bot$
network request to domain d (no cookie)	$\mathsf{net\_req}(d)$	$\bot$	$\bot$
loading a page at the screen	$\mathsf{ui\_page\_loaded}$	$\bot$	$\bot$
dummy	$\cdot$	$\bot$	$\bot$

CSRF Figure 3 gives a schematic overview of a classic CSRF attack. The user signs into web site A (messages 1-4) and opens a page in another tab from malicious web site E (messages 5-8), which implicitly sends a cross-origin request to load remote content (e.g. an image) from A (message 9). As the browser will attach all the cookies with this request to A, it will lead to a CSRF attack on A.

Figure 4 shows an encoding of this attack in our browser model, and shows how our enforcement mechanism stops the attack. Each line of the encoding is of the form $(E, [Rule]) : (L, R, []) \xrightarrow{n} (L', R', L_q)$, where E is the input or output event, $Rule$ is the semantics rule (Figure 2), (L, R, L_q) represents the state of the wrapper and n is the message number in the corresponding interaction diagram figure. Outputs are shown slightly indented, so that it is easy to see by which input event they are caused. We write L_0 for the set $\{\bot, \top\}$, and L_A for the set $\{\bot, A, \top\}$. For simplicity, the finite list $l_1 :: l_2 :: []$ is denoted with $l_1 :: l_2$. If we do not care about a specific component of the browser state, we write $\_$.

Events and semantics rules corresponding to each event in Figure 3 are shown in Figure 4. In this scenario, using a standard web browser, the attack would happen in message 9, where the request to A (initiated in response from E) would include cookies. However, under the wrapper, the attack is prevented.

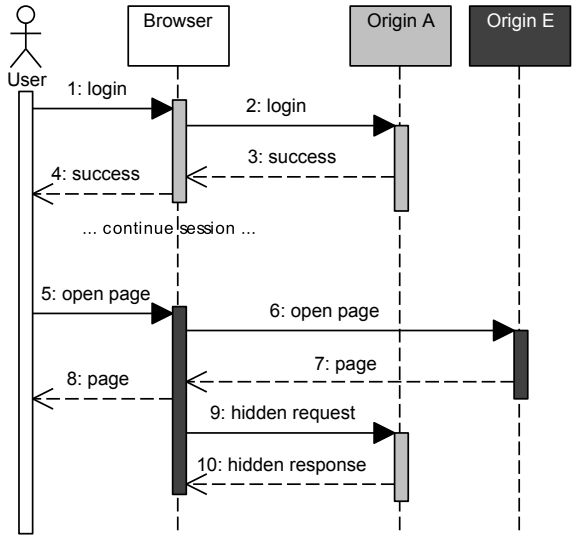

Fig. 3. Classic CSRF

Specifically, the basic semantics in Figure 1 would drop the request, since a low integrity sub-execution is not allowed to send A-labeled requests; the updated semantics in Figure 2, instead, would strip the cookies from the request for the very same reason. Both options are secure, but the second option will break less existing web sites.

Malicious script inclusion Figure 5 gives a schematic overview of a script inclusion attack. The user signs into web site A (messages 1-4) and then opens a page (messages 5-6). This page includes a script tag that will include a third party script from E. When the page from A is being rendered (messages 7-8), the remote script is loaded from the web site E (message 9-10). The script can then for instance install an event handler that will trigger an (authenticated) request to A at a later time.

This example is encoded in Figure 6. The response input from E (message 10) gets a $\bot$ label, hence is fed only into the low integrity sub-execution. All the requests to A initiated by the user (in the context of A) or directly by input from A are released from the sub-execution at level A, and hence are not affected by the script injected to the sub-execution at $\bot$. Requests released from the $\bot$ sub-execution may be affected but as those outputs do not include cookies, they are safe. In the example, the request to A (message 12) as the result of the user input (message 11) is released from the execution at $\bot$ (release line 12 in Figure 6). The sub-execution at A label never received the script from E, so it will not react to the link click, and just output a silent event $(\cdot)$.

1. ($\mathsf{login}(A)$, [LOGIN]): $(L_0, \_, []) \xrightarrow{1} (L_A, \_, \bot :: A)$
   2. suppress ($\mathsf{net\_req}(A)$, [DROP-C]): $(L_A, \_, \bot :: A) \dot{\rightarrow} (L_A, \_, A)$
   2. release ($\mathsf{net\_req}(A)$, [OUT-C]): $(L_A, \_, A) \xrightarrow{2} (L_A, \_, [])$
3. ($\mathsf{net\_resp}(A, h)$, [LOAD]): $(L_A, \_, []) \xrightarrow{3} (L_A, \_, \bot :: A)$
   4. release ($\mathsf{ui\_page\_loaded}$, [OUT-C]): $(L_A, \_, \bot :: A) \xrightarrow{4} (L_A, \_, A)$
   4. suppress ($\mathsf{ui\_page\_loaded}$, [DROP-C]): $(L_A, \_, A) \dot{\rightarrow} (L_A, \_, [])$
5. ($\mathsf{ui\_load}(u_E)$, [LOAD]): $(L_A, \_, []) \xrightarrow{5} (L_A, \_, \bot :: A :: \top)$
   6. release ($\mathsf{net\_req}(E)$, [OUT-C]): $(L_A, \_, \bot :: A :: \top) \xrightarrow{6} (L_A, \_, A :: \top)$
   6. suppress ($\mathsf{net\_req}(E)$, [DROP-C]): $(L_A, \_, A :: \top) \dot{\rightarrow} (L_A, \_, \top)$
   6. suppress ($\mathsf{net\_req}(E)$, [DROP-C]): $(L_A, \_, \top) \dot{\rightarrow} (L_A, \_, [])$
7. ($\mathsf{net\_resp}(E, h)$, [LOAD]): $(L_A, \_, []) \xrightarrow{7} (L_A, \_, \bot)$
   8. release ($\mathsf{ui\_page\_loaded}$, [OUT-P]): $(L_A, \_, \bot) \xrightarrow{8} (L_A, \_, \bot)$
   9. release w/o cookies ($\mathsf{net\_req}(A)$, [OUT-C]): $(L_A, \_, \bot) \xrightarrow{9} (L_A, \_, [])$

Fig. 4. Classic CSRF attack encoding and prevention

5 Extensions

The enforcement mechanism described by the formal semantics in Figure 2 enforces security policies to protect against attacks on session integrity, but by doing so it does break some common web scenarios that technically violate session integrity, but do so without malicious purposes. These scenarios can be handled in our approach by means of *endorsement* (the integrity variant of *declassification* [24, 28]).

Endorsements will typically have to be declared by the web site that the browser has an authenticated session with. In the two approaches below, these declarations are done by means of request headers, similar to how Content Security Policy (CSP) [27] policies are communicated to the browser.

Endorsing script inclusions A first, simple and common kind of endorsement is for script inclusion. The script inclusion example in Figure 5 is commonly *not* an attack: web site A includes the script from E intentionally and trusts it to influence the session. While some scripts can be usefully included without having the possibility to influence the session (e.g. analytics scripts), inclusion of other scripts is only useful when these scripts have the right to influence the session (e.g. the jQuery library).

Fortunately, endorsing script inclusions is straightforward. The server A declares in a HTTP header which origins can provide trusted scripts, and the browser uses this information to label outgoing and incoming requests to these white-listed origins from A's pages as being of level A. One could even argue that this should be the default interpretation of the CSP policy directives that allow script inclusions (e.g. the `script-src` directive).

Endorsements for collaborating applications Endorsements are also required for collaborating web applications such as e-payment systems (e.g. Paypal). Con-

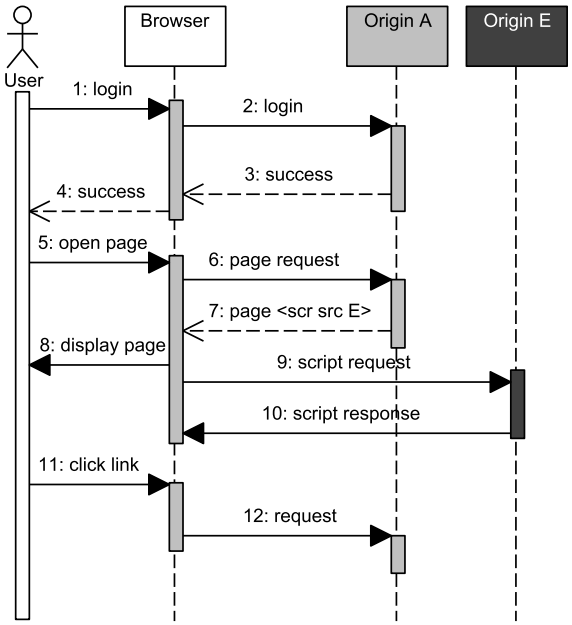

Fig. 5. Script inclusion attack

1. (login(A), [LOGIN]): $(L_0, \_, []) \xrightarrow{1} (L_A, \_, \bot :: A)$
   2. suppress (net_req(A), [DROP-C]): $(L_A, \_, \bot :: A) \xrightarrow{\cdot} (L_A, \_, A)$
   2. release (net_req(A), [OUT-C]): $(L_A, \_, A) \xrightarrow{2} (L_A, \_, [])$
3. (net_resp(A, h), [LOAD]): $(L_A, \_, []) \xrightarrow{3} (L_A, \_, \bot :: A)$
   4. release (ui_page_loaded, [OUT-C]): $(L_A, \_, \bot :: A) \xrightarrow{4} (L_A, \_, A)$
   4. suppress (ui_page_loaded, [DROP-C]): $(L_A, \_, A) \xrightarrow{\cdot} (L_A, \_, [])$
5. (ui_link_click(A), [LOAD]): $(L_A, \_, []) \xrightarrow{5} (L_A, \_, \bot :: A)$
   6. suppress (net_req(A), [DROP-C]): $(L_A, \_, \bot :: A) \xrightarrow{\cdot} (L_A, \_, A)$
   6. release (net_req(A), [OUT-C]): $(L_A, \_, A) \xrightarrow{6} (L_A, \_, [])$
7. (net_resp(A, h), [LOAD]): $(L_A, \_, []) \xrightarrow{7} (L_A, \_, \bot :: A)$
   8. release (ui_page_loaded, [OUT-P]): $(L_A, \_, \bot :: A) \xrightarrow{8} (L_A, \_, \bot :: A)$
   9. release (net_req(E), [OUT-C]): $(L_A, \_, \bot :: A) \xrightarrow{9} (L_A, \_, A)$
   8. suppress (ui_page_loaded, [DROP-P]): $(L_A, \_, A) \xrightarrow{\cdot} (L_A, \_, A)$
   9. suppress (net_req(E), [DROP-C]): $(L_A, \_, A) \xrightarrow{\cdot} (L_A, \_, [])$
10. (net_resp(E, h), [LOAD]): $(L_A, \_, []) \xrightarrow{10} (L_A, \_, \bot)$
    release ($\cdot$, [DROP-C]): $(L_A, \_, \bot) \xrightarrow{\cdot} (L_A, \_, [])$
11. (ui_link_click(A), [LOAD]): $(L_A, \_, []) \xrightarrow{11} (L_A, \_, \bot :: A)$
    12. suppress (net_req(A), [DROP-C]): $(L_A, \_, \bot :: A) \xrightarrow{\cdot} (L_A, \_, A)$
    12. release ($\cdot$, [OUT-C]): $(L_A, \_, A) \xrightarrow{\cdot} (L_A, \_, [])$

Fig. 6. Script inclusion attack encoding and prevention

sider, for example, a user who wants to buy an airline ticket at web site A and pay via *paypal.com* (Figure 7).

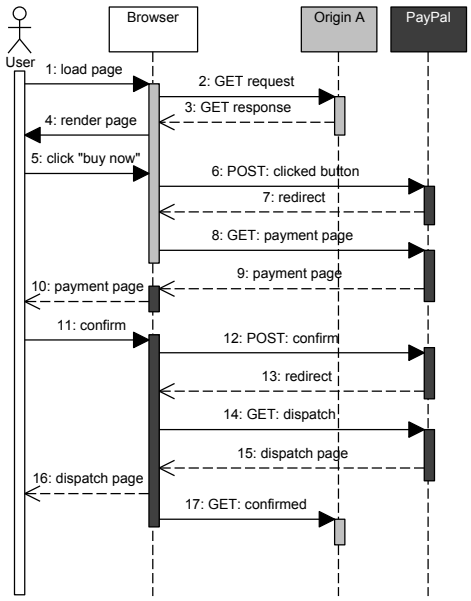

Fig. 7. E-payment scenario

The user opens a page from web site A where he clicks the *buy* button and then the user confirms the payment on the *paypal.com* web site. Messages 3-4 and 15-17 of Figure 7 are encoded in our model in Figure 8. We assume the user is logged into both A and P (Paypal), i.e. L contains both A and P.

The message 17 (*GET: confirmed*) is a cross-origin request to A and hence the wrapper will release it from the execution at $\bot$. As all the session cookies are erased, the payment operation will fail.

To support such collaborating web applications, endorsement is needed. For these cases, we propose the use of a response header, used by the web site to specify allowed entry points from different origins. A web site s (source of white-list) sends a list of URLs url pointing to s specifying that another site w (white-listed site) is allowed to send cross-origin requests to these URLs, by setting a *connect-destination (cd)* header $<cd: \{W{:}w, U{:}url\}>$ in the response.

The wrapper will keep track of these headers by updating a set ω of key-value pairs of the form (w, url), where w is the white-listed web site (the *who* part) and url is the list of URLs (the *how* part) specified as the allowed entry points white-listed for the w. The list of URLs url can also include URLs with wildcard character $*$ such as $s.com/*$, where the web site w can send cross-origin (authenticated) requests to any URL of the site $s.com$.

3. (net_resp(A, h), [LOAD]): $(L, \_, []) \xrightarrow{3} (L, \_, \bot :: A)$
   4. release (ui_page_loaded, [OUT-C]): $(L, \_, \bot :: A) \xrightarrow{4} (L, \_, A)$
   4. suppress (ui_page_loaded, [DROP-C]): $(L, \_, A) \dot{\rightarrow} (L, \_, [])$

...
(user clicks "buy" button, and confirms payment)
...

15. (net_resp(P, h), [LOAD]): $(L, R, []) \xrightarrow{15} (L, \_, \bot :: P)$
   16. release (ui_page_loaded, [OUT-P]): $(L, \_, \bot :: P) \xrightarrow{16} (L, R, \bot :: P)$
   17. release w/o cookies (net_req(u_A), [OUT-C]): $(L, \_, \bot :: P) \xrightarrow{17} (L, R, P)$
   16. suppress (ui_page_loaded, [DROP-P]): $(L, \_, P) \dot{\rightarrow} (L, \_, P)$
   17. suppress (net_req(u_A), [DROP-C]): $(L, \_, P) \dot{\rightarrow} (L, \_, [])$

Fig. 8. E-payment application encoding

As a simple example, assume two web sites A and B send the endorsement headers <*cd:* {*W:P, U:*[*a.com*/∗]}> and <*cd:* {*W:P, U:*[*b.com*/*u*1, *b.com*/*u*2]}> in their responses. Initially, when the response from A is received, the wrapper will store in ω an entry $(P, [a.com/*])$ and when the other response from B is received, it will add the two URLs to the value bound to P, hence ω will become $(P, [a.com/*, b.com/u1, b.com/u2])$. The URL $a.com/*$ represents all the URLs of web site A.

Now we have the required information to decide if a cross-origin request should be endorsed. After receiving the example headers above, an output from P to any URL of A or to any of the two URLs $b.com/u1$ and $b.com/u2$ of web site B should include cookies. On receipt of an input event i with label d, the wrapper will compute the set of URLs that d is allowed to send cross-origin requests to by looking it up in ω. Let us call the resulting set U_i.

We generalize the *release* predicate so that it takes U_i into account. An output is *released* from a state if (1) its label matches the label l of the current sub-execution or, (2) when $l = \bot$ and there is no sub-execution at the level of the output and the request URL is not white-listed, or (3) when $l \neq \bot$, and the request URL is white-listed. The predicate $release_{L,l,L_q}(o, u, U_i)$ is defined as follows:

$$l = lbl_L(o) \vee (l = \bot \wedge lbl_L(o) \notin L_q \wedge u \notin U_i) \vee (l \neq \bot \wedge u \in U_i).$$

We can now show that the Paypal example (Figure 7) works. We show an encoding in our model in Figure 9. (We show ω and U_i as the third and fourth component of the tuple representing the extended browser state.)

Assume the site A sends the header <*cd:* {*W:P, U:*[*a.com*/∗]}> in the response input (message 3, Figure 7) and the wrapper creates the entry $\omega = (P, [a.com/*])$. Later on, when the input in message 15 is received from P, the corresponding list of URLs for P is retrieved, that is, $U_i = [a.com/*]$. The encoding in Figure 8 will now change as shown in Figure 9. The *GET: confirmed*

cross-origin (legitimate) request to web site A is now sent from the sub-execution at label P *with* its authentication cookie.

...

15. (net_resp(P, h), [LOAD]): $(L, \_, \omega, [], []) \xrightarrow{15} (L, \_, \omega, [a.com/*], \bot :: P)$
    16. release (ui_page_loaded, [OUT-P]):
        $(L, \_, \omega, [a.com/*], \bot :: P) \xrightarrow{16} (L, \_, \omega, [a.com/*], \bot :: P)$
    17. suppress (net_req(u_A), [DROP-C]):
        $(L, \_, \omega, [a.com/*], \bot :: P) \dot{\rightarrow} (L, \_, \omega, [a.com/*], P)$
    16. suppress (ui_page_loaded, [OUT-P]):
        $(L, \_, \omega, [a.com/*], P) \dot{\rightarrow} (L, \_, \omega, [a.com/*], P)$
    17. release (net_req(u_A), [OUT-C]):
        $(L, \_, \omega, [a.com/*], P) \xrightarrow{17} (L, \_, \omega, [a.com/*], [])$

Fig. 9. E-payment application encoding (updated)

6 Implementation

Our prototype implementation is constructed as a modification of the FlowFox browser [12, 13]. Crucial for our implementation is the ability to keep track of all sites a user is logged into and to make sure that the labelling of JavaScript API calls can be dependent on this login history.

The biggest modification to FlowFox's core is the addition of a shared state variable, shared between all browser windows. This variable contains the login history log of the browser. This history log is a list of strings and contains all domain names for which the browser has established an authenticated session. In our prototype, authentication to a web site has to happen by means of a bookmarklet that interacts with this login history log to add authenticated domains.

The second modification is in the policy library that comes with FlowFox. This library now offers an API to query the login history log so that the labelling of JavaScript API calls can depend on this information. We illustrate in a small example how the extended FlowFox can be used.

New top level windows exist only in the low integrity copy of the browser, *unless* the new window is created by a login bookmarklet. In that case, the new window will exist in two levels of the browser: the $\bot$ level, and the level of the authenticated origin.

Consider again the classic CSRF scenario from Figure 3. This executes in our prototype as follows. First, the user starts an authenticated session with `mail.com` by selecting the appropriate bookmarklet. This bookmarklet posts the correct login credentials (stored in the bookmarklet) over HTTPS to `mail.com`. The bookmarklet also interacts with FlowFox's core to store `mail.com` in the login history log. Next, the user loads a page from the `attacker.com` site, and

a script on this page tries to influence the current session with `mail.com` by crafting an XMLHttpRequest.

For this example, we configure FlowFox with the policy that makes calls to XMLHttpRequest of d integrity if they go to a domain d in the login history log, and low integrity ($\bot$) otherwise.

When the user visits `attacker.com`, and the script performs an XHR request to `mail.com`, the window containing `attacker.com` exists only in the $\bot$ level of the browser, and the policy above causes the request to be suppressed. Hence, this policy effectively prevents the classic CSRF attack as described in Fig. 3. Requests that go to other sites (with no open authenticated session) would be left untouched. Blocking a request is done by making sure that the `skipCall` primitive used internally by FlowFox (it is hidden from the policy writer by the policy library, which is in fact a domain specific language on top of those primitives) returns the appropriate value.

The current protoype is just a proof-of-concept, and has important limitations. The most important one is that FlowFox only performs multi-execution of JavaScript code, and hence no policies can be enforced on network requests that are not triggered by scripts. If `attacker.com` tries to influence the session with `mail.com` via other means, e.g., an embedded image tag, thereby not relying on any JavaScript code, we have no way to intercept this in FlowFox. Removing this limitation is possible by multi-executing the entire browser, as proposed by Bielova et al. [6], but that would require a major overhaul of FlowFox and hence a substantial implementation effort. Despite this limitation, we believe the prototype is evidence of the feasibility of our proposed mechanism in real browsers.

Our prototype implementation is available online at `http://distrinet.cs.kuleuven.be/software/FlowFox/`.

7 Related Work

There has been a wide variety of work on web session integrity over the past decade. The lines of work most closely related to our contributions are: (1) formal models of web session integrity, (2) countermeasures against CSRF, and (3) information flow control for the web.

7.1 Formal models of web session integrity

Bohannon et al. [9] propose reactive non-interference, a non-interference property for reactive programs such as web scripts that is proposed to replace the Same Origin Policy in browsers. This was a direct inspiration for our notion of login history dependent non-interference. Later, Bohannon and Pierce [8] developed Featherweight Firefox, a formal model of a simple browser, with the purpose of formally studying confidentiality and integrity policies for browsers, including reactive non-interference policies. This browser model did not yet model session management, and very recently Bugliesi et al. [10] developed Flyweight Firefox, a

variant of Featherweight Firefox, and provided a formal definition of web session integrity as well as a provably sound enforcement mechanism. The advantage of our approach is that, by providing information flow control instead of access control, we can more precisely enforce session integrity.

An alternative approach to formally model session integrity was taken by Akhawe et al. [2]. They develop a coarse grained model of the entire web platform in Alloy, and use bounded model checking to find flaws in proposed web security techniques. They model the entire web platform, whereas in our approach we focus on modeling the browser only. Hence, their model is better suited to evaluate security techniques that span client and server, whereas our model is more suitable for pure client-side enforcement techniques.

7.2 Countermeasures against CSRF

CSRF is the most important session integrity attack that is not handled by just protecting the session implementation layer. Server-side countermeasures against CSRF are well-understood. The most widely deployed countermeasure is the use of anti-CSRF tokens. We limit our attention to related work on client-side enforcement. Client-side enforcement of CSRF protection was pioneered by RequestRodeo [20]. This system interposed a proxy between client and server, and stripped authentication information from suspicious requests. Many variants of RequestRodeo have been proposed [14, 26, 1], differing in (1) how suspicious requests are detected, (2) how suspicious requests are handled (either dropping them or stripping session cookies, or just detecting the attack), and (3) the implementation technique (as a proxy or as a browser extension). All these variants are useful but heuristic solutions, that provide no formal assurance. The only system that provides some formal guarantees is CsFire [14]: it was formally validated through bounded model checking to defend against CSRF in the formal model of the web developed by Akhawe et al. [2].

Our approach for endorsements, where the server tunes or sets a browser policy, is closely related to existing server-driven policies on the web, like Content Security Policies [27], or Allowed Referrer Lists [11].

7.3 Information flow control for the web

Information flow control in web scripts is usually proposed by means of dynamic mechanisms [22] due to the dynamic nature of the JavaScript language, the de facto programming language on the client side web applications. Our work is directly based on existing information flow secure browsers that use the mechanism of secure multi-execution [16] for information flow control. The theoretical development is based on Bielova et al. [6], whereas the implementation extends the FlowFox browser [12, 13]. Alternative dynamic information flow control mechanisms for browser scripts are usually monitors. Austin and Flanagan [3] and Hedin and Sabelfeld [17] study runtime monitors for non-interference in JavaScript-like languages. Bichhawat et al. [5] formalize and develop an information flow monitor at the level of JavaScript bytecode in the WebKit engine.

8 Conclusions

Web session security is a key cornerstone of web security. We have shown how client-side application-level web session integrity can be understood as a non-interference property. To make this possible, we introduce LHDNI, login-history-dependent non-interference, and show how this notion captures client-side web session integrity. We also developed and proved correct an enforcement mechanism based on secure multi-execution. A prototype implementation in the FlowFox browser is available online.

There are many avenues for future work. While we have formally proven security of our enforcement mechanism, we believe the mechanism has several other interesting properties that deserve a formal study. In particular we believe it to be *precise* in the sense that it does not impact the observable behaviour of the browser as long as the browser is only visiting secure sites. In other words, security is not overapproximating: the enforcement mechanism only does something observable if the browser is definitely behaving insecurely. We also believe that we can prove compatibility results saying that – under some conditions – behaviour of existing sites is preserved, even if they do something insecure; our approach of stripping session cookies instead of blocking requests could allow us to show that such sites behave *as if the browser was not logged into other sites.*

Acknowledgments. This research is partially funded by the Research Fund KU Leuven, by the IWT project SPION, and by the MIUR projects ADAPT and CINA. Willem De Groef holds a PhD grant from the Agency for Innovation by Science and Technology in Flanders (IWT).

References

1. `https://www.requestpolicy.com/security.html`
2. Akhawe, D., Barth, A., Lam, P.E., Mitchell, J., Song, D.: Towards a formal foundation of web security. In: CSF (2010)
3. Austin, T.H., Flanagan, C.: Multiple Facets for Dynamic Information Flow. In: Proc. of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 165–178 (2012)
4. Barth, A., Jackson, C., Mitchell, J.C.: Robust defenses for cross-site request forgery. In: Proceedings of the 15th ACM Conference on Computer and Communications Security. pp. 75–88 (2008)
5. Bichhawat, A., Rajani, V., Garg, D., Hammer, C.: Information flow control in webkits javascript bytecode. In: Principles of Security and Trust, pp. 159–178 (2014)
6. Bielova, N., Devriese, D., Massacci, F., Piessens, F.: Reactive non-interference for a browser model. In: Proc. of the International Conference on Network and System Security. pp. 97–104 (2011)
7. Bohannon, A.: Foundations of web script security. Ph.D. thesis, University of Pennsylvania (2012)
8. Bohannon, A., Pierce, B.C.: Featherweight firefox: Formalizing the core of a web browser. In: Proceedings of the 2010 USENIX Conference on Web Application Development. pp. 11–11. WebApps'10, USENIX Association, Berkeley, CA, USA (2010)

9. Bohannon, A., Pierce, B.C., Sjöberg, V., Weirich, S., Zdancewic, S.: Reactive Noninterference. In: Proceedings of the ACM Conference on Computer and Communications Security. pp. 79–90 (2009)
10. Bugliesi, M., Calzavara, S., Focardi, R., Khan, W., Tempesta, M.: Provably sound browser-based enforcement of web session integrity. In: CSF 2014
11. Czeskis, A., Moshchuk, A., Kohno, T., Wang, H.J.: Lightweight server support for browser-based csrf protection. In: Proceedings of the 22Nd International Conference on World Wide Web. pp. 273–284 (2013)
12. De Groef, W., Devriese, D., Nikiforakis, N., Piessens, F.: FlowFox: a Web Browser with Flexible and Precise Information Flow Control. In: Proc. of the ACM Conference on Computer and Communications Security. pp. 748–759 (2012)
13. De Groef, W., Devriese, D., Nikiforakis, N., Piessens, F.: Secure multi-execution of web scripts: Theory and practice. Journal of Computer Security (2014)
14. De Ryck, P., Desmet, L., Joosen, W., Piessens, F.: Automatic and precise client-side protection against csrf attacks. In: European Symposium on Research in Computer Security (Esorics) (2011)
15. De Ryck, P., Nikiforakis, N., Desmet, L., Piessens, F., Joosen, W.: Serene: Self-reliant client-side protection against session fixation. In: DAIS (2012)
16. Devriese, D., Piessens, F.: Noninterference Through Secure Multi-Execution. In: Proc. of the IEEE Symposium on Security and Privacy. pp. 109–124 (2010)
17. Hedin, D., Sabelfeld, A.: Information-Flow Security for a Core of JavaScript. In: Proc. of the IEEE Computer Security Foundations Symposium. pp. 3–18 (2012)
18. Johns, M.: On JavaScript Malware and Related Threats - Web Page Based Attacks Revisited. Journal in Computer Virology 4(3), 161 – 178 (August 2008)
19. Johns, M., Braun, B., Schrank, M., Posegga, J.: Reliable protection against session fixation attacks. In: Proceedings of the 2011 ACM Symposium on Applied Computing. pp. 1531–1537 (2011)
20. Johns, M., Winter, J.: In: Proceedings of the OWASP Europe 2006 Conference. pp. 5 – 17 (2006)
21. Khan, W., Calzavara, S., Bugliesi, M., De Groef, W., Piessens, F.: Client side web session integrity as a non-interference property: Extended version with proofs, available at `http://www.cs.kuleuven.be/publicaties/rapporten/cw/CW674.abs.html`
22. Le Guernic, G.: Confidentiality Enforcement Using Dynamic Information Flow Analyses. Ph.D. thesis, Kansas State University (2007)
23. Nikiforakis, N., Invernizzi, L., Kapravelos, A., Van Acker, S., Joosen, W., Kruegel, C., Piessens, F., Vigna, G.: You Are What You Include: Large-scale Evaluation of Remote JavaScript Inclusions. In: Proc. of the ACM Conference on Computer and Communications Security. pp. 736–747 (2012)
24. Rafnsson, W., Sabelfeld, A.: Secure multi-execution: Fine-grained, declassification-aware, and transparent. In: CSF (2013)
25. Sabelfeld, A., Myers, A.C.: Language-Based Information-Flow Security. IEEE Journal on Selected Areas of Communications 21(1), 5–19 (January 2003)
26. Shahriar, H., Zulkernine, M.: Client-side detection of cross-site request forgery attacks. In: Software Reliability Engineering (ISSRE), 2010 IEEE 21st International Symposium on. pp. 358–367 (Nov 2010)
27. Stamm, S., Sterne, B., Markham, G.: Reining in the web with content security policy. In: Proceedings of the 19th international conference on World wide web. pp. 921–930. ACM (2010)
28. Vanhoef, M., De Groef, W., Devriese, D., Piessens, F., Rezk, T.: Stateful declassification policies for event-driven programs. In: CSF (2014)