SUMMARY The emergence of Web 2.0 technologies such as Ajax and Mashup has revealed the weakness of the same-origin policy [1], the current de facto standard for the Web browser security model. We propose a new browser security model to allow fine-grained access control in the client-side Web applications for secure mashup and user-generated contents. We propose a browser security model that is based on information-flow-based access control (IBAC) to overcome the dynamic nature of the client-side Web applications and to accurately determine the privilege of scripts in the event-driven programming model.

key words: Web security, browser security, access control, information-flow control

1. Introduction

Ajax offers new modes of Web application construction involving asynchronous data exchanges between the clients (i.e., browsers) and servers, and using JavaScript to update the GUI via the DOM (Document Object Model) and style sheets. Ajax supports interactive user interfaces without reloading the webpages, and this makes possible desktop-application-like user experiences for Web applications. For example, word processors and spreadsheets have been the major native desktop applications on PCs, but they can now be implemented as Ajax applications [2].

Mashup is an application programing model that combines multiple content sources into a single user experience. A typical mashup application might integrate a third-party map service and a list of stores to interchangeably search for the shops near some location on the map, or to find the locations of the shops on the map.

The de facto security policy of current browsers is called the same-origin policy [1]. The same-origin policy assumes that contents (e.g., HTML documents) downloaded from the same server can trust each other, and therefore limits the communication between contents from different origins. The origin of the content is determined by the protocol, port number, and the server name of the content URLs.

Each set of same-origin content is confined within a sandbox, which is a browser window or a frame. Each sandbox holds a DOM (Document Object Model) tree representing an HTML document. The same-origin policy prohibits access between DOMs or JavaScript objects that belong to different origins. External script files imported into the HTML document by the $\texttt{<script src='...'/>}$ elements are regarded as part of the HTML document, and thus run within the same sandbox as the main document.

The XMLHttpRequest (XHR) is a de facto standard API that is implemented in most Web browsers. XHR allows a client-side script to issue an arbitrary HTTP request to a remote server. The same-origin policy is also applied to XHR, which means an HTTP request can only be issued to a server that belong to the same-origin as the client-side content. However, the emergence of Web 2.0 technologies such as Ajax and Mashup has revealed design flaws of the same-origin policy.

First, the same-origin policy assumes that the content on the same server is mutually trusted, which does not hold for content generated by many users or mashup applications. It is quite possible that the content on the same server includes malicious scripts from some users or third parties. In fact, Cross-Site Scripting attacks have been the most serious security threat to Web applications for some years [3].

Second, the same-origin sandboxes still need mechanisms to allow communication between content from the different origins. A typical example that bypasses the restriction of XHR is to use the $\texttt{src}$ attribute of an $\texttt{<img>}$ element to send arbitrary information to an arbitrary remote server, by passing the information as part of the URL request parameters. This technique is widely used in XSS attacks to steal session cookies and send them to remote attackers. If an attacker uses the $\texttt{src}$ attribute of a $\texttt{<script>}$ element, he can easily implement bi-directional communication between the browser and a remote server, taking advantage of the fact that the returned script content will be executed on the client-side [4]. Since the network accesses via URL references in the HTML attributes are not under the control of the same-origin policy, an attacker can communicate with arbitrary servers without using XHR. Similar loopholes exist in communications between windows and frames on a browser, which are deliberately used by Ajax frameworks [5], [6] to enable cross-domain communications.

This paper proposes a new secure browser model that mitigates the design flaws of the same-origin policy and mitigates the threats to Web applications. In our proposed browser model, all data, i.e., contents received from servers,
are associated with the security label which identifies the origin of the data. The labels represents the security domain of the URL from which the content is originated. In addition, the user and each content provider can define the access control policy to control how content (i.e., script) can access resources, such as the browser’s internal DOM tree and the document cookies. The access control policy can also control execution of script as well as network access by JavaScript (e.g., XMLHttpRequest) and HTML elements and attributes (e.g., by using an <img> element). The access control policy is defined based on the origin of the content. In order to enforce the access control policy properly in JavaScript, which has dynamic nature, the proposed access control model is built on top of information-flow based access control (IBAC) [13]. In IBAC, the privilege of the content is judged based on the origin of the data used in each operation. The origin of the data is tracked through script execution. IBAC has advantages over the Stack-based access control (SBAC), a code-origin based access control mechanism which is widely used in Java. Since IBAC assesses the privilege of the content not only based on stack inspection but also the origin of method parameters, our proposed model can enforce access control policies based on the originator of an action even in dynamic and self-mutating client-side Web applications.

The rest of the paper is organized as follows: Section 2 describes some attack scenarios. Section 3 offers some observations on the access control models. Section 4 describes the detailed design of a secure browser model and its operational semantics. Section 5 shows the correctness of the proposed model. Section 6 presents an example scenario and evaluates the safety of the model with sample attacks. Section 7 makes observation on some design decisions. Section 8 reviews related work. Section 9 concludes the paper with our future research agenda.

2. Motivating Scenarios: Cross-Site Scripting

This section describes an attack scenario that is enabled by the shortcomings of the current browser security model, and then gives overview of how the proposed browser model prevents the attacks.

XSS is a type of attack in which an attacker injects malicious script into an innocent web content, allowing the script to execute in a victim’s Web browser. XSS can be categorized into three types: persistent, reflected, and DOM-based.

**Persistent XSS.** Typically script is injected into a Web message board or blog comments. The injected script is stored persistently in the database, and executed when a victim browses the content.

**Reflected XSS.** The reflected XSS uses a vulnerability of a server that ‘reflects off’ unsanitized user input. An attacker may inject a malicious script into the request parameter of a URL, and embed the URL in spam mail, to trick the victim into clicking on the link to trigger the attack.

**DOM-based XSS.** The DOM-based XSS is possible when the client-side application is designed to read a URL string from document.location, document.URL or document.referrer, and inject part of it into the DOM [7].

Once a malicious script is triggered, it is executed within the same-origin sandbox of the innocent content, and therefore allows an attacker to steal document cookies or user passwords, or compromise data in a Web application for the purpose of phishing [8]. Likewise, Mashup applications have the same vulnerabilities as XSS, because third-party content is often integrated and executed in the same-origin context as trusted content.

A common practice to prevent XSS is to filter out scripts on the server side. However, in case of DOM-based XSS, server-side filtering is not possible because malicious URLs are handled only on the browser side. Even when the server side filtering is logically possible, it is difficult to correctly filter data because new ways of bypassing filters are constantly invented [9], [10]. It is reported that 73% of Web applications suffer from XSS vulnerabilities [3]. Therefore, it is important to provide a mechanism to protect users from vulnerable Web applications.

2.1 DOM-Based XSS and Our Approach

This section describes the details of how a DOM-based XSS works, and how our proposed browser model prevents the attack.

1. The user receives a malicious URL, such as http://foo.com/bar?name=john#<script>... ...</script> via a channel outside of the browser, such as in spam e-mail.
2. When the URL is passed to the browser, e.g., by the user double clicking on the link in the spam mail, the browser sends the HTTP request GET /bar?name=john to the server foo.com. The fragment identifier after the # sign will not be sent to the server.
3. The server foo.com returns a HTML document which includes vulnerable JavaScript in 1.
4. The URL of the document, including the fragment identifier, will be assigned to the browser’s built in variables such as document.location.
5. The returned HTML document is parsed and rendered by the browser, and the embedded JavaScript code is executed. The vulnerable JavaScript code in Fig. 1 tries to retrieve the name parameter in the URL, and then print it on the page. However, since the value of the variable s will include the entire string after name=, not only “Welcome john” but also the following script will be inserted into the document. As a result, the script will be executed.

This attack cannot be prevented by the same-origin policy, because the URL string is not explicitly associated with
any origins but implicitly trusted. It is implicitly regarded as in the same-origin as the HTML document, even if it comes from an untrustworthy source.

In our proposed browser model, all data, including the URL and any content received from servers, are associated with the security labels that identify the origin of the data. The labels are associated with the URL from which the content originated. Optionally, the labeling policy, which is defined by the user and the content providers, can define finer granular labels to parts of the content. In the above example, the URL string that comes from outside of the browser cannot be trusted, and thus is given the label ⊤ which indicates the distrusted status of the data.

The access control policy, which is also defined by the user and the content providers, decides how content (i.e., script) can access resources, such as the browser’s internal DOM tree and the document cookies. The access control policy can define that content with the label ⊤ does not have permission for script execution.

When the script in Fig. 1 is executed, the security label ⊤ of document.location will be propagated through the script execution. Therefore, the variable s will also have the label ⊤, because its value will be a substring of document.location. As a result, when the variable s is written into the DOM by the document.write method, the script stored in s will not be executed because of the lack of the execute permission for ⊤. Note that the value in s is associated with a set of security labels that represent all of the origins on which the value of s depends. For example, let’s say that the script of Fig. 1 is part of an HTML document from a.com and associated with a security label A. The security label of s will be the composition of the label A and ⊤, because s consists of substrings from both origins. All the security labels associated with s need to have execute permission in order to execute the string of s.

In addition, in the proposed model, a content provider may associate a finer grained security label with parts of the document. For example, a provider may associate a different security label with the <div> elements which contain user generated messages, and give fewer permissions to that content, in order to mitigate risks of persistent XSS.

### 3. Access Control Models

In order to enable the functionalities depicted in the motivating scenarios, our browser model control access based on the origin of the content (e.g., script).

Stack-based access control (SBAC), or stack-inspection, is an access control mechanism based on the origin of the code. In SBAC, when a privileged function is called, the security manager checks the origin of the functions on the call stack, and in principle, the action is granted only when all of the functions on the call stack have the required privileges. By checking the call stack, the mechanism prevents the confused-deputy attack by malicious code that tries to call privileged functions via trusted libraries. SBAC was introduced in Java2 [11] to control the behavior of untrusted mobile code.

Our browser model uses stack inspection in order to support access control based on the code origin. Stack inspection is necessary to prevent unauthorized code from accessing the sensitive features via trusted JavaScript library code, such as JavaScript widget libraries.

Some implementations of JavaScript have already adopted SBAC in order to enforce the same-origin policy. In particular, Mozilla supports signed JavaScript to grant extra permissions for trusted scripts. However, unlike Java which has namespace separation and private scopes to enable encapsulation, SBAC is not sufficient for JavaScript, which lacks the notion of the private scope and allows overriding the existing objects and functions. Therefore, an attacker may override objects and their properties, and then compromise the behavior of the trusted code. To prevent such attacks, the Mozilla runtime grants additional permissions to signed JavaScript code only when all of the scripts within a webpage are signed and have those permissions. If scripts are signed by different entities, only the intersection of the permission sets associated with all of the signers is granted [12]. This means that when a webpage includes a piece of unsigned script, no additional permissions are granted to any of the signed JavaScript within the page.

One problem with SBAC is that it cannot prevent the types of attacks that indirectly control the behavior of the code via function parameters [13]. In addition, the dynamic nature of JavaScript makes it more difficult for SBAC to control the privileges of the code. The built-in eval() function allows executing an arbitrary string as a piece of JavaScript code, and thus the originator of the string has full control over the program behavior. Dynamic DOM update, e.g., by document.write() or by the DOM API, will trigger JavaScript execution when the inserted nodes include scripts, and thus results in the same effect as the eval() function. The event-driven programming model of JavaScript also makes it difficult to identify the action initiator by examining the call stack, because the initiator of an action cannot be detected from the call stack when an event handler is invoked.

We propose a novel browser security model based on the Information-flow-Based Access Control (IBAC) [13] to identify the origin of the objects (including data and functions) involved in privileged operations. IBAC is an access control model which makes access decisions not only based on the origin of the code on the...
call stack but also based on the origin of the parameters. IBAC prevents the attacks using data that originated from distrusted code. By applying IBAC to JavaScript, we can overcome the problem of object overriding, and can determine the action initiators even when the objects are overridden.

Figure 2 shows a scenario where IBAC in the browser is effective. This example shows an HTML document $A$, which includes (or mash-up) an external map service $E$ with a `<script src='..'>` which includes (or mash-up) an external map service. The node $W$ denotes the browser window object. $E$ has write permission to the DOM node only under $A_3$ which is a `<div>` element for rendering the map image. Suppose $E$ has no execute permission. When $E$ writes malicious script under the node $A_3$ along with the map data, the script will be stored in the DOM tree as text data, but not executed due to the access control policy. However, if some script in $A$ somehow copies the data in $A_3$ and pastes it in $A_3$, the script is executed if we only employ the stack inspection. The information-flow based access control can detect the origin of the data in $A_3$ and prevents script from executing even in such a case.

4. Secure Browser Model

This section defines our secure browser model in more detail.

4.1 Simplified Browser Model

Table 1 shows a simplified model of the Web browser environment and its elements, as well as its JavaScript functionalities.

The set $\Sigma$ is the world of the Web, and represents mappings between URLs $u_l$ and data $D$, where each data is associated with policy set $P_D$.

$B$ is a browser instance which consists of windows (i.e., windows or frames) $w$, each of which contains a main document $D_w$ and multiple sub-contents $D$. A sub-content is an in-line image or external script file. The browser also contains 1) a set of primitive operations $PO$, 2) a cookie store $\mathcal{K}$ for the mapping from a URL $u_l$ to cookie $k$, 3) a variable store $\mathcal{H}$ for the browser’s object store, 4) a label store $\mathcal{L}$ for associating the runtime security labels to objects, such as JavaScript objects and DOM nodes in a given browser state ($\mathcal{L}[x]$ is the security labels for the object $x$), and 5) a policy store $P$ that stores the policies activated in the browser, i.e., the browser user’s preference and the policies associated with the content. (The labels and the policies are described in the next section)

The URL $u$ may be a standard URL $u_l$ that represents a network location (i.e., $u_l = d/p$ where $d$ is the domain and $p$ is the path) or a virtual javascript: URL $u_l$ that includes JavaScript.

Data $D$ is the content downloaded from some URL. An HTML Document $D_h$ includes DOM nodes $n$. For the sake of simplicity, we consider only JavaScript and passive inline images as the sub-content that can be imported into the document. The DOM node $n$ may be a `<script>`, `<img>` or other passive elements (such as `<div>`) for formatting purposes. A `<script>` element executes either embedded JavaScript code or an external JavaScript file. An `<img>` element downloads an image from the URL specified in the `src` attribute. Passive elements also may be associated with event handlers. For simplicity, we consider `<div>` to be a passive element and `onEvent` is its event handler attribute that has JavaScript code as a value.

Script statements include nop, expression, assignment, and sequences of script statements. Note that conditional branches and loops are intentionally excluded from the model for the sake of simplicity, because we consider only direct information flows. We do not consider implicit flows[14] in our model, since we assume that content from two origins do not conspire to attack each other, and implicit flows can do very little to compromise the code integrity under this assumption. (E.g., when $A$ does not conspire with $E$ to attack $A$ itself, we think that it is quite difficult for $E$ to control $A$’s behavior by implicit information flow.)
4.2 Access Control and Labeling Policy

Table 2 shows the labels and policies that are used in the proposed browser security model. In particular, the definitions of $l$, $L$, $P_l$, $P_A$, $s$, $o$, and $a$ correspond to the abstract grammar of the labeling and access control policies.

A primitive label $l$ is a shortname (or an alias) of a URL, and identifies the security domain that a content belongs to. Each content provider has its own namespace for the primitive labels; i.e., a primitive label defined by a content provider is effective only for the access control policy that is provided by the same content provider. In the rest of the paper, a primitive label defined by an entity $a$ is denoted as $s\text{string}_a$, where $\text{string}$ is the name of the label.

Security label $L$ is a set of primitive labels. We use Denning’s lattice model for the security labels [14]. That is, given a finite set of primitive security labels $L_{all} = \{l_1, l_2, \ldots, l_n\}$, we construct a lattice for the power set of $L_{all}$. The top of the lattice $T = L_{all}$ and the bottom $\bot = \{\}$. A binary operator $\cup$ denotes the least upper bound (LUB). The operator $\sqcap$ denotes a partial ordering of the lattice [14].

Policy $P^l_o = \{l_1, l_2, \ldots\}$ is a label policy which consists of policy elements $e = (l, u, x, \#xp)$, each of which associates a primitive label to a URL $u$. Optionally an XPath expression can be used to give a different label on some part of the document. $P^l[u, \#xp]$ is the labeling policy of a node specified by the XPath $xp$ in a document downloaded from a URL $u$.

Policy $P_A$ is the access control policy, which consists of the tuples of a subject $s$, an object $o$, and a signed action $\pm a$. The subject is the action initiator, and is identified by the security label of the content such as script. The object $o$ is the target of the action, such as a DOM node $n$ or a network location $u$. In order to specify a subject or an object in an access control policy, one may also use $*$ to specify all the labels in $L_{all}$ and $\neg$ to specify all the labels except for $l$. Action types include read ($r$), write ($w$), execute ($e$), automatic cookie attachment ($c$), and $+$ denotes all of them. The read action represents the Web browser behavior of reading (or receiving) data, such as reading data from a DOM or receiving data from a Web server. In contrast, the write action represents the behavior of writing or sending data. The execute action represents the execution of JavaScript code. The automatic cookie attachment action represents the behavior of a Web browser that attaches cookies to HTTP requests. More detailed meanings for these actions are defined in Sect. 4.4.

The sign $+$ or $-$ indicates the positive or negative permission for the action, so the rule $(s, o, -a)$ indicate that the action $a$ is not allowed. We write $(s, o, (+a_1, -a_2))$ as a shorthand of $(s, o, (+a_1), (s, o, -a_2))$. The absence of a sign implies $+$.

A content provider may associate the label policy $P^l_0$ and access control policy $P^a_o$ with each piece of data $D$. Each user may define the user policies $P^l_u$ and $P^a_u$ which are associated with his or her browser instance $B$.

Browser’s label store $L$ stores mapping from runtime data (such as JavaScript objects, variables and DOM nodes) to the security label $L$. Note that we use two notions of the label in this paper. First is the container label, which is defined in the label policy $P^l$. The container label identifies the container of data, such as a DOM node or a network location. Second is the data label, which indicates the origin of the data in a given browser state. The data label is stored in the label store $L$. The initial data label is determined from the container label which provides the data, and the data label propagates as the script is executed. On the other hand, the container label of a DOM node or a URL does not change even if data with different labels are written into it.

For example, assume that two HTML documents a.html and e.html are downloaded from a.com and e.com respectively. The document a.html is associated with the content policy $P^l_a$ and $P^a_a$ provided by a.com, and the document e.html is associated with the content policy $P^l_e$ and $P^a_e$ from e.com. When $P^l_a = \{A_a = \text{a.com}, (E_a = \text{e.com})\}$ and $P^l_e = \{A_e = \text{a.com}, (E_e = \text{e.com})\}$, then DOM nodes in a.html and e.html will be associated with the container label $L = \{A_a, A_e\}$ and $L = \{E_a, E_e\}$ respectively. (i.e., each DOM node in a.html has the primitive label $A_a$ defined by a.com and the primitive label $A_e$ defined by e.com).

If e.html executes script var v2="eee" and then e.html execute script var v1=v2+"aaa" and then the security label of each data is determined as follows:

1. The data label on string literal "eee" is $L["eee"] = L'$ because it inherits the container label.
2. The data label on variable v2 will be $L[v2] = L'$ after the assignment because the data propagates from the string literal to the variable.
3. The data label on string literal "aaa" is $L["aaa"] = L'["aaa"] = \ldots$
4. The data label on the variable $v$ will become $(L \cup L') = \{A_0, A_c, E_a, E_c\}$ and indicates $v$ is the composition of the data from a.html and e.html.

The subject ($s$) in the access control policy is determined from the composite label of the content involved in the operation, i.e., the composition of the labels of the functions on the call stack as well as the function parameters. The object ($o$) in the access control policy is identified by the container label of the target object.

For example, in the previous example, suppose that script in a.html tries to call a function $f_x$ in e.html, and $f_x$ is trying to send an HTTP request to 'http://a.com/'. In such a case, $L = \{A_0, A_c\}$ is the container label of the URL 'http://a.com/' which is the target (object) of the access. On the other hand, the composite label $(L \cup L') = \{A_0, A_c, E_a, E_c\}$ is the subject (object) of the access.

The access is granted and an HTTP request is issued only when $\{(A_0, A_c), (+r, +w)\}$ is defined in a.html's policy $\mathcal{P}_a^D$, and $\{(A_0, A_c), (+r, +w)\}$ is defined in e.html's policy $\mathcal{P}_e^A$.

The effective policy set that is enforced at run-time is the composition of the user-defined policies and the policies associated with the stakeholder’s content. A stakeholder is some content that is the target of the access. For example, when content in a.html accesses the DOM node or cookies in e.html, and then the access control policy of e.html will be observed.

**Definition 1** (Effective Policy Set): The effective policy set is $(\mathcal{P}_a^D \cup \mathcal{P}_e^A)$, where $\mathcal{P}_a^D$ is the access control policies in the user preferences, and $\mathcal{P}_e^A$ is the access control policies associated with the data $D$.

Not all DOM nodes or URLs are associated with an explicit container label. Intuitively, when no explicit labeling policy is defined for a DOM node, an explicit container policy defined for the closest ancestor node is applied. Likewise, for a document at URL $u_1$, an explicit container policy defined for URL $u_2$ will be applied when $u_2$ has the identical server name and the longest path name prefix.

Formally, we introduce an operator $\geq$ to denote hierarchical relationships between objects and data.

**Definition 2** (Object Hierarchy): Let $o_i$ and $o_j$ be DOM nodes. $o_i \geq o_j$ if $o_i$ is an ancestor node of $o_j$. Let $D_i$ and $D_j$ be documents at the URLs $d_i[p_i]$ and $d_j[p_j]$ respectively, then $D_i \geq D_j \iff d_i = d_j \land \text{prefix}(p_i, p_j)$, where $\text{prefix}(x, y) = true$ when the path $x$ is the prefix of the path $y$. The relation $\geq$ is reflexive, transitive and anti-symmetric.

**Definition 3** (Explicit and Implicit Container Label): Let $ep(o)$ and $ip(o)$ be the explicit and implicit container label of an object $o$ (i.e., either a data $D$ or a DOM node $n$ in $D$, i.e., $D.n$) respectively, and $o.u#xp$ be the URL of $D$ followed by the XPath $xp$ of the node $D.n$. The explicit container label of an object $o$ is $L'$, where $i = (L', u#xp') \in \mathcal{P}_L^D$ where $o.u#xp = u#xp'$. The implicit container label of $o$, $ip(o)$ is defined only if $ep(o) = \{\}$ as follows; $ip(o_i) = ep(o_j)$ where $ep(o_j) \neq \phi$ and there is no $o_p$ such that $o_j > o_p > o_i$ where $ep(o_p) \neq \phi$. If neither explicit nor implicit label is defined, the default label $\top$ is assumed.

### 4.3 Operational Semantics

Figure 3 shows the semantics of the operations in the big-step style [15]. An operational semantics of the browser is defined by a set of inference rules, each consists of pre-conditions (above the bar) and post-conditions (below the bar). That is, when a browser operation is being executed, the operation is evaluated as defined in the post-condition, when the pre-condition is satisfied.

We write $(S, C, B) \not\downarrow (v, C', B')$ to denote that when script $S$ is executed under the initial call stack $C$ and the browser state $B$, it evaluates to the value $v$ and terminates with the call stack $C'$ and the browser state $B'$.

We write $B[H]$ to denote the state of the variable store, and $B.L$ to denote the state of the label store in the browser. For example, we write $B.L[v]$ to denote the security label of a value $v$ stored in the browser’s label store. We write $B' = B.L[v \mapsto L]$ when, from an initial state $B$, the label of object $v$ is updated to $L$. The browser state after this transition is denoted as $B'$. Likewise, we write $B' = B[H][x \mapsto v]$ when the value of variable $x$ is updated to $v$ under the state $B$, and then the state transits to $B'$.

Rule 1 shows the semantics of when a string literal is evaluated. The value is evaluated as the string itself, and the browser’s call stack ($C$) and the state ($B$) do not change. The pre-condition is omitted in this rule since this operation does not depend on it. The initial label of a literal value is determined from the container label of the content that includes the literal, and is not explicitly stated in the rule because that simply means that the label does not change. We omit the rules for $true$, $false$ and $null$ since they are identical to Rule 1.

Rule 2 shows the semantics of when a variable reference is evaluated. The value $v$ of the variable $x$ is retrieved from the browser’s variable store, such as $v = B[H][x]$. In addition, the security label $L$ of the variable $x$ is retrieved from the label store $B.L$, such as $L = B.L[x]$, and then stored as the label of the value $v$, such as $B' = B.L[v \mapsto L]$. The post-condition shows that the browser state after this operation will be $B'$, which definition is given part of the pre-condition.

Rule 3 shows an assignment operation. First, an expression $E$ is evaluated to a value $v$, and then the label $L$ of the variable $v$ is taken from the label store $B.L$, and stored as the label of the variable $x$. The value $v$ is stored in the variable store with the variable name $x$, i.e., $B'' = B'.H[x \mapsto v]$. The post-condition of the assignment $x = E$ becomes $B''$.

Rule 4 shows a binary operation. We write $@$ to represent an arbitrary binary operator. The label of the composite
value \( v = v_1 \oplus v_j \) becomes the Least Upper Bound (LUB) of the labels of the original data \( v_1 \) and \( v_j \), i.e., \( L_1 \sqcup L_j \).

Rule 5 shows a function declaration. In JavaScript, a function is a first-class object. When a function is declared and its body consists of script \( S \), the label of a declared function object \( f \) is determined by the data label of the script \( S \) in the function body.

Rule 6 shows the execution of a sequential script. Rest of the rules show execution of a sensitive operation \( I \) that requires permission for the corresponding actions. When a sensitive operation \( I \) is invoked with the actual arguments \( E^* \), the permission of the invocation is determined from the composition of all the security labels of script in the call stack \( C \) and the arguments. The operation is allowed when each of the data labels in \( L_1 \) has permission to execute the action to the container object \( o \) in the access control policy.

Rule 7 shows the operational semantics of \( \text{readDOM}(\text{Exp}) \), where \( \text{Exp} \) is an expression that is evaluated to \( xp \), the XPath expression of the target DOM node. \( D_\text{h} \) is the URL of the main HTML document and \( D_\text{h}[xp] \) is the DOM node specified by an XPath \( xp \). Intuitively, the operation is carried out with the following steps:

1. Argument \( \text{Exp} \) is evaluated to the XPath \( xp \).
2. The composition of the data labels in the call stack \( L \) and the argument \( L[xp] \) is determined. This composite label is represented as \( L_2 \) in the rule and corresponds to the subject (or \( s \)) in the access control policy.
3. The container label \( L_3 \) of the target object, which is the DOM node \( D_\text{h}[xp] \) that can be identified by the URL and XPath \( (D_\text{h}.\#xp) \) is determined from the labeling policy \( P_L \). The label \( L_3 \) corresponds to the object (or \( o \)) in the access control policy. The value in the node is represented as \( v_n \).
4. The permission for reading the DOM node is determined from the access control policy \( P_A \). Intuitively, all primitive labels in \( L_2 \) need to have read permission for \( L_3 \), which is the container label that represents the security domain of the target DOM node. The operation is aborted if \( L_2 \) has insufficient permissions.
5. If \( L_2 \) has read permission, then the \( \text{readDOM} \) operation returns the node value \( v_n \). The browser state after the operation becomes \( B_3 \), and the label store \( L \) will be updated such that the security label of the value \( v_n \) is equal to the security label of the DOM node \( D_\text{h}[xp] \).

We introduce the predicate hasPerm\((L_2, L_3, a)\) to validate access permission.

\[\text{Definition 4 (Permission Check):} \quad \text{hasPerm}(L_2, L_3, a) \text{ checks whether the access being taken is allowed in the stakeholders' policy that is stored in the policy store } P_A. \text{ The stakeholders are the user and the entities who's resources are to be protected in the operation. In case of the access to DOM or cookies as well as function calls and the script execution by } \text{eval}, \text{ the stakeholders are the user and the HTML document } D_\text{h}. \text{ The stakeholders' policies are the user policy } P^u_A \text{ and the content policy } P^C_A. \text{ In case of the network access, the stakeholders are the user, the} \]

\[\text{function is a first-class object. When a function is declared and its body consists of script } S, \text{ the label of a declared function object } f \text{ is determined by the data label of the script } S \text{ in the function body.} \]

Rule 6 shows the execution of a sequential script.

Rest of the rules show execution of a sensitive operation \( I \) that requires permission for the corresponding actions. When a sensitive operation \( I \) is invoked with the actual arguments \( E^* \), the permission of the invocation is determined from the composition of all the security labels of script in the call stack \( C \) and the arguments. The operation is allowed when each of the data labels in \( L_1 \) has permission to execute the action to the container object \( o \) in the access control policy.

Rule 7 shows the operational semantics of \( \text{readDOM}(\text{Exp}) \), where \( \text{Exp} \) is an expression that is evaluated to \( xp \), the XPath expression of the target DOM node. \( D_\text{h} \) is the URL of the main HTML document and \( D_\text{h}[xp] \) is the DOM node specified by an XPath \( xp \). Intuitively, the operation is carried out with the following steps:

1. Argument \( \text{Exp} \) is evaluated to the XPath \( xp \).
2. The composition of the data labels in the call stack \( L \) and the argument \( L[xp] \) is determined. This composite label is represented as \( L_2 \) in the rule and corresponds to the subject (or \( s \)) in the access control policy.
3. The container label \( L_3 \) of the target object, which is the DOM node \( D_\text{h}[xp] \) that can be identified by the URL and XPath \( (D_\text{h}.\#xp) \) is determined from the labeling policy \( P_L \). The label \( L_3 \) corresponds to the object (or \( o \)) in the access control policy. The value in the node is represented as \( v_n \).
4. The permission for reading the DOM node is determined from the access control policy \( P_A \). Intuitively, all primitive labels in \( L_2 \) need to have read permission for \( L_3 \), which is the container label that represents the security domain of the target DOM node. The operation is aborted if \( L_2 \) has insufficient permissions.
5. If \( L_2 \) has read permission, then the \( \text{readDOM} \) operation returns the node value \( v_n \). The browser state after the operation becomes \( B_3 \), and the label store \( L \) will be updated such that the security label of the value \( v_n \) is equal to the security label of the DOM node \( D_\text{h}[xp] \).

We introduce the predicate hasPerm\((L_2, L_3, a)\) to validate access permission.

\[\text{Definition 4 (Permission Check):} \quad \text{hasPerm}(L_2, L_3, a) \text{ checks whether the access being taken is allowed in the stakeholders’ policy that is stored in the policy store } P_A. \text{ The stakeholders are the user and the entities who’s resources are to be protected in the operation. In case of the access to DOM or cookies as well as function calls and the script execution by } \text{eval}, \text{ the stakeholders are the user and the HTML document } D_\text{h}. \text{ The stakeholders’ policies are the user policy } P^u_A \text{ and the content policy } P^C_A. \text{ In case of the network access, the stakeholders are the user, the} \]
HTML document $D_h$, and the target network server. I.e., hasPerm($L_o, a = true \iff \forall l \in L_o, \forall sh \in SH : \exists p' = (s', o', +a) \in \mathcal{P}^a_{sh} \land \forall l' \in L_{o'} : \exists p'' = (s', o', -a) \in \mathcal{P}^a_{sh}$, where $s' = l \land o' = L_o$. $SH$ is the list of the stakeholders and $\mathcal{P}^a_{sh}$ is the access control policy of a stakeholder.

Rule 8 shows the operational semantics of writeDOM ($E_{xp}, E_{v}$), where $E_{xp}$ is the XPath of the parent node under which the new node will be appended, and $E_v$ is the value to write. When writeDOM is executed, the browser evaluates $E_{xp}$ and $E_v$ to $xp$ and $v$ respectively, and then parses the value $v$ and generates a set of DOM nodes $n_v$, i.e., $n_v = parse(v)$. The composite security label ($L_v$) of the call stack and the argument $xp$ is determined from the label store. The security label of the target node is determined from the labeling policy for $D_h$. The script $D_h$ is the $xp$ and the argument $v$ respectively. Then the composite security label 

Rule 9 shows the operational semantics of readCki. The permission is determined by checking if the composite label of the script on the call stack has write (or w) permission for the entire document $D_h$ with the URL represented as $D_h. ul$. When the access is granted, the cookie $v_k$ is retrieved from the browser’s cookie store $\mathcal{K}$. The label store is updated such that the security label of $v_k$ will become the same as the label of the cookie in $\mathcal{K}$.

Rule 10 shows the operational semantics of writeCki. The permission is determined if the composite label of the script on the call stack has write (w) permission for the entire document $D_h$ with the URL represented as $D_h. ul$. When the access is granted, the value $v$ is written into the browser’s cookie store $\mathcal{K}$, associated with the URL $D_h. ul$. The security label of the stored cookie becomes the label of the value $v$.

Rule 11 shows a function call. The rule specifies the list of arguments as expressions $E*_v$ and the values of them as $v*_v$. Then the composite security label $L_v$ is determined from the call stack and the arguments $v*_v$. If all the primitive labels in $L_v$ has execute (e) permission on the function object $f$, then the data label of the function $f$ is pushed to the call stack, and the script $S$ in the function body is evaluated under the updated call stack $C'$. The value returned from execution of the script $S$ in the function body is $v$.

Rule 12 shows the operational semantics of eval($E_v$), which executes an arbitrary string as a script. When the eval operation is invoked, the browser evaluates the argument $E_v$ into $v_e$, and then parses the argument into script, $S = parseScript(v_e)$. Then the composite security label $L_{v_e}$ is determined from the call stack and the argument $v_e$. The composite security label $L_{v_e}$ is determined from the label store on the value $v_e$. The script $S$ is executed only if all the primitive labels in $L_v$ have execute (e) permission on the value $v_e$. In that case, the call stack is first updated with the security label of $S$, and then the script $S$ is evaluated. The permission check on the argument $v_e$ prevents unauthorized string from being executed as a result of injection attack, such as case of the DOM-based attack.

Finally, Rule 13 shows the operational semantics of xhr($E_{url}, E_k$), which sends the data $E_k$ to the URL $E_{url}$ and receives a response. The primitive operation send represents the browser’s action of sending an HTTP request (See 4.4). Since xhr provides explicit bi-directional communication, access to a location $u_t$ by xhr requires both the write and read permissions.

Intuitively, the access permission of xhr is determined as follows:

1. Arguments $E_{url}$ and $E_k$ are evaluated to the URL $u_t$ and the value $v_k$, respectively.
2. The composition of the security labels in the call stack and the arguments is determined. This composite label is represented as $L_t$ in the rule.
3. The security label $L_t$ of the target object, which is the location $u_t$, is determined from the labeling policy $\mathcal{P}_t$. Intuitively, all primitive security labels in $L_t$ needs to have read and write permissions to $L_t$, which represents the security domain of the URL $u_t$. The operation is aborted if $L_t$ has no sufficient permissions.
4. The permission for automatically sending cookie ($c$) is determined from the access control policy. In other words, cookie is sent only when all primitive labels in $L_t$ have the cookie permission for $L_t$ (see the detailed algorithm below.) When $L_t$ has insufficient permissions, the HTTP request is sent without the cookie.

When an HTTP request is issued, the cookie permission for the target URL is determined, and the cookie is associated with the HTTP request only when the subject has the automatic cookie ($c$) permission. This restriction prevents the Cross-Site Request Forgery (CSRF) attacks, which details will be described in Sect. 6.

Definition 5 (Permission Check for Cookies):
allowedCookie($L_{url}, u_t = \mathcal{K} [u_t] \iff \forall l \in L_{url}, \exists p' = (s', o', +c) \land \forall l' \in L_{url}, \exists p''(s', o', -c), where\ s' = l \land o' = L_{url}$). Otherwise allowedCookie($L_{url}, u_t = null$).

4.4 Browser’s Primitive Operations

The browser’s primitive operations are defined in Fig. 4. The operation send($url, data, k$) represents the browser’s action of sending an HTTP request and receiving the response data $v_r$, with optional content policy $\mathcal{P}_r$ and cookie $k'$ in response (Rule 14). The send operation will update the policy store $\mathcal{P}$ and the cookie store $\mathcal{K}$ with $\mathcal{P}_r'$ and $k'$. The initial security label of the cookie for the URL, $\mathcal{K}[u_t]$, is equal to the label of $D_h$. The primitive operation load($u_t$) (Rule (15)) represents the browser’s behavior of navigating to a URL by following
a hyperlink or by the user’s action of entering a URL in the browser’s address bar. The load action may be initiated by other desktop applications that support hyperlinks. When loading an HTML document at the URL $ul$ into a browser window $w$, the browser issues an HTTP request to the URL.

When a page $D'$ is loaded as a result of a hyperlink in another webpage $D_h$, the label on the URL string in the content of $D_h$ will be inherited, and recorded as the security label of the document URL $D_h.ul$. The label on the document URL prevents DOM-based XSS, because the permission associated with the originator of a URL string will be inherited by the values derived from the URL. The returned data is parsed into DOM nodes, and each node is associated with the security labels $(\text{putlabel})$ as defined in the labeling policy $p^L$, and inserted into the document tree.

Rule (16) shows that the script $S$ in an $\text{on event}$ attribute is executed when an event occurs. The call stack is empty when an event handler is invoked, and thus only the $\text{execute (e)}$ permission of the attribute value is determined.

Rules between 17 and 21 show the browser’s $\text{addChild}$ behavior, which inserts a node into the DOM tree. The corresponding permissions are checked when the node insertion results in HTTP requests or script invocation.

When an inserted node includes only passive data (such as text or formatting elements), the DOM node is simply updated (Rule 17). When the node inserted by $\text{addChild}$ includes an $<\text{img}>$ element with a $\text{src}$ attribute which includes a location URL $ul$, that initiates an HTTP request to the URL $ul$ (Rule 18). The behavior is regarded as a one-way communication (given that the returned image is static) and thus requires the $\text{write}$ permission for the location $ul$.

If the inserted $<\text{img}>$ element has a JavaScript $uj$, then it requires the $\text{execute (e)}$ permission to execute the code (Rule 19). When the inserted node includes an embedded script, then the $\text{execute (e)}$ permission is needed to invoke the script (Rule 20). If the inserted script node has a URL reference in the src attribute, read, write, and execute permissions are required before the browser issues an HTTP request for the URL and execute the script (Rule 21).

5. Enforcement of Access Control

We prove the correctness of the model using two theorems: determination of the content origin and enforcement of access control on sensitive operations.

**Theorem 1 (Origin of Static and Dynamic content):** The origin of the static or dynamic content that are involved in an operation is determined from the effective labeling policy $p^L$ the direct information flows during script execution.

**Definition 6 (Content involved in an operation):** The content that is involved in an operation is the script (or functions) on the call stack and the sensitive parameters of the operation.

**Definition 7 (Static Content):** The static content is the HTML elements and the script code in $D_h$ and $D$.

**Lemma 1 (Origin of Static Content):** From Definition 1, 2 and 3 as well as rules (14) and (15), all data received as an HTTP response will be associated with the initial data label that represents the data origin, according to the policy in $p^L$. When no labeling policy is defined, the initial label of the data is $\bot$ from Definition 3.
From rules (1) and (5), the data label of the static content is the initial label. From rules (2), (4), and (3) the data labels are propagated when direct information flows occur within a function. From rule (11), labels are propagated as data propagates through function argument and return values, and the call stack is updated as a function is called. From rules (7), (8), (9), and (10), the label of the data is propagated as data is stored or retrieved from the DOM or cookie store, and thus origin of the can be tracked.

Definition 8 (Dynamic Content): Dynamic content is HTML elements or script code that is executed as a result of the DOM node update or the eval function.

Proof of Theorem 1 (Origin of Static and Dynamic Content): In addition to Lemma 1, the origin of the eval argument is determined as shown in rule (12). From rule (18), the origin of the inserted DOM node is determined from the origin of the original values.

Theorem 2 (Enforcement of AC policy): All of the sensitive operations, i.e., network access, read or write operations to the DOM and cookies, and the execution of script, are controlled by the effective policy set, based on the origin of the static or dynamic content that involve in the operation.

Definition 9 (Sensitive Operations): The sensitive operations are xhr, readCki, writeCki, readDOM, writeDOM, eval, any script execution, any network access caused by URL references in HTML, and automatic cookie attachment in HTTP requests. (However, page loading by a user is not a sensitive operation.)

Lemma 2 (AC Enforcement of Static Content): From Rule 13-10, permission check by Definition 4 is performed before carrying out the operations in static content. Rules (18), (20), and (21) show that network access by URL references in the HTML will have permission checks by Definition 4. Rules (13), (18), and (21) show that cookie permission is checked by Definition 5.

Proof of Theorem 2 (AC Enforcement of Static and Dynamic Content): Dynamic content execution by eval will have a permission check as shown in rule 12. The writeDOM operation (rule 8) will cause DOM update, that result in on of the addChild rules, i.e., (17), (18), (20), or (21). Dynamic content execution as a result of DOM updates are modeled in rules (18), (19), (20) and (21), and thus subject to the permission checks of Definitions 4 and 5. From Lemma 2s, Definition 4 and 5, and Theorem 1 all sensitive operations, either static or dynamic, are subject to access control by the effective policy set.

6. Examples

This section shows how the proposed browser security model mitigate risks of distrusted third-party services in a mashup application. In addition, we illustrate that the proposed security model can prevent other common Web application threats such as the Cross-Site Request Forgery (CSRF), and local network attacks.

6.1 Mashup Application Example

The example environment in Fig. 5 consists of a mashup application a.com, which integrates an external service from e.com. The user’s browser is in the intranet which includes an intranet server u.com, which may become a target of local network attacks. Figure 6(1) shows the content of the mashup application at a.com/a.html which includes the external content from e.com in two ways: 1) some content from e.com is integrated into the web page generated by a.com, in the form of server-side mashup, and 2) a script file at e.com/e.js (Fig. 6(2)) is imported into a.html by a script tag. We assume that a.com is trusted by the user for the service it provides, while e.com may potentially be a malicious third party. Both a.com and e.com use a trusted JavaScript widget library from g.com.

Now we assume that the environment consists of three sets of policies from stakeholders: 1) the user-defined policy, whose purpose is to protect the user and the intranet applications from attacks, 2) the content policy by a.com, in the form of server-side mashup, and 2) a script file at e.com/e.js (Fig. 6(2)) is imported into a.html by a script tag. We assume that a.com is trusted by the user for the service it provides, while e.com may potentially be a malicious third party. At the same time, e.com does not fully trust a.com either. Both a.com and e.com use a trusted JavaScript widget library from g.com.

Now we assume that the environment consists of three sets of policies from stakeholders: 1) the user-defined policy, whose purpose is to protect the user and the intranet applications from attacks, 2) the content policy by a.com, whose purpose is to protect its content from potentially ma-
licious external services (such as e.com), and 3) the content policy by e.com whose purpose is to protect itself from potentially malicious mashup applications (such as a.com). Since g.com is a generic JavaScript library, it does not claim any policies.

Figure 7 shows the set of example policies defined based on the policy grammar of Table 2. Note that \( \mathcal{P}_{D_0}^L \) and \( \mathcal{P}_{D_0}^A \) are the labeling and access control policies defined for the document \( D_0 \), while \( \mathcal{P}_{D_0}^L \) and \( \mathcal{P}_{D_0}^A \) are defined for \( D_e \). (The numeric subscript of the policy element is the index in each set of policies.) The user policy 1 defines the label \( U_u \) for the intranet applications in u.com. The user policy 2 disallows all Web applications, except for those labeled with \( U_u \), to access the intranet applications on u.com, in order to prevent local network attacks. The user policy permits all interactions between other contents that do not originate from u.com. Policy 3 is the labeling policy defined by a.com. The content from a.com is labeled as \( A_a \) (Note that the suffix \( a \) denotes that the label is defined by a.com). However, the content under the particular elements, which are expressed in the XPath \( /[@class="e"] \), are labeled as \( AE_e \). The policy also defines that the content from e.com and g.com are labeled as \( E_e \) and \( G_e \) respectively. Policy 4 is the access control policy defined by a.com, which gives the content with the label \( A_a \) full access permissions to the content with the same label \( A_a \). The policy also allows \( A_a \) to execute content with the label \( E_e \) and \( G_e \), implying that \( A_a \) may invoke services provided by \( E_e \) and \( G_e \) in the form of function calls. Permission of \( E_e \) is limited to the node with the label \( AE_e \). The library \( G_e \) is allowed read and write permission for both \( A_a \) and \( AE_e \). Policy 5 is the labeling policy defined by e.com, which defines e.com, a.com and g.com as \( E_e \), \( A_e \) and \( G_e \) respectively. Policy 6 is the access control policy defined by e.com, which gives \( E_e \) full access permission for itself, and \( A_e \) to read and write access to \( E_e \). The execute permission allows \( A_e \) to call functions in e.com.

Now, let’s see how permissions are granted for normal operations.

1. e.com/e.js defines a function updateMap which takes a street address, queries the map data from the server e.com, and update a DOM node <div id="e"> with the map data. The data label of the updateMap function object is \( \{E_e, E_e\} \), because it inherits the container label on e.com/e.js. Now, suppose a.com/a.html calls updateMap with a string literal, (Fig. 6 (1) Line 5). The updateMap function first

1) retrieve data from e.com with XHR, and then 2) write the data into the DOM node (Fig. 6 (2) Line 2-3). The data label of the address string is \( L_1 = \{A_a, A_e\} \) because it is originated from the content a.html. The composite data label of the call stack will be \( L_2 = \{A_a, A_e, E_e, E_e\} \) because the call stack include content from both a.html and e.js.

a. When updateMap calls xhr, the subject of the operation is identified by the set of labels \( L_s = L_1 \sqcup L_2 = \{A_a, A_e, E_e, E_e\} \). The object of the operation is \( \{E_e, E_e\} \), the container label on e.com. The stakeholder is e.com, and thus its policy 6 is respected. Finally, the operation is granted since policy 6 allows both \( A_e \) and \( E_e \) to have read and write access to \( E_e \). Assuming that returned data does not have explicit policy associated with it, it will have the same label as e.com/e.js as implicit label (i.e., the label of the map data will be \( \{E_e, E_e\}\)).

b. Likewise, when updateMap calls writeDOM, the subject of the operation is identified by \( L_s = \{A_a, A_e, E_e, E_e\} \). The target of the operation is the DOM node <div id="e">, and identified by its container label \( L_t = \{AE_e, A_e\} \). In this case, a.com/a.html is the stakeholder of the operation because it owns the DOM tree, and thus its policy 4 is respected. The writeDOM operation will be granted because policy 4 allows both \( A_a \) and \( E_e \) write access to \( AE_e \).

2. The content a.html can leverage the widget library G, to modify the DOM tree (policy 4-4, 4-5).

3. The content from u.com has full access to itself (policy 1-1, 2-1).

Now, assume that e.js tries to alter the behavior of a.html by overriding some variables (e1, e2, e3) used by a.html.

1. When a.html concatenate e1 with some string literal and calls eval on it, both of the subject label are \( L_s = L_0 = \{A_a, A_e, E_e, E_e\} \) (Note that the object label in this case is the data label on the parameter of eval, which is the concatenation of string). The stakeholder is a.html and thus its policy 4 is respected. However, the script will not be executed since policy 4 does not imply neither \( \{A_a, E_e, +e\} \) nor \( \{E_e, A_a, +e\} \).

2. When a.html tries to write e2 into a DOM node, it consists of two steps.

a. First, in writeDOM operation, the data label of the subject is \( L_s = \{A_a\} \) and the container label of the target DOM node is \( L_t = \{E_e, E_e\} \). The data label of the second parameter (which does not effect the subject label) is \( L_s = \{E_e, E_e\} \). The stakeholder of the operation is a.html. The write operation will be granted since \( A_a \) has write permission for \( A_a \).
b. And then, the addChild operation (Rule 20 in Fig. 4) will be invoked when the new node is inserted, because the node being inserted is a script node with embedded script. In this case, the subject label is the composition of the labels on the call stack and the parameter, i.e., $L_s = \{A_a, A_e, E_a, E_e\}$. The target of the operation is the data label on the new node, i.e., $L_d = \{E_a, E_e\}$. Again, a.html is the stakeholder, and according to its policy 4, the node will be added but the included script will not be executed due to the lack of permission; i.e., $A_a$ does not have an execute permission on $E_d$.

Now, let's see how other attacks are prevented.

1. If the application a.com/a.html has a DOM-based XSS vulnerability, the URL string will be assigned the label $\top$, which is associated with no permissions, and thus the injected script will not be executed, as we have described in Sect. 2.

2. e.com/e.js may be a malicious content which tries to attack a.com/a.html. But e.com/e.js cannot read or write the DOM tree of a.com/a.html.

3. e.com/e.js may modify or read the node under <div id='e'> by using the trusted library from g.com. However, g.com's library cannot access other nodes in a.com/a.html if it is invoked by e.com/e.js, or vice versa, due to the stack inspection.

An attacker in e.com or a.com may try to circumvent the access control by overriding data or content as follows:

1. (Object overriding) e.com may override JavaScript object properties to alter the behavior of a.html. In the proposed security model, JavaScript objects are not the target of access control, and thus e.com may override them. However, the object properties written by e.com will have the label $\{E_a, E_e\}$ associated with it. Therefore, even if script in a.html is going to invoke some operation with the overridden object, the subject of the operation will be $\{A_a, A_e\} \cup \{E_a, E_e\}$; i.e., the sensitive operation that involves the overridden object will require permission of both a.com and e.com.

2. (Fake labels) e.com may try to override the labeling policy to allow its access to a.com. E.g., by adding (A, e.com/) into the policy set 5. However, in the proposed model, label names defined by different content policy are distinguished. The labels are aliases for the URLs, and are translated into URL when it is evaluated in the corresponding access control policy.

3. (Fake policies) e.com may try to override the access control policies to allow its access to a.html. E.g., by adding $\{E_a, A_e, (+r, +w, +c)\}$ into the policy set 6. However, when e.com tries to access a DOM node of a.html, a.com will be the stakeholder who's policy will be observed. Since a.com does not allow $\{E_a, A_e, (+r, +w, +c)\}$, the operation will not be granted.

6.2 Cross-Site Request Forgery (CSRF)

CSRF misuses cookies and HTTP authorization to issue illegal commands by bypassing the request authentication. Many applications track the user’s authorized state by using cookies. The server returns the cookie as an authorization token, and the browser attaches the cookie in the next HTTP request to show that the request is coming from an authorized user. However, since the browser automatically attaches the cookie to the HTTP requests for the target server, the cookie will be attached to all requests, including those issued by malicious attacker’s content. CSRF allows an attacker to issue illegal commands to innocent servers by “riding” on the existing authorized session. Other browser-managed authorization mechanisms such as the HTTP authorization header and client-side SSL authentication are also vulnerable to CSRF.

Unlike XSS, CSRF does not require a malicious script to be injected into the content on an innocent server. CSRF can not only compromise the authenticity of the commands sent to the server, but also be used to compromise the confidentiality of information protected by authentication; when an HTTP response returns content in the JSON format [16], an attacker may steal information by overriding the object constructors [17].

There are two typical countermeasures for CSRF. First, the Web server can check the Referrer header of the HTTP request, which should indicate the URL of the content originating the request. However, some browsers allow disabling the Referrer header in order to protect users’ privacy, and thus this is not an always-reliable method. Second, an application can authenticate an HTTP request by using a secondary security token in the request parameter, in addition to the cookie. The secondary token insures that the request is originating from a genuine part of the Web application, which includes the token in the content (e.g., in a hidden input field). However, many Web applications do not implement these countermeasures correctly, and thus it is important that users can protect themselves from such vulnerable applications.

6.2.1 Our Approach for Preventing CSRF

In order to prevent CSRF, we introduce the notion of automatic cookie (c) permission in our browser model. For example, in the example environment of Fig. 5, the server a.com is a potential target of CSRF attacks by e.com. Therefore a.com’s policy defines that the automatic cookie (c) permission for the server a.com is given only to the content with the security label $A_a$.

Conversely, a.com may maliciously try to carry out a CSRF attack against e.com with the JavaScript code as follows.

```
document.imgs[0].src= 'http://e.com/x?cmd=doSomething'
```

This code will trigger an HTTP request to e.com with
the given URL. In the current browser, the request is also associated with the document cookie from e.com. However, in our proposed model, this JavaScript code is part of the content downloaded from a.com, and associated with the data label $A_e$ in the e.com’s policy (policy 5-2). Since the content with the data label $\top$ does not have the cookie (c) permission in the access control policy of e.com (policy 6), the cookie of e.com will not be sent along with the HTTP request.

6.3 Local Network Attacks

In browser-based port scanning attacks, malicious JavaScript attempts to access local IP addresses in the user’s network. By detecting the presence of default icon images of specific Web servers or from the type of errors (e.g., HTTP 404 Not Found or connection errors), the attacker can detect the presence of hosts and even determine the types of the Web servers. An attacker can also send commands to networked devices or services. For example, an attacker may use JavaScript to send arbitrary print commands to a networked printer [18], or to configure a victim’s home broadband router to change its DNS settings [19].

Since JavaScript is executed on the user’s browser after being transferred through an ordinary HTTP port, an attacker can easily attack the user’s local network behind a firewall. An attacker may also combine these techniques, e.g., finding potential victim servers by port scanning, and then send illegal commands to them, while leveraging CSRF to bypass the user authorization.

6.3.1 Our Approach for Preventing Local Network Attacks

Figure 8 shows an example of JavaScript-based port scanning. In this example, the JavaScript code downloaded from e.com tries to access the given port of the server by dynamically modifying the src attribute of an <img> element, and determines the presence of the host from the presence of images and the type of errors.

This attack cannot be prevented by the same-origin policy, because it only controls the network access by the XMLHttpRequest objects.

```javascript
function portScan(callback, host, port) {
    var timeout = 100;
    var img = new Image();
    img.onload = function () {
        callback(host, port, 'open');
    }; img.onerror = img.onload;
    setTimeout(function () {
        callback(host, port, 'closed');
    }, timeout);
    img.src = 'http://' + host + ':' + port;
}
```

![Port scanning](image)

In our proposed model, all network accesses, including those that use HTML elements and attributes are also the target of the access control. For example, in the example environment in Fig. 5, the user’s access control policy is defined such that only content from u.com can access URLs on u.com.

For example, assume that e.com/e.js includes script shown in Fig. 8. The script will be associated with the data label $\{\top, E_u, E_e\}$; since the user’s policy define no labels for e.com, it will be $\top_e$ in the context of the user defined policy. Since the content with the data label $\top_e$ does not have any permission in the access control policy 2, the browser does not issue HTTP requests to u.com, because of the lack of the access permission of.

7. Discussion

We chose XPath to identify parts of document that would include content with different trust levels, and to associate the access control policy with the subdocument. However, because of the flexibility of HTML and the client-side JavaScript, XPath is not an ideal mechanism for specifying the subdocument.

For example, a mashup server may try to confine the content from a third-party e.com within a <div> element, e.g., `<div id="e">content from e.com</div>`. However, e.com can change the document structure by inserting a fake close tag. For example, if the content from e.com includes a close tag `</div>`, then the label e will not be associated with content after this close tag. Such problem can be avoided by good programming practice, i.e., to insert third-party content using JavaScript and DOM API.

Likewise, document.write(s) can change the document structure when string s includes a close tag. This happens because the current browsers use the early evaluation approach, i.e., document.write() is executed while the browser parses and renders the main HTML document. Such behavior has significant impact in the secure browser architecture, since it allows an attacker to circumvent the access control policy. In order to enforce the labeling policy on the inserted content, we changed the behavior of document.write to late evaluation, i.e., the content will be inserted after the DOM of the main document is constructed, and thus does not change the document structure.

The paper did not discuss how to attach policies to the contents, but we assume that the content policy is sent on the same HTTP response as the content itself, most likely in the form of an HTTP header. It is important to make sure that the policy cannot be modified by the client-side JavaScript.

Several technical challenges need to be solved for the proposed model to be implemented. First, the JavaScript engine needs to be extended to support fine-grained dynamic information flow tracking. Such an extension might easily introduce a large performance overhead, and thus an effective implementation is necessary to insure the browser remains usable in real-life applications. Second, as mentioned above, a policy attachment mechanism has to be designed
and implemented. Third, the Web browser has to be modified so that every access to resources will be mediated by a reference monitor that enforces the access control policy based on the proposed security model.

8. Related Work

Due to the prevalence of threats against Web 2.0 applications, a number of proposals have been addressing the problems in the browser security model. Most of the work for secure mashup proposes confining third-party contents into sandboxes (or components) and providing with controlled cross-domain communication between components. Subspace [5] and SMash [6] achieve this goal with the current browsers, by taking advantage of the separation by iframe and browser’s cross-domain communication techniques. MashupOS [20] and the module tag [21] propose extensions of the browser architecture to enable explicit sandboxing and cross-domain communication. IE8 will support explicit cross-domain communication with extended browser APIs [22]. However, these proposals do not address the issue of XSS, CSRF and local network attacks (Sect. 2), because the content in the each sandbox is provided with capabilities similar to the current browser model, and there is no control over the browser’s networking capability using HTML attributes. In particular DOM-based XSS is difficult to prevent, due to browser’s lack of knowledge about the origin of the injected data.

Yu et al. [23] defined a formal model of browser and proved soundness for code instrumentation. Tateishi and Tabuchi modeled the information flow in a browser from confidentiality point of view [24]. Vogt et al. [25] propose an information-flow based mechanism to detect XSS attacks in which secret information is sent to a remote attacker. However, these approaches do not propose generic access control models for browsers. Seki [26] proposed DOM-based access control, but did not address the confused-deputy problem discussed in Sect. 3. The design of JavaScript has been seen as a problem and many improvements, such as namespace separation and private scopes, are proposed in ECMAScript4 [27]. Caja [28] is a safe subset of JavaScript that supports object-oriented capabilities. The improved versions of JavaScript do not address browser security issues, although they will advance the secure browser model with their language support for encapsulation and namespace separation.

There are several prior work propose finer granular mechanism of binding the domain information to each page, which summary is given in [29]. In this paper we chose a simple approach of distinguishing finer grained domains by URL path prefix, but it is possible to adopt other approaches, such as the server-side certificate, with some extension to the labeling policy in the proposed model.

9. Conclusion and Future Work

This paper proposes a novel Web browser security model that is built on the information-flow based access control in the browser environment. The proposed model overcomes the dynamic nature of JavaScript to allow precise access control based on the origin of the code. The policy model addresses various threats to the Web applications that we are suffering today.

The current model omitted support of implicit flows because it has little effect when assuming no conspiracy between trusted and untrusted content. Support of implicit flows will require further research due to the dynamic nature of JavaScript and the need for effective un-tainting or re-labeling. In this paper, various details of the browser behavior (e.g., SSL) have been omitted to keep the model simple. In addition, support of privilege overriding in a trusted code (similar to doPrivileged in Java) will also be needed in real-life applications. An effective and performable implementation is another topic that requires further research.

Acknowledgement

The authors wish to thank their colleagues at IBM Research and the Matsumoto-Lab in the Yokohama National University for their feedback. In particular, we wish to thank Michael Steiner and Marco Pistoia for their insightful comments and discussions.

References


Sachiko Yoshihama is an advisory researcher at Tokyo Research Laboratory of IBM, Japan. She received her M.E. in information security. Her research interests include trusted computing, information flow control, and Web security.

Takaaki Tateishi is a research staff member at IBM Tokyo Research Laboratory in Japan. He received his Ph.D. at Japan Advanced Institute of Science and Technology (JAIST) in 2003.

Naoshi Tabuchi is a researcher of IBM Research Tokyo, Japan. He received his master degree at The University of Tokyo in 2003. His major research interests are theories and principles of programming languages, in particular type-based program verification and analyses of modern programming languages such as Java, JavaScript or PHP.

Tsutomu Matsumoto is a Professor and the head of Division of Social Environment and Informatics, Graduate School of Environment and Information Sciences, Yokohama National University. His current roles include the Advisor, Research Center for Information Security, National Institute of Advanced Industrial Science and Technology, a Director of International Association for Cryptologic Research and a core member of the Cryptography Research and Evaluation Committees for governmental use of cryptographic technology. Starting from design and analysis of various cryptosystems and cryptographic protocols in the early 80’s, he has opened up the field of security measuring for logical and physical security mechanisms including human-machine cryptography, information hiding, software security, biometric security, side-channel security, and artifacts. He received the Achievement Award from IEICE in 1995 and the 2006 DoCoMo Mobile Science Award from the Mobile Communication Fund.