The Emperor's New APIs: On the (In)Secure Usage of New Client-side Primitives

Steve Hanna[§], Eui Chul Richard Shin[‡], Devdatta Akhawe[§], Arman Boehm[‡], Prateek Saxena[§], Dawn Song[§]
{*sch, ricshin, devdatta, boehm, prateeks, dawnsong*}
[§]@eecs.berkeley.edu
[‡]@berkeley.edu
University of California, Berkeley

***Abstract*—Several new browser primitives have been proposed to meet the demands of application interactivity while enabling security. To investigate whether applications consistently use these primitives safely in practice, we study the real-world usage of two client-side primitives, namely `postMessage` and HTML5's client-side database storage. We examine new purely client-side communication protocols layered on `postMessage` (Facebook Connect and Google Friend Connect) and several real-world web applications (including Gmail, Buzz, Maps and others) which use client-side storage abstractions. We find that, in practice, these abstractions are used insecurely, which leads to severe vulnerabilities and can increase the attack surface for web applications in unexpected ways. We conclude the paper by offering insights into why these abstractions can potentially be hard to use safely, and propose the *economy of liabilities* principle for designing future abstractions. The principle recommends that a good design for a primitive should minimize the liability that the user undertakes to ensure application security.**

I. Introduction

With the growing demand for interactivity from Web 2.0 applications, web application logic is significantly shifting from the server to the browser. This need to support complex client-side logic and cross-domain interaction has led to a proliferation of new client-side abstractions, such as the proposals in HTML5. A number of major web application providers (including Google and Facebook) have responded by offloading several security-critical parts of their functionality to the client.

However, due to the nascence of these primitives, the security implications of using these new client-side abstractions on the web application's overall security have received little evaluation thus far. To investigate this issue, we selected two primitives as case studies representative of the class of emerging client-side constructs. First, we study systems using `postMessage`, a primitive that enables cross-origin communication within the web browser. Specifically, we analyzed two new purely client-side protocols, namely Google Friend Connect and Facebook Connect, which are layered on `postMessage`. As a second case study, we analyze the usage of client-side storage primitives (such as HTML5 `localStorage`, `webDatabase` API and database storage in Google Gears) by popular applications such as Gmail, Google Docs, Google Buzz and so on.

The `postMessage` API is a message passing mechanism that can be used for secure communication of primitive strings between browser windows. However, if developers do not use the security features of the primitive fully or implicitly trust data arriving on this channel, a variety of attacks can result. We aim to study how consistently this API is used *securely* in practice, by analyzing two prominent client-side protocols using `postMessage`, namely Facebook Connect and Google Friend Connect. To systematically evaluate the security of these protocols, we first reverse engineer the protocol mechanics/semantics as their designs were not documented. In our evaluation, we find that both protocol implementations use the `postMessage` primitive unsafely, opening the protocol to severe confidentiality and integrity attacks. Worse, we observed that several sites using this protocol further widen their attack surface—in one we were able to achieve arbitrary code injection . We were able to concretely demonstrate proof-of-concept exploits that allow unauthorized web sites to compromise users protocol sessions, which can lead to stealing of users data or even injection of arbitrary code into benign web sites using Facebook Connect and Google Friend Connect protocols. In our evaluation, we also observed a strange inconsistency—developers, belonging to the same organization and sometimes of the same application, used the primitives safely in some places while using them unsafely in others. The vulnerabilities in communication primitives have been alluded to in research literature [3], [11]. However, these new client-side protocols have not been studied previously and we are first to demonstrate the practicality and severity of these vulnerabilities in the context of real-world client-side communication protocols.

As a second representative of a purely client-side abstraction, we study client-side data storage primitives and various applications that rely on these. We find that a large fraction (7 out of 11) of the web applications, including Google Buzz, Gmail and Google Maps, place excessive trust on data in client-side storage. As a result of this reliance, transient attacks (such as a cross-site scripting vulnerability) can persist across sessions (even up to months), while remaining invisible to the web server [5], [13]. In our results, as in the case of the `postMessage` study, we observed a similar inconsistency in developer's sanitization of the dangerous data. Our results show that despite some prior knowledge of the storage vulnerabilities [13], in practice, applications find it difficult to sanitize dangerous data at all places.

We observe a common problem with these new client-side primitives: to ensure security, every use of the primitive needs to be accompanied by custom sanity checks. This leads to repeated effort of developing sanity checks by each application that uses the primitive. And, often even

within one application similar checks may be distributed throughout the application code, a practice which is prone to errors. We propose the *economy of liabilities* principle in designing security primitives—a primitive must minimize the liability that the user undertakes to ensure application security. For example, in this context, the principle of economy of liabilities implies that client-side primitives should internally perform sanitization functionality critical to achieve the intended security property, as much as possible. New primitives today ignore this design principle, achieving security only 'in principle' rather than 'in practice'[1]. We hope the economy of liabilities principle will guide the designs of future primitives.

Retrofitting the economy of liabilities principle to the existing primitive designs is challenging as they have been adopted by real-world applications already. Furthermore, the exact sanitization policies vary significantly across applications. However, we suggest enhancements to these primitives which we believe achieves a reasonable compromise between security and compatibility. In particular, we suggest a declarative style, whitelist-based origin validation scheme that should be provided by the `postMessage` primitive and enforced by the browser to ensure channel integrity. For client-side database primitives, we suggest the browser database interface should automatically perform output sanitization to prevent persistent XSS attacks. We hope that these suggestions kick start discussion in the web community on refinements to reduce developer burden.

Summary of Contributions.

- We systematically examine two representatives of new client-side primitives which are in popular use by real-world applications: (a) `postMessage`, a cross-domain message passing API, and (b) persistent client-side database storage (HTML5 `localStorage`, `webDatabase` APIs and database storage in Google Gears).
- We present the first step towards understanding purely client-side protocols, by reverse engineering them directly from their implementation in JavaScript and formalizing them. We systematically extract the sanity checks that applications implement on the security-relevant data and use these to find new vulnerabilities in our target applications.
- We provide practical evidence of the pervasiveness of these new attacks on several important web application protocols (Facebook Connect and Google Friend Connect) and web applications (Gmail, Google Buzz, Google Docs and others).
- To eliminate the inconsistency we observe in safe usage of these client-side primitives, we propose the guiding principle of *economy of liabilities* and suggest remedies based on this principle to make the primitives more practical for safe use with the aim of garnering discussion and obtaining community feedback.

[1]giving the "Emperor" a false impression of his shiny new clothes

II. Attacks on Client-side Messaging

The `postMessage` API is a client-side primitive to enable cross-origin communication at the browser side. Originally introduced in HTML5, `postMessage` aims to provide a simple, purely client-side cross-origin channel for exchanging primitive strings [15]. Web browsers typically prevent documents with different origins from affecting each other [12]. A mashup specifically aims to overcome this restriction and communicate with another web site in order to provide a richer experience to the user. Barth et al. [2] study various client-side cross-origin communication channels and recommend the `postMessage` mechanism, due to the security guarantees(detailed below) it is able to provide.

The `postMessage` primitive aims to provide the dual guarantees of authenticity and confidentiality. Messages can be sent to another window by invoking the window's `postMessage` method. Note that this message exchange happens completely over the client side and no data is sent over the network. The security guarantees are achieved as follows:

- *Confidentiality*: The sender can specify the intended recipient's origin in the `postMessage` method call. The browser guarantees that the actual recipient's origin matches the origin given in the `postMessage` call, and code executing in any other origin's context is unable to see the message. The intended recipient's origin, specified in the method call, is called the `targetOrigin` parameter. For use cases in which confidentiality is not essential, a sender can specify the all-permissive '*' literal as the `targetOrigin`.
- *Authenticity*: The browser attributes each received message with the origin of the sender, as the `origin` property of the message event. The recipient is expected to validate the sender's origin as coming from a trusted source, thus achieving sender authenticity.

Note that if these checks are missed by the application, the browser does not guarantee anything about the security of the `postMessage` channel. For instance, a malicious website could send arbitrary messages to a benign website, and it is the latter's responsibility to ensure that it only processes messages from trusted senders. To avoid the aforementioned problems, the HTML5 proposal recommends websites to set the `targetOrigin` parameter for any confidential message and to always check the `origin` parameter on receiving a message.

Attacking postMessage Applications. We investigate two prominent users of the `postMessage` primitive, the Facebook Connect protocol and the Google Friend Connect protocol. We conjecture that for complex cross-domain interactions involving fine-grained origins, developers may fail to follow the recommended practice. In such a case, the channel would not provide a security property that the developer might have come to expect. Due to the complexity of the JavaScript code used by these protocols, we use the Kudzu [10] system to check for the absence of such validation in the code. We find that large parts of the protocols are undocumented, and we reverse engineer

these protocols based on the interactions we observe.

Scope of Attack. The threat model for our attacks on `postMessage` usage is the web attacker threat model [3]. In particular, we constrain the attacker to only controlling content on his own site. A user can visit the attacker's site, but may not necessarily trust content from it. Phishing attacks are outside the scope of this work. Bugs in browser implementations are also beyond the scope of this attack. An attacker can assume the user to have already logged onto Facebook and authorized Facebook Connect applications not controlled by the attacker.

Summary of Findings. We find various inconsistencies in the use of `postMessage`. Developers use these primitives correctly in some cases, while making mistaken assumptions in others. We demonstrate vulnerabilities in both Facebook Connect and Google Friend Connect protocols. In the following sections, we explain these two protocols in detail, point out vulnerabilities and demonstrate concrete attacks. We end our analysis of the `postMessage` primitive with a discussion of the observed real world usage of the `postMessage` primitive.

A. The Facebook Connect protocol

Facebook Connect is a system that enables a Facebook user to share his identity with third-party sites. Some notable users include TechCrunch, Huffington Post, ABC and Netflix. After being authorized by a user, a third party web site can query Facebook for the user's information and use it to provide a richer experience that leverages the user's social connections. For example, a logged-in user can view his Facebook friends who also use the third-party web site, and interact with them directly there. Note that the site now contains content from multiple principals—the site itself and `facebook.com`.

Mechanism. The same-origin policy does not allow a third-party site (e.g TechCrunch), called `implementor` in the paper, to communicate directly with `facebook.com`. To support this interaction, Facebook provides a JavaScript library for sites implementing Facebook Connect. This library creates two hidden iframes with an origin of `facebook.com` which in turn communicate with Facebook. The cross-origin communication between hidden iframes and the implementor's window are layered over `postMessage`[2].

Figure 1 details the protocol. The first iframe created by the library is used for the initial session negotiation with Facebook and the other is used for all subsequent data exchanges between the Facebook server and its client-side counterpart. More specifically, the first iframe (`loginFrame`, top middle in Fig 1) receives a secret key (K) and a session ID (S) from `facebook.com` and sends it to `implementor` (message 3). The second iframe (`proxyFrame`, bottom middle in Fig 1) also running in `facebook.com`'s origin, acts as a proxy for requests. Any query for data that `implementor` wants to make to `facebook.com` is first sent to `proxyFrame` (message 6), which then makes the request to `facebook.com` using

[2]In older browsers, other techniques are used which we do not discuss.

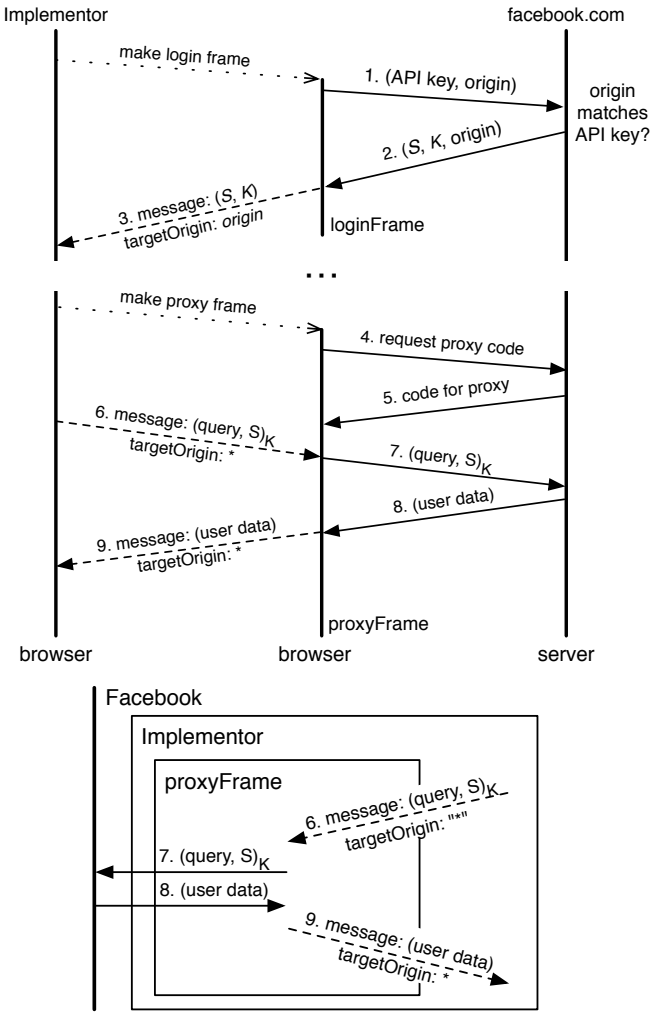


Figure 1: The Facebook Connect protocol. (top) Messages exchanged in the protocol. The dashed arrows represent client-side communication via `postMessage` and the solid arrows represent communication over HTTP. $(\texttt{query,S})_K$ represents a HMAC using the secret K. (bottom) Frame hierarchy for the Facebook Connect protocol. In this example, the `proxyFrame` is inside the main `implementor` window.

`XMLHttpRequest` (message 7) and then sends the response (message 8) back to `implementor` (message 9). At the end of this transaction, the user has essentially logged in to `implementor` using his Facebook credentials.

B. Vulnerabilities in Facebook Connect

Observation 1: During our testing, we noticed that the `origin` of received messages was sporadically verified. In particular, out of all of the messages exchanged, only about half were accompanied with an origin check in the receiver's code. Further investigation revealed that communication between `proxyFrame` and the implementor (message 6 and 9), neither participant checked the origin of received messages.

Additionally, we also noticed that the message 6 and 9 had the `targetOrigin` parameter set to the '*' literal, while in message 3, the `targetOrigin` parameter was correctly set. We also observe that a query for data is

authenticated by an HMAC with the shared secret K. This serves as a signature for every query (message 6) that the `proxyFrame` receives.

Attack on message integrity. As discussed before, validating the origin of received messages is necessary for ensuring sender authenticity. Based on *Observation 1*, a malicious attacker can inject arbitrary data in the communication between `proxyFrame` and `implementor`. In this particular case, we find that the data received over the channel is used in a code evaluation construct and thus allows an attacker to inject arbitrary code into `implementor`'s security context.

The attack is illustrated in Figure 2. In particular, an attacker replaces `proxyFrame` with a malicious iframe that he controls. By sending a malicious message in place of message 9, an attacker can inject a script into the implementor's security context. In the actual attack, the attacker has to include the `implementor` page in a iframe on a page controlled by him (see bottom of Figure 2). This gives the attacker the power to replace the benign Facebook `proxyFrame` with his own malicious `proxyFrame`. This attack is possible because on receiving message 9, the implementor does not validate the origin of the message sender, and thus processes a message from the attacker. The shared secret only provides authenticity of the query (message 6) and not for the response (message 9).

On our test site, we were able to inject a script payload into the benign implementor's security context[3]. We have also confirmed this attack on Facebook's reference implementation of a Facebook Connect site. As the Facebook Connect functionality is provided as a drop-in JavaScript library, we believe most real-world websites directly using this library are also vulnerable.

Attack on confidentiality. *Observation 2*: Setting the `targetOrigin` parameter to the '*' literal leaks sensitive user data like profile information and friend lists to the attacker. This data can then be used by the attacker to gain the real-world identity of a visitor to his website.

The attack is illustrated in Figure 3. Message 9 and Message 6 have the `targetOrigin` set to '*'. Based on *Observation 2*, this allows a malicious attacker to easily launch a man-in-the-middle attack against the communication between the `implementor` and the `proxyFrame` (message 6a in Fig 3). The fact that `implementor` does not validate the sender of messages (of message 9a in Fig 3, in particular) enables a complete man-in-the-middle attack, while the signature on the query provides no protection. The main attack occurs at message 9 (Fig 3), which consists of sensitive user data and is read by the attacker. In the actual attack, the attacker again includes the benign `implementor` page in an iframe and then replaces the `proxyFrame` with his man-in-the-middle frame, which in turn includes the real `proxyFrame` (bottom of Fig 3).

[3]We had previously discovered a similar flow of data to a critical code evaluation construct, which was fixed by Facebook by adding data sanitization routines [10]. This is not a scalable fix.

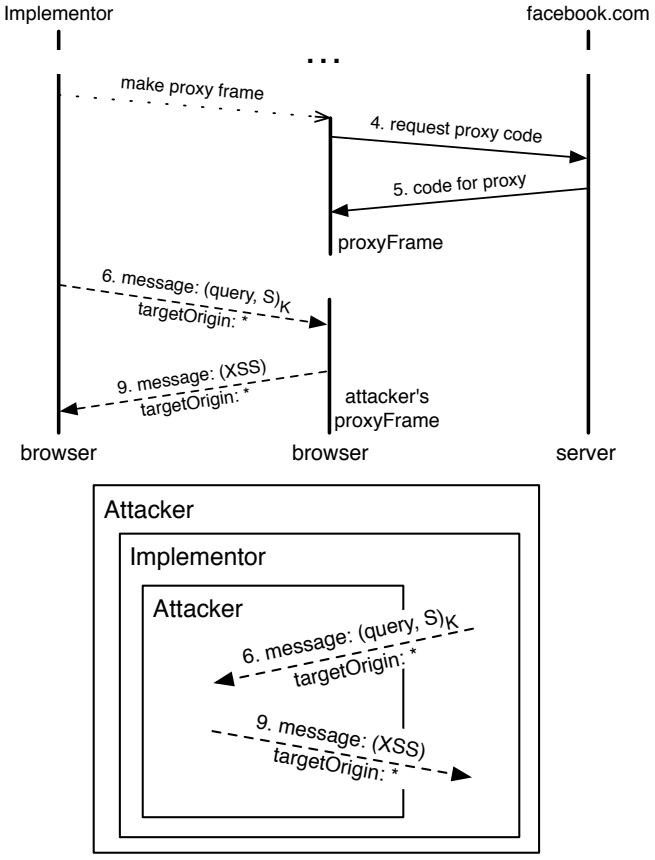


Figure 2: Integrity attack on Facebook Connect. (top) Messages exchanged in the protocol. Note that midway through the protocol (after message 5), the request proxy is replaced by an attacker-controlled proxy. (bottom) Frame hierarchy for the integrity attack. The topmost frame is owned by the attacker.

C. The Google Friend Connect protocol

Google Friend Connect is a system that provides similar functionality to Facebook Connect. An important difference is that Google Friend Connect allows a user to use multiple identity providers (like Yahoo!, Twitter, or Google) while signing onto various third-party sites. The aim, again, is to enable a richer social experience for users.

Mechanism. Typically, Google Friend Connect applications embed 'gadgets' inside iframes, which directly communicate with the relevant server. These gadgets communicate with the integrating page, referred to as `implementor` in the paper, via `postMessage` for parameters like colors, fonts and layouts. Like Facebook Connect, third-party websites interested in integrating Google Friend Connect in their sites need to include a Google JavaScript library in their pages.

Figure 4 details the protocol. The code running in the `implementor`'s context generates a random nonce (N), and creates an iframe that requests a gadget (message 1 in Fig. 4). The nonce is included in the request as a GET parameter. Subsequent communication (messages 4 and 5) between the gadget and the `implementor` includes this nonce. Notice that the private user data (`user info`) is never sent over a `postMessage` channel.

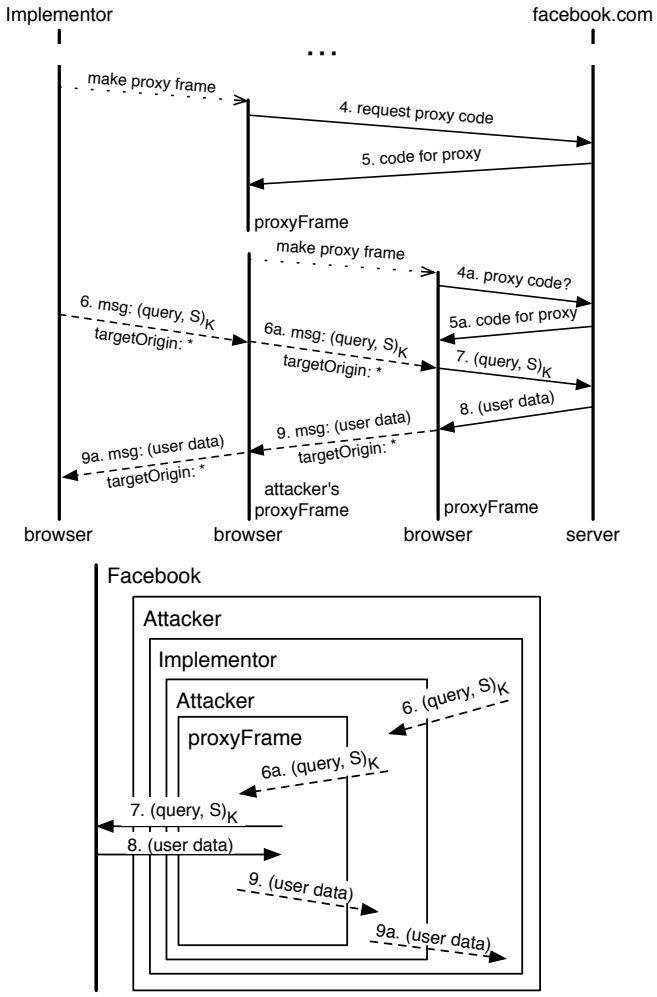


Figure 3: Confidentiality attack on Facebook Connect. (top) Message Exchange—note the replayed messages 6a and 9a. (bottom) Frame hierarchy for the confidentiality attack. Note the presence of two attacker frames—the main window frame and the man-in-the-middle frame.

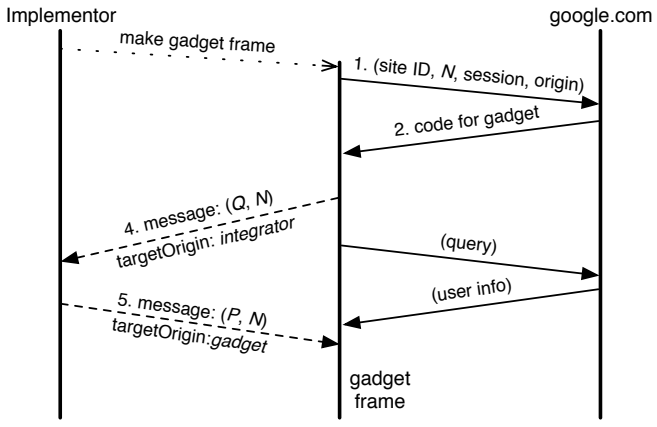


Figure 4: Google Friend Connect's gadget protocol: the nonce N is generated by the `implementor`. Message 4 is a query Q for parameters. The `implementor` responds with the parameters P in message 5.

D. Vulnerabilities in Google Friend Connect

Observation 3: During our testing, we noticed that all message exchanges in the Google Friend Connect protocol had the correct `targetOrigin` set. Analysis of the JavaScript code revealed the absence of any sender authenticity checks. In particular, for all the 12 messages that were exchanged, no participant checked the message sender's origin. Instead, we noticed checks for the nonce (N in Fig. 4). The protocol uses the nonce to authenticate all message exchanges. As the `targetOrigin` is correctly set for all messages, the nonce can never leak to an attacker.

Observation 4: The random number generator provided by the browser (via `Math.random`) is not cryptographically secure (as shown in [6]). With just one call to `Math.random()`, an attacker can guess all future values of `Math.random()`. This breaks the authentication used by Google Friend Connect. For example, on Firefox 3.6, we were able to exactly predict the nonce that would be used by the Google Friend Connect protocol.

Similar to the Facebook Connect attack, the attacker can embed the benign `implementor` in an iframe within his own malicious page. The attacker's page can then sample `Math.random()` to predict the value of the nonce, and then spoof any message exchanged by `implementor` and the gadget over `postMessage`, compromising the Google Friend Connect session (see figure 4). Based on *Observation 3* and *Observation 4* we observe that this attack would have failed if the Google Friend Connect protocol validated the message sender by checking the `origin`, rather than relying on predictable nonces. Correctly setting the `targetOrigin` on all messages makes the protocol secure against confidentiality attacks.

E. Discussion

Authenticity and confidentiality are strong properties that the `postMessage` API can provide, in principle. Our study of real world usage of the `postMessage` API reveals that developers do not use the abstractions provided by the `postMessage` primitive correctly. Designing in-house secure protocols is challenging—as we've seen. Both Facebook Connect and Google Friend Connect tried to achieve sender authenticity by using their own system (secret nonce or HMAC), instead of the recommended practice (checking the `origin` parameter). We were able to circumvent the authentication methods used by these protocols and insert malicious messages in the communication. In the case of Facebook Connect, we were also able to achieve arbitrary code execution.

Despite the fact that `postMessage` can provide foolproof authenticity and confidentiality, client-side protocol designers use complex, network-style protocols instead. We conjecture that this is a possibility because the 'simple' sender origin checks are perhaps not quite so simple. For instance, most specifications and papers include examples like the following:

```
if (event.origin == 'http://example.com') {
    // execute code
}
```

Such examples give a false sense of simplicity. In the real world, the source of messages could be one of many possible fine grained origins and possibly differing schemas. As a result, validating the origin becomes non-trivial. Additionally, for complex protocols, these checks must be repeated for every message—a tedious exercise which can be easily forgotten. In fact, in our discussion with Facebook, we were informed that they used the all-permissive '*' directive because `postMessage` does not support multicast and implementing this functionality would require a series of string-based verification comparisons—which is precisely the problem we have outlined above. Furthermore, if a mashup includes content from more than a couple of origins, these checks become even more taxing. Fundamentally, this is a usability issue of the API. In Section IV, we suggest potential enhancements to the specifications to mitigate these issues, in keeping with the *economy of liabilities* principle.

The use of the all-permissive '*' as the `targetOrigin` allows leakage of confidential data. The HTML5 specification [15] warns against the use of the '*' literal for confidential data. We believe giving developers the choice of insecure usage is not a good practice. Additionally, it is notoriously hard to figure out what data is privacy sensitive and what isn't [9]—and we believe this will only get more difficult. Based on these facts, we suggest a possible modification in Section IV.

III. Persistent, Server-Oblivious Client-side Database Attacks

In this section, we study the usage of new client side persistent storage mechanisms supported by HTML5 and Google Gears. We find that data stored in client-side databases is often used in code evaluation constructs without sanitization. Client-side databases, thus, provide additional avenues for attackers to persist their payloads across sessions. For instance, attackers only need to inject XSS attack payloads once into the client-side storage to have them repeatedly compromise the client-side code integrity for sustained periods of time (unlike a common reflected XSS issue which is fixed once patched). Additionally, because the attack payload is stored on the client-side, the server is oblivious to the nefarious activity. We show that the 7 out of 11 major applications we studied trust the client-side storage and are vulnerable to such *persistent* attacks, including: Gmail, Google Buzz, Google Documents and others.

A. Client-side Storage: Background

HTML5 proposes two persistent storage abstractions: `localStorage` and `webDatabase` [16], [17]. A limited number of browsers currently support these features. The client-side storage mechanisms work as follows:

- `localStorage` is a key/value store tied to an application's origin. Script executing within an origin can get or set values of the data store using the `localStorage` object.
- `webDatabase` is a client-side database that supports execution of SQL statements. The database is bound to the origin in which the code executes and web applications are restricted to only modifying the structure and contents of their associated origin's database. To execute SQL against the database one can use: `executeSql(sqlStatement, arguments, callback, errorCallback)`.
- Gears is a Google product designed to enable applications to work offline. Recently, Google has decided to deprecate Gears in favor of HTML5 [4]. Despite syntactic differences, Gears and HTML5 `webDatabase` data storage work in very similar ways.

In each of these cases, database modifications persist until the creating application destroys the data.

B. Persisting Server-Oblivious Attack Payloads

We consider two possible attack vectors in our threat model, a network attacker and a transient XSS vulnerability.

The goal of either attacker is to inject code into the local storage in order to gain a persistent foothold in the application—one that remains even when the transient attack vector is fixed. Once an application has been compromised, the attacker has control of the application until the client side database is cleared. In current implementations, this only occurs when the database is explicitly cleared by an application, making the attack have a long lifetime.

Network Attacker. Consider the case when a network attacker is able to modify packets destined to the victim. When the user visits a site using client-side storage the attacker modifies the victims network packets to allow the network attacker to inject arbitrary JavaScript. This allows the attacker to compromise the database with no trace server-side that a client-side exploit has occurred until the client-side database is cleared.

As an example of a realistic scenario, consider when a user visits a coffee shop with open wireless. Unbeknownst to him, the network attacker intercepts his network connections so that they are forwarded through the attacker's computer. When the user visits Google Buzz, the network attacker modifies the page returned to supply a script which modifies the client-side database. Now, whenever the data from the database is used in a code evaluation construct, the attack payload is executed instead. The user now leaves the cafe with a compromised machine and due to the stealthy injection (with no server side XSS required), little evidence remains that an attack occurred.

Transient XSS. As a second attack vector, suppose that an attacker has exploited a transient XSS vulnerability as a primary attack vector and has been able to execute arbitrary code within the context of the target site. The attacker is able to modify the database arbitrarily because the attacker has used the XSS to execute JavaScript with the same privilege as the code running within that origin. Not only is this attack persistent, it is also *stealthy*. Besides the initial XSS injection vector, all of the code execution and state modification happens on the client-side rendering the server oblivious to the attack.

For a concrete example, suppose an attacker finds an XSS attack on a web email application that uses

`webDatabase` to save emails. In such a case, the attacker writes an exploit such that its payload is stored inside an email in the database. When the user views the email, the injected code is executed. Now, even if the XSS vulnerability is fixed, the payload persists as long as the database.

In either case, it's important to note that if the injected database data is used in code evaluation constructs, such as `eval` or `document.write` without proper sanitization (as we observed), the attack can persist its attack payload. This payload can be used for a variety of attacks such as stealing passwords, cookies and email. The execution of the code on the client-side and resulting payload is *stealthy* because the server is oblivious to the compromise.

C. Approach

We evaluated **11** applications that use client-side storage using Kudzu. Kudzu, a systematic vulnerability finding tool built on the WebKit framework, is a dynamic symbolic execution engine framework which is designed to analyze JavaScript applications running in browsers [10]. We modified Kudzu to mark database outputs as symbolic and we note a possible vulnerability when a database output flows to a critical sink (like `innerHTML` or `eval`). All vulnerabilities were verified in Safari 4.0.4 by modifying the content of the database being targeted to contain executable code. Experiments using Google Gears were verified in Firefox 3.5.8. We verify that the code is executed by viewing the target application. In order to ensure that HTML5 features were used when applicable, we modified our `User-Agent` string to match the latest reported by an Apple iPhone.

Experimental Results. Figure 5 shows that we find vulnerabilities in **7** applications. In addition, it presents the type of persistent storage being used, and whether or not the database modification remains persistent.

Application	Storage Type	Vulns.	Persistent?
Gmail	Database	Yes	Yes
Google Buzz	Database	Yes	Yes
Google Calendar	Database	No	N/A
Google Documents	Gears	Yes	Yes
Google Maps	Database	Yes	Yes
Google Reader	Gears	Yes	Yes*
Google Translate	Database	No	N/A
Snapbird	localStorage	Yes	Yes
Remember The Milk	Gears	No	N/A
Yahoo Apps Mobile	Database	No	N/A
Zoho Writer	Gears	Yes	Yes*
Total	—	—	**7**

Figure 5: A security evaluation of applications using client-side database storage. The modified database persisted through reloading of the application, closing the browser, and logging in and logging back out. Note: (*) indicates that the attack only persisted while the application was in offline mode.

Gmail. We walk through a sample attack on Gmail to give an idea how a typical persistent attack may take place. First, we launch Gmail using Kudzu to analyze the application. We login to our account and are then taken to our Inbox. After this we close the browser. Kudzu then notifies us that it found data going from the database into the inner text of a `div` tag, without proper sanitization.

We concretely verified the attack. First, we note that Safari implements an SQLite database on a per origin basis. We open the database associated with Gmail, in this case `/Library/Safari/Databases/https mail.google.com_0`, and modify the `body` field of message found in the `cached_messages` table to include the text `<img src=dne onerror=alert(1);>`. When the Gmail application uses the database, the cached message containing the attack payload is executed.

D. Discussion

Our experimentation reveals a lot of inconsistency in the way that developers sanitize their database outputs before using them in critical constructs. We found that many prominent applications, such as Google Reader, Gmail and Google Buzz do not sanitize their database output at all. In contrast, we found a few applications aware of the severity of the mentioned attacks and they perform some kind of sanitization on their database output.

One such application, Google Calendar, sufficiently mitigates the attack. It uses a complex combination of JSON and XML to verify the data format, and sanitizes the user input to further ensure that scripts were not injectable.

Another application that mitigates code injection is Google Translate. When using Translate, the result of a translation is placed into a text node on the user's page. Therefore, the attack is mitigated as no code can be executed in a text node.

However, all of the other applications failed to sufficiently sanitize database outputs. We speculate that some applications did not sanitize database outputs because of the complexity of the sanitization process required to eliminate the attack. Consider Gmail and Google Buzz, two applications that have fields in their database representing the textual content of an email or buzz respectively, in both cases, containing HTML. When these fields are modified by an attacker, the original content and injected attack text are rendered to the user, without the attack text being sanitized. In Gmail and Buzz, the textual content is mixed with HTML and the task of stripping away all of the possible scripting elements which result in code execution is difficult. Thus, when an attacker views the email or buzz, the persistent code in the database executed.

We also found some intermediate cases, including Zoho Writer, a web browser based document editor, and Google Reader. Both applications were only susceptible to a transient client-side database attack. That is, the data only persists in the offline store for as long as the client was offline. When the user returned online, the cache was cleared and refreshed with new content.

These examples show that different applications vary in the richness of content that they store in the database. For instance, the juxtaposition of the policies of Gmail and Buzz versus Translate indicates that there is an inherent disconnect between what security features are necessary

and what are currently provided. In Section IV we suggest several enhancements to these primitives that make the secure use of database outputs easier.

IV. Enhancements

Client-side browser primitives expect users to perform multiple sanitization checks at various points in the code, to prevent the attacks we outlined. Further, such validation functionality is duplicated across applications. These checks are tedious, repetitive and sometimes complex, which adds unnecessary liability to developers leading to inconsistencies in use and errors. In Section I, we proposed the general principle of economy of liabilities in the design of abstractions which helps minimize the required liability on users to ensure security.

Retrofitting the principle in existing client-side primitive designs is challenging. Below we suggest enhancements to the primitives we study, in ways which are a compromise between the need for flexibility, compatibility and security.

A. Enhancing postMessage

In Section II, we raised the question of whether it should be possible to make the `postMessage` design easier for safe usage. We believe this is a topic of debate for the web community, in light of the empirical fact that early adopters of `postMessage` are using the primitives unsafely. On the flip side, we point out that any changes to the web platform come with cost to compatibility and generality too. We outline our suggestions below to stimulate the discussion on the best way to use these primitives securely.

Origin Whitelist. Based on the current usage, in order to ensure authenticity of messages received, we suggest a declarative system for specifying origins allowed to send messages will function better than manual origin checks. For instance, the Content Security Policy proposal allows a website to specify a whitelist of origins trusted to execute code in the website's security context [7]. We suggest extending CSP with a directive to specify origins allowed to send messages to the website. Moreover, the CSP proposal has gone through intense community discussion and at least one implementation—making it a potential starting point to build on.

In addition, from our experiments and evaluation of applications that use the `postMessage` API, in order to protect confidentiality, we recommend that broadcast should be disabled in favor of *multicast*, based on our discussions with Facebook during responsible disclosure. Currently, `postMessage` does not permit wildcard characters in domain names. However, to support multicast the API could be changed to allow the application declaratively specify a wildcard in a domain name (e.g. *.facebook.com). This would restrict the domains capable of sending messages without the need for complex regular expressions for parsing and verification. Additionally, if required, allowing for finer-grained control for recipients is also a possibility—the `postMessage` function could take a list of origins that are allowed to receive the specified message. With this primitive in place, it would be the *browser*'s responsibility to check the sender's origin with this whitelist before delivering the message.

Origin Comparison Primitive. Instead of requiring every user of the `postMessage` API to implement a function for comparing origins, it would be much more efficient for the browser to provide this as a primitive function. If the browser provided the primitive, such a function would support comparison based on some standard language for specifying origins (like the grammar in CSP [7]). Note that browsers already have to do such checks for enforcing the same origin policy [12]. The grammar for this list could be similar to the grammar for origins specified in Content Security Policies, omitting the all-permissive '*' [7].

B. Database output sanitization

Sanitizing the values stored by a database before using them in critical constructs can protect against persistent XSS attacks. We found few applications which performed any type of database output sanitization. But, like `postMessage`, we noticed that the output sanitization can often be complex and occur throughout the application code.

This is not a scalable approach. Instead, the browser should automatically remove any potentially executable script constructs inside database values before returning them. In order to accomplish this, browsers could take the output of the database and filter it through a function similar to `toStaticHTML`. This construct, found natively in Internet Explorer, removes dynamic HTML elements and attributes from a fragment of HTML [8]. In the exceptional case, where a web application requires that its own routines be used to sanitize and verify the database output, the call to the database could disable this check by including an optional boolean argument. In our experience, this change would not impact functionality of all applications that we studied, but would protect them against persistent XSS attacks.

Most importantly, no matter what the embodiment of the final primitive, the user needs to understand the full limitations of the API as to not be lulled into a false sense of security, as we have seen in the past [1].

C. A Cryptographically Secure PRNG

As we have seen in Google Friend Connect, the lack of a cryptographically secure Pseudo-Random Number Generator has not deterred developers from creating their own cryptographic protocols. We observe that if the implementation of `Math.random()` was cryptographically secure, our attack on Google Friend Connect would have been mitigated. Nonetheless, we reiterate that developers should use `postMessage` for enforcing authenticity and confidentiality in their applications instead of creating their own cryptographic solutions.

We realize that the above discussion to retrofit additional security involve changes to existing or developing specifications. As the APIs studied are relatively nascent, we are hopeful of a positive response from the community. In the present scenario, without modification, users of

these APIs can use JavaScript analysis techniques to detect and eliminate such attacks during testing [10], [14]. Analysis systems similar to ours can be extended to taint data from `postMessage`, `localStorage` and `webDatabase`, ensuring that no tainted data flows to critical code evaluation constructs without sufficient validation. We have had some success in the past with such an approach [10], [11].

V. Conclusion

New primitives, especially for browser-side functionality, are being designed and proposed at a rapid pace to facilitate the demand for interactivity while enabling security. However, a recurring problem in these designs is that these abstractions are not designed with the economy of liabilities principle in mind, i.e., they rely significantly on the developers to ensure security. In this paper, we found this to be true of two recent client-side abstractions: `postMessage`, a cross-domain communication construct and client-side persistent storage (HTML5 and Google Gears). In the case of `postMessage`, we reverse engineered the client-side protocols and systematically extracted the security-relevant checks in the code to find new vulnerabilities in them. In the case of client-side storage, we found that applications do not sanitize database outputs, which can lead to a stealthy, persistent, client-side XSS attack. We found bugs in several prominent web applications including Gmail and Google Buzz and uncovered severe new attacks in major client-side protocols like Facebook Connect and Google Friend Connect.

We hope our study encourages future primitives to be designed with the economy of liabilities principle in mind. We offer some enhancements to existing to the current APIs to shift the burden of verifying and ensuring security properties from the developer to the browser. And, we encourage developers to scrutinize their applications for similar problems using automated techniques.

VI. Acknowledgments

We thank Chris Grier, Adam Barth, Adrian Mettler, Adrienne Felt, Jon Paek, Collin Jackson, and the anonymous reviewers for helpful feedback on the paper and suggestions for improvements on the work.

This work is partially supported by the Air Force Office of Scientific Research under MURI Grant No. 22178970-4170, the National Science Foundation under Grant No. 0448452, and the National Science Foundation Trust user Grant No. CCF-0424422. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation or the Air Force Office of Scientific Research.

References

[1] PHP magic quotes. http://php.net/manual/en/security.magicquotes.php.

[2] A. Barth, C. Jackson, and W. Li. Attacks on JavaScript mashup communication. In *Web 2.0 Security and Privacy*, 2009.

[3] A. Barth, C. Jackson, and J. C. Mitchell. Securing frame communication in browsers. In *Proceedings of the 17th USENIX Security Symposium (USENIX Security 2008)*, 2008.

[4] I. Fette. Hello HTML5. http://gearsblog.blogspot.com/2010/02/hello-html5.html.

[5] B. Hoffman and B. Sullivan. *Ajax Security*.

[6] A. Klein. Temporary user tracking in major browsers and cross-domain information leakage and attacks, 2008. http://www.trusteer.com/sites/default/files/Temporary_User_Tracking_in_Major_Browsers.pdf.

[7] Content Security Policy. https://wiki.mozilla.org/Security/CSP/Spec.

[8] toStaticHTML Method. http://msdn.microsoft.com/en-us/library/cc848922\%28VS.85\%29.aspx.

[9] A. Narayanan and V. Shmatikov. Robust de-anonymization of large sparse datasets. In *Proceedings of 29th IEEE Symposium on Security and Privacy*, 2008.

[10] P. Saxena, D. Akhawe, S. Hanna, S. McCamant, F. Mao, and D. Song. A symbolic execution framework for JavaScript. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2010.

[11] P. Saxena, S. Hanna, P. Poosankam, and D. Song. FLAX: Systematic discovery of client-side validation vulnerabilities in rich web applications. In *17th Annual Network & Distributed System Security Symposium, (NDSS)*, 2010.

[12] Same origin policy for JavaScript. https://developer.mozilla.org/En/Same_origin_policy_for_JavaScript.

[13] M. Sutton. The Dangers of Persistent Web Browser Storage. www.blackhat.com/blackhat-dc-09-Sutton-persistent-storage.pdf, 2009.

[14] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Cross-Site Scripting Prevention with Dynamic Data Tainting and Static Analysis. In *Proceeding of the Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2007.

[15] W3C. HTML 5 specification. http://www.w3.org/TR/html5/.

[16] W3C. Web SQL Database. http://dev.w3.org/html5/webdatabase/.

[17] W3C. Web Storage. http://dev.w3.org/html5/webstorage/.