Signing Me onto Your Accounts through Facebook and Google: a Traffic-Guided Security Study of Commercially Deployed Single-Sign-On Web Services

Rui Wang
Indiana University Bloomington
Bloomington, IN, USA
wang63@indiana.edu

Shuo Chen
Microsoft Research
Redmond, WA, USA
shuochen@microsoft.com

XiaoFeng Wang
Indiana University Bloomington
Bloomington, IN, USA
xw7@indiana.edu

***Abstract*— With the boom of software-as-a-service and social networking, web-based single sign-on (SSO) schemes are being deployed by more and more commercial websites to safeguard many web resources. Despite prior research in formal verification, little has been done to analyze the security quality of SSO schemes that are commercially deployed in the real world. Such an analysis faces unique technical challenges, including lack of access to well-documented protocols and code, and the complexity brought in by the rich browser elements (script, Flash, etc.). In this paper, we report the first "field study" on popular web SSO systems. In every studied case, we focused on the actual web traffic going through the browser, and used an algorithm to recover important semantic information and identify potential exploit opportunities. Such opportunities guided us to the discoveries of real flaws. In this study, we discovered 8 serious logic flaws in high-profile ID providers and relying party websites, such as OpenID (including Google ID and PayPal Access), Facebook, JanRain, Freelancer, FarmVille, Sears.com, etc. Every flaw allows an attacker to sign in as the victim user. We reported our findings to affected companies, and received their acknowledgements in various ways. All the reported flaws, except those discovered very recently, have been fixed. This study shows that the overall security quality of SSO deployments seems worrisome. We hope that the SSO community conducts a study similar to ours, but in a larger scale, to better understand to what extent SSO is insecurely deployed and how to respond to the situation.**

Keywords*— *Single-Sign-On, Authentication, Web Service, Secure Protocol, Logic Flaw

1. INTRODUCTION

Imagine that you visit *Sears.com*, a leading shopping website, or using *Smartsheet.com*, a popular project management web app, and try to get in your accounts there. Here is what you will see (as in Figure 1): *Sears* allows you to sign in using your Facebook account, and *Smartsheet* lets the login go through Google. This way of authentication is known as *single sign-on* (SSO), which enables a user to log in once and gain access to multiple websites without the hassle of repeatedly typing her passwords. Web SSO is extensively used today for better user experience. According to a recent survey, a majority of web users (77%) prefer web SSO to be offered by websites [7].

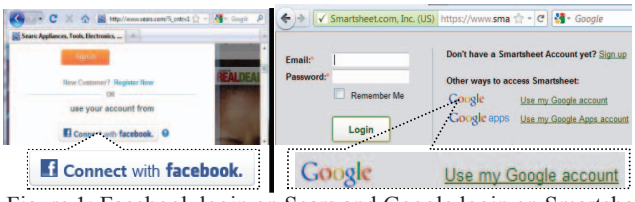

Figure 1: Facebook login on Sears and Google login on Smartsheet

SSO systems such as Kerberos have been there for years. However, never before has the approach seen such extensive commercial deployments as what happen on today's web, thanks to the increasing popularity of social networks, cloud computing and other web applications. Today, leading web technology companies such as Facebook, Google, Yahoo, Twitter and PayPal all offer SSO services. Such services, which we call *web SSO*, work through the interactions among three parties: the user represented by a browser, the ID provider (a.k.a, IdP, e.g., Facebook) and the relying party (a.k.a, RP, e.g., Sears). Like any authentication scheme, a secure web SSO system is expected to prevent an unauthorized party from gaining access to a user's account on the RP's website. Given the fact that more and more high-value personal and organizational data, computation tasks and even the whole business operations within organizations are moving into the cloud, authentication flaws can completely expose such information assets to the whole world.

Motivation of this research. Given the critical role of SSO today, it becomes imperative to understand how secure the deployed SSO mechanisms truly are. *Answering this question is the objective of our research.*

Actually, SSO has been studied in the protocol verification community for a while, which we will discuss in the related work section. The main focus of these studies was to design formal methods to find protocol flaws. However, no prior work includes a broad study on commercially deployed web SSO systems, a key to understanding to what extent these real systems are subject to security breaches. Moreover, even though formal verifications are demonstrated to be able to identify vulnerabilities in some SSO protocols [2], they cannot be directly applied here to answer our question, due to the following limitations. First, the way that today's web SSO systems are constructed is largely through integrating web APIs, SDKs and sample code offered by the IdPs. During this process, a protocol serves merely as a loose guideline, which individual RPs often bend for the convenience of integrating SSO into their systems. Some IdPs do not even bother to come up with a rigorous protocol for their service. For example, popular IdPs like Facebook and Google, and their RPs either customize published protocols like OpenID or have no well-specified protocols at all. Second, the security guarantee an SSO scheme can achieve also intrinsically depends on the system it is built upon. Vulnerabilities that do not show up on the protocol level could be brought in by what the system actually allows each SSO party to do: an example we discovered is that Adobe Flash's cross-domain capability totally crippled Facebook

DOI 10.1109/SP.2012.30

SSO security (Section 4.2). Finally, formal verification on the protocol level cannot find the logic flaws in the way that the RP misuses the results of an SSO for its decision-making. For example, we found that the RPs of Google ID SSO often assume that message fields they require Google to sign would always be signed, which turns out to be a serious misunderstanding (Section 4.1). These problems make us believe that a complete answer to our question can only be found by analyzing SSO schemes on real websites.

Challenge in security analysis of real-world SSO. Security analysis of commercially deployed SSO systems, however, faces a critical challenge: these systems typically neither publish detailed specifications for their operations nor have their code on the RP and IdP sides accessible to the public. What is left to us is nothing more than the web traffic that goes through the browser. On the bright side, such information is exactly what the adversary can also see. This makes our analysis realistic: whatever we can discover and exploit here, there is no reason why a real-world attacker cannot do the same.

Given our limited observation of the interactions between commercial IdPs and their RPs (as shown in Figure 2), we have to focus our analysis on the traffic and operations of the browser. Fortunately, the browser actually plays a critical role in web SSO. More specifically, an SSO system is typically built upon the RP's integration of the web APIs exposed by the IdP. Through these APIs, the RP redirects the browser to the IdP to authenticate the user when she attempts to log in. Once succeeds, the browser is given either a certified token for directly signing into the RP (the case of Smartsheet) or a secret token that the RP can use to acquire the user's identity and other information from the IdP (the case of Sears). Note that during this process, the browser must be bound to the authentication token to prove to the RP the user's identity that the browser represents. This requires the critical steps of an SSO, e.g., passing of the token, to happen within the browser. The browser-centric nature of web SSO makes it completely realistic to analyze the browser traffic to identify logic flaws.

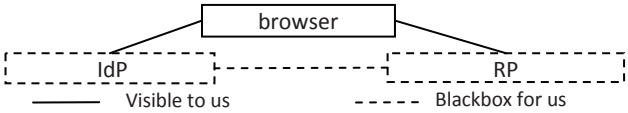

Figure 2: an SSO triangle and our visibility as an outsider

Our study and findings. The web services/websites we investigated include high-profile systems that utilize the aforementioned IdPs. Our study shows that not only do logic flaws pervasively exist in web SSO deployments, but they are practically discoverable by the adversary through analysis of the SSO steps disclosed from the browser, even though source code of these systems is unavailable. The web SSO systems we found to be vulnerable include those of Facebook, Google ID, PayPal Access, Freelancer, JanRain, Sears and FarmVille. *All the discovered flaws allow unauthorized parties to log into victim user's accounts on the RP*, as shown by the videos in [33]. We reported our findings to related parties and helped them fix those bugs, for which we were acknowledged in various ways, e.g., public recognitions, CEO's thank and monetary reward, which we will mention in Section 4.

Our methodology. When investigating an SSO case, our analysis begins with an automated black-box test on the HTTP messages, which the browser passes between the RP and the IdP for invoking the APIs on either side. We call these messages *browser relayed messages* (BRMs). This test identifies the HTTP field that carries the authentication token and other fields that directly or indirectly affect either the value of the token or the destination it will be sent to (e.g., a reply URL). What we are interested in is the subset of these fields that the adversary could access under different adversary assumptions that we will describe in Section 2.2. Once such knowledge has been gathered by the automatic test, we move on to understand whether the adversary has the capability to forge the token that is supposedly authentic or steal the token that is supposedly a secret. Oftentimes, this brings us directly to a set of specific technical questions that serve as sufficient conditions for an exploit to succeed. These questions are answered by doing more insightful system testing or by looking for knowledge from domain experts. Our experience proves that this analysis methodology indeed gives effective guidance in finding real-world SSO logic flaws.

Roadmap. The rest of the paper is organized as follows: Section 2 offers the background about web SSO and the adversary models we studied; Section 3 a number of basic concepts that Section 4 will base on, and our tool to extract basic ground truths of an SSO scheme; Section 4 presents the main study of this paper; Sections 5 and 6 discuss our retrospective thought and related work; Section 7 concludes.

2. BACKGROUND

2.1. Web Single Sign-On: a View from the Browser

SSO is essentially a process for an IdP to convince an RP that because *this browser* has signed onto the IdP as Alice, *this same browser* is now granted the capability to sign onto the RP as Alice. The tricky part here is that the IdP must bind Alice's capability to the correct browser that truly represents Alice. In all existing SSO systems, such a binding is through *proof-by-possession*: Alice's browser needs to present to the RP a token issued by the IdP to demonstrate that it possesses the capability that the IdP grants to Alice. Security of an SSO scheme depends on how the token is handled, so the browser naturally undertakes many critical steps, and thus is the focus of our investigation.

Browser relayed message (BRM). An SSO process can be described as a sequence of browser relayed messages exchanged between the RP and the IdP. Typically, an HTTP communication can be thought of as a sequence of request-response pairs, as shown in Figure 3 (upper). Each pair consists of an HTTP request *Xa*, where *X* is the number of requests the browser has made (i.e., 1a, 2a, etc.), and its corresponding HTTP response *Xb* (1b, 2b, etc.) to be sent

back from the server (either the RP or the IdP). A browser relayed message (BRM) refers to a response message *Xb* followed by a request (*X*+1)*a* in the next request-response pair, as illustrated in the figure.

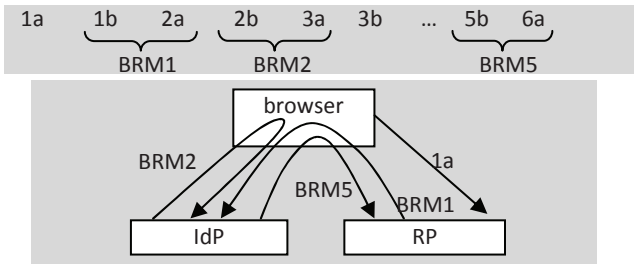

Figure 3: upper: a browser-relayed message (BRM) consists of a response and the next request; lower: a sample SSO process

Each BRM describes a step of the SSO in which the server handler (e.g., a web API) of step *X* passes data to the server handler of step *X*+1, with the browser state piggybacked. The entire SSO process is bootstrapped by request 1a sent to the RP. It triggers BRM1, which is, for example, for the RP to declare its website identity to the IdP. More BRMs may occur as needed afterwards. The last BRM (e.g., BRM5 in Figure 3 (lower)) finally convinces the RP of the user's identity that the browser represents.

A BRM can be, for example, (1) an HTTP 3xx redirection response (2) a response including a form for automatic submission, or (3) a response with a script or a Flash object to make a request. In this paper, we do not differentiate these implementations and instead, describe each BRM in a format described by the following example:

```
src=a.com  dst=Facebook.com/a/foo.php
   Set-cookies: sessionID=6739485
   Arguments: x=123 & user=john
   Cookies: fbs=a1b2c3 & foo=43da2c2a
```

Intuitively, this BRM is interpreted as: "*a.com* (source server) asks the browser to set cookie `sessionID = 6739485` for its domain and to send a request to destination URL *Facebook.com/a/foo.php*; the request contains arguments `x=123` and `user=john` provided by *a.com*, as well as cookies `fbs=a1b2c3` and `foo=43da2c2a` stored in the browser for the domain *Facebook.com*." In the above example, each underlined item is called an *element*, which includes the BRM's source, destination, or other name-value pairs of set-cookies, arguments and cookies.

2.2. Threat and Adversary Model

Threat. Web SSO faces various security and privacy threats, as studied in prior research [29][30][31][32], which we will describe in the related work section. Our research focuses on the type of security flaws that completely defeats the purpose of authentication: that is, *the unauthorized party Bob signs in as the victim user Alice.*

Adversary's roles. When evaluating the threat from the malicious party Bob, we need to understand who he can communicate with and what roles he can play in an SSO process. It is easy to see that Bob can actually interact with all SSO parties: not only can he talk to the RP and the IdP, but he can also set up a website, which, once visited by Alice, can deposit web content to Alice's browser. Such interactions are described in Figure 4.

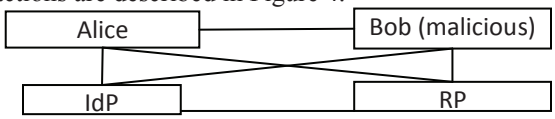

Figure 4: possible communications when Bob is involved

From the figure, we can see that because of Bob's involvement in the communication, there are four possible SSO triangles similar to the one shown in Figure 2. These SSO triangles are Alice-IdP-Bob, Bob-IdP-RP, Alice-IdP-RP and Alice-Bob-RP. In our study, we did not consider the last one, in which Bob acts as the IdP and can steal Alice's authentication information through phishing, as the focus of our research is logic flaws in SSO systems, not social engineering. In the remaining three relations described as scenarios (A), (B) and (C) respectively in Figure 5, Bob's roles allow him to identify and exploit SSO vulnerabilities. Specifically, in (A), Bob is a client in an SSO and attempts to convince the RP that his browser represents Alice, assuming that he knows Alice's username through a prior communication; in (B), when Alice visits Bob's website, Bob acts as an RP to the IdP, in an attempt to get Alice's credential for the target RP; in (C), Bob leaves malicious web content in Alice's browser during her visiting of his website, which can perform SSO operations through sending requests to the IdP and the RP. Of course, these three scenarios are just high-level strategies. How to carry out the strategies is exactly what we need to figure out from the study to be presented next.

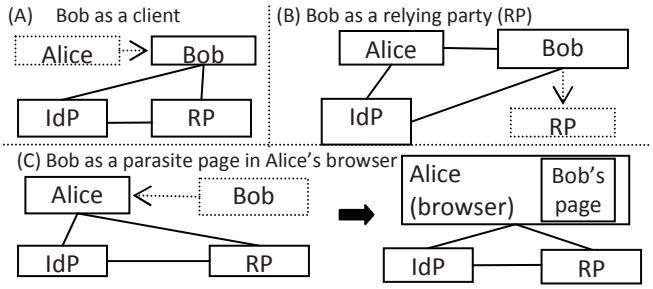

Figure 5: three basic types of exploitations by Bob

3. KEY CONCEPTS IN BRM-GUIDED ANALYSIS

The main findings of our study will be presented in Section 4, but in order to clearly explain the vulnerabilities and how we discovered them step-by-step, we need to introduce in this section some important basic concepts that section 4 will base upon. These concepts are derived from features in BRM traces by an automatic tool that we built, namely *the BRM analyzer*.

3.1. The BRM Analyzer

Our BRM analyzer was designed to perform a black-box, differential analysis on BRM traces. The analyzer needs to capture/parse BRMs and further modify/replay HTTP requests. To this end, we installed *Fiddler* [15], a

web proxy capable of uncompressing/decoding/parsing all HTTP messages, on the browser machines used in our research. We also utilized Firefox's debugging tool Firebug [16] to modify and replay browser requests.

Figure 6 shows how the analyzer works. To conduct an analysis, we need two test accounts (i.e., user1 and user2, with different user names, email addresses, etc.) to collect three traces, including two for user1's logins from two different machines and one for user2's login from one machine, which serve as the input to the analyzer. Each trace records all the BRMs observed by the browser during a login. These traces are processed by the analyzer through three steps (Figure 6), which perform comparisons, regular expression matching and some dynamic tests. These steps aim at identifying and labeling key elements in an SSO and other elements related to these elements. Their output describes the elements and their relations under the three adversarial scenarios in Figure 5.

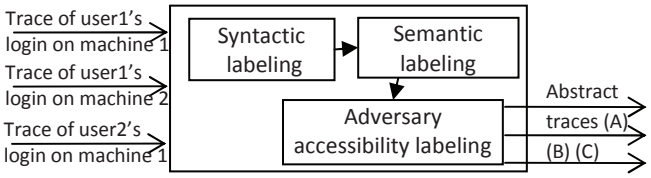

Figure 6: input, output and the three steps of the BRM analyzer

In the rest of the section, we elaborate these steps, which include *syntactic labeling*, *semantic labeling* and *adversary accessibility labeling*, using the following raw trace as an example.

```
BRM1: src=RP  dst=http://IdP/handler.php
    Arguments: notifyURL=http://RP/auth.php
    Cookies: sessionID=43ab56c2
BRM2: src=IdP dst=http://RP/auth.php
    Arguments: username=Alice & sig=11a3f69
```

Syntactic labeling. The first step of our analysis is to determine the syntactic types of individual elements in BRMs. Table 1 lists all the types with their examples. The lexical grammar we used to recognize these types is straightforward, which we do not elaborate here due to the space limitation. Our analyzer performs such type recognition using a single trace, labeling each element it identifies. For example, for the element "*notifyURL=http://RP/auth.php*", the analyzer attaches a label [URL] to it. To ensure the correctness of such labeling, our approach automatically compares the types of the same element (e.g., *notifyURL*) across all three traces: once an inconsistency is found, it reports to the human analyst for reconciliation, though this happened rarely in our study.

Table 1: types

Label	Example value
INT (decimal no longer than 4 digits)	123
WORD	Alice
BLOB (decimal longer than 4 digits, or a hexadecimal or alphanumeric number)	43ab56c2
URL	http://RP/auth.php
LIST	(x, y, z)

Semantic labeling. After the types of individual elements are labeled, our analyzer moves on to identify their semantic meanings. Table 2 summarizes the semantic attributes defined in our research, which are obtained through a series of black-box tests described below. Note that we include the descriptions for "UU (user-unique)", "MU (client-machine-unique)", "SU (session-unique)", "BG (browser-generated)", "SIG? (signature-like)" and "NC (newly-created)" in Table 2, since they are straightforward.

Table 2: semantic attributes

UU (user-unique): We compare the three input traces. An element is labeled "UU" if it has an identical value in the two traces of user1's logins, and a different value in the trace of user2's login. This element holds a value unique to the user.
MU (client-machine-unique): An element is labeled "MU" if it has an identical value in the two users' login traces on machine1, and a different value in the trace of user1's login on machine2.
SU (session-unique): An element is labeled "SU" if it has different values in all three input traces.
BG (browser-generated): an element not included in the response, but appearing in the request that follows.
SIG? (signature-like): It is a BLOB element whose name contains the substring "sig". Such an element is likely a signature. We need a replay test to confirm it.
pChain (propagation chain): An element uses this chain to find all elements in the trace that have the same value as this element.
NC (newly-created): it is an element whose pChain is null, indicating that the element does not come from a prior BRM.
SIG (signature): It indicates an element confirmed as a signature. We create a data structure to describe its properties, including its signer and whether it covers the entire argument list or only selectively.
SEC (secret): it indicates a secret specific to the current session and necessary for the success of the authentication.
"!" (must-be): When a src value of a BRM is prefixed with this label, it means that the element must have this value in order for the authentication to succeed.

pChain (propagation chain). To identify the elements accessible to the adversary under different circumstances, we need to understand how the value of an element is propagated to other elements across different BRMs. To this end, our analyzer attaches to every element a pChain attribute that serves to link related elements together. In the following we describe how to discover such connections: (1) for each element except `src` and `dst` (see the example) in a BRM, the analyzer compares its value with those of the elements on all its predecessors in a reverse chronological order; the element's pChain is set to point to the first (i.e., chronologically latest) element on the prior BRMs that contains the identical value; (2) we also set pChain of the `src` element on every BRM to point to the `dst` element of its prior BRM.

SIG label. To identify a signature on a BRM, we first look for those labeled as "SIG? (signature-like)" and "NC (newly created)". The presence of these two labels is a necessary yet insufficient condition for a signature in most web SSO systems, as discovered in our study. To avoid

false positives, our analyzer performs a dynamic test on such an element to find out whether it indeed carries a signature. Specifically, our analyzer first changes the element's value and replays the message: if the message is rejected, then the element is labeled as SIG. When this happens, the analyzer further adds and removes the elements in the message to find out those protected by the signature. In all the cases we studied, a signature either covered the whole URL, the whole argument list or some elements in the argument list. In the last situation, the message also contains a *LIST* element that indicates the names of protected elements.

SEC label. For every newly-created session-unique BLOB element (i.e., those with NC, SU and BLOB labels), the analyzer also changes a digit of its value and replays the message. If the message is rejected, this element is labeled SEC to indicate that it is a secret.

"!" (must-be) label. If a signature or a secret is created by a party in a benign scenario, then even in an attack scenario, it has to be created by the same party in order for the attack to succeed. In other words, no signature or secret can be faked by another party. Thus, for every BRM containing a newly created element of SIG or SEC, the analyzer prefixes the `src` value of the BRM with a "!", which also propagates to the `dst` of its prior BRM.

Ignoring pre-existing cookies. Our analysis only cares about the cookies set after a user starts an SSO process, so any cookie whose corresponding set-cookie element is not on the trace does not need to be analyzed, i.e., if a cookie's pChain does not lead to a set-cookie element, we ignore it.

Let's look back at the sample trace. After it has been processed by the analyzer, we obtain a trace below. Note that the analyzer removes the concrete values of all elements except those of `src`, `dst`, URL and LIST elements, and replaces them with labels of their semantic meanings. The dashed arrows depict pChain links in their opposite directions, which show propagations. BRM2 has a newly created signature element, so its `src` is labeled as "!IdP", which also causes the `dst` element in BRM1 to bear a "!". The cookie is ignored as it was set before the SSO starts.

```
BRM1: src=RP   dst=https://!IdP/handler.php
Arguments: notifyURL[URL]
Cookies: sessionID[BLOB]
BRM2: src=!IdP dst=https://RP/auth.php
Arguments:
username[WORD][UU] & sig[BLOB][SU][NC][SIG]
```

Adversary accessibility labeling. Over the trace labeled with individual elements' semantic meanings, our analyzer further evaluates whether the adversary, Bob, can read or write elements in the three SSO triangles in the scenarios illustrated in Figure 4: Bob-IdP-RP, Alice-IdP-Bob and (Alice+Bob)-IdP-RP. Here readability and writability are denoted by ↑ and ↓ respectively. Table 3 elaborates the rules we used to label individual elements, to indicate how they can be accessed by the adversary.

Table 3: labeling rules for adversary's accessibility

Scenario (A): Bob acts as a browser • All elements are readable; • An element not covered by a signature is writable; • For an element protected by a signature, if it is newly created (NC), then it is not writable; otherwise, inherit the writability label from its ancestor using pChain.
Scenario (B): Bob acts as an RP to the IdP in order to get Alice's credential for the target RP • Replace any occurrence of "RP" in the trace with "Bob"; • For any BRM sent to Bob (or the dst element is writable), all *Argument* or *Cookie* elements in the BRM are readable; • For any BRM made by Bob, the *dst* element, or any *Argument* or *Set-cookie* element in the BRM is writable, if the element is not protected by the IdP's signature; • For an element protected by a signature, if it is newly created (NC), then it is not writable; otherwise, inherit the writability label from its ancestor using pChain.
Scenario (C): Bob deposits a page in Alice's browser • No element is readable; • Cookies and set-cookies are not writable; • Because the BRM can be generated by Bob, the *dst* element or any *Argument* element in a BRM is writable, if the element is not protected by a signature; • For an element protected by a signature, if it is newly created (NC), then it is not writable; otherwise, inherit the writability label from its ancestor using pChain.

Output visualization. After analyzing the input traces, the BRM analyzer produces its output in dynamic HTML, which allows a human analyst to conveniently retrieve the understanding obtained through the automatic analysis using a browser. Figure 7 is a screenshot that displays an output trace. When the mouse hovers over an element, the element and all other elements on its pChain are all highlighted, which enables the analyst to examine how the value of the element propagates. The mouseover event also brings up a tip popup that shows the element's value.

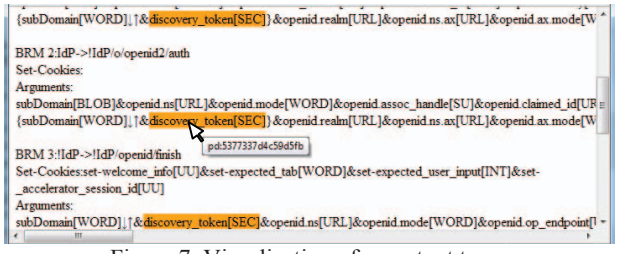

Figure 7: Visualization of an output trace

4. STUDYING SSO SCHEMES ON MAJOR WEBSITES

Like a debugger extracting ground truths about call stack, memory and registers, the BRM analyzer described in section 3 extracts necessary ground truths about an SSO scheme to be studied, e.g., what Bob could read or write, especially some key elements (e.g., those labeled with SEC or SIG, etc.). With this tool, we now can go onto the field study about leading commercial web SSO systems. The study covers popular SSO services on the web (e.g., Facebook, Google, JanRain and PayPal), and the SSO systems of high-profile websites/services (e.g., FarmVille, Freelancer, Nasdaq and Sears). The result shows that these prominent web SSO systems contain serious logic flaws that

make it completely realistic for an unauthorized party to log into their customers' accounts. These flaws are also found to be diverse, distributed across the code of RPs and IdPs, and at the stages of login and account linking. We elaborate these vulnerabilities in the rest of the section.

4.1. Google ID (and OpenID in general)

OpenID is a popular open standard for single sign on. It was reported that there were over one billion OpenID-enabled user accounts and 9 million websites using OpenID as of December 2009 [22]. Google ID is based on OpenID. The number of its relying websites is very significant.

Analysis result. Our analysis on Google ID started with the raw traffic. Not surprisingly, the raw traffic would be very time-consuming for human to parse and analyze. Using the BRM analyzer, we could automatically obtain the semantic information about the trace and the three adversarial scenarios in Figure 5. The trace for scenario (A) is shown in Figure 8, in which the RP is *Smartsheet.com* and the IdP is *Google.com*. All elements in the BRMs are readable in scenario (A), so the readability label (↑) is ignored. The figure only shows the writability label (↓). Note that a specific design of OpenID is that many enumerable values are expressed in the format of URL. This detail is not important to our description below, so we label them `[WORD]` to avoid potential confusion.

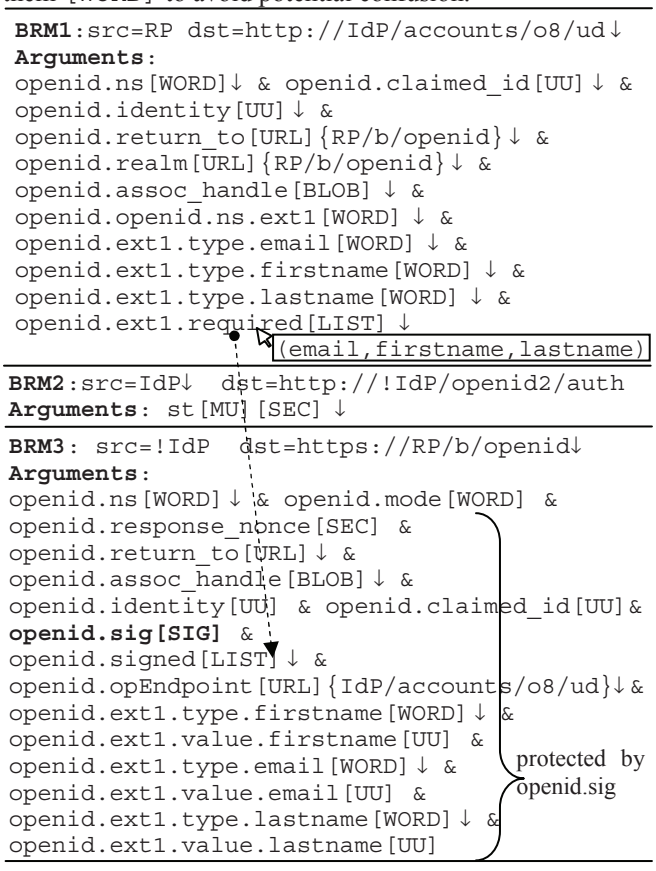

Figure 8: GoogleID+Smartsheet trace for scenario (A)

We found that BRM3 is the message for proving to the RP the identity of the user the browser represents. This message carries a SIG element `openid.sig`, indicating that the SSO is based on a signed token. The analysis further revealed the elements covered by the signature, as marked in Figure 8. Among these elements, `openid.signed` is a list that indicates the names for those signed elements. What is interesting here is that some of the signed elements were labeled by our analyzer as writable by the adversary. A closer look at them shows that their values are actually propagated from BRM1, which are not under any signature protection. Particularly, `openid.signed` contains the list from `openid.ext1.required` on BRM1, an element that describes which elements the RP requires the IdP to sign, such as `email`, `firstname` and `lastname`, as shown in the popup by the mouse cursor in Figure 8. However, since `openid.signed` (BRM3) can be controlled by the adversary through `openid.ext1.required` (BRM1), there is no guarantee that any of the elements that the RP requires the IdP to sign will be signed by the IdP (i.e., protected by `openid.sig`) in BRM3.

Flaw and exploit. It is very common for a website to use a user's email address (e.g., alice@a.com) as his/her username, which is probably why the RP requires `email` to be signed. The analysis above shows that an attacker in scenario (A) may cause the IdP to exclude the email element from the list of elements it signs, which will be sent back to the RP through BRM3. Therefore, the question to be asked about an actual system is:

> *Does the RP check whether the email element in BRM3 is protected by the IdP's signature, even though the protection has been explicitly required by BRM1?*

It turns out that this question indeed points to a serious logic flaw in Google ID SSO. Specifically, we tested the exploit on *Smartsheet*: when our browser (i.e., Bob's browser) relayed BRM1, it changed `openid.ext1.required` (Figure 8) to `(firstname,lastname)`. As a result, BRM3 sent by the IdP did not contain the email element (i.e., `openid.ext1.value.email`). When this message was relayed by the browser, we appended to it *alice@a.com* as the email element. We found that *Smartsheet* accepted us as Alice and granted us the full control of her account.

Broader impacts. We further discovered that the problem went far beyond *Smartsheet*. Google confirmed that the flaw also existed in open source projects OpenID4Java (an SDK that Google authentication had been tested against) and Kay Framework. In OpenID4Java, the function for an RP to verify BRM3 is `verify()`. The source code showed that it only checked whether the signature covered all the elements in the `openid.signed` list, so a "verified" BRM3 does not ensure authenticity of the elements that the RP required the IdP to sign. Besides smartsheet, we examined other popular websites *Yahoo! Mail*, *zoho.com*, *manymoon.com* and *diigo.com*. They were all vulnerable to this attack.

Responses from Google and OpenID Foundation. We reported our finding to Google, Yahoo and OpenID Foundation, and helped Google to fix the issue. Google and OpenID Foundation published security advisories about this issue, in which they acknowledged us. We provide these advisories in [33]. Several news articles reported these advisories, including those from eWeek, The Register, ZDNet, Information Week, etc [33]. We received a monetary reward from Google, who also added our names to its official acknowledgement page [18].

4.2. Facebook

Authentication on Facebook often goes through Facebook Connect, which is a part of Facebook's platform. We studied this SSO scheme.

Analysis result. We performed our automatic analysis on the traces collected from an SSO through Facebook Connect. The result (not involving the adversary) is illustrated in Figure 9. Here, the IdP is Facebook, and the RP is NYTimes.com. We can see here that BRM3 carries a secret token `result`, which the browser uses to prove to the RP the user's identity. The secret comes from BRM2 as an argument for the API call *http://!IdP/xd_proxy.php*[1]. This secret token enables the RP to acquire Alice's information from Facebook and also grant her browser access to her account. Also interesting here is BRM1, in which the RP declares to the IdP its identity (e.g., NYTimes) through `app_id` and provides other arguments. Note that though the element `cb` in the figure is also labeled as SEC, it was found to be generated by the browser (labeled BG, see Table 2) and thus not a secret shared between the RP and the IdP.

```
BRM1:src=RP dst=http://!IdP/permissions.req
Arguments: app_id[BLOB] & cb[SEC][BG] &
   next[URL]{
      http://!IdP/connect/xd_proxy.php?
      origin[BLOB]&transport[WORD]
   } & … & … & … (other 13 elements )
BRM2:src=!IdP  dst=http://!IdP/xd_proxy.php
Arguments: origin[BLOB] & transport[WORD] &
           result[SEC] & … & … (other 4 elements )
BRM3:src=!IdP  dst=http://RP/login.php
Arguments: origin[BLOB] & transport[WORD] &
           result[SEC] & … & … (other 3 elements )
```

Figure 9: the benign Facebook+NYTimes trace

Our analyzer further evaluated the trace in Figure 9 under different adversarial scenarios. Figure 10 shows what we found under Scenario (B), in which the adversary Bob impersonates the RP to Facebook when Alice is visiting his website. According to Table 3, all occurrences of "RP" are replaced with "Bob". A potential vulnerability immediately shows up here is that all elements in BRM1, including `app_id`, are writable, so Bob could declare that he was NYTimes using the `app_id` of NYTimes, which is public knowledge. As a result, the secret token `result` in BRM3, which Facebook generates specifically for Alice's access to NYTimes and for NYTimes to acquire Alice's Facebook data under her consent, now goes to Bob.

```
BRM1:src=Bob dst=http://!IdP/permissions.req
Arguments: app_id[BLOB]↓ & cb[SEC][BG] &
   next[URL]{
      http://!IdP/connect/xd_proxy.php↓?
      origin[BLOB]↓& transport[WORD]↓
   } & … & … & … (other 13 elements )
BRM2:src=!IdP   dst=http://!IdP/xd_proxy.php↓
Arguments: origin[BLOB]↓ & transport[WORD]↓ &
           result[SEC]↑ & … & … (other 4 elements )
BRM3:src=!IdP↓  dst=http://Bob/login.php
Arguments: origin[BLOB]↓ & transport[WORD]↓ &
           result[SEC]↑ & … & … (other 3 elements )
```

Figure 10: the Facebook+NYTimes trace in scenario (B)

Flaw and exploit. Again, we had to verify whether the above identified opportunity was indeed exploitable. This time, things turned out to be more complicated than they appeared to be. Specifically, we tested the exploit by setting all arguments of BRM1 to those on a normal Facebook+NYTimes SSO trace. We found that although Facebook indeed responded as if it was communicating with NYTimes (i.e., all the arguments, including `result`, were carried in BRM2), the browser failed to deliver these arguments to `http://Bob.com/login.php` in BRM3, and thus thwarted our exploit. This test clearly indicates that Facebook's web contents protect the secret token `result` within the user's browser.

Our manual analysis of the web contents reveals that such protection comes from the same-origin policy enforced by the browser, which Facebook leverages to ensure that the browser only transfers the secret token from Facebook's domain to the domains of authorized parties such as NYTimes, but not Bob.com. The browser mechanisms that Facebook utilizes for this goal include "postMessage", "Adobe Flash" and "fragment". A relying website, e.g., NYTimes.com or Bob.com, is allowed to choose one of them using the `transport` element in BRM1. Figure 11 shows how the protection works when Adobe Flash is used.

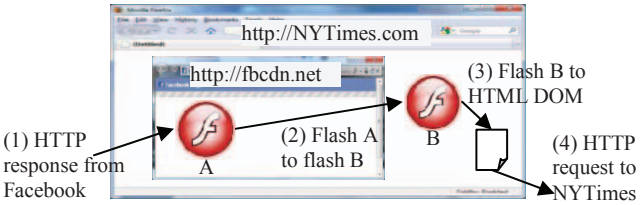

Figure 11: The complete view of a benign BRM3

The browser takes four steps to transfer the secret (i.e., `result` element) from Facebook to NYTimes. The cross-domain communication happens during Steps (2) and (3) between two windows, one rendering the content for NYTimes and the other for *fbcdn.net*, which is affiliated with Facebook. Each of them hosts a Flash object, denoted

[1] The hostname is !IdP, rather than IdP, because our test showed that Facebook server whitelists its allowed hostnames. It only allows a hostname under *facebook.com* or a Facebook-affiliated domain, such as *fbcdn.net*, etc.

by *A* and *B* respectively. Both objects are supposed to be downloaded from fbcdn.net during the SSO. This allows Flash *A* to pass the secret to Flash *B* because they are of the same origins (fbcdn.net). Flash *B* further sends the secret to the HTML DOM of its hosting page only if the page's domain is indeed NYTimes. Our exploit mentioned above was defeated by this defense mechanism, which seems logically secure: Flash's same-origin policy ensures that the secret will be passed only when Flash *B* is loaded from fbcdn.net, which implies that Flash *B* will only hand over the secret to NYTimes, not to other domains.

Let's look at our adversarial scenario, in which the domain of the hosting page is actually Bob.com, although it declares to be NYTimes.com in BRM1. To bypass the defense and obtain the secret token in Alice's browser, Bob must find a way to either let Flash *A* pass the secret token to a Flash downloaded from Bob.com website or convince the trusted Flash *B* (from fbcdn.net) to send the token even when Flash *B*'s hosting page is Bob.com, not NYTimes.com. In other words, the problem of attacking this SSO can be reduced to one of the following questions:

- *Is it possible to let Flash B (from fbcdn.net) deliver the secret to the web page from Bob.com?*
- *Is Flash A (from fbcdn.net) allowed to communicate with a Flash object from Bob.com?*

For the first question, we analyzed the ActionScript of Flash *B* from fbcdn.net and did not find any way to make it send the secret to a non-NYTimes page. For the second question, we found that the answer is positive, because of a unique cross-domain mode of Adobe Flash called *unpredictable domain communication* [23]: by naming a Flash object from Bob.com with an underscore prefix, such as "_foo", Flash *A* can communicate with it despite the fact that the Flash comes from a different domain. Note that this logic flaw was found thanks to the domain knowledge about how Flash communicates, which serves as the last link on the chain of our exploit. We made an exploit demo [33] to show how this exploit works: once Alice visits *Bob.com* while she has signed onto Facebook, *Bob.com* uses its Flash to acquire the secret token from Flash *A*, which allows Bob to log into NYTimes as Alice and also impersonate NYTimes to access Alice's Facebook data, such as her personal information (e.g., birthdate), status updates, etc.

Our communication with Facebook. Because the problem was on Facebook's side, all RP websites were subject to the same exploit that worked on NYTimes. We reported the finding to Facebook, and suggested a way to fix the issue. After 9 days, Facebook confirmed our finding through email, and applied our suggested fix on the same day. Facebook acknowledged us on its public webpage for security researchers [12] (before Facebook implemented the "bug bounty" monetary reward program). The finding was also reported in several news stories, including those on Computer World, The Register, eWeek, etc [33].

4.3. JanRain

JanRain is a prominent provider of social login and social sharing solutions for commercial businesses and websites. It claimed to have over 350,000 websites using its web SSO services. Its customers include leading websites such as *sears.com*, *nasdaq.com*, *savings.com*, etc. Its flagship product, *Janrain Engage*, wraps individual web SSO services from leading IdPs, including Google, Facebook, Twitter, etc, into a single web SSO service. By using the service, its customers adopt these SSO schemes altogether and thus avoid integrating them one by one. This service is interesting not only because of its popularity but also because of the unique role it plays in web SSO: it is a wrapper IdP service that relies on the wrapped IdPs for authentication. This potentially makes the already complicated web SSO systems even more complex.

Analysis result. Figure 12 shows the trace produced by the BRM analyzer when our test server did an SSO using Google ID through JanRain. Before we can come to the details of this analysis, a few issues need to be explained. First, in our adversarial scenarios, IdPs are the parties not under Bob's control, so we simply treat both JanRain and Google as a single IdP party for the convenience of the analysis. Second, to integrate JanRain's service, an RP needs to register with JanRain a unique application name (AppName) for the RP's web application, e.g., "*RP-App*". JanRain then creates a subdomain *RP-App.rpxnow.com* for this application (*rpxnow.com* is a domain owned by JanRain). This subdomain will be used by the RP to communicate with JanRain a set of settings for the SSO process. JanRain server stores these settings and refers to them through a handle, denoted as `settingsHandle`[2] in our analysis. Also note that in this analysis, we treat AppName as an argument, although it is a subdomain. For example, *http://AppName.rpxnow.com/a.php?foo&bar* is shown as:

```
src=xxx  dst=http://IdP/a.php
Arguments: AppName & foo & bar
```

Figure 12 describes 7 BRMs during this complicated SSO (login using Google ID through JanRain). When a user wants to sign onto an RP, the RP generates BRM1 to inform the IdP (i.e., JanRain) about its `AppName`, together with the settings for this SSO. Such settings include: `openid_url`, a URL for activating the Google ID authentication, and `xdReceiver` and `token_url`, which are the `dst` elements for BRM5 and BRM7 respectively. In the figure, BRM2 – BRM4 (enclosed in the dashed bracket) describe the traffic of Google ID authentication, as shown previously in Figure 8. By the end of BRM4, JanRain gets the user's Google profile data. BRM5 – BRM7 pass a secret `token` to the RP for retrieving the profile data from JanRain.

[2] In the actual implementations, this handle is called "discovery_token" in JanRain's wrapping of Yahoo and Google, and "_accelerator_session_id" in its wrapping of Facebook.

BRM1: src=RP dst=http://!IdP/openid/start **Arguments**: AppName & openid_url{http://IdP/account/o8/ud} & xdReceiver{http://IdP/xdcomm?AppName}& token_url{http://RP/finish-login} & … & … (other 2 elements)
BRM2:src=!IdP dst= http://IdP/account/o8/ud **Arguments**: all Google ID's arguments as shown in BRM1 in Figure 8, in which openid.return_to is set to *http://IdP/openid/finish?AppName&settingsHandle*
BRM3: Google ID's traffic, similar to BRM2 in Figure 8.
BRM4:src=!IdP dst=http://!IdP/openid/finish **Arguments**: AppName & settingsHandle[SEC] & AllOpenIDData (a pseudo element that we introduce for the sake of presentation simplicity. It represents all data returned from Google ID as in BRM3 in Figure 8)
BRM5: src=!IdP dst=http://IdP/xdcomm **Arguments:** AppName & redirectUrl { http://IdP/redirect?AppName&loc[SEC]}
BRM6: src=IdP dst=http://!IdP/redirect **Arguments:** AppName & loc[SEC]
BRM7: src=!IdP dst= http://RP/finish-login **Arguments:** token[SEC]

Figure 12: benign traffic of our website integrating JanRain that wraps Google ID

We further analyzed the BRMs under the three adversarial scenarios. Figure 13 shows the result for Scenario (B), where Bob impersonates the RP to the IdP.

BRM1: src=Bob dst=http://!IdP/openid/start **Arguments**: AppName↓ & openid_url↓ & xdReceiver↓ & token_url↓ & … & …
BRM2 – BRM4: (details omitted, see Figure 12)
BRM5: src=!IdP dst=http://IdP/xdcomm↓ **Arguments:** AppName↓ & redirectUrl { http://IdP/redirect?AppName&loc[SEC]↑}
BRM6: src=IdP dst=http://!IdP/redirect **Arguments:** AppName↓ & loc[SEC]↑
BRM7:src=!IdP dst=http://Bob/finish-login↓ **Arguments:** token[SEC]↑

Figure 13: adversarial scenario (B)

An opportunity that we can easily identify is BRM1, in which Bob could set AppName↓ to that of the target RP while pointing token_url↓ to his own domain. This would trick JanRain into collecting the user's profile data from Google for the RP and sending the secret token[SEC]↑ to Bob, as token_url serves as the dst element for BRM7.

Flaw and exploit. To understand whether this opportunity indeed works, we set up a server as a mock target RP of the attack. The test revealed that like Facebook, JanRain also puts in place some protection measures. JanRain requires every registered app to supply a whitelist for identifying the app's associated domains. For example, the whitelist for RP-App includes "RP-App.rpxnow.com" and "*.RP.com". The token_url of BRM1 needs to be on the whitelist. In our test, the arguments of BRM1 were AppName="RP-App" & token_url="http://Bob.com/finish-login", which JanRain found to be inconsistent with the whitelist (*Bob.com* not on the whitelist of RP-App) and thus stopped the SSO. Furthermore, we found that even if we temporarily added *Bob.com* to the mock RP's whitelist to let BRM1 succeed (and removed it from the whitelist after BRM1), the secret token obtained from BRM7 is still useless. This is due to another check against the whitelist: when a website uses the token to retrieve Alice's Google ID profile from JanRain, JanRain finds something wrong: the token was previously sent to *Bob.com* according to the token_URL; thus *Bob.com* is supposed to be on the RP's whitelist, but it is not.

Given the protection of whitelisting, it is clear that token_url in BRM1 must be in a domain on RP-App's whitelist (e.g., *http://RP.com/finish-login*). The trouble now is that dst on BRM7 is exactly token_url. In other words, once token_url is set according to the target RP's whitelist, there is no way that Bob can have BRM7 sent to him. This forced us to look back at the result of our analysis and try another opportunity. Actually, dst in BRM5 is propagated from the xdReceiver in BRM1, which Bob appears to be able to write. If he could change this element (e.g., to http://Bob.com/xdcomm) without being caught, he could have JanRain send him BRM5. BRM5 is also important, as it contains loc, another piece of secret. Stealing loc is as damaging as stealing token. If Bob obtains loc, his exploit will succeed, as loc is the only secret Bob needs in order to use his own browser to go through BRM6 and BRM7, which will get Alice's session into the browser. Therefore, we saw that stealing loc through BRM5 was a plausible idea.

Our test showed both encouraging and challenging sides of the idea. On the challenging side, we found that JanRain also checked xdReceiver in BRM1 against the whitelist and therefore thwarted the exploit at the very beginning; on the encouraging side, we confirmed that if we could succeed in setting xdReceiver to Bob.com /xdcomm, we would indeed get loc, and this loc value would indeed enable an end-to-end successful exploit.

The remaining question is how to set the RP's xdReceiver so that it points to Bob.com/xdcomm. Bob must accomplish this without being caught by the whitelist check in BRM1. The only option is to let Bob use his own AppName (i.e., Bob-App) in BRM1, because Bob can arbitrarily whitelist any domain that he wants for Bob-App. Essentially, it means Bob is not constrained by the whitelist check when BRM1 has argument AppName="Bob-App". How can this affect the settings (i.e., token_url and xdReceiver) for RP-App? Remember that after BRM1, the settings are referenced by settingsHandle collectively, which can be thought of as a secret session ID. The only hurdle for our exploit is how to bind this session ID (which is for Bob-App) to our target RP-App. Interestingly, we found that this binding is established by

BRM2 through its argument `openid.return_to` (Figure 8). This gives us another opportunity.

Here is our third plan, consisting of two steps: first, Bob's own browser makes the request of BRM1 with `AppName="Bob-App" & token_url="http://RP/finish-login" & xdReceiver="http://Bob/xdcomm"`. This not only gets him through the whitelist (which is defined by himself) but also gives him `settingsHandle` to represent the above two URLs. In the second step, Bob impersonates the RP: whenever Alice visits Bob's website, the website generates BRM2, which binds `RP-App` to Bob's `settingsHandle` through `openid.return_to`. As a result, Bob will get `loc` in BRM5, allowing his browser to impersonate Alice's, as described before. This plan turned out to work nicely. A video demo is in [33].

Other JanRain SSO schemes. We found that the same exploit also worked on JanRain's wrapping of YahooID SSO. However, JanRain's wrapping of Facebook SSO uses a different way to bind AppName and `settingsHandle`: it sets `settingsHandle` as a cookie under *AppName.rpxnow.com*. To exploit this SSO, we had to figure out a way to let *Bob-App.rpxnow.com* set the `settingsHandle` cookie for *RP-App.rpxnow.com*. In other words, the security of the scheme can be reduced to the following question:

Do browsers allow cross-(sub)domain cookies to be set?

Access control for browser cookies, especially between subdomains, is a complex issue, which has been studied for example in [8]. We learned from existing literature that browsers at least share cookies of an HTTP domain with its corresponding HTTPS domain. This implies a disappointing fact – *Facebook-wrapped JanRain SSO cannot secure HTTPS websites even when it is over HTTPS*. Imagine a banking website that runs this SSO scheme over HTTPS in order to protect the communication from a network attacker, e.g., a malicious router. Whenever the user visits any HTTP website, like google.com, the network attacker can insert a hidden iframe to access *http://RP-App.rpxnow.com*, which sets the `settingsHandle` cookie for this subdomain. The cookie will be shared with *https://RP-App.rpxnow.com* (the HTTPS domain), making the above exploit succeed.

Bug reporting and JanRain's responses. We have reported this issue to JanRain, who acted quickly to fix it within two days. Later JanRain notified us that due to a compatibility issue with their legacy systems, their fix for the JanRain-Facebook issue had to be rolled back. The developers were working on a new fix.

4.4. Freelancer.com, Nasdaq.com and NYSenate.gov

Freelancer.com is the world's largest online outsourcing marketplace [17], which helps match buyers' projects to the services that sellers can offer. The website has about 3 million users, 1.3 million projects and earned over 100 million dollars. Like many other websites today, it allows Facebook sign-on, but in a different fashion: a user first needs to register an account, as what happens on a website not supporting SSO; then, she can "link" this account to her Facebook account, which allows her to log in through Facebook afterwards. Therefore, the security of this SSO critically depends on the linking process.

We found other high-profile websites that also enable SSO through account linking, such as Nasdaq.com (linkable to Facebook accounts) and NYSenate.gov (linkable to Twitter accounts). We have confirmed that they all contain exploitable vulnerabilities similar to that of Freelancer, which we describe below as an example.

Analysis result. We used our analyzer to study the traces collected from a user's linking operation on Freelancer.com under different adversarial scenarios. Figure 14 describes what we found under Scenario (C), where Bob has a malicious web page in Alice's browser, which can call other websites' APIs. Specifically, BRM1 queries Facebook (the IdP) for Alice's profile data. BRM3 does the linking[3]. In BRM2, Facebook generates a secret `result`. As described in the previous Facebook example, BRM3 takes advantage of the browser-side security mechanism to pass `result` to the RP's page. Then, Freelancer.com (the RP) sets the value of `result` in cookie `fbs`, and calls `lnk.php` to do the linking. As we can see from the analysis, the system needs to ensure that `fbs` indeed holds Alice's Facebook profile data when `lnk.php` is called.

```
BRM1:src=RP dst=http://!IdP/permissions.req
Arguments: app_id[BLOB]↓ & cb[SEC][BG] &
    next[URL]{
        http://!IdP/connect/xd_proxy.php↓?
        origin[BLOB]↓&transport[WORD]↓
    } & … & … & … (other 14 elements )
BRM2:src=!IdP  dst=http://!IdP/xd_proxy.php↓
Arguments: origin[BLOB]↓& transport[WORD]↓&
           result[SEC] & … & … (other 4 elements )
BRM3:src=!IdP dst=http://RP/facebook/lnk.php
Arguments: auto_link[INT]↓ & goto_url[URL]↓
Cookies: fbs[SEC]
```

Figure 14: Traffic for scenario (C)

Flaw and exploit. The opportunity we see is that Bob can log into Freelancer as Alice if his web page in Alice's browser manages to link her Freelancer account to Bob's Facebook account. To this end, two things must happen: (1) the page signs Alice's browser onto Bob's Facebook account, and then (2) it makes the browser do the linking.

Linking from Alice's browser. Let us first assume that Step (1) has succeeded, and focus on (2). The trouble here is that Bob's page cannot produce BRM1, due to the presence of a browser-generated secret `cb`. Alternatively, we can try to directly invoke BRM3. The only hurdle here is that without BRM1–BRM2, cookie `fbs` would not been

[3] This step includes the client-side communication to pass the token `result` from an IdP's page to an RP's page (Section 4.2).

assigned the profile data of the current Facebook logon user. Interestingly, we found that by making the browser visit the page *http://freelancer.com/users/change-settings.php* (no argument required), the current Facebook user's profile is queried and set to cookie `fbs`. The visit is essentially an API call to accomplish BRM1–BRM2 with no secret. Bob's page can then make the request of BRM3 for the linking.

S*igning Alice's browser onto Bob's Facebook account*. Now we look at how to make step (1) happen. We analyzed the traffic of Bob signing onto Facebook from his own browser, which was a POST request to *https://www .facebook.com/login.php* with username and password as its arguments. The same request, however, was denied by Facebook when it was produced by Bob's page. A comparison between the traces of the two requests revealed that the *referrer* header in the successful one was set by Facebook.com, while that of the failed request was within Bob's domain. We had known from various sources that referrer-checking is an unreliable means for discriminating cross-site requests from same-site ones, because the referrer header is sometimes removed at the network layer for legitimate privacy reasons [5]. We tested the login request again with its referrer removed, Facebook accepted it. Thus, an exploit comes down to the answer to the question below:

How to send a POST request with no referrer header?

This question turned out to have known answers. Two browser experts pointed us to some working examples, as well as information resources, such as [26]. We tested one of the working examples, shown in Figure 15, and confirmed that it works on the latest versions of IE, Chrome and Firefox. Using this approach, we were able to sign in as Alice on Freelancer.com, thereby confirming the presence of the logic flaw in its integration of Facebook's SSO service. As discussed before, the same vulnerability exists on Nasdaq.com and NYSenate.gov. The SSO of NYSenate.gov is through Twitter.

a.html <iframe src="b.html"></iframe>

b.html

```
<iframe name="formFrame"></iframe>
<script>  formFrame.document.body.innerHTML= '<form
  name="tfm" action= "http://foo.com/bar" method="post"
  target= "_top" > <input type="text" name="arg"/><input
  type="submit"/> </form>';
formFrame.document.all.tfm.submit();  </script>
```

Figure 15: an implementation of referrer-free posting

Bug reporting and Freelancer's response. We reported the issue to Freelancer. The company's CEO Matt Barrie thanked us and asked for suggestions about the fix [33]. We offered two suggestions, of which Freelancer adopted one.

4.5. OpenID's Data Type Confusion

Our study on OpenID-based systems also uncovers a serious logic flaw, which is caused by the confusion between the RP and the IdP on the interpretation of BRM elements. We believe that the problem is pervasive. It has been confirmed on *Shopgecko.com*, one of the first adopters of *PayPal Access* (PayPal's new SSO service announced on 10/13/2011), and *Toms.com*, a shopping website. The findings were made a few days before our paper submission.

Flaws and exploits. Let's look at the BRM traffic of Smartsheet and GoogleID in Figure 8. Our analysis shows that `openid.ext1.type.email` (`type.email` for short), an element in BRM1 and BRM3, is writable under Scenario (A) (where Bob controls the web client). A further analysis of the element reveals that it affects the value of `openid.ext1.value.email` (`value.email` for short), a signed element in BRM3. The RP typically treats this element as a user's email address, but Google (the IdP) thinks differently. It actually sets the element's value according to `type.email`. Initially in BRM1, the RP sets the value of `type.email` to *http://schema.openid.net /contact/email*, OpenID's type for emails. However, Bob can change it to other types, such as *http://axscheme.org /namePerson/first* (OpenID's data type for first names). As a result, `value.email` in BRM3 can hold the user's first name. This enables an exploit if Bob could register with Google a first name "alice@a.com". Remember that *Smartsheet* uses the registered email of a user as her authentication token. This type confusion can lead to signing Bob onto Alice's account. We confirmed that *Smartsheet* indeed takes Bob's first name as an email during the exploit. We believe that the misunderstanding about the content of `value.email` is pervasive, given that Google developer's guide only uses `value.email` as an example of requested user attributes in its specification, and never mentions how its content is actually determined [19].

However, this exploit did not get through, because Google ID's user registration page does not treat "alice@a.com" as a valid first name. Therefore, a natural question produced by our analysis is whether there is a way to use "alice@a.com" as the value of any non-email field in Bob's Google ID profile, maybe through direct API calls instead of the user registration page.

Now we show where this exploit does work. *Shopgecko.com* identifies a user by her PayPal ID, which is not a secret. The type of the ID is *https://www.paypal .com/webapps/auth/schema/payerID*, which Bob can change to *http://schema.openid.net/contact/street2*, the type of "mailing address' second line". We successfully registered a user whose mailing address' second line is Alice's PayPal ID. For *toms.com*, we found the element "email" in fact contains a user's Twitter ID during a Twitter SSO, though it indeed carries email addresses in other SSOs, such as Google ID. Bob, a Google user, can register his first name as "AliceOnTwitter", which is Alice's Twitter ID, and sign in as Alice through Google.

Bug reporting. We have reported the end-to-end cases to PayPal, Google, OpenID Foundation, Toms.com and Magento (developer of Shopgecko). Google will fix it by checking the value of `type.email`. Google also asked us to directly bring this issue to the attention of the executive director of OpenID Foundation.

4.6. Other confirmed and potential flaws in studied cases

In the prior subsections, we describe serious logic flaws we found in several web SSO systems. They are actually only a tip of the iceberg: there are some other systems either vulnerable to our exploits or on the verge of being cracked. Table 4 lists eight more cases we studied.

Table 4: some other cases that we confirmed or found promising

	The SSO scheme and the specific system-level question
1 √	**SSO**: Facebook Legacy Canvas Auth **Question**: does a Facebook app check the signature of BRM3 that Facebook generates? (The flaw was confirmed on *FarmVille*.com)
2 √	**SSO:** Facebook Connect **Question**: does an RP of Facebook SSO redirect the user to an attacker's URL despite a failed whitelist checking? (The flaw was confirmed on *zoho.com*.)
3 √	**SSO:** JanRain's wrapping of Facebook **Question**: does an RP of JanRain-SSO whitelist **.rpxnow.com*, not specifically *RP-App.rpxnow.com* (The flaw was confirmed on *sears.com*)
4	**SSO:** Facebook SSO with the RP requesting `access_token` **Question**: what kind of damage can be done by the leakage of `access_token` alone? (We found that the `access_token` that *Groupon.com* requests can be obtained by the attacker.)
5	**SSO:** Facebook Connect **Question**: Can a Javascript in Bob.com read FlashVars of a Flash in the RP's domain, if the Flash allows cross-domain access? If so, we found that *nike.com* would be broken.
6	**SSO:** Facebook Connect Question: does a RP import Facebook's *xd_proxy.php* script for its cross-domain communication?
7	**SSO:** Facebook Connect **Question**: does an RP have an API for universal redirection, such as "*http://foo.com/redirect.php?url=http://bob.com*"?
8	**SSO:** SSO on *livingsocial.com*, *toms.com* and *diigo.com* **Question**: when Bob makes Alice's browser sign onto an RP as Bob, can Bob obtain his own session cookie in the browser?

Our analysis on these cases all led to potential exploit avenues, which come down to a few questions. Three of these cases (with √) were indeed confirmed and reported. More information of these eight cases is described in the full version of this paper [33].

5. RETROSPECTIVE DISCUSSION

As discussed at the beginning of the paper, our main contribution is an extensive security study of commercial web SSO systems, which aims at understanding their security quality and design pitfalls, even in the absence of their source code and detailed specifications. This study was made possible by a suite of analysis techniques we built. Such techniques just serve as a necessary tool for analyzing the SSO systems, and their designs, at the current stage, are still simple and preliminary: for example, our BRM analyzer does not seem to be very advanced. What is really important here is the discovery we made using these techniques, which reveals the gravity and pervasiveness of security-critical logic flaws within commercial web SSO systems. We hope that such a discovery will provoke soul-searching in web SSO community, and help build securer SSO systems. Here are our preliminary thoughts.

5.1. Understanding the SSO vulnerabilities

Commonalities in all our vulnerability investigations. All the logic flaws described in the paper, no matter how subtle they are, were all discovered through a simple and rather mechanical procedure at the high level:

(1) Understand whether the SSO is based on a secret token or an authentic token. Accordingly, there are only two types of problems – either a secret token sent to Bob or an authentic token forged by Bob.
(2) Locate the token in BRMs. Understand how it is propagated or how it is covered by a signature.
(3) Apply adversary scenarios to BRMs using Table 3, which corresponds to the only three strategies – Bob acting as another client, Bob acting as another RP and Bob acting as a page in Alice's client.

Our success indicates that the developers of today's web SSO systems often fail to fully understand the security implications during token exchange, particularly, how to ensure that the token is well protected and correctly verified, and what the adversary is capable of doing in the process.

Variations in the vulnerabilities. The variations are in the non-trivial details of individual systems. In this study, we spent a great amount of effort demonstrating such variations. In Section 4, we describe eight end-to-end confirmed cases, which differ significantly from each other in technical details (although for each case, we usually confirmed the similar vulnerability on several websites), e.g., how a signature's coverage is determined, how the browser protects the secrecy of a token, how BRM destinations are checked by servers, how accounts are linked together, how a website handles an anonymous visit, etc. This diversity comes from the way SSO services are integrated: each RP can integrate the same SSO service differently; the security of the integration depends not only on the program logic on RP and IdP sites, but also on the underlying web platform. Given such complexity, we feel that it can be hard to speculate about how a system can go wrong before looking at its details. This is why a lot of detailed investigations need to be conducted with human analyst's creativity and domain knowledge. We do believe, however, that for known vulnerabilities, one can build a tool to automatically identify other websites suffering from similar problems, but it is not the focus of this paper.

RP developers' due diligence. The complexity in implementation and system details suggest that it can be hard for IdP developers to anticipate all possible RP implementations in the world. Because RP developers are the people who put together a concrete system, they are naturally the final gatekeeper for its security. We suspect that most RP developers today may not realize the necessity of such a due diligence, but merely consider SSO implementation as a task of calling individual APIs on IdPs.

We believe that an analysis like what we did is helpful, so we will soon launch and maintain a service at http://sso-

analysis.org for developers to use our methodology. Developers are obviously in a better position to conduct the analysis than us, as they know precisely which data serve as the primary user ID, the underlying system features that the RP code relies on, and other insider knowledge.

5.2. Broader lessons on secure service integrations

Our previous work studied how merchant websites integrate third-party cashier services. We discovered many logic flaws that allow a malicious shopper (client) to shop for free [34]. The issues exposed in this paper, although about SSO, are similarly about service integration logic flaws. We believe that many lessons can be learned from the two studies together and applicable to other service integration scenarios in general, such as authorizing through OAuth, incorporating social networking functionalities, etc.

5.2.1. Challenges in secure service integrations

Service integration is done through an application (e.g., an RP or a merchant website) calling APIs of a service provider (e.g., an IdP or a cashier service). There are two reasons for these APIs to cause security problems:

Underlying execution platform matters. APIs are designed at a certain abstraction level. It is challenging to exhaustively examine their semantics on real operational systems. This challenge has caused security issues over and over again. For example, in the cashier service study, we found a problem due to API developers' neglect of the possibility of concurrent HTTP sessions of web servers (Section III.B.1 of [34]). In the current SSO work, we discovered that developers failed to consider Flash's unpredictable domain mechanism and the feasibility of posting a request without referrer. APIs designed without thorough understanding of their execution platforms and related security implications can be vulnerable.

Compared to secure implementation of APIs, how to call APIs securely can be even more challenging. Consider the notorious *strcpy*, which itself does not contain a buffer-overrun vulnerability, but can easily introduce one to the program that calls it. As an example, many Unix-like systems provide a family of uid-setting APIs, such as *setuid*, *seteuid* and *setguid*. "Demystifying" them and understanding their proper usage were known to be highly nontrivial [10]. We believe that the web APIs we studied also deserve the same effort to "demystify" the way to use them securely. They should be examined with all reasonable usage patterns of the calling sites, and with all conceivable adversary assumptions. For example, Google should have expected reasonable RP websites to use the *email* element to identify a user, and thus realized that Google ID APIs are problematic (see Section 4.1).

5.2.2. What kind of analysis tools are needed

Our experience in this study seems to be complimentary to that of a classic protocol-verification task in several aspects. If the verification community wants to extend the current methodologies to the actual system level, there are some new thrusts that need to be addressed by appropriate tools. Below are the main points distilled from our experience, which explain these thrusts.

Understanding a real-world system could be more challenging than analyzing its well-specified logic model. Verification techniques typically reason about logic models that have been extracted from real systems. For every case that we studied, we spent more time on understanding how each SSO system work than on reasoning at the pure logic level. This suggests that when it comes to examining a real system, we would love to have a tool to help us understand complex system details more than a tool that replaces us in logic reasoning. A desired tool should direct the analyst to grasp key details of the system, like a debugger, which does not find bugs for programmers, but presents key ground truths, such as the call stack, etc., to help programmers. Our BRM analyzer is designed toward this direction.

In-depth security analysis of a real system often happens under incomplete knowledge and needs to be adaptive, iterative and semi-automatic. Given the complexity of a real system, techniques that enable a fully automatic and also in-depth security analysis are still remote. Existing attempts to automate this process often require a complete model of the system, which needs to be manually constructed, before any automatic analysis can happen. However, such a model is hard to build and often too complicated to analyze. What we learned from our study is that security testing of a real system often needs to be performed without complete knowledge of the system, in an adaptive and iterative way: the analyst starts with partial knowledge of the system, designs new tests to probe it, reasons about the test results to improve her understanding of the system, and continues to walk through the process until a viable path is found. This strategy worked well in our study, helping us identify subtle logic flaws and implement complicated yet practical exploits, but we had to manage this process manually. A tool supporting this adaptive process is very needed for offloading analysts' burden.

How to effectively convert exploit conditions into known problems is a valuable research direction. We found that it is relatively easy to understand the security premises of the system, e.g., element `result` should not be obtained by Bob, or cookie `fbs` should not be forged by Bob, etc. However, it is more difficult to convert these premises into appropriate actionable questions that have potentially been studied before, such as "can Adobe Flash do cross-domain communication". A methodology/tool to help generate these questions has a great value.

5.2.3. Potential mitigations to consider

When a system is complex, developers make mistakes. This is especially true for integrations of multiple services involving different companies. Miscommunications is a common cause of logic flaws. We believe that good mitigations should provide a good control of the system complexity and/or minimize website developers'

programming load for integration. For example, the following two directions are worth consideration.

Using dedicated (or simplified) runtimes to replace the general-purpose web platform. There are reasons for the general-purpose web platform to be prefered, e.g., (1) every user knows how to use a browser; (2) web programming skill is readily available in the job market. However, from security standpoint, such a general platform is difficult to examine exhausively. API designers may not be aware of certain browser capabilities, which can lead to vulnerable implementation and open the avenue to potential exploits.

Admittedly, some serious attempts were made many years ago for security schemes not based on the web platform. However, they did not get real tractions in the market. For example, *Secure Electronic Transaction* (SET) [35] was a payment protocol which many big companies contributed to. It was designed at the same time when SSL was emerging, so some of SET's security goals competed with SSL. Eventually, the payment schemes widely deployed are PayPal, Amazon Payments, Google Checkout, etc, which are based on SSL and the general-purpose web technology. Another example is the InfoCard Sign-On scheme [4], introduced by Microsoft since Windows Vista. The client is a dedicated application named "Windows CardSpace". InfoCard was not widely adopted before it was retired. The SSO schemes really adopted are those that we analyzed in this study. These unsuccessful attempts suggest that web-based schemes indeed have a clear advantage for deployment. On the other hand, our paper shows that the easy deployment comes with the cost of significant security uncertainty. Therefore, a possible mitigation might be to build a simplified web platform for running security schemes. The programming language is still HTML with Javascript, but its functionalities are so restricted that the system details of the platform can be faithfully modeled.

Delivering security-critical services as "integrated circuits", not as "individual electronic components". Today the APIs of service providers (e.g., IdP and cashiers) are designed at a level which is too low. Integrating these APIs into a website is like wiring up many electronic components to implement a circuit. There is too much room for mistakes. We believe that it is better for the services to be provided as "integrated circuits". A potential argument in favor of "individual electronic components" is that they give flexibility to website developers. However, we argue that it is service providers' job to understand the level of flexibility that developers want, and build "integrated circuits" for them, but do not allow developers to abuse the flexibility. Website developers' task should be minimized: they only need to choose an integration scenario, include the corresponding library from the service provider, and make a single library call to do the whole work.

6. RELATED WORK

Research related to web SSO security covers many topics, including users' misconceptions about OpenID [31], chances for phishing attacks [29], and various privacy concerns [30][32]. Our work is focused on the type of SSO security flaws that totally defeats the purpose of authentication – the attacker signing in as the victim user.

The protocol analysis community developed frameworks and tools to model and examine many security protocols. Some classic approaches and tools include Millen's model [27], the NRL Protocol analyzer [25] and the BAN logic [9]. There are also specific studies about web SSO protocols, such as several protocols based on SAML (Security Assertion Markup Language) [28]. Groß's work attempted to formalize the SAML Single Sign-on Browser/Artifact Profile [20]. It found three protocol weaknesses based on the assumptions of an attacker being able to intercepting protocol traffic or spoofing DNS servers. Pfitzmann and Waidner discovered a protocol flaw in a protocol called Liberty-Enabled Client and Proxy Profile, which is also SAML-based. Hansen et al also used a static analysis approach to automatically analyze the SAML SSO protocol [21]. In 2008, Armando et al formally modeled SAML 2.0 Web Browser SSO Profile, and used an LTL (Linear Temporal Logic) model checker that the authors developed, namely SATMC, to discover an authentication flaw [2]. The practical consequence of the flaw was significant because the SAML-based SSO for Google Apps was an instantiation of the vulnerable protocol, thus Google Apps suffered from the vulnerability. Bhargavan et al used an automated theorem prover to prove certain security properties of InfoCard protocol [4]. Our work is complementary to protocol verification techniques in a number of aspects: (1) the primary motivation of our work is to do a "field study" about real SSO deployments, so our analyses starts with real systems, not documented protocols; (2) the key output of our analyses include semantics of message elements, server-side protections (e.g., whitelisting), important system assumptions that an SSO scheme relies on (e.g., same-domain communication) and how an RP consumes data from the IdP. A protocol verifier would need such analysis result as necessary input.

Research papers about SSO analysis also pointed out another type of vulnerabilities, which cause an opposite consequence, i.e., the victim user unknowingly signing in as the attacker. For example, Akhawe modeled WebAuth SSO in Alloy and used a model checker to find a flaw of this type [1]; in reference [3], Armando et al extended their previous model described in [2] and discovered such a flaw in the SAML-based SSO for Google Apps.

In Section 5.2, we summarized the similarity between this SSO study with our earlier study about logic flaws on merchants' integrations of cashier services [34]. The two studies, however, differ in two aspects: (1) most logic flaws in reference [34] were identified using merchants' source code; (2) reference [34] only considered the situation that the client is malicious, which is our scenario (A). Another related research direction is black-box security testing for web systems. For example, NoTamper [6] is a technique that tests if the client-side logic of a web app is duplicated on the server side, without access to the server source code.

It was not designed to find logic flaws in service integrations like SSO schemes.

Protocol reverse engineering has been studied for a while, e.g, [11]. Different from the prior research that focuses on recovering the message format of an unknown protocol, our aim is to identify the semantics of the HTTP fields in SSO BRMs and their relations.

7. CONCLUSIONS

In this paper, we report an extensive security study of commercial web SSO systems. The study shows that security-critical logic flaws pervasively exist in these systems, which can be discovered from browser-relayed messages and practically exploited by a party without access to source code or other insider knowledge of these systems. We elaborate our analysis steps performed on commercial systems and how they lead to discoveries. Every discovered flaw allows the attacker to sign in as the victim. The affected companies all acknowledged the importance of our findings, and expressed their gratitude in various ways.

In addition to those reported, we are discovering and confirming new flaws in other web SSO systems. This suggests the seriousness of the overall situation. Clearly the scale of the problem is beyond what we can cover as a single research team, so we wish this paper can be a call for a collaborative effort of the SSO community. The service that we will launch soon at http://sso-analysis.org enables developers and security analysts to conduct investigations similar to what we did. Such a collaborative study hopefully helps the community better understand security challenges in web SSO deployments and identify suitable solutions.

ACKNOWLEDGEMENT

We thank our shepherd Alex Halderman for valuable suggestions on the improvement of the paper. We also thank Zhou Li for pointing us to the Unpredictable Domain Communication of Adobe Flash, and Manuel Caballero and David Ross for referrer-free posting examples. We appreciate the comments from Martín Abadi, Shaz Qadeer, Nik Swamy and Helen Wang on the early draft of the paper, and the discussions with Cormac Herley and Yi-Min Wang. Authors with Indiana University were supported in part by the NSF Grants CNS-1017782 and CNS-1117106. Rui Wang was also supported in part by a Microsoft Research internship.

REFERENCES

[1] Devdatta Akhawe, Adam Barth, Peifung Lam, John Mitchell, Dawn Song. "Towards a Formal Foundation of Web Security," IEEE Computer Security Foundations Symposium, 2010

[2] Alessandro Armando, Roberto Carbone, Luca Compagna, Jorge Cuellar, Llanos Abad. "Formal Analysis of SAML 2.0 Web Browser Single Sign-On: Breaking the SAML-based Single Sign-On for Google Apps," ACM FMSE, 2008

[3] Alessandro Armando, Roberto Carbone, Luca Compagna, Jorge Cuellar, G. Pellegrino, A. Sorniotti. "From Multiple Credentials to Browser-based Single Sign-On: Are We More Secure?" IFIP Information Security Conference (SEC), 2011

[4] Karthikeyan Bhargavan, Cédric Fournet, Andrew D. Gordon, Nikhil Swamy. "Verified implementations of the information card federated identity-management protocol, ACM ASIACCS 2008.

[5] Adam Barth, Collin Jackson, and John C. Mitchell. "Robust Defenses for Cross-Site Request Forgery," ACM CCS, 2008

[6] Prithvi Bisht, Timothy Hinrichs, Nazari Skrupsky, Radoslaw Bobrowicz and V.N. Venkatakrishnan. "NoTamper: Automatically Detecting Parameter Tampering Vulnerabilities in Web Applications," ACM CCS 2010

[7] Blue Research. "Consumer Perceptions of Online Registration and Social Sign-In," http://janrain.com/consumer-research-social-signin

[8] Andrew Bortz, Adam Barth, and Alexei Czeskis. "Origin Cookies: Session Integrity for Web Applications," W2SP 2011.

[9] Michael Burrows, Martín Abadi, and Roger Needham. A logic of authentication. ACM Trans. Computer Systems 8, 1, 18-36. 1990.

[10] Hao Chen, David Wagner and Drew Dean. "Setuid demystified," USENIX Security Symposium, San Francisco, CA, August 2002

[11] Weidong Cui, Jayanthkumar Kannan, Helen J. Wang. "Discoverer: Automatic Protocol Reverse Engineering from Network Traces," USENIX Security Symposium 2007

[12] Facebook. "White hats," http://www.facebook.com/whitehat

[13] Facebook. "OAuth Dialog," http://developers.facebook.com/docs/reference/dialogs/oauth/

[14] Facebook Developers. "Legacy Canvas Auth," http://developers.facebook.com/docs/authentication/fb_sig/

[15] Fiddler Web Debugger. http://www.fiddler2.com/fiddler2

[16] Firebug. http://getfirebug.com/

[17] About Freelancer. http://www.freelancer.com/info/about.php

[18] Google. "Security Hall of Fame," http://www.google.com/about/company/halloffame.html

[19] Google Code. "Federated Login for Google Account Users," http://code.google.com/apis/accounts/docs/OpenID.html

[20] Thomas Groß. "Security analysis of the SAML single sign-on browser/artifact profile," ACSAC 2003

[21] S. M. Hansen, J. Skriver, and H. R. Nielson. "Using static analysis to validate the SAML single sign-on protocol," Workshop on Issues in the Theory of Security, 2005

[22] Brian Kissel. "OpenID 2009 Year in Review," http://openid.net/2009/12/16/openid-2009-year-in-review/

[23] LocalConnection (in flash.net). http://help.adobe.com/en_US/FlashPlatform/reference/actionscript/3/flash/net/LocalConnection.html?filter_flex=4.1&filter_flashplayer=10.1&filter_air=2

[24] Los Angeles Times. "The Sims Social bests FarmVille as the second-largest Facebook game," http://latimesblogs.latimes.com/entertainmentnewsbuzz/2011/09/sims-social-surpasses-farmville-as-second-largest-facebook-game.html

[25] Catherine Meadows. "Language Generation and Verification in the NRL Protocol Analyzer," Computer Security Foundations 1996.

[26] Microsoft. "INFO: Internet Explorer Does Not Send Referer Header in Unsecured Situations," http://support.microsoft.com/kb/178066

[27] Jonathan K. Millen. "The Interrogator Model," IEEE Symposium on Security and Privacy 1995.

[28] OASIS Standard. Assertions and Protocols for the OASIS Security Assertion Markup Language (SAML) V2.0, 2005.

[29] OpenID Wiki. "OpenID Phishing Brainstorm," http://wiki.openid.net/w/page/12995216/OpenID_Phishing_Brainstorm

[30] Birgit Pfitzmann and Michael Waidner. "Analysis of Liberty Single-Sign-on with Enabled Clients," IEEE Internet Computing, 7(6) 2003.

[31] San-Tsai Sun, Eric Pospisil, Eric Pospisil, Ildar Muslukhov, Nuray Dindar, Kirstie Hawkey, Konstantin Beznosov. "What Makes Users Refuse Web Single Sign-On? An Empirical Investigation of OpenID," Symposium On Usable Privacy and Security, 2011

[32] Manuel Uruena and Christian Busquiel. "Analysis of a Privacy Vulnerability in the OpenID Authentication Protocol," IEEE Multimedia Communications, Services and Security, 2010.

[33] Rui Wang, Shuo Chen, XiaoFeng Wang. "Signing Me onto Your Accounts through Facebook and Google: a Traffic-Guided Security Study of Commercially Deployed Single-Sign-On Web Services". http://www.informatics.indiana.edu/xw7/papers/websso.pdf. Supporting materials: http://research.microsoft.com/~ruiwan/sso/supp

[34] Rui Wang, Shuo Chen, XiaoFeng Wang, and Shaz Qadeer. "How to Shop for Free Online – Security Analysis of Cashier-as-a-Service Based Web Stores," IEEE Symposium on Security and Privacy, 2011

[35] Wikipedia, "Secure Electronic Transaction," http://en.wikipedia.org/wiki/Secure_Electronic_Transaction