Explicating SDKs: Uncovering Assumptions Underlying Secure Authentication and Authorization

Rui Wang (MSR Redmond) *
Yuchen Zhou (University of Virginia) *
Shuo Chen (MSR Redmond)
Shaz Qadeer (MSR Redmond)
David Evans (University of Virginia)
Yuri Gurevich (MSR Redmond)

Technical Report
MSR-TR-2013-37

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052

March 19th, 2013
(Revised June 7, 2013)

* The two lead authors are alphabetically ordered.

Explicating SDKs: Uncovering Assumptions Underlying Secure Authentication and Authorization

Rui Wang[†][*], Yuchen Zhou[‡][*], Shuo Chen[†], Shaz Qadeer[†], David Evans[‡], Yuri Gurevich[†]

[†] Microsoft Research
Redmond, WA, USA
{ruiwan, shuochen, qadeer, gurevich}@microsoft.com

[‡] University of Virginia
Charlottesville, VA, USA
{yz8ra, evans}@virginia.edu

Abstract

Most modern applications are empowered by online services, so application developers frequently implement authentication and authorization. Major online providers, such as Facebook and Microsoft, provide SDKs for incorporating authentication services. This paper considers whether those SDKs enable typical developers to build secure apps. Our work focuses on systematically explicating implicit assumptions that are necessary for secure use of an SDK. Understanding these assumptions depends critically on not just the SDK itself, but on the underlying runtime systems. We present a systematic process for identifying critical implicit assumptions by building semantic models that capture both the logic of the SDK and the essential aspects of underlying systems. These semantic models provide the explicit basis for reasoning about the security of an SDK. We use a formal analysis tool, along with the semantic models, to reason about all applications that can be built using the SDK. In particular, we formally check whether the SDK, along with the explicitly captured assumptions, is sufficient to imply the desired security properties. We applied our approach to three widely used authentication/authorization SDKs. Our approach led to the discovery of several implicit assumptions in each SDK, including issues deemed serious enough to receive Facebook bug bounties and change the OAuth 2.0 specification. We verified that many apps constructed with these SDKs (indeed, the majority of apps in our study) are vulnerable to serious exploits because of these implicit assumptions, and we built a prototype testing tool that can detect several of the vulnerability patterns we identified.

1 Introduction

Modern applications commonly consist of a client program and an online service that provides functionality such as cloud storage, social networking, and geographic data. Accessing the service requires authentication of users and authorization of resource requests. Traditionally, the authentication and authorization mechanisms were provided by operating systems and carefully implemented in a few core apps such as SSH, remote desktop, etc; with modern apps, however, many developers end up needing to implement such mechanisms. To aid this, major identity providers have developed SDKs that developers can use to integrate authentication and authorization into their apps such as the three SDKs we study in this work: the Facebook PHP SDK, the Microsoft Live Connect SDK, and the Windows 8 Authentication Broker SDK. According to our sampling of popular apps in Windows App Store, 52% of them use these SDKs (see Appendix A).

Authentication/authorization SDKs are becoming a critical foundation for apps. However, no previous study has rigorously examined the security these SDKs provide to real-world apps. Typically, SDK providers simply release SDK code, publish documentation and examples, and leave the rest to app developers. An important question remains: *if developers use the SDKs in reasonable ways, will the resulting applications be secure?* We show in this paper that the answer today is "No". The majority of apps built using the SDKs we studied have serious security flaws. This is not due to direct vulnerabilities in the SDK, but rather because achieving desired security properties by using an SDK depends on many implicit assumptions that are not readily apparent to app developers. These assumptions are not documented anywhere in the SDK or its developer documentation. In several cases, even the SDK providers are unaware of the assumptions (see Section 5.2).

The goal of our work is to systematically identify the assumptions needed to use an SDK to produce secure applications. We emphasize that it is not meaningful to verify an SDK by itself. Instead, our goal is to explicate the assumptions upon which secure use of the SDK depends. We do this by devising precise definitions of desired security properties, constructing an explicit model of the SDK and the complex services with which it interacts, and systematically exploring the space of applications that can be built using the SDK. Our approach involves a combination of manual effort and automated formal verification. Any counterexample found by the

[*] The two lead authors are ordered alphabetically.

verification tool indicates either (1) that our system models are not accurate, in which case we revisit the real systems to correct the model; or (2) that our models are correct, but additional assumptions need to be captured in the model and followed by application developers. The explication process is an iteration of the above steps so that we document, examine and refine our understanding of the underlying systems for an SDK. At the end, we get a set of formally captured assumptions and a semantic model that allow us to make meaningful assurances about the SDK: *an application constructed using the SDK following the documented assumptions satisfies desired security properties.*

We argue that explication should be part of the engineering process of developing an SDK. Identified SDK assumptions can either be removed by modifying the SDK, or be documented precisely. In addition, in some cases it is feasible to develop automatic tests that detect common ways applications violate the assumptions (we provide an example in Section 6.2).

Results. The work presented in this paper reflects a 12 person-month effort (six months of two lead authors) in systematically explicating the three target SDKs. The resulting models (https://github.com/sdk-security) are publicly released so that the community can review and enhance them. As a result of the explication process, we uncovered many SDK assumptions (summarized in Section 5). Some assumptions were especially serious because they can be violated when an app developer has a reasonable alternative interpretation of the developer's guide (*dev guide*) or when an app runs on certain realistic platforms. These reports were treated very seriously by the SDK providers: five cases that we reported to Facebook have been fixed (three of which were rewarded by Facebook bounties [14]). An issue uncovered in the Live Connect SDK resulted in Microsoft improving its dev guide. Our report to the OAuth Working Group convinced the group to add a subsection to the OAuth 2.0 draft.

With all the SDK assumptions specified, we were able to successfully verify all the models with the uncovered assumptions (Section 4). Uncovering these SDK assumptions also enables effective app testing since a violation of an assumption often leads to a successful exploit. Our study shows that many released apps are indeed vulnerable due to violations of these assumptions. We tested three sets of apps, including client apps in Windows 8 App Store and service apps using Facebook sign-on, and found that 78%, 86% and 67% of these apps suffer from vulnerabilities related to the implicit assumptions we uncovered (Section 6.2).

2 Illustrative Example

To motivate our work, we describe a simple example in the context of the Live Connect SDK. It illustrates what can go wrong when SDKs are provided without thoroughly specifying their underlying security assumptions.

2.1 Intended Use

Suppose we want to develop an app using Live ID as the Identity Provider (IdP). We start with the dev guide for Live Connect [25]. The hyperlinks in the start page lead us to a page of detailed instructions about "signing users in" [26] which provides code snippets in Javascript, C#, Objective-C and Java showing how to use Live Connect SDK to sign users into a client app. Ignoring the specifics in these different languages, all the code snippets essentially cover the authentication logic shown in Figure 1.

In the figure, WL stands for "Windows Live". A developer first needs to call WL.login. The call takes an argument value, "wl.basic", indicating that the app will need to read the user's basic information after WL.login returns an access token in step (2). The access token is a concept in the OAuth protocol [22]. It is an opaque string dynamically created by the Live ID server for each call to WL.login. Once the app gets the access token, it calls the REST API me to get the user's basic info using this HTTP request:

```
https://apis.live.net/v5.0/me?access_token=ACCESS_TOKEN
```

The Live ID service responds with the user's basic information in message (4), such as her full name and user ID. This completes the process, authenticating the user with the provided information.

2.2 Hazardous Use

The developer guide as depicted in Figure 1 is valid for a client-only app, but it does not make it clear that the same logic *must not* be used with an app that also incorporates an online service. Without stating this explicitly, developers may be inclined to use the SDK insecurely as shown in Figure 2. The interactions with the Live ID

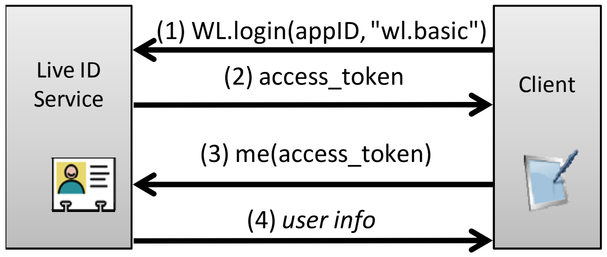


Figure 1. Authentication Logic for "Signing Users In".

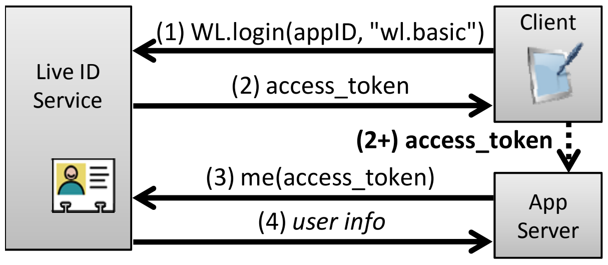


Figure 2. Hazardous Use.

service are identical in the two figures. The only difference is that in the second scenario, the access token is sent to the service app (i.e., the server side of the app) in message (2+) and it is the service app that calls me to authenticate the user.

This can lead to a serious vulnerability that allows any app on the device to sign into the service app as the user. The rogue app sends a request to the Live ID service for an access token to view public information for the victim, such as a profile record on Facebook. Live ID responds with an access token. The problem is this token, intended for authorizing access to the public resource, is mistakenly used by the service app to authenticate its owner as the victim. This allows the rogue app to get into the victim's account on the service app. This mistake is fairly common in real-world apps. Although we first observed it analyzing the Live Connect SDK, we later found that many apps using the Facebook SDK have the same issue. As described in Section 6.2, we tested 27 apps in the Windows 8 App Store and found that 21 of them are vulnerable due to this mistake.

2.3 Resolution

From one perspective, this is simply a matter of developers writing buggy apps, and the blame for the security vulnerability rests with the app developers. We argue, though, that the purpose of the SDK is to enable typical developers to produce apps that use authentication and authorization in a way that provides desired security properties, and the prevalence of buggy apps created using this SDK indicates a failure of the larger engineering process. The developer exercised reasonable prudence by using the access token to query the ID service for user information and followed exactly the process described in the SDK's documentation (depicted in Figure 1). The problem is lack of a deeper understanding of the relationship between authentication and authorization, and the role of the access token (i.e., why is it safe to use the access token as shown in Figure 1 but not as used in Figure 2). Correct use depends on subtle understanding of what kind of evidence each message represents and whether or not the whole message sequence establishes an effective proof for a security decision. It is unrealistic to expect most developers to understand these subtleties, especially without clear guidance from the SDK.

We contacted the developers of some of the vulnerable apps. A few apps have been fixed in response to our reports. We also notified the OAuth Working Group (WG) in June 2012 about these vulnerable apps.[1] Dick Hardt, editor of OAuth 2.0 specification (RFC 6749) [22], emailed us requesting language to be included in the specification dealing with this issue. We proposed the initial text and discussed with WG members. This resulted in Section 10.16 "*Misuse of Access Token to Impersonate Resource Owner in Implicit Flow*" being added to the OAuth specification.

The key point this example illustrates is that security of apps constructed with an SDK depends on an understanding of the external services the app depends on, as well as subtleties in the use of tokens and assumptions about evidence used in authentication and authorization decisions. We believe the prevalence of vulnerable apps constructed using current SDKs is compelling evidence that a better engineering process is needed, rather than just passing the blame to overburdened developers. In particular, we advocate for a process that explicates SDKs by systematically identifying the underlying assumptions upon which secure usage depends.

3 Explicating SDKs

In order to explicate the SDKs, we need to clearly define the desired security properties. This section introduces our target scenario and threat model, and then defines the desired security properties and overviews our process for uncovering implicit SDK assumptions.

3.1 Scenario

A typical question about security is whether some property holds for a system, even in the presence of an adversary interacting with the system in an unconstrained manner. We can view this as a software testing problem: the system is a concrete program, while the adversary is an abstract one (i.e., a *test harness* in the terminology of software testing) that explores all interaction sequences with the concrete system. In our scenario, however, the target system is not concrete. We wish to reason about *all* applications that can be built with the SDK follow-

[1] Subsequently, we learned that John Bradley, a WG member, had posted a blog post in January 2012 about a similar issue [10]. The post considers the problem a vulnerability of the protocol, while we view it as a consequence of an unclear assumption about SDK usage because there are correct ways to use OAuth for client+service authentication.

ing documented guidelines. Hence, we need to consider both the client app and service as abstract modules.

Figure 3 illustrates the modules in our setup. There are three main components: a client device, the application server *foo.com*, and the identity provider (IdP). The bottom layer of the client device is the client runtime, such as the HTML engine or the HTTP layer. The middle layer is the client SDK. The client app, FooApp_C, is created by the developer to interact with the application server. We assume FooApp_C always uses the client SDK for authentication and authorization. Like the client, the application server has three layers: the service runtime represents the server platform, such as PHP or ASP.NET; the server side of the SDK we study; and the application server code. We assume that FooApp_S does not directly interact with the service runtime, but only uses it via the service SDK. Note that both FooApp_C and FooApp_S identify themselves to IdP as "FooApp" with an app ID pre-assigned by IdP. The IdP cannot tell if the caller is a client or the application server.

The modules with brick pattern backgrounds are *concrete modules* with concrete implementations. They can be divided into two layers. *The SDK layer* consists of the Client SDK and the Service SDK. The *underlying system layer* consists of the client runtime, the service runtime, and the IdP. These are complex modules that one typically does not understand in detail in the beginning of the study. Developing a semantic model for these components involves substantial systems investigation effort (as described in Section 4.3) because the seemingly clear SDK logic actually depends on a much more mysterious (and often incompletely documented) underlying layer. We consider the formal semantic models resulting from this study as one of the main contributions of this work.

The client and server application modules are *abstract modules*. They do not have concrete implementations: our goal is to reason about all possible apps built using the SDK. Nevertheless, the app modules do have constraints on their behaviors: FooApp_C and FooApp_S are only allowed to use the target SDKs for authentication and authorization, and must not violate rules documented in the SDK developer guides.

3.2 Threat Model

We want to reason about security properties of *all* apps that could reasonably be constructed with the SDK. We assume a malicious application, MalApp_C, may be installed on the user's device. MalApp_C's behavior is not constrained by the client SDK, but it is limited to functionality provided by the client runtime (e.g., it cannot access cookies of other domains or handcraft HTTP requests). The attacker also controls an unconstrained external machine, which we call "Mallory". As shown in Figure 4, we can think of Mallory as a combination of a client and server that can freely communicate with the client, application server, and IdP. We model MalApp_C and Mallory as abstract modules.

3.3 Security Properties

Our analysis depends on a formal definition of the security properties the SDK is intended to provide.

Granularity: session. Informally, people often say things like "a client is authenticated as Alice", or "a server is authorized on Alice's behalf". However, it is important to point out explicitly that it is not the client or the server, but the *session* between them, that is authenticated or authorized. More specifically, the end result of an authentication/authorization protocol between a client and a server is to know whom *the session* represents and what *the session* is allowed to do. It should not affect the identity or permission of any other session. Therefore, we always keep the session (identified by its session ID) explicit in our modeling.

Basis of security: secrets and signed data. All mechanisms we study share a commonality: they use secrets or data signed by the identity provider as unforgeable evidence to differentiate some entities from others. These secrets and signed data are either preconfigured or generated at runtime at the underlying system layer.

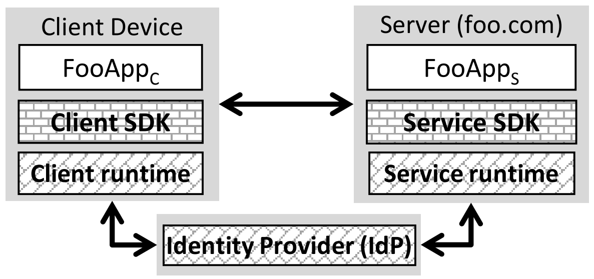


Figure 3. Modules in Client+Service App.

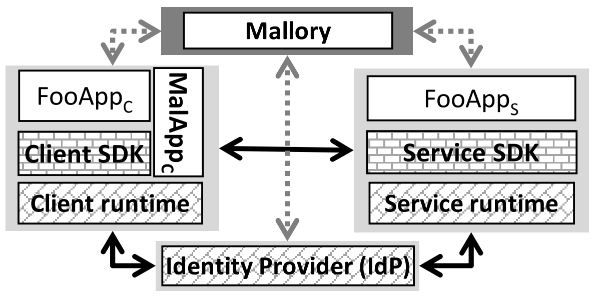


Figure 4. Threat Model.

We distinguish five types of secrets in the studied SDKs: access tokens, Codes[2], refresh tokens, app secrets and session IDs. The first four are protocol data in OAuth, which we will explain in later examples. The only identity-provider-signed data we have seen are *signed requests*, defined by Facebook, and *authentication tokens*, defined by Live ID. They are signed data structures containing some or all of the following data: access token, Code, app ID and user ID.

The desired security properties, therefore, need to consider what data the adversary may have obtained. This is made explicit by adding a *knowledge pool* to the model. All secrets and signed data received by the attacker are recorded in the knowledge pool, and can be used by the attacker in all subsequent actions.

Desired security properties. We define the security goal of the authentication/authorization SDKs based on the protections they provide to apps. Apps written using the SDK following explicit programming guidelines should be protected from the following violations:

(1) *Authentication violation*. If some knowledge, *k*, is about to be added to the pool, and *k* is sufficient to convince the authentication logic of FooApp_S that the knowledge holder is Alice, it implies that Mallory (and MalApp_C, since they share the knowledge pool) can authenticate as Alice, which is an authentication violation.

(2) *Authorization violation*. Depending on the type of *k*, there are two kinds of authorization violations. If *k* is Alice's access token, Alice's Code, or the session ID for the session between FooApp_C and FooApp_S, it implies that Mallory has obtained the permission to do whatever the session can do. Another authorization violation is when *k* is the app secret of FooApp. This would allow Mallory to do whatever FooApp_S can do on the identity provider.

(3) *Association violation*. The ultimate goal of authentication/authorization is not only to know who the user is or what she can do, but to correctly bind three pieces of data: the user's identity (i.e., the authentication result), the user's permissions (i.e., the authorization result), and the session's identity (usually known as session ID). This association is actually the end result of authentication/authorization and is what the application logic depends on after the process is accomplished. Mistakes in the association (such as binding Mallory's identity to Alice's permission, or binding Alice's identity to Mallory's session) are security violations.

[2] To avoid confusion with other meanings of "code", such as "source code", we always capitalize the first letter to refer to the "OAuth Code" in this paper.

3.4 The Process of Explicating SDKs

Figure 5 rearranges the modules (from Figure 4) and combines the concrete modules one each layer into one. The dashed line between abstract and concrete modules represents the interface between the test harness and the target system. The essential question is: *what assumptions are necessary for FooApp to achieve the desired security properties*?

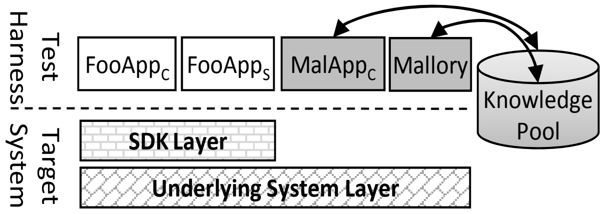
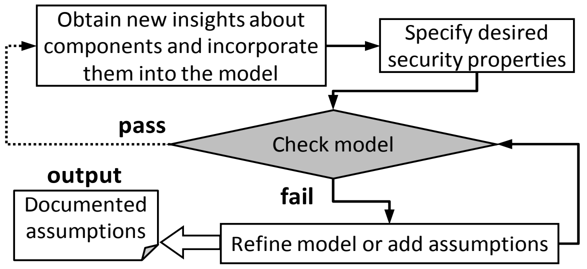


Figure 5. Modules Rearranged for Explicating.

Explicating SDKs is a systematic investigation effort to explicitly document our knowledge about these modules and examine the knowledge against defined security goals. As shown in Figure 6, it is an iterative process, in which we repeatedly refine our model and formally check if it is sufficient to establish the security properties or additional assumptions are needed. A failed check (i.e., a counterexample in the model) indicates either that our understanding of the actual systems needs to be revisited or that additional assumptions are needed to ensure the desired security properties.

The outcome of the process is the assumptions we explicitly added to the model. In Section 5.2, we show that many of the uncovered assumptions can indeed be violated in realistic situations.


Obtain new insights about components and incorporate them into the model
Specify desired security properties
pass
Check model
output
Documented assumptions
fail
Refine model or add assumptions
Figure 6. Engineering Process for Explicating SDKs.

4 Semantic Modeling

This section gives an overview of the semantic modeling effort for the three SDKs. The resulting models are available at *https://github.com/sdk-security/*. They reflect six months of effort by our two lead authors (i.e., 12 person-months) in creating and refining the system models.

4.1 Modeling language

To specify the semantics of the modules, we want a language that has a suitable formal analysis technology for verification. In the first period of our investigation, we used Corral [24], a property checking tool that can perform bounded verification on a C program with embedded assertions. Corral explores all possible execution paths within a bound to check if the assertions can be violated. Later, we re-implemented all the models in Boogie [9], a language for describing proof obligations that can then be tested using an SMT solver, which allowed us to fully prove the desired properties. This provides a higher assurance than the bounded verification done by Corral, but the basic ideas and approach are the same for both checking strategies. For concreteness, this section describes the Boogie version to explain our modeling.

The key Boogie language features needed to understand this paper are:

- The * symbol represents a non-deterministic Boolean value.
- HAVOC *v* is a statement that assigns a non-deterministic value to variable *v*.
- ASSERT(*p*) specifies an assertion that whenever the program gets to this line, *p* holds.
- ASSUME(*p*) instructs Boogie to assume that *p* holds whenever the program gets to this line.
- INVARIANT(*p*) specifies a loop invariant. Boogie checks if *p* is satisfied at the entry of the loop, and inductively prove *p*'s validity after each iteration.

If Boogie fails to prove an assertion or an invariant, it reports a counter-example. This leads us to refine the model, adding assumptions when necessary.

4.2 Modeling abstract modules

The test harness interacts with the concrete modules in a non-deterministic manner. It implements the abstract modules representing both the unknown (benign) application and the attacker's resources. The test harness consists of a loop with the loop count depth. Each iteration calls the function TestHarnessMakesCall. This function is implemented as a non-deterministic switch (i.e., a statement of "switch(*){…}") that chooses to call FooApp$_c$Runs, MalAppcMakesCall, or MalloryMakesCall. Eventually, through a series of non-deterministic choices as shown in Figure 7, one of the functions in a concrete module will be called.

Using the knowledge pool. As mentioned in Section 3.3, we use a knowledge pool to model the information obtained by an attacker. Different types of knowledge, such as access tokens, Codes, and session IDs, are explicitly differentiated. We do not consider attacks that involve providing arguments of the incorrect type, e.g., giving a session ID to a function expecting an access token. There is an AddKnowledge function for each knowledge type. After each call to MalApp$_C$MakesCall and MalloryMakesCall, the function AddKnowledge_*Type* is called to add any acquired knowledge to the pool. There is a corresponding DrawKnowledge_*Type* function for non-deterministically drawing knowledge of a particular type from the knowledge pool. It is implemented using HAVOC *i*, where *i* is the array index of the piece of knowledge non-deterministically chosen.

4.3 Modeling concrete modules

Concrete modules do not have any non-determinism. The key aspects of building semantic models for the concrete modules are summarized below.

Data types. The basic data types in the models are int and several types for enumerables. We also define structs and arrays over the basic types. In the actual systems, the authentication logic is constructed using string operations such as concatenation, tokenization, equality comparison, and name/value pair parsing. We found that most string values are essentially enumerable, except those of domain names and user names, which we canonicalize as *Alice*, *Mallory*, *foo.com*, *mallory.com*, etc. Thus, the basic types, structs, and arrays are sufficient to model data used in the concrete modules.

SDKs. The sizes of these SDKs are moderate (all under 2000 lines) and their source code is public. The SDKs we modeled were implemented in HTML, JavaScript and PHP, so we needed to first translate the SDKs function-by-function into Boogie. We do this translation manually, but it is not hard to imagine tools that could mostly automate it. Table 1 shows two functions in the Facebook PHP SDK and our corresponding Boogie procedures. For getUserFromAvailableData, the changes are essentially line-by-line translations. For getLogoutUrl, the PHP code performs a string operation and re-

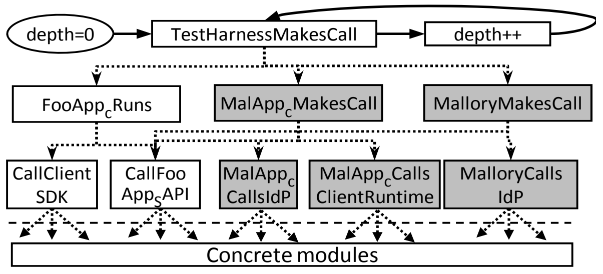


Figure 7. Test Harness.
(Dotted lines represent non-deterministic choices.)

```
protected function getUserFromAvailableData() {
  if ($signed_request) {
    ...
    $this->setPersistentData('user_id',
                                    $signed_request['user_id']);
    return 0;
  }
  $user = $this->getPersistentData('user_id', $default = 0);
  $persist_token =
        $this->getPersistentData('access_token');
  $access_token = $this->getAccessToken();
  if ($access_token &&
      !($user && $persist_token == $access_token)) {
          $user = $this->getUserFromAccessToken();
          if ($user)
            $this->setPersistentData('user_id', $user);
          else $this->clearAllPersistentData();
    }
    return $user;
}

public function getLogoutUrl() {
    return $this->getUrl(
        'www',  'logout.php',
        array_merge(array(
            'next' => $this->getCurrentUrl(),
            'access_token' => $this->getAccessToken(), ), …));
  }
```

```
procedure  {:inline 1} getUserFromAvailableData()  returns (user:User) {
   if (IdP_Signed_Request_Records__user_ID[signed_request] != _nobody) {
          ...
          user := IdP_Signed_Request_Records__user_ID[signed_request];
          call setPersistentData__user_id(user);
          return;
   }
   call user := getPersistentData__user_id();
   call persisted_access_token := getPersistentData__access_token();
   call access_token := getAccessToken();
   if (access_token >= 0 &&
       !(user != _nobody && persisted_access_token == access_token)) {
            call user := getUserFromAccessToken(access_token);
             if (user != _nobody) {
               call setPersistentData__user_id(user);
             } else {
               call clearAllPersistentData();
             }
   }
   return;
}
procedure {:inline 1} getLogoutUrl()
  returns (API_id: API_ID, next__domain: Web_Domain, next__API: API_ID,
             access_token: int)  {
       API_id := API_id_FBConnectServer_login_php;
       call access_token := getAccessToken();
       call  next__domain,  next__API := getCurrentUrl();
}
```

Table 1. Example of a PHP function and its Boogie model.

turns a string. Our Boogie translation in this case is not obviously line-by-line. For example, our procedure returns a four-element vector instead of a string. The PHP function calls getUrl and array_merge, which concatenate substrings, therefore, are implicitly modeled by the four-element return vector.

Underlying system layer. Unlike the SDK, which is simple enough to model completely, the identity provider, client runtime, and server runtime are very complex and may not even have source code available. Completely modeling every detail of these systems is infeasible, but our analysis depends on developing suitable models of them. By studying the target SDKs, we identified three aspects of these systems that need to be carefully understood to perform verification. These aspects are the basis for the security goals the SDKs are designed to achieve:

(1) The identity provider's behaviors according to different input arguments and various app settings in its web portal. Each identity provider has a web page for app developers to enter a number of app settings that the identity provider needs to know, such as app ID, app secret, service website domain, and return URL. Many of these settings are critical for the identity provider's decision-making. Further, different inputs to the provided APIs cause different responses. Because we do not have the source code for the identity providers, we tested these behaviors by constructing different requests and app settings. For example, in the models we've built, the identity provider APIs dialog_permissions_request(), RST2_srf() and oauth20_authorize_srf()[3] involved 11, 8 and 6 if-statements respectively, to describe different behaviors we observed in testing.

(2) Data passing on the client runtime. As with the identity providers, we do not have access to source code to understand detailed behaviors of the client runtime. Our models were based on observations made during testing. We focused on the client's decision-making about passing data from one server to another (by redirection), delivering data to FooApp$_C$ or MalApp$_C$, and attaching cookies to outgoing requests. These decisions are important for security. We maintain a cookie structure for each client app, i.e., FooApp$_C$ or MalApp$_C$. The cookie structure contains a session ID field and some optional fields specific to the SDK, such as *signed_request* and *authentication_token*.

(3) Sessions, requests, and cookies on the service runtime. In our model, the service runtime is a layer that defines data structures for sessions, requests and cookies for the service SDK and FooApp$_S$. (Note that although cookies are in the headers of requests, we separate them to flatten the data structure.) The cookie structure is the same as previously described. The request structure is defined according to the SDK's specifica-

[3] The APIs are accessed as *https://www.facebook.com/dialog/permissions.request*, *https://login.live.com/RST2.srf*, and *https://login.live.com/oauth20_authorize.srf*

tion. For example, requests for the Facebook PHP SDK use a structure containing a Code, a state and an optional *signed_request*. The session structure contains a session ID and a collection of session variables (keys) defined by the SDK.

4.4 Security assertions

We use ASSERT statements to document and test the desired security properties, covering each of the security violations described in Section 3.3.

Authentication violation. An authentication violation occurs when an attacker acquires some knowledge that could be used to convince $FooApp_S$ that the knowledge holder is Alice. A simple example is the case we described in Section 2.2, in which the knowledge is an access token. In addition to access tokens, we also consider IdP-signed data such as Facebook's signed messages or Live ID's authentication tokens. To detect these violations, when a Facebook Signed Request *k* is added to the knowledge pool, we assert that

```
k.user_ID != _alice && k.app_ID != _foo_app_ID &&
TokenRecordsOnIdP[k.token].user_ID != _alice
```

where TokenRecordsOnIdP represents IdP's database storing the records of access tokens.[4]

Authorization violation. To detect authorization violations, we add ASSERT statements inside each AddKnowledge_*Type* function. For example, the assertion in function AddKnowledge_Code is:

```
ASSERT(!(c.user_ID == _alice && c.app_ID == _foo_app_ID))
```

This checks that the Code added to the knowledge pool is not associated with Alice on FooApp. Similar assertions are added to the AddKnowledge functions for refresh tokens and session IDs. The app secret is different from the above knowledge types, because it is tied to the app not the user. When *k* is an app secret, we assert that *k* != _foo_app_secret.

Association violation. At the return point of every web API on FooAppS, we need to ensure the correct association of the user ID, the permission (represented by an access token or Code), and the session ID. For example, for Facebook PHP SDK, the assertion is the following. It This ensures that the three session variables of the session identified by cookie.sessionID all involve the same user. Concrete cases are given in Section 5.2.

```
Sessions[cookie.sessionID].user_ID ==
        CodeRecordsOnIdP[ Sessions[
                    cookie.sessionID].code].user_ID
 && Sessions[cookie.sessionID].user_ID ==
        TokenRecordsOnIdP[Sessions[
                    cookie.sessionID].token].user_ID
```

5 Results

We applied our approach to explicate the Facebook PHP SDK, Live Connect SDK and Windows 8 Authentication Broker. The Facebook PHP SDK is the only server-side SDK provided on Facebook's developers' website and is currently among the most widely used authentication/authorization SDKs. Facebook also has SDKs for Android and iOS apps, which have many concepts similar to the PHP SDK, but we have not studied them in detail. The Live Connect SDK is provided by Microsoft for developing metro apps that use Live ID as the identity provider. The Windows 8 Authentication Broker is for metro apps to use an OAuth-based (not only Live ID) identity provider, such as Facebook or Twitter.

5.1 Assumptions Explicated

The models resulting from our study formally capture what we learned about the SDKs and the systems. Our assumptions are specified in two ways: (1) all the ASSUME statements that we added; (2) when we need to assume particular program behaviors, such as a function call must always precede another, we model the behaviors accordingly, and add comment lines to state that the modeled behaviors are assumptions, rather than concrete facts. All the assumptions are added in order to satisfy the assertions that described in Section 4.4. The assertions are fairly uniform — they are all about sensitive data added to the knowledge pool and binding errors in associating sessions, users and permissions.

Verification. After all the assumptions were added, the models were automatically verified by Corral with the bound[5] set to 5, meaning that in the test harness (Figure 7), the counter of the main loop (variable depth) does not exceed 5. Such a depth gives a reasonable confidence that the security properties are achieved by the models and the added assumptions: the properties could only be violated by attacks consisting of six or more steps. Running on a Windows server with two 2.67GHz processors and 32GB RAM, it took 11.0 hours to check the Facebook PHP SDK, 26.3 hours to check Live Connect SDK and 15.1 hours to check the Windows 8 Authentication Broker.

[4] To improve presentation readability, the syntax of the above predicate is slightly changed from the syntax allowed by Boogie; see *https://github.com/sdk-security/* for the exact syntax.

[5] Corral is a fully automatic tool for exploring code paths symbolically. The full automation, however, comes with the limitation that it only performs a bounded search.

Name[a] (SDK)	Assumption[b]	consequence of violation	exploit opportunity	vendor response
A1 (FB)	In FooApp$_C$MakesACall, we assume FooAppC.cookie.sessionID == _aliceSession.	The ASSERT in Table 1 will be false. Mallory's session is associated with Alice's user ID.	When the SDK is used in subdomaining situations, e.g., cloud domains	Counter-measure on service platform
A2 (FB)	For any PHP page, if getUser is called, then getAccessToken must be called subsequently.	Alice's user ID will be associated with Mallory's access token.	When FooAppS contains a PHP page that directly returns the user ID	SDK code fix
A3 (FB)	Before getLogoutUrl returns to client, we assume logoutURL.params.access_token != getApplicationAccessToken().	App access token is added to the knowledge pool (owned by the adversary).	When a PHP page does not have the second code snippet shown in the dev guide	SDK code fix
A4 (LC)	In saveRefreshToken on FooAppS, we assume user_id != refresh_token.user_id.	Alice's refresh token will be associated with Mallory's session on FooAppS.	When the term "*user id on the site*" in the dev guide is interpreted as the user's Live ID	Dev guide revision
A5 (WA)	In callAuthenticateAsyncFromMalApp, we assume (app_id == _MalAppID || user == _Mallory).	Alice's access token or Code for FooApp is obtained by MalAppC.	When a client allows automatic login or one-click login	See Section 5.2.3
A6 (FB)	We assume FooAppC always logs in as Alice, i.e., the first argument of dialog_oauth is "_Alice".	Alice's session will be associated with Mallory's user ID and access token.	When request forgery protection for app logon is missing or ineffective	Notifying developers

Table 2. Critical assumptions uncovered in our study.

[a] FB stands for Facebook PHP SDK, LC for Live Connect and WA for Windows Auth Broker

[b] Boogie syntax does not allow the dot operator to refer to a child element. For simplicity of presentation, we use it in this column.

The verification being bounded is a limitation of the models built for Corral, so we subsequently re-implemented all three models in Boogie language [9]. Verification of Boogie models is not automatic. It requires human effort to specify preconditions and postconditions for procedures, as well as loop invariants (i.e., the invariant clauses). The Boogie verifier checks that (1) every precondition is satisfied by the caller; (2) if all preconditions of the procedure are satisfied, then all the postconditions will be satisfied when the procedure returns; (3) every loop invariant holds initially, and if it holds before an iteration then it will still hold after the iteration. By induction, the verified properties hold for an infinite number of iterations. Rewriting the three models in Boogie took 14 person-days of effort, including a significant portion on specifying appropriate loop invariants. The Boogie modeling did not find any serious case missed in the Corral modeling, but provides a higher level of confidence.

Examining the assumptions in the real world. We manually examined each assumption added to assess whether it could be violated in realistic exploits. This effort requires thinking about how apps may be deployed and executed in real-world situations. Table 2 summarizes the assumptions uncovered by our study that appear to be most critical. These assumptions can be violated in the real world, and the violations result in security compromises. Based on our experience in communicating with SDK providers, finding realistic violating conditions is a crucial step to convincing them to treat the cases with high priority. This step requires extensive knowledge about systems, and does not appear to be easily automated. We describe these assumptions in more detail in Section 5.2. Table 3 lists some assumptions uncovered that, if violated, would also lead to security compromises. But, unlike the assumptions in Table 2, we have not found compelling realistic exploits that violate these assumptions. A few additional assumptions, listed in Appendix B, are needed to complete the verification. They correspond to some simplifications we made to the models. It is unclear if their violations lead to security compromises, but we make it explicit that we have not considered the situations violating these assumptions.

5.2 Confirmed Exploitable Assumptions

This subsection explains each of the critical assumptions in Table 2. These results show concretely how the SDK's security assurance depends on actual system behaviors and app implementations, illustrating the importance of explicating the underlying assumptions upon which secure use of the SDK relies.

5.2.1 Facebook SDK

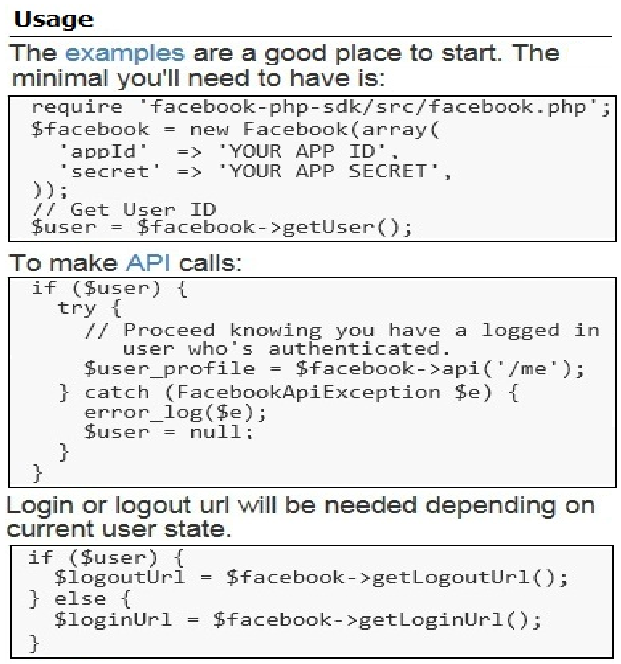
Assumptions A1, A2, A3, and A6 concern the Facebook PHP SDK.

name	assumption	consequence of violation	proposed fix
B1 (FB)	Result of getAccessToken returned to client is not equal to getApplicationAccessToken()	App access token is added to the knowledge pool.	Develop checker to examine the traffic from FooApp$_S$
B2 (FB)	In dialog_oauth, we assume FooApp.site_domain != Mallory_domain	Alice's access token or Code for FooApp is obtained by Mallory.	Develop checker to examine if the "Site Domain" app setting is properly set
B3 (FB)	Before FooApp$_C$ sends a (non-NULL) request, we assume request.signed_request.userId == _Alice	Alice's session will be associated with Mallory's user ID and access token.	Enhance dev guide to require a runtime check on FooApp$_C$
B4 (LC)	In HandleTokenResponse, we assume auth_token.app_ID == _foo_app_ID	Alice's authentication token for MalApp will be used by Mallory to log into FooApp$_S$ as Alice	Develop checker to examine if the signature in the auth_token is verified.
B5 (LC)	In constructRPCookiefromMallory, we assume (RP_Cookie.access_token.user_ID == RP_Cookie.authentication_token.user_ID)	Alice's ID associate with Mallory's access token, or vice versa	Enhance dev guide to require a runtime check on FooApp$_S$

Table 3. Assumptions uncovered that would lead to security vulnerabilities if violated but no realistic exploits known.

Assumption A1. This assumption states that the cookie associated with Alice's client must match Alice's session ID. Figure 8 is a screenshot of the usage instructions given in the *readme* file in the Facebook PHP SDK [17]. It seems straightforward to understand: the first code snippet calls getUser to get the logged-in user's ID (it returns null if the user is not logged in). The second snippet demonstrates how to make an API call, such as me. The third snippet toggles between login and logout, so that a logged-in user will get a *logoutURL* and a logged-out user will get a *loginURL* in the response.

The SDK's implementation for the getUser method is very simple. It calls the getUserFromAvailableData


Usage

The examples are a good place to start. The minimal you'll need to have is:

```
require 'facebook-php-sdk/src/facebook.php';
$facebook = new Facebook(array(
  'appId'  => 'YOUR APP ID',
  'secret' => 'YOUR APP SECRET',
));
// Get User ID
$user = $facebook->getUser();
```

To make API calls:

```
if ($user) {
  try {
    // Proceed knowing you have a logged in
       user who's authenticated.
    $user_profile = $facebook->api('/me');
  } catch (FacebookApiException $e) {
    error_log($e);
    $user = null;
  }
}
```

Login or logout url will be needed depending on current user state.

```
if ($user) {
  $logoutUrl = $facebook->getLogoutUrl();
} else {
  $loginUrl = $facebook->getLoginUrl();
}
```

Figure 8. Facebook PHP SDK usage instructions. (Screenshot from *https://github.com/facebook/facebook-php-sdk/blob/master/readme.md*)

function shown in Table 1. There are two statements (italicized in Table 1) calling setPersistantData, which is to set a PHP session variable denoted as _SESSION['user_id']. Setting _SESSION['user_id'] is a binding operation because it associates the user's identity with the session, which may affect the predicate that we define against association violations — specifically, if Alice's user ID is assigned to the _SESSION['user_id'] of Mallory's session, it would allow Mallory to act on FooApp$_S$ as Alice. Because the session ID is a cookie in the HTTP request, the assertion must depend on how a client runtime handles cookies.

Violating the assumption using subdomaining. Normally, because of the same-origin-policy of the client, cookies attached to one domain are not attached to another. However, the policy becomes interesting when we consider a cloud-hosting scenario. In fact, Facebook's developer portal makes it very easy to deploy the application server on Heroku, a cloud platform-as-a-service. Each service app runs in a subdomain under *herokuapp.com* (e.g., FooApp$_S$'s subdomain runs as *foo.herokuapp.com*). Of course, Mallory can similarly run a service as *mallory.herokuapp.com*.

The standard cookie policy for subdomains allows code on *mallory.herokuapp.com* to set a cookie for the parent domain *herokuapp.com*. When the client makes a request to *foo.herokuapp.com*, the cookie will also be attached to the request. Therefore, if Alice's client visits the site *mallory.herokuapp.com*, Mallory will be able to make the client's cookie hold Mallory's session ID. Thus, FooApp$_S$ binds Alice's user ID to Mallory's session.

In response to our report, Facebook developed a countermeasure, which has been applied on the Heroku platform. It generates a new session ID (unknown to Mallory) when a client is authenticated. Facebook offered us a bounty three times the normal Bug Bounty amount for reporting this issue, as well as the same award each for Assumptions A2 and A3 discussed next.[6]

Assumption A2. This assumption is a case in which Corral actually discovered a valid path for violating an assertion completely unexpected to us. The path indicated that if a PHP page on FooApp$_S$ only calls getUser (e.g., only has the first code snippet from Figure 8), Mallory is able to bind her user ID to Alice's session. The consequence is especially damaging if the session's access token is still Alice's. Corral precisely suggested the possibility (see Table 1): if there is a *signed_request* containing Mallory's user ID, then the first setPersistentData call will be made, followed by a return. The method sets _SESSION['user_id'] to Mallory's ID without calling getAccessToken, which would otherwise keep the access token consistent with the user ID. Therefore, the association between the user ID and the access token is incorrect. The session will operate as Mallory's account using Alice's access token. After investigating our report about this, Facebook decided to add checking code before processing the signed request to the SDK to avoid the need for this assumption.

Assumption A3. This assumption requires that any PHP page that includes the third snippet in Figure 8 must also include the second snippet. In the example code in the figure, it is not obvious why the second snippet is required before the third snippet. However, when we modeled getAccessToken, as shown in Table 4, we realized that in Facebook's authentication mechanism there are two subcategories of access token: *user access token*, which is basically what people usually refer to as "access token", and *application access token*, which is described in Facebook's dev guide [18]. The application access token is provided to a web service for a number of special purposes, such as "publishing instances of 'secure Open Graph actions'". In fact, the app secret can be derived solely from the application access token, so it is a serious authorization violation if Mallory or MalApp$_C$ can obtain it.

Method getLogoutUrl in snippet 3 constructs a URL to send back to the client. The URL contains the result of getAccessToken. To obtain the application access token, Mallory only needs to send a request that hits a failure condition of getUserAccessToken, which prevents $this->accessToken from being overwritten in the bold line in Table 4. We confirmed that this can be done by using an invalid Code in the request.

[6] We donated all three bounties to charities. The donations were one-to-one matched by Facebook.

```
public function getAccessToken() {
   …
   $this->accessToken= $this->getApplicationAccessToken();
   $user_access_token = $this->getUserAccessToken();
    if ($user_access_token) {
      $this->accessToken=$user_access_token;
   }
   return $this->accessToken;
}
```

Table 4. SDK source code of getAccessToken

Interestingly, getAccessToken is also called by getUser in snippet 1 in Figure 8. If a PHP page includes snippet 2, the access token will be used to call a REST API. When it is an application access token, the API will raise an exception, which foils the exploit. That is why snippet 2 is required before snippet 3.

In response to our report on this issue, Facebook modified the SDK so that getLogoutUrl now calls getUserAccessToken instead of getAccessToken, thus avoiding the need for developers to satisfy this assumption.

Assumption A6. This assumption requires that the user on FooApp$_C$ should not be Mallory. Otherwise, Mallory would be able to associate its access token and user id with Alice's session. In Section 6.2, we show that many apps (14 out of 21 tested) indeed violate this assumption. Moreover, this association violation can be particularly damaging when the service app has its own credential system, and supports linking a Facebook ID to Alice's password-protected account. Once the linking can be done in the session, Mallory will be able to sign into Alice's account using Mallory's Facebook ID. We confirmed that among the 14 service apps which violate the assumption, 6 of them support linking, and thus allow Mallory to login as Alice. We reported this issue to Facebook, who undertook the effort of notifying app and website developers.

5.2.2 Live Connect

Assumption A4 concerns how the Live Connect SDK handles "single sign-on for apps and websites" [27]. The sample */LiveSDK/Samples/PHP/OAuthSample* [28] demonstrates how to implement a PHP service app that allows single sign-on. This sample code is essentially the dev guide given as a program skeleton, with comment blocks for app developers to implement. The core of the problem lies in the following function, whose implementation is empty except for a comment:

```
function saveRefreshToken($refreshToken) {
   // save the refresh token associated with the
   //     user id on the site.
}
```

This is precisely what we call a *binding operation*. The refresh token is the input parameter, but it is not clear where the user id comes from. Within the scope of this function, the only place to obtain a user ID is from a cookie called AUTHCOOKIE, which contains the user's Live ID. However, the SDK's logic is not sufficient to ensure that Alice's refresh token is associated with her user ID. Appendix C provides technical details.

We built a proof-of-concept exploit to send to Microsoft. The Live ID team responded that our attack is valid, but it "does not reflect the scenarios we are targeting". The target scenario is a website which has its own credential system, such as a university website, so "the user id on the site" means, for example, the student ID. We replied to the team that an unclear context like this was exactly what we believe needs to be uncovered and at least documented clearly (indeed, explicating such assumptions is one of our main goals). In this case, the context was almost completely hidden: the *OAuthSample* sample is the only sample provided in */LiveSDK/Samples/PHP/*, so it is expected to target more generic scenarios. This is why if saveRefreshToken targets a specific scenario, the context must be made explicit. The team replied us that they would "*add more comments to that code to make the sample code clear on this*." Recently we found that the comment has been revised to "*save the refresh token and associate it with the user identified by your site credential system.*" This change was also made in the ASP.NET version of the sample code.

5.2.3 Windows Authentication Broker

Assumption A5 concerns the Windows 8 Web Authentication Broker, used by Windows 8 apps with OAuth-based identity providers. For concreteness of presentation, we assume the Facebook Identity Provider. In the Auth Broker, the only function for authentication is authenticateAsync. Figure 9 illustrates the data passing through this function when the app requests an access token. The key observation is that the client does not conform to the same-origin policy, because the 302 response is in the context of *https://facebook.com*, while on Windows 8, an app runs in its own domain, *ms-appx://packageID*. Without the same-origin-policy, we were unable to see why Alice's access token for FooApp is guaranteed to be passed to FooApp$_C$, not MalApp$_C$. To test this, we implemented a proof-of-concept MalApp$_C$. It indeed got the access token, which allowed it to do everything FooApp$_C$ can do.

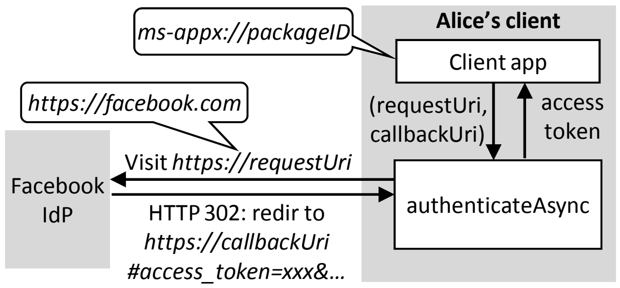


Figure 9. Data flow through authenticateAsync.

We reported this finding to Microsoft and Facebook, and learned their differing perspectives about the responsibility and severity of this issue. Microsoft considered it "*a shortcoming of the OAuth protocol and not specific to our implementation*." Facebook pointed out that when authenticateAsync is called, an embedded browser window (usually called a WebView) is always prompted for Facebook password. This lowered the severity of the attack. We consider this a shaky security basis: if authenticateAsync someday allows a user to login automatically or with one click without using a password, the basis will become invalid.

We investigated how SDKs on other platforms handle the data passing, and found a similar issue with the Facebook SDK for Android. However, on Android, there is a mechanism to skip the password prompt to get the access token automatically. In response to our report, Facebook is developing a fix for its Android SDK.

6 Automated Testing

One additional value of explicating the SDKs is that it may be possible to provide tools that test apps for violations of critical assumptions. Such tests may not be able to guarantee the app always upholds the assumption, but rather focus on testing apps for common vulnerability patterns identified as a result of the explicating process. We developed a prototype to show the feasibility of building such a tester.

6.1 Design

Figure 10 shows our testing framework. For each vulnerability pattern to test, the test case defines the actions of the tester app, the proxy, and a set of server-side tester APIs (e.g., PHP or ASP.NET files). The tester app behaves as MalApp$_C$. The proxy does the necessary traffic manipulations for requests and responses. It also behaves as the unconstrained machine Mallory. Tester APIs implement specific checks for session states, especially for the associations we focus on.

We implemented test cases checking for violations of four assumptions: the vulnerability described in Section 2 (about using an access token for authentication), and vulnerabilities corresponding to the violations of assumptions A1 (concerning the session ID across sub-domains), A6 (about Mallory's user ID associated with Alice's session) and A4 (about binding the user ID with refresh token). Only the test for A4 requires a tester API on the app server.

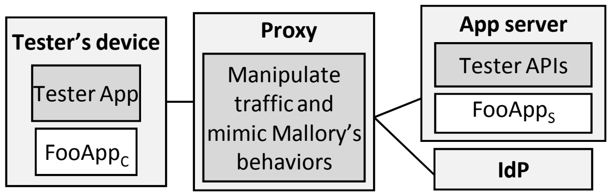


Figure 10. Testing Framework.
(Grey boxes constructed for testing.)

In the first test, the tester app performs the IdP's sign-on steps as Alice, requests an access token, then presents the token to the app server to see if the authentication succeeded. In the second test (regarding A1), if the app server's hostname is *foo.a.com*, the proxy creates another hostname *mallory.a.com*. The test follows the steps described in Section 5.2.1. Eventually the proxy checks if the authentication is successful, but the associated session ID is identical to that of Mallory's session on *foo.a.com*. In the third test (for A6), the proxy observes the HTTP request that FooApp_C sends to Facebook. It finds out which type of data is used as the proof for authentication (a.k.a., the *authenticator*), which can be either a Code or signed request. The proxy also tries to find a field named state, which is an argument supported by Facebook to prevent request forgery for login [16]. The proxy then replaces the authenticator and the state field (if it exists) with the ones that Mallory's session owns. After sending the request, the proxy checks whether Mallory can associate her Facebook ID with Alice's session, and reports a violation if it sees a successful server response.

The fourth test (A4) requires the help of a tester API on the server because it tests whether the refresh token is associated with an appropriate user ID. The test uses the proxy to manipulate the AUTHCOOKIE in the request header so that it contains Mallory's authentication token in Alice's request. The proxy then mimics Mallory to call the tester API, which calls readRefreshToken and checks if it returns Alice's refresh token.

6.2 Results

In general, the testing framework is designed for app developers so that they can avert the common pitfalls in their own implementations. Nevertheless, since some of the tests do not need tester APIs on the server, they can be used with access to the apps alone. This opens the possibility of a third party (such as the SDK provider) performing the tests on submitted apps.

We tried using the tests to check Windows 8 and Facebook apps found in the wild. The sets of apps that we tested are named Set 1, Set 2 and Set 3, corresponding to the first three aforementioned tests respectively. The test apps were obtained as objectively as possible.

Test Set	Number of Apps	Vulnerable
1 (Section 2)	27	21 (78%)
2 (assumption A1)	7	6 (86%)
3 (assumption A6)	21	14 (67%)

Table 5. Test Results.

To construct set 1, we queried "Facebook" in the free apps in Windows 8 App Store, which returned about 572 apps. We ranked the apps by user ratings and examined the apps with a rating of 3+ stars. Apps without a backend service were excluded. We then selected apps that authenticate users through identity providers. This left us with a total of 27 apps.

Set 2 was constructed by doing a Google query for "*herokuapp.com login*", which gave us many URLs on *herokuapp.com*. We visited each URL to see if the website ran a PHP server and appeared reasonably functional. This gave us a list of 20 websites. We then examined the traffic of each website to determine if it used the Facebook PHP SDK. Seven of the sites did, and these were used for Set 2.

To construct Set 3, we used the Google search query "*login.php*" and visited the first 40 result pages[7] to examine which URLs correspond to PHP websites that support Facebook sign-on. We found 21 candidate websites that comprise Set 3.

Table 5 shows the number and percentage of apps that matched the vulnerability pattern in each set. The results for Set 1 show that 78% of tested services with Facebook sign-on mechanism indeed use the access token for server-side authentication. The results for Set 2 reinforce the value of our SDK analysis — when we studied the SDK, we only hypothesized the possibility of this vulnerability. The vulnerability we conceived on a hypothetical service app (FooApp_S) accurately reflects the reality of 86% of services tested in Set 2. The results for Set 3 indicate that 67% of the tested apps would allow Mallory's Facebook ID to be associated with Alice's session. This violation is mainly due to missing or insufficient request forgery protections for user login. This association mistake can be particularly dangerous when the service apps support certain functionalities. For example, we found that many service apps have their own credential systems, and allow a user to link her Facebook ID to her password-protected account. After the linking, the user can use a Facebook login to sign into the password-protected account. When assumption A6 is violated, Mallory is able to link her Facebook ID to Alice's account in the

[7] We needed to examine so many result pages because most webpages matched the query "login.php" for reasons not about our intent, e.g., popular pages containing both words "login" and "php" are often considered a match.

session, and thus able to sign into Alice's account. We confirmed that 6 of the service apps could be exploited in this way.

7 Related Work

The idea of formally verifying properties of software systems goes back to Alan Turing [34], although it only recently became possible to automatically verify interesting properties of complex, large scale systems. Our work makes use of considerable advances in model checking that have enabled model checkers to work effectively on models as complex as the ones we use here. Our work is most closely related to other work on inferring and verifying properties of interfaces such as APIs and SDKs, which we review briefly next.

API and SDK misuses. It is no longer a mystery that APIs and SDKs can be misunderstood and the results often include security problems. On various UNIX systems, setuid and other related system calls are non-trivial for programmers to understand. Chen et al. "demystified" (that is, explicated) these functions by comparing them on different UNIX versions and formally modeling these system calls as transitions in finite state automata [11]. Wang et al. showed logic bugs in how websites integrate third-party cashier services and single-sign-on services [35][36]. Many of the bugs found appear to result from website developers' confusions about API usage. Georgiev et al. showed that SSL certificate validations in many non-browser applications are broken, which make the applications vulnerable to network man-in-the-middle attacks [19]. Our work started from a different perspective — our primary goal is not to show that SDKs can be misused, but to argue that these misuses are so reasonable that it is SDK providers' lapse not to explicate the SDKs to make their assumptions clear. We expect that our approach could be adapted to other contexts such as third-party payment and SSL certificate validation.

Interface Verification. Many researchers have considered issues related to verifying interfaces and their use. Spinellis and Louridas [32] propose a static analysis framework for verifying Java API calls. Library developers are required to write imperative checking code for each API to assist the verification process. Henzinger et al. [1][7] propose languages and tools to help model the interfaces and find assumptions that need to be met for two APIs to be *compatible*, i.e., there is no environment for which they reach an error state. JIST [2] uses a similar approach to synthesize interface specifications for Java classes. This line of work is complementary to ours. Our main effort has been to systematically understand systems and construct semantic models. Currently, we manually add assumptions when counterexamples are found in the models. The assumptions could be considered as a type of "interface specifications" of the SDKs. We believe that our semantic models would be even more valuable with tools that can automatically synthesize high-quality assumptions.

Software testing. Static techniques such as the Static Driver Verifier (SDV) for Windows drivers [4] and dynamic analysis such as symbolic execution [3][12] and fuzz testing [13][20] are widely studied in software testing community. To test websites' of single-sign-on authentications, Bai et al. developed AUTHSCAN [5], which is a technology to automatically recover an authentication protocol from concrete website implementations.

OAuth Protocol analyses. Bansal et al. [6] modeled OAuth 2.0 protocol and verified it using ProVerif [8]. They also built a library for future researchers to model web APIs into ProVerif language more easily. Pai et al. [31] used Alloy framework [23] to verify OAuth 2.0 and discovered a previously known vulnerability. Sun et al. discussed a number of website problems affecting OAuth's effectiveness, such as not using HTTPS, having XSS and CSRF bugs [33]. Although the three SDKs we studied are based on OAuth, our work does not focus particularly on the OAuth protocol. The fact that all three studied SDKs are based on OAuth is mainly because of its widespread adoption, but the security issues we found concern the SDKs and services rather than flaws inherent in the OAuth protocol.

8 Final Remarks

Security exploits nearly always stem from attackers finding ways to violate assumptions system implementers relied upon. Such assumptions are often not carefully documented, and often only implicit in the minds of the system designers. Our study of three important authentication and authorization SDKs supports the need for systematically explicating SDKs to uncover these assumptions. We advocate that a systematic explication process should be part of the engineering process for developing SDKs. Although our current process still requires considerable manual effort in understanding and modeling system behaviors, we believe the need for this effort reveals flaws in the current engineering processes: SDK developers, including those building widely-used security-focused SDKs, have not systematically understood or documented the SDKs' behaviors for producing secure applications. In our study, we found assumptions that were critical to secure use of the SDKs, but that were not clearly documented and were subtle enough to be missed by the majority of tested apps.

Acknowledgments
We thank Martín Abadi, Longze Chen, Cormac Herley, Kevin Sullivan, Helen Wang, Yi-Min Wang, and Westley Weimer for valuable comments on this work and an early draft of the paper. Mike Barnett offered great advice on building and checking the semantic models. We also appreciate the technical help from Eric Lawrence and David Ross about recording Live ID traffic on Windows 8. This work was partly funded by grants from the National Science Foundation and Air Force Office of Scientific Research (but does not necessary reflect the views of the US Government). Yuchen Zhou was also supported in part by a Microsoft Research internship.

Availability
The Boogie models of the three studied SDKs are available at *https://github.com/sdk-security/*.

References
[1] Luca de Alfaro and Thomas A. Henzinger. Interface Automata. In *8th European Software Engineering Conference* (held jointly with *9th ACM SIGSOFT International Symposium on Foundations of Software Engineering*), 2001.
[2] Rajeev Alur, Pavol Cerny, P. Madhusudan, and Wonhong Nam. Synthesis of Interface Specifications for Java Classes. In *32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (POPL), 2005.
[3] Thanassis Avgerinos, Sang Kil Cha, Brent Lim Tze Hao, and David Brumley. AEG: Automatic Exploit Generation. In *Network and Distributed System Security Symposium* (NDSS). February 2011.
[4] Tom Ball, Ella Bounimova, Byron Cook, Vladimir Levin, Jakob Lichtenberg, Con McGarvey, Bohus Ondrusek, Sriram K. Rajamani, and Abdullah Ustuner. Thorough Static Analysis of Device Drivers. *EuroSys*. 2006.
[5] Guangdong Bai, Jike Lei, Guozhu Meng, Sai Sathyanarayan Venkatraman, Prateek Saxena, Jun Sun, Yang Liu, and Jin Song Dong. AUTHSCAN: Automatic Extraction of Web Authentication Protocols from Implementations. In *Network and Distributed System Security Symposium* (NDSS). February 2013.
[6] Chetan Bansal, Karthikeyan Bhargavan and Sergio Maffeis. Discovering Concrete Attacks on Website Authorization by Formal Analysis. *IEEE Computer Security Foundations* (CSF). 2012.
[7] Dirk Beyer, Arindam Chakrabarti, and Thomas A. Henzinger. Web Service Interfaces. In *14th International Conference on World Wide Web* (WWW). 2005.
[8] Bruno Blanchet. An Efficient Cryptographic Protocol Verifier Based on Prolog Rules. In *14th IEEE Computer Security Foundations Workshop* (CSFW), 2001.
[9] Boogie: An Intermediate Verification Language. http://research.microsoft.com/en-us/projects/boogie/
[10] John Bradley. *The Problem with OAuth for Authentication*. http://www.thread-safe.com/2012/01/problem-with-oauth-for-authentication.html
[11] Hao Chen, David Wagner and Drew Dean. Setuid Demystified. *USENIX Security Symposium*. 2002
[12] Chia Yuan Cho, Domagoj Babíc, Pongsin Poosankam, Kevin Zhijie Chen, Edward XueJunWu, and Dawn Song. MACE: Model-inference-assisted Concolic Exploration for Protocol and Vulnerability Discovery. In *20th USENIX Security Symposium*. 2011.
[13] Adam Doupé, Ludovico Cavedon, Christopher Kruegel, and Giovanni Vigna. Enemy of the State: a State-aware Black-box Web Vulnerability Scanner. In *21st USENIX Security Symposium*. 2012.
[14] Facebook. “The Facebook bounty program,” http://www.facebook.com/whitehat/bounty/
[15] Facebook. *SDK Reference - Facebook SDK for PHP*. http://developers.facebook.com/docs/reference/php/
[16] Facebook. OAuth Dialog, https://developers.facebook.com/docs/reference/dialogs/oauth/
[17] Facebook. *PHP SDK Usage*. https://github.com/facebook/facebook-php-sdk/blob/master/readme.md
[18] Facebook. *Using App Access Tokens*. http://developers.facebook.com/docs/opengraph/using-app-tokens/
[19] Matin Georgiev, Subodh Iyengar, Suman Jana, Rishita Anubhai, Dan Boneh, Vitaly Shmatikov. The Most Dangerous Code in the World: Validating SSL Certificates in Non-Browser Software. *ACM CCS*. 2012.
[20] Patrice Godefroid, Michael Y. Levin and David Molnar. Automated Whitebox Fuzz Testing. *Network and Distributed System Security Symposium*. 2008
[21] Google. *Using OAuth 2.0 to Access Google APIs*. https://developers.google.com/accounts/docs/OAuth2
[22] Dick Hardt. *The OAuth 2.0 Authorization Framework (RFC6749)*. http://tools.ietf.org/html/rfc6749
[23] Daniel Jackson. *Alloy: A Language and Tool for Relational Models*. http://alloy.mit.edu/alloy/index.html.
[24] Akash Lal, Shaz Qadeer, and Shuvendu Lahiri. Corral: A Solver for Reachability Modulo Theories. *Computer Aided Verification* (CAV). 2012
[25] Microsoft. *Live Connect Developer Center – Metro Style Apps*. http://msdn.microsoft.com/en-us/library/live/hh826551.aspx
[26] Microsoft. *Live SDK developer guide – Signing users in*. http://msdn.microsoft.com/en-us/library/live/hh243641#signin
[27] Microsoft. *Live SDK Developer Guide – Single Sign-on for Apps and Websites*. http://msdn.microsoft.com/en-us/library/live/ hh826544.aspx
[28] Microsoft. "LiveSDK's OAuth Sample Code in PHP," https://github.com/liveservices/LiveSDK/tree/master/Samples/PHP/OauthSample
[29] Microsoft. *Live Connect Developer Center – REST Reference*. http://msdn.microsoft.com/en-us/library/live/hh243648.aspx
[30] Microsoft. *Windows.Security.Authentication.Web namespace*, http://msdn.microsoft.com/library/windows/apps/BR227044
[31] S. Pai, Y. Sharma, S. Kumar, R.M. Pai, and S. Singh. Formal Verification of OAuth 2.0 Using Alloy Framework. In *Communication Systems and Network Technologies* (CSNT). 2011.

[32] Diomidis Spinellis and Panagiotis Louridas. A Framework for the Static Verification of API Calls. *Journal of Systems and Software*. July 2007.
[33] San-Tsai Sun and Konstantin Beznosov. The Devil is in the (Implementation) Details: An Empirical Analysis of OAuth SSO Systems. *ACM CCS*. 2012.
[34] Alan Turing, "Checking a Large Routine", presented at EDSAC Inaugural Conference, 1949. (available from http://www.turingarchive.org/browse.php/B/8)
[35] 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
[36] 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. *IEEE Symposium on Security and Privacy*. 2012
[37] Rui Wang, Yuchen Zhou, Shuo Chen, Shaz Qadeer, David Evans, Yuri Gurevich. Explicating SDKs: Uncovering Assumptions Underlying Secure Authentication and Authorization. *Microsoft Research Technical Report MSR-TR-2013-37*, March 2013

Appendix A. Prevalence of SDKs.

To understand how widely-used different SDKs are we first searched for keyword "Facebook" in the Windows App Store and filtered the results by selecting free and trial apps only, which left us with a total of 572 apps. We then sorted the results by users' rating, after which we went through the top of the list one by one to check if the app has Facebook or Live connect SSO built-in. We also monitored network traffic using Fiddler on those apps that have SSO feature, and this allows us to eliminate the ones that do not run an online service. We excluded non-English apps and also apps that do not work properly. After the selection process we came up with a total of 27 apps as listed below:

App Name	SDK(s)
Soluto	WA(FB)
Givit	Unknown
Fliptoast	WA(FB)
Donelo	Unknown
IM+	WA(FB)/LC
Interference	Live
Norton Satellite	Unknown
Slide Ur buddy	WA(FB)
EuroCup	Unknown
Shufflr	WA(FB)
Social Umami	Unknown
SumAttack	WA(FB)
Guess Who	WA(FB)
Flixpicks	WA(FB)/LC
TwentyOne	Unknown
Apyo	Unknown
Where's my stuff	Unknown
Mahjong 31	Unknown
Tic Challenge	WA(FB)
Color orbs	Unknown
tagmap	WA(FB)
word gap	LC
word town	Unknown
noots	Unknown
RecipeHouse	WA(FB)
Alaska Airlines	Unknown
Captain Dash	LC

WA(FB): Windows Auth Broker using Facebook IdP
LC: Live Connect
Unknown: We could not identify the observed authentication traffic.

Appendix B. Additional Assumptions.

The following assumptions were needed to complete the verification, but not included in Table 2 or Table 3 since they do not appear to have any likely security consequences.

C1: (Live Connect)
There are two sets of Live Connect APIs, one of Microsoft apps and services, such as Skydrive, the other for non-Microsoft apps and services. We assume the two sets of APIs cannot be called together, i.e., any sequence of calling these APIs is confined to only one of the two sets.

C2: (Live Connect, Windows Auth Broker)
We assume no possibility of executing a script provided by Mallory/MalApp$_C$ inside FooApp$_C$. (Actually, we are concerned that DOM methods like `InvokeScript` and `ScriptNotify` may violate this assumption, but have not yet identified a clear security issue.)

C3: (all)
As explained in Section 4.2, we assume that access token, Code, authentication token, app secret, app ID, user ID, session ID and so on are of different types, although in reality they are all strings. We do not allow type mismatches.

Appendix C. Details about assumption A4

The entry function of a PHP page is handlePageRequest, shown below. It allows a client app to present a Code or a refresh token in the HTTP request in order to authenticate into the service app.

```
function handlePageRequest()
{  $Code = $_GET[CODE];
   if (!empty($Code))
   {  $token = requestTokenByCode($Code);
      handleTokenResponse($token);
      return;
   }
   $refreshToken = readRefreshToken();
   if (!empty($refreshToken))
    {  $token = requestTokenByRefreshToken($refreshToken);
      handleTokenResponse($token);
      return;
   }
}

function handleTokenResponse($token)  {
     $authCookie = $_COOKIE[AUTHCOOKIE];
     $cookieValues = parseQueryString($authCookie);
     if (!empty($token))
     {   $cookieValues[ACCESSTOKEN] =
                        $token->{ACCESSTOKEN} ;
        $cookieValues[AUTHENTICATION_TOKEN] =
                        $token->{AUTHENTICATION_TOKEN};
        $cookieValues[SCOPE] = $token->{SCOPE};
        if (!empty($token->{ REFRESHTOKEN })) {
L1:        saveRefreshToken($token->{ REFRESHTOKEN });
        }
      }    ...
L2:  setrawcookie(AUTHCOOKIE,$cookieValues,…);
}
```

HandlePageRequest uses either the Code or the refresh token to request a data structure called "token" from the IdP (i.e., Live ID service). The "token" is a structure that contains an access token field, a refresh token field, a scope field and others. (The term "token" here is confusing since we are discussing so many different types of tokens in the paper, and this "token" is not even one of them, but a structure containing them. We stick to the term in order to explain the source code, but this notion of "token" is particular to this appendix section.)

The token is passed to handleTokenResponse, which updates a cookie AUTHCOOKIE using the token: it constructs an array cookieValues, calls saveRefresh-Token at Line L1, and tries to set the cookie using cookieValues at Line L2.

Functions readRefreshToken and saveRefreshToken are shown verbatim below. They are empty except for the comment lines.

```
function readRefreshToken() {
   // read refresh token of the user identified by the site.
   return null;
}
function saveRefreshToken($refreshToken)  {
    // save the refresh token associated with the user id
       on the site.
}
```

For saveRefreshToken, app developers are instructed to "save the refresh token associated with the user id on the site". This is precisely what we call a binding operation. When we tried to build the semantic model, it was necessary to understand precisely the instruction. "The refresh token" of course refers to the function parameter, but where to get "the user id"? Within the function scope of saveRefreshToken, the only place to obtain a user ID is from the AUTHCOOKIE, because both cookieValues and token are just local variables inside handleTokenResponse. Thus, the only way to implement saveRefreshToken appears to be something like:

```
$authCookie = $_COOKIE[AUTHCOOKIE];
$aCV = parseQueryString($authCookie);
saveToDatabase(refreshToken,
   $aCV[AUTHENTICATION_TOKEN]->{USER_ID});
```

The vulnerability. For a binding operation, we need to examine if there is an association violation. In this case, the refresh token comes from the Code, which is an argument in the HTTP request, and the user ID comes from the cookie. An association violation occurs when we make two calls to handlePageRequest (i.e., two post requests to *callback.php*), one with Mallory's code, the other with Alice's. The details are given below. It results in Mallory getting Alice's permission.

(i) When Alice's client visits *mallory.com*, it will execute a script that posts a request to *foo.com/callback.php?Code=[MalloryValidCode]*, in which MalloryValidCode is the code that Mallory obtained in his own authentication with *foo.com*. This request makes _COOKIE[AUTHCOOKIE] contain the authentication token of Mallory, which contains her user ID.

(ii) The script then starts a normal authentication by posting a request to *login.live.com/oauth20_authorize.srf?client_id=[FooAppID]&redirect_uri=https://foo.com/callback.php&....*

The Live ID server will redirect the client to *foo.com/callback.php?code=[AliceValidCode]*. Note that at Line L1, the cookie is unchanged since step (i). Thus, saveRefreshToken binds Mallory's user ID with the refresh token obtained using AliceValidCode. From this point, when Mallory makes request to *foo.com*, the PHP code will retrieve the refresh token, and thus the session actually possesses Alice's permission.