Vetting Single Sign-On SDK Implementations via Symbolic Reasoning

Ronghai Yang, *The Chinese University of Hong Kong, Sangfor Technologies Inc.;*
Wing Cheong Lau, Jiongyi Chen, and Kehuan Zhang, *The Chinese University of Hong Kong*

https://www.usenix.org/conference/usenixsecurity18/presentation/yang

This paper is included in the Proceedings of the
27th USENIX Security Symposium.

August 15–17, 2018 • Baltimore, MD, USA

ISBN 978-1-931971-46-1

Open access to the Proceedings of the
27th USENIX Security Symposium
is sponsored by USENIX.

Vetting Single Sign-On SDK Implementations via Symbolic Reasoning

Ronghai Yang[1,2], Wing Cheong Lau[1], Jiongyi Chen[1], and Kehuan Zhang[1]

[1]*The Chinese University of Hong Kong*,
[2]*Sangfor Technologies Inc.*

Abstract

Encouraged by the rapid adoption of Single Sign-On (SSO) technology in web services, mainstream identity providers, such as Facebook and Google, have developed Software Development Kits (SDKs) to facilitate the implementation of SSO for 3rd-party application developers. These SDKs have become a critical foundation for web services. Despite its importance, little effort has been devoted to a systematic testing on the implementations of SSO SDKs, especially in the public domain. In this paper, we design and implement S3KVetter (Single-Sign-on SdK Vetter), an automated, efficient testing tool, to check the logical correctness and identify vulnerabilities of SSO SDKs. To demonstrate the efficacy of S3KVetter, we apply it to test ten popular SSO SDKs which enjoy millions of downloads by application developers. Among these carefully engineered SDKs, S3KVetter has surprisingly discovered 7 classes of logic flaws, 4 of which were previously unknown. These vulnerabilities can lead to severe consequences, ranging from the sniffing of user activities to the hijacking of user accounts.

1 Introduction

Single Sign-On (SSO) protocols like OAuth2.0 and OpenID Connect have been widely adopted to simplify user authentication and service authorization for third-party applications. According to a survey conducted by Janrain [29], 75% users choose to use SSO services, instead of traditional passwords, to login applications. As a conservative estimate in [49], 405 out of Top-1000 applications support SSO services, indicating that SSO login has already become a mainstream authentication method and still continues its strong adoption.

Motivated by the prevalence of SSO services, mainstream Identity Providers (IdPs) like Google and Facebook, have provided their Software Development Kits (SDKs) to facilitate the implementation of third party services (e.g. IMBD and Uber), which are referred to as the Relying Parties (RP) under the SSO framework.

To further enhance flexibility, some high-profile open source projects [3, 21] have integrated SSO SDK modules from different IdPs so that an RP application can readily support multiple IdPs at the same time. These SDKs are the core component of SSO services and have enjoyed millions of downloads (see Table 1).

Typically, an SSO SDK provider would release the source code of its SDK and provide documentations, together with simple usage examples. It then leaves the rest to the RP developers. Without fully understanding the SDK internals, most RP developers simply follow the sample codes to invoke the SDK functions. As such, one important question is that: *Is an SSO SDK itself secure?* Note that if the internals of a SDK already contain vulnerabilities, then all RP applications using the vulnerable SDK become susceptible. Given the popularity of these SDKs and the nature of SSO services, any security breach can lead to critical implications. For example, an attacker may be able to log into billions of user accounts [48].

The goal of this work is to systematically test whether an SSO SDK is vulnerable by itself. We will focus on the logic vulnerabilities of a SDK, which allow an attacker to log into RP applications as a victim. To the best of our knowledge, this is the first work to analyze the SSO SDKs. Most existing work on SSO security does not analyze the code of the SSO system, let alone the SDK. More specifically, there are mainly two types of work in the literature. The first type reasons about the *specification* of the standard SSO protocols [23, 39] by different methods including model checking [5,7,15,19], cryptographic proof [11] and manual analyses [34]. The other type aims to discover vulnerabilities of real-world SSO implementations via network traffic analysis [43,44,47,48] and large-scale automated testing [18,33,49,51]. The former does not care about the

SSO implementation, and the latter treats the implementation as a black box. Consequently, both cannot detect logic flaws buried deep in the SSO SDKs.

To this end, this paper introduces S3KVetter, a tool which automatically identifies vulnerabilities in the SSO SDK internals. Our key insight is to leverage dynamic symbolic execution, a widely used technique for program analysis (*e.g.*, [9, 22]), to track feasible execution paths and the associated predicates of the SSO SDK under test. For each path, S3KVetter then utilizes a theorem prover[1] to check whether the predicates violate SSO security properties. Although these techniques have been heavily studied, they cannot be directly applied to SSO-like applications due to the multi-party nature and multiple-lock-step operations of SSO services. We have thus developed new techniques including request order scheduling and multi-party coordination for this kind of multi-party applications.

We have implemented a full-featured prototype of S3KVetter and applied it to check 10 popular SSO SDKs. These SDKs are all carefully engineered and enjoy a large number of downloads (see Table 1). They support different SSO protocols (OAuth2.0 or OpenID Connect) and various grant flows (authorization code flow and implicit flow). To our surprise, S3KVetter has discovered, among these security-focused SDKs, 7 classes of serious logic vulnerabilities and 4 of them are previously unknown. The security impact can range from sniffing user activities at the RP, to the total hijacking of the victim's RP account. In summary, we have made the following contributions:

- *Measurement study and new findings.* We have systematically conducted an in-depth security analysis on 10 commercially deployed SSO SDKs, the first of this kind. We discover 7 types of serious logic vulnerabilities, 4 of which are previously unknown. We demonstrate these vulnerabilities can lead to critical security implications. Our findings show that the overall security quality of SSO SDKs (and thus their deployment) is worrisome.
- *Effective vulnerability detection for distributed systems via symbolic reasoning.* We have designed and implemented S3KVetter to perform security analysis of SDK internals based on dynamic symbolic execution and a theorem prover. In particular, we develop a set of new techniques, including symbolizing request orders and multi-party coordination, to improve symbolic execution for multi-party distributed systems with multiple-lock-step interactions.

The remainder of this paper is organized as follows: Section 2 introduces the background. Section 3 presents the overview of S3KVetter. Section 4 discusses its detailed design. Additional implementation considerations are given in Section 5. We evaluate the performance of S3KVetter in Section 6 and detail the discovered vulnerabilities in Section 7. We discuss the lessons learned in Section 8 and summarize related works in Section 9. We conclude the paper in Section 10.

[1]We will use the terms theorem prover, constraint solver and Satisfiability Module Theories (SMT) solver interchangeably.

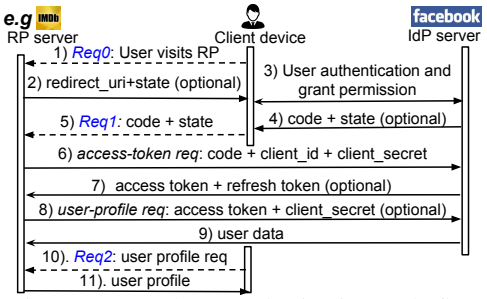

Figure 1: OAuth 2.0 authorization code flow

- Dash lines represent symbolic links that can be controlled by an attacker.

2 Background

OAuth2.0 [23] and OpenID Connect [39] (OIDC) have become the *de facto* SSO standard protocols. Therefore, in this paper, we only focus on these two protocols[2]. In an SSO ecosystem, there are three parties: a User, a Relying Party server (RP server) and an Identity Provider server (IdP server)[3]. The goal of SSO services is to allow the user to log into the RP via the IdP. To achieve this goal, the IdP issues an access token (as in the case of OAuth2.0), and sometimes together with an id_token (as in the case of OIDC), to the RP so that the latter can retrieve the user identity information hosted by the IdP. To complete the process, both SSO protocols have developed multiple authorization grant flows, but only two of them, namely, the authorization code flow and the implicit flow, are commonly deployed in practice. While S3KVetter supports both protocols and both authorization flow types for the web and mobile platforms, we use the authorization code flow of OAuth2.0 under the web platform as the running example throughout this paper.

2.1 Authorization Code Flow of OAuth2.0

Fig. 1 presents the authorization code flow of OAuth2.0. At a high level, the call flow consists of the following five phases:

I. (Step 1-3) The user initiates the Single-Sign-On process with the RP and gives the IdP his approval regarding the permissions requested by the RP;

[2]We use SSO to represent these two protocols, if not specified otherwise.

[3]For the ease of presentation, we use the terms IdP server and IdP, as well as, RP server and RP interchangeably.

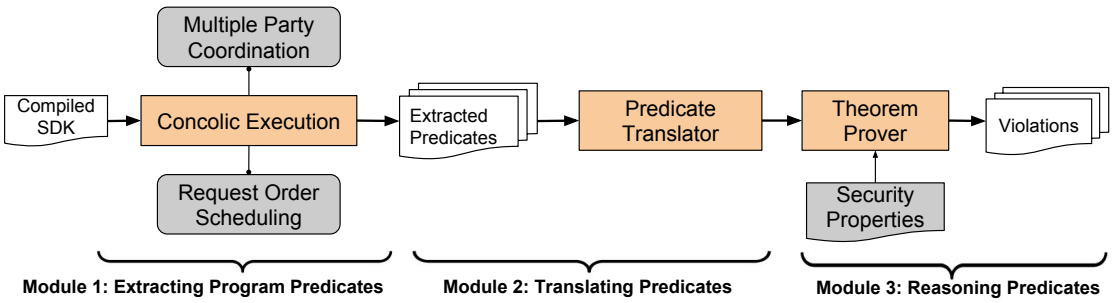

Figure 2: S3KVetter architecture

II. (Step 4-5) The IdP returns an intermediate proof (*code*) to the RP via the user;

III. (Step 6-7) The RP approaches the IdP with this proof and its own credentials to exchange for an access token ;

IV. (Step 8-9) The RP can then use this token to access the information of the user hosted by the IdP ;

V. (Step 10-11) The user can then access his information hosted by the RP.

Refer to Appendix A for detailed descriptions of the individual steps in Fig. 1. Notice that, from the perspective of the RP, the messages exchanged in Fig. 1 are typically handled by the SSO SDK. While we will use Fig. 1 as an illustrative example throughout this paper, our work actually goes beyond Fig. 1. For example, we will discuss the vulnerability associated with MAC_key (Section 7.4) that is not presented in Fig. 1.

3 Overview

In this paper, we focus on analyzing the authentication issues of an SSO SDK. In particular, we use S3KVetter to analyze whether the implementation of a target SDK contains errors that would allow an attacker to login as victims. It is worth to note that S3KVetter can also be extended to study the security of other multi-party applications like payment services as discussed in Section 6.5.

Threat Model

We assume the attacker has the following capabilities: (1) The attacker can lure the victim to visit a malicious RP (mRP)[4]. (2) The attacker can setup an external machine and use his/her own account to freely communicate with the client, IdP and RP server. (3) If the victim does not use HTTPS, the attacker can eavesdrop the communication of the victim's client device. Besides that, the attacker does not have any other advantages (*e.g.*, he/ she does not have the source code or binary executable of the remote IdP server).

[4]For the web platform, mRP is a malicious web page. For mobile platforms, mRP can be an APK file installed on the victim's mobile device. Regardless, mRP does not require any privileged permissions.

System Architecture

Fig. 2 presents the high-level system architecture of S3KVetter, which contains three components: an extended concolic (dynamic symbolic) execution engine, a predicate translator and a theorem prover. The concolic execution engine aims to explore the target SSO SDK exhaustively and output all the feasible program paths in the form of a predicate tree. To support formal reasoning, the predicate translator then expresses this predicate tree using a precise syntax that lends itself to precise semantics. Finally, taking the translated predicate tree and our manually developed list of security properties as inputs, the theorem prover reasons about each program path for security property violation. If there is no satisfiable solution, then the SDK is considered to be secure. Otherwise, the theorem prover outputs the concrete inputs (in the form of SSO handshake messages and parameters) that can trigger the violation.

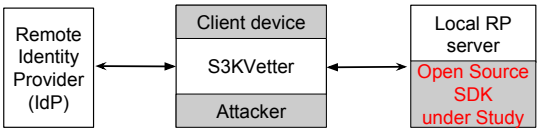

Figure 3: The Role of S3KVetter

Fig. 3 shows the setup of the overall system in which S3KVetter simulates the client device to communicate with the RP server (*i.e.*, SDK) and IdP server. S3KVetter also acts as the attacker to intercept and manipulate the victim's messages (*e.g.*, via malicious RP or eavesdropping). These messages are then fed to the SDK for symbolic exploration. Since the open-source SDK is freely available online, the analyst can build a local RP server to symbolically explore the SDK.

4 Design of S3KVetter

In this section, we present the innovations introduced by S3KVetter to tackle the special technical challenges of testing multi-party systems with multiple-lock-step operations. We will also illustrate how conventional dynamic symbolic execution schemes, without our extensions, can incur false positives, miss bugs, or get stuck at shallow,

non-core error-processing paths, when analyzing multi-party protocols/ systems.

4.1 Symbolic Exploration of SDKs

Based on dynamic symbolic execution, S3KVetter can track how the operations on specific symbolic fields/ variables affect the final computation result. We leverage these messages to build a so-called symbolic predicate tree. One example is presented in Fig. 4, which represents the conditional-checkings of the RequestOAuthLib SDK [3], a popular SSO SDK. Here, the non-leaf nodes in the tree represent symbolic constraints enforced by the corresponding path, and the leaf nodes represent the final computation results (*e.g.*, an access token or the identity of a logged-in user in the context of SSO). For the ease of presentation, we have simplified the tree by omitting numerous branches, nodes and removing multiple constraints (shown as dashed lines in the figure). This SDK involves 649 different execution paths[5], which would require laborious manual effort by testers/ developers to generate. By contrast, S3KVetter, leveraging high-coverage symbolic execution, automatically explores different corner-case situations.

Intuitively, the symbolic predicate tree has captured rich semantic information: The leftmost path in Fig. 4 corresponds to the case where the user skips `Req0` (*i.e.*, Step 1 in Fig. 1) and directly sends `Req1` (Step 5) to the SDK. Upon receiving `Req1`, the SSO SDK under test first checks whether the communication uses HTTPS, followed by verifying the existence of a `code` parameter in the URI. If these conditions are satisfied, the SDK will send an access-token request (Step 7) to the IdP server. Such semantic information is essential and effective for vulnerability detection. For example, this leftmost path does not check the `state` variable but still allows a user to login successfully. This corresponds to the vulnerability of use-before-assignment of the `state` variable, as to be detailed in Section 7.3.

4.1.1 Symbolizing Request Orders

An SSO system requires multiple interactions with the user to complete a task (e.g., authentication and authorization). To be realistic, S3KVetter should allow attackers to randomly and symbolically select execution orders such as making out-of-order requests, skipping/ replaying requests. Although existing symbolic execution studies [10, 31, 40] have proposed different techniques to support asynchronous event/ request orders, they require expert-level domain knowledge of the application under test to provide all the possible external

[5] We only consider OAuth-related paths without counting those non-core paths, *e.g.*, those related to encoding.

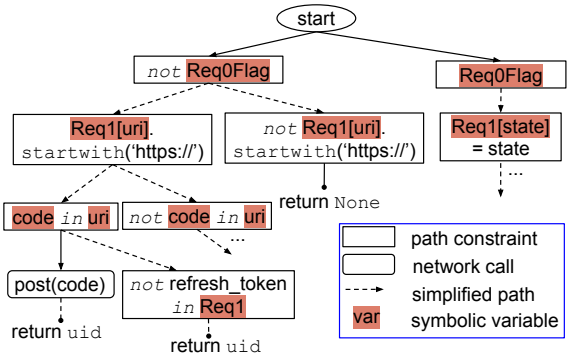

Figure 4: One example of symbolic predicate tree

events (*e.g.*, atomic rule updates and flow independence reduction for OpenFlow application [10]). In short, their approaches cannot be readily generalized for other applications. more thoroughly, S3KVetter should allow attackers to randomly, symbolically select execution orders such as making out-of-order requests, skipping/ replaying requests.

We develop a general and simple scheduling algorithm, which does not require any application-specific heuristic from the analyst, to systematically explore execution paths by generating inputs and *schedules* (*i.e.*, request orders) one by one. The algorithm first guides S3KVetter to run the SDK under test with the sample input and the normal schedule. Then the algorithm does the following loop to sweep possible schedules and feasible program paths: (1) it tries to explore all the feasible program paths of the SDK under the selected schedule; (2) it then generates a new schedule with the goal to explore different program paths.

The remaining issue is to generate a new schedule based on the normal one. Recall that we are interested in the authentication property only, which is typically completed by the last request in the call-flow. Therefore, all of our generated schedules end with the last request. We use Fig. 5, which contains three requests `Req0`, `Req1` and `Req2`, to illustrate how to generate a new schedule as follows:

1. Develop the power set of the normal execution order and exclude the empty set or those subsets which do not contain the last request. The resultant schedule includes: {`Req2`}, {`Req1`,`Req2`}, {`Req0`, `Req2`}, {`Req0`, `Req1`, `Req2`}
2. Consider the ordering in the remaining subsets. For example, a subset {`Req0`, `Req1`, `Req2`} can mean two possible execution orders: {`Req0`, `Req1`, `Req2`} and {`Req1`, `Req0`, `Req2`}. Note that we keep the order of the last request (*i.e.*, `Req2`).
3. Put all the well-ordered subsets into a scheduling queue. For Fig 5, we have 5 schedules in total.

The intuition behind this scheme is that S3KVetter

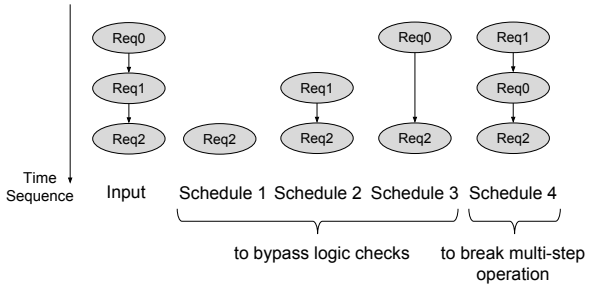

Figure 5: Scheduling for out-of-order requests

attempts to skip any important logic check, break the multi-step operations or replay requests (as can be seen in Figure 5). For example, the schedule of {`Req1`, `Req2`} guides S3KVetter to skip the first request, a key milestone of the SSO business process, which leads to the discovery of the vulnerability of use-before-assignment of the `state` variable (Section 7.3). Another important feature is to break/ subvert the order of requests (*e.g.*, {`Req1`, `Req0`, `Req2`}), which can lead to the so-called "failure to revoke authorization" problem [49]. Finally, the replay function is achieved since every schedule (*e.g.*, {`Req2`}, {`Req1`,`Req2`}) will start to explore the SDK with the same requests (where `Req2` is replayed).

Note that S3KVetter will not generate a complete set of request orderings since an attacker, in theory, can generate infinite number of request orderings, *e.g.*, by repeating each request arbitrary number of times. However, according to our experience, the scheduler we incorporated into S3KVetter can generate a rich set of promising patterns/ request orderings. Nonetheless, with the framework of S3KVetter, it is relatively straightforward to incorporate additional patterns, if any, developed in the future.

4.1.2 Coordinating among Multiple Parties Silently

SSO applications need to communicate among multiple parties. Unfortunately, existing symbolic execution frameworks are *not* designed for distributed multi-party systems. To fill this gap, researchers actually have developed different approaches, but none of them work perfectly for SSO-like applications. The key problem of existing solutions is that different parties have *different views of the entire system status* if we break the request orders. The case becomes worse in the existence of one-time-use parameters (*e.g.*, `code`, `state`, *etc.*). Below we illustrate the limitations of existing approaches.

The first approach is to concretely run the external functions. However, since the IdP server typically imposes limit on API access rate, a large number of invocations of the external functions can easily hit the control threshold and lead to unexpected responses. Worse still, the widely used one-time-use parameters cannot be correctly generated/ processed in the case of symbolizing request orders. We take the `code` variable as the example to illustrate the problem. With `Req0` (*i.e.*, Step 1 of Fig. 1), S3KVetter can get a `code` from the IdP in Step 4 (note that S3KVetter simulates the client device). If S3KVetter skips this request and directly sends `Req1`, to exchange for an access token in Step 6, S3KVetter has no choice but to either use an old value or locally generate a seemingly legitimate `code`. For both cases, the IdP returns `error` since the `code` should be generated by the IdP server and can only be used for once. As such, the first approach will get stuck in non-core error-processing paths.

The second solution is to check the return type of the external function and then returns a random value of this type without executing the external functions (e.g., DART [22]). However, this solution can lead to false positives. Consider the example above, even when a `code` is already used, DART may still return an access token string (instead of an error message) to the SDK. In this case, the testing tool may report a false positive: An attacker can use an old `code` to login. The third approach (*e.g.*, KLEENet [40]) is to symbolically explore the external functions as well. However, this is not a viable approach for our case as we do not have the source code or binary of the remote IdP server to support symbolic exploration.

Solution. Due to the different views perceived by different parties, some requests with nonce parameters, which are considered to be legitimate by the RP, may be rejected by the IdP. To tackle such inconsistency, S3KVetter concretely simulates, and more importantly, modifies, the entire external world for the SDK under test. Specifically, S3KVetter analyzes the IdP behaviors and directly responds to the RP SDK as if it is the IdP. Instead of strictly following the IdP's behaviors, S3KVetter modifies the response so that every party has the same synchronized view on the global system state. To be more specific, S3KVetter simulates a slightly different IdP as follows:

1. Once a nonce parameter is consumed, S3KVetter, unlike the real-world IdP server, will first generate a new nonce value internally.
2. When S3KVetter starts to explore another path, it will first check whether the previously generated nonce value satisfies the constraints of the path to be explored or not. If so, directly use this new value.
3. Otherwise, S3KVetter checks the local SDK conditions related to this nonce. Therefore, it uses the value solved by the constraint solver and stores the previously generated value for later use.

Since S3KVetter drives the SDK execution, the status of the SDK is closely tracked by S3KVetter. Therefore, S3KVetter can internally force its simulated remote IdP

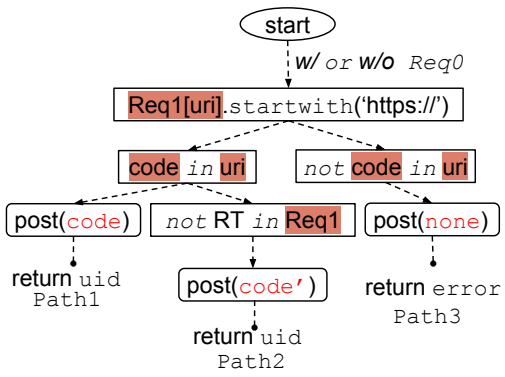

Figure 6: Illustration of multiparty coordination

server to synchronize its own state with the SDK. As such, both parties can automatically share the same view *without any code changes by the SDK*. Below, we illustrate this idea with the `code` example.

Code Example: Fig. 6 illustrates how S3KVetter can coordinate multiple parties with the `code` example. Through Path1, the RP can obtain the user information with a fresh `code`. Upon completion of Path1, the used `code` is invalidated. But at the same time, S3KVetter dynamically generates a new random `code'`. When exploring Path2 (where we skip `Req0`), S3KVetter finds that `code'` satisfies the path constraint ($\texttt{code}' \neq \texttt{None}$) and therefore provides the `code'` for the SDK. Now this `code'` is pre-generated and becomes valid. For Path3, S3KVetter finds that this path requires $\texttt{len(code)} = 0$. As such, S3KVetter provides an empty value solved by the constraint solver for the SDK (and puts another on-the-fly generated `code''` aside).

Implementations: The implementation requires to model the IdP server so that S3KVetter, in most cases, can rely on the SDK as the real IdP. One key observation is that IdPs typically follow the specification and provide similar functions. Therefore, we just need to model one IdP server, and the resultant model can work for multiple SDKs. The implementation involves two major steps. The first step is to infer and model the real-world IdP behaviors, which turns out to be not that challenging. On one hand, we follow existing work [5, 46] to perform blackbox differential fuzzing analysis (*i.e.*, under different input arguments and app settings) for a better understanding of the conditional checking enforced by real IdPs. On the other hand, we also refer to the prototype IdP implementations provided by some open source projects [17]. Second, we implement stub methods for all the common network API methods of Python (*e.g.*, requests, urllib, *etc.*). Upon any network requests, our instrumented functions are invoked instead and reply the SDK on behalf of the IdP server.

4.2 Translating the Predicate Tree

To support formal reasoning, we should translate the extracted tree (*e.g.*, Fig. 4) to a set of Boolean logic formulae. Given the simple syntax of logic languages (*e.g.*, SMT-Lib v2.0), the translation is relatively straightforward. We also observe that every node in the predicate tree can be readily represented as a logic formula. Observe from Fig. 4 that the node which checks whether `uri` contains a `code` parameter can be represented as (`str.contains uri code`) in the language of SMT-Lib. To get the final computation result (*i.e.*, reach the leaf node), all the node logic formulae from the root to the target leaf node should be satisfied. Therefore, a program path can be represented as the conjunction of all the node logic formulae along this path. Similarly, we can use the disjunction of all the path logic formulae to represent the entire predicate tree.

4.3 Reasoning Predicates

The goal of S3KVetter is to detect flawed SDK implementations by checking the logic in the SDK internals. To achieve this goal, we may proceed in two ways. The first is to model all the incorrect logic patterns. However, it is difficult to generate such an exhaustive list. Therefore, we take an alternative approach by modeling the correct logic that should be enforced by the SDK. Then we can check whether the SDK under test follows these logical conditions or not.

4.3.1 Defining Security Property

An SSO system involves interactions among the user, the RP server and the IdP server, where any weak communication links (*i.e.*, 11 steps in Fig. 1) can lead to logic flaws. It is difficult to develop the security requirements for each link since neither protocol specification nor developer documentation explicitly defines the security goal for each method/ API call. Typically, the developer guidelines instruct a party to complete a set of operations and hope that the final security guarantee can be automatically reached by these operations. It is therefore more intuitive to define the final security goal (*i.e.*, authentication property) for the RP server, which is the focus of this paper.

In particular, we have one key observation to secure the Single Sign-On service: *An RP server should login a user if and only if the exact user has actually authorized this specific RP*. To be more specific, an RP server can accept a user's login request in Step 5 of Fig. 1 if and only if the exactly same user has authenticated and/or authorized this specific RP in Step 3. Given this insight, we develop the predicates which must be satisfied by a secure SSO transaction, as presented in Listing 1.

The clause in Line 1 (Clause 1) asserts that the user stored by the RP session should be the owner of the received access token, so does the code and refresh token (if exist) in the second and third clauses. Clause 4 and Clause 5 assert that the access token and refresh token (if any) should be correctly passed to the intended RP, not to any other RPs (which would then use this token to log into *this* RP illegally). Clause 6 reflects the requirements that the final logged-in user should be the one who authenticates/ authorizes with the IdP. We know that S3KVetter simulates the IdP behavior. Therefore, the IdP's session data can be readily accessed by S3KVetter.

Listing 1: Security Property for SSO Services[6]

```
1 RPsession.uid == TokenRecordsOnIdP[
      RPsession.access_token].uid and
2 RPsession.uid == CodeRecordsOnIdP[
      RPsession.code].uid and
3 RPsession.uid == TokenRecordsOnIdP[
      RPsession.refresh_token].uid and
4 client_id == TokenRecordsOnIdP[RPsession
      .access_token].client_id and
5 client_id == TokenRecordsOnIdP[RPsession
      .refresh_token].client_id and
6 RPsession.uid == IdPsession.uid
```

By checking against the required list of security properties, one can effectively expose the presences of numerous vulnerabilities. Any violation of a security property can lead to a vulnerability in practice. For example, if Clause 1 does not hold, then it means the RP does not use the access token to identify the user, which can make profile attacks [47] possible. A more elaborated example is Clause 6, which can be violated in two different cases: (1) it is possible that an attacker eavesdrops the victim's `code` and uses it to sign into the RP (*i.e.*, `RPsession.uid = victim` **and** `IdPsession.uid = attacker`) ; (2) it can also be the result of a CSRF attack, in which the attacker makes the victim's browser to send the RP a crafted request with the attacker's `code` (*i.e.*, `RPsession.uid = attacker` **and** `IdPsession.uid = victim`).

5 Implementations of S3KVetter

We have implemented a full-featured prototype of S3KVetter in Python with 5064 lines of code. While its current implementation only focuses on SSO SDKs written in Python, our techniques can be naturally applied to SDK developed in other languages. To avoid reinventing the wheel, we have integrated and extended several open-source programs as supporting modules for S3KVetter. In Module 1 of Fig. 2, we extend PyExZ3 [6], a concolic execution engine for Python, to enhance the extraction of program predicates from production-level SDKs. We also substitute the default constraint solver of PyExZ3 (Z3) with CVC4 because the latter has better support for our heavily-used string operations with negligible performance penalty[7]. For Module 2 in Fig. 2, we choose SMT-Lib v2.0 which uses first-order logic with quantifier to represent the translated predicate tree. The logic language provided by SMT-Lib is not only expressive enough but also widely accepted by most theorem provers. This also allows us to directly use CVC4 in Module 3 of Fig. 2 to reason about the program predicates.

6 Evaluation

To determine the effectiveness of our approach, we perform evaluations on ten popular Single-Sign-On SDKs. S3KVetter shows considerable improvement in terms of code coverage when comparing to an unmodified symbolic execution engine (without our proposed extensions and heuristics). More importantly, we uncover four types of previously unknown vulnerabilities and provide new insights of SSO services.

6.1 Dataset

Table 1 shows the statistics of the SDKs under test. These SDKs are carefully selected from official references and high-profile open source SDKs in Github. In particular, they have covered the two most popular protocols (*i.e.*, OAuth2.0 and OpenID Connect) and both of the widely used authorization grant flows, namely, the implicit flow and the authorization code flow. The number of downloads for each SDK was retrieved on Oct 2017 from PyPI statistics [2] – a website which provides runtime statistics of PyPI published packages. Note that these statistics provide a conservative estimate on the usage of these SDKs: only the installation of released version via *pip* counts. If developers install a SDK directly from its source code (*e.g.*, via official webpage or Git), the suggested way for many IdPs (*e.g.*, Facebook, Weichat, Renren, Douban), then the installation will not be included in the statistics.

Regarding the lines of code, some libraries (*e.g.*, Request-OAuthLib and OAuthLib) are considerably larger. This is because those SDKs provide generalized, full-featured and specification-compliant support for *multiple* IdPs. In contrast, some small SDKs only implement simple and basic functions for a specific IdP.

[6] For ease of presentation, we use the line number to represent the clause. For example, the clause in the first line is denoted as Clause 1.

[7] CVC4 and Z3 perform very similarly in different benchmarks during the Satisfiability Module Theories (SMT) competition [4].

Table 1: Statistics of SDK under Study

SDK Names	Lines of code	# of downloads	Grant flow under study	Baseline with unmodified PyExZ3				Improved result with S3KVetter			
				# of path discovered	statement coverage	branch coverage	# of bugs discovered	# of path discovered	statement coverage	branch coverage	# of bugs discovered
Facebook SDK	976	602,291	implicit	8	45%	37%	2	40	58%	56%	2
Request-OAuthLib	15432	4,785,778	code	322	37%	31%	0	649	42%	35%	2
OAuthLib	17917	6,476,894	code	640	41%	33%	1	1282	46%	39%	5
Sinaweibopy	800	28,019	code	2	43%	39%	2	6	47%	44%	2
OAuth2Lib	971	not found	code	2	73%	68%	0	4	83%	77%	1
Rauth	9241	487,275	code	2	41%	34%	2	14	43%	36%	2
Python-weixin	2736	1,404	code	2	32%	29%	2	6	38%	35%	2
Boxsdk	15277	77,074	code	2	44%	37%	2	12	55%	47%	2
Renrenpy	251	10,387	code	2	54%	46%	1	12	56%	50%	1
Douban-client	2092	30,601	implicit	1	49%	52%	2	2	62%	60%	3

- [1]: Facebook SDK supports OIDC, and the other SDKs support OAuth2.0 protocol.

6.2 Experiment Setup and Performance

We run S3KVetter on an LXC instance of a Ubuntu 14.04 machine with 8 core CPU and 64GB memory. The testing of each SSO SDK can be completed within 5 seconds. Such runtime efficiency of S3KVetter can be attributed to the following 2 design decisions: Firstly, we internally simulate the external parties and thus spare S3KVetter from executing the most time-consuming network requests. Secondly, we concretely execute non-core methods. As such, the number of paths to be explored as well as the complexity of path constraint to be solved are significantly reduced. Without these two heuristics, it can take several minutes for testing even a small SDK.

6.3 Program Coverage

S3KVetter is able to overcome the fundamental weakness of traditional symbolic execution when dealing with multi-party, asynchronous distributed systems. By that, we mean that, when a conventional symbolic execution engine is unable to obtain correct/ meaningful results (*e.g.*, `code`) from external parties (and thus gets stuck in error-processing paths), S3KVetter can either "generate" valid results, or schedule to other paths, to continue exploring meaningful paths beyond the error-processing paths. Therefore, as shown in Table 1, S3KVetter can achieve 2%-13% higher statement coverage and 2%-19% higher branch coverage for the SDKs under test. Such coverage data is measured by coverage.py [1]. While increasing the code coverage by modifying a limited set of inputs is increasingly harder for higher values, even small increases in code statements can significantly discover more program paths.

Despite the improvement, we note that S3KVetter is far from achieving 100% coverage. This is in line with our expectation for two reasons: Firstly, a SDK often contains functions beyond the scope of SSO (*e.g.*, advertisement, notification, *etc.*). For example, Facebook has developed over 80 functions in their Graph API to support data ingestion and interchange for the Facebook's platform. These functions therefore are not considered by S3KVetter. Secondly, only a limited set of inputs (*e.g.*, Step 1, 5 and 10 in Fig. 1) can be controlled by an attacker. With such limited capability, the attacker can only reach part of the code statements. Since S3KVetter cannot reach more paths than the attacker, incomplete coverage is expected.

6.4 Vulnerabilities Discovered

As presented in Table 2, S3KVetter has found 7 types of vulnerabilities among these SDKs. While some vulnerabilities have been well studied in the literature, four of them are uncovered by S3KVetter for the first time. The damages of these newly discovered vulnerabilities vary depending on the specific implementations. The security impact can range from sniffing user activities at the RP, to the hijacking of the victim's RP account. There is only one requirement for the exploitation of these vulnerabilities[8]: the attacker needs to setup a malicious RP (mRP) and lure a victim user to login to the mRP. Once this condition is satisfied, the attacker can remotely control the victim's account of any RP which uses the vulnerable SSO SDK. We detail these newly discovered vulnerabilities in Section 7.

6.4.1 Detection Accuracy

We have manually verified all the reported vulnerabilities and found no false positive. However, S3KVetter can contain false negatives (like the state-of-the-art symbolic analysis techniques) for two main reasons. Firstly, our developed security property only focuses on the authentication issues. Yet, there may be other important properties. Secondly, S3KVetter may not be able to explore all execution paths due to the following limitations:

[8]For the use-before-assignment of the `state` variable, the requirement is even simpler: the victim just needs to visit a malicious web page.

Table 2: Summary of Discovered Vulnerabilities

SDK	Existing classes of vulnerabilities			New classes of vulnerabilities			
	Token substitution	no check of TLS	misuse or no use of state	use-before-assignment of state variable	Bypass MAC key protection	refresh_token injection	access_token injection
Facebook SDK	N	Y	Y	N.A	N.A	N	N
Request-OAuthLib	N	N	N	Y	N.A	Y	N
OAuthLib	Y	N	Y	N.A	Y	Y	Y
Sinaweibopy	N	Y	Y	N.A	N.A	N	N
OAuth2Lib	N	N	Y	N.A	N.A	N	N
Rauth	N	Y	Y	N.A	N.A	N	N
Python-weixin	N	Y	Y	N.A	N.A	N	N
Boxsdk	N	Y	Y	N	N.A	N	N
Renrenpy	N	N	Y	N.A	N.A	N	N
Douban-client	Y	Y	Y	N.A	N.A	N	N

- The underlying SMT solver assumes a query does not have a feasible solution when it takes too long to solve. However, it can be the case that the constraint under query is too complex. We cannot cover those feasible paths related to such a complex constraint.
- PyExZ3 uses class inheritance to track program execution. However, if the SDK explicitly casts the input data to native data type, PyExZ3 will lose the control for this variable (We seldom observe such cases in practice though).
- We concretely run non-core methods (*e.g.*, URL-encode) and do not check whether these non-core methods contain bugs.

6.5 Usability

It is straightforward to apply S3KVetter on an SSO SDK. Only two manual steps are required by an analyst. Firstly, the analyst should build a sample app, based on the SDK under test, so that S3KVetter can actually execute/ explore the app and thus the target SDK. Thanks to the widely available developer documentation and official sample codes, this step is relatively straightforward. Secondly, the analyst should mark which functions can be reached by which part of the attacker's input[9]. Although there can be thousands of functions in a SDK , the attacker usually can only reach very few of them. For example, only three functions of the Request-OAuthLib SDK can be directly invoked by an attacker. Given the small number of these functions, it becomes trivial to identify which part of the user inputs is symbolic. For instance, the Request-OAuthLib SDK authenticates a user only based on the variable of `request.url`. Therefore, only this variable is marked as symbolic (one example can be found in Appendix B). The other variables like cookies and HTTP headers, though controllable by an attacker, are treated as concrete since they are not processed by the SDK of interest.

[9]While we assume an attacker can control all packets sent to the RP server, only part of these packets would be processed by the SDK.

To apply S3KVetter on other multi-party systems, one additional manual step is to develop the required security properties (*i.e.*, the counterpart of Listing 1) for the specific domain of applications. Fortunately, the required security properties are high-level in nature and do not need to be developed by a domain expert. For example, the list of the required security properties for payment services can be developed by codifying the following statement: *A merchant* `M` *should accept an order if and only if the user has paid to the cashier in the correct amount for that specific order associated with merchant* `M`.

Note that the developed scurity property is not necessarily an exhaustive list of all protocol states. In fact, the analyst is free to specify the properties of interest. For instance, if an SSO system only supports the implicit call-flow (where the `code` variable is not involved), Clause 2 in Listing 1 is no longer needed. Note also that S3KVetter is agnostic to how the security properties are derived. While other researchers have managed to automatically extract the required security properties from the source code [5] or protocol specification [16], their results are complementary to ours and can be adopted to further extend the capabilities of S3KVetter.

6.6 Comparison with Existing Testing Tools for SSO

To the best of our knowledge, there is no existing work (except [49]) which performs comprehensive blackbox fuzzing/ testing on SSO SDKs.

- [18, 33, 51] build tools to check specific, previously known vulnerabilities (e.g.,CSRF), but could not discover new ones.
- While our earlier work on model-based security testing for OAuth2.0 (OAuthTester) [49] has the potential, at least in theory, to discover all the vulnerabilities listed in Table 2, our testing shows that OAuthTester can only detect two out of the seven types of vulnerabilities (TLS and `state` misuse) listed. This is because some vulnerabilities discovered by S3KVetter can only be triggered under very specific

conditions. Without the source code, it is very difficult for blackbox-testers (like [49]) to uncover such fine-grain, condition-specific problem.

7 Case Study of Vulnerabilities Discovered

7.1 Access_token Injection

As the result of SSO, an access token is issued to the RP. Based on the access token, the RP can identify the user. The authenticity of the access token is therefore a critical security requirement. As such, many IdPs (*e.g.*, Facebook, Sina) have provided an access_token-debug API for RPs to verify the access tokens they received. This API is heavily used by RPs running the implicit flow [13] but seldom by those implementing the authorization-code flow. This is because an access token obtained via the authorization-code flow is generally believed to be secure by SDK developers or IdPs. Such belief is based on the fact that, under the authorization-code flow, the access token is exchanged over a secure TLS connection routed directly between the IdP and RP, without passing through the mobile (client) device which may be controlled/ tampered by the attacker. However, we will show that an access token obtained using the authorization-code flow can still be insecure under the presence of the so-called "access_token injection" vulnerability. This vulnerability is caused by the ill-conceived design of SSO SDKs. For any RP using a SDK with the "access_token injection" vulnerability, an attacker can remotely inject any access token of her choice to the vulnerable RP. As a result, as long as the attacker can obtain a valid (but different) access token of Alice (*e.g.*, by luring Alice to login to a malicious RP controlled by the attacker), the attacker can log into the vulnerable RP as Alice.

Listing 2: Root Cause of Access Token Injection and Bypass MAC Key Protection in OAuthLib

```
def _populate_attributes(self, resp):
  if 'code' in resp:
    self.code = resp.get('code')
  if 'access_token' in resp:
    self.access_token = resp.get('
        access_token')
  if 'mac_key' in resp:
    self.mac_key = resp.get('mac_key')
```

7.1.1 Vulnerability Analysis

Below, we use OAuthLib [21], a popular SDK with more than 6 million downloads, to illustrate this vulnerability. When the IdP passes the `code` parameter to the RP in Step 5 of Fig. 1, this SDK will first verify the correctness of this response. For example, it checks whether it is a secure channel and the `state` parameter to protect against CSRF attacks. Thereafter, it calls the function of **_populate_attributes** to populate/ store some commonly used variables for later use. However, if this function is not carefully designed, an attacker can control the value to be stored.

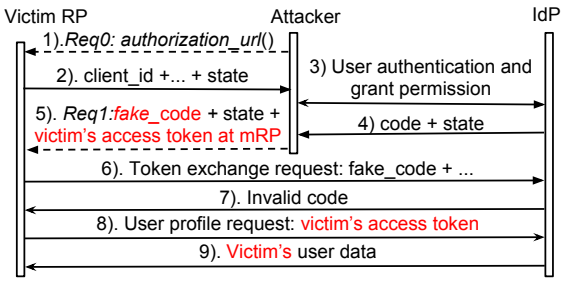

Figure 7: Exploit for access token injection

As presented in Listing 2, this SDK stores the value of `code` if it exists in the response `resp` (*i.e.*, Step 5 in Fig 1). Surprisingly, if the response `resp` contains `access_token`, its value is also stored. More specifically, if an attacker feeds the URL input shown in Listing 3 to the RP in Step 5, an attacker-controlled access token is stored by the SDK and used for authentication later on. In this case, two security properties are violated. Firstly, Clause 4 is violated since the victim RP uses the access token issued to mRP. Secondly, Clause 6 is also violated: the IdP believes the current user is the attacker while the RP thinks she/ he is the victim.

Listing 3: An Exploit URL for Access Token Injection

```
https://RP.com?state=xxx&code=fake_code
    &access_token=victim_access_token_at_mRP
```

7.1.2 Exploit

The exploit only requires the attacker to obtain Alice's access token, *e.g.*, via a malicious RP. As presented in Fig 7, the attack procedure is as follows:

1-4. The attacker logs into a victim RP using her own IdP account and her own device.

5. The attacker intercepts and substitutes the normal response with an invalid `code` as well as the victim Alice's access token of *mRP*.
6. After verifying the response, the SDK stores the code and Alice's access token. The SDK then makes a token exchange request with this fake code.
7. Since the code is invalid, the IdP returns error. Therefore, the previously stored access token will not be overwritten.
8. The RP retrieves the user data using Alice's access token.
9. The IdP returns Alice's user information and thus the attacker can log into the victim RP as Alice.

7.2 Refresh_token Injection

For SSO protocols, an access token often has a short lifespan, just enough to cover the typical duration of a login session. Thereafter, the RP will need to prompt the user to perform re-authorization, which can degrade user experience. To avoid this problem, it is common for an IdP to issue another long-term "refresh token" to the RP, together with the initial access token. The RP can subsequently use the refresh token to request a new access token from the IdP without user intervention. As such, the mishandling of this refresh token can have severe security consequences similar to that of the access token.

It is generally believed that the refresh token is secure since it is delivered over a secure channel (together with the access token) in Step 7 of Fig. 1. Meanwhile, some SDK developers have enough security expertise and realize the risk of directly storing the value from the end-user (*e.g.*, the access token injection vulnerability). Therefore, these SDK developers attempt to pre-process the user input and stores it only after it has passed the security checkings.

Despite these seemingly strict security checks, we will show that the so-called refresh_token injection vulnerability is still possible. This vulnerability enables an attacker to specify any refresh token of her choice and then login as the victim. Below, we use the Request-OAuthLib SDK, which supports auto-token-refresh mechanism, to illustrate the problem.

7.2.1 Vulnerability Analysis

This vulnerability, though superficially similar to the access token injection, is actually more complicated. The first step is similar: this SDK checks the refresh token in Step 5 of Fig. 1, and if exists, stores it in the variable of `oauth._client.refresh_token`. The difference is that this SDK realizes such a variable is highly security sensitive and attempts to apply more secure measures to protect/ verify it (but still fails). Such attempts are presented in Listing 4 with much simplification for the ease of presentation.

Specifically, this SDK first checks whether there is a refresh token either in the arguments provided by the API caller or in the `oauth.token` object delivered via a secure server-to-server communication. Unfortunately, the former by default is `None` and the latter can be indirectly manipulated/ controlled by the attacker. For example, the attacker can feed an invalid `code` in Step 5 of Fig. 1 so that the `oauth.token` object will not be overwritten by a refresh token exchanged with the IdP server. In this case, `oauth.token` will use its default value `None`. As such, the attacker can invoke the `prepare_refresh_body` function with an argument of `refresh_token` = `None`. The `prepare_refresh_body` function therefore has no choice but to use the attacker-controlled variable of `oauth._client.refresh_token`.

Listing 4: Attempts to Filter User Input

```
def refresh_token(self, refresh_token =
    None, **kwargs):
  # self.token is the oauth.token object
  refresh_token = refresh_token or
    self.token.get(@'refresh_token'@)
  ...
  body = self._client.
      prepare_refresh_body(body=body,
      refresh_token=refresh_token, scope
      =self.scope, **kwargs)
```

7.2.2 Exploit

There exist multiple exploits for this vulnerability. Below, we present one exploit which requires the least capability of the attacker (Eve): As long as Eve can obtain Alice's refresh token associated with a malicious RP (run by Eve), Eve can login as Alice to any RP which uses the vulnerable SDK (as shown in Fig. 8):

1-4. The attacker follows the normal protocol flow to log into the victim RP using her own IdP account with her own device.

5. When the IdP returns an authorization code, the attacker then injects the victim's refresh token.

6. Once the access token expires, the SDK will automatically renew the access token using Alice's refresh token.

7. The IdP then returns Alice's access token to the RP according to the refresh token.

When the RP uses this newly obtained access token to retrieve the user data, the IdP will return the victim's information. The damage depends on how the user data is utilized. In the worst case where the user data is for authentication, the attacker can log into the vulnerable RP as the victim user.

Note that the above exploit only works for those IdPs (*e.g.*, Fitbit) which do not require `client_secret` in Step 6 of Fig. 8. For specification-compatible IdPs requiring this parameter, we need to assume a stronger threat model: the attacker can obtain the victim's refresh token issued for the vulnerable RP.

7.3 Use-before-assignment of `state`

To thwart CSRF attacks, the OAuth2.0 specification [23] strongly suggests the use of the `state` parameter, which should be generated and handled as a nonce. Note that the process of the `state` parameter is tightly related to

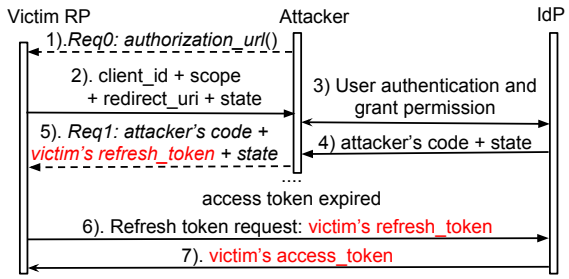

Figure 8: Exploit for refresh token injection

session management, for which the application developers have multiple options. It is therefore difficult for the SDK, which is supposed to define the core functionality only, to consider the different operations among numerous session management tools. This may explain why 9 out of 10 SDKs (see Table 2) are vulnerable to different existing attacks related to the `state` parameter: These SDK developers often rely on the RP developers to implement the `state` parameter by themselves. Unfortunately, as shown in [49], 55% RP implementations fail to handle this `state` parameter correctly.

Towards this end, the Request-OAuthlib SDK pays considerable attention to carefully implement the `state` parameter and has fixed all previously known vulnerabilities associated with this parameter. Unfortunately, the fix itself unexpectedly contains a new bug, making CSRF attack possible again (but in a different way). By leveraging the CSRF attack, the attacker can either spoof the victim's personal data [43] or control the victim's RP account [49].

7.3.1 Vulnerability Analysis

Listing 5 presents the vulnerable code snippet when using the `state` parameter. It contains three key functions: init(), callback() and profile(), which correspond to `Req0`, `Req1` and `Req2` in Fig. 1, respectively. When the user clicks the "login with Facebook" button, the browser will send `Req0` to the RP server and invokes the "init" function. This function generates an authorization URL (Line 5) which includes a random `state` parameter to prevent CSRF attacks: Upon receiving `Req1`, the "callback" function will be invoked to parse and verify `auth_response`. In particular, it compares the `state` parameter generated in Line 4 and the one in the `auth_response` in Line 17 (which was stored in the `params` variable). In case of mismatch, an error will occur.

At a first glance, the program appears to be correct. However, a so-called "use-before-assignment" vulnerability of the `state` variable exists. Specifically, if an attacker skips `Req0` (thus "init" function does not get executed), and instead directly sends `Req1` to invoke the

Listing 5: Root Cause of Use-before-Assignment of State Variable

```
1  oauth = OAuth2Session(client_id,...)
2  @app.route("/")
3  def init():
4    auth_url, state = oauth.
         authorization_url(base_url)
5    return redirect(auth_url)
6  @app.route("/callback", methods=["GET"])
7  def callback():
8    token = oauth.fetch_token(token_url,
         secret, auth_response=request.url)
9    session['oauth_token'] = token
10   return redirect(url_for('.profile'))
11 @app.route("/profile", methods=["GET"])
12 def profile():
13   return oauth.get('https://idp/user')
14
15 def fetch_token(token_url, secret,
     auth_response):
16   ...
17   if state and params.get("state", None)
         !=state:
18     raise MismatchingStateError()
```

"callback" function, then the first occurrence of `state` in Line 17 becomes the default value, *i.e.*, `None`. As a result, the program will not check the second condition (`params.get("state",None)` != `state`). Instead, it directly exchanges for an access token (as long as the other fields in Step 6 of Fig. 1 are valid).

7.3.2 Exploit

This vulnerability allows an attacker to bypass the verification of the `state` parameter and thus makes CSRF attacks possible again. The exploit is presented in [43] (Section 4.4). Specifically, an attacker performs the following steps:

1. Sign into an RP using her own account from the IdP,
2. Intercept the code on her browser (Step 5 in Fig 1) and then,
3. Embed the intercepted code in an HTML construct (e.g., img, iframe) that causes the browser to automatically send the intercepted code to the RP's sign-in endpoint when the exploit page is viewed by a victim user.

This vulnerability can have high security implication, ranging from sniffing the victim's activity at the vulnerable RP via a "login CSRF" attack [8], to controlling the victim's RP account by account hijacking attack [26]. When it is combined with the amplification attack via Dual-Role IdPs [49], the consequence can be even more severe. Refer to the above references for details of the

corresponding exploits.

7.4 Bypass MAC_key Protection

SSO protocols support two usage types for an access token: the commonly used bearer token and the yet-to-be-standardized MAC token. Fig. 1 shows the standard use of the bearer token: any party in possession of an access token can retrieve the user data hosted by the IdP. Therefore, if the access token is disclosed (*e.g.*, via eavesdropping or insecure storage), an attacker can directly login as the token owner [13]. To protect the access token against leakage, more and more IdPs (*e.g.*, Facebook, Sina, *etc.*) start to support the MAC token.

The MAC token protocol is supposed to be more secure by signing the original bearer token. Specifically, in Step 7 of Fig. 1, MAC-token-enabled IdPs will return a random *secret key*[10] along with the access token to the RP. When making user-profile requests, the RP needs to compute a cryptographic hash message (*e.g.*, HMAC-SHA-256) to prove its possession of the secret key. Only if both the hash value (MAC) and the access token are valid would the IdP return the user data to the RP. Unfortunately, some SDKs cannot implement this function correctly. As a result, the purpose of MAC token is totally broken.

7.4.1 Vulnerability Analysis and Exploit

As presented in Listing 2, an attacker can specify any secret key of her choice using the following input:

```
https://RP.com/callback?state=xxx&code=
    fake_code_value&access_token=victim
    access_token&mac_key=victim_mac_key
```

Fig. 9 presents the exploit, which is similar to Fig. 7 with two exceptions: At Step 5, besides an invalid code and the victim's access token, the attacker also feeds a *MAC key* of mRP. At Step 8, the RP retrieves the user data using the victim's access token and the MAC value computed by the MAC key. Since the access token and MAC key are paired, the IdP returns the victim's user data to the RP for authentication.

8 Lessons Learned

Least privilege. We find that the aforementioned vulnerabilities are largely caused by the failure of the SDK developers in adhering to the principle of least privilege. Specifically, during each message exchange, the SDK developer should design a separate function to store the corresponding variable/ parameter so that the SDK can easily decide whether a variable/ parameter can be accessed and/or altered by the user or not. However, many SDK developers, for simplicity, store all key variables/ parameters using one single function. Furthermore, this function can be invoked by the user. As a result, even if the SDK developers attempt to filter out the user-provided variables, an intelligent attacker can still manipulate sensitive variables (*e.g.*, access token, refresh token) that she should not be allowed to.

[10]Previously, the secret key was the app secret, which is generated when the RP registers in the IdP platform. But the updated draft has made it a session secret and will be delivered upon every authorization request.

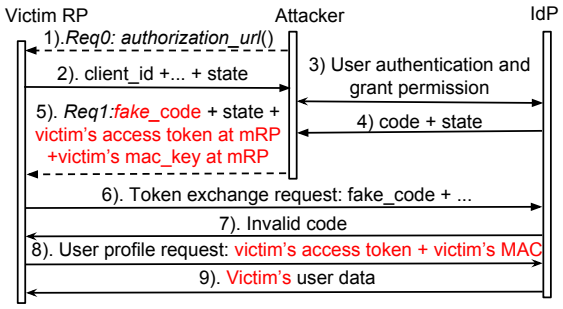

Figure 9: Exploit for MAC key injection

Less is more. Another observation is that the more IdPs/ functions a SDK supports, the more susceptible it would be. The reason is that, since the SSO specifications only serve as a high-level guideline, IdPs typically have various application-specific logic flows, unique APIs and security checks. To support multiple IdPs, a SDK will need to develop an additional layer to provide a new, generalized interface to glue various IdP-specific implementations together. For example, the Request-OAuthLib SDK defines two objects (*i.e.*, `oauth._client` and `oauth.token`) to manage the OAuth-related variables. When making requests to different IdPs, the SDK can thus retrieve the required variable from these two objects. Unfortunately, this generalized interface has enable the most important attack vector. would like to provide, the more vulnerable it can be. *e.g.*, OAuthLib, Request-OAuthLib.

9 Related Work

SSO security analysis. Given the critical SSO services, extensive efforts have been devoted to their security analysis. Firstly, the protocol specification [23, 39] has been verified by different formal methods including model checking [5, 7, 15, 19, 20, 36], manual analyses [28, 32] and cryptographic proof [11]. These formal methods have uncovered different protocol design flaws. However, these methods are mainly used to prove the correctness (or find violations) of the specification. As a result, the discovered vulnerabilities may not be realistic and can be unexploitable (unlike ours). For example, al-

though [19] discovers the so-called 307 Redirect attack that allows an attacker to learn the victim's password in IdP, real-world SSO systems actually use 302 redirection instead.

Despite these theoretical works, the practical implementations of the protocols were often found to be incorrect due to the implicit assumptions enforced by the IdP SDKs [46] or the incorrect interpretation of ambiguous specification [13]. Towards this end, researchers start to analyze the security issues of real-world implementations. The most popular method relies on network traffic analysis [25, 30, 43–45, 48, 49], to infer a correct system model for guiding subsequent fuzzing. Another attempt was to analyze how the security issues of the underlying platform can affect the SSO security, as discussed in [13, 47]. Motivated by numerous types of vulnerabilities discovered by these methods, researchers have built different automatic tools [18, 33, 51] to perform large-scale testing of SSO implementations against known classes of vulnerabilities. These studies do not consider the security of SDK internals and thus are different from ours in nature.

The work most similar to ours should be [46] which identifies the implicit assumptions in order for an SSO SDK to be used in a secure way. However, their work requires labor-intensive code translation for each SDK. As a result, the scheme is not scalable and the resultant semantic model can be inaccurate. More importantly, they focus on how a SDK can be insecurely used while we concern the vulnerabilities of SDK internals, which can be exploited even if the RP developers strictly follow the Best Current Practices. can be insecure by itself.

SDK security analysis. Modern software is often developed on the top of SDKs. To detect the SDK usage errors, many different tools and methodologies have been proposed. Most of these works focus on checking whether the SDK follow a specification, which can be either manually specified (*e.g.*, SSLint [24]), extracted from code [5] or learned from other libraries [35, 50]. However, all of them emphasize on the API invocation patterns. In contrast, relatively few efforts have been devoted to the security analysis on the SDK internals.

Asynchronous events studies. Previous research has shown that asynchronous events can lead to serious problems. Petrov *et al.* [37] formulate a *happens-before relation* to strictly specify the web event orders (*e.g.*, script loading should happen before execution) for detecting dangerous race-conditions in web applications. Such a *happens-before relation* was developed based on in-depth study of relevant specifications (*e.g.*, those of HTML and Javascript) and browser behavior. As such, it is rather difficult to generalize their findings to cover other protocols. Furthermore, the *happens-before relation* cannot characterize the much more complicated security properties of multi-party SSO protocols. Another related work is CHIRON [27], which can detect semantic bugs of stateful protocol implementations by considering different request orders. However, CHIRON mainly focuses on two-party systems and cannot maintain a consistent system state for more general multiple party systems. As a result, the work cannot be readily applied to the 3-party SSO system.

Symbolic execution. Using systematic path exploration techniques, symbolic execution tools like KLEE [9], S2E [14], UC-KLEE [38] are very effective in non-distributed software bug detection, especially for low-level memory corruption problems [41] (but not for web apps). More recently, the symbolic execution approach [10, 31, 40] has been extended to handle asynchronous apps (*e.g.*, OpenFlow and sensor networks) where events of interest can occur at any time. However, previous extensions require expert-level domain knowledge and cannot be applied for general asynchronous apps. Researchers have also used symbolic execution to verify web applications (*e.g.*, [12, 42]), but they did not consider challenges arise from multi-lock-step operations or the multi-party coordination. In contrast, S3KVetter has developed new techniques to test the implementations of multi-party protocols/ systems.

10 Conclusion

In this paper, we have presented S3KVetter, an automated testing tool which can discover logic bugs/ vulnerabilities buried deep in SSO SDKs by utilizing symbolic reasoning techniques. To better explore a 3-party SSO system, we developed new techniques for symbolic execution and realized them in S3KVetter. We have evaluated S3KVetter on ten popular SSO SDKs/ libraries which support different SSO protocols and modes of authorization grant flow. In addition to existing vulnerabilities, S3KVetter successfully discovers 4 new types of vulnerabilities, all of which can result in serious consequences including application account hijacking or user privacy leakage. Our findings demonstrate the efficacy of S3KVetter in performing systematic reasoning on SDKs and provide a reality-check on the implementation quality of popular "industrial-strength" SSO SDKs.

Acknowledgements and Responsible Disclosure

We thank our shepherd Prof. Cristina Nita-Rotaru and the anonymous reviewers for their valuable comments which help to improve the paper considerably. This work is supported in part by the Innovation and Technology Commission of Hong Kong (project no.

ITS/216/15), National Natural Science Foundation of China (NSFC) under Grant No. 61572415, the CUHK Technology and Business Development Fund (project no. TBF18ENG001), and Hong Kong S.A.R. Research Grants Council (RGC) Early Career Scheme/General Research Fund No. 24207815 and 14217816.

We have reported the newly discovered vulnerabilities to all the affected vendors and have received various confirmations and acknowledgments.

References

[1] Code Coverage. https://coverage.readthedocs.io.

[2] PyPI statistics. http://www.pypi-stats.com/package/.

[3] Requests-OAuthLib. https://github.com/requests/requests-oauthlib.

[4] Satisfiability modulo theories competition. http://smtcomp.sourceforge.net/2017/.

[5] BAI, G., LEI, J., MENG, G., VENKATRAMAN, S. S., SAXENA, P., SUN, J., LIU, Y., AND DONG, J. S. AUTHSCAN: automatic extraction of web authentication protocols from implementations. In *NDSS* (2013).

[6] BALL, T., AND DANIEL, J. Deconstructing dynamic symbolic execution. *Dependable Software Systems Engineering 40* (2015), 26.

[7] BANSAL, C., BHARGAVAN, K., AND MAFFEIS, S. Discovering concrete attacks on website authorization by formal analysis. In *CSF* (2012).

[8] BARTH, A., JACKSON, C., AND MITCHELL, J. C. Robust defenses for cross-site request forgery. In *CCS* (2008), ACM.

[9] CADAR, C., DUNBAR, D., AND KLEE, D. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proc. Operating System Design and Implementation (OSDI 08)*, pp. 209–224.

[10] CANINI, M., VENZANO, D., PERESINI, P., KOSTIC, D., AND REXFORD, J. A nice way to test openflow applications. In *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (2012), no. EPFL-CONF-170618.

[11] CHARI, S., JUTLA, C. S., AND ROY, A. Universally composable security analysis of OAuth v2.0. Cryptology ePrint Archive, Report 2011/526, 2011.

[12] CHAUDHURI, A., AND FOSTER, J. S. Symbolic security analysis of ruby-on-rails web applications. In *CCS* (2010), ACM.

[13] CHEN, E. Y., PEI, Y., CHEN, S., TIAN, Y., KOTCHER, R., AND TAGUE, P. OAuth demystified for mobile application developers. In *CCS* (2014), pp. 892–903.

[14] CHIPOUNOV, V., KUZNETSOV, V., AND CANDEA, G. S2E: A platform for in-vivo multi-path analysis of software systems. *ACM SIGPLAN Notices* (2011).

[15] D. FETT, R. KÜSTERS, AND G. SCHMITZ. An expressive model for the web infrastructure: Definition and application to the Browser ID SSO system. In *IEEE Symp. on Security and Privacy, S&P* (2014).

[16] DITTMER, M. S., AND TRIPUNITARA, M. V. The unix process identity crisis: A standards-driven approach to setuid. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security* (2014), ACM, pp. 1391–1402.

[17] FERRERO, N. OAuth2Lib. https://github.com/NateFerrero/oauth2lib.

[18] FERRY, E., O'RAW, J., AND CURRAN, K. Security evaluation of the OAuth 2.0 framework. *Inf. & Comput. Security 23*, 1 (2015), 73–101.

[19] FETT, D., KÜSTERS, R., AND SCHMITZ, G. A comprehensive formal security analysis of OAuth 2.0. In *CCS* (2016).

[20] FETT, D., KÜSTERS, R., AND SCHMITZ, G. The Web SSO Standard OpenID Connect: In-Depth Formal Security Analysis and Security Guidelines. In *IEEE 30th Computer Security Foundations Symposium (CSF)* (2017).

[21] GAZIT, I. OAuthLib. https://github.com/idan/oauthlib.

[22] GODEFROID, P., KLARLUND, N., AND SEN, K. Dart: directed automated random testing. In *ACM Sigplan Notices* (2005), ACM.

[23] HARDT, D. The OAuth 2.0 authorization framework, 2012. RFC 6749.

[24] HE, B., RASTOGI, V., CAO, Y., CHEN, Y., VENKATAKRISHNAN, V., YANG, R., AND ZHANG, Z. Vetting SSL usage in applications with SSLint. In *Security and Privacy (S&P), 2015 IEEE Symposium on* (2015), IEEE, pp. 519–534.

[25] HOMAKOV, E. *The Achilles Heel of OAuth or Why Facebook Adds Special Fragment*.

[26] HOMAKOV, E. *The Most Common OAuth2 Vulnerability*.

[27] HOQUE, E., CHOWDHURY, O., CHAU, S. Y., NITAROTARU, C., AND LI, N. Analyzing operational behavior of stateful protocol implementations for detecting semantic bugs. In *DSN* (2017).

[28] HU, P., YANG, R., LI, Y., AND LAU, W. C. Application impersonation: problems of OAuth and API design in online social networks. In *Proceedings of the second ACM conference on Online social networks* (2014), ACM.

[29] JANRAIN. Social login continues strong adoption.

[30] JING, W. *Covert Redirect Vulnerability*.

[31] KOTHARI, N., MILLSTEIN, T., AND GOVINDAN, R. Deriving state machines from tinyos programs using symbolic execution. In *Proceedings of the 7th international conference on Information processing in sensor networks* (2008), IEEE Computer Society, pp. 271–282.

[32] MAINKA, C., MLADENOV, V., AND SCHWENK, J. Do not trust me: Using malicious IdPs for analyzing and attacking Single Sign-On. In *Security and Privacy (EuroS&P), 2016 IEEE European Symposium on* (2016), IEEE, pp. 321–336.

[33] MAINKA, C., MLADENOV, V., SCHWENK, J., AND WICH, T. Sok: Single sign-on security–an evaluation of openid connect. In *EuroS&P* (2017).

[34] MLADENOV, V., MAINKA, C., KRAUTWALD, J., FELDMANN, F., AND SCHWENK, J. On the security of modern Single Sign-On protocols: OpenID Connect 1.0. *CoRR* (2015).

[35] NGUYEN, H. A., DYER, R., NGUYEN, T. N., AND RAJAN, H. Mining preconditions of APIs in large-scale code corpus. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* (2014), ACM, pp. 166–177.

[36] PAI, S., SHARMA, Y., KUMAR, S., PAI, R. M., AND SINGH, S. Formal verification of OAuth 2.0 using Alloy framework. In *Communication Systems and Network Technologies (CSNT)* (2011), IEEE.

[37] PETROV, B., VECHEV, M., SRIDHARAN, M., AND DOLBY, J. Race detection for web applications. In *ACM Sigplan Conference on Programming Language Design and Implementation* (2012), pp. 251–262.

[38] RAMOS, D. A., AND ENGLER, D. R. Under-constrained symbolic execution: Correctness checking for real code. In *USENIX Security* (2015), pp. 49–64.

[39] SAKIMURA, N., BRADLEY, J., JONES, M., DE MEDEIROS, B., AND MORTIMORE, C. OpenID Connect core 1.0.

[40] SASNAUSKAS, R., LANDSIEDEL, O., ALIZAI, M. H., WEISE, C., KOWALEWSKI, S., AND WEHRLE, K. Kleenet: discovering insidious interaction bugs in wireless sensor networks before deployment. In *Proceedings of the 9th ACM/IEEE International Conference on Information Processing in Sensor Networks* (2010), ACM, pp. 186–196.

[41] STEPHENS, N., GROSEN, J., SALLS, C., DUTCHER, A., WANG, R., CORBETTA, J., SHOSHITAISHVILI, Y., KRUEGEL, C., AND VIGNA, G. Driller: Augmenting fuzzing through selective symbolic execution. In *Proceedings of the Network and Distributed System Security Symposium* (2016).

[42] SUN, F., XU, L., AND SU, Z. Detecting logic vulnerabilities in e-commerce applications. In *NDSS* (2014).

[43] SUN, S., AND BEZNOSOV, K. The devil is in the (implementation) details: an empirical analysis of OAuth SSO systems. In *CCS* (2012).

[44] WANG, H., ZHANG, Y., LI, J., LIU, H., YANG, W., LI, B., AND GU, D. Vulnerability assessment of OAuth implementations in Android applications. In *ACSAC* (2015).

[45] WANG, R., CHEN, S., AND WANG, X. Signing me onto your accounts through Facebook and Google: A traffic-guided security study of commercially deployed Single-Sign-On web services. In *S&P* (2012).

[46] WANG, R., ZHOU, Y., CHEN, S., QADEER, S., EVANS, D., AND GUREVICH, Y. Explicating SDKs: Uncovering assumptions underlying secure authentication and authorization. In *USENIX Security* (2013).

[47] YANG, R., AND LAU, W. C. Breaking and fixing mobile app authentication with OAuth2.0-based protocols. In *ACNS* (2017).

[48] YANG, R., LAU, W. C., AND LIU, T. Signing into one billion mobile app accounts effortlessly with OAuth 2.0. In *Black Hat, Europe* (2016).

[49] YANG, R., LI, G., LAU, W. C., ZHANG, K., AND HU, P. Model-based security testing: An empirical study on OAuth 2.0 implementations. In *AsiaCCS* (2016).

[50] YUN, I., MIN, C., SI, X., JANG, Y., KIM, T., AND NAIK, M. APISan: Sanitizing API usages through semantic cross-checking. In *25th USENIX Security Symposium (USENIX Security 16)*.

[51] ZHOU, Y., AND EVANS, D. SSOScan: Automated testing of web applications for Single Sign-On vulnerabilities. In *USENIX Security* (2014).

A Detailed Description of the Authorization Code Flow of OAuth2.0

The individual steps of authorization code flow, as shown in Fig. 1, are detailed below:

1. The user initiates the SSO process with the RP by specifying his intended IdP;
2. The RP redirects the user to the IdP for authentication. The RP may include the optional `state` parameter which is used for binding the request (in Step 2) to the subsequent response in Step 5;
3. The user operates the client device (*e.g.*, the browser or the mobile app) to authenticate himself to the IdP. He also confirms with the IdP to grant the permissions requested by the RP.
4. The IdP returns to the user an authorization code with the optional `state` parameter (typically its value is the hash of cookies and a nonce).
5. The user is redirected to the RP. The RP would reject the request if the received `state` parameter does not match the one, if specified, in Step 2.
6. The RP then requests the access token directly from the IdP (without going through the user/ client device) by sending the `code` parameter and its *client secret*.
7. The IdP responds with an access token upon validation of the identity of the RP and the `code` parameter submitted by the RP.
8. Using this access token, the RP can request data of the user from the IdP server.
9. The IdP responds to the RP with the user data (*e.g.*, profile) so that the RP can confirm the user's identity and allow the user to login to the RP.
10. The user can subsequently request to access his information/ resource, e.g. the user profile, hosted by the RP server.
11. The RP server responds to the user with the requested information accordingly.

B Marking Symbolic Variables

Given the marked sample app, S3KVetter must identify which (ranges of) symbolic input fields (*e.g.*, the entire `request.url` or just the `code` in Listing 6) determine a path and then extracts all the path constraints related to these fields. To reduce the overhead for the constraint solver[11], we maintain each input field as an individual symbolic variable (*e.g.*, `code`, `state`) once these fields are split or decoded. Yet, we still allow byte-level access to the entire symbolic input (*e.g.*, `request.url`) in case we cannot identify input fields correctly.

Listing 6: Example for marking symbolic variables

```
1 @symbolic(request.url='http://RP.com/
      callback?code=code&state=1234'})
2 def callback():
3   token = oauth.fetch_token(token_url,
        secret, auth_response=request.url)
4   ...
```

[11] Otherwise, the constraint solver needs to remember all the operations on the entire symbolic input.