Model-based Security Testing: an Empirical Study on OAuth 2.0 Implementations

Yang Ronghai
The Chinese Univ. of HK
yr013@ie.cuhk.edu.hk

Guanchen Li
The Chinese Univ. of HK
leeguanchen@gmail.com

Wing Cheong Lau
The Chinese Univ. of HK
wclau@ie.cuhk.edu.hk

Kehuan Zhang
The Chinese Univ. of HK
khzhang@ie.cuhk.edu.hk

Pili Hu
The Chinese Univ. of HK
e@hupili.net

ABSTRACT

Motivated by the prevalence of OAuth-related vulnerabilities in the wild, large-scale security testing of real-world OAuth 2.0 implementations have received increasing attention lately [31, 37, 42]. However, these existing works either rely on *manual discovery* of new vulnerabilities in OAuth 2.0 implementations or perform automated testing for *specific, previously-known* vulnerabilities across a large number of OAuth implementations. In this work, we propose an adaptive model-based testing framework to perform automated, large-scale security assessments for OAuth 2.0 implementations in practice. Key advantages of our approach include (1) its ability to identify existing vulnerabilities and discover new ones in an automated manner; (2) improved testing coverage as all possible execution paths within the scope of the model will be checked and (3) its ability to cater for the implementation differences of practical OAuth systems/ applications, which enables the analyst to offload the manual efforts for large-scale testing of OAuth implementations. We have designed and implemented OAuthTester to realize our proposed framework. Using OAuthTester, we examine the implementations of 4 major Identity Providers as well as 500 top-ranked US and Chinese websites which use the OAuth-based Single-Sign-On service provided by the formers. Our empirical findings demonstrate the efficacy of adaptive model-based testing on OAuth 2.0 deployments at scale. More importantly, OAuthTester not only manages to rediscover various existing vulnerabilities but also identify several previously unknown security flaws and new exploits for a large number of real-world applications implementing OAuth 2.0.

Keywords

Single Sign-On; OAuth 2.0; Security Testing

1. INTRODUCTION

The OAuth 2.0 protocol has been adopted by mainstream Single-Sign-On services to support user authentication and authorization for 3rd party applications. Using OAuth 2.0, an identity provider (IdP) (e.g. Facebook) can grant data access privileges by issuing an "access token" to a third-party application (App) (e.g. Priceline) upon approval by the user. The App can then use the access token to retrieve the protected user data hosted by the IdP and thus assume the identity of the authorizing user without knowing his/ her log-in credential.

Due to the complexity of the multi-party interactions and trust relationships in OAuth 2.0, numerous security problems and practical attacks such as Covert Redirect [24] and App Impersonation [21] have been discovered recently. Worse still, many third-party application developers may not have the resource or know-how to properly implement OAuth-based services. As a result, the security analysis and testing of deployed OAuth 2.0 applications/systems have received increasing attention lately [17, 31, 37, 42]. However, all these studies either rely on *manual analysis* to discover new vulnerabilities in OAuth 2.0 implementations or perform automated testing for a limited set of *specific, previously-known* vulnerabilities across a large number of OAuth implementations. In contrast, we propose an adaptive model-based testing tool, OAuthTester, which enables the automatic discovery of vulnerabilities via systematic testing and evaluation of OAuth implementations.

Based on the OAuth protocol specification, we first construct an initial state-machine-based system model which abstracts and defines the expected behavior of the multi-party system, namely, the IdP, the 3rd-party App and the authorizing User. OAuthTester then automatically augments the initial model with implementation specifics extracted from network traces. Based on the resultant model, OAuthTester autonomously generates and executes test cases (in the form of HTTP request sequences) and collects responses from the IdP and App to adapt, augment as well as rectify the current system model in a self-learning manner. In particular, OAuthTester compares the expected behavior predicted by the current system model with the observed responses to determine whether the test target is vulnerable. Deviation of the observed responses from the predicted behavior is an indication that either 1) possible vulnerabilities are discovered or 2) the current model overlooks some situations/ corner cases of the system and OAuthTester would then rectify the system model automatically after confirming that no predefined security properties have been violated. OAuthTester continues this iterative process until all the paths in the evolving state machine have been covered.

While previous works also build a state machine to drive the verification of OAuth 2.0[1], their models are built only based on either the protocol specification [5] or some implementation details extracted from the related network traces [37, 39], but not both. In

[1] For the rest of the paper, we use OAuth to denote OAuth 2.0 if not specified otherwise.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ASIA CCS '16, May 30-June 03, 2016, Xi'an, China

© 2016 ACM. ISBN 978-1-4503-4233-9/16/05...$15.00

DOI: http://dx.doi.org/10.1145/2897845.2897874

contrast, we combine these two pieces of information to construct a more comprehensive system model in an adaptive and iterative way.

As such, our adaptive model-based security testing provides a higher code coverage guarantee, which enables the discovery of new vulnerabilities while reducing the number of possible false negatives. Furthermore, our method enables one to substantially reduce the manual efforts required by large-scale security testing of OAuth implementations because our approach can cater for the implementation differences of practical OAuth systems via observations extracted from network traces. In summary, this paper has made the following technical contributions:

- We propose an adaptive model-based testing framework by combining the protocol specification and the network trace observations, followed by iterative refinement of the model with implementation specifics.
- We design and implement OAuthTester to realize the proposed framework.
- We use OAuthTester to conduct security assessment for four major OAuth-based IdPs as well as 500 top-ranked web applications in US and China. To the best of our knowledge, this is the first work which systematically performs a model-based analysis of commercial implementations of OAuth at scale.
- Our empirical study discovers 3 previously unknown yet critical vulnerabilities while demonstrating new exploits for one old problem. In addition, our findings help to quantify the prevalence of different classes of vulnerabilities among mainstream OAuth-based applications and platforms.

The rest of the paper is organized as follows: In Section 2, we introduce the background of the OAuth 2.0 protocol. We describe the system architecture of OAuthTester in Section 3. We then discuss the modeling specifics of OAuthTester in Section 4. Additional implementation details are given in Section 5. Section 6 presents our empirical findings. Possible extensions of OAuthTester are covered in Section 7. We survey related work on OAuth security and testing tools in Section 8 and conclude our work in Section 9.

2. BACKGROUND ON OAUTH PROTOCOL

The OAuth framework consists of three entities: *IdP*, *User* and *Application (App)*[2]. In order to determine User's identity, the 3rd party App should request the authorization from User to access User's identifier (e.g., user profile) hosted by IdP. With this identifier, App can then log User in. To achieve this goal, RFC6749 [18] defines four types of authorization flows. Regardless of the specific authorization flow, OAuth ultimately gives an access token to App so that it can access User's profile hosted by IdP. In this section, we introduce two most popular authorization flow types, namely, the call flows for the Authorization Code Grant and the Implicit Grant.

2.1 Authorization Code Grant Flow

As illustrated in Fig. 1, the steps of authorization code flow are as follows: 1. User visits App and tries to log into App with IdP; 2. App redirects User to IdP for authentication with an optional STATE parameter to bind this request (Step 2) to the subsequent response at Step 5; 3. User authenticates with IdP and then authorizes the permissions requested by App. 4. IdP returns to User an authorization code with an optional STATE parameter (typically the value is the hash of cookies and a nonce); 5. User is redirected to the *redirection endpoint* where App should reject the request if the received STATE parameter does not match with that at Step 2. 6. App then requests the access token by sending the *code* and its

[2] In this paper, we only focus on the web applications.

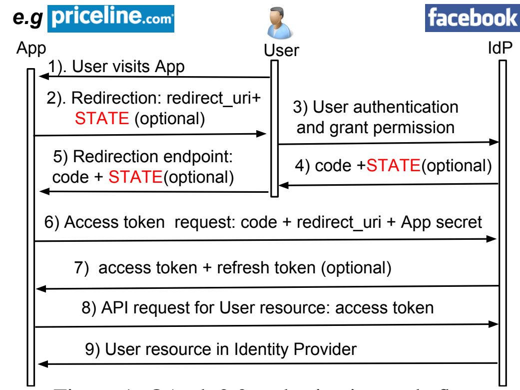

Figure 1: OAuth 2.0 authorization code flow

App secret to IdP; 7. After checking the validity of the code and the identity of App, IdP responds with an access token; 8. App requests user data via the access token; 9. IdP responds user data (e.g., profile) to App so that App can assume User's identity and log User in.

Note that Step 6 – Step 9 are executed on App server. From the user's perspective, he is immediately logged into App once he is redirected to App at Step 5.

2.2 Implicit Grant Flow

Unlike the authorization code flow, implicit grant flow directly relays the access token issued by IdP through User to App at Step 4 of Fig. 1. (TLS). This authorization flow is intended to lower the barrier of application development. It is useful in cases where App cannot protect the *app_secret* or the cryptographic primitives are too heavy to be implemented or executed by App.

2.3 Other Scenarios to Be Considered

The above two code flows only cover the basic scenarios. In reality, we also need to consider the following practical situations:

2.3.1 Identity Federation

After retrieving User's identity profile at Step 9 of Fig. 1, App has two choices to log User in: a) Log User in directly with this identity profile; b) Via *Identity Federation*: Under this approach, if User does not have an account for the App (local account), App creates one and binds this local account to the identity profile. Otherwise, if User is logged into the local account at Step 9, App may automatically bind these two identifiers together. In the latter case, when a user attempts to log into App with IdP later on, App can look for a mapping between IdP's identity profile and the identifier of a local account. If such a mapping exists, App then logs the user into the corresponding local account directly.

2.3.2 Revoke Authorization

IdPs often provide an *App Management Page* which contains all the previously authorized applications for User so that User can revoke the authorization just by clicking a delete button. After revocation, the original access token becomes invalid and thus App cannot access User's information any more.

2.3.3 Automatic Authorization

Once a user has authorized an application, IdP may directly redirect the user to the application in Step 5 of Fig. 1 upon receiving an authorization request. The user thereby does not need to verify and grant the same permissions requested by the application before. In this way, automatic authorization can improve the user experience and thus has been widely deployed.

3. SYSTEM OVERVIEW

In this section, we first discuss the threat model and then present the system architecture of OAuthTester.

3.1 Threat Model

We assume there are two users: a normal user Alice and an attacker Eve. Alice would follow normal instructions (e.g., Fig. 1) and send requests to App and IdP as defined by the OAuth protocol. Unlike Alice, Eve does not follow the protocol and would try to find logic flaws of IdP and App by sending arbitrary requests (e.g., out-of-order requests) at any time to any party in the OAuth system.

Since we mainly focus on the logic flaws of App and IdP, the normal user Alice in fact can be a different account controlled by the attacker. Therefore, the attacker can collect and analyze the network trace of Alice to build a more accurate system model. Note however that, Eve should *not* use Alice's network trace to construct the test case unless such network traffic is not protected, e.g., by TLS. In other words, Eve can eavesdrop unencrypted information of Alice to construct message exchanges with IdP/ App.

3.2 System Architecture of OAuthTester

Fig. 2 depicts the system architecture of OAuthTester, where solid lines represent the functionalities implemented by OAuthTester and dashed lines represent actions of other parties, i.e., IdP and App. The workflow of the system is as follows:

1. Based on the specification of OAuth [18, 26], we manually define a coarse-grained system model, which is then automatically initialized via identifying the key parameters from the network trace as stated in Section 4.1. This model abstracts and defines the normal behavior of IdP and App.
2. Given the system model, OAuthTester then automatically generates test cases in form of user inputs, e.g., HTTP requests.
3. After tampering the test cases as discussed in Section 4.2, the Test Harness sends executable test cases to IdP and App.
4. OAuthTester collects responses from IdP and App, e.g., HTTP responses and the state of the application/ IdP, etc.
5. The Test Oracle (Section 4.3) then compares the real system response with the expected behavior predefined in the system model and determines whether the response is normal or not.
6. For any abnormal behavior, OAuthTester determines whether it can lead to possible exploits. If so, list the test cases and then go to Step 8; otherwise, go to Step 7.
7. We need to go back to refine/ rectify the system model.
8. OAuthTester extracts useful knowledge (e.g., the values of the access token, the STATE parameter and their security properties) from the collected responses and then adds the knowledge to the Knowledge pool of the system model.
9. Based on the newly added knowledge, we can refine or customize the system model for a specific implementation as presented in Section 4.4. We continue this iterative process until OAuthTester covers every path in the state machine.

To generate test cases in Step 2, which cover all the execution paths of OAuth, we utilize model-based testing (MBT) techniques. Specifically, we build our tool on top of PyModel [22], an open-source project based on NModel [23], which takes a system model as input and output test cases. The PyModel tool supports on-the-fly testing so that we can continue to refine the system model throughout the testing process. Most importantly, PyModel can help to formally enumerate all the paths defined by the system model and thus provides some form of coverage guarantees for the tester.

To follow the aforementioned system architecture for the secu-

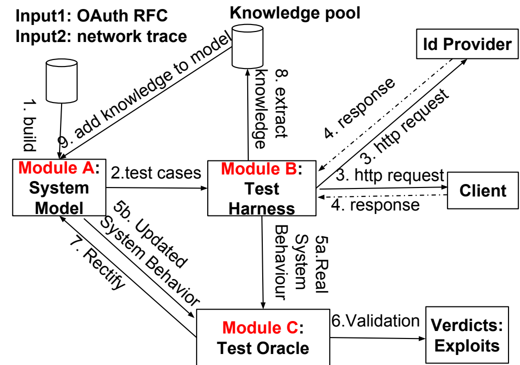

Figure 2: System Architecture of OAuthTester

rity testing of OAuth implementations, we design and realize three components (modules) in OAuthTester: the System Model, Test Harness and Test Oracle which correspond to Module A, B, C in Fig. 2, respectively. The System Model (Module A), as demonstrated in Section 4.1, is presented in form of a state machine which abstracts and describes the system behavior of OAuth. Based on the model, PyModel can generate test cases. The Test Harness (Module B) then tampers and executes the test cases as discuss in Section 4.2. Finally, the Test Oracle (Module C) compares the real system behavior with the expected one in Section 4.3 and determines whether the system is vulnerable or not. As PyModel only provides a framework to formally generate test cases to cover every path in the state machine, OAuthTester therefore needs to define the system model, implement the test harness and describe the normal system behavior for the test oracle.

4. DESIGN OF OAUTHTESTER

In this section, we first describe how to initialize the state machine with OAuth specification and the network trace, so that PyModel can automatically generate test cases. For each test case, we then present the testing scheme via fuzzing key security-related parameters and the execution of out-of-order requests. After executing the test case, we discuss the test oracle which can determine whether the system under test is normal or not. We finally discuss the refinement of the system model for each individual implementation.

4.1 Define the State Machine with the Specification

As shown in Fig. 3, we use a state machine to represent the OAuth call flow. Every state transition affects the relationship among the users (i.e. Alice and Eve), the application and the IdP. We therefore use the relationship among these four entities to describe a state in the system model as follows:

- *Alice_Status* : (whether Alice logs into App and IdP as Alice/ Eve);
- *Eve_Status* : (whether Eve logs into App and IdP as Eve/ Alice);
- *App_IdP* : (whether App is authorized by Alice/ Eve);

For sure, the above three variables are too coarse-grained to reflect the logic of OAuth. For example, State S_4 and S_5 would be depicted by these three variables in the same way. For this reason, more OAuth specific information should be considered to describe each state. Taking advantage of the protocol specification, we can know what security-related key parameters should be returned to the user for each request in advance. These RFC-defined key parameters therefore can be used to complement the representation of the state. Specifically, we have identified *redirect_uri*,

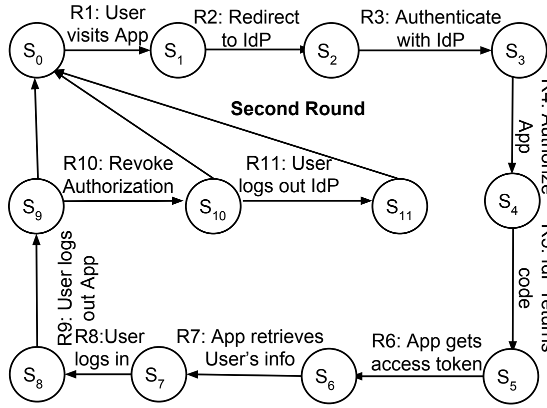

Figure 3: State Machine of Normal Operation of A Single User

- [1]: We do not consider State S_6, S_7 afterwards, as they occur in the server side of the application.
- 2: This figure is another representation of Fig. 1. We often exchange these two figures in the rest of this paper.

STATE, *scope*, *client_id*, *code*, *access_token*, *response_type* and *client_secret* as key parameters per RFC6749. Towards this end, we define another variable *knowledge_pool* to encode these key parameters obtained so far.

Therefore, State S_4 can be represented as follows:

- *Alice_Status=(Alice logs into IdP as Alice, Alice does not log into App)*;
- *Eve_Status = N.A*[3];
- *App_IdP = (App is authorized by Alice)*;
- *knowledge_pool = (client_id, scope, response_type, redirect_uri, STATE)*

But the *knowledge_pool* for S_5 is *(client_id, scope, response_type, redirect_uri, STATE,* ***code****)*.

Now that we have defined the states, the next step is to define the corresponding state transition(s) to complete the state machine. For each state transition, we define another variable *action* which represents the required HTTP request made by the user (Alice or Eve) so that the system can move to the next state. Fortunately, such requests are usually predictable. For example, Request $R3$, $R4$ and $R5$ in Fig. 3 always have similar structures as shown in Fig. 4. Therefore, we can use the normal representation of HTTP requests to represent *action* by the following attributes: the user who executes the action (Alice or Eve), the URL scheme (HTTP or HTTPS), the domain name, the request method (Get or Post) and key parameters contained in the above *knowledge_pool*.

4.1.1 Initialize the State Machine with the Network Trace

We can directly determine the URLs for those predictable requests (i.e., state transitions) in Fig. 4. However, other requests such as the login request $R2$ and the logout request $R9$ may vary widely depending on specific implementations. For such requests, we first sift through candidate requests by the key words contained in the request (e.g., login, logout, App name, etc.). We then send each candidate request to servers and compare the real response with the expected one. For example, if a candidate request of $R2$ leads to a login form, then we assume such a URL corresponds to a state transition defined in the system model. Since these URLs always contain the key parameters specified by the *knowledge_pool*, OAuthTester thereby can easily extract the values of these parameters.

[3]Fig. 3 does not include the behavior of the attacker.

R3:https://www.facebook.com/login.php?skip_api_login=1&api_key=127621437303857&signed_next=1&next=https://www.facebook.com/v2.0…&state=71e958b3edc02fb0368c81e4d71917a0#_=_&display=page

R4:https://www.facebook.com/v2.0/dialog/oauth?redirect_uri=http://imgur.com/signin/facebook&state=71e958b3edc02fb0368c81e4d71917a0&scope=email&client_id=127621437303857&ret=login&ext=1422535973&hash=AebXSOM95eUr5q5t

R5:http://imgur.com/signin/facebook?code=AQD7...7PMO&state=71e958b3edc02fb0368c81e4d71917a0#_=_

Figure 4: Part of traces for the running example

Note however that IdPs and applications often deploy OAuth in a different way from the specification and may invent new parameters for their own business consideration. For example, the parameters in the small (red) boxes of Fig. 4 are defined by individual IdP and application. Therefore, we should also consider these self-defined parameters to model the implementation features. Specifically, we only focus on those security-related key parameters among the overwhelming self-defined parameters.

To determine the key parameters generated by specific App and IdP, we carefully construct requests by randomizing the value of this parameter or removing the parameter. After sending the crafted request to servers, we monitor whether the response changed or not. If not, then such parameters should not be important and thus will not be encoded to the *knowledge_pool*. The rationale is that a key parameter should be valuable and its value or existence should affect the subsequent response. The fuzzing result turns out that api_key and $next$ are key parameters while others such as ret, ext and $hash$ of Request $R4$ are non-key parameters. As a result, the *knowledge_pool* of State S_4 can be represented as: *(client_id, scope, response_type, redirect_uri, STATE,* ***api_key, next****)*. These key parameters contained in the *knowledge_pool* can then be used to construct the *action*. For example, *action* $R4$ can be represented as: *[Alice, https, facebook.com/v2.0/dialog/oauth, GET, redirect_uri, state, scope, client_id*[4]*]*.

4.2 The Testing Process

With the above system model, PyModel can automatically generate test cases to cover every execution path of OAuth. The test case is represented by the state transition (*action*) and the expected state resulted from this *action*. For example, the test case of $R4$ in Fig. 4 is:

- *action R4=[Alice, https, facebook.com/v2.0/dialog/oauth, GET, redirect_uri, state, scope, client_id]*;
- State S_4 = *[Alice_Status, Eve_Status, App_IdP, knowledge_pool]*.

For each test case, OAuthTester would either fuzz one key parameter contained in the *action* or try to break the execution order.

4.2.1 Fuzzing Key Parameters

With the *action* (e.g., $R4$) generated by PyModel, Test Harness first determines a parameter (e.g., *STATE*) to be fuzzed. By executing this fuzzed action, the system would either go to an error page (i.e., remain in the original state such as S_3) or move to the expected state S_4. For the former, Test Harness keeps fuzzing the key parameters of action $R4$ until all key parameters in this request

[4]Other parameters such as *ret*, *ext* and *hash*, have been identified as non-key parameters by fuzzing techniques.

Table 1: Properties and its Correspondig Fuzzing Scheme of Key Parameters

Propety		Fuzzing Scheme
constant		Compare the values between different sessions and users
variable	mandatory parameter	Remove this parameter and randomize its values
	used for once or multiple times	Substitute the value with an used one and compare the response
	user-specific	Substitute the value with a fresh one of another user
	session-specific	Open a new browser and get a fresh value of this parameter to substitute the existing one

are fuzzed. For the latter, PyModel would continue to generate the next state transition (e.g., $R5$) to be fuzzed. The verification of the original action ($R4$) will be carried on only if this *action* is determined (by PyModel) to be the next state transition again. And we will keep track of the *action* which is not completely fuzzed so that PyModel can come back to this *action* later.

To fuzz parameters more efficiently, we first infer the security properties of key parameters, which also helps to better understand the logic of a real system (e.g., how parameters should be created, processed and deleted). Specifically, we consider properties listed in Table. 1. With the specification in hand, we can understand the security purpose of the key parameters defined by the specification. As scuh, we can manually initialize the properties for these parameters. For example, the property of *client_id* is ($constant$), and the property of *STATE* is (*variable, once, session-specific, user-specific*). For other parameters defined by specific IdP and App, we do not know their properties in the beginning, and would initialize them to (*variable, once, user-specific, session-specific*) so as to test all possible cases. The properties of all parameters can be learned and rectified during the iterative model-refining process in Section 4.4.

There are two different formats of parameters which corresponds to different fuzzing mechanisms. For parameters with special constructs like *redirect_uri*, *response_type* and *scope*, we leverage domain knowledge to tamper them. For example, we carefully craft *redirect_uri* using constructs outlined in Covert Redirect [24]. For other parameters such as the *STATE* parameter, the fuzzing scheme shown in Table. 1 is adopted. For example, to verify whether the *STATE* parameter is user specific, OAuthTester would substitute Eve's value with a fresh one of Alice.

Next we build an HTTP request by encoding the fuzzed *action*. This HTTP request is then sent to IdP or App via Firefox and Selenium [30] which can simulate the behavior of the user (e.g., button-clicking and link-navigation). We also use Firebug and NetExport, two plug-ins of Firefox, to automatically monitor and export responses from IdP/App. Values of key parameters are extracted from the responses so that we can construct subsequent fuzzing test cases. For example, to substitute a value with a parallel-session one, we do not need to start a new session. Instead, this value can be obtained from the previous response of a parallel session.

4.2.2 Break Execution Ordering

According to our threat model, we allow the attacker, regardless of her current state, to launch any state transition. This is to reflect the fact that Eve, the attacker, would always try to break the request sequence which is supposed to be enforced by the application and the IdP. The naive way would randomly select a request for the attacker to execute, which is definitely inefficient and ineffective. Towards this end, we only focus on the so-called *waypoints* in [34] which describes states that play an important role in the interactions of OAuth entities. As these waypoints always contain the logic condition that is required to move to the next state, they are supposed not to be bypassed in any path. For example, State S_3, S_4 in Fig. 5 should never be bypassed and are manually identified as the waypoints due to the following rationale/ logic: a user cannot authorize an application (S_4) without the user logging into the IdP (S_3). On the other hand, the App cannot get the authorization result (S_5) unless the user authorizes the request (S_4).

Therefore, a bypass of the waypoint implies a logic violation. To discover logic flaws, OAuthTester thereby manages to bypass the waypoints by exploiting the technique of PyModel. The basic idea is that, according to the current state of Eve, PyModel can generate the next state which follows the logic of OAuth. When the current state is a waypoint, then this current state is mandatory for the next state. In other words, if the attacker rollbacks the system to the previous state (i.e., move out from the waypoint), this originally enabled next state is supposed to be impossible. Specifically, the state transition from the previous state to the originally enabled next state is exactly a sequence of requests that would bypass the waypoint and thus break the logic.

In summary, the attacker constructs the illogical state transitions as follows: 1) Gets the enabled next state according to the current state; 2) When the current state is a waypoint, rollbacks the system to the previous state by exploiting the knowledge of the specification; 3) Tries to reach the originally enabled next state from the previous state.

Take Fig. 5 as an example. The App has already been authorized at State S_4, thereby IdP can issue a *code* to App as the authorization result. However, the attacker would stop at State S_4 and rollback the system to the previous state S_3 by revoking the authorization with Request $R10$. At State S_3, State S_5 is supposed to be impossible. However the attacker would make an *automatic authorization request*[5] and try to reach State S_5 without user's authorization. Surprisingly, two IdPs allow such an execution path (i.e., $S_3 \rightarrow S_4 \rightarrow S_3 \rightarrow S_5$) as demonstrated in Section 6.3. Another attempt shown in Fig. 5 by dashed lines is to authorize an App without logging into IdP.

4.3 Test Oracle

By comparing the real and expected system behavior (i.e., expected state), the test oracle can determine whether the system is normal or not. Since we have either mixed up the request sequences or tampered key parameters, the current state is supposed to be different from the expected one. To get the current state, OAuthTester first queries App and IdP for the relationship of the four entities of OAuth. Then it updates the *knowledge_pool*, especially the properties of the key parameters, by observing the response(s). Deviation of the current system state from the expected behavior as defined by the system model (i.e., cannot reach the expected state in the state machine) can imply loopholes or incorrect definition of the system model. To validate whether such non-conformance would lead to system insecurity, we check the following three security properties:

- *Authentication:* Eve can obtain information such as access token, code or session id, to convince the App/ IdP that she is the victim. Or she can log into the application as Alice.
- *Authorization:* Eve can bypass the authorization or obtain information to do anything as in the authorized session.

[5]This request is not shown in Fig. 3. It takes advantage of the *automatic authorization* feature which is possible only when the application is authorized.

- *Association:* The goal of OAuth is to correctly bind three pieces of data: the user's identity, the user's permissions and the session's identity. This association is what applications depend on to identify the user and his permissions.

Any violation of these properties demonstrates an exploit opportunity. In particular, OAuthTester would prompt a warning if the properties of key parameters defined by the specification have been updated. This is because these parameters are carefully designed against subtle attacks. Once the predefined properties are not satisfied, it may lead to subtle breaches. For example, if *redirect_uri* is updated as a *variable*, then the covert redirect attack [24] is possible. If no violation is discovered, OAuthTester executes the next test case until every path is covered and no more new knowledge can be learned.

4.4 Iteratively Refine the System Model

The system refinement consists of two major components: 1. update the properties of key parameters; 2. rectify the implementation of IdP. Firstly, OAuthTester can automatically update the properties for the key parameters by observing the response from OAuth entities. For example, if an old value of *STATE* parameter in Request $R5$ still leads the system to state S_8, then OAuthTester automatically updates the property to *multiple times* other than *once*. Given the updated property, OAuthTester can generate different state-transitions (*action*) on the fly. For example, if the *STATE* parameter can be used for multiple times, for the purpose of verifying its session-specific property, we do not need to get a fresh value of *STATE* parameter and just reuse the old one of a parallel session.

Another refinement is due to the different implementations of IdPs, which often add new features for their own business and service regardless of the definition of RFC6749. For example, Let's consider the automatic authorization feature which is not discussed in the OAuth standard specification. While testing OAuth implementation in action, we observe that Request $R2$ would immediately lead to State S_5 if the user has authorized the application. OAuthTester does not expect such a response and thus output an error notification. After inspecting the log information, we finally identify this feature and manually encode this feature into the model. For another example, Facebook recently applies a new parameter $signed_request$ to confirm the identity of a user. Because many applications adopt this parameter incorrectly, the properties of this parameter in different applications become different. As such, OAuthTester helps to identify this parameter is also a key parameter defined by IdP. With the document of IdP, we then can correctly initialize its security-related property. OAuthTester thereby can prompt warnings once its property is changed as if this parameter is also defined by the specification.

Although the above refinement requires manual intervention, such manual effort only needs to be done once per IdP under test and then be reused across all of its real-world implementations.

5. IMPLEMENTATION

We have implemented OAuthTester in Python with 5668 lines of code. It takes, on average, 20.3 minutes to complete the testing of an OAuth-based application, using a machine with a 1.4GHz dual core CPU and 4GB memory running Ubuntu 12.04. In this section, we first discuss the automation level of OAuthTester and then present various heuristics to speed up the testing process as well as to boost the detection accuracy.

5.1 Efficiency Consideration

Since every attempted button click or HTTP request involves a high-latency round-trip with IdP/ App, reducing the number of test cases is important to complete the testing within a reasonable time. We thus utilize the following heuristics to discover as many potential security flaws as possible within a limited time: (1) For those features which are shared among applications, we just test them once (e.g. for only one application). For example, once confirming the authorization *code* (in Step $R5$ of Fig. 3) is one-time used, it is not necessary to infer the property of this parameter for other applications of the same IdP. (2) We first check the applications for previously known loopholes to find all potential vulnerabilities for an application as soon as possible. (3) We mainly focus on the *waypoint* to construct out-of-order requests.

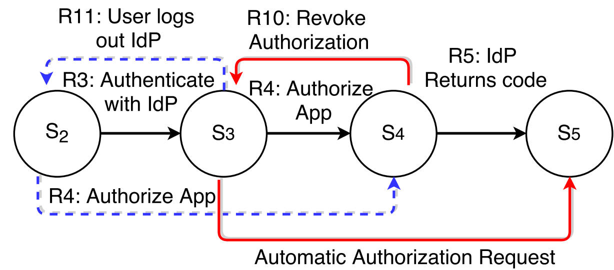

Figure 5: Design Request Sequence to Break Execution Ordering

5.2 Detection Accuracy

In general, OAuthTester should not report any false positives since we will check whether the system moves to the expected state after each state transition and re-validate the security properties for any abnormal behavior. We confirmed this hypothesis by randomly selecting 40 out of 405 applications for manual validation. And the manual checking does not find any false positives. On the other hand, false negative is possible because of the following three reasons. First, we can only guarantee to cover all paths defined by the state machine rather than the real implementation. If the model is too coarse grained, then we may miss some vulnerabilities in practice. Second, we only concern the SSO process without paying attention to the subsequent operations of this application under test after authentication. Finally, OAuthTester only looks for the original formats of key parameters or their values (e.g., access token=xxx) – it cannot find any leakage once the key parameter has been further obfuscated.

6. EMPIRICAL RESULTS

Using OAuthTester, we systematically examined 500 top-ranked websites in US and China[6], of which there are 405 websites implementing the OAuth services provided by 4 major IdPs including Facebook, Sina, Renren and Tencent Weibo. OAuthTester has enabled us to discover different forms of widespread misuses/ mishandling of the critical STATE parameter in OAuth and two classes of previously unknown logic flaws on the IdP side. We also provide new exploits for a well-known vulnerability, i.e., *failure to adopt TLS protection*. Besides discovering these previously unknown vulnerabilities, the automated OAuthTester also successfully detected numerous existing security flaws, some of which were identified manually by earlier research.

6.1 Discovered Vulnerability 1: Misuse of the STATE Parameter

6.1.1 Observed Results

According to RFC6749 [18], the STATE parameter should be generated and handled as a nonce. In addition, it should be bound to

[6]Ranked by Alexa: http://www.alexa.com/topsites.

Table 2: Statistics of STATE Parameter Misuse[1 2]

IdPs (NO. of Applications)	Lack STATE Validation	Lenient STATE Validation	STATE Not Bind to User	STATE Replay			Summary
				Multiple Use Until Next Login	Same STATE for One Browser	Constant STATE Parameter	
Facebook (79)	12.50%	14.29%	21.42%	5.37%	14.29%	7.14%	35.72%
Sina (182)	13.32%	13.32%	59.97%	38.66%	10.68%	32.01%	74.67%
Renren (68)	18.17%	9.09%	30.31%	18.19%	12.12%	24.23%	65.13%
Tencent Weibo[3] (36)	12.50%	12.50%	12.50%	37.50%	12.50%	0	50.00%
Average (405)	13.09%	11.85%	37.53%	18.52%	11.36%	20.00%	55.31%

- [1] We divide the number of App with specific problem by the number of tested applications which use the STATE parameter.
- [2] There may be overlaps. For example, applications lacking of validation may also be susceptible to lenient STATE validation.
- [3] Only 8 applications use the STATE parameter, thus the distribution is skewed.

a session to defend against CSRF attacks reported in [5,20,37]. Unfortunately, we have found that 61.23% of applications under study do not use the STATE parameter. Even worse, for those applications which support the STATE parameter, 55.31% of them are still vulnerable to CSRF attacks due to the misuse/ mishandling of the STATE parameter. In particular, OAuthTester has discovered the following misuse cases via fuzzing as discussed in Section 4.2.1:

- *Lack of STATE Validation*: STATE parameter is not validated by App.
- *Lenient STATE Validation*: If there is a STATE parameter, applications can verify it correctly. However, App also accepts requests with missing STATE parameter.
- *STATE Does Not Bind to User*: The App assumes all the STATE parameters generated by itself are valid and fails to check whether this parameter is bound to the user's session. As a result, the attacker can substitute the victim's STATE parameter with her own in a forged request.
- *STATE Replay*: The STATE parameter can be reused for several times. According to the validity of the STATE parameter, there are three cases. The first two cases require an attacker to possess the victim's STATE parameter somehow. As shown in Fig. 6, 43.95% of the tested applications are vulnerable to the STATE Replay attack and they do not use TLS to protect the STATE parameter at the same time. Therefore, the attacker can obtain the STATE parameter by eavesdropping.
  - *Multiple Use Until Next Login*: The STATE parameter remains valid until the victim tries to login the application again. Only at this point would a new STATE value be generated.
  - *Same STATE for One Browser*: The same value is used for the same user as long as the user uses the same browser.
  - *Constant STATE parameter*: The parameter remains unchanged across different sessions and users.

6.1.2 Security Implications

Due to the misuse of the STATE parameter, an attacker can then launch a CSRF attack[7], which can lead to at least two problems. The first one is the so-called "Login CSRF" [6], which allows an attacker to log the victim into an application as the attacker. Depending on the services provided by the application, the consequence includes an attacker being able to sniff at the victim's activities on the application. Although this attack may seem mild, its impact can be amplified by our newly discovered Dual-Role IdP attacks which will be the subject of Section 6.2.

Besides Login CSRF attack, it can also result in account hijacking, i.e., an attacker can log into the application as the victim as shown in [20]. Different from previous works [5, 18, 20, 36, 37] which believed that these attacks are only possible when the STATE parameter is missing, our OAuthTester showed that even if the STATE parameter is used, it could be misused in many different forms which could lead to various attacks.

[7]The attack procedure can be found in the appendix.

6.1.3 Plausible Root Causes

Surprised by the large percent of applications which cannot handle the STATE parameter correctly, we set out to analyze the underlying reason by examining the OAuth software development kits (SDKs) and programming guides provided by the four IdPs under study as well as Google. We found that all of the official SDKs provided by the IdPs do not include routines for application developers to manage the STATE parameter. Instead, most of the IdPs merely state that the STATE parameter should be a nonce without giving a code snippet or providing a routine for checking whether the STATE parameter is implemented correctly. In other words, application developers must implement the STATE parameter by themselves, which turns out to be more complicated than we expect. In fact, even some IdPs implement or manage this parameter incorrectly. For example, the STATE parameter "should be" a confidential knowledge in case of the STATE Replay attack. However, Renren, one of the major Chinese IdPs, fails to follow this requirement and simply redirects the STATE parameter to a non-TLS endpoint. In another example, Sina is a dual-role IdP (see Section 6.2) and employ Facebook to provide the OAuth service for users outside China. However, its implementation for Facebook does not validate the STATE parameter at all.

6.1.4 Recommended Mitigations

As a remedy, we propose to generate and verify the STATE parameter in the SDK so that a typical application developer can also implement this parameter correctly without much effort. As a proof of concept, we have extended Google's OAuth Python SDK with merely 12 lines of code to add two functions, namely, the generation and verification of the STATE parameter. To process the STATE parameter in the SDK, application developers are required to explicitly invoke these two functions, respectively. Although integrating the STATE parameter into the SDK seems straightforward, this solution has the following limitations, which may explain why IdPs do not adopt the scheme:

- The process of the STATE parameter is tightly related to session management, for which the application developers have multiple options. It is difficult for the SDK, which is supposed to define the core functionality only, to consider the different operations among numerous session management tools. As a result, the SDK-based solution may not be applicable to every programmer.
- While the SDK can help to generate and verify the STATE parameter, 61.23% Apps under study do not send this parameter to IdP, which renders the server-side SDK code useless.

To overcome the former limitation, we suggest the SDK designer to support the popular session management tools (e.g, KVSession of Python Flask) and provide a template for the programmers to specify their choice of session management tool. With such support, the SDK can manage the STATE parameter stored by the session tool accordingly. To address the latter limitation, IdP should make the STATE parameter mandatory by checking its presence in specific requests (Step 3 in Fig. 1) and warn the programmer if it is missing.

We also observe Google goes one step further by providing a toy application, which shows the correct usage of the STATE parameter. However, the sample codes for managing the STATE parameter are scattered across different parts of the application. Without highlights, application developers may ignore or miss part of the codes. Furthermore, the toy application fails to follow the best practice of deleting the one-time STATE parameter after verification. While the objective of this practice is to allow a logout-user to re-login the application without reloading the page, such implementation in turn makes the toy application vulnerable to the STATE Replay attack, i.e., *Multiple Use Until Next Login* in Table 2.

In short, it is not straightforward to correctly reuse or adapt the sample code of a toy application to a real-world one. Worse still, some applications may implement OAuth in their own ways without using the SDK. Therefore, we propose a complementary solution beyond the modification of SDK. Specifically, we provide a step-by-step guideline for the programmers to add the following code snippet to the right place of their application. While we only use Python and a lightweight Web framework Flask as the example, the steps can be applicable for other web languages such as PHP.

1. Generate a random STATE value at Step 2 of Fig. 1:

```
state = ''.join(random.choice(
string.ascii_uppercase + string.digits)
for x in xrange(32))
```

2. Bind the STATE parameter to the session once the STATE parameter is generated:

```
session['state'] = state
```

3. Compare the STATE parameter when receiving the response at Step 5 of Fig. 1:

```
if request.args.get('state', '') !=
session['state']:
    return error
```

4. After comparison, delete this one-time STATE parameter:

```
del session['state']
```

6.2 Discovered Vulnerability 2: Amplification Attack via Dual-Role IdPs

6.2.1 Observed Results

A *Dual-Role IdP* is an IdP which offers OAuth-based authentication for its applications while using the OAuth services provided by other *leading* IdPs. For example, while AOL is an application of Facebook, it also offers OAuth-based authentication service to other applications on its own (AOL) platform. Unfortunately, we have discovered that, if a dual-role IdP could not implement OAuth correctly (as an application) for its leading IdP(s), all applications on the dual-role IdP's platform, no matter whether their implementations are secure or not, would be affected. An attacker can thus significantly amplify the impact of his attacks by targeting OAuth implementation flaws in a dual-role IdP platform instead of attacking an individual 3rd party application. In particular, there is a strong motivation for any IdP to become a dual-role IdP to facilitate new users to login, since many users may only have an account in a subset of the leading IdPs. As such, the dual-role IdP is quite common in practice. For example, 10 out of 13 Chinese IdPs under study are dual-role ones.

On the other hand, a dual-role IdP often supports multiple leading IdPs as its providers. For example, Sohu, the 8th most popular Chinese websites ranked by Alexa, provides 9 leading IdPs for its users to login. Another observation is that a dual-role IdP may implement OAuth securely for some leading IdPs while fail to do so for others. Combining these two findings, it is very likely for an attacker to find a vulnerable OAuth implementation of a dual-role IdP for one of its leading IdPs. For example, 8 out of the 10 Chinese dual-role IdPs are vulnerable due to their incorrect OAuth implementation for other leading IdPs.

6.2.2 Security Implications and Remedies

Such an amplification attack is quite general in nature and can be applied in many scenarios. We demonstrate its power by the following two examples. Firstly, Jing [24] shows that an attacker can obtain the victim's *code* by the covert (open) redirect attack. Equipped with this *code*, Eve can log into the application as the victim by substituting the *code* with the victim's one in Step 5 of Fig. 1. To hijack a large number of application accounts, Eve would normally need to find an open redirect URI for every application which is time consuming and may not be even feasible as it is unlikely for small/ simple applications to have such a URI. However, if Eve can find the URI for a flawed dual-role IdP, she can then obtain the *code* to log into the dual-role IdP as the victim. Since the victim may use the dual-role IdP to authenticate many applications in advance, the attacker can then easily log into these applications as the victim via a dual-role IdP.

More interestingly, the dual-role IdP can aggravate some seemingly mild problems. For example, the 'Login CSRF' attack logs the victim into the application as the attacker. If the application becomes the dual-role IdP, then Alice may bind the attacker's account on the dual-role IdP to her own application account. For this binding step, the attacker can even exploit the feature of automatic authorization to authorize many applications in advance to avoid any interactions with the victim. Thereafter, the attacker can login these applications as the victim without being detected.

Apart from these examples, this amplification attack can lead to more possibilities depending on specific use cases. To prevent the aforementioned amplification attack, a dual-role IdP must provide a secure SSO service for applications on its own platform and implement OAuth correctly for *all* of leading IdPs' OAuth service it supports.

6.3 Discovered Vulnerability 3: Failure to Revoke Authorization

6.3.1 Observed Results

As discussed in Section 4.2.2, we rollback the system to the previous state (e.g., State S_3) and then try to visit the next state (e.g., State S_5). By this way, we discover a deviation from the normal execution sequence ($S_3 \rightarrow S_4 \rightarrow S_5 \rightarrow \ldots$) to an abnormal behavior ($S_3 \rightarrow S_5 \rightarrow \ldots$), which leads to an authorization violation since the attacker can bypass the authorization step. In other words, IdP would automatically issue a *code* (State S_5) for a unauthorized application when receiving the authorization request ($R4$ in Fig. 3), as long as this application has been authorized once (but now has

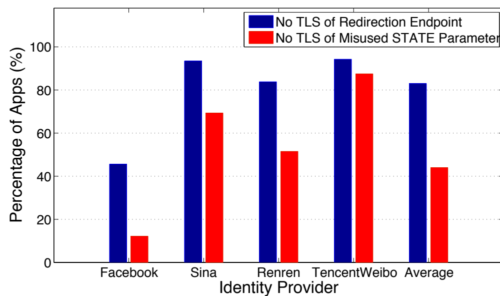

Figure 6: No TLS of Misused STATE Parameter

been revoked). Specifically, we have identified such a logic flaw for specific applications in cooperation with Sina or Renren.

6.3.2 Security Implications and Remedies

This logic flaw can lead to unauthorized access to the user resource hosted by IdP. After revoking authorization, the original access token has become invalid and the App cannot use it to access user's data anymore. However, the App may exploit this logic flaw to obtain a fresh token (or activate the original one) as follows: 1). When the victim visits the application, the application sends an authorization URI ($R4$ in Fig. 3) to IdP via the victim's browser. 2). As Sina and Renren fail to completely validate the status of the application, they mistakenly assume that the application has been authorized (not revoked). Therefore, Sina and Renren exploit the *automatic authorization* feature to issue a fresh access token (and activate the invalid one) to the application without the user's confirmation and awareness. By this way, an application can obtain an access token to stealthily access the user's information again.

As no interactions are required, the user cannot realize such an attack unless he checks the *App Management Page*[8] carefully. Worse still, suppose the user can realize that he is under attack, no mechanism can be employed to protect himself. We find that even when the user changes his password in IdP, the issued access token is still valid and the attack steps have not been interfered. In this sense, the ultimate goal of OAuth has been totally broken since the application can access the user resource without authorization.

The failure in revoking previous authorization has been confirmed by Sina who explained that this problem was a side-effect of special cooperation between Sina and some privileged applications on the platform. Although we only identified two vulnerable applications for Renren and one for Sina, we expect more applications with this loophole exist given the prevalence of privileged applications [21] on those platforms. To prevent this logic flaw, IdP should make complete validations for all applications, no matter whether they are in special cooperations or not.

6.4 New Discoveries on the Failure to Adopt TLS Protection

6.4.1 Observed Results

The security of OAuth critically relies on the usage of TLS. As shown in Fig. 6, however, 82.96% of the applications under study do not deploy the TLS scheme. It is well-known that the lack of TLS allows an on-path attacker to eavesdrop the confidential *code* or *access token*. With such information, the attacker then can either login the application as the victim or retrieve the victim's data [37] hosted in the IdP. Although this is an old problem, we discuss below new exploits of this vulnerability.

[8]This page would show all the authorized applications excluding the applications that have been revoked authorization.

6.4.2 Security Implications

While it is usually difficult for an "ordinary" attacker to sit on the path and eavesdrop the user's communication, if the communication is forced to flow through a on-path device controlled by a monitoring authority as in the cases of the Great Firewall (GFW) or Great Cannon (GC) [28] of China (GFW), then the monitoring authority can readily access to the IdP or gain control of App which is supposed to be outside its jurisdiction. Note that this new exploit only requires the following two conditions and does not need any direct involvement from the App or IdP.

1. The network traffic to/from either the IdP or the App should pass the monitoring device of the authority.
2. The traffic passed through the authority should not be protected by TLS.

For the former, it is difficult for a normal user to know in advance whether an IdP or App, especially for mobile applications/ services, is hosted behind the monitoring device of the authority. For the latter, our study shows that 91.24% OAuth-based applications inside China do not adopt TLS to protect their OAuth sessions. Depending on whether the App or the IdP is hosted inside the on-path monitoring authority, there can be two different scenarios.

1. Gain Access to IdP outside the Jurisdiction of the Monitoring Authority. In this case, the App is hosted behind the on-path monitoring device of the authority, who can eavesdrop the *access token* to access to user's data hosted by IdP. Worse still, with on-path devices like the recently reported Great Cannon of China, one can readily tamper the OAuth user request and launch large-scale App impersonation attacks [21] to acquire the *access tokens* of the victims in a whole-sale manner.
2. Gain Control of App outside the Jurisdiction of the Monitoring Authority. If the IdP is behind the on-path monitoring device of the authority, the authority can directly eavesdrop the identity credentials (e.g., *code* or *access token*) issued by the IdP at Step 4 of Fig. 1. We indeed observe some major IdPs do not adopt TLS in this step. As a result, the monitoring authority can use the credential to log into an application as the user and gain the full control of the user's offshore application account such as checking the user's activities on the application.

6.5 Rediscover Existing Vulnerabilities

In addition to the discovery of the new vulnerabilities and exploits discussed above, OAuthTester has also successfully rediscovered existing OAuth security flaws during our automated vulnerability assessment. Such existing loopholes include those caused by the 4 vulnerabilities discovered by SSOScan [42], 5 discovered by AuthScan [4], 4 presented in [37], the App Impersonation attack [21] and covert redirect [24]. Since other vulnerabilities have been well investigated, we only demonstrate two more recent attacks, namely, the App impersonation and covert redirect attack, to show the power of OAuthTester. OAuthTester can recoganize the system is in a wrong state when fuzzing the *redirect_uri* and *response_type* parameters. In addition, the properties of these two parameters are changed.

After the debut of these two vulnerabilities, Facebook and Sina have provided corresponding fixes, as shown in Fig. 7, by allowing applications to restrict the *redirect_uri* and the *response_type*. Despite these efforts, there are still 62.69% and 89.60% of tested applications vulnerable to these two attacks, respectively.

7. DISCUSSIONS

Owing to the relatively small set of possible state transitions and request/ response parameters, our proposed model-building/ testing approach turns out to be effective to discover vulnerabilities of

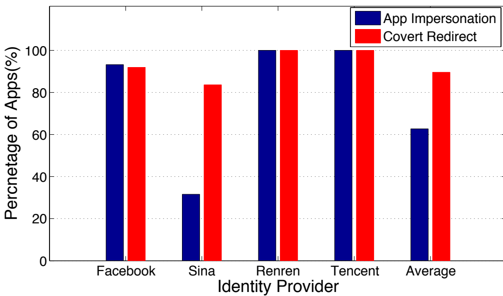

Figure 7: Statistics of App Impersonation and Covert Redirect

OAuth. However, we can further refine OAuthTester, whose power depends on the granularity of system model. Although we have managed to refine the model by the $knowledge_pool$ variable according to the characteristics/ details of the system response(s), we may still fail to infer deeper insights of the system. For example, we consider neither the implementation of SSL nor the detailed HTTP header such as cookies. We also do not consider "non-key" parameters that might have an implicit effect on the system security. Hence, it is worthwhile to consider possible ways to further refine our current system/ protocol modeling methodology. To indicate our future research directions, we discuss below some potential security flaws which are not well studied by the current model:

Inconsistency of Subsystem: App may consist of a main system and multiple subsystems. The main system generally is more important and only supports a more secure IdP. Meanwhile, a user will automatically log into the main system once he logs into the subsystem. Therefore, an attacker may target the subsystem to break the main system if the subsystem does not incorrectly implement OAuth. In the context of Web, this feature may lead to various attack vectors. For example, the attacker can bypass the same origin policy since these two systems often share the same domain [40].

Failure to Re-authenticate App: As required by RFC6749, IdP should authenticate App by *app_secret* during refreshing access token, but some IdPs, e.g. the Tencent Weibo, do not authenticate their applications at all. Hence, an attacker can get an access token once he obtains a refresh token somehow. As the refresh token is sent in a URL fragment, it can be easily disclosed with any 302 redirect to malicious domain [19].

Expected Permission: When a user authorizes an application, IdP will show a list of permissions requested by the application so that the user can reset and take back the privileged and unnecessary permissions. However, the user never knows whether the IdP actually issues/ revokes the permissions as expected.

8. RELATED WORK

OAuth Security Studies. Previous research [18] and [26] mainly focused on the study of the security model of OAuth from the protocol design and specification perspectives. For example, the authorization code grant flow has been proven to be cryptographically secure [8] under the assumption that transport layer security (TLS) is used. In particular, model-checking method has been extensively used. Pai [33] formalized the specification of OAuth and rediscovered a known security flaw. Bansal et al. [5] applied ProVerif to reveal two unknown vulnerabilities, i.e., Covert Redirect and Social CSRF attack, based on a customized attacker model and finer-grained web security mechanisms. By blackbox fuzzing and whitebox program analysis, AuthScan [4] extracted the protocol from real implementations and evaluated the protocol via off-the-shelf model checking tools. Inspired by these works, Fett proposed an expressive infrastructure model to facilitate a comprehensive checking in [16]. For more general studies, researchers [3, 29, 38] have employed similar schemes to analyze the security of other Single-Sign-On protocols including OpenID, SAML and Facebook Connect. While these studies mainly focused on the specification analyses, our goal is to discover security problems caused by implementation flaws in practice.

Despite the power of model checking, many practical loopholes remain undiscovered due to different interpretations of the ambiguous specification as shown in [10]. To clarify the specification as well as implementation subtleties, Wang et al. [40] utilized the program verification techniques to analyze the recommended software development kits (SDK) provided by IdPs and showed that the SDKs often contain hidden, implicit assumptions. Wang et al. [39] recovered important semantic information from the network traffic and identified 8 unknown logical flaws. Based on these works, InteGuard [41] further managed to protect the application of OAuth by analyzing the relationship of parameters. A CST [9] approach is then proposed to protect a more general multiparty protocol transactions. While these works mainly focus on specific vulnerabilities, our work enables one to systematically evaluate the security of real-world OAuth implementations at scale.

Security Testing. Gibbons [17] manually tests the Top-50 websites in Ireland against the security model defined by RFC6819. For large scale testing, Sun [37] then builds a semi-automatic tool to test five specific vulnerabilities for 96 applications, which are discovered by manual. While SSOScan [42] can improve [37] in terms of the testing scale and automation level, SSOScan also only aims at detecting five previously known vulnerabilities. Shernan et. al [36] crawl the Alexa top 10,000 domains and identify that 25% of websites using OAuth do not use the STATE parameter. Note that these tools available so far are specially designed for a limited set of *specific, previously-known* loopholes based on the manual understanding of the system. In contrast, OAuthTester is able to discover new vulnerabilities thanks to a formal system model and higher code coverage.

Common testing tools such as WebScarab [32], HTTP Fuzzer [1] and NoTamper [22], only accept per-request fuzzing and thus fail to analyze dependencies between different tuples of requests with responses. Blackbox testing like [13, 34] fail to consider protocol-specific information and cannot guarantee the coverage. Unlike these works, model-based testing (MBT) can guarantee to cover all the execution paths. It is shown in [35] that MBT is more efficient in finding defects than manual testing and thus has been widely applied in software testing [12], for embedded systems, client-server systems and web applications. Most of MBT works such as [14, 15, 27] assume the existence of a correct model, which is not true for OAuth due to the different interpretation of the RFC.

Other Model-Based Inference/Verification Tools. Table 3 shows the key differences between OAuthTester and some general model-based tools. To construct a state machine, various inputs, e.g., executables, network trace and specifications, have been extensively analyzed. For example, Prospex [11] applies the dynamic data taint techniques to analyze the binary executables, which, however, are often not available for modern web applications. In addition, pure network-trace-based scheme like ScriptGen [25], [34] and [39] only analyzes the network traces (i.e., Input2 in Fig. 2) to infer the data dependency of HTTP request/ response parameters. Such method lacks the knowledge of the expected behavior of the protocol (as specified in its standards) which makes it difficult to identify security problems caused by the incorrect implementation of the specification. On the other hand, pure specification-based scheme such as Bansal [5] builds the model only based on the protocol specification (i.e., Input1 in Fig. 2), which cannot discover the vulnerabilities caused by implementation specifics.

Table 3: Comparison between OAuthTester and Other Existing Model-based Inference/Verification Tools

Selected Tools	Required Input			Methodology			Output		Purpose
	specification	network trace	executable or source code	passive analyses	active testing	iterative refinement	data dependency	state machine	
Prospex [11]			Yes	✓			✓		spec extraction
ScriptGen [25]		Yes		✓			✓		honeypot script
Doupé [13]		Yes		✓				✓	fuzzing
Pellegrino [34]		Yes		✓	✓		✓		logic flaws
Bansal [5]	Yes			✓				✓	model checking
AuthScan [4]		Yes	Partial	✓	✓	✓	✓	✓	model checking
SMACK [7]	Yes		Yes		✓			✓	TLS testing
Antunes [2]	Yes	Yes		✓				✓	spec complement
Our work	Yes	Yes		✓	✓	✓	✓	✓	OAuth testing

Therefore, hybrid-input methods have been proposed. AuthScan [4] manages to infer the system model from the client-side JavaScript code and the network trace. Due to the limited information in the JavaScript code, the built model is not expressive enough and thus can only discover general Web attacks (e.g., replay attack, token leakage, etc) without any specification-dependent vulnerabilities. Meanwhile, SMACK [7] relies on the source code of SSL library to analyze the deviation of the real behavior from the standard model built from the TLS specification. However, the source code is often not available for OAuth applications.

As a result, the combination of specification and the network trace has attracted increasing attention. For instance, Antunes [2] builds an initial model based on the specification and then complement it with the network trace. However, this model is not iteratively refined by active testing and its major concern is to discover the commands or message formats that are not published/ documented. Unlike these previous works, OAuthTester combines the specification and the network trace information to iteratively/ adaptively build a more comprehensive system model for security testing.

9. CONCLUSION

We proposed an adaptive model-based testing tool, OAuthTester, to systematically evaluate the implementations of OAuth. With OAuthTester, we have examined the implementations of 4 major IdPs as well as 500 top-ranked US and Chinese websites which use the Single-Sign-On service provided by these four IdPs. Apart from existing loopholes, we have discovered 3 previously unknown vulnerabilities, all of which can result in serious consequences including large-scale resource theft and application account hijacking. Our findings demonstrate the efficacy of OAuthTester in performing systematic, large-scale security analyses for real-world OAuth implementations and deployments. Our testing results also provide a reality-check on the quality of the current OAuth implementations across the industry. These findings, hopefully, would facilitate the design and deployment of more secure OAuth systems in the future.

10. ACKNOWLEDGMENTS

We informed all the impacted identity providers of their corresponding vulnerabilities and provided them a compiled list of vulnerable applications on their platforms. Facebook and Sina had subsequently confirmed our findings and helped to notify the affected applications. For the other IdPs which did not respond to our disclosure, we had re-informed them with an updated list of vulnerable applications.

We thank anonymous reviewers for their insightful comments. This work is supported in part by a CUHK-Hong Kong RGC Direct Grant (Project No. 4055031), a CUHK Technology and Business development Fund (Project No. TBF15ENG003), NSFC (Grant No. 61572415), and the General Research Funds (Project No. CUHK 4055047 and 24207815) established under the University Grant Committee of the Hong Kong SAR, China.

11. REFERENCES

[1] R. Abela. *HTTP Fuzzer*. acunitex.

[2] J. Antunes and N. Neves. Automatically complementing protocol specifications from network traces. In *Proceedings of the 13th European Workshop on Dependable Computing*. ACM, 2011.

[3] A. Armando, R. Carbone, L. Compagna, J. Cuellar, and L. Tobarra. Formal analysis of SAML 2.0 web browser single sign-on: breaking the SAML-based single sign-on for Google apps. In *Proceedings of ACM workshop on Formal methods in security engineering*, 2008.

[4] G. Bai, J. Lei, G. Meng, S. S. Venkatraman, P. Saxena, J. Sun, Y. Liu, and J. S. Dong. AuthScan: Automatic extraction of web authentication protocols from implementations. In *NDSS*, 2013.

[5] C. Bansal, K. Bhargavan, and S. Maffeis. Discovering concrete attacks on website authorization by formal analysis. In *CSF*, 2012.

[6] A. Barth, C. Jackson, and J. C. Mitchell. Robust defenses for cross-site request forgery. In *CCS*. ACM, 2008.

[7] B. Beurdouche, K. Bhargavan, A. Delignat-Lavaud, C. Fournet, M. Kohlweiss, A. Pironti, P.-Y. Strub, and J. K. Zinzindohoue. A messy state of the union: Taming the composite state machines of TLS. In *S&P*, 2015.

[8] S. Chari, C. S. Jutla, and A. Roy. Universally composable security analysis of OAuth v2.0. *IACR Cryptology ePrint Archive*, 2011.

[9] E. Y. Chen, S. Chen, S. Qadeer, and R. Wang. Securing multiparty online services via certification of symbolic transactions. 2015.

[10] E. Y. Chen, Y. Pei, S. Chen, Y. Tian, R. Kotcher, and P. Tague. OAuth demystified for mobile application developers. In *CCS*. ACM, 2014.

[11] P. M. Comparetti, G. Wondracek, C. Kruegel, and E. Kirda. Prospex: Protocol specification extraction. In *S&P*. IEEE, 2009.

[12] A. C. Dias Neto, R. Subramanyan, M. Vieira, and G. H. Travassos. A survey on model-based testing approaches: a systematic review. In *Proceedings of ACM international workshop on Empirical assessment of software engineering languages and technologies*, 2007.

[13] A. Doupé, L. Cavedon, C. Kruegel, and G. Vigna. Enemy of the state: A state-aware black-box web vulnerability scanner. In *USENIX Security*, 2012.

[14] J. Ernits, R. Roo, J. Jacky, and M. Veanes. Model-based testing of web applications using NModel. Springer, 2009.

[15] J. Ernits, M. Veanes, and J. Helander. Model-based testing of robots with NModel. *Proc. Microsoft Research*, 2008.
[16] D. Fett, R. Kusters, and G. Schmitz. An expressive model for the web infrastructure: Definition and application to the Browser ID SSO system. In *S&P*. IEEE, 2014.
[17] K. Gibbons, J. O. Raw, and K. Curran. Security evaluation of the OAuth 2.0 framework. *Information Management and Computer Security*, 22(3), 2014.
[18] D. Hardt. RFC6749: The OAuth 2.0 authorization framework. 2012.
[19] E. Homakov. *The Achilles Heel of OAuth or Why Facebook Adds Special Fragment*.
[20] E. Homakov. The most common OAuth2 vulnerability. http://homakov.blogspot.hk/2012/07/saferweb-most-common-oauth2.html.
[21] P. Hu, R. Yang, Y. Li, and W. C. Lau. Application impersonation: problems of OAuth and API design in online social networks. In *Proceedings of the ACM conference on Online social networks*, 2014.
[22] J. Jacky. Pymodel: Model-based testing in Python. In *Proceedings of the Python for Scientific Computing Conference*, 2011.
[23] J. Jacky, M. Veanes, C. Campbell, and W. Schulte. *Model-based software testing and analysis with C#*. Cambridge University Press, 2007.
[24] W. Jing. Covert redirect attack. http://tetraph.com/covert_redirect.
[25] C. Leita, K. Mermoud, and M. Dacier. ScriptGen: an automated script generation tool for honeyd. In *Computer Security Applications Conference, 21st Annual*. IEEE, 2005.
[26] T. Lodderstedt, M. McGloin, and P. Hunt. RFC6819: OAuth 2.0 threat model and security considerations. 2013.
[27] G. Maatoug, F. Dadeau, and M. Rusinowitch. Model-based vulnerability testing of payment protocol implementations. In *HotSpot'14-2nd Workshop on Hot Issues in Security Principles and Trust*, 2014.
[28] B. Marczak, N. Weaver, J. Dalek, R. Ensafi, D. Fifield, S. McKune, A. Rey, J. Scott-Railton, R. Deibert, and V. Paxson. China's great cannon. *Citizen Lab*, 2015.
[29] M. Miculan and C. Urban. Formal analysis of Facebook Connect Single Sign-On authentication protocol. In *SOFSEM*, 2011.
[30] B. Muthukadan. *Selinum with Python*.
[31] OAuth.io. *CasperJs Automated Testing for The OAuth Flow*.
[32] OWASP. *Fuzzing with WebScarab*.
[33] 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) IEEE, 2011*, 2011.
[34] G. Pellegrino and D. Balzarotti. Toward black-box detection of logic flaws in web applications. In *NDSS*, 2014.
[35] C. Schulze, D. Ganesan, M. Lindvall, R. Cleaveland, and D. Goldman. Assessing model-based testing: an empirical study conducted in industry. In *Companion Proceedings of the International Conference on Software Engineering*. ACM, 2014.
[36] E. Shernan, H. Carter, D. Tian, P. Traynor, and K. Butler. More guidelines than rules: CSRF vulnerabilities from noncompliant OAuth 2.0 implementations. In *Detection of Intrusions and Malware, and Vulnerability Assessment*. 2015.
[37] S.-T. Sun and K. Beznosov. The devil is in the (implementation) details: an empirical analysis of OAuth SSO systems. In *CCS*, 2012.
[38] S.-T. Sun, K. Hawkey, and K. Beznosov. Systematically breaking and fixing OpenID security: Formal analysis, semi-automated empirical evaluation, and practical countermeasures. *Computers & Security*, 2012.
[39] R. Wang, S. Chen, and X. Wang. 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.
[40] R. Wang, Y. Zhou, S. Chen, S. Qadeer, D. Evans, and Y. Gurevich. Explicating SDKs: Uncovering assumptions underlying secure authentication and authorization. In *USENIX Security*, 2013.
[41] L. Xing, Y. Chen, X. Wang, and S. Chen. Integuard: Toward automatic protection of third-party web service integrations. In *NDSS*, 2013.
[42] Y. Zhou and D. Evans. SSOScan: Automated testing of web applications for Single Sign-On vulnerabilities. *USENIX Security*, 2014.

APPENDIX

A. ATTACK PROCEDURE FOR MISUSED STATE PARAMETER

An attacker can behave as a normal user to log into App with IdP in Step 1-3; Afterwards, the attacker can 4) intercept the URI which contains the STATE parameter and the authorization code (or access token); 5) exploiting the misuse problem, the attacker then constructs a malicious URI which contains the attacker's code and a 'valid' STATE parameter; 6) launch CSRF attack (e.g., via malicious website) to send the crafted URI to the vulnerable application via the victim's browser; 7) For the forged request, the vulnerable application cannot correctly check the STATE parameter and mistakenly assumes the operation is from the victim.

The above attack assumes the attacker can obtain a valid STATE parameter of the victim somehow. This stringent requirement, however, can be bypassed by the different types of misuses of the STATE parameter.

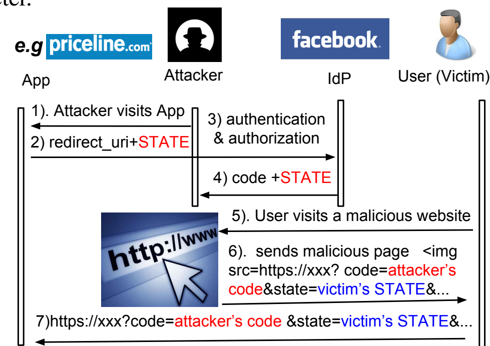

Figure 8: The Work Flow of CSRF When the STATE Parameter Is Misused

To enable the account hijacking, it requires the application to support the Identity Federation as described in Section 2.3.1 and the victim being logged into the application when the attack is launched. With these two prerequisites, this attack can bind the victim's application account to the attacker's IdP account. As a result, the attacker can stealthily log into the application as the victim later on with his own IdP account and manipulate/ operate the victim's account accordingly.