An Approach for Mitigating Potential Threats in Practical SSO Systems

Menghao Li[1,2], Liang Yang[1,3], Zimu Yuan[1(✉)], Rui Zhang[1], and Rui Xue[1]

[1] Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China
{limenghao,yangliang,yuanzimu,zhangrui,xuerui}@iie.ac.cn
[2] University of Chinese Academy of Sciences, Beijing, China
[3] School of Information Engineering, Tianjin University of Commerce, Tianjin, China

Abstract. With the prosperity of social networking, it becomes much more convenient for a user to sign onto multiple websites with a web-based single sign-on (SSO) account of an identity provider website. According to the implementation of these SSO system, we classify their patterns into two general abstract models: independent SSO model and standard SSO model. In our research, we find both models contain serious vulnerabilities in their credential exchange protocols. By examining five most famous identity provider websites (e.g. Google.com and Weibo.com) and 17 famous practical service provider websites, we confirm that these potential vulnerabilities of the abstract models can be exploited in the practical SSO systems. With testing on about 1,000 websites in the wild, we are sure that the problem that we find is widely existing in the real world. These vulnerabilities can be attributed to the lack of integrity protection of login credentials. In order to mitigate these threats, we provide an integral protection prototype which help keeping the credential in a secure environment. After finishing the designation, we implement this prototype in our laboratory environment. Furthermore, we deploy extensive experiments for illustrating the protection prototype is effective and efficient.

Keywords: Single Sign-on · Web security · Integrity

1 Introduction

As a convenient and popular authorization method, single sign-on (SSO) is widely deployed by multiple websites as a way for logging in with a third-party account. For example, you can easily log into Smartsheet.com and Rememberthemilk.com using your Google account instead of individual accounts from each of them. It means that your Google account is authorized to access their resources by both websites. SSO reduces password fatigue from different username and password combinations and time spent on re-entering passwords for the same identity.

Thanks to the prosperity of social networking, multiple SSO systems, such as OpenID [4], Google AuthSub [20], SAML [7], and OAuth [5, 13], have been widely deployed on commercial websites. The SSO system works through the interactions among three parties: a client browser (the user), the identity provider (IDP, e.g.

D. Lin et al. (Eds.): Inscrypt 2015, LNCS 9589, pp. 209–226, 2016.
DOI: 10.1007/978-3-319-38898-4_13

Google.com), and service provider (SP, e.g. Smartsheet.com). The security of an SSO system is expected to prevent an unauthorized client from accessing to a legitimate user's account on the SP side. Given the fact that more and more high-value personal data are stored on the Internet, such as cloud websites, the flaws in SSO systems can completely expose the private information assets to the hackers. It forces SSO system developers to try their best to patch the flaws or build up a safer SSO system. However, in recent years, more and more logic flaws and vulnerabilities have been discovered.

By analyzing many popular commercial websites, we abstract the practical SSO systems into two categories. The first category of SSO systems is deployed with OAuth2.0 protocol, which is standardized by RFC 6749 [11] and is used to replace the previous SSO systems such as OpenID and AuthSub. The previous work on OAuth2.0 mostly focuses on the formal analysis [2, 15, 29] and auto detection of the vulnerabilities [2, 39]. But they do not come up with practical solutions. We focuses on the practical OAuth2.0 SSO systems deployed on the commercial websites, such as Google and Weibo, then extracts the workflows of the practical SSO OAuth2.0 systems. Besides, we also analyze the independent developed SSO systems. We find that those independent developed SSO systems follow a simple communication model which has only three steps. Without doubt, we find that both of these categories of SSO models have vulnerabilities.

By rechecking the commercial websites under our built general SSO models, we find that almost all of them obey the models and the vulnerabilities are similar on each website. Moreover we also find that some websites deploy SSO systems that mix the two general model together. This mixed model makes the analysis a bit complex. But we still find the integrity problems in the mixed model. We give a real world example of the mixed model SSO system in Sect. 4.

As the vulnerabilities can all be attributed to the lack of integrity protection on the login credential, we attempt to protect the credential's integrity with cryptographic method and try to not affect the original performance of the SSO system. In this paper, we propose protection prototype in Sect. 5. Our prototype can prevent the attackers from stealing the victim's credential and logging into victim's account with the entire access rights as the original victim.

Contributions. We first classify current popular SSO systems into two categories and build two abstract SSO models for analyzing the security of practical SSO systems. Then we parse the workflow of two kinds of SSO models in depth and find the vulnerabilities in those models.

Second, we verify that the vulnerabilities which pervasively existing in practical SSO websites obey the logic vulnerabilities we discovered in the abstract models.

Our third contribution is attempting to design a protection prototype. For mitigating the vulnerabilities, we focus on the integrity protection of the credentials by binding them with a protected parameter. As the channel that has the user browser's participation is not secure enough, our protection prototype exploit a direct channel (or private channel) between IDP and SP to deliver the binding parameter. The prototype can guarantee the integrity of the credentials and mitigate the threats from the network attacker and web attacker. The evaluation also shows that the overhead of prototype's performance is low comparing with the original SSO model.

2 Abstract Models of SSO Protocols

In this section, we discuss about our abstract models which are extracted from the practical SSO systems. We parse these practical systems in our research and focus on the information and data exchange workflows in them. In order to construct the models, we first investigate those websites that provide SSO login method and parse the login APIs of these websites with practical login actions. We manually analyze the massive SSO login documentations and extract the key parameters that should be pay much more attention during the parse of practical SSO login actions. As a result, we classify our models into two categories, which are named independent SSO model and standard SSO model. The independent model reflects the SSO models which the websites developed independently. The standard model represent those websites who follow the standard SSO information exchange protocols such as [11].

In our analysis, we summarize that a basic SSO system contains three entities, which are named IDP (Identity Provider), SP (Service Provider) and Client (Users), and the communication channels that connect each of the three entities together. The IDP is a server or a service cloud that stores user's account and password. It provides authentication of the identity of an individual user and authorizes the SP to access user's account on the IDP side. The SP, which is also called RP(resource provider) in some previous researches, is also a server or cloud that provides application services, such as a forum website, a cloud storage or a news subscription website. The client, in our research, represents a web browser that is connected to the internet which plays both as a redirection device and a resource visitor.

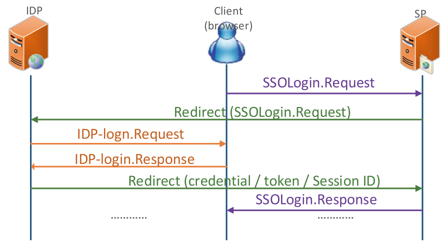

Fig. 1. Independent SSO model

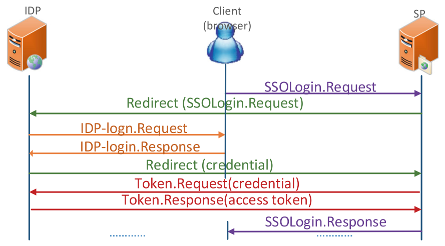

Fig. 2. Standard SSO Model

2.1 Independent SSO Model

In the independent SSO model, we find that the IDP and SP only exchange data or messages through the Client (which is specifically a web browser). The Client acts as redirect party who can get all the messages and data between the IDP and SP. In Fig. 1, we show the detail workflow of the independent SSO model and the key parameters delivered in the communication channels. In the model, we mark out three channels in

3 different colors. We call the 3 channels as SSO-login channel, redirect channel and IDP-side verification channel. The SSO-login channel is only between the Client and the SP(the purple part of Fig. 1). It represents the SSO login request and response round trip in the model, and it stands at the first and last steps in the workflow. The redirect channel exploits the redirect functionality of the Client's browser (the green part of Fig. 1). In this part, the Client works as the redirect device who has the ability to receive and forward the messages between IDP and SP. The verification channel is used to deliver the messages between IDP and Client for verifying the user's identity who is on the Client-side (the orange part).

Now, we depict the workflow of SSO login and authentication in this model step by step.

- **Step 1**: When the Client want to log in the SP using the SSO method, it generates an *SSOlogin.Request* and delivers the login request to the SP server through the SSO-login channel.
- **Step 2**: When the SP receives this SSO login request, a redirect channel is generated among IDP, SP and Client. Then the SP redirects Client's SSO login request to the IDP through the Client's browser which acts as a relayed device.
- **Step 3**: After the IDP gets the redirected SSO login request, The IDP firstly need to sponsor a verification channel with the Client directly. Then the IDP verifies the identity of the user by checking the user's username and password which is supplied from the Client.
- **Step 4**: Once the verification is successfully accomplished, the IDP responses a credential (it could also be a token or a session ID) to the SP using the redirect channel.
- **Step 5**: After the SP gets the redirected credential, it responses the Client with an *SSOlogin.Response* under the *SSOlogin* channel.

When the user on the Client side receives this *SSOlogin.Response*, the user is capable to browse the custom content on the SP server, such as the news subscription.

Security Analysis. First of all, we review the model from the communication entities' perspective. There are three entities on the inter-connected channels (IDP, SP and Client), we discuss the security capability of them respectively. As the IDP and SP are represented as the servers in the model, they could be mass-flowed Internet websites in the real world, such as Google and NetEase. These websites have large quantity of sensitive data, which need to be protected, and enough financial investment on the security part. So the IDP and SP have much stronger security capability than just a personal PC or laptop. However, on the opponent side, the Client could just be a computer or smart mobile device. The investment on these personal devices security is limited, many malwares and Trojans focus on exploiting the personal devices other than a website.

Next, we review the model from the communication channels' perspective. With the TLS/SSL encryption technics used in the Internet communication, it shows that an encrypted channel are safer than an unencrypted channel. However, our research shows that only a few practical SSO systems in this model used HTTPS (which supports TLS/SSL) as one of their communication channels.

From the security analysis on the two aspects, we can conclude that the messages which are redirected by the Client on the redirect channels could expose the content into insecure environment. The key point of the independent SSO model's security should be focus on the step 4 of the model's workflow. In other words, this model's security depends on the confidentiality and integrity of the significant parameters, such as credentials, tokens or sessionIDs in the redirect channel through in step 4.

2.2 Standard SSO Model

The IDP and SP exchange messages not only through the Client as the redirect party, but also through a direct connection between them. In Fig. 2, we show the detail of this model's workflow. Comparing with Fig. 1, it has 4 channels: *SSOlogin* channel, the redirect channel, the verification channel and the direct channel. As the first three channels have been described in Sect. 2.1, we skip the discussion on them. Here we focus on the fourth channel – the direct channel (the red part). This channel is built between the IDP and SP directly without the participation of the Client. The functionality of this channel is to check whether the credential is generated by the same IDP and exchange for the second credential– access token.

Now we depict the details of the login workflows in the standard OAuth2.0 SSO model. The first 4 steps are similar with the independent model, and the step 5 and step 6 shows the additional token exchange in this SSO model.

- **Step 1**: When the Client starts a login request to the SP using the SSO method, it generates an*SSOlogin.Request* and send it to the SP through the *SSOlogin* channel.
- **Step 2**: Then the SP redirects Client's SSO login request to the IDP through the Client's browser which acts as a relayed device.
- **Step 3**: After the IDP gets the redirected SSO login request in step 2, the IDP sponsors a verification channel with the Client directly. Then the IDP verifies the identity of the user by checking the user's username and password which is supplied from the Client. The step is shown as IDP-login.Request and IDP-login.Response in the orange part.
- **Step 4**: Once the verification is successfully accomplished, the IDP responses a primary credential to the SP using the redirect channel as the response to Redirect(SSOlogin.request).
- **Step 5**: When the SP gets the redirected credential, it does not directly response the Client on the SSOlogin channel. What the SP has to do is to resend the credential back to the IDP to get the access token on the direct channel, which is used to allow the user on the Client to access the resources on the SP. This step is shown as the Token.Request(*credential*) and Token.Response(access token) in Fig. 2.
- **Step 6**: After the SP gets the access token, it response the Client with an SSOlogin response through the firstly established channel.

Now if the user successfully passed all the 6 steps, he should be able to visit the special subscription recourses on the SP.

Security Analysis. We still analyze the standard model from two perspectives. From the perspective of communication entities, the vulnerability in the three entities lies on the Client side which has the weakest protection technic. From the perspective of communication channels, the vulnerability exists in the insecure channel. Here it refers to the redirected channel where the Client takes part in.

Combining these two aspects, our analysis focuses on the Client side and the communication channels nearby it. It means that the redirect channel is still significant in our security analysis.

As is shown in Fig. 2, the standard SSO model extends the independent model with extra credential exchange steps. These steps are used for checking the correctness and availability of the credential and exchange for the real token. In order to keep these steps secure, this model uses the private direct connection between the IDP and SP without the participation of the Client and the redirect channels. It makes the attackers on the redirect channel environment have no chance to get the access token for login. From this point, this model is much safer than the independent model.

But when we go further, we find that the standard model still has its vulnerability which is analogous to the independent model. The integrity of the credential in step 4 is still not well-protected. Even though the following steps provide the direct channel for the security, the attacker can still stealthily get the content that contains the victim's credential on the redirect channel. Neither the SP nor the IDP checks whether the credential matches the Client's identity.

3 Adversary Models

We consider two different adversary models called network attacker [2, 29] and web attacker [21] which have the potential capability to exploit the vulnerabilities of practical SSO systems.

3.1 Network Attacker

Network attacker can be separated into two categories: active attacker and passive attacker. The active attacker is capable to intercept and modify the packages in the channel where it lies. The passive attacker is only capable to eavesdrop the packages on the channel, but cannot intercept or modify them. We consider man-in-the-middle attacker as our network attacker model, which belongs to one of the active attacker patterns. The man-in-the-middle attacker can intercept the messages on the channel between Client and the IDP or on the channel between Client and the SP. The credentials redirected by the Client could be intercepted and modified by this attacker.

In practice, for mitigating the threats from the man-in-the-middle attack, many web-based data transfers are available only under secured channels (for example, HTTPS). The encrypted channel makes the man-in-the-middle attack becomes unavailable because the attacker cannot tell which parameter is the correct credential from the cipher text. However, recent researches have indicated that the encrypted channel cannot completely stop the man-in-the-middle attack on the Internet. The attacker is able to deploy some HTTPS proxies [33–37] on the channel between the Client and Server to intercept the encrypted data stream and modify them on the proxy. On those proxies, the messages are decrypted, the attackers can understand the messages and pick out the credentials in the data stream. The trick of these HTTPS proxies is to pretend to be the forged server to the real client or forged client to the real server. These proxies just sit in the middle, decrypting traffic from both sides. Here how to trick the victim to install these HTTPS proxies is a kind of social engineering attack projects, and it is out of the scope of our paper.

Figure 3 shows the two roles the attacker is able to play in the communication between client and server.

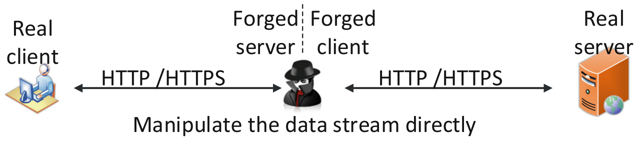

Fig. 3. Network Attacker

3.2 Web Attacker

Web attacker refers to those who control a malicious website on the Internet. The web attacker first lures the victim to visit this malicious website by following a malicious URI in a hyper-linked image or a malicious link address, such as a misleading link or image. When victim visits the malicious website, the attacker injects malicious code into victim's browser (e.g. XSS attack [30]) or replace victim's credential with attacker's (e.g. CSRF attack [28]). In the SSO login situation, the web attacker can require the victim delivering the credential to the malicious website under his control (XSS attack) or pushing the attacker's credential on the victim's browser for cheating the victim to login the SP as the attacker (CSRF attack).

Figure 4 shows the capability of the web attacker.

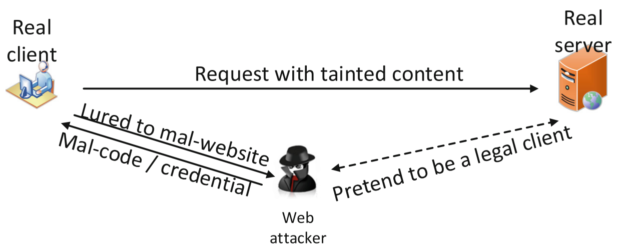

Fig. 4. Web Attacker

Our practical attack experiments (Sect. 4) and our protection prototype (Sect. 5) consider the threats under these two adversary models.

4 Case Study of Practical SSO Websites

In this section, we discuss our practical attack experiments on some of those famous websites in China, including Google, Weibo [22, 24], Tencent QQ [14], Alipay [17, 27], Taobao [26]. These five websites that we picked out all play the role of the IDP. Besides the Alipay websites deploys as our independent SSO model, the rest implement the standard OAuth2.0 SSO model we summarized in Sect. 2. For each IDP, we register two test account, namely Alice and Bob, and test whether the vulnerabilities work when logging into a practical SP. In our experiments, we login Bob's account with Alice's username and password by stealthily getting Bob's credential when Bob starts his login workflow.

Our experiment environment is as follow. First of all, we build up a local area network (LAN) to impersonate our test environment and connect two computers to the LAN. Then we deploy windows 7 as the operating system and play the role of victim (which means to be Alice) on one of the computers. We deploy Ubuntu14.10 as the attacker (which means to be Bob). On the Alice's computer, we install a web debugger tool – fiddler [9] for analyzing the web packages the victim gets and sends. On the Bob's computer, we install mitm-proxy [33], which is able to intercept the HTTPS data stream traffic on it, to filter the victim's SSO login messages for intercepting the Alice's login credentials.

4.1 Google Account

There are many service provider websites deploy Google account as one of their login method. In this part, we choose an online project management software – *smartsheet.com* [23] as our test SP. Although there are some SSO flaws have been reported in the previous research [3], their research focuses on the logic flaws on the smartsheet.com that the developers do not consider carefully and talks little about the vulnerabilities in the SSO protocol which is implemented between Google and Smartsheet. Besides, when we begin our study, Google has changed its SSO protocol from OpenID to OAuth2.0. So we cannot directly get experience from the previous research.

Fortunately, our study shows that the Google SSO login model follows our standard SSO model in Sect. 2.2. In our experiments, we register two new Google accounts, for example, *Alice@gmail.com* and *Bob@gmail.com*, and login smartsheet.com.

We search Alice's decrypted messages on the proxy and find the credential is named as *code*. Then we let Bob intercept Alice's following data traffic and stealthily keep Alice's code value in Bob's proxy. Now we start Bob's login workflow and also block the data stream when Bob gets his own *code*. Then Bob replaces his own *code* with Alice's, which is cut from her login workflow, and releases the modified redirect data stream to *smartsheet.com*. Without doubt, Bob successfully logs into Alice's account and controls the whole content of Alice's. Now Bob can do whatever he want to on the Alice's account.

During our impersonated attack, the only protection on this redirect message depends on the HTTPS protocol. But the integrity of this *code* is not protected. That is why Bob can exploit Alice's account without being detected by either *Google* or *smatsheet.com*.

4.2 Weibo.Com

Weibo.com also depends on standard OAuth2.0 SSO framework. It redirects the login credential through user's browser to the SP and it also calls this credential as *code*. However, different from the Google SSO login method, *Weibo* does not implement encrypted channels among the three abstract entities. Both network attacker and web attacker can be able to easily steal the victim's login credential.

In our experiment, we choose *Baidu* [38], a famous search engine service and cloud storage service provider in China, as the instance of the SP server. Like what we do in the Google case, we also register two *Weibo* accounts, which we still call them Alice and Bob, and confirm the availability of each account. Then we start our vulnerability exploit test. We put Bob on the proxy which Alice's login messages have to go through. On the proxy, we filter Alice's traffic data stream and search for the login credential which *Weibo* redirects to *Baidu*. As the channels are not encrypted every network package on the internet is displayed in plaintext. Bob is able to read Alice's packages directly and gets the login *code* of Alice's *Weibo* account.

Weibo redirects the *code* through a piece of JavaScript code in the response to the Alice's browser. The JavaScript code of Alice and Bob are shown as below:

On Alice's side, the code is as follows:

```
<script language=`javascript'>
callbackfunc({
http://baidu.com/.../afterauth?mkey=xxx
&code=code-of-alice});
</script>
```

On Bob's side, the code is as follows:

```
<script language=`javascript'>
callbackfunc({
http://baidu.com/.../afterauth?mkey=yyy
&code=code-of-bob});
</script>
```

Comparing the JavaScript code of two accounts, we find that the only difference of the redirect URI is the parameters: *code* and *mkey*, where the code is the login credential and the mkey is a ticket for preventing the CSRF attack. On the browsers, we intercept the redirection of the credentials of both Alice and Bob and replace Bob's *code* with Alice's. Then we redirect the modified Bob's URI back to *Baidu*. As a consequence, *Baidu* accepts the modified URI and regards Bob as Alice because Bob gives *Baidu* Alice's credential.

4.3 Alipay.Com

Alipay.com is an online payment and e-commerce management website (like PayPal) hosted by the Alibaba Group, a very famous Chinese online trade company. In practice, Alipay accounts can be used to login some other popular websites in China, such as

Xunlei and *Youku*. In our test, we choose *Xunlei* as the test SP and login it with Alipay accounts. Alice still plays the role of victim and Bob is the attacker.

In our test, we find that the Alipay is not following our standard SSO model, it is constructed under the independent SSO model which is discussed in Sect. 2.1. The SP does not resend the credential back to IDP for checking the validity. So we focus on the credential, which has been redirected through the user's browser, and detect whether it could be modified without being known by the SP.

Unfortunately, our test shows that the credential is composed with three parameters which is very different from the only one parameter in the standard OAuth2.0 model. These three parameters are *User_ID*, *token* and *sign*.

Although there exist a signature to protect the credential, we still find a way to let Bob hack into Alice's Alipay account. We test the Alipay SSO login method a lot of times, and find that the signature *sign* only protect the parameter of *token*.

It means that we can modify the *User_ID* to any value we want without being detected by *Xunlei.com*. Furthermore, we discover that the *User_ID* is a constant and plaintext. Each time we login no matter Alice's account or Bob account, the User_ID is an invariant. It means that the User_ID is guessable which is similar to the vulnerabilities in [2, 3, 15]. What the attacker need to do is to follow some rules to guess a legal User_ID. With this guessed User_ID attacker can log into any legitimate user's *Xunlei* account and get their sensitive data.

The Alipay SSO system also deploy a piece of javascript code as the redirect method. At the same time, its redirect messages only depend on HTTP which is insecure for delivering URL and significant parameters. The redirection URI is like: `http://xunlei.com/…/entrance.php?…token=xxx&user_id=USERID&sign=xxx&…`

Unlike the vulnerability in the standard OAuth2.0 SSO model, this vulnerability can be attributed to the logic flaws when the developers design the entire system. So it only suit for the Alipay SSO system and is not universal.

4.4 Taobao.Com

Taobao.com [26] is the most famous online shopping website in China. It also provides SSO login method, which is called AliSSO system. AliSSO system mixes the features of both independent SSO model and standard model together. From the perspective of the three entities of IDP, SP and Client, AliSSO follows the independent SSO model. When the credential is got by the SP, it does not need to send it back to IDP for checking the validity.

However, the SP does not directly accepts this credential. AliSSO separates the SP into two parts, in which, one is a resource server and the other is an authentication server. The resource server stores the user's data and information and provides services to the user. The authentication server is in charge of certificating the identity of the legitimate user. When the SP gets the credential, it firstly generates another access token and redirects the token to the authentication server through user's browser after the authentication server gets the second access token, it generates a ticket and directly send to the resource server without the participation of user's browser. These steps are much more like the standard OAuth2.0 SSO model.

In our experiment, we choose *weibo* as an instance of our SP websites. Then we register two *taobao* accounts, namely Alice and Bob, and confirm the availability of each account. After that we begin our vulnerability exploit test. We suppose Bob as the attacker and put it on a proxy which Alice has to go through.

When we catch the data stream of Alice between taobao and weibo, we find that it is hard to modify the credential, which is named as *tbp*. As this parameter is protected by a signature, any change of the *tbp* will not be accepted by weibo. Then we let Alice's login workflow continues. After weibo gets the credential tbp and check the signature, it generates a second credential and redirects it to the authentication sub-server, login.weibo.com. This redirection also goes through Alice's browser, we can catch it on the proxy. When the sub-server gets the second credential, *alt*, it directly send *alt* to resource.weibo.com following the standard OAuth2.0 SSO model. After resource.weibo.com gets the *alt*, it responses Alice with her personal content.

In this login workflow, we find the second credential, alt, is not well protected. As Bob is on the proxy that Alice has to go through, he can replace his alt with Alice's and login Alice's account on weibo.com without any prevention from either weibo.com or taobao.com.

We have reported this vulnerability to the technic support group of *Weibo*, and got their thanks email in two days. Before we write our paper, this vulnerability has been patched.

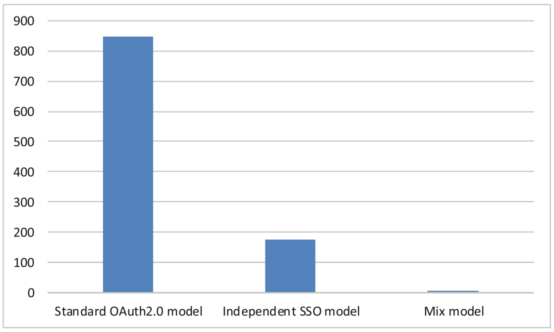

Fig. 5. Classified SSO Models

In practice, we have tested 1,037 websites manually. Most websites, except Google, in our experiment are located in China because some most famous websites, such as Facebook and Twitter, cannot visit in China mainland. But this problem does not affect our research. The conclusion of our tests is that most websites deploy the standard OAuth2.0 SSO model. The rest are independent SSO model and mixed SSO model (such as the taobao.com). The mixed model is not a new model, it is just combined from the two abstract SSO models together. The classified model graph is shown in Fig. 5. Then we pick up 9 typical SP websites and 5 IDP websites from our tested SSO websites. And we list the vulnerabilities and flaws of them in Table 1.

Table 1. SSO threats in real-world websites

SP / IDP	Smart-sheet	Remember-themilk	Weibo	Baidu	Youku	Sohu	Xunlei	Iqiyi	JD
Google	△	△							
Weibo				△	△	△	△	△	
QQ			△	△	△	△		△	△
Alipay					□	□	□	□	
Taobao			○				○		

Note: △ – Standard OAuth2.0 SSO model; □ – Independent SSO model; ○ – MixedSSO model;

5 Integrity Protection and Threat Mitigation

We can attribute the vulnerabilities we discuss in previous sections to the lack of the login credentials' integrity protection. In this section, we give out our prototype scheme for protecting the login credentials integrity. Our prototype can mitigate the threats from the network attack and web attack which are under the adversary models in Sect. 3. We build up our test environment in our lab with a LAN and two servers which play the roles of IDP and SP. Then we implement our prototype on those two servers and test it through another computer which acts as the Client. Finally, we compare the performance of our prototype and the original SSO system. The consequence shows that the performance of our prototype is acceptable.

5.1 Prototype Design

Our basic purpose is to avoid web attackers or network attackers stealing the legitimate user's login credentials and protect the credentials integrity. In this part, we first describe how our prototype prevents the web attackers and then we talk about how it prevents the network attackers. The workflow of our prototype is shown in Fig. 6.

Protection from Web Attackers. We use Same Origin Policy (SOP) [32] and HTTPOnly Policy [31] on the SP side to perform the protection. This protection can avoid attacker luring victims to login attacker's account unconsciously.

On the SP side, we add a parameter, *stat*, in the SSO redirect URL and set the browser's *cookie* with a parameter, *signstat*, which is a signature of stat and label this cookie as HttpOnly. When the IDP gets the redirect URL, it regards the parameter of *stat* as a component of the URL and append the credential after it. Then the IDP delivers it to the Client's browser. When the redirection URL that contains the credential and *stat* comes into the Client's browser, the browser redirects the credential to the SP with *cookie* back. When the SP gets the credential, *stat* and *cookie* back, it first computes whether the signature of *stat* in the URI matches the signature value in the *cookie*. If the signature of *stat* matches the value in the *cookie*, it means that this URL is not from the web attacker. The SP believes the user on the Client is a legitimate user.

The security of this design of *stat* depends on SOP and HTTPOnly which need the participation of the cookie. As the web attacker lures the victim to visit a malicious website under his control, the attacker prefers to put his own credential as a redirect URL in the response and send back to victim browser. When the victim gets the redirect URL that contain attacker credential, the browser wants to send the URL to the SP. If there is not protection, the attackers credentials would be send to SP and the SP would regard the victim as the attacker. In case the victim does not notice that he has logged into a wrong account and upload some significant files in this account, attacker can get those files a few minutes later just by legally login his account. However, with the help of SOP and HTTPOnly, this threat is blocked.

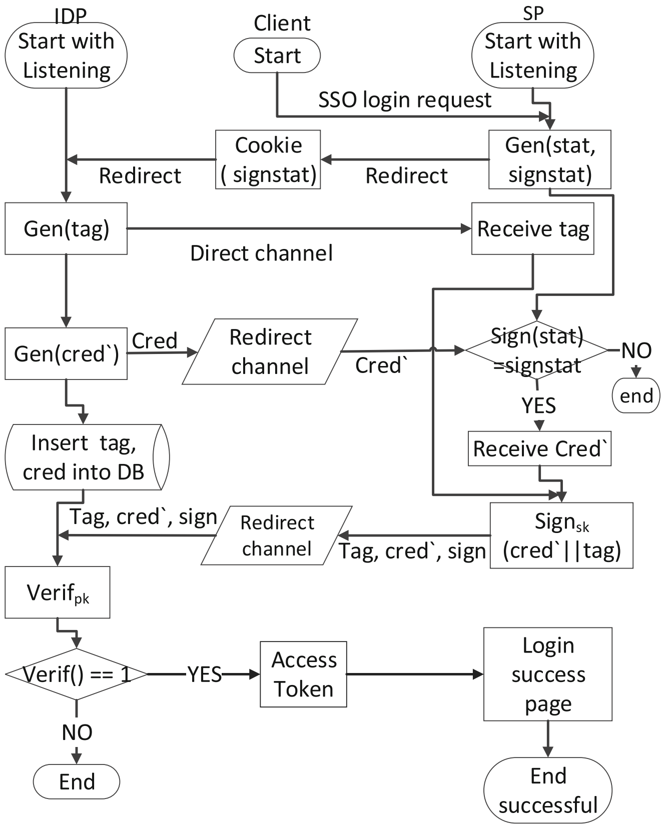

Fig. 6. WorkFlow of the Protection Prototype

Protection from Network Attackers. In order to mitigate the threats from network attackers, we need the participation of both IDP and SP. Besides, we also need two different channels: one is the redirect channel through the Client, the other is the direct or private channel between the IDP and SP.

In our adversary models, the network attacker can hack into an encrypted channel with the help of the SSL-proxy tools (such as mitmproxy). What the attacker need to do is to stealthily install a HTTPS proxy certificate on the victim's computer. But this work is out of our scope, we do not discuss it in our paper. This strong capability makes the confidentiality invalid on the HTTPS channels. In this situation, the integrity of the credential becomes a very significant point in the SSO system. But neither standard OAuth2.0 framework nor independent developed SSO system protect the integrity very well. We have easily logged into another user's account without knowing his or her username and password (Sect. 4). For mitigating the threats from the network attackers, we use the direct channel between IDP and SP to deliver a binding parameter, which we call it *tag*, for verifying the credential's integrity. Supposed that this direct channel is invisible in the attacker's view. So the *tag* is delivered securely between IDP and SP. After IDP delivers the *tag* directly to SP, it generates a corresponding credential which is bonded to the tag. And we let the IDP keep the pair of the original $(tag,\ credential)$ in its database for checking the integrity of credential that delivered back from the SP. Then the IDP redirect the credential to Client's browser. On the SP side, it gets the tag from the direct channel and gets the credential from the redirect channel. Once the SP gets the login credential, we call credential' from the redirect channel, it binds the credential and the tag with a signature function $sign_{sk}(credential' || tag)$. The *sk* is the secrete key which is negotiated between IDP and SP. It is used for signing the value of $credential' || tag$. Then SP delivers the signature back to IDP through the direct channel with the $(tag, credential')$ pair. Correspondingly, the IDP has a public key *pk* for verifying the signature. After the IDP gets the signature and $(tag, credential')$ pair, it first searches the database with the value of tag. Then IDP verifies the signature of $sign_{sk}(credential' || tag)$ with the verify function $verif_{pk}(tag, credential, sign_{ak})$. If the verification successes $(verif_{pk} = 1)$, it means that the attacker does not modify the credential when redirecting it. At this time, the IDP sends the access token directly to the SP, then SP notices the Client it has logged in SP successfully. If the verification fails, IDP reports an error and drop the $(tag, credential)$ pair in the database.

5.2 Implementation

We deploy two desktop computers to impersonate the real SP and IDP called s-SP and s-IDP. Both of the computers have an Intel Core i7-3770 3.4 GHz CPU and 4 GB memory. The operation system is Ubuntu 14.10 LTS. We install the service software, including PHP 5.5.11, Apache 2.4.9 and MySQL server 5.6, and configure the web environment on both computers.

In our implementation, we deploy our prototype on the standard OAuth2.0 SSO framework and we call the login credential as *code*. In order to simplify the workflow of the impersonated SSO system, we omit the user's IDP-login steps. When an SSO login request comes from s-SP, s-IDP circumvents the verification steps and directly begins the authorization and login operations. During the authentication and authorization steps, we give s-SP a secrete key, *sk,* for signing the *code* with a binding parameter, *tag,* which is got through the direct channel from s-SP, and we give s-IDP a public key *pk* for verifying the signature of *code* that is given by the s-SP.

On the s-SP side, we add a parameter, stat, for preventing the attack from a malicious website. This parameter not only exists in the redirect URL but also has a signature in the user browser cookie. With the help of the SOP and HTTPOnly policies, the web attackers cannot get the signature of *stat* in the cookie between browser and the real SP. Once the forged *stat* is delivered back to s-SP, the server finds that the *stat* does not match the signature in the cookie and it will stop the following login workflow. This parameter can perfectly prevent the CSRF and XSS attacks that are sponsored by the web attackers.

Another thing need to pay attention on the s-SP is the synchronization of the parameters for generating the signature. Here they refer to tag and code specifically. It should be careful to handle this problem, because tag and code come from different channels. The tag comes from the direct channel between the s-IDP and s-SP and it is delivered to s-SP before the code. But the code comes through the redirect channel which is relayed from the user's browser. These two parameters cannot arrive at s-SP at the same time. If we do not consider the synchronization of these two parameters, s-SP may put Alice's code and Bob's tag together and compute a signature of the mixed-user parameters which is not correct for the s-IDP for verification. This problem might cause Bob logs into Alice's account. Our solution on this problem is simple. We build a concurrence lock on the s-SP side, which makes the s-SP can only deal with one user's login request.

5.3 Evaluation

Our implementation is about 100 lines of PHP and JavaScript code. Our evaluation depends on the execution time of the code. We set two timestamps in the entire login workflow. The first one is set at the SSO login page, when the user clicks the SSO login button, we get a timestamp. The second one is set on the login success page, if the user login successful, we record the second timestamp. The execution time is the difference of the two timestamps. Then we execute 400 times, and get the average time as the general execution time. The comparison between the original SSO model and our protection prototype is shown in Fig. 7.

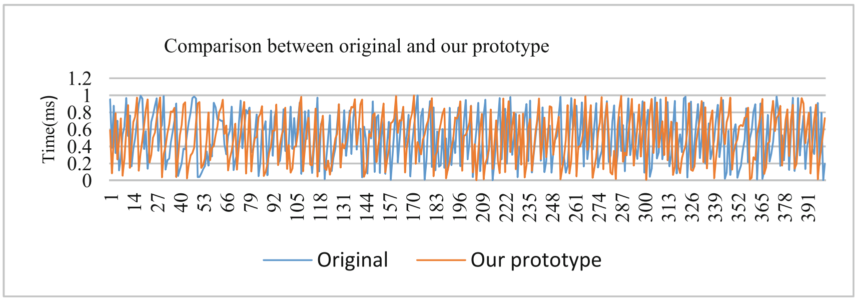

Fig. 7. Time spending comparison between original SSO model and our protection prototype

For the performance, we compare our prototype with the original SSO model which do not show any protections on the integrity of the credentials. Averaged 400 independent executions of each model, the overhead of the protection prototype is only increased by 0.418 % compared with the original SSO model. It means that the performance of our prototype is acceptable.

6 Related Work

Many previous works have been done to study the security of SSO systems. Wang et al. [3] discovered the SSO flaws in OpenID [4] and Flash. The flaws of OpenID cause the IDP to exclude the email element from the list of element it signs, which is sent back to the SP through a BRM. When the flaws of OpenID are reported to Google by the authors, Google replaces OpenID with OAuth2.0 as the SSO system [18, 19]. Armando et al. [10] studied on SAML-based SSO for Google Apps and gave the formal analysis of SAML 2.0 [6, 7] web browser SSO system. They used formal method to extract the abstract protocol in SAML 2.0 and built up the formal model of SAML. Somorovsky et al. [1] did a lot of researches in revealing vulnerabilities in formal SAML SSO systems. They revealed the threat from XML signature wrapping attacks is a big problem in the systems.

Bansal et al. [15] and Sun et al. [29] discovered the attacks on OAuth2.0 by formal analysis of the basic document of RFC 6749 [11]. They analyzed the formalized OAuth2.0 protocol and revealed that the potential threats coming from CSRF attack or token stolen during the redirection.

Before we finish our work, a vulnerability named *Covert Redirect* [16, 25] was reported about the OAuth2.0 on the Internet. It describes a process where a malicious attacker intercepts a request from an SP to an IDP and changes the parameter called "redirect_uri" with the intention of causing the IDP to direct the authorization credentials to a malicious location rather than to the original SP, thus exposing any returned secrets (e.g. credentials) to the attacker.

Zhou et al. [39] have built an automated SSO vulnerabilities test tool. This tool can detect whether a commercial website exists popular vulnerabilities, such as access_token misuse or OAuth credentials leak. But they only deploy the Facebook as the IDP site.

7 Conclusion

In this paper, we disclose the reason of the vulnerabilities that exist in commercial web SSO systems. We studied the SSO systems on 17 popular websites and classified them into two abstract models. Then we verify our models on about 1,000 SSO supported websites in the wild. Most websites follow the standard OAuth2.0 SSO model but there still some other websites prefer developing their own SSO system that depends on the independent model. We also elaborate our security analysis on these practical commercial websites that deploy different SSO models. That is the credentials could be intercepted by the attackers to log into the SP as the victim. For mitigating the threats focus on the credential's integrity, we give our protection prototype on guaranteeing the integrity of the credentials which is simple and efficient to deploy in practice. It not only fixes the vulnerabilities of the two abstract SSO models and the mixed model, but also mitigates the threats from the two adversary models mentioned in Sect. 3. However, our prototype also has its limitation. For example, on the SP side, it does not support concurrent SSO requests so far. Our prototype has to deploy on both IDP and SP server-sides. That is a trivial and cumbersome work. In the future work, we want to improve our prototype on these two problems and try our best to make our protection prototype to be a convenient independent third party middle-ware which can be deployed on any IDP or SP websites.

Acknowledgement. This work is supported by the "Strategic Priority Research Program" of the Chinese Academy of Sciences, Grants No. XDA06010701, National Natural Science Foundation of China (No.61402471, 61472414, 61170280), and IIE's Cryptography Research Project. Thanks to Wei Yang for helping recording the experiments. Thanks to a number of anonymous reviewers and Prof. Jian Liu who gave us very useful feedback on a previous version of this paper.

References

1. Juraj, S., Andreas, M., Jörg, S., Marco, K., Meiko, J.: On breaking SAML: be whoever you want to be. In: USENIX Security (2012)
2. Bai, G., Lei, J., Meng, G., Venkatraman, S.S., Saxena, P., Sun, J., Liu, Y., Dong, J.S.: AUTHSCAN: automatic extraction of web authentication protocols from implementations. In: NDSS (2013)
3. Wang, R., Chen, S., Wang, X.: Signing me onto your accounts through facebook and google: a traffic-guided security study of commercially deployed. In: IEEE S&P (2012)
4. OpenID. http://openid.net/
5. OAuth Protocols. http://oauth.net/
6. Technology report SAML protocol. http://xml.coverpages.org/saml.html
7. SAML2.0 Wikipedia. http://en.wikipedia.org/wiki/SAML 2.0
8. Wang, R., Chen, S., Wang, X., Qadeer, S.: How to shop for free online security analysis of cashier-as-a-service based web stores. In: IEEE S&P (2011)
9. Fiddler–The free web debugging proxy. http://www.telerik.com/fiddler
10. Armando, A., Carbone, R., Compagna, L., Cuellar, J., Abad, L.: Formal analysis of SAML 2.0 web browser single sign-on: breaking the SAML-based single sign-on for google apps. In: ACM FMSE (2008)
11. OAuth2.0 Authorization Framework. http://tools.ietf.org/html/rfc6749
12. Google Accounts Authentication and Authorization. https://developers.google.com/accounts/docs/OAuth2
13. OAuth2.0 documentation. http://oauth.net/documentation/
14. Wikipedia Tencent. http://en.wikipedia.org/wiki/Tencent
15. Bansal, C., Bhargavan, K., Maffeis, S.: Discovering concrete attacks on website authorization by formal analysis. In: IEEE CSF (2012)
16. Covert Redirect. http://tetraph.com/covert_redirect/
17. AlipayOpenAPI. https://openhome.alipay.com/doc/docIndex.htm
18. Google Accounts authorization and authentication Open ID 2.0 migration. https://developers.google.com/accounts/docs/OpenID?hl=en-US
19. Google Accounts authorization and authentication Using OAuth2.0 for login (OpenID Connect). https://developers.google.com/accounts/docs/OAuth2Login?hl=en-US
20. Google AuthSub. https://developers.google.com/accounts/docs/AuthSub
21. Akhawe, D., Barth, A., Lam, P.E., Mitchell, J., Song, D.: Towards a formal foundation of web security. In: CSF (2010)
22. Sinaweibo, Wikipedia. http://en.wikipedia.org/wiki/SinaWeibo

23. Smartsheet.com, one online project management software. https://www.smartsheet.com/
24. Weibo openAPI. http://open.weibo.com/wiki/
25. Covert Redirect Vulnerability Related to OAuth 2.0 and OpenID. http://tetraph.com/covert_redirect/oauth2_openid_covert_redirect.html
26. Taobao, Wikipedia. http://en.wikipedia.org/wiki/Taobao
27. AlipayWikipedia. http://en.wikipedia.org/wiki/Alibaba_Groupn#Alipay
28. Cross-Site Request Forgery (CSRF), The Open Web Application Security Project (OWASP). https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF)
29. Sun, S.T., Beznosov. K.: The devil is in the (implementation) details: an empirical analysis of OAuth SSO systems. In: ACM CCS (2012)
30. Cross-Site Scripting (XSS), The Open Web Application Security Project (OWASP). https://www.owasp.org/index.php/XSS
31. HttpOnly, The Open Web Application Security Project (OWASP). https://www.owasp.org/index.php/HttpOnly
32. Same Origin Policy, W3C Web Security. https://www.w3.org/Security/wiki/Same_Origin_Policy
33. MitmProxy, An interactive console program that allows traffic flows to be intercepted, inspected, modified and replayed. https://mitmproxy.org/
34. SSL Man in the Middle Proxy. http://crypto.stanford.edu/ssl-mitm/
35. Cloudshark Appliance. https://appliance.cloudshark.org/
36. SSLsplit - transparent and scalable SSL/TLS interception. https://www.roe.ch/SSLsplit
37. Sslsniff, A tool for automated MITM attacks on SSL connections. http://www.thoughtcrime.org/software/sslsniff/
38. Baidu, Wikipedia. http://en.wikipedia.org/wiki/Baidu
39. Zhou, Y., Evans, D.: SSOScan: automates testing of web applications for single sign on vulnerabilities. In: 23rd USENIX Security Symposium (2014)