The Power of Recognition – Secure Single Sign-On using TLS Channel Bindings

Jörg Schwenk
Ruhr-University Bochum
Bochum, Germany
joerg.schwenk@rub.de

Florian Kohlar
Ruhr-University Bochum
Bochum, Germany
florian.kohlar@rub.de

Marcus Amon
Ruhr-University Bochum
Bochum, Germany
marcus.amon@rub.de

ABSTRACT

Today, entity authentication in the TLS protocol involves at least three complex and partly insecure systems: the Domain Name System (DNS), Public Key Infrastructures (PKI), and human users, bound together by the Same Origin Policy (SOP). To solve the security threats resulting from this construction, a new concept was introduced at CCS '07: the *strong locked same origin policy (SLSOP)*. The basic idea behind the SLSOP is to strengthen the *identification* of web servers through domain names, certificates and browser security warnings by a *recognition* of public keys to authenticate servers. Many weaknesses of current protocols emerging from an insecure PKI or DNS can thus be handled, even without involving the user. This concept has also been adapted by the IETF in RFC 5929.

The contribution of this paper is as follows: First we present a new SLSOP-based login protocol and use it to design a secure Single Sign-On (SSO) protocol. Second we provide a first full proof-of-concept of such a protocol and also the first implementation of the channel binding described in RFC 5929, implementing a cross-domain SLSOP both for a new type of authentication cookies, as well as for the HTML-based POST and Redirect bindings. Finally we evaluate the security of this protocol and describe, how our protocol copes with modern attack vectors.

Primary Classification:
H. Information Systems - H.m MISCELLANEOUS
Additional Classification:
E. Data - E.m MISCELLANEOUS
General Terms:
Security
Keywords:
DNS, PKI, TLS, Identity Management
Single Sign-On, SOP, SLSOP

1. INTRODUCTION

Today, complex and error-prone *identification* is used on the Internet, where much simpler *recognition* techniques may be sufficient. This can best be illustrated by HTTP session cookies, in combination with TLS with server authentication:

- HTTP session cookies are stored under the Same Origin Policy (SOP), i.e. a human-readable domain name is the access key. Thus attacks on the Domain Name System (DNS, e.g. [18]) directly influence the security of these cookies.
- Typically it is argued that domain names are protected by TLS server certificates. However, it has been shown that weaknesses in the hash function MD5 can be used to fake valid certificates for any domain [34]. Moreover, most Internet users tend to ignore browser warnings on invalid certificates [8].
- Now if the browser is presented with a fake, but valid certificate, or if the user accepts a fake certificate, the HTTP session cookie is sent to the wrong server, and the security of the session is compromised.

Thus the security of an HTTPS session not only depends on the (local) security of the TLS protocol, but also on the (global) security of the DNS and the Public Key Infrastructure (PKI) for TLS server certificates, and last but not least on user behavior. The problems resulting from the complexity of this identification process became obvious with the advent of Phishing and Pharming attacks starting in 2004 (although they have been described at least seven years earlier), and meanwhile DNS [18] and MD5 based PKIs [34] have been completely broken.

A much simpler method has been proposed by Karlof *et al.* at CCS 07 [20], and independently by Masone *et al.* at FC 2007 [28]: The Strong Locked Same Origin Policy (SLSOP) needs neither DNS nor PKI: SLSOP cookies are stored using the public key of the TLS server certificate (or the hash of this certificate, etc. [2]) as the access key, and it even works with self-signed certificates. In 2010 Channel Bindings for TLS have been proposed by the IETF for standardisation in RFC 5929 [2]. One of these adapts the idea of making a hash of the server public key available for higher layer protocols, e.g. Javascript, HTML, HTTP or others.

Thus, the contributions of this paper are as follows: First we show, how to form a secure browser-based Single Sign-On (SSO) protocol based on the ideas of TLS Channel Bindings and Strong Locked Origins and then we show the applicability of these concepts by presenting a proof of concept

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. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
DIM'11, October 21, 2011, Chicago, Illinois, USA.
Copyright 2011 ACM 978-1-4503-1006-2/11/10 ...$10.00.

(realized as a browser extension for Firefox 3.6). Our solution yields the first known implementation of RFC 5929 [2] and implements this secure SSO protocol both for a novel type of cookies (which we call SLSOP-cookies) as well as for the standard HTTP POST and Redirect bindings which are used in most internet services as of today. We will also provide evidence that our protocol is easy to deploy (especially if relying on the HTTP bindings), as only small changes have to be applied to the processing of HTTP requests over an SSL transport.

2. RELATED WORK

SLSOP and WSKE-Cookies. At the CCS 2007 Karlof *et al.* presented a strong attack on the domain name system called *Dynamic Pharming.* They discovered several flaws in the domain name system that could lead to scenarios in which a domain is able to impersonate any other domain. As access decision based on the ordinary same-origin policy only rely on used domain, protocol and port they proposed to add several cryptographic means in order to prohibit such exploitation of DNS bugs to gain unauthorized access. They proposed to improve the common same-origin policy by presenting two new policies called (weak and strong) locked same-origin policies [20]. The strong locked SOP associates secured web content to the originating server's public key and certificate. The weak locked same-origin policy, being easier to implement but less secure, relied on the validity of certificate chains, so server's with mismatching or self-signed certificates were unable to access content secured by this policy.

A similar approach for cookies was presented at FC '07 by Masone *et al.* [28]. They proposed that cookies set by a web server after an initial TLS handshake were only to be returned to that same server when the identical keypair is used in following TLS sessions.

Channel Bindings for TLS. In 2010, RFC 5929 [2] was published as proposed standard. The authors of this document describe three types of channel bindings for TLS, namely "tls-unique", "tls-server-end-point", and "tls-unique-for-telnet". In the "tls-server-end-point"-description, the binding to the TLS channel is achieved using a hash of the server's public key, which is essentially identical to the ideas presented in this paper.

Attacks on DNS and on MD5 based PKIs. In 2008, Dan Kaminski gave a talk at Blackhat [19] where he presented a DNS cache poisoning attack based on the birthday paradox, with an attack complexity of 2^8. This vulnerability could not be patched within the DNS protocol itself, the UDP protocol was used to fix it. But even with UDP source port randomization employed, the attack only has complexity 2^{16}, so DNS is not (and never was) a secure system.

The fact that the hash function MD5 has random collision is known for a while now [7]. In 2009, this fact was used by Stevens, Lenstra and de Weger in a first realistic attack on MD5 based PKIs for TLS server certificates [34]. The team was able to compute a fake certificate with the same MD5 hash as a valid TLS certificate, with the difference that it could be used as a CA certificate. Thus arbitrary new certificates, for any domain name, could be issued.

SSO (OpenID, Cardspace, SAML). A Single Sign-On protocol is enabling a user/client to authorize and authenticate to some service provider. The client uses a token (in most cases a cookie), which he obtains from a trusted third party (TTP) after authenticating to this party (usually by means of username/password or hardware tokens), in order to gain access to all services to which this user is entitled to. Over the past years, SSO evolved from the original MIT Kerberos protocol to a browser-based variant and now exists in various types. The benefit of using such a three-party protocol is that it suffices to remember only one loginID and one password for all services (namely the one used to authenticate to the TTP).

The original Kerberos protocol and related three party schemes have been intensely studied without finding severe security deficiencies [5, 3]. However, especially the browser-based federated identity management protocols that later evolved turned out to have some vulnerabilities: In addition to the Microsoft Passport analyses of Kormann, Rubin and Slemko [22, 33], Groß analyzes SAML, an alternative framework for the exchange of authentication and authorization information used for Single Sign-On protocols [12, 26]. He shows that SAML is vulnerable to adaptive attacks where the adversary intercepts the authentication token contained in the URL. The flaws in the SAML protocol have led to a revised version of SAML [6]. Groß and Pfitzmann analyzed this version, again finding the need for improvements [13]. Similar flaws have also been found in the analysis of the Liberty Single Sign-On protocol [30] due to Pfitzmann and Waidner. The authors point out some weaknesses against man-in-the-middle attacks. Another variant of SSO protocols are distributed (or decentralized) Single Sign-On protocols, e.g. OpenID [31]. To access a server through OpenID, a user must have created an OpenID-Identity, provided by an OpenID-Provider. Due to the distributed nature of the architecture of this protocol there exist many OpenID-Providers.

Secure SSO with Client certificates. There has also been some work on adding client certificates to Single Sign-On protocols. At ESORICS 2008 Gajek *et al.* proposed a variant of a browser-based Kerberos scheme which uses TLS with client authentication[11]. At SWS '08 Gajek *et al.* presented stronger SAML bindings for TLS [32] and at ARES 2010 Kohlar *et al.* presented a Single Sign-On protocol [21] that uses SAML-tokens for the login, gaining the advantage of being able to sign and/or encrypt these tokens for the receiving party.

Recognition for TLS. For TLS, there are three possible recognition scenarios, with different roles for client and server:

RECOGNITION OF THE BROWSER: CLIENT CERTIFICATES. In this scenario, the server recognizes the client based on a TLS client certificate. This is similar to the SLSOP, with the duties of client and server exchanged. Again, no complex PKI validation strategies are needed, the server may simply use the public key from the (probably self-signed) client certificate (or an hash of the certificate, etc.) as the access key for an authorization database.

The security of this solution is well-studied, since client authentication is part of the TLS handshake here. For login, this solution is available on all browsers and web servers,

but was seldom used due to the belief that a complex PKI structure for clients would be needed.

For SSO, this solution has been discussed within the SAML security group of OASIS [6], and its security has been proven in [11]. The main disadvantage of this approach is that the browser becomes uniquely recognizable, which is in conflict with privacy considerations.

Recognition of the Server: TLS-server-end-point binding. As described in [2], the end-point binding enables to make security decisions on the basis of the public key of the TLS server certificate: The client recognizes the server. Therefore, the browser may remain anonymous in most situations, and only discloses its identity to a well-known server.

The security of this solution is hard to investigate, because two protocols on different levels of the protocol hierarchy are combined: TLS and HTTP.

Recognition of the TLS Session: tls-unique binding. Here client and server both recognize a certain TLS session. A cryptographic value (the first `Finished` message sent during the TLS handshake protocol) which captures the essence of this section is used in subsequent, higher layer message exchanges. Client and server can thus verify if they are using the same session or different sessions (in case of a man-in-the-middle attack on the HTTP connection).

To give an illustration of the concept: We can use the first of the two `Finished` messages from the TLS handshake as an intermediate value and then we can combine this value with password based authentication in the following way: The password is not transmitted in the clear over TLS, but is instead hashed together with this first `Finished` message. The server will compare this value with the hash of the stored password together with the first `Finished` message of the server's TLS handshake protocol. If there is a man-in-the-middle attack on the HTTP connection, the two values will not match and therefore access will be denied.

Jager *et al.* showed at ASIACRYPT '10 [17], how such a binding can be used in a generic way. To stick to our earlier example, TLS could be used only as a key exchange protocol and the `Finished` message(s) (or the whole message transcript) could be adopted in a subsequent authentication protocol. Kohlar *et al.* have shown at ARES '10 [21], how such a binding can be achieved for SAML-Authentication.

Origin-Problems. At W2SP 2008 Jackson and Barth published some weaknesses regarding the scripting-policy used in browsers and the locked same-origin policies [16], which we take as basis to construct our new Single Sign-On scheme. The underlying problem is, that the SLSOP only protects web objects from being accessed by other web objects originating from different (possibly malicious) domains. Specially prepared sub-resources in the same domain (e.g. script-objects) can still be used to compromise the origin of protected objects in sub-origins, i.e. restricted cookies can be accessed by injecting a script into a document with the appropriate path.

However, this attack is not applicable in full to our work as we introduce a novel type of cookies which is not accessible by standard means, explicitely to cope with such attacks. Regarding the HTTP Redirect and HTTP POST bindings and without any additional security measures, our solution is still be vulnerable to the mentioned attack. The authors suggest some extensions to browser policies, one of which is very close to the ideas in our work, namely the usage of "YURLs" [1]. A YURL is basically a URL which includes public key information in front of the hostname. In our case, the URL could be extended with the public key (or the fingerprint of the corresponding certificate) of the service provider.

Remark: Another way to easily deny access to cookies from scripts executed at the client would be to use the HTTP-Only flag[14]. This flag tells the browser that this particular cookie should only be accessed by the corresponding server. This does not however protect the cookie from being sent to a malicious website impersonating the honest server.

(Login) CSRF-attacks. At CCS '08 Barth *et al.* [4] presented a new type of Cross-Site Request Forgery (CSRF) attacks, called *login CSRF* in which the attacker forges a cross-site request to the honest site's login form, resulting in logging the victim into this web site as the attacker. As proposed in the paper, security can either be preserved by evaluating the referrer (in those cases, where such a referrer is transmitted) or by adding an additional field to the ticket (cookie), containing the issuing server and/or its public key. Though the existence of such solutions, the authors also described shortcomings for all three.

3. SECURE SSO WITH SLSOP

In this chapter, we describe a secure SSO protocol based on the tls-server-end-point binding. In the following we will substitute the term "tls-server-end-point binding" with the term SLSOP. Though the terms are not identical in meaning, the basic idea of "pinning" information (in our case the identity of the service provider) to a certain certificate/public key is the same for both. We propose three cross-domain data transport mechanisms for our solution, which we have implemented in our Firefox-based prototype: HTML forms (POST binding), HTTP redirect (Redirect binding) and a new type of cross-domain cookies that we call SLSOP cookies and that we describe in detail in section 3.3.

3.1 SSO with SLSOP-Cookies

Figure 1 gives a fairly detailed overview of the actions that should take place in a ideal browser implementation. (We will see later that there are some limitations with current browsers.)

We assume that an initial authentication of the user has taken place using the current browser, such that an SLSOP authentication cookie for the Identity Provider (IP) is already stored within the browser. There is no SLSOP authentication cookie for the relying party (RP), thus SSO must be performed. All connections to IP and RP are established over TLS, thus a TLS server certificate is always available. For sake of simplicity we assume that there is only a single server with a single certificate for IP and RP, resp. However, our solution works even if IP and RP are implemented as server farms, with different certificates for each server.

During the first connection to the RP, the browser B discovers that no SLSOP authentication cookie is stored for the public key presented by RP, thus resulting in an authentication request by the RP.

After the TLS handshake with IP, the browser B retrieves

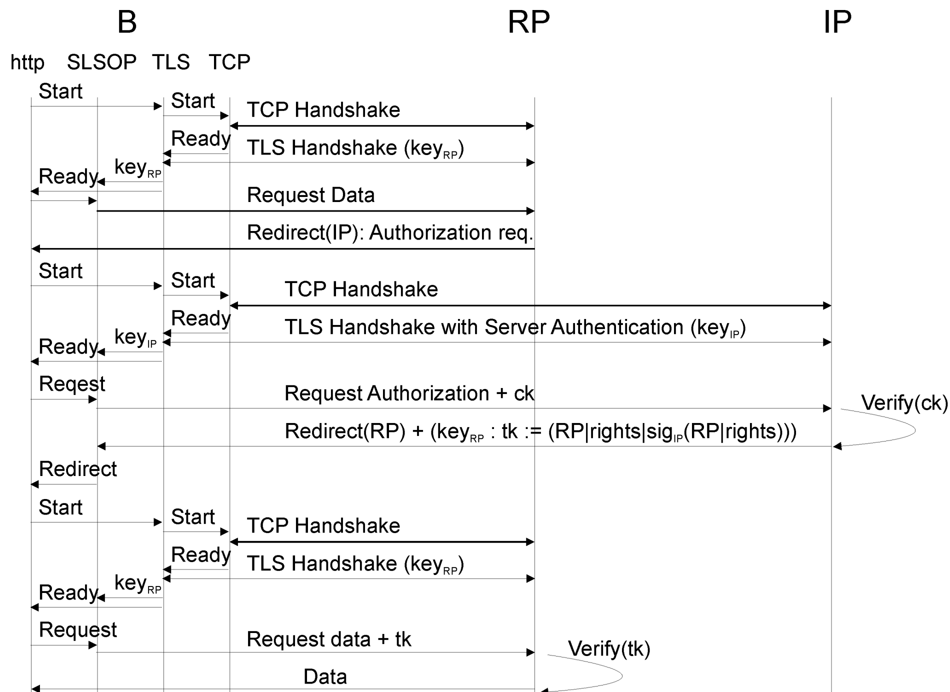

Figure 1: Full Single Sign-On Procedure based on the Strong Locked Same Origin Policy

the authentication cookie for key_{IP} from the local database, and adds a `X-SLSOPCookie:` header, which contains the value of the authentication cookie, to the HTTP request. The URL (not the public key) of RP is also sent to IP, in another HTTP header line.

After verifying the authentication cookie, IP retrieves the authorization parameters $rights$ of the user from its database, and constructs an authenticator token tk that contains the URL of RP, $rights$, and a signature on both values. Then IP looks up the (list of) public key(s) pk'_{RP} of the transmitted URL, and adds an `Set-SLSOP-Cookie:` header to the HTTP response, which contains two parameters: First the public key(s) (corresponding to the URL of RP) under which the token shall be stored, and second the token itself.

After receiving the answer, the browser first stores the token tk under the key pk'_{RP}, and then performs the HTTP `Redirect` to RP. (Remark: Here, an attack on DNS and/or PKI may result in a TLS connection with the adversary, rather than with RP.) After completion of the TLS handshake with server key pk_{RP}, the browser searches his local database for an entry with key pk_{RP}. If there is an entry, the value will be sent to RP in an `X-SLSOPCookie:` header.

Remark: Please note that the authentication token tk destined for RP will only be sent if $pk'_{RP} = pk_{RP}$. It is also important to notice, that the choice of where this token will be stored has a severe security impact. For our SLSOP cookies we explicitely chose to use an external database, that is not accessible via the browser's DOM. Considering the vast number of known attacks, that enable an adversary to gain access (read/write/alter) to information stored in the DOM without requiring him to impersonate a target website, we think it would be unwise to store our security token in this insecure memory area. However, maintaining the current status quo and allowing for such cookies to be stored in the DOM would still provide a higher level of security than before and lower the difficulty of implementations.

3.2 HTTP Cross-Domain Communication

In this section we shortly describe the two HTTP Cross-Domain Bindings we chose to implement. One may notice, that these two bindings provide weaker security guarantees than our SLSOP-Cookie solution. The reason why we decided to implement these two bindings is that these two bindings are used in many web sites for the transport of information and we can therefore provide a sort of "backwards compatibility". We stress however, that maximum security can only be achieved by adapting the mechanism presented in Section 3.3.

3.2.1 HTTP Redirect Binding

To secure the data exchanged via HTTP-Redirect, the origin server can append the public key(s) of the target server to the URL enclosed with ** next to the `Location:` Header field:

`Location: https:://URL/**`PK_1`,...,`PK_N`**`

The redirect will only be followed, if the target server is able to present one of the supplied public keys during the successful execution of the SSL Handshake. After successful verification, the User Agent (UA) extracts the appended public key(s) and follows the redirect to the intended URL. To prevent any harm, the user should not be able to enforce the execution of the redirect. An exemplary code snippet of how this binding can be realized is shown in Listing 8.

3.2.2 HTTP-POST Binding

To ensure the security of POST-data exchanged between two hosts the issuing host can request the UA to retrieve the

public key of the target host by invoking a certain JavaScript function incorporated in the UA. The function takes the URL of the target host as its argument and returns the public key in string representation:

```
String getPublicKey (
  [String:   URL of target host]
);
```

Once the function gets called, the UA initiates a SSL transport to the given URL. If the SSL transport could successfully be established, the public key of the target host is returned, otherwise the empty string is returned. The host serving the form then checks, if the returned value matches one of the intended public keys from its list. While this has the advantage to enable the server to assist the user if any error occurs, it has (at least in our prototype implementation) the drawback of being prone to a dynamic pharming attack. This could be mitigated by reusing the same socket for retrieving the cert and sending the POST data.

Listings 6 and 5 show how the public key can be extracted from inside the DOM if the token was transmitted via a POST Request.

3.3 Structure of the SLSOP Cookie

To be able to set persistent, identity bound authentication tokens, we introduce a new cookie-like mechanism. In contrast to ordinary cookies our SLSOP Cookie is capable of being securely shared across domain-boundaries. As well as for the HTTP-POST and HTTP-Redirect bindings the data included in a SLSOP-Cookie is also bound to public keys.

The syntax for the SLSOP-Set-Cookie response header (according to [23]) is the following:

Listing 1: SLSOP-Cookie Response Header

```
slsop-setcookie = "SLSOP-Set-Cookie:" slsopcookies
slsopcookies    = 1#slsopcookie
slsopcookie     = NAME "=" VALUE
                  *(";" slsopcookie-av)
NAME            = attr
VALUE           = value
slsopcookie-av  = "Receivers" "=" value
                | "Domain"    "=" value
                | "Max-Age"   "=" value
                | "Path"      "=" value
```

- Receivers: A comma separated list of Public Keys, which are entitled to receive the SLSOP-Cookie.
- Domain: The domain, to which the SLSOP-Cookie should be returned (optional).
- Max-Age: The time period, until which the cookie will be stored (optional).
- Path: The path of a URI to which the SLSOP-Cookie should be sent (optional).

The header syntax for returning a SLSOP-Cookie within an HTTP Request is:

Listing 2: SLSOP-Cookie Request Header

```
slsopcookie  = "SLSOPCookie:"
               1*((";" | ",") cookie-value)
slsopcookies = 1#slsopcookie
slsopcookie  = NAME "=" VALUE [";" path]
               [";" domain]
NAME         = attr
VALUE        = value
path         = "$Path"   "=" value
domain       = "$Domain" "=" value
```

The following rules apply to choosing applicable cookie-values from among all the SLSOP-Cookies the UA possesses:

- **Public Key Selection:** The public key presented during the SSL handshake between the UA and the target server must match one of the public keys supplied over the receiver field.
- **Domain Selection:** The target server's fully-qualified host name must domain-match the Domain attribute of the XSLSOP-Cookie. This is primarily meant to limit the scope of the SLSOP-cookie and especially to cope with wildcard certificates (see [27]).
- **Path Selection:** The Path attribute of the cookie must match a prefix of the request-URI.
- **Expire Selection:** XSLSOP-Cookies that have expired should have been rejected and thus are not send to the server.

4. DEMONSTRATOR

Description

To show the applicability of the above concepts, we implemented a browser extension for the Mozilla Firefox 3.6 that realizes SSO protocol for SLSOP-cookies as well as for the HTTP Redirect and POST binding. In this section we will give a brief description of the functionality and inner structure of our extension.

To be able to interact with HTTP-requests and responses the plugin first registers itself as a so-called *observer* for the following events:

- http-on-modify-request
- http-on-examine-response

These two events are fairly standard and provide in combination with an nsIHttpChannel means to manipulate in- and outgoing http-messages. We additionally created a third event called **getPublicKeyEvent** that triggers our extension when a public key is requested. We realized it as a custom DOM event, to enable safe message exchange between unprivileged (DOM) and privileged (Chrome) javascript. It can be raised by the javascript residing in the DOM to retrieve the public key of the desired URL.

Once an http request is started, we first check the scheme of the URL. If we detect an oncoming SSL/TLS handshake in the target URL (by looking for "https"), we interrupt and pause the request. At this point we expected to be able to retrieve the status information of the underlying SSL transport and manipulate the request according to our SLSOP. But the internal SSL-processing of firefox (see also [28]) proved us wrong:

1. The HTTP request is about to be queued for sending (This is where our http-on-modify-request event gets raised).

2. The socket for sending the HTTP-request is initiated (SSL information becomes available).

3. The HTTP requests has been sent.

Thus, when our `http-on-modify`-event gets raised and this is the first request which initiates the socket, we do not have the required SSL-status information at hand yet, which we need to evaluate the request appropriately. We work around this issue by establishing a SSL socket to the URL the actual request is intended to reach:

Listing 3: Dummy Socket

```
fetchCert : function(aHost) {
  var transportService = Cc["@mozilla.org/network/
      socket-transport-service;1"].getService(Ci.
      nsISocketTransportService);
  var socket = transportService.createTransport(["
      ssl"],1,aHost,443,null);
  //attach input and outputstreams, needed to
      initialize the socket
  var inputStream  = socket.openInputStream(Ci.
      nsITransport.OPEN_BLOCKING,0,0);
  var outputStream = socket.openOutputStream(Ci.
      nsITransport.OPEN_BLOCKING,0,0);
  // give it some time to finish socket connect
      and ssl handshake
  // will break after timeout or successful
      retrieval of ssl-cert
  var date = new Date();
  var curDate = null;
  do {
    curDate = new Date();
    if(socket.securityInfo instanceof Ci.
        nsISSLStatusProvider) {
      var secInfo = socket.securityInfo.QI(Ci.
          nsISSLStatusProvider).SSLStatus;
      if(secInfo) {
        var cert = socket.securityInfo.QI(Ci.
            nsISSLStatusProvider).SSLStatus.QI(Ci.
            nsISSLStatus).serverCert;
        break;
      }
    }
  }
  while(curDate-date < timeout);
}
```

Once this socket is established and the handshake was completed successfully, we return the certificate and extract the included Public Key:

Listing 4: Public Key Extraction

```
extractPubKey : function (aCert) {
  // Get the public Key out of the ASN.1 structure
  var asn1Tree = Cc["@mozilla.org/security/
      nsASN1Tree;1"].createInterface(SLSOP_Ci.
      nsIASN1Tree);
  asn1Tree.loadASN1Structure(aCert.ASN1Structure);
  //Get Public Key field and normalize output
  var pubkey = asn1Tree.getDisplayData(12);
  pubkey = pubkey.slice((pubkey.indexOf(':')+2),
      pubkey.lastIndexOf("\n\n"));
  pubkey=pubkey.replace(/\s/g,"");
  return pubkey;
},
```

[Remark: We are aware that this workaround enables an attacker to perform active MITM attacks during the second connection stage (after fetching the certificate). But this is only a limitiation of our extension at the time being, an actual correct implementation would not be prone to such an attack.]

Having obtained this certificate and extracted the public key, we now possess the public key we requested.

Because of the in many ways different format of our SLSOP cookies in comparison to common browser cookies, we manage a separate SQLite database for our plugin in which we store the SLSOP cookies.

Upon encountering an SSL-secured request and retrieving the corresponding public key of the server we want to access, we search in this SQLite database for an applicable cookie.

In a normal case we gained such a cookie in some previous connection to a trusted Identity Provider and so we should be able to find the desired cookie. If we do not find a cookie for the server, which could be the result of a MITM attack (wrong public key) or a missing registration and/or cookie request at the IP, we abort the connection or simply do not attach any cookie. If we found the cookie, we again do the preliminary checks as described in section

If all checks passed up to this point we append the cookie using the additional header field `X-SLSOPCookie: ...` to the original http request:

Listing 5: Public key retrieval by the browser extension

```
onModifyRequest : function(aHttpChannel) {
  //We only care about SSL-secured responses
  if(aHttpChannel.URI.schemeIs("https")) {
    var cert = this.fetchCert(aHttpChannel.URI.
        host);
    if(cert == null) {return;}    //silent fail...
    var pubKey = this.extractPubKey(cert);
    //Get database entries for supplied public key
        and URI.
    var value = cookieDB.getCookie(pubKey,
        aHttpChannel.URI);
    if(value == null) {return;} //no cookies found
        , so do nothing
    //Cookie found, append it
    aHttpChannel.setRequestHeader("X-SLSOPCookie",
        pubKey , false);
  }
},
```

If the IP wants to transmit the token via a POST-request, it can raise the **getPublicKeyEvent** from inside the DOM to obtain the Public Key. In short, the following happens once this event gets triggered: a dummy socket as described earlier is established and the received public key will be appended as an attribute to the DOM at the position of which the event was triggered:

Listing 6: DOM-Access

```
<script>
var element = document.createElement("
    SLSOPCookiesDataElement");
element.setAttribute("id", "MyPubkeyRetriever");
element.setAttribute("getPublicKey", "https://xxx.
    xxx.xxx/");
document.documentElement.appendChild(element);
var evt = document.createEvent("Events");
evt.initEvent("getPublicKeyEvent", true, false);
    // is triggered, when public key is requested
element.dispatchEvent(evt); // branch to the
    extension

var PK = document.getElementById("
    MyPubkeyRetriever").getAttribute("PublicKey");
     // get the public key

if((PK==PK1)||(PK==PK2)||....) //search for
    retrived public key in cookie database
 {form.submit()}
```

```
else
  (alert("Wrong PK");) // or optionally silent
      fail
</script>
```

The following listing shows the code that enables our extension to retrieve the public key. We basically fetch the public key using the above mentioned dummy socket and append it to the DOM.

Listing 7: Public key retrieval by the browser extension

```
getPublicKeyEvent : function (evt) {
  var url = makeURI(evt.target.getAttribute("
      GetPublicKey"), null, null); // fetch url
      and check for validity
  if (url != null) {
    var pubKey = this.retrievePubKey(url.spec); //
        fetch public key using the dummy socket
    evt.target.setAttribute("PublicKey", pubKey);
        // append the public key to the DOM of the
        source document
  }
```

To be able to detect redirects and handle them accordingly we have to check the status code of every incoming HTTP responses. If we detect a SLSOP secured redirect we initiate our dummy socket and obtain the corresponding public key of the target. Then we check if the retrieved public key matches anyone from the appended one. If that is the case we follow the redirect, otherwise we cancel the transport:

Listing 8: Redirect Binding

```
//Get statuscode and see if it is a redirect
var statusCode = aHttpChannel.responseStatus;
var dest = aHttpChannel.getResponseHeader("
    Location");
if ((statusCode == 302) && (dest != null)) {
//create an URI-object
var destUrl = this.makeURI(dest);
//see, if there is a pk appended, if so get dest
    url and pk back
var url_and_pubkey = this.extractPubKeyFromUrl(
    destUrl.spec);
//if nothing is there, we don't care...
if (url_and_pubkey == null)
  return;
//get the cert
var pubKeyDest = this.fetchCert(destUrl.host);
//if the retrieved pubkey does not match the
    appended one, do NOT
//follow the redirect
if (url_and_pubkey[1] != pubKeyDest) {
  aHttpChannel.cancel(null);
}
//rewrite location header, without the appended
    pks
aHttpChannel.setResponseHeader("Location",
    url_and_pubkey[0], false);
}
```

5. SECURITY ANALYSIS

Unfortunately the security of the TLS protocol has not yet been proven secure in the standard model. To prove the security of our scheme, we would therefore first have to prove the security of TLS, which is beyond the scope of this work. Nevertheless, we would like to give a practical security analysis regarding the threat classes defined above: As we solely use TLS to secure the data-transport and to ensure the authenticity of the communication peers, we rely on a foundation that is considered to be secure. Additionally, our concept is based on the server's public key presented during the TLS handshake and not on a complex and error-prone PKI system. We therefore elude the problems that may arise operating and relying on such an infrastructure.

For illustration purposes we split the SSO-Protocol into several phases as illustrated in Figure 2

Passive Attackers.

As described before, we assume that all data is transmitted via TLS. There has been no evidence that TLS is susceptible to passive attacks and research carried out so far suggests that it is very unlikely that we will see any successful passive attacks on TLS in the near future.

As long as the ciphers both peers agreed on during the TLS handshake are considered secure, passive attackers have no other possibility than brute forcing the derived session key in order to reveal any information. With present technology this is not possible within practical timeframe and we therefore exclude passive attackers as a present risk.

Active Attackers.

1. **Initial Resource Request**: Here an active attacker could reroute the initial resource request to a server under his control, by manipulating the DNS-mapping or by exploiting deficiencies in the underlying (routing) protocols. However, as no authentication ticket is present yet, there is nothing an adversary could steal.

2. **Sign-On Phase**: In this phase an active attacker could strike in and reroute the traffic to its own server trying to scam the user into surrendering his credentials. Using these credentials the attacker could then impersonate the legitimate user. The feasibility of this attack is strongly related to the authentication system used. If a password based scheme is employed and the user is not able to adequately identify the server, such an attack is within the bounds of possibility.

   We therefore propose an out of band initialization, which issues a *long-term* SLSOP-authentication token in the form of a SLSOP-Cookie. This token should be bound to the public key of the IP and is used in subsequent request to authenticate the user. As a result, this token will only be transmitted to a server possessing the corresponding private key and thus rendering network attacks in this phase useless.

3. **Ticket Exchange Phase**: The authentication-ticket issued to the client by the IP is bound to the public key of the target SP. Hence, this ticket is only sent to the server, which is able to perform the TLS handshake with the corresponding private key. Therefore an active attacker manipulating the communication flow in this stage, is not able to intercept this ticket.

4. **Application Data Transfer**: In this phase every authentication step has already taken place and the actual data of the requested application is transferred. An attacker against this phase could only try to decrypt the application data, which would again imply breaking the confidentiality of the TLS channel.

In the following we present a few attacks that may seem to compromise the security of our system independent of our

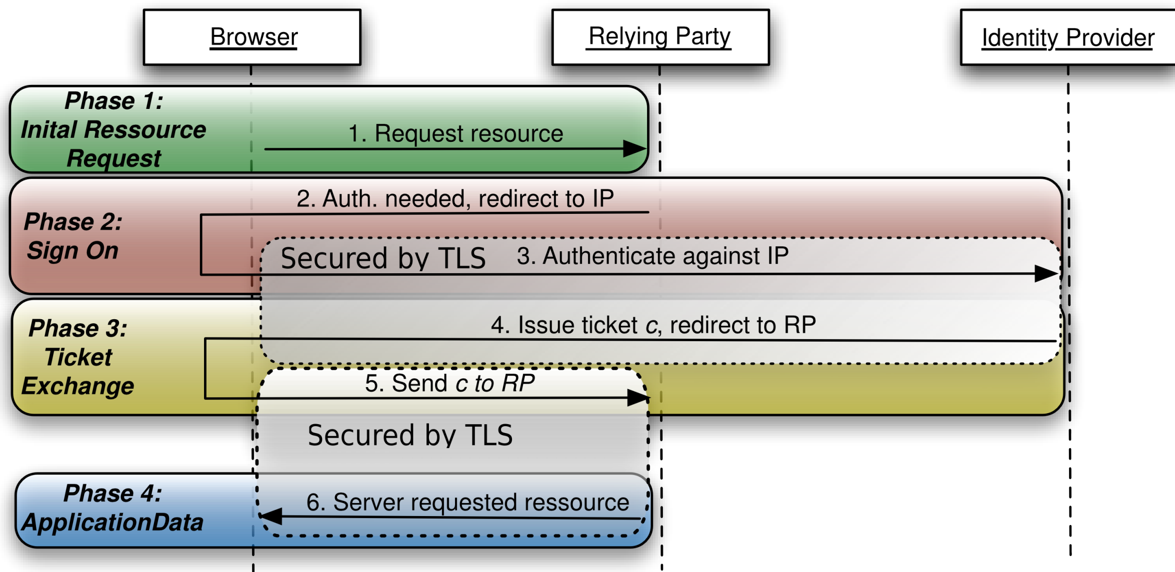

Figure 2: Phases of a typical Web-based SSO session

protocol phases and we provide evidence, that these attacks do not apply to our solution.

Hash Collisions for MD5. In 1996, Hans Dobbertin first showed that it is (theoretically) feasible to find collisions for the hash function MD5 [10]. At this time he already suggested to sustain from using MD5 as a compression function in security critical applications. In 2005 for the first time, it was practically possible to find collisions for PostScript documents [9] and X.509 certificates [25, 24]. Shortly after, the collision resistance MD5 was considered broken (admitted also by it's designer Ron Rivest).

At Eurocrypt 2007 Stevens *et al.* presented an attack that enables an adversary to find chosen-prefix collisions for MD5 [35]. This attack was improved in 2008 to find a way to to change a normal SSL certificate (issued by RapidSSL) into a working CA certificate using a cluster of Sony Playstations, that could be verified under their MD5 hashes [29]. These certificates could then be used to create other certificates that would appear to be legitimate and issued by RapidSSL. This attack was later again improved to be able to find short chosen-prefix collision [36].

All those attacks do not apply to our protocol, as a successfull attack would require MD5 to be vulnerable against second preimage attacks. However, although MD5 is (for good reasons) not considered collision resistant any more, there are no feasible second preimage attacks against the compression function.

Cross-Site Scripting Attacks. As we still use some kind of authentication token in our solution, it may seem that our proposed protocol may be vulnerable to theft of this token using Cross-Site Scripting attacks. We cope with this attack by storing our token in a seperate database not accessible by the browser's DOM. If a seperate database is not manageable we can still provide security by modifying the transport mechanisms used to transmit this token (e.g. always checking if the public key of the recipient is matching to the token).

6. CONCLUSION AND OUTLOOK

While our solution seems to enable a secure binding between service providers and their clients, some open problems still exist. Although easy to implement, our solution does need some minor adjustments on the server side, detailed in the following section. As mentioned in section 2, a session could be compromised by downloading malicicious applets or scripts. To cope with this threat, we need additional security checks, which are outlined in Section 6.2.

6.1 Deployability

To be compliant to the zero-footprint requirement of browsers, our XSLSOP scheme is solely based on technologies already incorporated in all major browsers. Only small changes have to be applied to the processing of HTTP requests over an SSL transport. In order to fulfill our requirements, the browser needs to recognize the public key learned during the establishment of the SSL transport and has to provide means to ensure that data, which is bound to a specific public key, will only be send over a corresponding SSL secured channel. To satisfy the conformance criteria for the HTTP-Post Binding, a new JavaScript function had to be implemented in the user agent. On the server side however, there is technically no work to be done. The option of setting custom header fields and constructing dynamic URLS is a standard function implemented in all common server-side scripting languages as for example PHP or ASP.NET. The only major issue we are facing here is the exchange and management of public key information between Relying Party and Identity Provider. But as we focus solely on SSO scenarios, where data exchange is performed on a regular basis, this issue can be considered neglectable.

6.2 Treatment of Mixed Content

To secure the subsequent communication between Service

Provider and Client against origin-contamination, as described by Barth *et al.* [4], we may further enhace our scheme to enforce, that all content has to be retrieved over SSL and is only accepted/executed, if the corresponding channel was established to a server possessing a certain, matching public key. This could be realized, for example, by enforcing a special response header after receiving the ticket (this has to be configured at the Service Provider), similar to Force-HTTPS as introduced in [15]. Another option would be to stricten transport security, e.g. to enforce a policy that lists all public keys of the server, that is entitled to serve content to the user and blacklisting all other keys.

7. REFERENCES

[1] Decentralized identification. `http://www.waterken.com/dev/YURL/`.

[2] J. Altman, N. Williams, and L. Zhu. Channel Bindings for TLS. RFC 5929 (Proposed Standard), July 2010.

[3] M. Backes, I. Cervesato, A. D. Jaggard, A. Scedrov, and J.-K. Tsay. Cryptographically sound security proofs for basic and public-key kerberos. Cryptology ePrint Archive, Report 2006/219, 2006. `http://eprint.iacr.org/`.

[4] A. Barth, C. Jackson, and J. C. Mitchell. Robust defenses for cross-site request forgery. In *CCS '08: Proceedings of the 15th ACM conference on Computer and communications security*, pages 75–88, New York, NY, USA, 2008. ACM.

[5] A. Boldyreva and V. Kumar. Provable-security analysis of authenticated encryption in kerberos. Cryptology ePrint Archive, Report 2007/234, 2007. `http://eprint.iacr.org/`.

[6] S. Cantor, J. Kemp, R. Philpott, and E. Maler. Assertions and Protocol for the OASIS Security Assertion Markup Language (SAML) V2.0. OASIS Standard, 15.03.2005, 2005. `http://docs.oasis-open.org/security/saml/v2.0/saml-core-2.0-os.pdf`.

[7] B. den Boer and A. Bosselaers. Collisions for the compression function of md5. In *EUROCRYPT '93: Workshop on the theory and application of cryptographic techniques on Advances in cryptology*, pages 293–304, Secaucus, NJ, USA, 1994. Springer-Verlag New York, Inc.

[8] R. Dhamija, J. Tygar, and M. Hearst. Why phishing works. In *Proceedings of the SIGCHI conference on Human Factors in computing systems*, pages 581–590. ACM, 2006. `http://graphics8.nytimes.com/images/blogs/freakonomics/pdf/Why_Phishing_Works-1.pdf`.

[9] Dobbertin. Postscript collisions for md5, 2005.

[10] H. Dobbertin. Cryptanalysis of MD5 Compress - presented at the Rumpsession of Eurocrypt '96, May 1996.

[11] S. Gajek, T. Jager, M. Manulis, and J. Schwenk. A browser-based kerberos authentication scheme. In *ESORICS '08: Proceedings of the 13th European Symposium on Research in Computer Security*, pages 115–129, Berlin, Heidelberg, 2008. Springer-Verlag.

[12] T. Groß. Security analysis of the SAML single sign-on browser/artifact profile. In *Annual Computer Security Applications Conference*. IEEE Computer Society, 2003.

[13] T. Groß and B. Pfitzmann. Saml artifact information flow revisited. Research Report RZ 3643 (99653), IBM Research, 2006. `http://www.zurich.ibm.com/security/publications/2006.html`.

[14] HttpOnly cookies. First implemented by Microsoft Internet Explorer developers for Internet Explorer 6 SP1, 2002.

[15] C. Jackson. Forcehttps: Protecting high-security web sites from network attacks. In *In Proceedings of the 17th International World Wide Web Conference*, 2008.

[16] C. Jackson and A. Barth. Beware of finer-grained origins. In *In Web 2.0 Security and Privacy (W2SP 2008)*, 2008.

[17] T. Jager, F. Kohlar, S. Schäge, and J. Schwenk. Generic compilers for authenticated key exchange. pages 232–249, 2010.

[18] D. Kaminski. Dns server+client cache poisoning, issues with ssl, breaking *forgot my password* systems, attacking autoupdaters and unhardened parsers, rerouting internal traffic; `http://www.doxpara.com/DMK_BO2K8.ppt`. -, 2008.

[19] D. Kaminsky. It's the end of the cache as we know it - black ops 2008. *Black Hat Briefings, Las Vegas, Nevada, USA*, July 2008.

[20] C. Karlof, U. Shankar, J. D. Tygar, and D. Wagner. Dynamic pharming attacks and locked same-origin policies for web browsers. In *CCS '07: Proceedings of the 14th ACM conference on Computer and communications security*, pages 58–71, New York, NY, USA, 2007. ACM.

[21] F. Kohlar, J. Schwenk, M. Jensen, and S. Gajek. Secure bindings of saml assertions to tls sessions. In *ARES*, pages 62–69, 2010.

[22] D. Kormann and A. Rubin. Risks of the passport single signon protocol. *Computer Networks*, 33(1–6):51–58, 2000.

[23] D. Kristol and L. Montulli. Http state management mechanism, Oct. 2000.

[24] A. Lenstra, X. Wang, and B. de Weger. Colliding x.509 certificates. Cryptology ePrint Archive, Report 2005/067, 2005. `http://eprint.iacr.org/`.

[25] A. K. Lenstra and B. de Weger. On the possibility of constructing meaningful hash collisions for public keys. pages 267–279, 2005.

[26] E. Maler, P. Mishra, and R. Philpott. Assertions and Protocol for the OASIS Security Assertion Markup Language (SAML) V1.1. OASIS Standard, 02.09.2003, 2003. `http://www.oasis-open.org/committees/download.php/3406/oasis-sstc-saml-core-1.1.pdf`.

[27] M. Marlinspike. More tricks for defeating ssl in practice. Blackhat DC, 2009. https://www.blackhat.com/presentations/bh-dc-09/Marlinspike/BlackHat-DC-09-Marlinspike-Defeating-SSL.pdf.

[28] C. Masone, K.-H. Baek, and S. Smith. Wske: Web server key enabled cookies. In S. Dietrich and R. Dhamija, editors, *Financial Cryptography*, volume

4886 of *Lecture Notes in Computer Science*, pages 294–306. Springer, 2007.

[29] D. Molnar, M. Stevens, A. Lenstra, B. de Weger, A. Sotirov, J. Appelbaum, and D. A. Osvik. MD5 considered harmful today - Creating a rogue CA Certificate. *25th Chaos Communication Congress, Berlin, Germany*, 2008.

[30] B. Pfitzmann and M. Waidner. Analysis of liberty single-signon with enabled clients. *IEEE Internet Computing*, 7(6):38–44, 2003.

[31] D. Recordon and D. Reed. Openid 2.0: a platform for user-centric identity management. In *DIM '06: Proceedings of the second ACM workshop on Digital identity management*, pages 11–16, New York, NY, USA, 2006. ACM.

[32] J. Schwenk, L. Liao, and S. Gajek. Stronger bindings for saml assertions and saml artifacts. In *Proceedings of the 5th ACM CCS Workshop on Secure Web Services (SWS'08)*, pages 11–20. ACM Press, 2008.

[33] M. Slemko. Microsoft passport to trouble, 2001. `http://alive.znep.com/~marcs/passport/page2.html`.

[34] M. Stevens, A. Lenstra, and B. de Weger. Chosen-prefix Collisions for MD5 and Applications. Submitted to Journal of Cryptology, June 2009. `https://documents.epfl.ch/users/l/le/lenstra/public/papers/lat.pdf`.

[35] M. Stevens, A. K. Lenstra, and B. de Weger. Chosen-prefix collisions for MD5 and colliding X.509 certificates for different identities. pages 1–22, 2007.

[36] M. Stevens, A. Sotirov, J. Appelbaum, A. K. Lenstra, D. Molnar, D. A. Osvik, and B. de Weger. Short chosen-prefix collisions for MD5 and the creation of a rogue CA certificate. pages 55–69, 2009.