Securing Web Services

Breno de Medeiros

Department of Computer Science
Florida State University
Most provably secure authentication/key-exchange protocols assume a symmetric setup (both parties have certificates or pre-shared keys).
Securing Web Services

Most provably secure authentication/key-exchange protocols assume a symmetric setup (both parties have certificates or pre-shared keys).

- This setting is inadequate in the case of web services, as browsers do not have client certificates.
Most provably secure authentication/key-exchange protocols assume a symmetric setup (both parties have certificates or pre-shared keys).

- This setting is inadequate in the case of web services, as browsers do not have client certificates.

- Even if certificates were available to clients, they would not authenticate the user, but the browser application. How would the user authenticate itself from a different machine?
Securing Web Services

Most provably secure authentication/key-exchange protocols assume a symmetric setup (both parties have certificates or pre-shared keys).

△ This setting is inadequate in the case of web services, as browsers do not have client certificates.

△ Even if certificates were available to clients, they would not authenticate the user, but the browser application. How would the user authenticate itself from a different machine?

△ Browsers execute web-service protocols in an oblivious fashion (they do not know which protocol they are executing, unlike say, an SSH client).
Securing Web Services

Most provably secure authentication/key-exchange protocols assume a symmetric setup (both parties have certificates or pre-shared keys).

△ This setting is inadequate in the case of web services, as browsers do not have client certificates.

△ Even if certificates were available to clients, they would not authenticate the user, but the browser application. How would the user authenticate itself from a different machine?

△ Browsers execute web-service protocols in an oblivious fashion (they do not know which protocol they are executing, unlike say, an SSH client).

△ These features indicate the need for a browser model and for the use of web service-specific security models.
Web Services-Federation Passive Requestor Interop

WSFPI protocol. Interrupted lines are multiple-step processes specified in the protocol.
Proof structure

The proof structure for WS is based on the reactive systems approach described by Pfitzmann and Waider, and applied by Backes-Pfitzmann-Waidner to the Needham-Schroeder-Lowe protocol.
Proof structure

The proof structure for WS is based on the reactive systems approach described by Pfitzmann and Waider, and applied by Backes-Pfitzmann-Waidner to the Needham-Schroeder-Lowe protocol.

First, an idealized model is created, where the real-world operations are substituted by a library of operations that are guaranteed to be secure. The security equivalence of real and ideal systems follows from standard arguments.
The proof structure for WS is based on the reactive systems approach described by Pfitzmann and Waider, and applied by Backes-Pfitzmann-Waidner to the Needham-Schroeder-Lowe protocol.

First, an idealized model is created, where the real-world operations are substituted by a library of operations that are guaranteed to be secure. The security equivalence of real and ideal systems follows from standard arguments.

Second, a proof of security for the ideal protocol version is provided. However, in the case of federated identity, security in the ideal world is not immediate, as the protocol is very complex.
Browser model, instantiated to a federated identity protocol

Figure 2: Key to the state diagrams

Figure 3: System architecture for browser-based protocols, here for identity federation protocols.
The browser leaks various types of information. For instance, information is often stored as cache cookies, cache entries, and history (and search history) entries.
Peculiarities of the browser model

- The browser leaks various types of information. For instance, information is often stored as cache cookies, cache entries, and history (and search history) entries.

- In the attack model, the adversary is able to request all the information stored in the cache and cookies. This is useful to model insecure environments, such as when the user browses from a public machine.
The browser leaks various types of information. For instance, information is often stored as cache cookies, cache entries, and history (and search history) entries.

In the attack model, the adversary is able to request all the information stored in the cache and cookies. This is useful to model insecure environments, such as when the user browses from a public machine.

The WSPFI protocol uses scripted POST forms in HTTP protocol responses to re-direct the browser to other sites. (For instance, the Identity Consumer $C$ redirects the user to an Identity Supplier $S$.) Browsers generate extra information flows to servers. The adversary is allowed to see these—which are often not-protected even if the forwarded channel is secure.
Server posting: \textit{POSTForm}(adr : URLHost, path : URLPath, query : \sum^*, \text{close} : \text{Bool}, \text{nocache} : \text{Bool}).

- Posts the abstract \textit{query} to URL address \textit{adr/path}. In HTTP1.1, if \textit{nocache} = \text{TRUE} then HTTP no-cache and no-store directives are explicitly set.
Server posting: \texttt{POSTForm}(adr : URLHost, path : URLPath, query : $\Sigma^*$, close : Bool, nocache : Bool).

- Posts the abstract \textit{query} to URL address \textit{adr/path}. In HTTP1.1, if \textit{nocache} = TRUE then HTTP no-cache and no-store directives are explicitly set.
- Browser establishes channel to \textit{adr} and sends \textit{path} and \textit{query} via POST message over the channel.
Server posting: \texttt{POSTForm}(adr : URLHost, path : URLPath, query : \Sigma^*, close : Bool, nocache : Bool).

- Posts the abstract \texttt{query} to URL address \texttt{adr/path}. In HTTP1.1, if \texttt{nocache} = TRUE then HTTP no-cache and no-store directives are explicitly set.
- Browser establishes channel to \texttt{adr} and sends \texttt{path} and \texttt{query} via POST message over the channel.
- Channel type is determined automatically by protocol name “http” or “https” in \texttt{adr}.
Server posting: POSTForm\((adr : \text{URLHost}, path : \text{URLPath}, query : \Sigma^*, close : \text{Bool}, nocache : \text{Bool})\).

- Posts the abstract \(query\) to URL address \(adr/path\). In HTTP1.1, if \(nocache = \text{TRUE}\) then HTTP no-cache and no-store directives are explicitly set.
- Browser establishes channel to \(adr\) and sends \(path\) and \(query\) via POST message over the channel.
- Channel type is determined automatically by protocol name “http” or “https” in \(adr\).

Client posting:
\[
\text{POST}(path : \text{URLPath}, query : \Sigma^*, login : \Sigma^*, info\_leak : (\Sigma^* \times \Sigma^*)^*)
\]
**POST forms**


- Posts the abstract `query` to URL address `adr/path`. In HTTP1.1, if `nocache = TRUE` then HTTP no-cache and no-store directives are explicitly set.
- Browser establishes channel to `adr` and sends `path` and `query` via POST message over the channel.
- Channel type is determined automatically by protocol name “http” or “https” in `adr`.

Client posting:
`POST(path : URLPath, query : Σ*, login : Σ*, info_leak : (Σ* × Σ*)*)`.

- If client posting follows a server `POSTForm`, then contents of `query` match. `path` is the resource to retrieve at the host.

- Posts the abstract query to URL address adr/path. In HTTP1.1, if nocache = TRUE then HTTP no-cache and no-store directives are explicitly set.
- Browser establishes channel to adr and sends path and query via POST message over the channel.
- Channel type is determined automatically by protocol name “http” or “https” in adr.

Client posting:
POST(path : URLPath, query : Σ*, login : Σ*, info_leak : (Σ* × Σ*)*).

- If client posting follows a server POSTForm, then contents of query match. path is the resource to retrieve at the host.
- login are the credentials for the (password-based) user authentication request, including account name.
**POST forms**

Server posting: \(\text{POSTForm}(adr : \text{URLHost}, path : \text{URLPath}, query : \Sigma^*, close : \text{Bool}, nocache : \text{Bool})\).

△ Posts the abstract \(query\) to URL address \(adr/path\). In HTTP1.1, if \(nocache = \text{TRUE}\) then HTTP no-cache and no-store directives are explicitly set.

△ Browser establishes channel to \(adr\) and sends \(path\) and \(query\) via POST message over the channel.

△ Channel type is determined automatically by protocol name “http” or “https” in \(adr\).

Client posting:
\(\text{POST}(path : \text{URLPath}, query : \Sigma^*, login : \Sigma^*, info\_leak : (\Sigma^* \times \Sigma^*)^*)\).

△ If client posting follows a server POSTForm, then contents of \(query\) match. \(path\) is the resource to retrieve at the host.

△ \(login\) are the credentials for the (password-based) user authentication request, including account name.

△ \(info\_leak\) models information flows generated by browsers to servers, for instance the addresses of referring sites. It is a list of name-value pairs.
Persistent variable \( \text{prev\_run} = (\text{ch\_type}, \text{adr}, \text{form}) \). Stores information about the preceding HTTP transaction.
Persistent variable $prev\_run = (ch\_type, adr, form)$. Stores information about the preceding HTTP transaction.

- $ch\_type$ keeps the channel type, $adr$ the address retrieved by that transaction.
Persistent variable $prev\_run = (ch\_type, adr, form)$. Stores information about the preceding HTTP transaction.

- $ch\_type$ keeps the channel type, $adr$ the address retrieved by that transaction.
- $form$ contains hidden value fields, including security tokens and credentials, that are used by WSPFI.
Persistent variable $prev\_run = (ch\_type, adr, form)$. Stores information about the preceding HTTP transaction.

- $ch\_type$ keeps the channel type, $adr$ the address retrieved by that transaction.
- $form$ contains hidden value fields, including security tokens and credentials, that are used by WSPFI.

*History* and *Cache* are other relevant persistent variables.
Persistent variable $\text{prev\_run} = (ch\_type, adr, form)$. Stores information about the preceding HTTP transaction.

- $ch\_type$ keeps the channel type, $adr$ the address retrieved by that transaction.
- $form$ contains hidden value fields, including security tokens and credentials, that are used by WSPFI.

*History* and *Cache* are other relevant persistent variables.

The value $\text{prev\_run.\_adr}$ is leaked to the server.
Persistent variable $prev\_run = (ch\_type, adr, form)$. Stores information about the preceding HTTP transaction.

- $ch\_type$ keeps the channel type, $adr$ the address retrieved by that transaction.
- $form$ contains hidden value fields, including security tokens and credentials, that are used by WSPFI.

*History* and *Cache* are other relevant persistent variables.

The value $prev\_run.\text{adr}$ is leaked to the server.

To prevent other leaks from the browser, servers issuing POSTForms must set the directive nostore.
Figure 4: Request handling phase of an HTTP transaction.
At the end of handling a POSTForm, the browser sends a request to itself. This will cause it to read the *prev_run* variable and submit its *prev_run.form* variable as the content of a POST command, to *prev_run.adr*.
At the end of handling a POSTForm, the browser sends a request to itself. This will cause it to read the `prev_run` variable and submit its `prev_run.form` variable as the content of a POST command, to `prev_run.adr`.

At this point, if a change of channel type is required (secure/insecure or vice-versa) the browser checks if notification of the user is needed.
Figure 5: Request initiation and local negotiation phase of $B$. 
Figure 6: Channel establishment phase of B.
The identity supplier $S$ can invoke any user authentication machine. There are available, provably secure password-based user authentication within browser channel models.
Identity Supplier specification

The identity supplier $S$ can invoke any *user authentication machine*. There are available, provably secure password-based user authentication within browser channel models.

Clearly, $S$ must maintain a database of user identities and login information.
Identity Supplier specification

The identity supplier $S$ can invoke any user authentication machine. There are available, provably secure password-based user authentication within browser channel models.

Clearly, $S$ must maintain a database of user identities and login information.

In addition, $S$ maintains a database of known identity consumers. It must contain the URI where each consumer $C$ accepts security tokens. Also, each identity consumer is associated with a list of attribute names which $S$ should provide to $C$. 
The identity supplier $S$ can invoke any user authentication machine. There are available, provably secure password-based user authentication within browser channel models.

Clearly, $S$ must maintain a database of user identities and login information.

In addition, $S$ maintains a database of known identity consumers. It must contain the URI where each consumer $C$ accepts security tokens. Also, each identity consumer is associated with a list of attribute names which $S$ should provide to $C$.

$S$ is known by identity $sid_S$ for the purpose of establishing secure channels, and as $name_S$ as the signer of security tokens.
Finite state machine: Identity Supplier

Secure_channel_accepted

Failure: Not identity supplier URI

REQUESTED_UAuth

Failure: connection

UAuth_success

Failure: unauthenticated

Failure: POST addr insecure

[ctype(padr) = secure] //
channel_in_s(send, cid, POSTForm(host(padr), path(padr), res, TRUE, TRUE));
channel_in_s(!)

Request_received

[wa = wsignin1.0 ∧
ec := MetaC_s(URI = wtrealm) ≠
ε ∧ (wtrealm = ⊥ ∨ wtrealm ⊆
wtrealm)] // uauth_in_s(start, cid), uauth_in_s(!)

uauth_out_s?(done, cid, idu)[idu ≠ ε] //
atu ← eval(DB, idu, ec, att_n);
wresult ← sign(name_s,
genc_token(URIS, wtrealm, idu, att ));
res ← gen_res(wsignin1.0, wresult, wctx)

Success: POSTForm sent

[ctype(padr) ≠ secure] //
padr := wtrealm;

[ctype(padr) = secure] //
cchannel_in_s(send, cid, POSTForm(host(padr), path(padr), res, TRUE, TRUE));
channel_in_s(!)

[ctype(padr) = secure] //
cchannel_in_s(send, cid, POSTForm(host(padr), path(padr), res, TRUE, TRUE));
channel_in_s(!)
Each identity consumer maintains knowledge of $name_S$ and URIS, the resource locator for its identity supplier $S$’s service.
Identity Consumer specification

- Each identity consumer maintains knowledge of $name_S$ and URIS, the resource locator for its identity supplier $S$’s service.

- It has two phases. In the first phase, it simply sends a redirect instruction to the user’s browser to the identity supplier.
Each identity consumer maintains knowledge of $name_S$ and URIS, the resource locator for its identity supplier $S$’s service.

It has two phases. In the first phase, it simply sends a redirect instruction to the user’s browser to the identity supplier.

The second phase consists of parsing a response from $S$ and validating it. It checks the signature on the token, verifies that it names the identity supplier $S$, the consumer $C$, and that it contains some non-empty user identity $id_u$. If so, the token is accepted.
**Finite state machine: Identity Consumer**

![Finite state machine diagram]

**Figure 8:** Main part of state machine of the identity consumer C.
Theorem (Authenticity of WSFPI): Let a user $U$, an identity consumer $C$, its identity supplier $S$ and a channel machine $\text{secchan}$ fulfill the WSFPI setup assumptions. Let these machines and the user’s browser $B$ be correct, and let $id_U$ be defined as in Definition 2. Then if $C$ makes an output $(\text{accepted}, cid, id_U, att)$ at $\text{ssou\textunderscore out}_C$, there exists a secure channel instance $\text{SCh}_{bc}$ in $\text{secchan}$ with $\text{SCh}_{bc}.cid = cid \land \text{SCh}_{bc}.state = \text{established} \land \text{SCh}_{bc}.Partner = B, C$, unless an adversary can guess $\text{login}_{U,S}$ based on a priori knowledge of its distribution, its length, and the results of previous guessing attempts, which each exclude one potential value.