Cryptographic protocols (II)

Myrto Arapinis
School of Informatics
University of Edinburgh

March 17, 2016

Credit card payment protocol

Credit card payment

Credit card payment

- Is it a real card?

Credit card payment

- Is it a real card?
- Is the pin protected?

Behavior in the usual case

1. The waiter introduces the credit card

Behavior in the usual case

1. The waiter introduces the credit card
2. The waiter enters the amount m of the transaction

Behavior in the usual case

1. The waiter introduces the credit card
2. The waiter enters the amount m of the transaction
3. The terminal authenticates the card

Behavior in the usual case

1. The waiter introduces the credit card
2. The waiter enters the amount m of the transaction
3. The terminal authenticates the card
4. The costumer enters his secret pin

Behavior in the usual case

1. The waiter introduces the credit card
2. The waiter enters the amount m of the transaction
3. The terminal authenticates the card
4. The costumer enters his secret pin
   If the amount m is greater than 100 euros (and in only 20% of the cases)
   4.1 The terminal asks for authentication of the card
   4.2 The bank provides authentication

More details

4 actors: Bank, Customer, Card, and Terminal

Bank owns:

- a secret signing key sk_B
- a public verification key pk_B
- a secret symmetric encryption key per card K_{CB}

Card owns:

- Data: last name, first name, card's number, expiration date
- Signature's value $VS = \{\mathsf{hash}(Data)\}_{\mathsf{sk}_B}$
- a secret symmetric encryption shared with the bank K_{CB}

Terminal owns:

- the public verification key pk_B

Credit card payment protocol (in short)

The terminal reads the card:

$$1. \quad Ca \quad \rightarrow \quad T \quad : \quad Data, \{\mathsf{hash}(Data)\}_{\mathsf{sk}_B}$$

Credit card payment protocol (in short)

The terminal reads the card:

$$1. \quad Ca \quad \rightarrow \quad T \quad : \quad Data, \{\mathrm{hash}(Data)\}_{\mathrm{sk}_B}$$

The terminal asks for the secret pin:

$$\begin{array}{lllll} 2. & T & \rightarrow & Cu & : \ \text{pin?} \\ 3. & Cu & \rightarrow & Ca & : \ 1234 \\ 4. & Ca & \rightarrow & T & : \ \text{ok} \end{array}$$

Credit card payment protocol (in short)

The terminal reads the card:

$$1.\quad Ca \rightarrow T : Data, \{\mathrm{hash}(Data)\}_{\mathrm{sk}_B}$$

The terminal asks for the secret pin:

$$\begin{array}{lllll} 2. & T & \rightarrow & Cu & : \text{pin?} \\ 3. & Cu & \rightarrow & Ca & : 1234 \\ 4. & Ca & \rightarrow & T & : \text{ok} \end{array}$$

The terminal calls the bank

$$\begin{array}{lllll} 5. & T & \rightarrow & B & : \text{auth?} \\ 6. & B & \rightarrow & T & : N_B \\ 7. & T & \rightarrow & Ca & : N_B \\ 8. & Ca & \rightarrow & T & : \{N_B\}_{K_{Cb}} \\ 9. & T & \rightarrow & B & : \{N_B\}_{K_{Cb}} \\ 10. & B & \rightarrow & T & : \text{ok} \end{array}$$

Some flaws

The security was initially ensured by:

- the cards were difficult to reproduce
- the protocol (!) and keys were secret

Some flaws

The security was initially ensured by:

- the cards were difficult to reproduce
- the protocol (!) and keys were secret

But:

- cryptographic flaw: 320-bit keys can be broken (1988),
- logical flaw: no link between the secret code and the authentication of the card,
- fake cards can be built.

Some flaws

The security was initially ensured by:

- the cards were difficult to reproduce
- the protocol (!) and keys were secret

But:

- cryptographic flaw: 320-bit keys can be broken (1988),
- logical flaw: no link between the secret code and the authentication of the card,
- fake cards can be built.

$\Rightarrow$ "YesCard" built by Serge Humpich (France, 1998)

How does the "YesCard" work?

Logical flaw

$$
\begin{array}{lllll}
1. & Ca & \rightarrow & T & : & Data, \{\mathrm{hash}(Data)\}_{\mathrm{sk}_B} \\
2. & T & \rightarrow & Cu & : & \mathrm{pin?} \\
3. & Cu & \rightarrow & Ca & : & 1234 \\
4. & Ca & \rightarrow & T & : & \mathrm{ok}
\end{array}
$$

How does the "YesCard" work?

Logical flaw

$$
\begin{array}{lllll}
1. & Ca & \rightarrow & T & : & Data, \{\mathsf{hash}(Data)\}_{\mathsf{sk}_B} \\
2. & T & \rightarrow & Cu & : & \mathsf{pin?} \\
3. & Cu' & \rightarrow & Ca' & : & 5678 \\
4. & Ca' & \rightarrow & T & : & \mathsf{ok}
\end{array}
$$

How does the "YesCard" work?

Logical flaw

1. $Ca \rightarrow T : Data, \{\text{hash}(Data)\}_{\text{sk}_B}$
2. $T \rightarrow Cu : \text{pin?}$
3. $Cu' \rightarrow Ca' : 5678$
4. $Ca' \rightarrow T : \text{ok}$

There is always someone to debit

How does the “YesCard” work?

Logical flaw

1. $Ca \rightarrow T : Data, \{\mathrm{hash}(Data)\}_{\mathrm{sk}_B}$
2. $T \rightarrow Cu : \text{pin?}$
3. $Cu' \rightarrow Ca' : 5678$
4. $Ca' \rightarrow T : \text{ok}$

There is always someone to debit
$\longrightarrow$ creation of a fake card

How does the "YesCard" work?

Logical flaw

$$
\begin{array}{llcll}
1. & Ca & \rightarrow & T & : \; Data, \{\mathrm{hash}(Data)\}_{\mathsf{sk}_B} \\
2. & T & \rightarrow & Cu & : \; \mathrm{pin?} \\
3. & Cu' & \rightarrow & Ca' & : \; 5678 \\
4. & Ca' & \rightarrow & T & : \; \mathrm{ok}
\end{array}
$$

There is always someone to debit

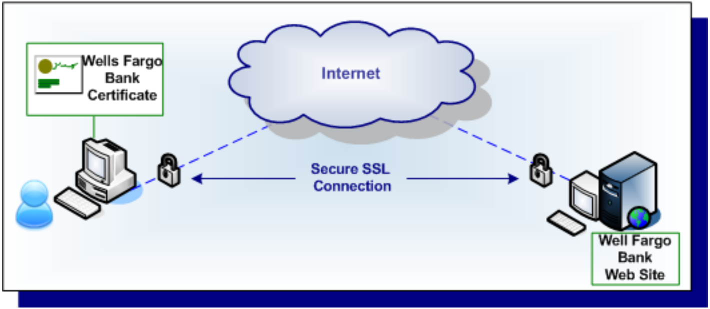
$\longrightarrow$ creation of a fake card

$$
\begin{array}{llcll}
1. & Ca' & \rightarrow & T & : \; XXXX, \{\mathrm{hash}(XXXX)\}_{\mathsf{sk}_B} \\
2. & T & \rightarrow & Cu' & : \; \mathrm{pin?} \\
3. & Cu' & \rightarrow & Ca' & : \; 0000 \\
4. & Ca' & \rightarrow & T & : \; \mathrm{ok}
\end{array}
$$

The SSL/TLS protocol

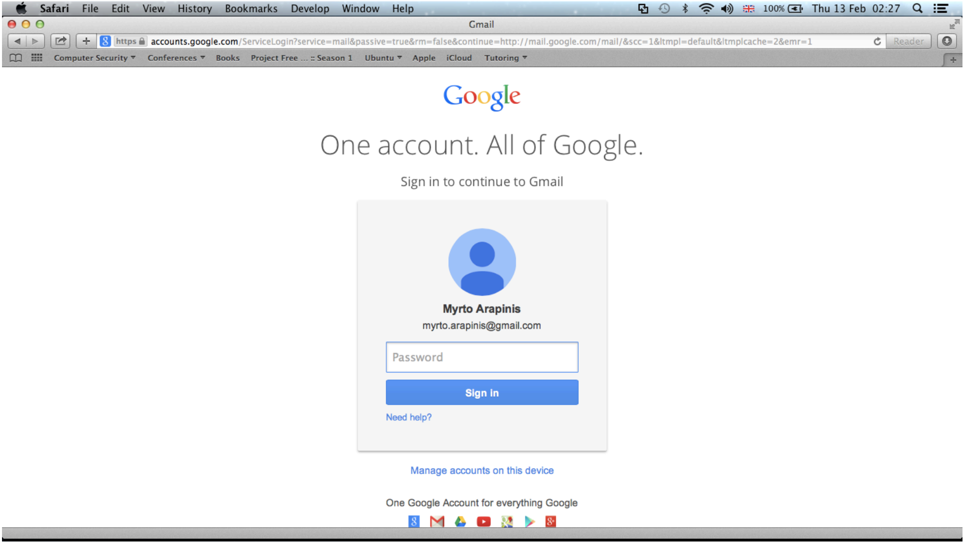
SSL/TLS protocol

Goals: Confidentiality, Integrity, Non repudiation

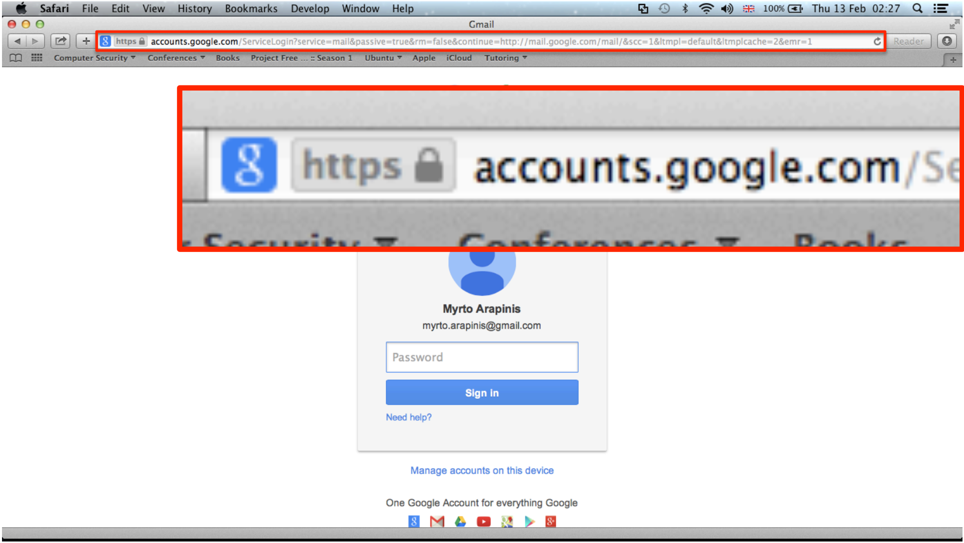


SSL/TLS use X.509 certificates and hence asymmetric cryptography to exchange a symmetric key. This session key is then used to encrypt subsequent communication. This allows for **data/message confidentiality**, and message authentication codes for **message integrity** and thus, **message authentication**.

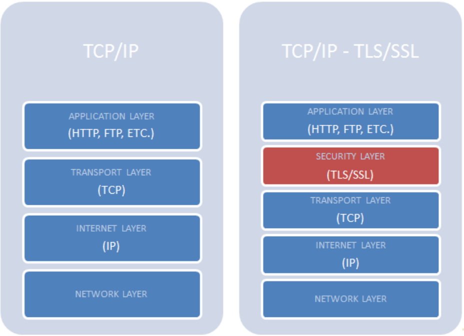
SSL/TLS protocol



SSL/TLS protocol

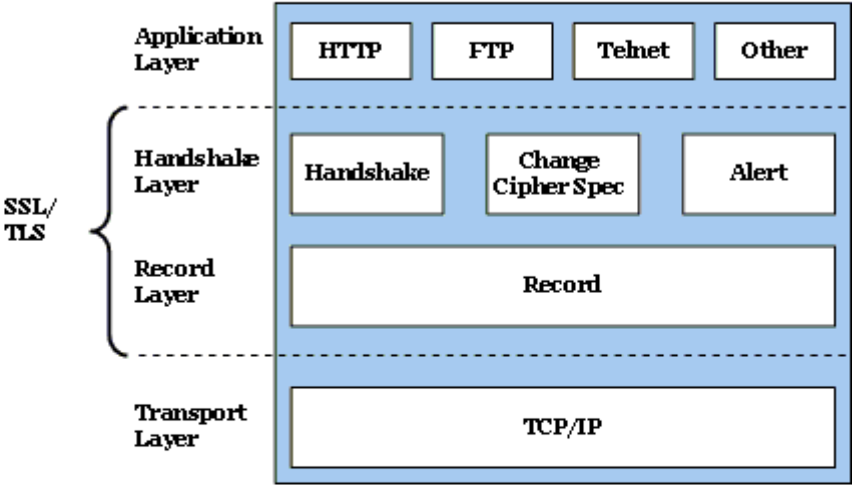


TCP/IP protocol stack

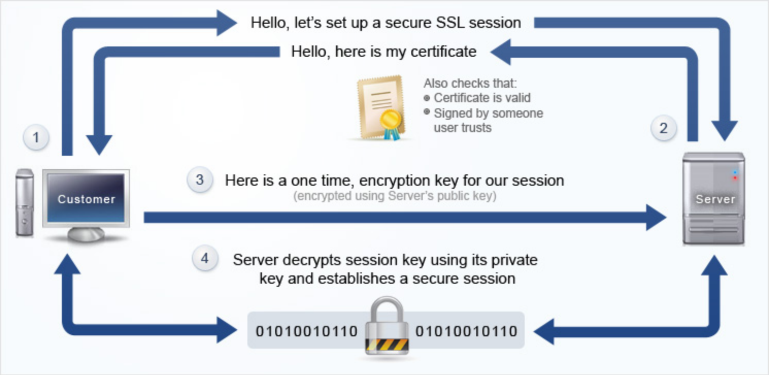


- TCP/IP provides end-to-end connectivity and is organized into four abstraction layers which are used to sort all related protocols according to the scope of networking involved
- The SSL/TLS library operates above the transport layer (uses TCP) but below application protocols

SSL/TLS protocol layers



SSL/TLS handshake protocol



SSL/TLS renegotiation

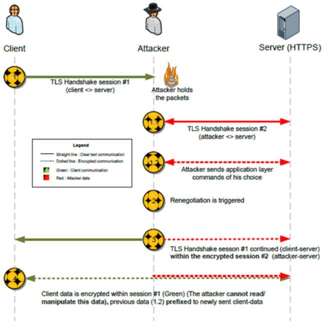
Client and server are allowed to initiate renegotiation of the session encryption in order to:

- Refresh keys
- Increase authentication
- Increase cipher strength
- ...

Client or server can trigger renegotiation by sending a hello meesage

SSL/TLS renegotiation weaknesses

- Renegotiation has priority over application data!
- Renegotiation can take place in the middle of an application layer transaction!



(Detailed on the board)

Incorrect implicit assumtion: the client doesn't change through renegotiation

Marsh Ray's plaintext injection attack on HTTPS

Marsh Ray's plaintext injection attack on HTTPS

Attacker:

```
GET /pizza?toppings=pepperoni;address=attacker_str HTTP/1.1
X-Ignore-This:(no carriage return)
```

Marsh Ray's plaintext injection attack on HTTPS

Attacker:

```
GET /pizza?toppings=pepperoni;address=attacker_str HTTP/1.1
X-Ignore-This:(no carriage return)
```

Victim:

```
GET /pizza?toppings=sausage;address=victim_str HTTP/1.1
Cookie:victim_cookie
```

Marsh Ray's plaintext injection attack on HTTPS

Attacker:

```
GET /pizza?toppings=pepperoni;address=attacker_str HTTP/1.1
X-Ignore-This:(no carriage return)
```

Victim:

```
GET /pizza?toppings=sausage;address=victim_str HTTP/1.1
Cookie:victim_cookie
```

Result:

```
GET /pizza?toppings=pepperoni;address=attacker_str HTTP/1.1
X-Ignore-This:GET /pizza?toppings=sausage;address=victim_str HTTP/1.1
```
Cookie:victim_cookie

Marsh Ray's plaintext injection attack on HTTPS

Attacker:

```
GET /pizza?toppings=pepperoni;address=attacker_str HTTP/1.1
X-Ignore-This:(no carriage return)
```

Victim:

```
GET /pizza?toppings=sausage;address=victim_str HTTP/1.1
Cookie:victim_cookie
```

Result:

```
GET /pizza?toppings=pepperoni;address=attacker_str HTTP/1.1
X-Ignore-This:GET /pizza?toppings=sausage;address=victim_str HTTP/1.1
```

Cookie:victim_cookie

⇒ **Server uses victim's account to send a pizza to attacker!**

Anil Kurmus' plaintext injection attack on HTTPS

Anil Kurmus' plaintext injection attack on HTTPS

Twitter status updates using its API by posting the new status to http://twitter.com/statuses/update.xml, as well as the user name and password

Anil Kurmus' plaintext injection attack on HTTPS

Twitter status updates using its API by posting the new status to http://twitter.com/statuses/update.xml, as well as the user name and password

Attacker:

```
POST /statuses/update.xml HTTP/1.1
Authorization: Basic username:password
User-Agent: curl/7.19.5
Host: twitter.com
Accept:*/*
Content-Length: 140
Content-Type: application/x-www-form-urlencoded
status=
```

Anil Kurmus' plaintext injection attack on HTTPS

Twitter status updates using its API by posting the new status to http://twitter.com/statuses/update.xml, as well as the user name and password

Attacker:

```
POST /statuses/update.xml HTTP/1.1
Authorization: Basic username:password
User-Agent: curl/7.19.5
Host: twitter.com
Accept:*/*
Content-Length: 140
Content-Type: application/x-www-form-urlencoded
status=
```

Victim:

```
POST /statuses/update.xml HTTP/1.1
Authorization: Basic username:password...
```

Anil Kurmus' plaintext injection attack on HTTPS

Twitter status updates using its API by posting the new status to http://twitter.com/statuses/update.xml, as well as the user name and password

Attacker:

```
POST /statuses/update.xml HTTP/1.1
Authorization: Basic username:password
User-Agent: curl/7.19.5
Host: twitter.com
Accept:*/*
Content-Length: 140
Content-Type: application/x-www-form-urlencoded
status=
```

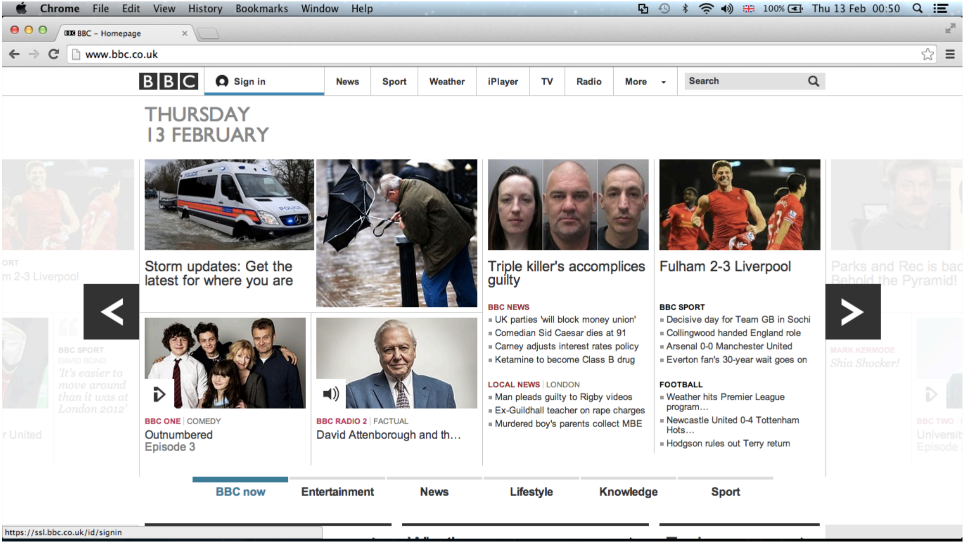
Victim:

```
POST /statuses/update.xml HTTP/1.1
Authorization: Basic username:password...
```

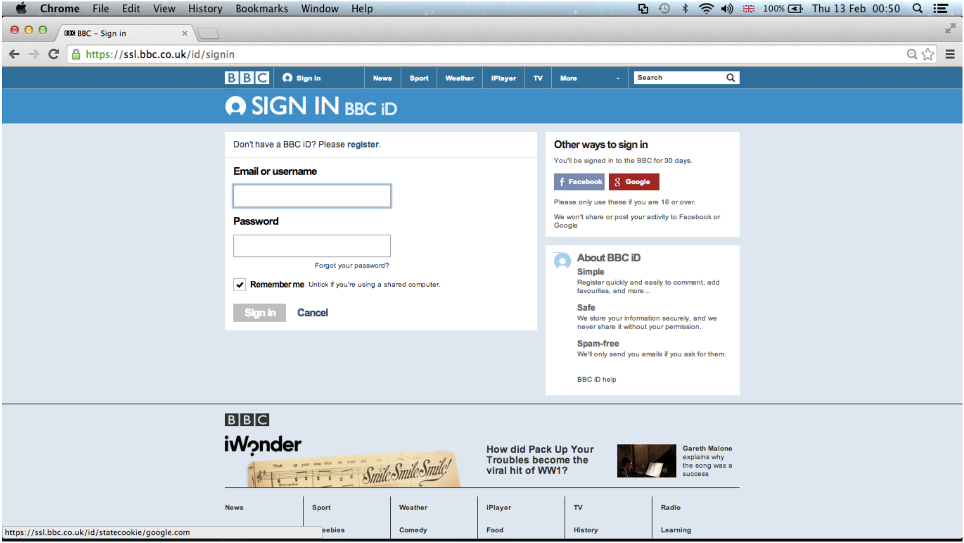
$\Rightarrow$ **the attacker gets the user name and password of the victim!**

The SAML Signle Sign On (SSO) protocol

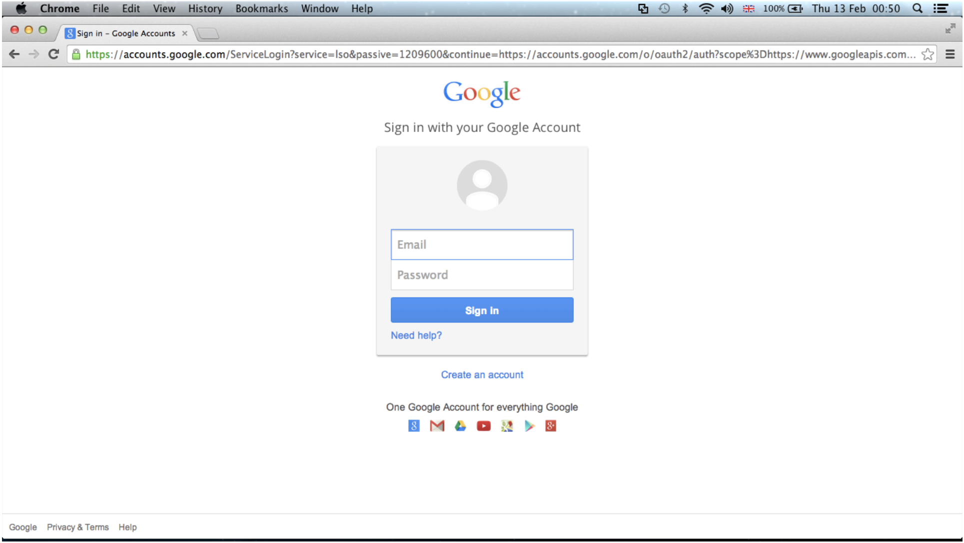
SAML SSO protocol



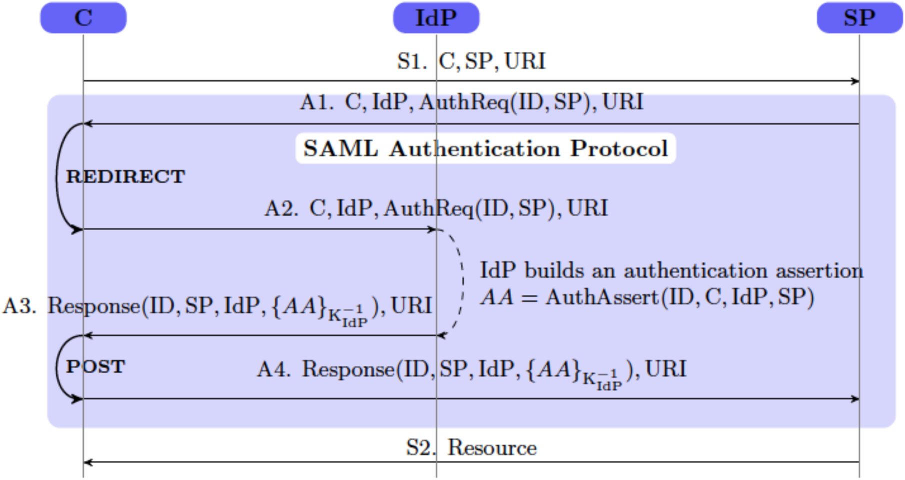
SAML SSO protocol



SAML SSO protocol



SAML SSO protocol (OASIS 2005)



Google's implementation of SSO

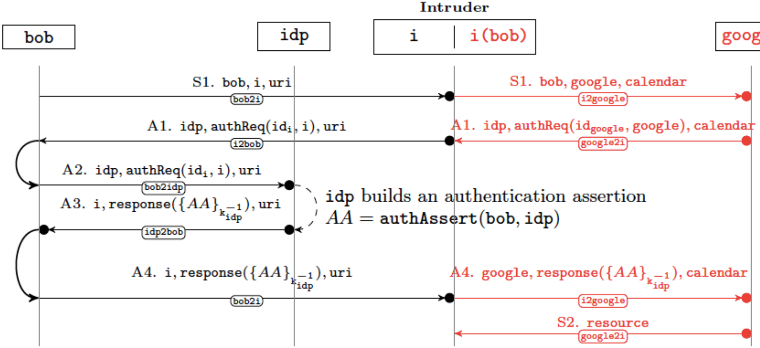
Google's SAML-based Single Sign-On for Google Applications deviates from the above protocol for a few, seemingly minor simplifications in the messages exchanged:

G1. *ID* and *SP* are not included in the authentication assertion, *i.e.* $AA = \text{AuthAssert}(C; IdP)$ instead of $\text{AuthAssert}(ID; C; IdP; SP)$;

G2. *ID*, *SP* and *IdP* are not included in the response, *i.e.* $Resp = \text{Response}(\{AA\}_{K_{IdP}^{-1}})$ instead of $\text{Response}(ID; SP; IdP; \{AA\}_{K_{IdP}^{-1}})$.

Attack Google's SSO implementation

[A. Armando, R. Carbone, L. Compagna, J. Cullar, L. Tobarra, "Formal analysis of SAML 2.0 web browser single sign-on: breaking the SAML-based single sign-on for google apps", (FMSE'08)]



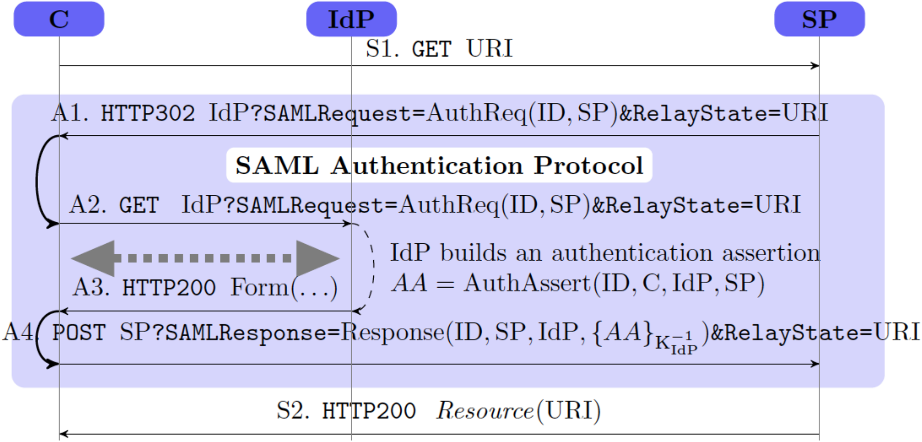
Legend:

$A \xrightarrow[ch]{M}\bullet\ B$: A sends M on ch confidential to B

$A\ \bullet\!\xrightarrow[ch]{M} B$: A sends M on ch authentic for A

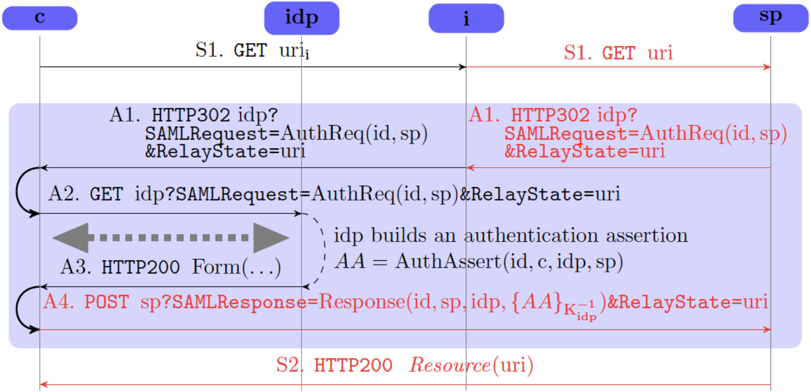
$A\ \bullet\!\xrightarrow[ch]{M}\bullet\ B$: M is sent on ch authentic for A and confidential to B

SAML SSO protocol (OASIS 2012)



Attack SAML SSO protocol (OASIS 2012)

[A. Armando, R. Carbone, L. Compagna, J. Cullar, G. Pellegrino, A. Sorniotti, "From Multiple Credentials to Browser-Based Single Sign-On: Are We More Secure?", Chapter in Future Challenges in Security and Privacy for Academia and Industry]



⇒ XSS attack on SAML-base SSO for Google Apps