Cryptographic protocols (II)

Myrto Arapinis
School of Informatics
University of Edinburgh

February 13, 2014
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 $VS = \{\text{hash}(Data)\}_{sk_B}$
- 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. \( Ca \rightarrow T : Data, \{\text{hash}(Data)\}_{sk_B} \)
Credit card payment protocol (in short)

The terminal reads the card:
1. \( \text{Ca} \rightarrow \text{T} : \text{Data, \{hash(Data)\}_{sk_B}} \)

The terminal asks for the secret pin:
2. \( \text{T} \rightarrow \text{Cu} : \text{pin?} \)
3. \( \text{Cu} \rightarrow \text{Ca} : \text{1234} \)
4. \( \text{Ca} \rightarrow \text{T} : \text{ok} \)
Credit card payment protocol (in short)

The terminal reads the card:
1. \( Ca \rightarrow T : \text{Data, } \{\text{hash(Data)}\}_{sk_B} \)

The terminal asks for the secret pin:
2. \( T \rightarrow Cu : \text{pin?} \)
3. \( Cu \rightarrow Ca : 1234 \)
4. \( Ca \rightarrow T : \text{ok} \)

The terminal calls the bank
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. \( B \rightarrow T : \text{ok} \)
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.

⇒ “YesCard” built by Serge Humpich (France, 1998)
How does the “YesCard” work?

Logical flaw

1. \( Ca \rightarrow T \) : \( Data, \{\text{hash}(Data)\}_{sk_B} \)
2. \( T \rightarrow Cu \) : pin?
3. \( Cu \rightarrow Ca \) : 1234
4. \( Ca \rightarrow T \) : ok
How does the “YesCard” work?

Logical flaw

1. $Ca \rightarrow T : \text{Data, } \{\text{hash(Data)}\}_{sk_B}$
2. $T \rightarrow Cu : \text{pin?}$
3. $Cu' \rightarrow Ca' : 5678$
4. $Ca' \rightarrow T : \text{ok}$
How does the “YesCard” work?

Logical flaw

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

There is always someone to debit
$\rightarrow$ creation of a fake card
How does the “YesCard” work?

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

There is always someone to debit
$\rightarrow$ creation of a fake card
1. $Ca' \rightarrow T : XXXX, \{ \text{hash}(XXXX) \}_{sk_B}$
2. $T \rightarrow Cu' : \text{pin}$
3. $Cu' \rightarrow Ca' : 0000$
4. $Ca' \rightarrow T : \text{ok}$
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 handshake protocol

1. Hello, let's set up a secure SSL session
2. Hello, here is my certificate
   - Also checks that:
     - Certificate is valid
     - Signed by someone user trusts
3. Here is a one time, encryption key for our session
   (encrypted using Server’s public key)
4. Server decrypts session key using its private key and establishes a secure session

01010010110 01010010110
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 message
SSL/TLS renegotiation weaknesses

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

Incorrect implicit assumption: the client doesn’t change through renegotiation

(Detailed on the board)
Marsh Ray’s plaintext injection attack on HTTPS

Attacker:
GET /pizza?toppings=pepperoni;address=attacker
X-Ignore-This:(no carriage return)

Victim:
GET /pizza?toppings=sausage;address=victim
Cookie:victim

Result:
GET /pizza?toppings=pepperoni;address=attacker
X-Ignore-This:GET /pizza?toppings=sausage;address=victim
Cookie:victim

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

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

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

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...

⇒

the attacker gets the user name and password of the victim!
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...

⇒ the attacker gets the user name and password of the victim!
The SAML Single Sign On (SSO) protocol
SAML SSO protocol
SAML SSO protocol
SAML SSO protocol (OASIS 2005)

S1. C, SP, URI

A1. C, IdP, AuthReq(ID, SP), URI

SAML Authentication Protocol

A2. C, IdP, AuthReq(ID, SP), URI

IdP builds an authentication assertion

\[ AA = \text{AuthAssert}(ID, C, IdP, SP) \]

A3. Response(ID, SP, IdP, \( \{ AA \}_{K_{IdP}^{-1}} \)), URI

A4. Response(ID, SP, IdP, \( \{ AA \}_{K_{IdP}^{-1}} \)), URI

S2. Resource
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:

\[ \begin{align*}
A \xrightarrow{M} B & : A \text{ sends } M \text{ on } ch \text{ confidential to } B \\
A \xrightarrow{M} B & : A \text{ sends } M \text{ on } ch \text{ authentic for } A \\
A \xrightarrow{M} B & : M \text{ is sent on } ch \text{ authentic for } A \text{ and confidential to } B
\end{align*} \]
SAML SSO protocol (OASIS 2012)

A1. HTTP302 IdP?SAMLRequest=AuthReq(ID, SP)&RelayState=URI

A2. GET IdP?SAMLRequest=AuthReq(ID, SP)&RelayState=URI

A3. HTTP200 Form(…)

IdP builds an authentication assertion

AA = AuthAssert(ID, C, IdP, SP)

A4. POST SP?SAMLResponse=Response(ID, SP, IdP, \{AA\}_{K_{IdP}}^{-1})&RelayState=URI

S2. HTTP200 Resource(URI)
[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