MashSSL ALLIANCE

MashSSL
Quick Summary

Siddharth Bajaj
VeriSign Inc.
sbajaj@verisign.com
www.verisign.com

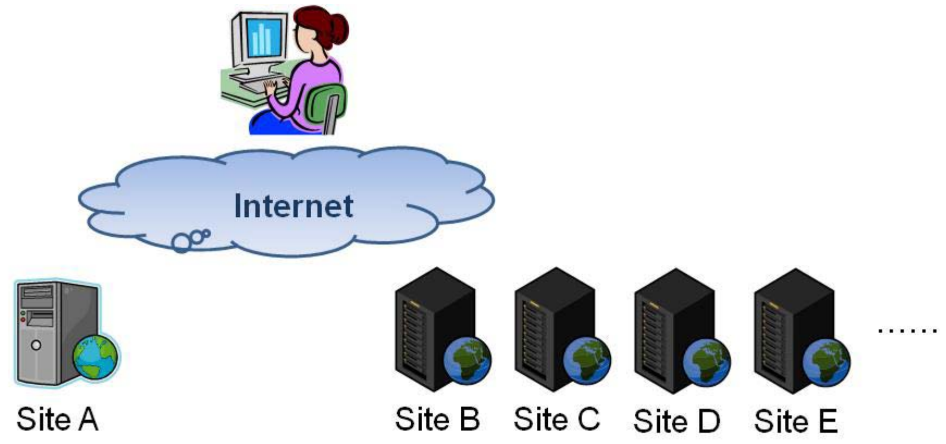
Ravi Ganesan
SafeMashups Inc.
ravi@safemashups.com
www.safemashups.com

info@mashssl.org
www.mashssl.org

Outline

- Why do we need a new security standard
- Why is it preferable to start with TLS
- What is MashSSL
- How MashSSL and TLS work together
- Some MashSSL innovations that TLS might want to consider
- Towards standardization…
- Back up slides

Why MashSSL: An Example



- Site A “talks” through Alice to various other sites…
    - Site B provides a payment button.
    - Site C acts as Identity Provider using OpenID
    - Site D is an OAuth Service Provider to Site A
    - Site E wants to accept cross domain XHR request from Site A
    - And so on….web experiences are increasingly mashups…

•But Alice could be Evil Eve! How do sites authenticate each other via browser?

• Today for each problem different underlying crypto protocols are ginned up and credentials distributed. Site A has crypto s/w and credential management headache!

Would be nice if we could build standard secure pipe from site to browser to site.

Why start with TLS?

- Pros:
  - New protocols take years to mature.
  - TLS handshake is probably the world's most carefully studied mutual authentication and key exchange protocol.
  - Very efficient: Only initial handshake needs PKI processing.
  - Trust infrastructure exists (get and manage SSL certificates)

- Cons:
  - TLS does not allow MITMs. Our problem is multi-party.
  - TLS usually runs over TCP. For us the "transport" is HTTP.
  - TLS (for very good reason) is large and complex. World seems to want simple RESTful protocol…

Very powerful "pros" suggest we start with TLS, and try and address the "cons".

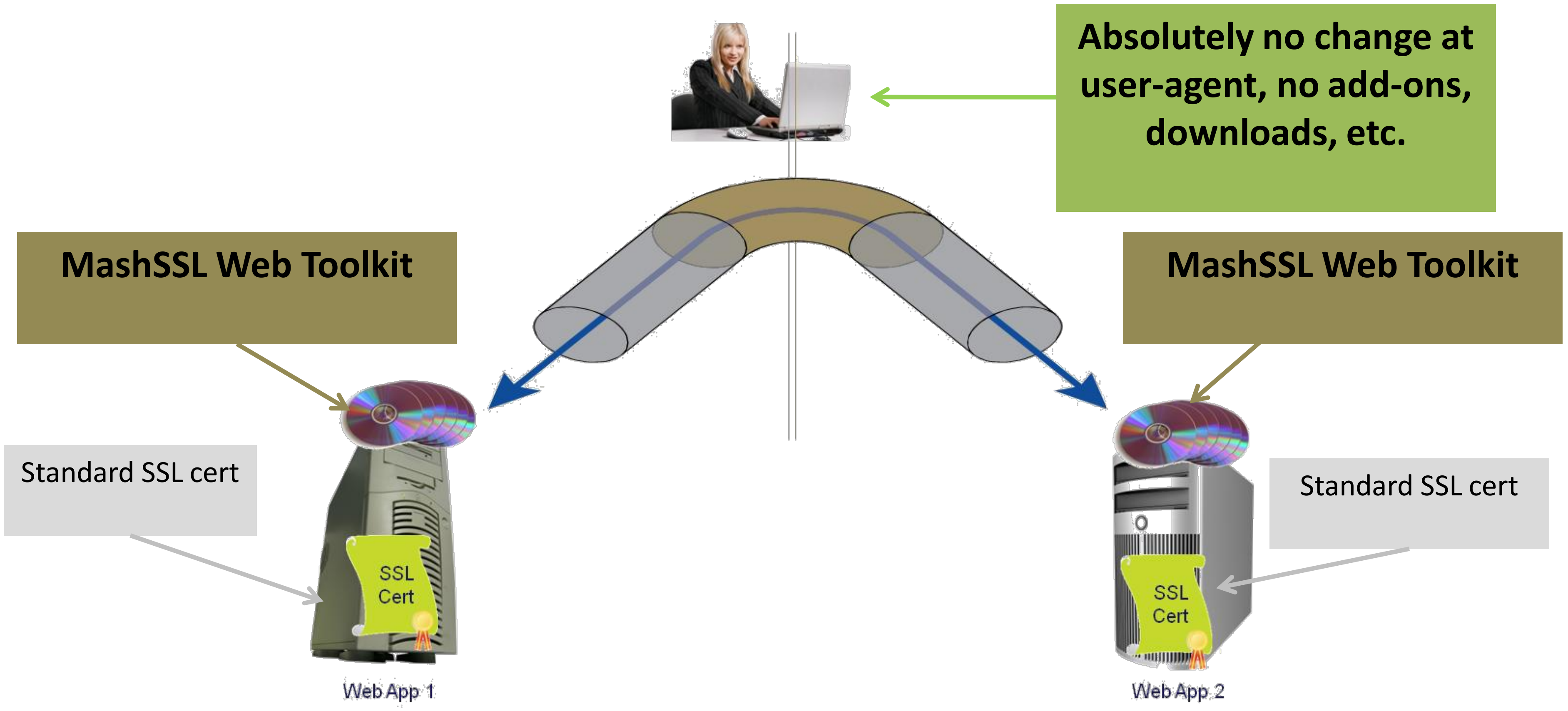
MashSSL ALLIANCE

MashSSL in a nutshell

- Introduces concept of "friend in the middle" to make TLS multi-party.
- Takes RFC 5246 handshake messages and exchanges them as name value pairs over HTTP via browser.
- Only implements core subset of TLS (e.g. no renego!) to keep it simple.

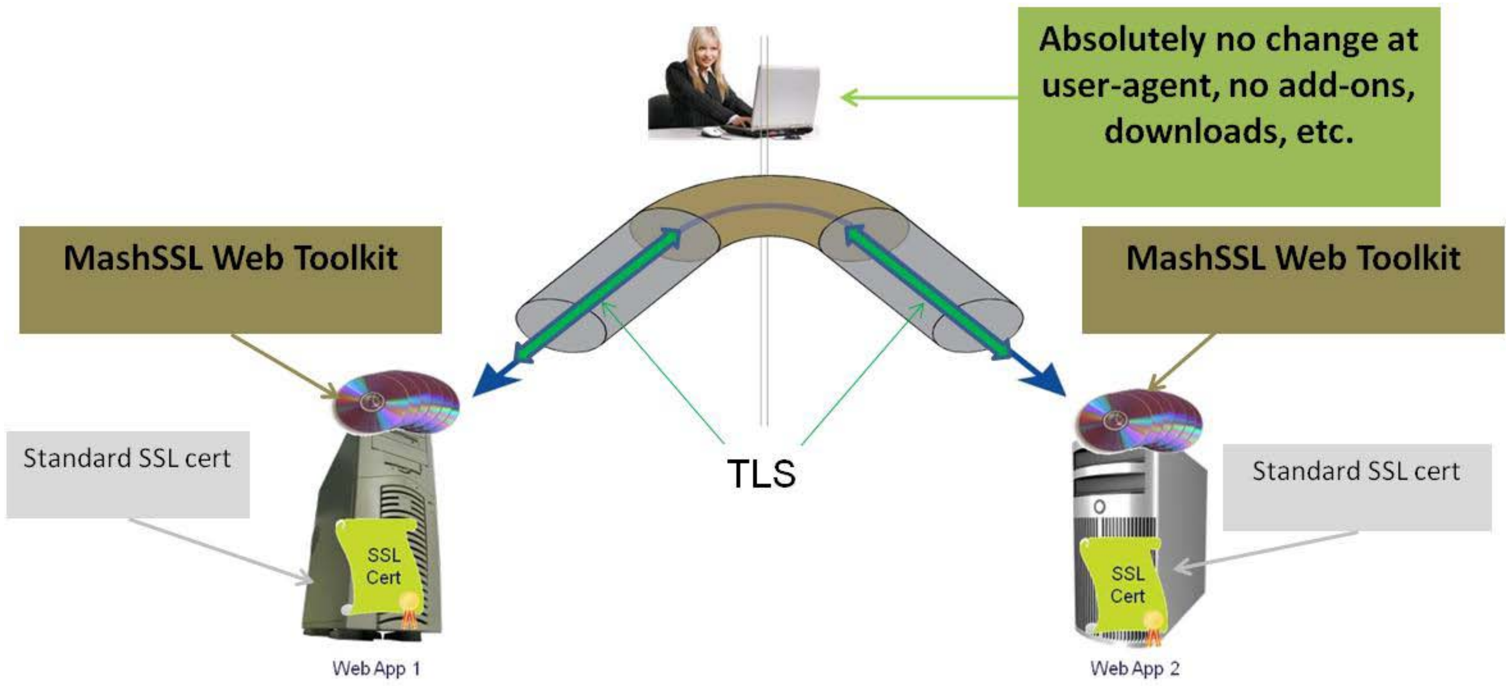
- In general never defines something already defined in RFC 5246 and simply points to it. For example:
  - **server_hello**.*certificate*.certificate_list
    - The Server's certificate with the chain up until or including the root. The certificate must make sense in the context of the cipher-suite chosen. *Further reference: RFC 5246, Page 46.*

MashSSL in a picture



Result: Secure pipe between two web apps over which any "mashup protocol" can be run.

MashSSL and TLS



- Likely scenario:
  - Two independent TLS sessions over TCP pipes
  - Over which are two independent HTTP sessions
  - Over which is overlaid a MashSSL pipe
  - On top of which one can run OAUth, OpenID, cdXHR, etc.

Some aspects useful for TLS?

- It is expected that web apps will do MashSSL:
  - "full handshake" occasionally (say once a day) *directly* between themselves.
  - "abbreviated handshake" (aka resume) through user browsers
  - PKI operations only once a day!
- For this we have defined 2-legged MashSSL
  - Which could as easily run between browser and a server.
  - 2-legged MashSSL is kind of a HTTP binding for TLS.
  - Lots of advantages to authenticate and encrypt at Layer 7 (performance, avoid mobile gateway MITMs, etc.)
- HTTP is inherently request response:
  - So MashSSL handshakes end up requiring two round trips.
  - MashSSL will have a single round trip optimization a Server can choose to accept (or not). This could be added to TLS. (see http://www.ietf.org/mail-archive/web/tls/current/msg05728.html)

Towards standardization

- For current specification and open source software please visit MashSSL Alliance site: www.mashssl.org

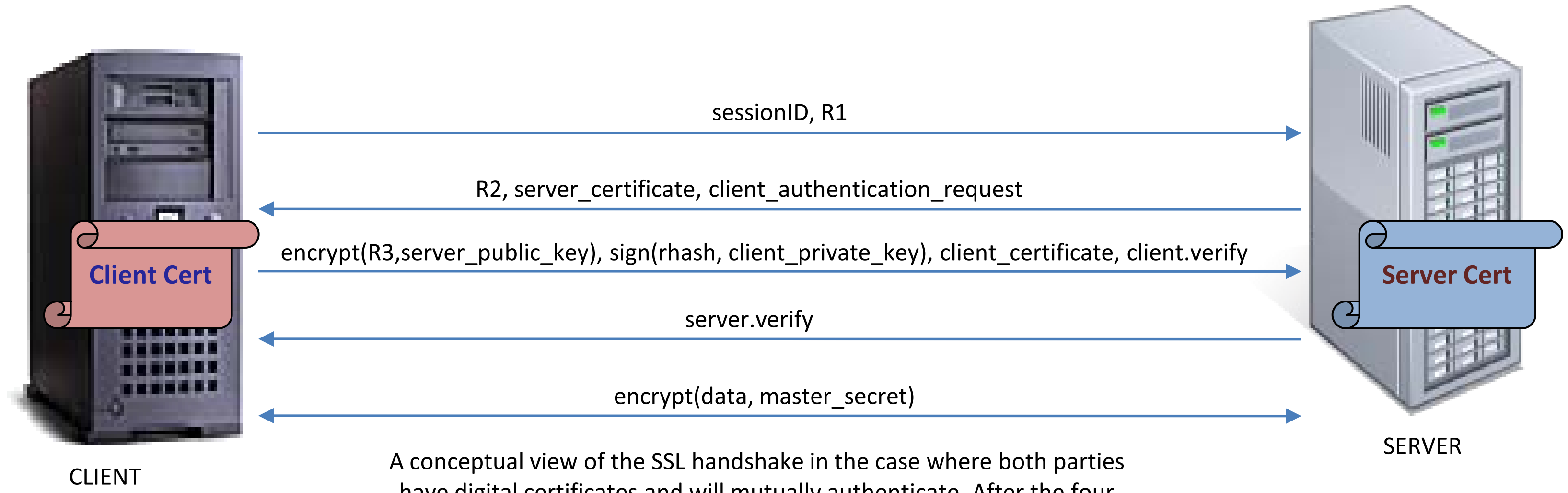
- Have formed W3C Incubator: www.w3.org/2005/Incubator/MashSSL

- Welcome participation from all!

Thank you!

(back up slides follow)

SSL revisited

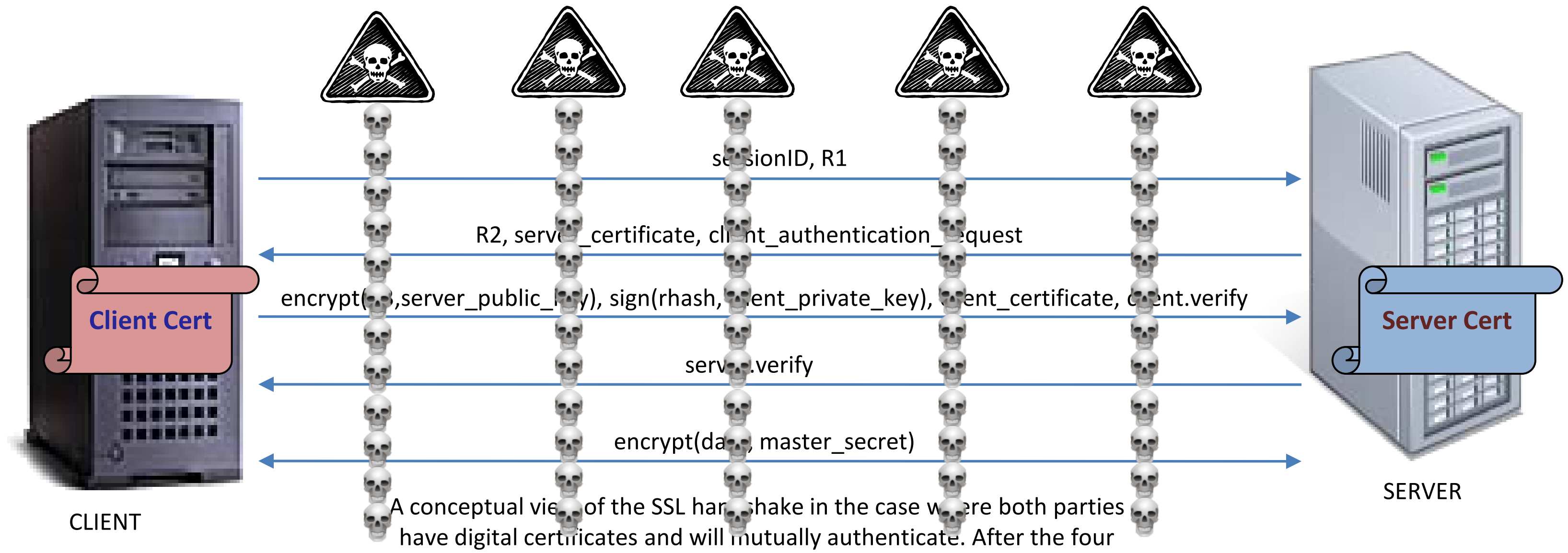


sessionID, R1
R2, server_certificate, client_authentication_request
encrypt(R3,server_public_key), sign(rhash, client_private_key), client_certificate, client.verify
server.verify
encrypt(data, master_secret)
Client Cert
Server Cert
CLIENT
SERVER

A conceptual view of the SSL handshake in the case where both parties have digital certificates and will mutually authenticate. After the four messages, both sides can encrypt data using a shared master_secret which is derived from R1, R2 and R3.

SSL protocol is widely used and trusted. Ability to reuse session without repeating PKI operations. Trust infrastructure CAs already exist. And is constantly being improved (TLS 1.2, EV certs). Many mashup apps already have SSL certs!

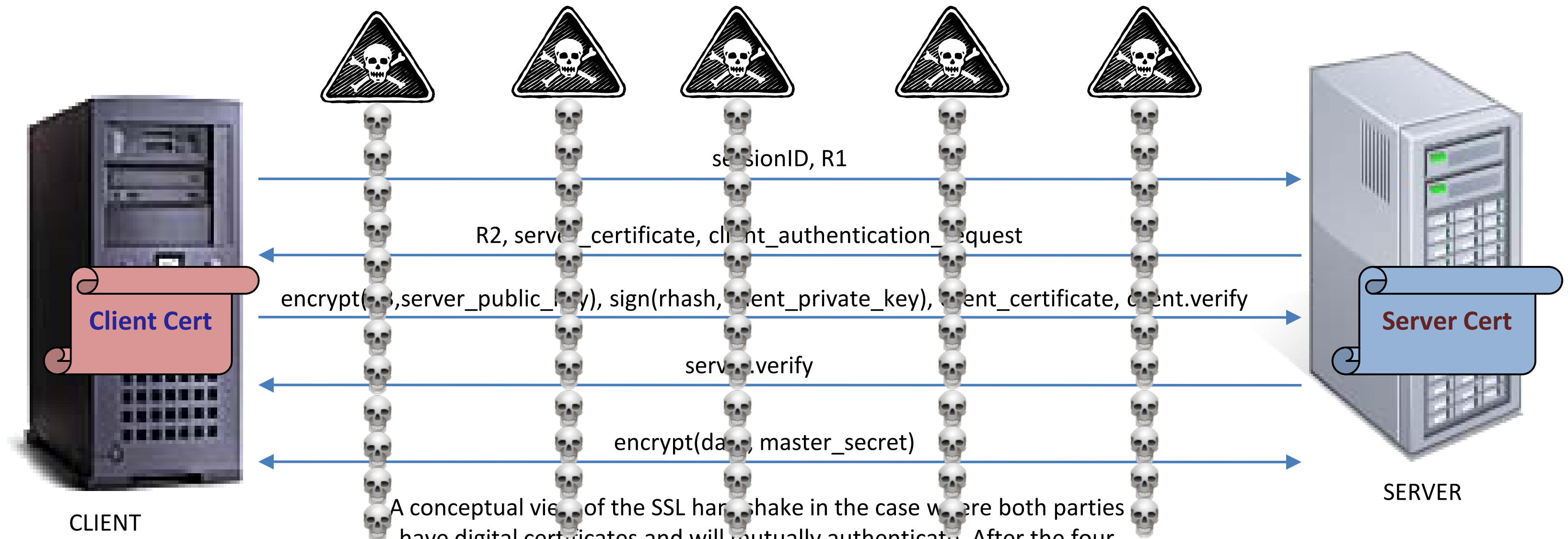
SSL revisited


sessionID, R1
R2, server_certificate, client_authentication_request
encrypt(R3,server_public_key), sign(rhash, client_private_key), client_certificate, client.verify
server.verify
encrypt(data, master_secret)
Client Cert
Server Cert
CLIENT
SERVER

A conceptual view of the SSL handshake in the case where both parties have digital certificates and will mutually authenticate. After the four messages, both sides can encrypt data using a shared master_secret which is derived from R1, R2 and R3.

And, it is believed that the protocol is secure even in the presence of one or more MITM(s). ***One of the strongest claims one can make of a crypto protocol!***

Lets ask for the impossible…


sessionID, R1
R2, server_certificate, client_authentication_request
encrypt(R3,server_public_key), sign(rhash, client_private_key), client_certificate, client.verify
server.verify
encrypt(data, master_secret)
Client Cert
Server Cert
CLIENT
SERVER

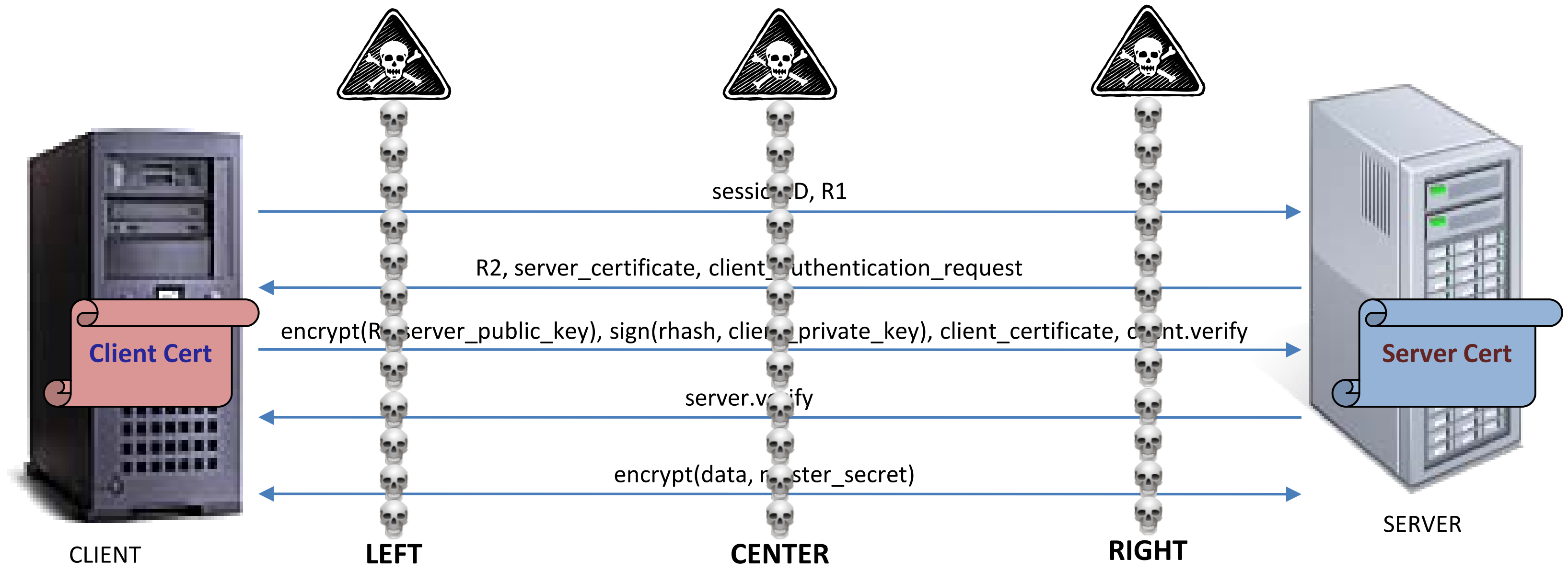
A conceptual view of the SSL handshake in the case where both parties have digital certificates and will mutually authenticate. After the four messages, both sides can encrypt data using a shared master_secret which is derived from R1, R2 and R3.

Can we insert MITMs that manipulate protocol messages, but which

1. Allow successful execution of protocol
2. Do not compromise underlying security of protocol

<u>The very surprising non-intuitive answer is: YES WE CAN!</u>

The impossible becomes possible if…


session ID, R1
R2, server_certificate, client_authentication_request
encrypt(R, server_public_key), sign(rhash, client_private_key), client_certificate, client.verify
server.verify
encrypt(data, master_secret)
Client Cert
Server Cert
CLIENT
LEFT
CENTER
RIGHT
SERVER

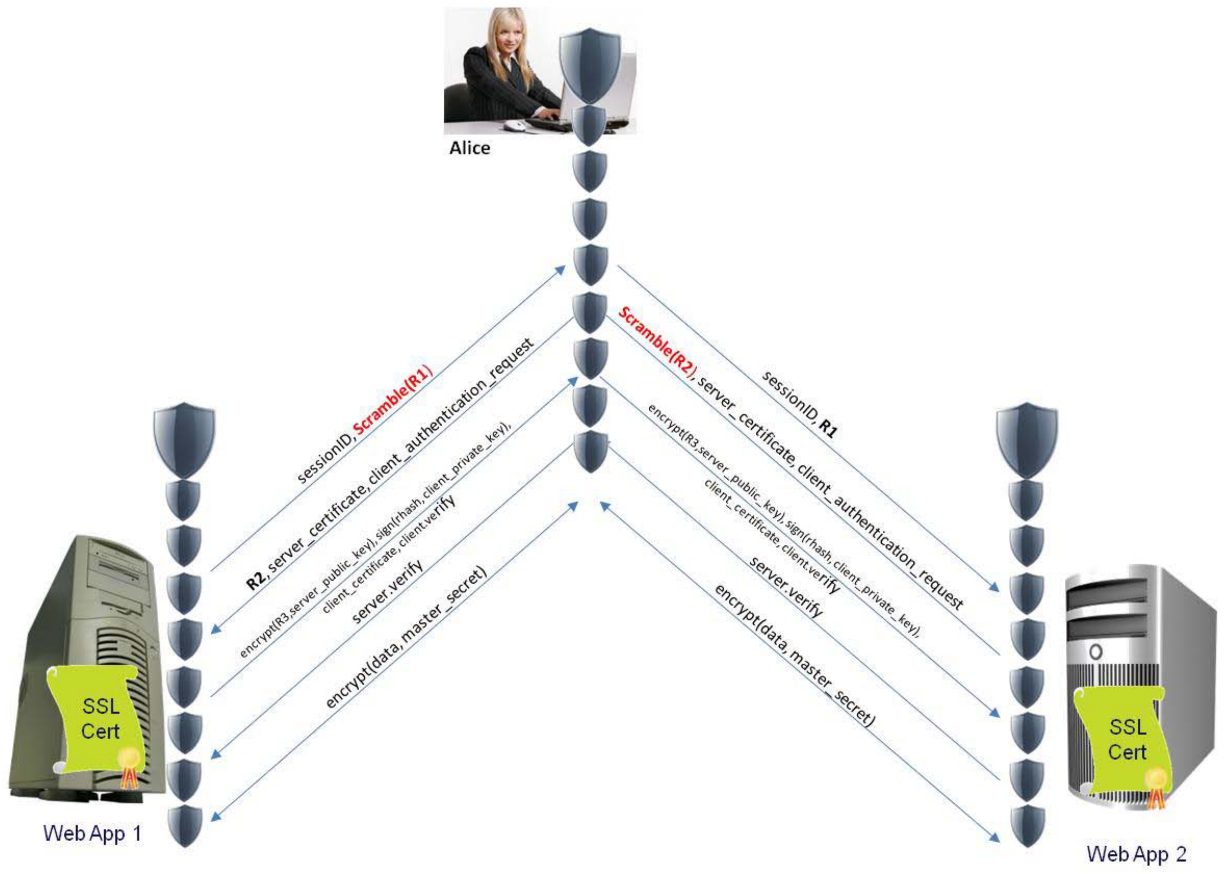
<u>the changes the MITMs make cancel each other out en route!</u>

Example:

- CLIENT sends random number R1. MITM-LEFT adds 100.
- MITM-Center subtracts 50. MITM-Right subtracts another 50.
- SERVER receives original random number R1!

- So of what possible practical use is this insight???

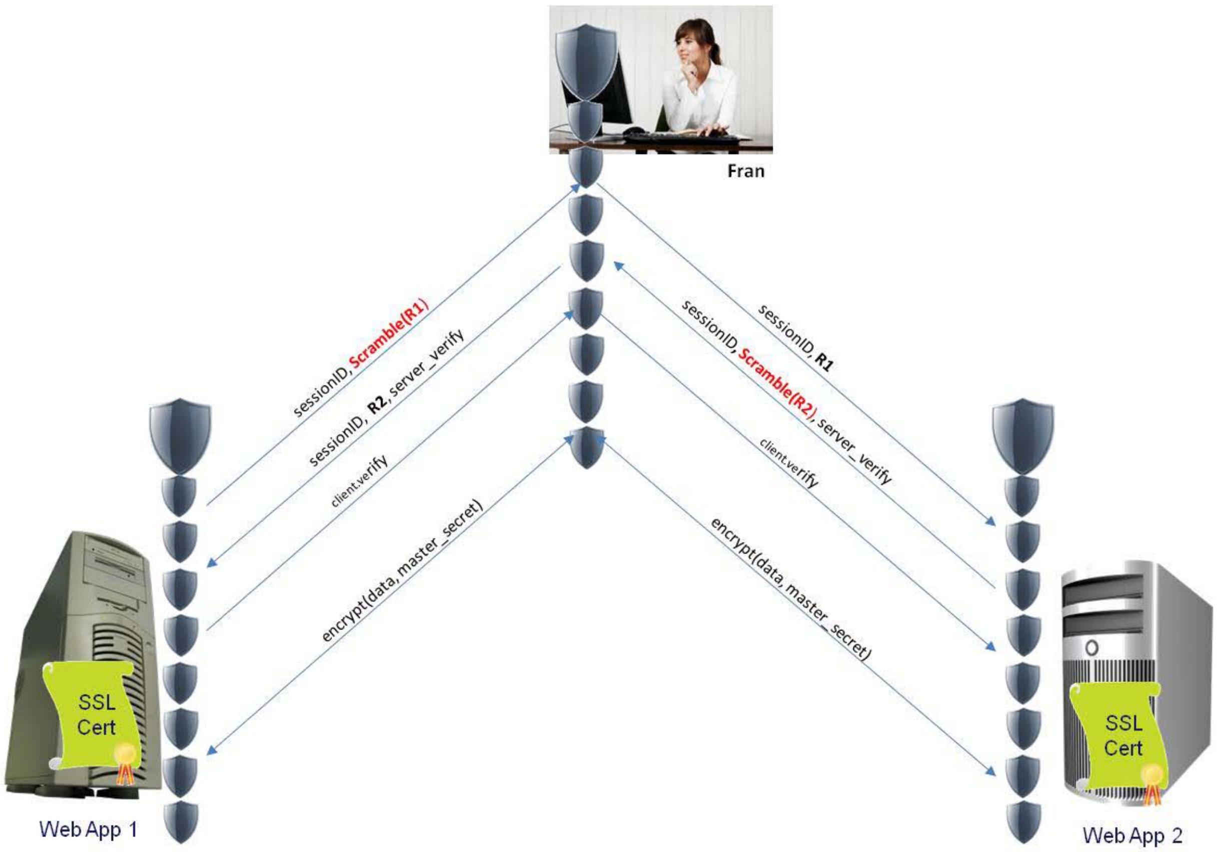
Introducing MashSSL



Alice
sessionID, Scramble(R1)
R2, server_certificate, client_authentication_request
encrypt(R3,server_public_key), sign(rhash, client_private_key),
client_certificate, client.verify
server.verify
encrypt(data, master_secret)
sessionID, R1
Scramble(R2), server_certificate, client_authentication_request
encrypt(R3,server_public_key), sign(rhash, client_private_key),
client_certificate, client.verify
server.verify
encrypt(data, master_secret)
SSL Cert
Web App 1
SSL Cert
Web App 2

MashSSL uses the browser as a "friend in the middle". The two web apps generate SSL messages, then manipulate (scramble) them such that only Alice at her browser can unscramble them before reaching the other side.

Introducing MashSSL


Fran
sessionID, Scramble(R1)
sessionID, R2, server_verify
client.verify
encrypt(data, master_secret)
sessionID, R1
sessionID, Scramble(R2), server_verify
client.verify
encrypt(data, master_secret)
SSL Cert
Web App 1
SSL Cert
Web App 2

And, if they subsequently do mashup for another user Fran they simply use the SSL abbreviated handshake which avoids repeating the PKI operations. (Note in practice the web apps can do 2-legged MashSSL for full handshake)

How to scramble in MashSSL?

- Hundred of links in Google for “recipes for scrambled eggs” !
- Similarly methods for scrambling in MashSSL are endless
- Like with scrambled eggs not all results will be tasty! See white paper for guidelines on security. (https://www.safemashups.com/downloads/MashSSL_Towards_Multi_Party_Trust_in_the_Internet.pdf)
- E.g. can use any existing authentication method: passwords, OTPs, smartcards, etc.
- Can even ‘scramble virtually’ if user is already logged in and a session cookie identifies session.

How does MashSSL stack up?

✓ 1. Single solution for all situations where problem manifests.
  - *MashSSL is a fundamental Internet building block that has countless uses.*

✓ 2. Lightweight RESTful application level protocol (HTTP).
  - *Standard defined in simple RESTful fashion.*

✓ 3. No new crypto protocol please. Takes up to a decade to build trust.
  - *Reuses SSL. Reuses whatever authentication is in place for scrambling.*

✓ 4. Trust browser as little as possible.
  - *Browser cannot spoof either web application!*

✓ 5. Don't ask us to get and manage new credentials from new authorities.
  - *Standard SSL certificates can be used.*

✓ 6. Don't use user authentication as proxy for B2B authentication.
  - *Web applications authenticating each other (through browser)*

✓ 7. Think scale. Do not repeat expensive PKI operations.
  - *Reuses SSL abbreviated handshake to avoid repeating PKI operations.*