4 Altro Scenario

4.1 Attori e scenario:

- **Alice**: *End User*
- **Operfox Explorer**: *User-Agent*
- **Bob**: *Consumer*
- **Carol**: *OpenID Server*
- **http://carol.example.com/Alice**: *Identity*
- **Ive**: *malicious attacker*

Scenario:

- Alice vuole autenticarsi preso Bob
- Bob supporta l'utilizzo del protocollo Open ID
- Alice è autenticata preso l'OpenID Provider Carol, è possiede un account
- http://carol.example.com/Alice, è l'identifier di Alice presso Carol
- Ive vuole spacciarsi per Alice presso Bob

Domande:

Cosa succedere quando Alice inserisce il suo identifier ?

Come fa Bob ad essere sicuro che si tratta veramente di Alice e non di Ive ?

Cosa deve fare Carol per assicurare Bob che si tratta di Alice?

4.1 Act 1:

Alice inserisce la sua identità OpenID, nella form che Bob gli presenta e clicca sul pulsante "Authenticate". Operfox Explorer, lo User Agent di Alice processa la form e la invia ad un CGIs implementato presso Bob (il consumer). Il CGI di Bob normalizza l'url inserito da Alice ed analizza il documento che li viene restituito da tale Url. Fin qui niente di speciale, semplicemente una richiesta di un file via HTTP GET. Quando il consumer (Bob), prende tale file dall'identity url inserito da Alice, lo esplora cercando un tag specifico nella sessione header di tale documento:

"<link rel="opened.server" href=http://carol.example.com/openid-server.cgi">"

Nella modalità "dumb" Bob reindirizza (redirect) l'End User di Alice a tale URL (Carol's OpenID server - http://carol.example.com/openid-server.cgi), aggiungendo alcune informazioni (HTTP GET):

- **openid.mode** = checkid_setup. É una delle modalità possibili. Questo significa che il consumer Bob intende controllare (effettuare un check) un'identità, quella di Alice e sta passando il controllo dello

User Agent al server Carol per effettuare tale controllo. Bob si aspetta che il server risponde, una volta effettuato il controllo.

- **openid.identity** = http://carol.example.com/Alice Questa è l'identifier inserito da Alice, ed è questo che Bob vuole verificare con l'aiuto di Carol.

- **openid.return_to** = http://bob.example.com/comment.cgi?session_id=Alice&nonce=123456

Questo è l'url dove Bob vuole che lo User Agent di Alice si posizioni dopo che Carol ha effettuato l'autenticazione. Inoltre Bob si aspetta dei dati aggiuntivi su questo Url.

4.2 Act 2 Ive:

Vediamo il parametro **openid.return_to.** Questo è il punto dove Bob si aspetta che Alice si posizioni dopo l'autenticazione da parte di Carol. Bob ha da quache parte la session_id di Alice legato a ciò che Alice doveva fare preso Bob. Ma a noi interessa qualcosa in più che una semplice sessione di autenticazione. Ive, per rubare l'identità di Alice deve solo mettersi in ascolto della conversazione Alice – Bob, memorizzare tale conversazione ed effettuare un "replay attack".

Anche se la conversazione fosse crittografata e Ive non potesse vedere cosa ci sia all'interno, per lui sarà ancora possibile effettuare lo "spoofing" dell'identità di Alice. Dobbiamo procurarsi qualcosa in più per assicurarsi che Ive non può farlo, e questo è il parametro extra "**nonce**".

Il consumer Bob, inserisce un identificatore, un numero random, all'Url per ogni richiesta di autenticazion che fa. Grazie a questo valore, "nonce", ogni richiesta di autenticazione ha il suo valore univoco e quindi per Ive sarà impossibile effettuare un "replay attack" poiché il valore return_to è univoco a quella sessione.

4.3 Act 3 Torniamo da Carol

Torniamo da Carol che ha del lavoro da svolgere. Bob le ha chiesto di confermare che Alice veramente possiede l'url che ha inserito, reindirizzando lo user agent di Alice presso Carol. Carol, adesso, ha il controllo dello User Agent di Alice e quindi può autenticare che ci sia veramente Alice dall'altra parte dello schermo.

Come fa ???

Come dicevamo in precedenza questo non fa parte del protocollo OpenID. Per OpenID questa verifica è una scatola nera. OpenID di occupa dell'identità di Alice e non della fiducia che Carol ha in Alice. Quello che ci interessa, ed è quello che interessa anche a Bob, è quello che ci risponde Carol riguardante Alice. Bob si fida di Carol, altrimenti che senso avrebbe interrogare Carol per l'identità di Alice. Ciò che sappiamo è che Alice è la stessa Alice che Carol dice di lei. Oltre a questo, Alice non può far finta di essere qualcun'altra, almeno non senza l'aiuto di Carol. E se cosi fosse dobbiamo smettere di prestare attenzione a Carol ed a tutti gli utenti che usano Carol come OpenId Provider.

Carol si decide che sia davvero Alice dall'altra estremità dello User Agent. Ora, Carol deve solo convincere Bob che sia davvero Alice in linea, e anche convincerlo che essa è davvero Carol. Facile, vero? Il primo step in questo processo è per Carol quello di reindirizzare lo User Agent di Alice nel cgi di Bob http://bob.example.com/comment.cgi?session_id=Alice&nonce=123456 (return_to url indicato da Bob) con in aggiunta questi parametri (GET):

- **openid.mode** = id_res Questo valore indica che Carol afferma che Alice possiede l'identifier da lei dichiarato. Questo valore può essere anche "cancel" che sta ad indicare che Alice non vuole continuare l'autenticazione.

- openid.return_to = http://bob.example.com/comment.cgi?session_id=Alice&nonce=123456
Lo stesso URL che Bob ha inviato a Carol nella richiesta di autenticazione per Alice.

- **openid.identity** = http://carol.example.com/Alice L'identifier che Alice dice di possedere e che Carol ha affermato.

- **openid.signed** = mode,identity,return_to. Questo è l'elenco dei parametri a cui dobbiamo fornire una firma. Ovviamente vogliamo firmare la modalità e l'identità che stiamo autenticazione, ma anche l'URL di return_to per impedire attacchi di tipo replay, come discusso prima.

- **openid.assoc_handle** = *opaque handle*

- **openid.sig** = *base 64 encoded HMAC signature*. (HMAC = **keyed-hash message authentication code**)

Vediamo in dettaglio gli ultimi due parametri:

Il primo **assoc_handle** è definito nel protocollo OpenID come "an opaque handle". Questa è una handle per il segreto. Carol deve poter prendere questo handle opaco e ricercare, internamente, quali segreti ha utilizzato per la firma dell'affermazione per Bob. Come lei fa questo è un problema interno di Carol, ma l'unica cosa che lei ha bisogno e quello di distinguere tra un normale assoc_handles e quella della modalità stateless.

Carol costruisce questo segreto e lo lega al assoc_handle, ma a che serve questo segreto? Bene, viene utilizzato per creare il secondo elemento di interesse, la firma. Carol raggruppa insieme i campi che abbiamo detto in precedenza e che intende firmare (nel nostro caso: mode, identity e return_to) e quindi esegue l'algoritmo HMAC-SHA1* per la firma su di esso, utilizzando il segreto che ha legato alla assoc_handle come chiave di crittografia per HMAC. Questo genera una firma che ella quindi codifica in base 64 da inviare come parametro a Bob.

***HMAC** (**keyed-hash message authentication code**) è una tipologia di codice per l'autenticazione di messaggi basata su funzione di hash, utilizzata in diverse applicazioni legate alla sicurezza informatica. Tramite HMAC è infatti possibile garantire sia l'integrità che l'autenticità di un messaggio. HMAC utilizza infatti una combinazione del messaggio originale e una chiave segreta per la generazione del codice. Peculiare di HMAC è il non essere legata a nessuna funzione di hash particolare, questo per rendere possibile una sostituzione della funzione nel caso fosse scoperta debole. Nonostante ciò le funzioni più utilizzate sono MD5 e SHA-1

4.4 Showtime Bob! Or is it?

Grande! Bob ha ora un'affermazione firmata che Alice è davvero chi lei afferma di essere e tutto ciò è andato bene, giusto? Bene ci sono ancora dei dubbi. In primo luogo, Bob effettivamente non può controllare la firma da solo, dopo tutto non consce il segreto. Tutto quello che possiede è questa handle che con essa egli non può fare nulla e una firma che può o non può essere valida. In secondo luogo, tutta questa informazione proviene direttamente dallo User Agent di Alice in ogni caso, e l'intero set di bit possa essere compromesso. Bob a questo punto non sa se questa informazione ha da fare con Carol o no. È quindi?

Come abbiamo detto prima, Bob lavora in modalità dumb e quindi deve ricontattare Carol. Il CGI che lo User Agent di Alice ha contattato presso Bob aprirà una sessione HTTP POST a Carol un'altra volta prima di fidarsi dell'autenticazione di Alice. Questa ultima sessione avviene direttamente fra Bob e Carol e lo User Agent di Alice non ha a che fare. Bob invia via il metodo HTTP POST, i seguenti parametri al cgi di Carol (http://carol.example.com/openid-server.cgi):

- **openid.mode** = check_authentication: dicendo a Carol che intende confermare quanto

Carol ha detto di Alice.

- **openid.signed** = mode,identity,return_to: firmati con la firma di Bob

- **openid.assoc_handle** = *opaque handle*: Lo stesso di prima

- **openid.sig** = *base 64 encoded HMAC signature*

Quando Carol riceve questa richiesta andrà a fare tutto il lavoro che ha già fatto in precedenza.

Lei aggiungerà all'elenco dei parametri firmati insieme ancora una volta, ricercherà il segreto che ha legato alla assoc_handle fornito e creare una firma HMAC-SHA1 dei parametri utilizzando questo segreto come la chiave di crittografia, proprio come ha fatto prima. Quindi lei confronterà la firma digitale con la firma che Bob ha fornito nel suo messaggio. Se essi corrispondono, allora Carol sa che deve essere stata lei che ha inviato l'affermazione originale (o altrimenti, qualcuno che conosce i suoi segreti), e restituirà un file di testo normale da Bob con la sua risposta definitiva: "**is_valid:true**".

Ovviamente se le due firme non corrispondono Carol invierà il messaggio "**is_valid:false**", e Bob saprà che qualcosa è andato storto.

4.5 Bob e Javascript - (AJAX)

In questo caso viene utilizzato il messaggio "checkid_immediate" il quale come "checkid_setup" openid.mode è inviato al server OpenID di Carol con tutta l'informazione necessaria per autenticare l'identifier di Alice. L'unica differenza di questa modalità è che, come indicato dal nome, viene effettuata immediatamente senza prendere il controllo dello User Agent di Alice. Se il server può autenticare in loco, tutto continua come normale e il consumatore (Bob) deve seguire con un messaggio "check_authentication".

Tuttavia, se il server OpenID di Carol non può fare un'affermazione positiva sull'identità, "checkid_immediate" restituisce quindi istantaneamente un'asserzione fallita e un parametro singolo "openid.user_setup_url" invece di assumere il controllo dello User Agent. A questo punto, il consumatore (Bob) può decidere che cosa fare. Può reindirizzare lo User Agent nell'Url "openid.user_setup_url" fornito, o fare ciò con un popup. Il punto chiave, tuttavia, è che il consumatore ha il controllo di come dovrebbe comportarsi lo User Agent.

Quando un utente tenta di autenticarsi, invece di inviare la form direttamente al server OpenID server tramite lo User Agent, si usa JavaScript per aprire un HTTP Request al server OpenID. E anziché inviare una modalità "checkid_setup", che si aspetta di poter assumere il controllo dello User Agent, inviano invece un messaggio "checkid_immediate". In questo modo si può aprire una nuova finestra web (popup) con l'url "**openid.user_setup_url**" dove l'utente può finalizzare i passi necessari al server OpenID per consentire l'autenticazione. Seguento, con un altro HTTP Request per inviare il comando "check_authentication" l'intero processo ha avuto luogo senza interrompere lo User Agent minimamente.

4.6 Associate – Smart mode

La procedura come vista finora richiede un utilizzo massiccio della rete. Inoltre anche le altre risorse del server come la CPU e la Memoria sono sovraccaricate dal lavoro che devono fare, poiché per ogni richiesta di autenticazione devono generare una chiave segreta.

Se il consumer (Bob) potrebbe ricordare anche per un po di che cosa stava parlando, egli può risparmiare a Carol un sacco di lavoro ed ad entrambi molta banda. L'idea base è semplice: in modalità

“dumb", il consumer (Bob) semplicemente riporta indietro la handle (che non tiene traccia dello stato - stateless) che gli è stato inviato da Carol . Ma cosa succederebbe se Bob potrebbe effettivamente ricordasse il segreto condiviso con Carol utilizzandolo e riutilizzandolo questo segreto condiviso per un lasso di tempo ogni volta che ha bisogno di parlare con Carol?

Questa è la modalità “**Smart**”

Il consumer (Bob) stabilisce un segreto condiviso con l’OpenID server di Carol in anticipo sui tempi e ricordandolo questo segreto per qualche periodo di tempo ragionevole. Egli stabilisce il segreto condiviso inviando una richiesta POST al server di Carol,

http://carol.example.com/openid-server.cgi con questi parametri:

- **openid.mode** = associate Questo indica a Carol che vuole condividere una chiave segreta con lei.

- **openid.assoc_type** = HMAC-SHA1

+ The type of secret he wants to share; only HMAC-SHA1 is currently supported.

- openid.session_type = *blank* Indica come il segreto deve essere stabilito, un valore vuoto significa in chiaro il che non è 100% sicuro. "DH-SHA1" indica che verrà usato il protocollo Diffie – Hellman per lo scambio delle chiavi

Quando Bob effettua questo scambio? Può essere fatto la prima volta che qualcuno chiede di autenticare il suo identifier con Carol, o se tale chiave segreta è scaduta. Ma la cosa importante da notare qui è che, questo processo è complettamente indipendente ad ogni richiesta di autenticazione fatta, ache se può essere attivata da una tale richiesta.

Una volta che Bob ha inviato il suo messaggio di “associate request”, Carol genererà una chiave segreta e condivisa (come avrebbe fatto anche prima) e lo aggiunge a “assoc_handle”. Diversamente dalla modalità “dumb” in questa modalita questo handle è un handle stateful, ovvero tiene traccia dello stato della conversazione.Carol quindi risponde a Bob con un documento * chiave = valore * formattato in risposta alla sua richiesta POST che contiene tutti i normali parametri vari. I due parametri magici, tuttavia, sono il * assoc_handle * e * mac_key * (o * enc_mac_key * se è stato utilizzato Diffie-Hellman). Con questi due elementi, Bob ora può tenere traccia, internamente, a tale handle e la chiave segreta condivisa.

Ma a che servono questi due valori ? Ora, ogni volta che Bob ha bisogno di autenticare l'identità con Carol, egli invierà la handle con il messaggio "checkid_setup" o "checkid_immediate" invece di aspetta da Carol la generazione della handle ogni volta. Questo consente a Carol di risparmiare del lavoro. Una volta Bob ottiene una risposta firmata da Carol (supponendo un'affermazione positiva), egli non ha più bisogno di inviare una richiesta "check_authentication" perché sa gia la chiave segreta condiviso che è stata utilizzata per firmare l'affermazione e lo può controllare da solo. Prima Bob doveva verificare con Carol perché non aveva nessuno modo di sapere se la firma utilizzata è era autentica (di Carol) oppure falsa e quindi doveva interagire con Carol per tale controllo. Ma ora poiché Bob sa già la chiave segreta utilizzata per la firma può controllare lui da solo la validità della firma.

5 Flusso del protocollo