HashCookies – A Simple Recipe

J. Fitzpatrick

7th May 2009

MWR InfoSecurity

MWR INFOSECURITY

Contents

MWR INFOSECURITY

1 Abstract

Since HTTP is stateless it utilises sessions in order to track a user's state when using web based applications. Several vectors exist which could permit an attacker to gain access to a user's session resulting in the compromise of the user's account or other sensitive information. The use of a changing and expiring session ID can enable a user's session to be protected from a number of attacks. By transmitting a random salt to a web browser the web browser is able to use this salt in order to generate a new cookie by hashing information which only the web browser and web server know; this cookie is a HashCookie. Provided the salt is protected during the initial exchange, or an attacker is not in a position to intercept this communication, then in all instances even if an attacker is able to obtain a valid session ID for a user of a web based application the use of HashCookies would provide them no leverage over the user's session. Implementation requires HashCookie support from both the web browser and web server.

MWR INFOSECURITY

2 Introduction

Session hijacking is a risk faced by users of web based applications. A variety of vectors exist which could be exploited to expose users' session IDs and so provide an attacker with full access to those users' sessions. These vectors include Cross Site Scripting (XSS), session fixation, weak session identifiers and any clear text transmission of session identifiers.

Equally however, methods are available to help prevent session hijacking attacks succeeding; for example, the SecureSessionModule within the ASP.NET framework mitigates the risk of session hijacking somewhat by tying a session ID to an IP address, or rather to a network range. This prevents a session ID being used by an attacker, unless the attacker was able to send requests which originated from the targeted user's IP address range. The extensive use of NAT and proxies in current environments means that this is not a safe assumption to make. The need to bind a session ID to a network range rather than a single IP address (in order to accommodate instances when traffic is load balanced and so switches between IP addresses) increases the surface area for successful attacks. HashCookies ensure that even when an IP address changes a session to the web application can still be maintained.

Setting a new cookie after each request can be used to prevent an attacker who has been able to recover a previously used session ID from hijacking the session to which it corresponds. However, an attacker who was able to obtain a valid session would still be able to exploit this functionality in order to hijack the session and render it unusable by the legitimate user. It would simply be a race between the user and the attacker to use the session ID first.

HashCookies address the issues described above, although their implementation would place additional requirements upon both the web browser and the web server.

If session cookies are constantly changing, an attacker who was able to intercept a session ID by whatever means would not be able to use it in order to hijack a session. Obtaining a session ID through XSS, for example, would only provide the attacker with the session ID which had been used to identify the user on their last request to the web server. Since this has already been used it would no longer be valid and so will be of no use to an attacker seeking to gain access to a user's session.

MWR INFOSECURITY

3 Implementation

HashCookies make use of three values:

- Session ID – this is a static value which identifies a user in the same manner that a traditional session ID would identify a user.
- Salt – this is a secret value which should be known only by the client and the server. It is used to generate IDs which are not known to any other party.
- Sequence number – since web browsers are capable of making multiple requests to a web server these may arrive out of order and so it is important that the web server knows the correct order of requests.

The three values described above are used by the web browser in order to generate a HashCookie.

3.1 Client Implementation

In order to identify to a web server that a browser supports HashCookies it will specify this in the Accept section of the HTTP header used in a HTTP request:

```
GET / HTTP/1.0
Host: www.mwrinfosecurity.com
User-Agent: Mozilla/5.0 (X11; U; Linux i686; en-GB; rv:1.9.0.3) Gecko/2008092816
Firefox/2.0.0.6 (Debian-3.0.3-3)
Accept: text/html,application/xml;q=0.9,*/*;q=0.8,hash-cookie
Accept-Language: en-gb,en;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Proxy-Connection: keep-alive
```

On receiving the request the server will set the appropriate cookie as usual, however, will add an additional flag which contains the salt to be used by the HashCookie:

```
HTTP/1.1 200 OK
Date: Thu, 04 Dec 2008 17:37:29 GMT
Server: server
Expires: Thu, 19 Nov 1981 08:52:00 GMT
Cache-Control: no-store, no-cache, must-revalidate, post-check=0, pre-check=0
Set-Cookie: SESSION=cb58609ecb4b8f5b4fd1235c7bd60aeb; path=/; HttpOnly;
salt=ea043ecb41517205154ddf8c658b6d0961c17fe3
Pragma: no-cache
Content-Length: 4347
Keep-Alive: timeout=5, max=100
Connection: Keep-Alive
Content-Type: text/html
```

Ideally, the exchange of the salt should be performed over an SSL channel in order to ensure maximum security. However, even if SSL were not used for this exchange, HashCookies would still provide protection from session hijacking attacks which do not depend on the capture of clear text traffic from the network. It is vital for the security of HashCookies that the salt should be of a length that cannot feasibly be determined through brute forcing. During communication the sequence number will be exposed and so should be considered to be known by any attacker.

When a client receives the session ID and the salt they can use the SHA-1 algorithm to hash the salt, current session ID and the sequence number in order to generate a HashCookie which can then be used in the next request.

```
hashCookie = sha1(currentSessionID+salt+sequenceNumber)
```

The subsequent request utilising HashCookies would appear as shown in the following request:

```
GET /nextPage.mwr HTTP/1.0
Host: www.mwrinfosecurity.com
User-Agent: Mozilla/5.0 (X11; U; Linux i686; en-GB; rv:1.9.0.3) Gecko/2008092816
Firefox/2.0.0.6 (Debian-3.0.3-3)
Accept: text/html,application/xml;q=0.9,*/*;q=0.8,hash-cookie
Accept-Language: en-gb,en;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Proxy-Connection: keep-alive
Cookie: SESSION=cb58609ecb4b8f5b4fd1235c7bd60aeb-
a29befed094761ea3dfa9e9de164b5fdfbc7d6a9-1
```

As soon as a HashCookie has been used any future requests which use the same HashCookie should be rejected by the web server. If this is not the case then the use of HashCookies will offer no additional security benefits over a standard session ID.

Since a web browser may make multiple concurrent requests to a web server these may arrive out of order and it is therefore important that HashCookies can handle this. The solution proposed for this is to pass the sequence number in addition to the session ID and HashCookie (the final "-1" as shown in the output above). This will allow the HashCookies to be associated with the sequence number which was used to construct them. It will also then be possible to specify a window of valid sequence numbers; a sequence number would then need to be within this window in order to be accepted by the server.

The reason for transmitting both the initial session ID and the HashCookie (rather than using solely the HashCookie as the session ID) is twofold. Firstly, as nothing changes in the initial session ID, this simplifies session identification; all lookups of HashCookie data can simply be performed against the session ID which could now be treated as a form of key field. Secondly, it would be highly inefficient to rotate the session ID each time as this would require the server to compute the next expected cookie values for every request. This could lead to a collision between HashCookies, although it is acknowledged that this would be improbable. Identifying sessions would become further complicated when multiple concurrent requests need to be considered and a lookup based on the information provided would quickly prove to be not only an unnecessarily expensive task, but one which could potentially lead to the inadvertent hijacking of a user's session should a collision occur. Although this is considered an unlikely occurrence, as it would introduce the potential for the very vulnerability which HashCookies attempt to mitigate, the risk is considered unacceptable. The use of a static session ID is therefore considered essential the implementation.

3.2 Server Implementation

Appropriate configurations settings will need to be applied server side when HashCookies are in use. For instance, it will be necessary to define what action is taken for a user session if an invalid HashCookie is passed: should the session be terminated, or should the request simply be ignored? Terminating a session because a session ID had been replayed would add to the security of an application but could in some instances cause issues if an old request was replayed. Additionally, legitimate requests arriving out of order should be accepted rather than rejected. A representation of how memory could be managed with regards to HashCookies is given below:

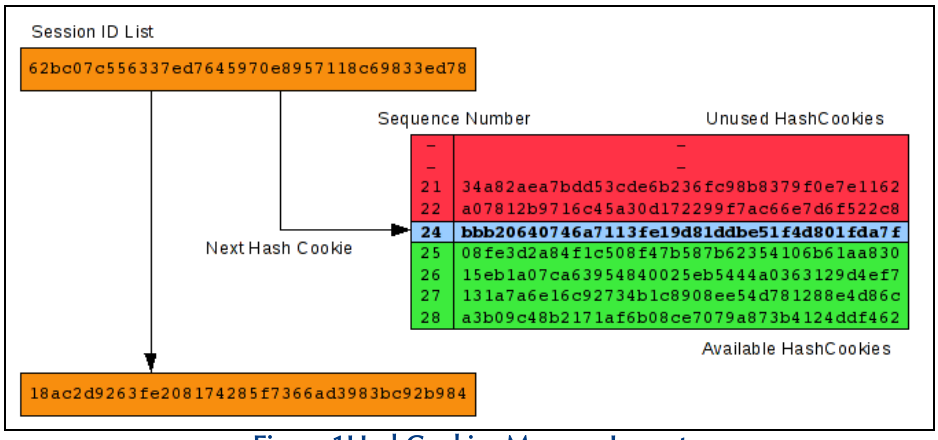

Figure 1 HashCookies Memory Layout

In this model, each session ID has a window of HashCookies associated with it, as shown by the red, blue and green shaded area of the diagram. When a request is made to a server the available HashCookies section of the HashCookies window associated with that session ID would be consulted. If the HashCookie presented in the HTTP request falls within the available section of this window, then the request is honoured. The Next HashCookie pointer then increments to point to the HashCookie with sequence number one greater than the HashCookie just used and the just used HashCookie is removed from the window completely. Any HashCookies in the Available window with a sequence number less than that of the HashCookie just used are moved to the unused HashCookies section of the window. If the unused section is full the oldest HashCookie is removed. Unused HashCookies are therefore any HashCookies which have not yet been used by the client and whose sequence number is less than that of the Next HashCookie.

For example, if the Next HashCookie used had sequence number 26 then the HashCookie associated with sequence number 26 would be removed from the window so that it could not be used again. HashCookies 25 and 24 would then be shifted into the unused HashCookies container and HashCookie 27 set as the Next HashCookie. The request would then be honoured and any response returned.

It is important for the security of this model that the size of the unused HashCookies container and the number of available HashCookies are clearly defined. In addition, the permitted variance in sequence number either side of the next sequence number must also be explicitly set. This should be no more than the size of the unused and available HashCookies containers respectively. Consequently, a check will be required to ensure that the variance between the Next HashCookie value and the presented HashCookie value falls within these parameters for every request received.

If the HashCookie presented is not found in the Available HashCookies section of the window then the Unused HashCookies section is consulted. Unused HashCookies are also honoured, but there are differences in operation if an unused HashCookie is received. In such cases, no change to the Next HashCookie pointer occurs and so no HashCookies are added to the unused HashCookies container. The HashCookie which has just been used is still removed from the window so that it cannot be reused. In the diagram above the HashCookies with sequence number 21 and 22 are unused and so would be accepted by the server. For example, if the HashCookie associated with 21 was presented it would be checked that the sequence number (21) falls within the accepted variance from the sequence number of the Next HashCookie value (24). If this was the case, the HashCookie with sequence number 21 would be removed from the window and the request would be honoured. If the acceptable variance was set at 2 then 21 would not fall within this acceptable variance and so the request with this sequence number would be rejected.

MWR INFOSECURITY

4 Security Benefits of HashCookies

If HashCookies were used then even if a session token was obtained it would not be possible for anyone else to hijack the associated session unless they also knew the salt provided by the server. It is therefore essential that the salt is afforded the greatest level of protection possible. Any attacker would also need a valid sequence number, although, as sequence numbers are not protected, this should not be seen as an additional layer of security.

It is important to ensure that HashCookies do not introduce security weaknesses and careful implementation is required to ensure that this does not happen.

MWR INFOSECURITY

5 Considerations

Any implementation of HashCookies will require support by both the server and the client (although it is expected that the majority of any changes would be server side). Consequently, any implementation will place an increased load on the server, although it is not expected that this would be significant in normal operation. Nevertheless, it is important that this functionality should not be vulnerable to manipulation by an attacker seeking to exploit denial of service opportunities.

This paper is intended as an introduction to the concept of HashCookies rather than an in depth technical analysis. However, the manner in which the concept has been designed to ensure its resilience against one such form of attack is discussed here, both to illustrate some possibly less obvious design choices, and to highlight the importance of ensuring that the security benefits of this concept cannot be undermined by poor implementation.

It is important that the use of HashCookies should not allow a malicious client to cause the server to compute a large number of HashCookies. Essentially, it is important that the server places an upper bound on the number of HashCookies which can be generated and that this limit cannot be circumvented (for example, by submitting a request with a high sequence number).

In the early designs the intention was to rehash a session cookie before each request, no static session ID was used and so the session ID was the HashCookie. This approach was entirely adequate for single threaded environments, but it quickly became apparent that there could be problems with multi threaded browsers making multiple requests which may arrive out of order and so sequence numbers were introduced. With HashCookies containing sequence numbers, if a HashCookie arrived with a sequence number more than 1 greater than the last request received, then the server would simply hash the session cookie and then the result of this hashing and then check that the HashCookie actually received matched this. In other words the hashing process was iterative; to obtain the value of a HashCookie five down the line it would require five hashes to computed. However, this simple approach could allow a malicious user to submit a request with a sequence number millions greater than the current sequence number – this would involve the computation of all the intervening HashCookie values and so could be exploited to exhaust the host's resources and cause a Denial of Service condition.

This situation could be relatively easily addressed by placing a limit on requests such that only sequence numbers, for instance, 16 greater than the last valid sequence number should be accepted. However, instead of introducing mitigating controls for what was essentially a design flaw, the issue was addressed head on and the concept redesigned. It is critical that flaws such as this are not introduced in to server side implementations of HashCookies. Many existing server technologies already use hashing for functionality such as CSRF protection and it is expected that the impact which HashCookies would place on a server would be negligible compared to this. However, flawed implementations of this security feature could prevent the widespread adoption of HashCookies and so it is important that in any proposed implementation careful consideration is given to these types of attack and the manner in which malicious users could seek to circumvent or exploit this feature.

MWR INFOSECURITY

6 Summary and Future Work

Currently, I am working on a Firefox plugin to handle the client side requirements of HashCookies. Others are working on a Ruby on Rails plugin which will form the basis of a server side solution. These should be available in the very near future so that they can be tested for effectiveness and robustness.

All feedback and comments on this concept are greatly appreciated, whether they are positive or constructively negative.

Email
john.fitzpatrick[at]mwrinfosecurity[dot]com

MWR InfoSecurity
St. Clement House
1-3 Alencon Link
Basingstoke, RG21 7SB
Tel: +44 (0)1256 300920
Fax: +44 (0)1256 844083
mwrinfosecurity.com
labs.mwrinfosecurity.com