Independent Submission
Request for Comments: 6896
Category: Informational
ISSN: 2070-1721

S. Barbato
S. Dorigotti
T. Fossati, Ed.
KoanLogic

March 2013

SCS: KoanLogic's Secure Cookie Sessions for HTTP

Abstract

This memo defines a generic URI and HTTP-header-friendly envelope for carrying symmetrically encrypted, authenticated, and origin-timestamped tokens. It also describes one possible usage of such tokens via a simple protocol based on HTTP cookies.

Secure Cookie Session (SCS) use cases cover a wide spectrum of applications, ranging from distribution of authorized content via HTTP (e.g., with out-of-band signed URIs) to securing browser sessions with diskless embedded devices (e.g., Small Office, Home Office (SOHO) routers) or web servers with high availability or load-balancing requirements that may want to delegate the handling of the application state to clients instead of using shared storage or forced peering.

Status of This Memo

This document is not an Internet Standards Track specification; it is published for informational purposes.

This is a contribution to the RFC Series, independently of any other RFC stream. The RFC Editor has chosen to publish this document at its discretion and makes no statement about its value for implementation or deployment. Documents approved for publication by the RFC Editor are not a candidate for any level of Internet Standard; see Section 2 of RFC 5741.

Information about the current status of this document, any errata, and how to provide feedback on it may be obtained at http://www.rfc-editor.org/info/rfc6896.

Table of Contents

1. Introduction

This memo defines a generic URI and HTTP-header-friendly envelope for carrying symmetrically encrypted, authenticated, and origin-timestamped tokens.

It is generic in that it does not force any specific format upon the authenticated information, which makes SCS tokens flexible, easy, and secure to use in many different scenarios.

It is URI and HTTP header friendly, as it has been explicitly designed to be compatible with both the ABNF "token" syntax [RFC2616] (the one used for, e.g., Set-Cookie and Cookie headers) and the path or query syntax of HTTP URIs.

This memo also describes one possible usage of such tokens via a simple protocol based on HTTP cookies that allows the establishment of "client mode" sessions. This is not their sole possible use. While no other operational patterns are outlined here, it is expected that SCS tokens may be easily employed as a building block for other types of HTTP-based applications that need to carry in-band secured information.

When SCS tokens are used to implement client-mode cookie sessions, the SCS implementer must fully understand the security implications entailed by the act of delegating the whole application state to the client (browser). In this regard, some hopefully useful security considerations have been collected in Section 7.2. However, please note that they may not cover all possible scenarios; therefore, they must be weighed carefully against the specific application threat model.

An SCS server may be implemented within a web application by means of a user library that exposes the core SCS functionality and leaves explicit control over SCS tokens to the programmer, or transparently, by hiding a "diskless session" facility behind a generic session API abstraction, for example. SCS implementers are free to choose the model that best suits their needs.

2. Requirements Language

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [RFC2119].

3. SCS Protocol

The SCS protocol defines:

- the SCS cookie structure and encoding (Section 3.1);
- the cryptographic transformations involved in SCS cookie creation and verification (Section 3.2);
- the HTTP-based PDU exchange that uses the Set-Cookie and Cookie HTTP headers (Section 3.3);
- the underlying key management model (Section 4).

Note that the PDU is transmitted to the client as an opaque data block; hence, no interpretation nor validation is necessary. The single requirement for client-side support of SCS is cookie activation on the user agent. The origin server is the sole actor involved in the PDU manipulation process, which greatly simplifies the crypto operations -- especially key management, which is usually a pesky task.

In the following sections, we assume S to be one or more interchangeable HTTP server entities (e.g., a server pool in a load-balanced or high-availability environment) and C to be the client with a cookie-enabled browser or any user agent with equivalent capabilities.

3.1. SCS Cookie Description

S and C exchange a cookie (Section 3.3) whose cookie value consists of a sequence of adjacent non-empty values, each of which is the 'URL and Filename safe' Base64 encoding [RFC4648] of a specific SCS field.

(Hereafter, the encoded and raw versions of each SCS field are distinguished based on the presence, or lack thereof, of the 'e' prefix in their name, e.g., eATIME and ATIME.)

Each SCS field is separated by its left and/or right sibling by means of the %x7c ASCII character (i.e., '|'), as follows:

```
scs-cookie        = scs-cookie-name "=" scs-cookie-value
scs-cookie-name   = token
scs-cookie-value  = eDATA "|" eATIME "|" eTID "|" eIV "|" eAUTHTAG
eDATA             = 1*base64url-character
eATIME            = 1*base64url-character
eTID              = 1*base64url-character
eIV               = 1*base64url-character
eAUTHTAG          = 1*base64url-character
```

Figure 1

Confidentiality is limited to the application-state information (i.e., the DATA field), while integrity and authentication apply to the entire cookie value.

The following subsections describe the syntax and semantics of each SCS cookie field.

3.1.1. ATIME

Absolute timestamp relating to the last read or write operation performed on session DATA, encoded as a HEX string holding the number of seconds since the UNIX epoch (i.e., since 00:00:00, Jan 1 1970).

This value is updated with each client contact and is used to identify expired sessions. If the delta between the received ATIME value and the current time on S is larger than a predefined "session_max_age" (which is chosen by S as an application-level parameter), a session is considered to be no longer valid, and is therefore rejected.

Such an expiration error may be used to force user logout from an SCS-cookie-based session, or hooked in the web application logic to display an HTML form requiring revalidation of user credentials.

3.1.2. DATA

Block of encrypted and optionally compressed data, possibly containing the current session state. Note that no restriction is imposed on the cleartext structure: the protocol is completely agnostic as to inner data layout.

Generally speaking, the plaintext is the "normal" cookie that would have been exchanged by S and C if SCS had not been used.

3.1.3. TID

This identifier is equivalent to a Security Parameter Index (SPI) in a Data Security SA [RFC3740]) and consists of an ASCII string that uniquely identifies the transform set (keys and algorithms) used to generate this SCS cookie.

SCS assumes that a key-agreement/distribution mechanism exists for environments in which S consists of multiple servers that provide a unique external identifier for each transform set shared amongst pool members.

Such a mechanism may safely downgrade to a periodic key refresh, if there is only one server in the pool and the key is generated in place -- i.e., it is not handled by an external source.

However, when many servers act concurrently upon the same pool, a more sophisticated protocol, whose specification is out of the scope of the present document, must be devised (ideally, one that is able to handle key agreement for dynamic peer groups in a secure and efficient way, e.g., [CLIQUES] or [Steiner]).

3.1.4. IV

Initialization Vector used for the encryption algorithm (see Section 3.2).

In order to avoid providing correlation information to a possible attacker with access to a sample of SCS cookies created using the same TID, the IV MUST be created randomly for each SCS cookie.

3.1.5. AUTHTAG

Authentication tag that is based on the plain string concatenation of the base64url-encoded DATA, ATIME, TID, and IV fields and is framed by the "|" separator (see also the definition of the Box() function in Section 3.2):

```
AUTHTAG = HMAC(base64url(DATA)  "|"
               base64url(ATIME) "|"
               base64url(TID)   "|"
               base64url(IV))
```

Note that, from a cryptographic point of view, the "|" character provides explicit authentication of the length of each supplied field, which results in a robust countermeasure against splicing attacks.

3.2. Crypto Transform

SCS could potentially use any combination of primitives capable of performing authenticated encryption. In practice, an encrypt-then-MAC approach [Kohno] with encryption utilizing the Cipher Block Chaining (CBC) mode and Hashed Message Authentication Code (HMAC) [RFC2104] authentication was chosen.

The two algorithms MUST be associated with two independent keys.

The following conventions will be used in the algorithm description (Sections 3.2.5 and 3.2.6):

- Enc/Dec(): block encryption/decryption functions (Section 3.2.2);
- HMAC(): authentication function (Section 3.2.2);
- Comp/Uncomp(): compression/decompression functions (Section 3.2.3);
- e/d(): cookie-value encoding/decoding functions (Section 3.2.4);
- RAND(): random number generator [RFC4086];
- Box(): string boxing function. It takes an arbitrary number of base64url-encoded strings and returns the string obtained by concatenating each input in the exact order in which they are listed, separated by the "|" char. For example:

  Box("akxI", "MTM", "Hadvo") = "akxI|MTM|Hadvo".

3.2.1. Choice and Role of the Framing Symbol

Note that the adoption of "|" as the framing symbol in the Box() function is arbitrary: any char allowed by the cookie-value ABNF in [RFC6265] is safe to be used as long it has empty intersection with the base64url alphabet.

It is also worth noting that the role of the framing symbol, which provides an implicit length indicator for each of the atoms, is key to the accuracy and security of SCS.

This is especially relevant when the authentication tag is computed (see Section 3.1.5). More specifically, the explicit inclusion of the framing symbol within the HMAC input seals the integrity of the blob as a whole together with each of its composing atoms in their exact position.

This feature makes the protocol robust against attacks aimed at disrupting the security of SCS PDUs by freely moving boundaries between adjacent atoms.

3.2.2. Cipher Set

Implementers MUST support at least the following algorithms:

- AES-CBC-128 for encryption [NIST-AES];
- HMAC-SHA1 with a 128-bit key for authenticity and integrity,

which appear to be sufficiently secure in a broad range of use cases ([Bellare] [RFC6194]), are widely available, and can be implemented in a few kilobytes of memory, providing an extremely valuable feature for constrained devices.

One should consider using larger cryptographic key lengths (192- or 256-bit) according to the actual security and overall system performance requirements.

3.2.3. Compression

Compression, which may be useful or even necessary when handling large quantities of data, is not compulsory (in such a case, Comp/Uncomp is replaced by an identity matrix). If this function is enabled, the DEFLATE [RFC1951] format MUST be supported.

Some advice to SCS users: compression should not be enabled when handling relatively short and entropic state, such as pseudorandom session identifiers. Instead, large and quite regular state blobs could get a significant boost when compressed.

3.2.4. Cookie Encoding

SCS cookie values MUST be encoded using the alphabet that is URL and filename safe (i.e., base64url) defined in Section 5 of Base64 [RFC4648]. This encoding is very widespread, falls smoothly into the encoding rules defined in Section 4.1.1 of [RFC6265], and can be safely used to supply SCS-based authorization tokens within a URI (e.g., in a query string or straight into a path segment).

3.2.5. Outbound Transform

The output data transformation, as seen by the server (the only actor that explicitly manipulates SCS cookies), is illustrated by the pseudocode in Figure 2.

1. IV := RAND()
2. ATIME := NOW
3. DATA := Enc(Comp(plain-text-cookie-value), IV)
4. AUTHTAG := HMAC(Box(e(DATA), e(ATIME), e(TID), e(IV)))

Figure 2

A new Initialization Vector is randomly picked (step 1). As previously mentioned (Section 3.1.4), this step is necessary to avoid providing correlation information to an attacker.

A new ATIME value is taken as the current timestamp according to the server clock (step 2).

Since the only user of the ATIME field is the server, it is unnecessary for it to be synchronized with the client -- though it needs to use a fairly stable clock. However, if multiple servers are active in a load-balancing configuration, clocks SHOULD be synchronized to avoid errors in the calculation of session expiry.

The plaintext cookie value is then compressed (if needed) and encrypted by using the key-set identified by TID (step 3).

If the length of (compressed) state is not a multiple of the block size, its value MUST be filled with as many padding bytes of equal value as the pad length -- as defined by the scheme given in Section 6.3 of [RFC5652].

Then, the authentication tag, which encompasses each SCS field (along with lengths and relative positions), is computed by HMAC'ing the "|"-separated concatenation of their base64url representations using the key-set identified by TID (step 4).

Finally, the SCS-cookie-value is created as follows:

```
scs-cookie-value = Box(e(DATA), e(ATIME), e(TID), e(IV),
                       e(AUTHTAG))
```

3.2.6. Inbound Transform

The inbound transformation is described in Figure 3. Each of the 'e'-prefixed names shown has to be interpreted as the base64url-encoded value of the corresponding SCS field.

```
0.  If (split_fields(scs-cookie-value) == ok)
1.      tid' := d(eTID)
2.      If (tid' is available)
3.          tag' := d(eAUTHTAG)
4.          tag := HMAC(Box(eDATA, eATIME, eTID, eIV))
5.          If (tag = tag')
6.              atime' := d(eATIME)
7.              If (NOW - atime' <= session_max_age)
8.                  iv' := d(eIV)
                    data' := d(eDATA)
9.                  state := Uncomp(Dec(data', iv'))
10.             Else discard PDU
11.         Else discard PDU
12.     Else discard PDU
13. Else discard PDU
```

Figure 3

First, the inbound scs-cookie-value is broken into its component fields, which MUST be exactly 5, and each at least the minimum length specified in Figure 3 (step 0). In case any of these preliminary checks fails, the PDU is discarded (step 13); else, TID is decoded to allow key-set lookup (step 1).

If the cryptographic credentials (encryption and authentication algorithms and keys identified by TID) are unavailable (step 12), the inbound SCS cookie is discarded since its value has no chance to be interpreted correctly. This may happen for several reasons: e.g., if a device without storage has been reset and loses the credentials stored in RAM, if a server pool node desynchronizes, or in case of a key compromise that forces the invalidation of all current TIDs, etc.

When a valid key-set is found (step 2), the AUTHTAG field is decoded (step 3) and the (still) encoded DATA, ATIME, TID, and IV fields are supplied to the primitive that computes the authentication tag (step 4).

If the tag computed using the local key-set matches the one carried by the supplied SCS cookie, we can be confident that the cookie carries authentic material; otherwise, the SCS cookie is discarded (step 11).

Then the age of the SCS cookie (as deduced by ATIME field value and current time provided by the server clock) is decoded and compared to the maximum time-to-live (TTL) defined by the session_max_age parameter.

If the "age" check passes, the DATA and IV fields are finally decoded (step 8), so that the original plaintext data can be extracted from the encrypted, and optionally compressed, blob (step 9).

Note that steps 5 and 7 allow any altered packets or expired sessions to be discarded, hence avoiding unnecessary state decryption and decompression.

3.3. PDU Exchange

SCS can be modeled in the same manner as a typical store-and-forward protocol in which the endpoints are S, consisting of one or more HTTP servers and the client C, an intermediate node used to "temporarily" store the data to be successively forwarded to S.

In brief, S and C exchange an immutable cookie data block (Section 3.1): the state is stored on the client at the first hop and then restored on the server at the second, as in Figure 4.

1. dump-state:
   S --> C
       Set-Cookie: ANY_COOKIE_NAME=KrdPagFes_5ma-ZUluMsww|MTM0...
          Expires=...; Path=...; Domain=...;

2. restore-state:
   C --> S
       Cookie: ANY_COOKIE_NAME=KrdPagFes_5ma-ZUluMsww|MTM0...

Figure 4

3.3.1. Cookie Attributes

In the following subsections, a series of recommendations is provided in order to maximize SCS PDU fitness in the generic cookie ecosystem.

3.3.1.1. Expires

If an SCS cookie includes an Expires attribute, then the attribute MUST be set to a value consistent with session_max_age.

For maximum compatibility with existing user agents, the timestamp value MUST be encoded in rfc1123-date format, which requires a 4-digit year.

3.3.1.2. Max-Age

Since not all User Agents (UAs) support this attribute, it MUST NOT be present in any SCS cookie.

3.3.1.3. Domain

SCS cookies MUST include a Domain attribute compatible with application usage.

A trailing '.' MUST NOT be present in order to minimize the possibility of a user agent ignoring the attribute value.

3.3.1.4. Secure

This attribute MUST always be asserted when SCS sessions are carried over a Transport Layer Security (TLS) channel.

3.3.1.5. HttpOnly

This attribute SHOULD always be asserted.

4. Key Management and Session State

This specification provides some common recommendations and practices relevant to cryptographic key management.

In the following, the term 'key' references both encryption and HMAC keys.

- The key SHOULD be generated securely following the randomness recommendations in [RFC4086];
- the key SHOULD only be used to generate and verify SCS PDUs;
- the key SHOULD be replaced regularly as well as any time the format of SCS PDUs or cryptographic algorithms changes.

Furthermore, to preserve the validity of active HTTP sessions upon renewal of cryptographic credentials (whenever the value of TID changes), an SCS server MUST be capable of managing at least two transforms contemporarily: the currently instantiated one and its predecessor.

Each transform set SHOULD be associated with an attribute pair, "refresh" and "expiry", which is used to identify the exposure limits (in terms of time or quantity of encrypted and/or authenticated bytes, etc.) of related cryptographic material.

In particular, the "refresh" attribute specifies the time limit for substitution of transform set T with new material T'. From that moment onwards, and for an amount of time determined by "expiry", all new sessions will be created using T', while the active T-protected ones go through a translation phase in which:

- the inbound transformation authenticates and decrypts/decompresses using T (identified by TID);
- the outbound transformation encrypts/compresses and authenticates using T'.

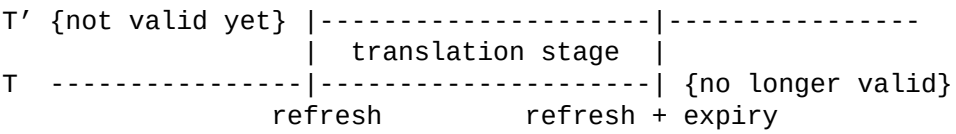

Figure 5

As shown in Figure 5, the duration of the HTTP session MUST fit within the lifetime of a given transform set (i.e., from creation time until "refresh" + "expiry").

In practice, this should not be an obstacle because the longevity of the two entities (HTTP session and SCS transform set) should differ by one or two orders of magnitude.

An SCS server may take this into account by determining the duration of a session adaptively according to the expected deletion time of the active T, or by setting the "expiry" value to at least the maximum lifetime allowed by an HTTP session.

Since there is also only one refresh attribute in situations with more than one key (e.g., one for encryption and one for authentication) within the same T, the smallest value is chosen.

It is critical for the correctness of the protocol that in case multiple equivalent SCS servers are used in a pool, all of them share the same view of time (see also Section 3.2.5) and keying material.

As far as the latter is concerned, SCS does not mandate the use of any specific key-sharing mechanism, and will keep working correctly as long as the said mechanism is able to provide a single, coherent view of the keys shared by pool members -- while conforming to the recommendations given in this section.

5. Cookie Size Considerations

In general, SCS cookies are bigger than their plaintext counterparts. This is due to the following reasons:

- inflation of the Base64 encoding of state data (approximately 1.4 times the original size, including the encryption padding);
- the fixed size increment (approximately 80/90 bytes) caused by SCS fields and framing overhead.

While the former is a price the user must always pay proportionally to the original data size, the latter is a fixed quantum, which can be huge on small amounts of data but is quickly absorbed as soon as data becomes big enough.

The following table compares byte lengths of SCS cookies (with a four-byte TID) and corresponding plaintext cookies in a worst-case scenario, i.e., when no compression is in use (or applicable).

plain	SCS
11	128
102	256
285	512
651	1024
1382	2048
2842	4096

The largest uncompressed cookie value that can be safely supplied to SCS is about 2.8 KB.

6. Acknowledgements

We would like to thank Jim Schaad, David Wagner, Lorenzo Cavallaro, Willy Tarreau, Tobias Gondrom, John Michener, Sean Turner, Barry Leiba, Robert Sparks, Stephen Farrell, Stewart Bryant, and Nevil Brownlee for their valuable feedback on this document.

7. Security Considerations

7.1. Security of the Cryptographic Protocol

From a cryptographic architecture perspective, the described mechanism can be easily traced to an "encode then encrypt-then-MAC" scheme (Encode-then-EtM) as described in [Kohno].

Given a "provably-secure" encryption scheme and MAC (as for the algorithms mandated in Section 3.2.2), the authors of [Kohno] demonstrate that their composition results in a secure authenticated encryption scheme.

7.2. Impact of the SCS Cookie Model

The fact that the server does not own the cookie it produces, gives rise to a series of consequences that must be clearly understood when one envisages the use of SCS as a cookie provider and validator for his/her application.

In the following subsections, a set of different attack scenarios (together with corresponding countermeasures where applicable) are identified and analyzed.

7.2.1. Old Cookie Replay

SCS doesn’t address replay of old cookie values.

In fact, there is nothing that assures an SCS application about the client having returned the most recent version of the cookie.

As with "server-side" sessions, if an attacker gains possession of a given user’s cookies -- via simple passive interception or another technique -- he/she will always be able to restore the state of an intercepted session by representing the captured data to the server.

The ATIME value, along with the session_max_age configuration parameter, allows SCS to mitigate the chances of an attack (by forcing a time window outside of which a given cookie is no longer valid) but cannot exclude it completely.

A countermeasure against the "passive interception and replay" scenario can be applied at transport/network level using the anti-replay services provided by e.g., Secure Socket Layer/Transport Layer Security (SSL/TLS) [RFC5246] or IPsec [RFC4301].

A native solution is not in scope with the security properties inherent to an SCS cookie. Hence, an application wishing to be replay-resistant must put in place some ad hoc mechanism to prevent clients (both rogue and legitimate) from (a) being able to replay old cookies as valid credentials and/or (b) getting any advantage by replaying them.

The following illustrate some typical use cases:

- Session inactivity timeout scenario (implicit invalidation): use the session_max_age parameter if a global setting is viable, else place an explicit TTL in the cookie (e.g., validity_period="start_time, duration") that can be verified by the application each time the client presents the SCS cookie.

- Session voidance scenario (explicit invalidation): put a randomly chosen string into each SCS cookie (cid="$(random())") and keep a list of valid session cids against which the SCS cookie presented by the client can be checked. When a cookie needs to be invalidated, delete the corresponding cid from the list. The described method has the drawback that, in case a non-permanent storage is used to archive valid cids, a reboot/restart would invalidate all sessions (it can't be used when |S| > 1).

- One-shot transaction scenario (ephemeral): this is a variation on the previous theme when sessions are consumed within a single request/response. Put a nonce="$(random())" within the state information and keep a list of not-yet-consumed nonces in RAM. Once the client presents its cookie credential, the embodied nonce is deleted from the list and will be therefore discarded whenever replayed.

- TLS binding scenario: the server application must run on TLS, be able to extract information related to the current TLS session, and store it in the DATA field of the SCS cookie itself [RFC5056]. The establishment of this secure channel binding prevents any third party from reusing the SCS cookie, and drops its value altogether after the TLS session is terminated -- regardless of the lifetime of the cookie. This approach suffers a scalability problem in that it requires each SCS session to be handled by the same client-server pair. However, it provides a robust model and an affordable compromise when security of the session is exceptionally valuable (e.g., a user interacting with his/her online banking site).

It is worth noting that in all but the latter scenario, if an attacker is able to use the cookie before the legitimate client gets a chance to, then the impersonation attack will always succeed.

7.2.2. Cookie Deletion

A direct and important consequence of the missing owner role in SCS is that a client could intentionally delete its cookie and return nothing.

The application protocol has to be designed so there is no incentive to do so, for instance:

- it is safe for the cookie to represent some kind of positive capability -- the possession of which increases the client's powers;

- it is not safe to use the cookie to represent negative capabilities -- where possession reduces the client's powers -- or for revocation.

Note that this behavior is not equivalent to cookie removal in the "server-side" cookie model, because in case of missing cookie backup by other parties (e.g., the application using SCS), the client could simply make it disappear once and for all.

7.2.3. Cookie Sharing or Theft

Just like with plain cookies, SCS doesn't prevent sharing (both voluntary and illegitimate) of cookies between multiple clients.

In the context of voluntary cookie sharing, using HTTPS only as a separate secure transport provider is useless: in fact, client certificates are just as shareable as cookies. Instead, using some form of secure channel binding (as illustrated in Section 7.2.1) may cancel this risk.

The risk of theft could be mitigated by securing the wire (e.g., via HTTPS, IPsec, VPN, etc.), thus reducing the opportunity of cookie stealing to a successful attack on the protocol endpoints.

In order to reduce the attack window on stolen cookies, an application may choose to generate cookies whose lifetime is upper bounded by the browsing session lifetime (i.e., by not attaching an Expires attribute to them.)

7.2.4. Session Fixation

Session fixation vulnerabilities [Kolsec] are not addressed by SCS.

A more sophisticated protocol involving active participation of the UA in the SCS cookie manipulation process would be needed: e.g., some form of challenge/response exchange initiated by the server in the HTTP response and replied to by the UA in the next chained HTTP request.

Unfortunately, the present specification, which is based on [RFC6265], sees the UA as a completely passive actor whose role is to blindly paste the cookie value set by the server.

Nevertheless, the SCS cookies wrapping mechanism may be used in the future as a building block for a more robust HTTP state management protocol.

7.3. Advantages of SCS over Server-Side Sessions

Note that all the above-mentioned vulnerabilities also apply to plain cookies, making SCS at least as secure, but there are a few good reasons to consider its security level enhanced.

First of all, the confidentiality and authentication features provided by SCS protect the cookie value, which is normally plaintext and tamperable.

Furthermore, neither of the common vulnerabilities of server-side sessions (session identifier (SID) prediction and SID brute-forcing) can be exploited when using SCS, unless the attacker possesses encryption and HMAC keys (both current ones and those relating to the previous set of credentials).

More in general, no slicing nor altering operations can be done over an SCS PDU without controlling the cryptographic key-set.

8. References

8.1. Normative References

[NIST-AES] National Institute of Standards and Technology, "Advanced Encryption Standard (AES)", FIPS PUB 197, November 2001, <http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf>.

[RFC1951] Deutsch, P., "DEFLATE Compressed Data Format Specification version 1.3", RFC 1951, May 1996.

[RFC2104] Krawczyk, H., Bellare, M., and R. Canetti, "HMAC: Keyed-Hashing for Message Authentication", RFC 2104, February 1997.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

[RFC2616] Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., and T. Berners-Lee, "Hypertext Transfer Protocol -- HTTP/1.1", RFC 2616, June 1999.

[RFC4086] Eastlake, D., Schiller, J., and S. Crocker, "Randomness Requirements for Security", BCP 106, RFC 4086, June 2005.

[RFC4648] Josefsson, S., "The Base16, Base32, and Base64 Data Encodings", RFC 4648, October 2006.

[RFC5652] Housley, R., "Cryptographic Message Syntax (CMS)", STD 70, RFC 5652, September 2009.

[RFC6194] Polk, T., Chen, L., Turner, S., and P. Hoffman, "Security Considerations for the SHA-0 and SHA-1 Message-Digest Algorithms", RFC 6194, March 2011.

[RFC6265] Barth, A., "HTTP State Management Mechanism", RFC 6265, April 2011.

8.2. Informative References

[Bellare] Bellare, M., "New Proofs for NMAC and HMAC: Security Without Collision-Resistance", 2006.

[CLIQUES] Steiner, M., Tsudik, G., and M. Waidner, "Cliques: A New Approach to Group Key Agreement", 1996.

[Kohno] Kohno, T., Palacio, A., and J. Black, "Building Secure Cryptographic Transforms, or How to Encrypt and MAC", 2003.

[Kolsec] Kolsec, M., "Session Fixation Vulnerability in Web-based Applications", 2002.

[RFC3740] Hardjono, T. and B. Weis, "The Multicast Group Security Architecture", RFC 3740, March 2004.

[RFC4301] Kent, S. and K. Seo, "Security Architecture for the Internet Protocol", RFC 4301, December 2005.

[RFC5056] Williams, N., "On the Use of Channel Bindings to Secure Channels", RFC 5056, November 2007.

[RFC5246] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246, August 2008.

[Steiner] Steiner, M., Tsudik, G., and M. Waidner, "Diffie-Hellman Key Distribution Extended to Group Communication", 1996.

Appendix A. Examples

The examples in this section have been created using the 'scs' test tool bundled with LibSCS, a free and opensource reference implementation of the SCS protocol that can be found at (http://github.com/koanlogic/libscs).

A.1. No Compression

The following parameters:

- Plaintext cookie: "a state string"
- AES-CBC-128 key: "123456789abcdef"
- HMAC-SHA1 key: "12345678901234567890"
- TID: "tid"
- ATIME: 1347265955
- IV: \xb4\xbd\xe5\x24\xf7\xf6\x9d\x44\x85\x30\xde\x9d\xb5\x55\xc9\x4f

produce the following tokens:

- DATA: DqfW4SFqcjBXqSTvF2qnRA
- ATIME: MTM0NzI2NTk1NQ
- TID: OHU7M1cqdDQt
- IV: tL3lJPf2nUSFMN6dtVXJTw
- AUTHTAG: AznYHKga9mLL8ioi3If_1iy2KSA

A.2. Use Compression

The same parameters as above, except ATIME and IV:

- Plaintext cookie: "a state string"
- AES-CBC-128 key: "123456789abcdef"
- HMAC-SHA1 key: "12345678901234567890"
- TID: "tid"

- ATIME: 1347281709

- IV:
  \x1d\xa7\x6f\xa0\xff\x11\xd7\x95\xe3\x4b\xfb\xa9\xff\x65\xf9\xc7

produce the following tokens:

- DATA: PbE-ypmQ43M8LzKZ6fMwFg-COrLP2l-Bvgs

- ATIME: MTM0NzI4MTcwOQ

- TID: akxIKmhbMTE8

- IV: HadvoP8R15XjS_up_2X5xw

- AUTHTAG: A6qevPr-ugHQChlr_EiKYWPvpB0

In both cases, the resulting SCS cookie is obtained via ordered concatenation of the produced tokens, as described in Section 3.1.

Authors' Addresses

Stefano Barbato
KoanLogic
Via Marmolada, 4
Vitorchiano (VT), 01030
Italy

EMail: tat@koanlogic.com

Steven Dorigotti
KoanLogic
Via Maso della Pieve 25/C
Bolzano, 39100
Italy

EMail: stewy@koanlogic.com

Thomas Fossati (editor)
KoanLogic
Via di Sabbiuno 11/5
Bologna, 40136
Italy

EMail: tho@koanlogic.com