

Network Working Group  
Internet-Draft  
Intended status: Standards Track  
Expires: September 2, 2007

E. Rescorla  
Network Resonance  
March 01, 2007

How to Implement Secure (Mostly) Stateless Tokens  
draft-rescorla-stateless-tokens-01.txt

Status of this Memo

By submitting this Internet-Draft, each author represents that any applicable patent or other IPR claims of which he or she is aware have been or will be disclosed, and any of which he or she becomes aware will be disclosed, in accordance with [Section 6 of BCP 79](#).

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF), its areas, and its working groups. Note that other groups may also distribute working documents as Internet-Drafts.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

The list of current Internet-Drafts can be accessed at <http://www.ietf.org/ietf/lid-abstracts.txt>.

The list of Internet-Draft Shadow Directories can be accessed at <http://www.ietf.org/shadow.html>.

This Internet-Draft will expire on September 2, 2007.

Copyright Notice

Copyright (C) The IETF Trust (2007).

Abstract

A common protocol problem is to want to arrange to maintain client state with little or no local storage. The usual design pattern here is to provide the client with a token which is returned with subsequent interactions. In order to prevent tampering, forgery, and privacy issues, such tokens should be cryptographically protected. This draft describes one workable mechanism for constructing such tokens.

Internet-Draft

Stateless Tokens

March 2007

## Table of Contents

<a href="#">1.</a>	Introduction . . . . .	<a href="#">3</a>
<a href="#">2.</a>	Conventions Used In This Document . . . . .	<a href="#">3</a>
<a href="#">3.</a>	General Principles . . . . .	<a href="#">3</a>
<a href="#">3.1.</a>	Reference Architecture . . . . .	<a href="#">3</a>
<a href="#">3.2.</a>	Token Construction and Processing . . . . .	<a href="#">4</a>
<a href="#">3.3.</a>	Invalidation and Supersession . . . . .	<a href="#">5</a>
<a href="#">4.</a>	Example Token Format . . . . .	<a href="#">5</a>
<a href="#">4.1.</a>	Key Rollover . . . . .	<a href="#">8</a>
<a href="#">5.</a>	ITLs Data Structures . . . . .	<a href="#">9</a>
<a href="#">5.1.</a>	Simple Lists . . . . .	<a href="#">9</a>
<a href="#">5.2.</a>	Bit Fields . . . . .	<a href="#">9</a>
<a href="#">5.3.</a>	Bloom Filters . . . . .	<a href="#">10</a>
<a href="#">6.</a>	Security Considerations . . . . .	<a href="#">10</a>
<a href="#">6.1.</a>	Failure to Authenticate . . . . .	<a href="#">10</a>
<a href="#">6.2.</a>	Failure to Encrypt . . . . .	<a href="#">11</a>
<a href="#">6.3.</a>	Distinguishable Encryption . . . . .	<a href="#">11</a>
<a href="#">6.4.</a>	Replay Attacks . . . . .	<a href="#">11</a>
<a href="#">6.5.</a>	Token Binding . . . . .	<a href="#">12</a>
<a href="#">7.</a>	Acknowledgments . . . . .	<a href="#">12</a>
<a href="#">8.</a>	References . . . . .	<a href="#">12</a>
<a href="#">8.1.</a>	Normative References . . . . .	<a href="#">12</a>
<a href="#">8.2.</a>	Informational References . . . . .	<a href="#">12</a>
	Author's Address . . . . .	<a href="#">13</a>
	Intellectual Property and Copyright Statements . . . . .	<a href="#">14</a>

## [1.](#) Introduction

A common protocol problem is to want to arrange to maintain client state with little or no local storage. The usual design pattern here is to provide the client with a token which is returned with subsequent interactions. One such application is TLS tickets [\[4\]](#), which allow the server to offload the TLS session cache onto the client. Another application is globally routable unique identifiers [\[5\]](#) (GRUUs) which bind a second URI to a SIP AOR. GRUUs can be defined in a stateless mode, which requires no storage on the SIP registrar. Another application for this kind of technique is to build Web "shopping carts".

Because the state token is stored on a remote node it is susceptible to inspection and /or tampering by untrusted third parties. Therefore, it becomes important to cryptographically secure tokens, typically by encrypting them to provide confidentiality and adding a message integrity check (MIC) to provide integrity for the token data and assurance that it was generated by the consuming node. Note that the remote node doesn't need to do anything with the token other than echo it back, therefore there is no need for it to be able to access the internals of the token.

Although the general techniques for constructing tokens of this type are well understood, there are some subtle issues involved and there is no single reference that describes acceptable constructions. Accordingly, each token-using application has had to design its own construction. The purpose of this document is to provide such a reference.

## [2.](#) Conventions Used In This Document

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 [\[1\]](#).

### [3.](#) General Principles

This section provides a somewhat abstract overview of the techniques for constructing stateless tokens. In the next section we will describe precise example formats that implement these principles.

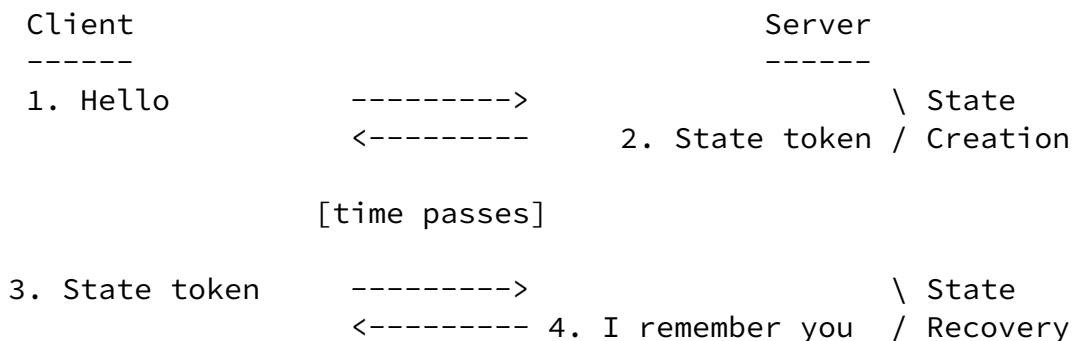
#### [3.1.](#) Reference Architecture

The reference architecture for this kind of system is shown in the figure below:

Rescorla Expires September 2, 2007 [Page 3]

---

Internet-Draft Stateless Tokens March 2007



In message 1, the client contacts the server. The server creates some state (e.g., creates a shopping cart structure), serializes that into a state token, and sends it to the client in message 2. From the client's perspective, this token is opaque, though it of course has some internal structure that is parseable by the server.

Some time later, the client wants to talk to the server again. When it reconnects, it sends the state token back in message 3. The server decodes the state token and recovers the previously created state and can process whatever transaction the client is trying to perform.

#### [3.2.](#) Token Construction and Processing

We assume that the server has some static cryptographic keying material, consisting of:

- K<sub>e</sub> -- an encryption key
- K<sub>m</sub> -- a message integrity (MAC) key

These keys SHOULD be randomly generated and of sufficient length to match whatever cryptographic algorithms are in use. They are stored in semi-permanent storage and used for protecting all tokens.

In order to construct a token, the server takes its state and packs it into a single string  $S$ .  $S$  should be constructed so that it can be unambiguously parsed by the server. The server then encrypts the token and applies a MAC. For instance:

$$\begin{aligned} EA &= \text{Encrypt}(K_e, S) \\ \text{Token} &= EA \parallel \text{MAC}(K_m, EA) \end{aligned}$$

The encryption algorithm SHOULD to be chosen to have the following properties:

1. Given any  $(EA, S)$  pair, it is computationally infeasible to determine whether  $EA$  was derived from  $S$ . (Note that you may obtain some information from length).

2. Multiple encryptions of the same value  $S$  produce different tokens
3. It is computationally infeasible to determine any information about the relationship between the  $S$  values embedded in two separate tokens.

AES [6] in either CBC or counter (CTR) with randomly chosen IVs both meet these requirements.

The MAC algorithm is simply a standard MAC, such as HMAC [2].

The design described above works in a variety of situations but has the significant drawback that it does not permit the server to invalidate or supersede states. Consider the case where the token is being used to contain login state. One function such interfaces typically offer is a "logout" button. However, if the server is stateless, then there is no way to detect that a token which is presented corresponds to a session which has been invalidated. A related issue is "state rollback". If a server gives a client a new state token (e.g., decrementing an account balance) the client can send an older token instead; this is effectively a replay attack.

### [3.3](#). Invalidation and Supersession

Invalidating or superceding tokens requires the server to be able to store some state. The usual procedure is for the server to retain a list of invalid tokens. In the most primitive implementation, this requires the server to keep all invalid tokens since the beginning of time. Obviously this is inconvenient, so tokens are created with an expiration time, thus limiting the invalid list to unexpired tokens which are invalid (there is a parallel here to certificate revocation lists [3]).

Even with expiry, high invalidation rates can lead to large Invalid Token Lists (ITLs). This is especially true if you are having state  $n+1$  supercede state  $n$  (this is done by invalidating state  $n$  and then issuing state  $n+1$ ). A number of techniques can be used to minimize the size of the ITL. For instance, it may be stored in a Bloom filter or each token may be assigned an index with the ITL stored as a bit field. These options are examined in more detail in [Section 5](#).

#### 4. Example Token Format

In this section we describe one acceptable token format that complies with the guidelines in the previous section. For reference, consider a state value as shown in Figure 1:

Rescorla Expires September 2, 2007 [Page 5]

---

Internet-Draft Stateless Tokens March 2007

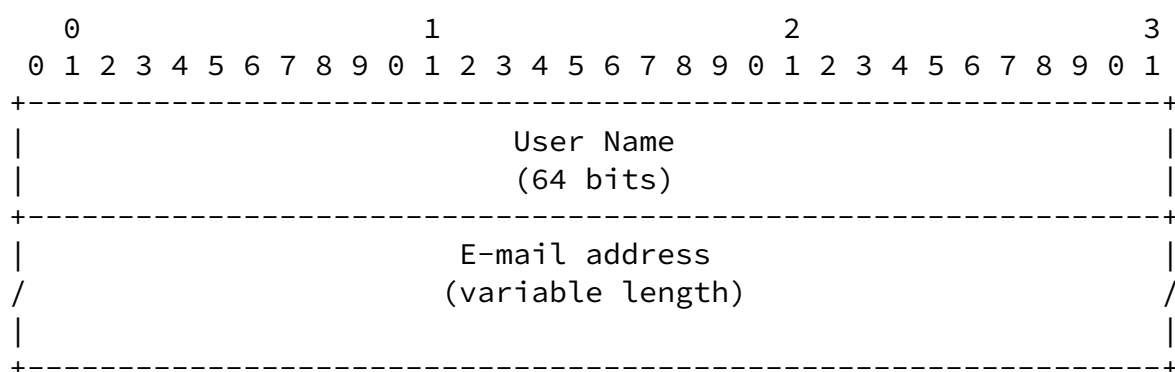
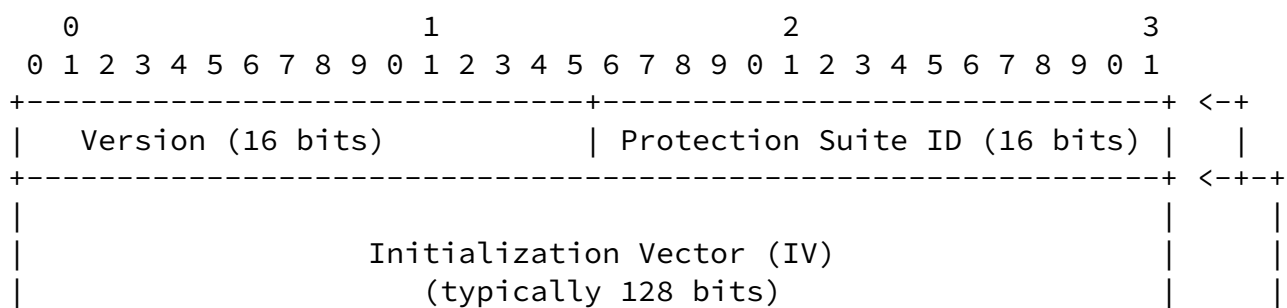


Figure 1: Example state value

This state value would allow the server to "remember" some registration information, e.g., for a web bulletin board. Note that

the token construction is agnostic about the state value as long as it can be represented as a byte string.

The token format is shown in Figure 1.



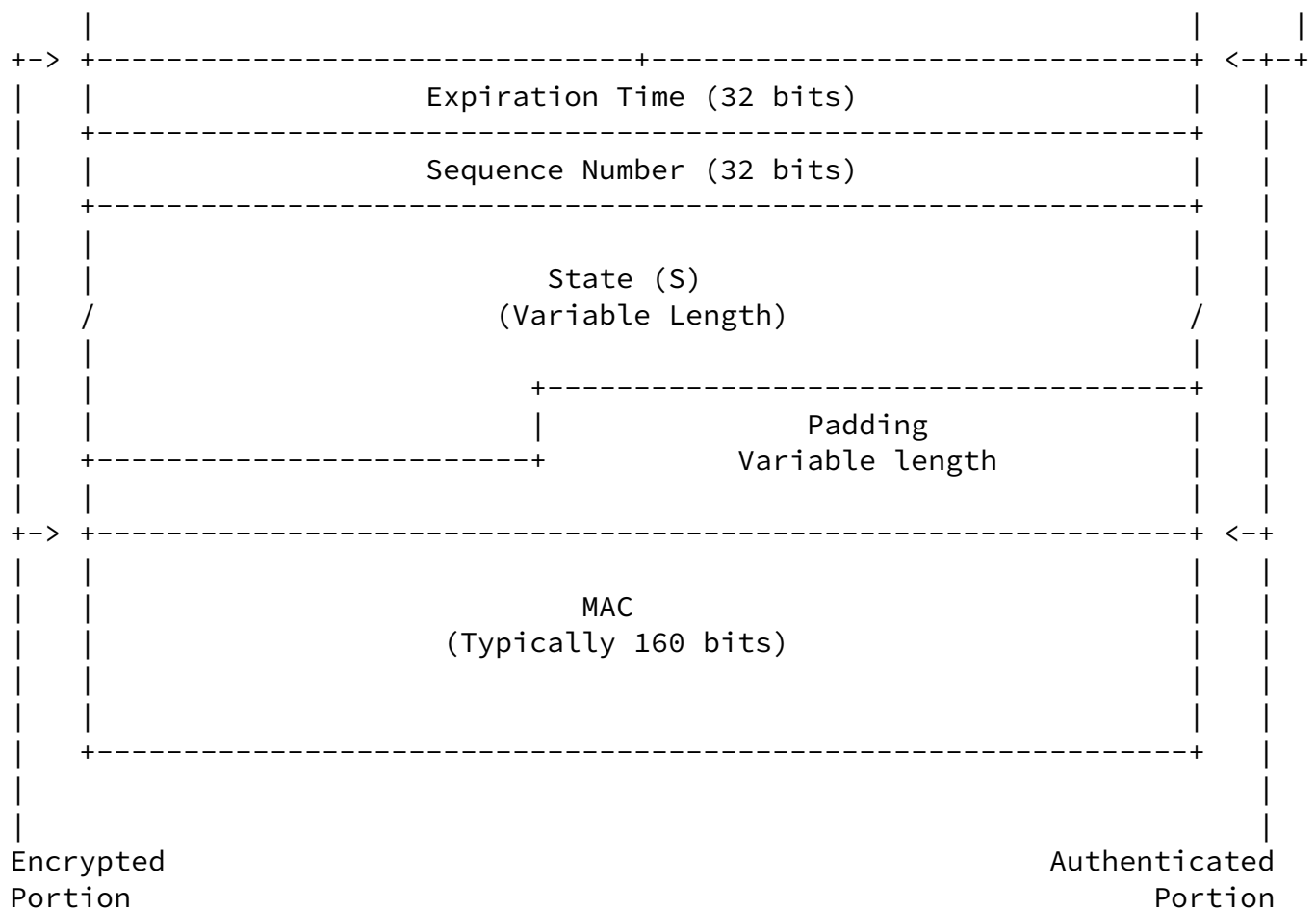


Figure 2: Example stateless token format

We assume that the state value that the server wishes to deliver is a byte string  $S$ .

To construct a token, follow the following steps.

1. Construct  $TI = ET || SEQ || S$  where  $ET$  is the expiry time in seconds since the UNIX epoch and  $SEQ$  is the token sequence number. Both are 32-bit integers so  $TI$  can be unambiguously parsed.

2. Generate a random 128-bit initialization vector.

3. Set EA equal to the encryption of TI with key  $K_e$  and initialization vector IV:  $EA = \text{Encrypt}(K_e, IV, TI)$
4. Compute the MAC over the Version, Protection Suite ID (PSID), and EA:  $H = \text{MAC}(K_m, \text{Version} || \text{PSID} || EA)$
5. Pack the values into token T.

Because the producer and consumer of the token are the same machine, the Version and Protection Suite IDs are strictly unnecessary because the consumer knows what kind of tokens it produces. However, in the interest of tokens being self-describing and ease of version/algorithm transition, they are included in the token. Similarly, there is no need to have standardized Version and Protection Suite ID (PSID) values. However, we recommend that Version be the integer 1 and the following Prot\_Id values:

Protection Suite	PSID	Encryption	MAC
AES_128_CBC_WITH_SHA1	1	AES-128-CBC	HMAC-SHA1
AES_256_CBC_WITH_SHA_256	2	AES-256-CBC	HMAC-SHA-256

Table 1

Note that this follows the "Encrypt-then-Authenticate (EtA) paradigm recommended by Krawczyk [7].

To verify token T, perform the following steps:

1. Compare  $\text{MAC}(K_m, \text{Version} || \text{Prot\_ID} || EA)$  to the MAC in the token. If they don't match, reject the token.
2. Decrypt EA using  $K_e$  and IV to recover TI:  $TI = \text{Decrypt}(K_e, IV, EA)$
3. Break up TI into ET, SEQ, and S.
4. If the current time is after ET, the token is expired and should be rejected.
5. If applicable, verify that the token is not on the ITL.
6. Deliver the token to the application

#### [4.1](#). Key Rollover

One concern that people sometimes have is that you may wish to periodically roll over keys. In general, this is not necessary since modern cryptosystems do not require rekeying with the traffic volumes relevant here. If this is a concern, then there are several easy options, including adding a key ID, placing the sequence number in the clear, or overloading the IV or protection suite ID. If only a

few keys are used, trial verification can be used to determine which one is active.

## 5. ITLs Data Structures

In this section, we discuss three data structures for maintaining the ITL.

### 5.1. Simple Lists

The most natural implementation is to simply store a list of all the invalid but unexpired tokens. This requires  $I \times \text{size}(\text{Token})$  bytes of storage where  $I$  is the number of tokens. If you instead store a list of sequence numbers, then the required storage becomes  $4I$ . It's probably a good idea to keep this list sorted so that binary search can be used for looking up potential tokens. Note that if you have a very high invalidity rate it is more efficient to maintain a valid token list.

### 5.2. Bit Fields

In environments where a large fraction ( $> 1/32$ ) of tokens will eventually be invalidated, a superior data structure is a bitmask vector. What needs to be recorded here is:

- o The sequence number of the earliest valid token (the low-water mark)
- o A bit vector with one bit for every token from the low water mark to the latest unexpired invalid token (the high-water mark).

For instance, if you have issued tokens 0-18 and tokens 1, 3, 5, 7, 11 12, and 15 have been invalidated, but tokens 0-7 have also expired, you would need to store a low-water of 8 (the earliest valid token), plus to store an 8-bit bitmask vector as shown in Figure 3. Note that because the high-water mark is 15, there is no need to store bits corresponding to sequence numbers 16-18.

```
      1 1 1 1 1 1
    8 9 0 1 2 3 4 5
  +--+--+--+--+--+
  |0 0 0 1 1 0 0 1|
  +--+--+--+--+--+
```

Figure 3: Bitmask vector invalidation list

The bitmask may also be stored compressed (e.g., run length encoded), though of course this provides slower access.

### [5.3.](#) Bloom Filters

With lower invalidation rates, Bloom filters [\[9\]](#) can be used to store the ITL. Bloom filters have the significant advantage that they are much smaller (about 10x smaller for bitmasks for 1% revocation rates). They have the drawback that they have false positives (tokens which appear to be invalidated but are not in fact invalid). In settings where the state being stored is soft (e.g., [\[4\]](#)), this isn't a problem but when it is hard state, then it can be. The Bloom filter can be tuned for arbitrary false positive rates, but improved specificity requires larger Bloom filter sizes.

Another issue with simple Bloom filters is that they do not allow you to delete entries from the ITL when the token expires. The result is that the filter fills up with expired tokens and produces a monotonically increasing false positive rate. One approach here is to use counting Bloom filters [\[10\]](#). However, these can still overflow and produce false positives. A superior solution is simply to use multiple Bloom filters corresponding to different expiry periods and then delete a Bloom filter once the current time sweeps past the expiry period represented by the filter.

## [6.](#) Security Considerations

In this section, we address a number of easy mistakes to make in designing mechanisms of this type.

### [6.1.](#) Failure to Authenticate

One common error is to simply encrypt the token without using any authentication or message integrity. The result is that tokens are susceptible to a variety of forgery attacks. This is a particular problem if a stream cipher such as counter mode is used, because an attacker can make targeted changes to any bit in the token, but attacks are possible with CBC mode as well.

Consider a token format like the one presented in Figure 1 but without an integrity check. The attacker contacts the server and

gets a state token T, containing his identity. The first block (128 bits) of ciphertext contain ET, SEQ, and the first 64 bits of S, which contain the username, which we call I. The attacker wants to pose as username I'. The attacker then generates a new IV' with the low order 64-bits set to IV XOR I XOR I' and builds a new token T' = IV' || EA. Because of the properties of CBC, when T' is decrypted its first block will decrypt to EQ || SEQ || I', thus allowing a user to pose as any other user. Such attacks are much more serious with CTR mode, where the whole plaintext may be tampered with. This is

simply a special case of the general cryptographic rule that encryption cannot be counted on to provide integrity.

## [6.2.](#) Failure to Encrypt

Encryption of tokens is not strictly necessary in order to provide integrity for the tokens. However, in most settings it is desirable to provide confidentiality. In the case of TLS tickets, that is because secret keying material is carried in the token. In the case of GRUUs one of the purposes of the construction is to provide privacy. Therefore, despite the possibility that confidentiality is not required, in general we recommend encrypting the data unless there is a clear requirement for it not to be.

## [6.3.](#) Distinguishable Encryption

In cases such as GRUU where privacy is a requirement, then it is important for tokens to be unlinkable; at minimum, it must be infeasible for an attacker to determine whether two tokens issued by the same server correspond to the same or related underlying state information. Optimally, it should be infeasible for an attacker to determine whether two tokens were generated by the same server or different servers (obviously, this depends on them using the same token format and similar state value lengths.)

This places a requirement on the algorithms used to encrypt the token; repeated encryptions of the same (or related) plaintexts must produce ciphertexts that cannot be distinguished from encryptions of different plaintexts. Specifically, block ciphers in ECB mode are not suitable here and block ciphers in CBC or CTR mode require unique initialization vectors. In order to avoid attackers determining the temporal relationship between two tokens, the IV should be generated

with a cryptographically secure random or pseudorandom number generator. This is also the rationale for encrypting the expiry time and sequence number. Note, however, that this is a tradeoff; having these values in the clear would allow immediate rejection of invalidated or expired tokens. Instead, the server has to decrypt the token in order to check its status.

#### [6.4.](#) Replay Attacks

As described in [Section 5](#), if one wishes to be able to invalidate tokens, one must keep state on the server. This is obvious in contexts where you wish to invalidate tokens directly (such as logout) but in cases where you simply wish one state to supersede another, the necessity for invalidating the old state is less obvious. However, the failure to do so subjects you to replay attacks. Implementations which replace state tokens therefore need

Rescorla

Expires September 2, 2007

[Page 11]

---

Internet-Draft

Stateless Tokens

March 2007

to maintain invalid token lists.

#### [6.5.](#) Token Binding

In order to prevent a token from being used outside its intended context it may be necessary to bind the token to a set of verifiable information associated with the intended context. For example, if a token belongs to a particular user then including the username in the token allows the server to verify that the user authenticated during the current session is indeed the user that was issued the token. Another possibility is to bind the token to a particular key that the client possess and is included in the token (note that in this case the token must be encrypted). Other information may be bound within the token, which may further limit its use in other contexts. In some cases privacy or other considerations may prevent inserting this information into the token. In these cases the system should attempt prevent the disclosure of the token to third parties through the use of encryption and other access control mechanisms.

### [7.](#) Acknowledgments

The material in [Section 5](#) is based on the analysis in [\[8\]](#). The material in [Section 6.5](#) was contributed by Joe Salowey. Thanks to Joe Salowey for review comments.

## [8.](#) References

### [8.1.](#) Normative References

- [1] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", [BCP 14](#), [RFC 2119](#), March 1997.

### [8.2.](#) Informational References

- [2] Krawczyk, H., Bellare, M., and R. Canetti, "HMAC: Keyed-Hashing for Message Authentication", [RFC 2104](#), February 1997.
- [3] Housley, R., Polk, W., Ford, W., and D. Solo, "Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile", [RFC 3280](#), April 2002.
- [4] Salowey, J., Zhou, H., Eronen, P., and H. Tschofenig, "Transport Layer Security (TLS) Session Resumption without Server-Side State", [RFC 4507](#), May 2006.
- [5] Rosenberg, J., "Obtaining and Using Globally Routable User

Agent (UA) URIs (GRUU) in the Session Initiation Protocol (SIP)", [draft-ietf-sip-gruu-11](#) (work in progress), October 2006.

- [6] National Institute of Standards and Technology, "Specification for the Advanced Encryption Standard (AES)", FIPS 197, November 2001.
- [7] Krawczyk, H., "The Order of Encryption and Authentication for Protecting Communications (or: How Secure Is SSL?)", CRYPTO 2001.
- [8] Schacham, H., Boneh, D., and E. Rescorla, "Client-Side Caching for TLS", ACM Trans. Info. & Sys. Security 7(4):553-75.
- [9] Bloom, B., "Space/time trade-offs in hash coding with allowable errors", Comm. ACM 13(7):422-6.

- [10] Fan, L., Cao, P., Almeida, J., and A. Broder, "Summary Cache: A Scalable Wide-Area Web Cache Sharing Protocol", SIGCOMM .

#### Author's Address

Eric Rescorla  
Network Resonance  
2483 E. Bayshore #212  
Palo Alto, CA 94303  
USA

Email: [ekr@networkresonance.com](mailto:ekr@networkresonance.com)

Rescorla	Expires September 2, 2007	[Page 13]
----------	---------------------------	-----------

---

Internet-Draft	Stateless Tokens	March 2007
----------------	------------------	------------

#### Full Copyright Statement

Copyright (C) The IETF Trust (2007).

This document is subject to the rights, licenses and restrictions contained in [BCP 78](#), and except as set forth therein, the authors retain all their rights.

This document and the information contained herein are provided on an "AS IS" basis and THE CONTRIBUTOR, THE ORGANIZATION HE/SHE REPRESENTS OR IS SPONSORED BY (IF ANY), THE INTERNET SOCIETY, THE IETF TRUST AND

THE INTERNET ENGINEERING TASK FORCE DISCLAIMS ALL WARRANTIES, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY WARRANTY THAT THE USE OF THE INFORMATION HEREIN WILL NOT INFRINGE ANY RIGHTS OR ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

## Intellectual Property

The IETF takes no position regarding the validity or scope of any Intellectual Property Rights or other rights that might be claimed to pertain to the implementation or use of the technology described in this document or the extent to which any license under such rights might or might not be available; nor does it represent that it has made any independent effort to identify any such rights. Information on the procedures with respect to rights in RFC documents can be found in [BCP 78](#) and [BCP 79](#).

Copies of IPR disclosures made to the IETF Secretariat and any assurances of licenses to be made available, or the result of an attempt made to obtain a general license or permission for the use of such proprietary rights by implementers or users of this specification can be obtained from the IETF on-line IPR repository at <http://www.ietf.org/ipr>.

The IETF invites any interested party to bring to its attention any copyrights, patents or patent applications, or other proprietary rights that may cover technology that may be required to implement this standard. Please address the information to the IETF at [ietf-ipr@ietf.org](mailto:ietf-ipr@ietf.org).

## Acknowledgment

Funding for the RFC Editor function is provided by the IETF Administrative Support Activity (IASA).