Network Working Group
Internet-Draft
Intended status: BCP
Expires: April 2, 2009

P. Gutmann
University of Auckland
September 29, 2008

Key Management through Key Continuity (KCM)
draft-gutmann-keycont-01.txt

Status of this Memo

Abstract

This memo provides advice and Best Current Practice for implementors and deployers of security applications that wish to use the key continuity method of key management.

Table of Contents

1. Introduction

There are many ways of managing the identification of remote entities. One simple but also highly effective method is the use of key continuity, a means of ensuring that that the entity a user is dealing with today is the same as the one they were dealing with last week (this principle is sometimes referred to as continuity of identity). In the case of cryptographic protocols the problem becomes one of determining whether a file server, mail server, online store, or bank that a user dealt with last week is still the same one this week. Using key continuity to verify this means that if the remote entity used a given key to communicate/authenticate itself last week then the use of the same key this week indicates that it's the same entity. This doesn't require any third-party attestation because it can be done directly by comparing last week's key to this week's one. This is the basis for key management through key continuity: Once you've got a known-good key, you can verify a remote entity's identity by verifying that they're still using the same key. This document describes the principles that underly key management through key continuity, and provides guidelines for its use in practice.

1.1. 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 [RFC2119].

2. Key Management through Key Continuity

In its most basic form, key management through key continuity consists of two steps:

Step 1 On the first connection exchange key(s), possibly with additional out-of-band authentication.
Step 2 On subsequent connections ensure that the key being used matches the one exchanged initially.

In more formal terms, the key continuity method of key management may be regarded as a variation of the baby-duck security model [Duckling1] [Duckling2] in which a newly-initialised device (either one fresh out of the box or one reset to its ground state) imprints upon the first device that it sees in the same way that a newly hatched duckling imprints on the first moving object that it sees as its mother.

2.1. Key Continuity in SSH

SSH [SSH] was the first widely-used security application that used key continuity as its primary form of key managment. The first time that a user connects to an SSH server, the client application displays a warning that it's been given a new public key that it's never encountered before, and asks the user whether they want to continue. When the user clicks "Yeah, sure, whatever" (although the button is more frequently labelled "OK"), the client application remembers the key that was used, and compares it to future keys used by the server. If the key is the same each time, there's a good

chance that it's the same server (SSH terminology refers to this as the known-hosts mechanism). In addition to this SSH allows a user to verify the key via its fingerprint, which can be conveniently exchanged via out-of-band means. The fingerprint is a universal key identifier consisting of the hash of the key components or the hash of the certificate if the key is in the form of a certificate.

SSH is the original key-continuity solution, but unfortunately it doesn't provide complete continuity. When the server is rebuilt, the connection to the previous key is lost unless the sysadmin has remembered to archive the configuration and keying information after they set up the server (some OS distributions can migrate keys over during an OS upgrade, so this can vary somewhat depending on the OS and how total the replacement of system components is). Since SSH is often used to secure access to kernel-of-the-week open-source Unix systems the breaking of the chain of continuity can happen more frequently than would first appear.

Some of the problem is due to the ease with which an SSH key

changeover can occur. In the PKI world this process is so painful that the same key is typically reused and recycled in perpetuity, ensuring key continuity at some cost in security since a compromise of a key recycled over a period of several years compromises all data that the key protected in that time unless a mechanism that provides perfect forward secrecy is used (it rarely is). In contrast an SSH key can be replaced quickly and easily, limiting its exposure to attack but breaking the chain of continuity. A solution to this problem would be to have the server automatically generate and certify key n+1 when key n is taken into use, with key n+1 saved to offline media such as a floppy disk or USB memory token for future use when the system or SSH server is reinstalled/replaced. In this way continuity to the previous, known server key is maintained. Periodically rolling over the key (even without it being motivated by the existing system/server being replaced) is good practice since it limits the exposure of any one key. This would require a small change to the SSH protocol to allow an old-with-new key exchange message to be set after the changeover has occurred.

2.2. Key Continuity in SSL/TLS and IPsec

Unlike SSH, SSL/TLS [TLS] and IPsec [IPsec] were designed to rely on

an external key management infrastructure, although at a pinch both can function without it by using shared keys, typically passwords. The lack of such an infrastructure has been addressed in two ways. In SSL the client (particularly in its most widespread form, the web browser) contains a large collection of hardcoded CA certificates (over a hundred) that are trusted to issue SSL server certificates. Many of these hardcoded CAs are completely unknown, follow dubious practices such as using weak 512-bit keys or keys with 40-year lifetimes, appear moribund, or have had their CA keys on-sold to various third parties when the original owners went out of business [NotDead]. All of these CAs are assigned the same level of trust, which means that the whole system is only as secure as the least secure CA, since compromising or subverting any one of the CAs compromises the entire collection (in PKI terminology what's being implemented is unbounded universal cross-certification among all of the CAs).

The second solution, used by both SSL/TLS and IPsec, is to use self-issued certificates where the user acts as their own CA and issues themselves certificates that then have to be installed on client/peer machines. In both cases the security provided is little better than for SSH keys unless the client is careful to disable all CA certificates except for the one or two that they trust, a process that requires around 700 mouse clicks for Internet Explorer 6. A further downside of this is that the client software will now throw up warning dialogs prophesying all manner of doom and destruction

when an attempt is made to connect to a server with a certificate from a now-untrusted CA, although research by security usability researchers indicates that invalid or untrusted certificates have little to no effect on users anyway so this is probably not a big deal [Hardening].

The same key-continuity solution used in SSH can be used here, and is already employed by some SSL clients such as MTAs which have to deal with self-issued and similar informal certificates more frequently than other applications such as web servers. This is because of their use in STARTTLS, an extension to SMTP that provides opportunistic TLS-based encryption for mail transfers. Similar facilities exist for other mail protocols such as POP and IMAP, with the mechanism being particularly popular with SMTP server administrators because it provides a means of authenticating

legitimate users to prevent misuse by spammers. Since the mail servers are set up and configured by sysadmins rather than commercial organisations worried about adverse user reactions to browser warning dialogs they typically make use of self-issued certificates since there's no point in paying a CA for the same thing.

Key continuity management among STARTTLS implementations is still somewhat haphazard. Since STARTTLS is intended to be a completely transparent, fire-and-forget solution, the ideal setup would automatically generate a certificate on the server side when the software is installed and use standard SSH-style key continuity management on the client, with optional out-of-band verification via the key/certificate fingerprint. Some implementations (typically open-source ones) support this fully, some support various aspects of it (for example requiring tedious manual operations for certificate generation or key/certificate verification), and some (typically commercial ones) require the use of certificates from commercial CAs, an even more tedious (and expensive) manual operation.

2.3. Key Continuity in Other Security Protocols

A similar model is used in SIP, in which the first connection exchanges a (typically) self-signed certificate which is then checked for continuity on subsequent connects. Further measures such as the use of speaker voice recognition can be used to provide additional authentication for the SIP exchange. A similar principle has been used in several secure IP-phone protocols, which (for example) have one side read out a hash of the key over the secure link, relying for its security on the fact that real-time voice spoofing is relatively difficult to perform.

3. Using Key Continuity for Key Management

Section 2 outlined a number of considerations that need to be taken into account when using key continuity as a form of key management. These are covered in the following subsections.

3.1. Key Generation

The simplest key-continuity approach automatically (and transparently) generates the key when the application that uses it is installed or configured for the first time. If the underlying protocol uses certificates then the application would generate a standard self-signed certificate at this point, otherwise it can use whatever key format the underlying protocol uses, typically raw public key components encoded in a protocol-specific manner.

3.2. Optional out-of-band Authentication

If possible, the initial exchange should use additional out-of-band authentication to authenticate the key. A standard technique is to generate a hash or fingerprint of the key and verify the hash through out-of-band means. All standard security protocols have a notion of a key hash in some form, whether it be an X.509 certificate fingerprint, a PGP/OpenPGP [OpenPGP] key fingerprint, or an SSH key fingerprint and the use of key authentication channels such as PGP key fingerprints printed on business cards or in email signatures is a long-established practice. Unfortunately research has shown that this form of authentication is (at least in the case of SSH fingerprints) relatively easily defeated [Fingerprints], see Section 3.9 for a more practical solution to this problem

An extended form of this style of authentication has been proposed in the form of cryptoIDs, long-lived fingerprints tied to a (user-controlled) root key that's used to certify sub-keys on demand. The intent of a cryptoID is to associate it with user information such as their name and email address and treat it as a standard part of the user identity data. For example a key change notification would be handled in the same way as an email address change notification, making it part of the standard user information-propagation channels.

The out-of-band verification of the hash/fingerprint is done in a situation-specific manner. For example when the key is used in a VoIP application the communicating parties may read the hash value over the link, relying on speaker voice recognition and the difficulty of performing real-time continous-speech spoofing for security. When the key is used to secure access to a network server the hash may be communicated in person, over the phone, printed on a business card, or published in some other well-known location. When

the key is used to secure access to a bank server the hash may be

communicated using a PIN mailer, or by having the user visit their bank branch. Although it's impossible to enumerate every situation here, applying a modicum of common sense should provide the corerct approach for specific situations.

Other distribution mechanisms are also possible. For example when configuring a machine the administrator can pre-install the key information when the operating system is installed, in the same way that many systems come pre-configured with trusted X.509 certificates.

3.3. Key Rollover

When a key needs to be replaced the new key should ideally be authenticated using forward-chaining authentication from the current key. For example if the key is in the form of an X.509 certificate or PGP key, the current key can sign the new key. If the key consists solely of raw key components exchanged during a protocol handshake, this type of forward-chaining authentication isn't possible without modifying the underlying protocol. Protocol designers may wish to take into account the requirements for forward-chaining authentication when designing new protocols or updating existing ones.

3.4. Key <-> Host/Service Mapping

A key will usually be associated with a service type (for example "SSH" or "TLS"), a host, and a port (typically the port is specified implicitly by the service type, but it may also be specified explicitly if a nonstandard port is used). When storing key continuity data, the service/host/port information should always be stored exactly as seen by the user, without any subsequent expansion, conversion, or other translation. For example if the user knows a host as www.example.com then the key continuity data should be stored under this name, and not under the associated IP address(es). Applying the WYSIWYG principle to the name that the user sees prevents problems with things like virtual hosting (where badguy.example.com has the same IP address as goodguy.example.com), hosts that have been moved to a new IP address, and so on.

3.5. User Interface

Providing a usable user interface for security features has historically proven to be an extremely difficult undertaking [Phishing]. A feature of key continuity management is that in its simplest form it requires very little UI: if the key crosses a certain trust threshold (length of use, number of times it's been

used, rating(s) from an external key continuity authority, and so on) then the connection can be allowed, otherwise it's disallowed. If an external authority is used this becomes particularly simple.

Unfortunately this basic strategy may not be suitable in all cases. In particular if no external authority is available or the key is unknown then requiring that the user wait for a set number of days until it's certain that the key doesn't belong to an ephemeral phishing site will no doubt cause user acceptance problems. The general-case key bootstrap problem appears to be an unsolveable one, althoug a great many hypotheses and workable but situation-specific solutions do exist. If should be remembered though that the intent of key continuity management is to ensure key continuity and not key initial authentication. Designing an appropriate user interface for a key continuity mechanism is (as with all security user interface designs) a challenging problem. [Usability] provides extensive guidance on effectively communicating the issues to users, and potential traps and pitfalls to avoid.

3.6. Trust Lifetimes

Like real-world trust, trust of keys can both increase and decrease over time. In current key trust models, trust is purely boolean, starting out untrusted and then becoming completely trusted for all eternity once the user clicks OK on a dialog requesting that the key be trusted. In the real world trust doesn't work like this, but must be built up over time. Similarly, trust can decay over time.

Rather than making trust a purely boolean value, you can remember how many times a key has been safely used of for how long the key has been around and increase or decrease the trust in it over time. For example a new key should be regarded with suspicion unless verified by out-of-band means, which slowly decreases with repeated use. A key that's been used once should have a much lower trust level than one that's been used a hundred times, something that the current boolean model that's typically used with certificates isn't capable of providing.

3.7. Key Hash/Fingerprint Truncation

The use of the full hash/fingerprint when the authentication process is being performed by humans can be quite cumbersome, requiring the transmission and verification of (for the most common hash algorithm, SHA-1), 40 hex digits. Implementors may consider truncating the hash value to make it easier to work with. For example a 64-bit hash value provides a moderate level of security while still allowing the

value to be printed on media such as business cards and communicated and checked by humans. Although such a short hash value isn't secure

against an intensive brute-force attack, it is sufficient to stop all but the most dedicated attackers, and certainly far more secure than simply accepting the key at face value as is currently widely done. Implementors should consider the relative merits of usability vs. security when deciding to truncate the key hash/fingerprint.

3.8. Key Information Storage

Configuration information of this kind is typically stored using a two-level scheme, systemwide information set up when then operating system is installed or configured and managed by the system administrator and per-user information which is private to each user. For example on Unix systems the systemwide configuration data is traditionally stored in /etc or /var, with per-userconfiguration data stored in the user's home directory. Under Windows the systemwide configuration data is stored under an OS service account and the per-user configuration data is stored in the user's registry branch. The systemwide configuration provides an initial known-good set of key <-> identity mappings, with per-user data providing additional user-specific information that doesn't affect any other users on the system.

Note that the use of plaintext host names may make the information vulnerable to address harvesting by an attacker who has gained read access to the file, since it identifies remote hosts that the user/local host software implicitly trusts. For example a worm might use it to identify remote hosts to attack. In addition a plaintext record of sites that a user has interacted with cryptographically may present privacy concerns. [AddressHarvest] contains more information on this type of attack, and possible countermeasures.

The key continuity data should be protected by at least OS-level security measures if they are available and if it's regarded as particularly sensitive or if it's used on a shared machine like a home PC where everyone shares the same account then it can be given additional protection through encapsulation in PGP or S/MIME security envelopes or through the use of other cryptographic protection mechanisms such as cryptographic checksums or MACs. When encapsulated using PGP or S/MIME the key data is no longer in its

original form and will need to be extracted in order to be used. Alternatively if integrity protection only is considered sufficient then a mechanism like a PGP or S/MIME detached signature can be stored alongside the key data so that the data can be used directly while still allowing it to be verified.

3.9. Key Continuity via External Authorities

Directly managing key continuity on end-user systems may not always be the best approach to take. For one thing the system needs to manage and record a (potentially unbounded) number of locations that the user has visited, which can be problematic if the system has limited storage capacity, although in practice this is likely to be fairly limited since most users interact with only a very small number of sites that use encryption. A larger concern though is the fact that when the user visits a site that they've never been to before they initially have no key continuity information to rely upon, the key/trust bootstrap problem.

The bootstrap problem can be addressed through the use of an external authority to manage the key continuity information. A key continuity external authority records how long a particular key has been in use by a particular cryptographic service and returns this information in response to queries. Note that this differs very significantly from the information that CAs provide. A commercial CA guarantees that money changed hands and that some form of identity checking took place while the only information that a key continuity authority provides is the amount of time that a particular key has been in use by that service. Since the lifetime of a typical phishing site currently runs at around 12 hours (well under the response time of any blacklist or site-rating system), a key continuity authority provides a means of singling out phishing sites and similar short-lived traps for users, making it necessary to operate a phishing site continuously at the same location for a considerable amount of time in order to build up the necessary key continuity history. A key continuity authority therefore implements a form of key whitelist that can be run directly alongside the (relatively ineffective) site blacklists already used by some applications like web browsers. Note

that in order to ensure true continuity of use key continuity authorities should re-check keys at random intervals to ensure that a site and its key really are around for the entire time interval and not just during the initial check and the final phish.

Use of an external key continuity authority carries with it a variety of aplication- and situation-specific considerations such as which external authority to trust, whether the communications need to be authenticated and/or encrypted or not, potential privacy concerns when using an external authority, tradeoffs of speed/throughput vs. security, and so on. A more detailed analysis may be found in [Perspectives].

4. Key Continuity Data Storage

[[Note: This may be better off in its own RFC, although since it's pretty cross-jurisdictional there's no obvious domain to put it under. The main motivation for including it here is to avoid a profusion of incompatible homebrew formats, with applications having to support a mass of oddball variants in order to authenticate keys]]

Applications require a standardised means of associating hosts with keys. The following text-based format, inspired by the /etc/passwd format, is recommended for easy exchange of key continuity data across applications. The format of the key data file using using Augmented BNF [ABNF] is as follows.

```
keydata = keydef | comment | blank

keydef = algorithm ":" key-hash ":" service ":" host ":" port rfu CRLF

comment = "#" *(WSP / VCHAR) CRLF

blank = *(WSP) CRLF

algorithm = *(ALPHA / DIGIT)

key-hash = *(HEXDIG)
```

```
service = *(ALPHA)

host = [wherever this is specified]

port = *(DIGIT) / ""

rfu = "" / ":" *(WSP / VCHAR)
```

The algorithm field contains the hash/fingerprint algorithm, usually "sha1". This allows multiple hash algorithms to be used for a fingerprint. For example while the current standard algorithm is SHA-1, some legacy implementations may require MD5, and future implementations may use SHA-1 successors.

The key-hash field contains the hash/fingerprint of the key. This value may be truncated as described in section 3. When comparing truncated hashes for equality, the first min(hash1_length, hash2_length) bytes of the two values are compared.

The service field specifies the service or protocol that the hash/fingerprint applies to. For example if both a TLS and and SSH server were running on the same host, the protocol field would be used to

distinguish between the key hashes for the two servers.

The host-name and (optional) host-port fields contain the host name and port that the key corresponds to. Typically the port is implicitly specified in the service field, but it may also be explicitly specified here.

For example a typical key continuity data file might consist of:

```
# Sample key continuity data file

sha1:B65427F83CED23A70263F8247C8D94192F751983:tls:www.example.com:443
sha1:17A2FE37808F3E84:ssh:ssh.example.com:22
md5:B2071C526B19F27C:ssh:ssh.example.com:22
```

The first entry contains the fingerprint of an X.509 certificate used by the web server for www.example.com. The second and third entries contain the (truncated) fingerprint of the SSH key used by the server

ssh.example com, first in the standard SHA-1 format and then in the alternative MD5 format.

5. Discussion

The intent of this format is to follow the widely-used and recognised /etc/passwd file format with which many users will be familiar. The format has been kept deliberately simple in order to avoid designing a general-purpose security assertion language such as KeyNote [KeyNote] or SAML [SAML]. While this will no doubt not suit all users, it should suffice for most, while remaining simple enough to encourage widespread adoption.

There are two options available for storing the key-continuity data,

the single-file format described above, and one entry per file. The latter makes it possible to use mechanisms like rsync to update individual entries/files across systems, but leads to an explosion of hard-to-manage tiny files, each containing a little piece of configuration data. It also makes it impossible to secure the configuration data via mechanisms such as PGP or S/MIME. Finally, the number of users who would use rsync to manage these files, when compared to the total user base, is essentially nonexistent. For this reason the single-file approach is preferred.

6. Security Considerations

Publishing a BCP on the topic of key/trust verification may make the authors a lightning rod for complaints of the form "this is just

pretend security, you really need a <insert sender's favourite authentication system>".

7. IANA Considerations

This document has no IANA Actions.

8. References

8.1. Normative References

[ABNF] Crocker, D. and P. Overell, "Augmented BNF for Syntax Specifications: ABNF", RFC 4234, October 2005.

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

8.2. Informative References

[AddressHarvest]
Schechter, S., Jung, J., Stockwell, W., and C. McLain, "Inoculating SSH Against Address-Harvesting", Proceedings of the 2006 Network and Distributed Systems Security Symposium 2006, February 2006.

[Duckling1]
Stajano, F. and R. Anderson, "The Resurrecting Duckling: Security Issues in Ad-Hoc Wireless Networking", Proceedings of the 7th International Workshop on Security Protocols 1999, Springer-Verlag Lecture Notes in Computer Science 1796, April 1999.

[Duckling2]
Stajano, F., "The Resurrecting Duckling - What Next?", Proceedings of the 8th International Workshop on Security Protocols 2000, Springer-Verlag Lecture Notes in Computer Science 2133, April 2000.

[Fingerprints]
"Fuzzy Fingerprints: Attacking Vulnerabilities in the Human Brain", October 2003.

[Hardening]
Xia, H. and J. Brustoloni, "Hardening Web Browsers Against Man-in-the-Middle and Eavesdropping Attacks", Proceedings of the 14th International World Wide Web Conference 2005, May 2005.

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

[KeyNote] Blaze, M., Feigenbaum, J., Ioannidis, J., and A. Keromytis, "The KeyNote Trust-Management System Version 2", RFC 2704, September 1999.

[NotDead] Gutmann, P., "PKI: It's Not Dead, Just Resting", IEEE Computer August 2002, August 2002.

[OpenPGP] Callas, J., Donnerhacke, L., Hal, H., Shaw, D., and R. Thayer, "OpenPGP Message Format", RFC 4880, November 2007.

[Perspectives] Wendlandt, D., Andersen, D., and A. Perrig, "Perspectives: Improving SSH-style Host Authentication with Multi-Path Probing", Proceedings of the 2008 USENIX Annual Technical Conference 2008, June 2008.

[Phishing] Jakobsson, M., Ed. and S. Myers, Ed., "Phishing and Countermeasures", 2007.

[SAML] "Security Assertion Markup Language (SAML), Version 2.0", 2008.

[SSH] Ylonen, T. and C. Lonvick, "The Secure Shell (SSH) Transport Layer Protocol", RFC 4253, Janaury 2006.

[TLS] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol", RFC 4346, April 2006.

[Usability] Gutmann, P., "Security Usability", September 2008.

Author's Address

Peter Gutmann
University of Auckland
Department of Computer Science
New Zealand

Email: pgut001@cs.auckland.ac.nz

Full Copyright Statement

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.