Belegarbeit

Implementation of IETF Syslog Protocols in the NetBSD `syslogd`

von

Martin Schütte

Universität Potsdam
Institut für Informatik
Professur Betriebssysteme und Verteilte Systeme

Potsdam

17. August 2009

Contents

1 Introduction

1.1 BSD Syslog

BSD Syslog is the *de facto* standard for event logging on Unix-like systems. It is a collection of an API, a message format, a simple network protocol, and a daemon to collect or forward the log data.

The only formally specified part is the API which is part of the POSIX standard [IG04]. Because the daemon was included in 4.3BSD (the influential Berkeley Standard Distribution of Unix), many systems use the same codebase for their syslogd with only minor modifications or extensions. Likewise, the message format and network protocol were never formally specified but convention and compatibility requirements established a *de facto* standard as later described in RFC 3164 [Lon01].

1.2 Google Summer of Code Project

The described implementation was part of the Google Summer of Code 2008. Google Summer of Code is a global program that offers student developers stipends to write code for various open source software projects. It asks cooperating open source projects to name mentors and lets students apply with project proposals to contribute to one of the projects.

Every accepted student is paired with a mentor. The project's progress towards its deliverables is monitored with two evaluations (mid-term and final) from both the student and the mentor.

My project was to improve the syslogd in NetBSD by implementing the IETF standards. So the basis was the NetBSD version of `syslogd(8)` and `syslog(3)` (which evolved from the original 4.3BSD) in July 2008[1]. The project was successful and met its deliverables; the resulting code was committed to the NetBSD CVS repository in October 2008.

1.3 Conventions

Code samples and text references to program elements (data types, function names, system calls, etc.) are set in `typewriter` font. System calls, C standard library functions, and system utilities are set with their respective manpage section, e. g. `syslog(3)`, `ls(1)`.

Because the term "syslog" has multiple meanings I will distinguish between "BSD Syslog" for the whole logging system, "syslogd" for some syslog daemon, "`syslogd(8)`" for the actual NetBSD syslog daemon, and "`syslog(3)`" for the API used by user programs.

References to source code are given with filename and line number, e. g. `syslogd.c:291`. The described codebase is the `syslogd(8)` and `syslog(3)` source from NetBSD-Current development branch available from the NetBSD CVS.[2]

1.4 `syslogd(8)` overview

The `syslogd(8)` daemon has a simple program structure built around a central event loop, waiting for messages and processing them one after another as they arrive.

[1] The version included in the NetBSD 5.0 release.

[2] Also accessible by WWW under `http://cvsweb.netbsd.org/bsdweb.cgi/src/usr.sbin/syslogd/` and `http://cvsweb.netbsd.org/bsdweb.cgi/src/lib/libc/gen/syslog.c`.

The main data structure is a linked list of `struct filed` (cf. Listing 1) with information on every destination. At program startup (or restart, cf. `init()`), it is filled from the configuration file; every line of the configuration leads to the creation of one `struct filed` object holding the selector, the destination, and minimal state information for that destination.

Once initialized, the program enters the event loop: in older versions this is basically a set of signal handlers and a `select(2)` system call. In NetBSD, this was replaced with the kernel event notification mechanism (`kqueue(2)`) that unifies the interface for I/O, signals, and timer events. As part of my implementation, I replaced the use of `kqueue(2)` with calls to the libevent library [Pro] which provides a more portable interface for event handling. On NetBSD it will still use `kqueue(2)` internally, but it allows for easier porting to systems with other event notification mechanisms like FreeBSD (which has a slightly different `kqueue(2)` interface) or Linux (where the `epoll(4)` system call would be preferable).

From this event loop (contained in the libevent call to `event_dispatch()`) the daemon waits for messages to arrive (from the kernel, the local log socket, or – if in server mode– from the network). Every message is parsed (`printline()`, cf. section 2), its configured destinations are found (`logmsg()`), it is formatted, and finally sent to every destination (`fprintlog()`)). Besides incoming messages, there are two other events: a timer event for periodic tasks (calling `domark()`) and signals to cause a program shutdown (on `SIGTERM`, calling `die()`) or a restart (on `SIGHUP`, calling `init()`).

2 Syslog Protocol

2.1 Format Description

The traditional BSD Syslog message format is described in RFC 3164 [Lon01] and uses a priority, timestamp, hostname, processname, and free text description for every event. (With many different conventions and variations.) Convention and backward compatibility limits every event to a length of 1024 octets and exclusive use of ASCII text.

To overcome these limitations, a new format – the syslog protocol [Ger09] – was created. It defines eight fields for every syslog event: priority, version, timestamp, hostname, application name, process ID, message ID, structured data, and free text message. Except for priority and version, all fields are separated by a space. Except for the free text message, all fields are mandatory for every event (possibly using a dash to indicate missing values).

The main improvements of the syslog protocol over the BSD Syslog format are the extended timestamp and the Structured Data field. Timestamps now have ISO format (as defined in RFC 3339 [KN02]) and include year, timezone, and time of day (optionally with sub second resolution[3]). Structured Data is a simple markup that allows one to use attributes (qualified by namespace) with a fixed semantic in syslog messages. Minor changes include the preferred use of fully qualified domain name (FQDN) instead of unqualified hostnames, the new Message-ID field, and the option to send UTF-8 encoded messages.

Figure 1: Examples of syslog messages.
In BSD Syslog (top) and in syslog protocol format (below).

```
<34>Oct 11 22:14:15 mymachine su: 'su root' failed for lonvick on /dev/pts/8
<38>Mar 17 21:57:57 frodo sshd[701]: Connection from 211.74.5.81 port 5991
<52>Mar 17 13:54:30 192.168.0.42 printer: paper out

<34>1 2003-10-11T22:14:15.003Z mymachine.example.com su - ID47 -↩
'su root' failed for lonvick on /dev/pts/8
<165>1 2003-10-11T22:14:15.003Z frodo.example.com evnts - -↩
[exampleSDID@0 iut="3" eventID="1011" eventSource="Application"]↩
An application event log entry
```

2.2 Input Parsing

After a message is received it is passed to `printline()` (or `printsys()` in case of kernel messages) which determines its format and calls either `printline_kernelprintf()`, or `printline_bsdsyslog()`, or `printline_syslogprotocol()`. These split the message into its fields and create a `msg_buf` (cf. Listing 2) object which is used for all further processing. Because every message has only one `msg_buf` object that may be sent to multiple destinations and stored in send queues its memory allocation and deallocation is not trivial. Thus it includes a reference counter and has its own set of auxiliary functions and macros[4].

2.3 Output Formatting

Output formatting is handled by the function `format_buffer()` and build around a call to `snprintf(3)` to fill all message fields with their respective values.

The output format (either BSD Syslog or syslog protocol) is set globally for all destinations. This allows some optimizations in the process so that every message (every `msg_buf` object) is only formatted once. For this a message is always formatted for TLS output with its length as a prefix (cf. 4.2). Depending on the destination type the resulting string is sent completely (TLS), with an offset to skip the length prefix (UDP), or with an offset to skip the length prefix and the priority field (all other types)[5].

3 Buffering

3.1 Requirements

The original syslogd had no notion of buffering a message: as soon as a message arrived it was immediately forwarded (by UDP or local pipe) or written to file. All destinations were either always available (UDP, wallmsg), only available until an error occurred (console,

[3] The second may have up to six decimal places, resulting in a maximum resolution of one microsecond.
[4] `buf_msg_new()`, `buf_msg_free()`, `NEWREF()`, and `DELREF()`
[5] Cf. `fprintlog()` in `src/usr.sbin/syslogd/syslogd.c:2223ff`.

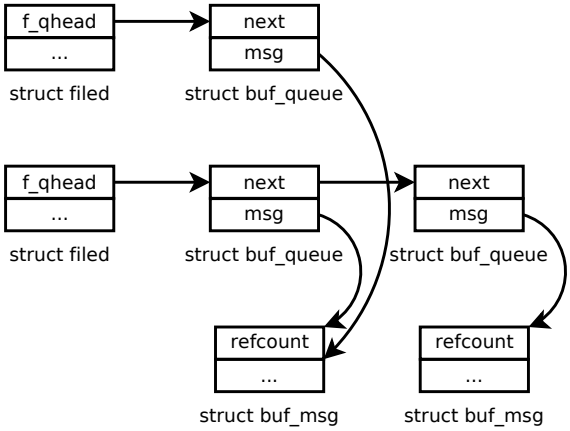

Figure 2: data structures for message buffers and queues

tty, files), or could be quickly reopened after an error (pipes[6]).

TLS is the first syslog transport that can be temporarily unavailable – either because the last message is still being sent (this can happen because TLS operations are non-blocking and might only process parts of the message) or because the network connection is lost. To assert reliable delivery in these cases the TLS implementation added a send queue and a message buffer to `syslogd(8)` which are active for TLS, file, and pipe destinations. This buffer is designed to bridge short network outages by storing messages in memory and delivering them as soon as the destination becomes available again.

3.2 Implementation

The implementation uses a per-destination send queue of message references (cf. Listing 3). Because the queue preserves the correct order for every destination, there is no need to copy the message itself; every message (`buf_msg` object) exists only once and may be referenced by multiple send queues.

To prevent memory exhaustion, queues have fixed limits both for the number of entries and for the memory usage of all referenced messages. These limits are periodically checked (as part of the `domark()` function). If too many messages accumulate, the queue will be purged (`message_queue_purge()`).

The central message delivery function `fprintlog()` takes the queue entry as an optional argument to distinguish between new and already queued messages. Now every new TLS message is first added to the destination's send queue and before it is passed to `tls_send()`. It is removed from the send queue only after a message is successfully written (check in `dispatch_tls_send()` and removal via `free_tls_send_msg()`).

[6] If the destination process for a pipe does not exist it is restarted, if it still does not exist the second time, a permanent error is assumed and the destination is removed.

4 TLS

4.1 Overview

Traditional BSD Syslog used the User Datagram Protocol (UDP) to forward messages to other hosts. Although that was appropriate in the 1980s, it does not meet current demands in security and reliability. This lead to different (and incompatible) implementations using the Transmission Control Protocol (TCP) as transport protocol [Scha, Ger], thus providing a reliable transport that can also be tunneled through SSL for added security and authentication.

For the new syslog standards, the IETF Working Group decided not to specify a TCP, but a TLS transport [MMS09]. The Transport Layer Security (TLS) protocol provides privacy and data integrity between two communicating applications [DR08]. The building blocks for these goals are mutual authentication, encryption, reliable transport (usually using TCP), and message authentication codes.

RFC 5425 (TLS transport) basically defines two parts: the encapsulation of messages and authentication policies between client and server.Because `syslogd(8)` previously did not have support for reliable connections, it was also necessary to add some connection control code and a buffering scheme to preserve messages for the time of connection loss.

4.2 Record Format

TLS provides a stream oriented connection (like TCP) whereas BSD Syslog is a record (or datagram) based protocol. Therefore the transmission of syslog messages over TLS requires an encapsulation of datagrams into a stream.

The two common solutions are to use a record marker or a length prefix. A marker simply splits two records by inserting a dedicated symbol between them, e. g. the NULL byte or the ASCII newline. This limits the kind of payload because the marker symbol must not be part of the data itself (or has to be replaced by some escape sequence). One advantage of this method is its self-synchronization, because a receiver can simply skip a damaged record by discarding everything until the next marker and continue to read the next record.

RFC 5425 uses the length prefix: every message is prefixed by its length in octets (encoded as a decimal ASCII string). After reading the prefix a receiver will know the length of the message and can read it without examining its content. The added processing has two main advantages: it allows the receiver to allocate enough memory beforehand (or alternatively to easily skip messages that do not fit in its available memory) and it allows the transport of arbitrary data without special consideration for unallowed marker signals.

The implementation to add this prefix at the sender is quite simple and uses an additional call to `sprintf(3)` to determine the message length (`bool format_buffer()`). The receiver uses a dynamically sized input buffer (one for every incoming connection) and will skip messages that will not fit into available memory (`void tls_split_messages()`).

4.3 Library

This implementation uses the OpenSSL library [Ope] for all functions related to TLS connections, X.509 certificates and cryptographic operations because it is available under

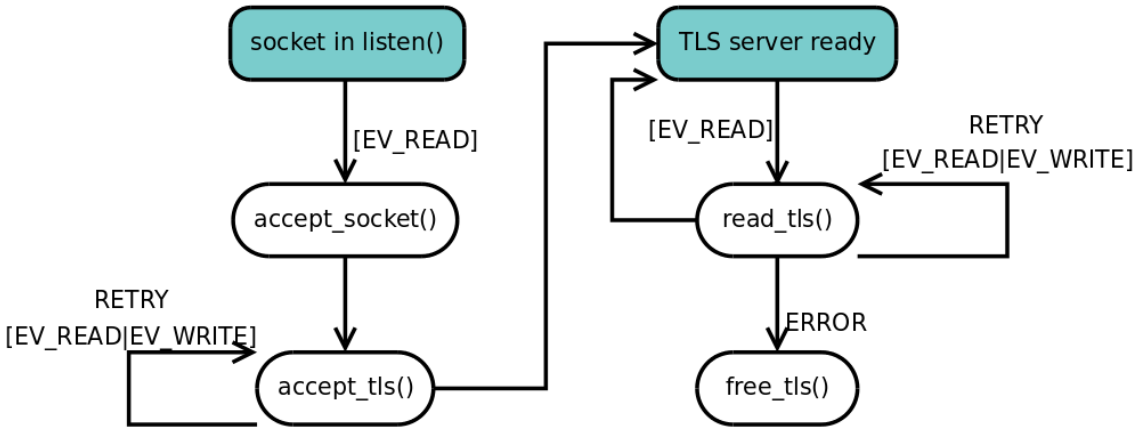

Figure 3: States of incoming TLS connections

a BSD compatible licence and already part of NetBSD.

One important pattern in accepting and reading from TLS connections is the separation between the underlying TCP and the TLS on top of it. Because the operating systems only controls the TCP connection, the event notification occurs as soon as data arrives at the TCP socket – this might or might not be enough to start processing at the TLS level. For example, an incoming TLS connection will pass through the following stages: as soon as a TCP connection arrives, libevent will notify the application by calling `dispatch_socket_accept()`. This function contains the `accept(2)` system call to establish the TCP connection. Only then is the SSL object initialized and `dispatch_tls_accept()` is called to establish the TLS connection with `SSL_accept()`.

A look at `dispatch_tls_accept()` immediately shows another pattern used throughout the TLS code: all TCP sockets and TLS operations are used in non-blocking mode. An incomplete TLS operation will return an error code of `TLS_RETRY_READ` or `TLS_RETRY_WRITE` to indicate it has to read or write additional data. In this case, a new event is registered (using the `tls_conn_settings->retryevent` object) in order to execute the operation again as soon as the underlying socket is ready to read or write more data. As a result every event callback (`dispatch_tls_*`) is called as often as necessary until it finishes its task.

4.4 Connection management

Naturally syslogd manages two kinds of connections: incoming connections from clients (when in server mode) and outgoing connections to servers. Every connection is associated with a `tls_conn_settings` object (cf. Listing 4) to hold all TLS specific connection attributes, most importantly the authentication information and the connection state.

For incoming connections (cf. Figure 3) all configuration options (the own key and certificate files as well as the list of client certificates to accept) are stored in the global configuration object of `struct tls_global_options_t` (initialized by `read_config_file()`). The `tls_conn_settings` object is created and filled when the TCP and TLS connections

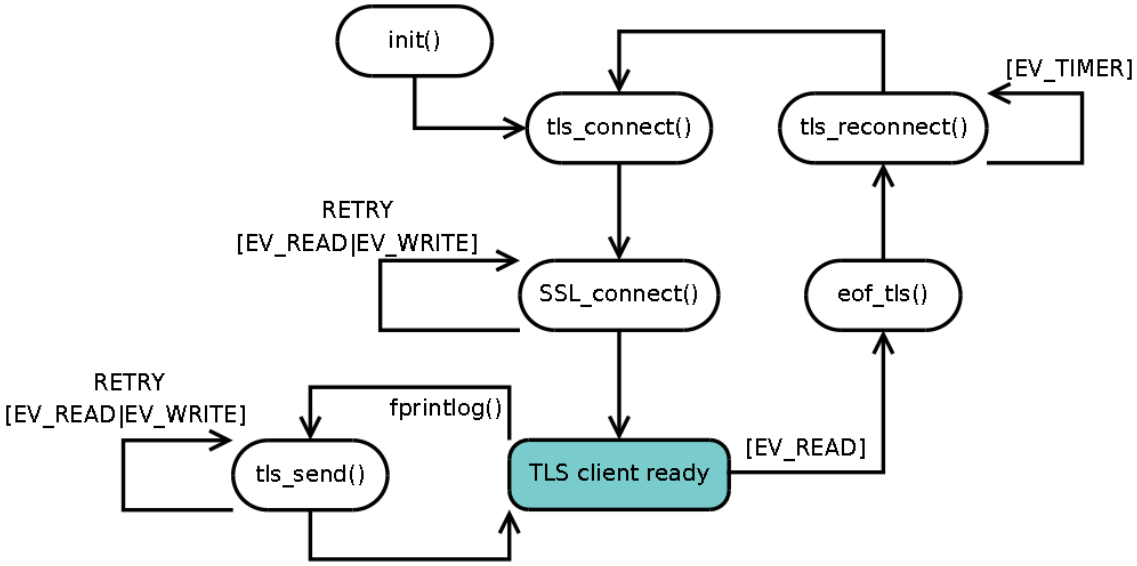

Figure 4: States of outgoing TLS connections

are accepted (in `dispatch_socket_accept()` and `dispatch_tls_accept()`). The new connection is also registered with libevent, so any incoming data will be handled by the callback function `dispatch_tls_read()`. This callback is responsible for reading data from the non-blocking sockets (with several retries if necessary), for processing incoming messages (through `tls_split_messages()`), and for closing the connection when an error occurs. Closing a TLS connection requires several other actions: calling `SSL_shutdown()` (several times in non-blocking mode), unregistering all event handlers, and freeing all memory objects; all these steps are taken care of by the `free_tls_conn()` function.

The life cycle of outgoing TLS connections (from clients connecting to a remote server) is considerably more complex (cf. Figure 4) because the client has to reconnect after network errors. In this case, every configuration line with a TLS destination results in a `tls_conn_settings` object as part of the `filed` destination description and the fields (hostname, expected certificate, etc.) are initialized in `parse_tls_destination`. TCP and TLS connections are established immediately in the `init()` function (i. e. a TLS destination will be connected even if no messages are send) with a call to `tls_connect()`. The connect and write operations are structured just like the accept and read operations on incoming connections (also using non-blocking sockets and event-driven retries). The biggest difference is the reaction when the connection is lost, which might be either because the socket is closed (handled by the `dispatch_tls_eof()` callback) or because of some serious error on the TLS layer. In both cases a timer event is registered to schedule a call to `tls_reconnect()`. The interval between reconnects is initially 10 seconds[7] and increases exponentially until either `tls_connect()` is successful or a maximum timespan of 2 hours[8] is reached and the destination is marked as permanently unavailable.

[7] `TLS_RECONNECT_SEC` in `src/usr.sbin/syslogd/tls.h:62`.
[8] `TLS_RECONNECT_GIVEUP` in `src/usr.sbin/syslogd/tls.h:73`.

4.5 Authentication

An important part of the TLS specification is the description of different policies and criteria of mutual authentication between client and server.

The recommended method is to use a Certificate Authority (CA) as a trust anchor that has signed all other certificates in use. Then every host can check the signature on its peer's certificate and a certificate is trusted if a trust chain exists between CA and certificate. Because even a signed certificate should only be used by only one computer, it is common to additionally compare the peer's hostname with the `commonName` and the `subjectAltName` contained in the certificate. In `syslogd(8)`, this check is only performed by a client that verifies a server certificate.

As an alternative for small sites without an own PKI and CA, it is also possible to use copies of trusted certificates or certificate fingerprints (i. e. the hash value of a host's certificate). In this case the server can have a list of trusted certificates or fingerprints (configuration keywords `tls_allow_clientcerts`/`tls_allow_fingerprints`) while the client can set one certificate and one fingerprint for every server it connects to.

The checks themselves are performed in function `check_peer_cert()`, which is a callback function executed from `SSL_connect()` and `SSL_accept()`. If the peer's certificate is signed by a CA, the callback is called for every certificate in the trust chain and the last return value determines whether the connection is established or rejected.

5 API

5.1 `syslog(3)`

The API to log events from user programs and applications is the `syslog(3)` function in the system's standard library. It is specified by the POSIX standard [IG04] and available on all Unix-like operating systems.

The function's signature is

```
void syslog(int priority, const char *message, ...);
```

The `priority` is usually set using constants from `<sys/syslog.h>` and the `message` is a `printf`-style format string followed by a variable number of arguments. For example:

```
syslog(LOG_DEBUG, "Connection from host %d", CallingHost);
syslog(LOG_INFO|LOG_LOCAL2, "foobar error: %s in line %d",
        errmsg, __LINE__);
```

When an application calls `syslog(3)` with its log message, the library function will add the metadata (timestamp, hostname, process name), format a complete BSD Syslog message, and send it to the local syslogd daemon. The connection between `syslog(3)` (which is executed as part of the user's process) and the syslogd is not specified, but the *de facto* standard is to use a local Unix domain socket in datagram mode, owned by the superuser and with a name defined in `<sys/syslog.h>`. That way all (unprivileged) processes can send syslog messages and only a privileged syslogd process can receive and process them.

5.2 Changes in syslog(3)

Neither the interprocess communication mechanism nor the message format between `syslog(3)` and `syslogd(8)` are formally specified. So they may be changed just like the internal implementation of `syslog(3)` as long as the API is preserved and the library is always compatible with the system's syslogd. In case of such changes the syslogd should not only be compatible with the current standard library but also be able to receive messages from previous library versions, because there may always be statically linked programs still using the old mechanism.

Part of this GSoC project was to change the `syslog(3)` library. This means the new function (implementation in `vsyslogp_r()`) will generate full timestamps, use a full hostname, and format the message according to the new syslog protocol format (all new fields, i. e. message ID and structured data, are left empty).

In this case the necessary conditions are satisfied because the `syslogd(8)` understands both message formats and because all components (`syslog(3)` and `syslogd(8)`) are part of the NetBSD operating system, so they are distributed and updated together. If standard library and syslogd are maintained by different organizations, the transition becomes more difficult, e. g. one cannot use an older version of syslog-ng [Scha] as an alternative syslogd because it would not understand messages in syslog protocol.

5.3 New function syslogp(3)

Because applications cannot use `syslog(3)` to add a message ID or structured data to log messages, there has to be a new and extended function for syslog protocol messages.

This function was named `syslogp(3)` and takes additional arguments for the message ID and structured data[9]; its signature is

```
void syslogp(int priority, const char *msgid, const char *sdfmt,
        const char *message, ...);
```

The functionality is intentionally kept at the pure minimum necessary to provide access to the new event message fields. All three `char*` parameters are format strings; for unused fields the parameter is given as `NULL`. Thus the call `syslog(LOG_INFO, message, ...)` is equivalent to the call `syslogp(LOG_INFO, NULL, NULL, message, ...)`[10]. Other examples:

```
syslogp(LOG_DEBUG, "e%d", NULL, "context %d: errno %d",
        errID, cid, errno);
syslogp(LOG_INFO, NULL, "[err@0 context=\"%d\" errno=\"%d\"]", NULL,
        cid, errno);
```

Like the other change, this function is now available on NetBSD-current only. It is intended to propose it for implementation in other system's standard libraries once it has proven itself to be useful and more syslogd implementations support the syslog protocol.

[9] For its implementation see `vsyslogp_r` in `src/lib/libc/gen/syslog.c:224`.

[10] That is basically the function definition; internally both `syslog(3)` and `syslogp(3)` are wrappers to prepare different arguments for the actual functionality in `vsyslogp_r()`. And while `syslogp(3)` does pass three format strings `syslog(3)` will only pass the message format string and set the other two to `NULL`.

Figure 5: Syslog-sign outline

Sent:	**Received:**	**Verified:**
CB(public key)	CB(public key)	
message 1	message 1	$\#_1$: message 1
message 2	message 3	$\#_2$: message 2
message 3	message 2	$\#_3$: message 3
message 4	message 5	$\#_4$: *missing*
SB($\#_1, \#_2, \#_3, \#_4$)	SB($\#_1, \#_2, \#_3, \#_4$)	
message 5	message 7	$\#_5$: message 5
message 6	message 8	$\#_6$: message 6
message 7	message 6	$\#_7$: message 7
message 8	SB($\#_5, \#_6, \#_7, \#_8$)	$\#_8$: message 8
SB($\#_5, \#_6, \#_7, \#_8$)		

A simplified outline of syslog-sign: The message sender adds to its messages a certificate block (CB) and signature blocks (SB) containing message hashes ($\#_x$). With UDP the server might not receive all messages or receive them out of order. But it can use the hashes to infer the correct message sequence and fill in all received and verified messages.

6 Syslog-Sign

Syslog-sign is a protocol to digitally sign syslog messages [KCC09]. This complements syslog with end-to-end message integrity, authenticity, and ordering (including the detection of lost messages). Whereas the TLS transport provides these properties (as well as a reliable connection without message loss) for point-to-point connections between two hosts, the assurances of syslog-sign hold also for UDP connections and across relay chains with multiple hops.

Unlike other schemes to secure log data (e. g. based on message authentication codes and encrypted log files [BY97, SK99]), syslog-sign works as an overlay and does not require a new or special message format, thus it is backward compatible to existing log infrastructures.

6.1 Overview

The two basic design elements of syslog-sign are: (a) Hashing every message, thus forming a stream of hashes corresponding to the stream of messages and (b) injecting special control messages into the message stream. These messages which distribute the key material and message hashes are the ones that include the signatures.

The sequence of hashes asserts the messages' integrity and order; signing the control messages asserts the authenticity of the sequence.

A receiver can later verify a sequence of messages by extracting the syslog-sign control messages, checking the used public key, checking the signatures of all control messages, and finally comparing its sequence of received messages with the authenticated sequence of message hashes. This verification can happen immediately at its reception (online) to

verify the network transport or at some later time (offline) to verify the logfile integrity.[11] `syslogd(8)` does not implement any verification of received syslog messages. But a simple offline verifier was implemented in Perl to test the correct implementation of hashes and signature.[12]

6.2 Signature Groups

Nearly every discussion about syslog messages already implies the notion of a message stream, one ordered sequence of messages that is collected by a syslogd, sent over the network, written into logfiles, and so on.

Under more careful consideration this is not the case, because even simple syslog configurations yield multiple message streams going to different destinations. For example a mailserver might have a `syslog.conf(5)` like the following to write messages to one mail related logfile, one general logfile, and a central logserver:

```
mail.*              /var/log/maillog
*.*;mail.none       /var/log/messages
*.*                 @logserver.example.net
```

To cover this problem, syslog-sign introduces the concept of signature groups which are sets of messages that are signed together. Basically every signature group forms one stream, one sequence of messages to be authenticated with syslog-sign (independently from all other signature groups). This allows a complete verification with multiple destinations, each receiving only a subset of messages.

The possibilities to define signature groups are rather limited: The Internet Draft defines three strategies to divide all messages into Signature Groups and one option for implementation defined associations:

0 one global Signature Group signing a message stream of all messages,

1 use 192 Signature Groups, one per priority,

2 associate Signature Groups with user-defined ranges of priorities,

3 this value is reserved for implementation-specific strategies.

So in the example configuration given above one could use `SG=0` to verify all messages on `logserver.example.net` – but verifying the local logfile `/var/log/messages` would fail because all mail-related messages were missing.
With `SG=1` every single priority value (of which there are 192) is associated with its own signature group; thus in our example above the 8 signature groups corresponding to facility mail (with the 8 severities form debug to emergency) are written to `/var/log/maillog`, the 184 other priorities are written to `/var/log/messages`, and `logserver.example.net` would receive all 192 signature groups. Thus all destinations receive multiple Signature Groups and all messages can be verified.
With `SG=2` one can define ranges of priorities; for our example we would define three

[11] A detailed description with algorithms for online and offline verification can be found in the Internet Draft [KCC09, section 7].

[12] The verifier is not yet included in NetBSD and only available from SVN at https://barney.cs.uni-potsdam.de/svn/syslogd/trunk/src/syslogd/verify.pl.

signature groups: one for the range 0-15 (containing the facilities kernel and user), one for range 16-23 (facility mail), and one for range 24-191 (for all other facilities). As a result messages can be verified at all destinations but less Signature Groups are used.
`syslogd(8)` also implements a custom strategy for `SG=3`: in this case one signature group is associated with every configured destination, i. e. every line in `syslog.conf(5)`. This makes it the only configuration in which signature groups may be non-disjunct (a message may be in more than one signature group) and which may be defined by other attributes besides the priority (because a selector may only apply to certain programs or hostnames). It follows that with `SG=3` every destination receives exactly one Signature Group that authenticates all messages.

All Signature Groups are also defined by their originator, the tuple (`HOSTNAME`, `APP-NAME`, `PROCID`), so that a receiver can distinguish between multiple Signature Groups from multiple signing applications on the same host. For the same reason messages passing multiple relays may be signed by several different applications. As long as all signatures are verified in order (with the last signatures first) this will not cause any problems.[13]

6.3 Implementation

As soon as `syslogd(8)` is started and configured to sign messages, `sign_global_init()` is called to initialize the global settings in `struct sign_global_t` (cf. Listing 5). If TLS uses DSA keys, the same public/private key pair is used for signing as well, otherwise a new key pair is generated. After keys are loaded and OpenSSL contexts are allocated, the signature groups are initialized.

Signature Groups are implemented using a linked list of `struct signature_group_t` (cf. Listing 6). Depending on the configuration for syslog-sign, the required number of `signature_group_t` objects are created and initialized in `sign_sg_init()`. Every `signature_group_t` object contains a queue of strings containing the message hashes of that Signature Group. It is also bidirectionally linked to all `struct filed` destinations possibly associated with it; this makes it easy to find the Signature Group(s) for a given message in `sign_get_sg()`.

At runtime the syslog-sign code is called from `fprintlog()`: just before sending a message its hash is computed (`sign_msg_hash()`) and appended to the right Signature Group (`sign_append_hash()`); just after sending it, it is checked whether a Signature Block should be sent as well (`sign_send_signature_block()`).

Because syslog-sign may be used to secure UDP connections against packet loss it is always possible that control messages are lost as well; to strengthen the protocol `syslogd(8)` adds redundant control messages. Certificate Blocks with public keys are sent multiple times.[14] Signature Blocks with message hashes are sent using the queue of hashes and a sliding window algorithm so that every message hash is included in several Signature Block messages.[15] The check in `sign_send_signature_block()` only considers the length of

[13] On the other hand, if the first signature is verified first then all syslog-sign control messages with later signatures should be detected as additional, non-signed messages and cause a warning.

[14] By default twice, configurable at compile time by setting the constant `SIGN_RESENDCOUNT_CERTBLOCK` in `src/usr.sbin/syslogd/sign.h:84`.

[15] By default in three messages, configurable at compile time with `SIGN_RESENDCOUNT_HASHES` in `src/usr.sbin/syslogd/sign.h:85`.

the queue of hashes, thus after every n messages a Signature Block is sent.[16] If a Signature Group contains very few messages then it can take some time until a Signature Block is created to authenticate the messages, possibly causing problems when a receiver uses online verification and does not process the messages their signature is checked. Thus it is desirable to add a timer event to `signature_group_t` objects and make sure every log message is authenticated with a Signature Block after a certain amount of time has passed.

At program shutdown or restart the function `sign_global_free()` is called. It will always send last Signature Blocks for all Signature Groups and then free all syslog-sign related objects.

6.4 Security Properties

The following is a brief investigation of possible attacks against syslog integrity and the protection provided by syslog-sign (and possible other measures). It is based on the assumptions that (a) the algorithms used for hashing and signing are cryptographically secure, i. e. it is effectively impossible to find a second message for a given hash value or to forge a signature without access to the private key, and (b) the adversary is able to modify the local logfiles at will[17].

The basic function of syslog-sign is to chain multiple messages into a signed sequence (a signature group), so that all message modifications and deletions inside that sequence are detected. Thus in order to modify log messages an adversary cannot simply edit single messages in the logfile but can only delete complete Signature Groups or replace complete Signature Groups with forged ones (containing fabricated messages, signed with a new key).

A first configuration advice to increase security is to use one public/private key pair per host (or per signing application) and have it signed by a central authority. Then only users with access to the correct private key are able to forge Signature Groups; whereas a Signature Group signed by an ad hoc key could be replaced by a forged Signature Group signed by another ad hoc key.

Because syslog-sign does not define an "end of session" control message it is possible to unnoticeably truncate a Signature Group by (recursively) deleting the last Signature Block and all messages which are only covered by hashes in that Signature Block.[18] In configurations with more than one Signature Group (i. e. other than `SG=0`, cf. 6.2) the Signature Groups are cross linked by their Global Block Counter, so one would have to truncate all Signature Groups simultaneously (by removing Signature Blocks in order of decreasing Global Block Counter values).[19] In practice a truncation could be detectable

[16] In the source code n is defined as `SIGN_HASH_DIVISION_NUM` in `src/usr.sbin/syslogd/sign.h:110`; it is determined at compile time and depends on the hash algorithm, the desired message length, and the redundancy setting.

[17] For simplicity this discussion is limited to the protection of stored logfiles on a log server. From a network perspective the ability to modify the local logfile is equivalent to controlling a network connection between client and log server, e. g. a successful man in the middle attack.

[18] Depending on the log type and event frequency the deletion might or might not be detectable by a gap in event timestamps.

[19] Verifying the Global Block Counter requires all messages from the signing application. So in practice it is not always possible to verify this, e. g. if different Signature Groups are intentionally processed independently on different log servers.

by implementation specific behaviour of the signing application to indicate the "end of session" status; e. g. when `syslogd(8)` is stopped or restarted it always sends two last Signature Blocks with fewer hashes than normal.

One way to increase the protocol's strength is to add cross linkage between Signature Groups and between Reboot Sessions. For example a signing application can save the last signature at program shutdown and add it (as an additional structured data element) to the first Certificate Block at the next program start. Verifying these links between Signature Groups will detect the Deletion and Truncation of Signature Groups; Replacing a Signature Group becomes more difficult because the first Certificate Block of the following Signature Group has to change as well.

Another even more promising mechanism is the use of a timestamping server to authenticate the time and date of signatures. This makes log manipulation considerably more difficult because it has to be done at roughly the same time as the real log creation itself and later manipulation of old logfiles can always be detected. It also adds an external confirmation on the correctness of the logged timestamps.

As a result syslog-sign provides a strong protection for log data and the biggest remaining vulnerability is that of a Denial of Service attack. If the log system is divided into different administrative domains (e. g. using an external timestamping server or having different administrators for production and log servers), then even a system administrator cannot modify the logs unnoticedly – an important property if log data is used as legal evidence or proof of policy compliance.

7 Open Issues

7.1 RFC Compliance

Some specification details were changed after the initial implementation, so there are still some minor changes required for compliance with the RFCs.

A first issue occurred in the certificate check for TLS transport. Here the standard requires a wildcard matching so an X.509 certificate issued e. g. for `*.example.net` is valid for a host named `server.example.net` [MMS09, section 5.2].

A second issue came up when the IETF Working Group decided how to encode the signatures in syslog-sign messages. Because this was not specified for some time this implementation uses the PEM format as often used in an SSL/TLS context. The current draft requires an alternative encoding as used in OpenPGP context [KCC09, sections 4.2.8 and 5.3.2.8].

7.2 Software Architecture

The monolithic architecture of syslogd leads to increased complexity and decreased maintainability as more features are added and interact with each other. The extensive code changes from this project have already reduced the maintainability to a critical degree.

One option to increase the code quality is the modularization of output code, possibly encapsulating every output type in its own dynamic library with a standard interface. An alternative would be the separation of syslog-sign code into its own library, making it reusable for other signing applications.

7.3 Additional transports

Now that UDP is no longer the only common transport protocol for syslog, it becomes possible to consider other protocols as well. There are already Internet Drafts to use the Datagram Transport Layer Security (DTLS) protocol to use encryption and authentication over UDP. Another promising candidate for syslog is the Stream Control Transmission Protocol (SCTP), which not only combines some advantages of UDP and TCP but also offers new possibilities like multihoming and multiple streams.

7.4 Configuration

The format of the `syslog.conf(5)` configuration file has co-evolved with the codebase and has reached a similar limit of complexity. It would be best not to extend the old configuration format any further, but to introduce a new format well suited for multiple per-destination settings, global options, and extensibility for new output channels.

8 Summary

The Google Summer of Code project has been successful; the proposed deliverables were completed and the code was later committed to the NetBSD CVS.

In addition this is one of the first implementations of Syslog Protocol [Lon01] and TLS transport [MMS09], which made it possible to perform a first interoperability test between rsyslog [Ger] and `syslogd(8)` using the IETF protocols. It is also the first implementation of syslog-sign [KCC09]; it was used to generate the examples for the Internet-Draft and provided valuable input for the working group discussion.

The changes in the libc standard library made NetBSD-Current the first system to use syslog protocol internally (between libc and syslogd) and also the first system to provide an API for it (`syslogp(3)`).

Listing 1: `struct filed` (`src/usr.sbin/syslogd/syslog.h:335`)

```
335 struct filed {
        struct  filed *f_next;          /* next in linked list */
        short   f_type;                 /* entry type, see below */
        short   f_file;                 /* file descriptor */
        time_t  f_time;                 /* time this was last written */
340     char    *f_host;                /* host from which to record */
        u_char  f_pmask[LOG_NFACILITIES+1];     /* priority mask */
        u_char  f_pcmp[LOG_NFACILITIES+1];      /* compare priority */
#define PRI_LT  0x1
#define PRI_EQ  0x2
345 #define PRI_GT  0x4
        char    *f_program;             /* program this applies to */
        union {
                char    f_uname[MAXUNAMES][UT_NAMESIZE+1];
                struct {
350                     char    f_hname[MAXHOSTNAMELEN];
                        struct  addrinfo *f_addr;
                } f_forw;               /* UDP forwarding address */
#ifndef DISABLE_TLS
                struct {
355                     SSL     *ssl;                   /* SSL object  */
                        struct tls_conn_settings *tls_conn;
                        /* certificate info */
                } f_tls;                /* TLS forwarding address */
#endif /* !DISABLE_TLS */
360             char    f_fname[MAXPATHLEN];
                struct {
                        char    f_pname[MAXPATHLEN];
                        pid_t   f_pid;
                } f_pipe;
365     } f_un;
#ifndef DISABLE_SIGN
        struct signature_group_t *f_sg;     /* one signature group */
#endif /* !DISABLE_SIGN */
        struct buf_queue_head f_qhead;      /* undelivered msgs queue */
370     size_t                f_qelements;  /* elements in queue */
        size_t                f_qsize;      /* size of queue in bytes */
        struct buf_msg       *f_prevmsg;    /* last message logged */
        struct event         *f_sq_event;   /* timer for send_queue() */
        int                   f_prevcount;  /* repetition cnt of prevmsg*/
375     int                   f_repeatcount; /* number of "repeated" msgs*/
        int                   f_lasterror;  /* last error on writev() */
        int                   f_flags;      /* file-specific flags */
#define FFLAG_SYNC      0x01    /* for F_FILE: fsync after every msg */
#define FFLAG_FULL      0x02    /* for F_FILE | F_PIPE: write PRI header */
380 #define FFLAG_SIGN      0x04    /* for syslog-sign with SG="3":
                                   * sign the messages to this destination */
};
```

Listing 2: `struct buf_msg` (`src/usr.sbin/syslogd/syslogd.h:280`)

```
280 struct buf_msg {
        size_t   refcount;
        int      pri;
        int      flags;
        char    *timestamp;
285     char    *recvhost;
        char    *host;
        char    *prog;
        char    *pid;
        char    *msgid;
290     char    *sd;         /* structured data */
        char    *msg;        /* message content */
        char    *msgorig;    /* in case we advance *msg beyond header fields
                                we still want to free() the original ptr */
        size_t   msglen;     /* strlen(msg) */
295     size_t   msgsize;    /* allocated memory size   */
        size_t   tlsprefixlen; /* bytes for the TLS length prefix */
        size_t   prilen;       /* bytes for priority and version  */
};
```

Listing 3: `struct buf_queue` (`src/usr.sbin/syslogd/syslog.h:301`)

```
301 struct buf_queue {
        struct buf_msg* msg;
        STAILQ_ENTRY(buf_queue) entries;
};
305 STAILQ_HEAD(buf_queue_head, buf_queue);
```

Listing 4: `struct tls_conn_settings` (`src/usr.sbin/syslogd/tls.h:122`)

```
122 struct tls_conn_settings {
        unsigned      send_queue:1, /* currently sending buffer      */
                      errorcount:4, /* counter [0;TLS_MAXERRORCOUNT] */
125                   accepted:1,   /* workaround cf. check_peer_cert*/
                      shutdown:1,   /* fast connection close on exit */
                      x509verify:2, /* kind of validation needed     */
                      incoming:1,   /* set if we are server          */
                      state:4;      /* outgoing connection state     */
130     struct event *event;        /* event for read/write activity */
        struct event *retryevent;   /* event for retries             */
        SSL          *sslptr;       /* active SSL object             */
        char         *hostname;     /* hostname or IP we connect to  */
        char         *port;         /* service name or port number   */
135     char         *subject;      /* configured hostname in cert   */
        char         *fingerprint;  /* fingerprint of peer cert      */
        char         *certfile;     /* filename of peer cert         */
        unsigned      reconnect;    /* seconds between reconnects    */
};
```

Listing 5: `struct sign_global_t` (`src/usr.sbin/syslogd/sign.h:167`)

```
167 struct sign_global_t {
        /* params for signature block, named as in RFC nnnn */
        const char   *ver;
170     uint_fast64_t rsid;
        int           sg;
        uint_fast64_t gbc;
        struct signature_group_head SigGroups;
        struct string_queue_head    sig2_delims;
175
        EVP_PKEY     *privkey;
        EVP_PKEY     *pubkey;
        char         *pubkey_b64;
        char          keytype;
180
        EVP_MD_CTX   *mdctx;       /* hashing context */
        const EVP_MD *md;          /* hashing method/algorithm */
        unsigned      md_len_b64;  /* length of b64 hash value */

185     EVP_MD_CTX   *sigctx;      /* signature context */
        const EVP_MD *sig;         /* signature method/algorithm */
        unsigned      sig_len_b64; /* length of b64 signature */
};
```

Listing 6: `struct signature_group_t` (`src/usr.sbin/syslogd/sign.h:150`)

```
150 struct signature_group_t {
        unsigned                        spri;
        unsigned                        resendcount;
        uint_fast64_t                   last_msg_num;
        struct string_queue_head        hashes;
155     struct filed_queue_head         files;
        STAILQ_ENTRY(signature_group_t) entries;
};
STAILQ_HEAD(signature_group_head, signature_group_t);
```

Listings

List of Figures

References

[BY97] Mihir Bellare and Bennet Yee. Forward integrity for secure audit logs. Technical report, Computer Science and Engineering Department, University of California at San Diego, November 1997. Available from: http://www.loganalysis.org/sections/research/fi.pdf.

[DR08] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246 (Proposed Standard), August 2008. Available from: http://www.ietf.org/rfc/rfc5246.txt.

[Ger] Rainer Gerhards. Rsyslog. Available from: http://www.rsyslog.com/.

[Ger09] R. Gerhards. The Syslog Protocol. RFC 5424 (Proposed Standard), March 2009. Available from: http://www.ietf.org/rfc/rfc5424.txt.

[IG04] The IEEE and The Open Group. *The Open Group Base Specifications Issue 6 IEEE Std 1003.1*. The IEEE and The Open Group, 2004. Available from: http://www.opengroup.org/onlinepubs/009695399/.

[KCC09] J. Kelsey, J. Callas, and A. Clemm. Signed syslog Messages. Internet-Draft draft-ietf-syslog-sign-26, May 2009. Available from: http://tools.ietf.org/id/draft-ietf-syslog-sign-26.txt.

[KN02] G. Klyne and C. Newman. Date and Time on the Internet: Timestamps. RFC 3339 (Proposed Standard), July 2002. Available from: http://www.ietf.org/rfc/rfc3339.txt.

[Lon01] C. Lonvick. The BSD Syslog Protocol. RFC 3164 (Informational), August 2001. Obsoleted by RFC 5424. Available from: http://www.ietf.org/rfc/rfc3164.txt.

[MMS09] F. Miao, Y. Ma, and J. Salowey. Transport Layer Security (TLS) Transport Mapping for Syslog. RFC 5425 (Proposed Standard), March 2009. Available from: http://www.ietf.org/rfc/rfc5425.txt.

[Net] The NetBSD Project. The NetBSD Project. Available from: http://netbsd.org/.

[Ope] OpenSSL Project. OpenSSL: The open source toolkit for SSL/TLS. Available from: http://www.openssl.org/.

[Pro] Niels Provos. libevent – an event notification library. Available from: http://www.monkey.org/~provos/libevent/.

[Scha] Balázs Scheidler. syslog-ng logging system. Available from: http://www.balabit.com/network-security/syslog-ng/.

[Schb] Martin Schütte. Status Page NetBSD-SoC: Improve syslogd. Available from: http://netbsd-soc.sourceforge.net/projects/syslogd/.

[SK99] Bruce Schneier and John Kelsey. Secure audit logs to support computer forensics. *ACM Trans. on Information and System Security*, 2(2):159–176, 1999. Available from: http://www.schneier.com/paper-auditlogs.html.