FOSDEM 2020

Secure Logging with `syslog-ng`

Tamper evidence and confidentiality

Erik–Oliver Blass
Stephan Marwedel

01.02.2020
Security Devroom

This document describes the design, implementation, and configuration of a secure logging service with `syslog-ng`. Its aim is to provide tamper evident logging, i.e., to adequately protect log records of an information system against tampering and disclosure and to provide a sensor indicating attack attempts.

Contents

1. Design rationale

Log records are normally produced by any information system in order to perform monitoring during normal operations and for troubleshooting in case of technical problems. Log information is equally important for retaining the security of an information system, as security relevant events are recorded and can later be monitored for unusual patterns which may indicate an attack attempt. Examples include log on and log off, startup and shutdown, network service access, network filter rule application, storage access, etc. Log records may also contain valuable information about a system that a potential attacker intends to compromise. If an attacker is able to successfully compromise a system, they are also able to tamper with log records, potentially hiding their traces. This makes forensic analysis extremely difficult, as no reliable data source about system behavior immediately before the attack is available to a security analyst performing incident investigation. Therefore, log information should be appropriately protected.

The aim of the secure logging service is to provide tamper evident logging, i.e., to adequately protect log records of an information system and to provide a sensor indicating attack attempts. The secure logging service achieves this by authentically encrypting each log record with an individual cryptographic key used only once and protects integrity of the whole log archive by a cryptographic authentication code. Each attempt to tamper with either an individual log record or the log archive itself will be immediately detected during log archive verification. Therefore, an attacker can no longer tamper with log records without being detected.

2. Security Properties

We define forward integrity, i.e. tamper evidence and confidentiality, in two steps. First, we present a rather informal description capturing the security intuition in a real world environment. To avoid misunderstandings and to allow formal reasoning, we then proceed and give rigorous game-based definitions of forward integrity and confidentiality.

2.1. Intuitive description

Informally, both forward integrity (tamper evidence) and forward confidentiality only focus on securing log messages received by a logging device or logging service before the adversary has compromised the device. At the time of compromise, the adversary receives full access to the log device, learns all data and potential secrets stored on the device, can change or add data, and can even deviate from normal program execution. Without assuming existence of secure hardware or online connectivity, it is impossible to provide any security guarantees after the time of compromise.

Integrity The notion of forward integrity targets protection of log messages created before the time of compromise. Roughly speaking, if a log message is received by an non-compromised log device, its integrity will be protected in the future, even if the device will

get compromised at some point. Towards defining integrity in this context, we note that after the adversary has compromised the log device, they can modify any data stored on it. So, as we cannot prevent adversarial modifications to log data, integrity means making tampering with previous log data evident.

More specifically, if a verifier ("analyst") verifies a log file from a compromised log device, the goal is to detect whether log data from before the time of compromise has been tampered with. That is, if a total of n log messages have been stored before the time of compromise, the verifier should detect modifications to the first n log messages during log verification. This includes detecting whether or not the first n log messages are included in the log file.

Note that the verifier does not know the time of compromise. If a log is successfully verified, this does not allow the verifier to decide whether a system has been compromised or not, and which log messages are benign, unmodified log messages. Instead, successful verification only implies that all log messages written before the time of compromise have been preserved.

Confidentiality Similarly, we introduce the notion of forward confidentiality to protect sensitive information against prying adversaries. Again, we are only concerned about confidentiality of log messages from before the time of compromise.

A folklore description of confidentiality is that, by looking at ciphertexts, an adversary should not learn anything about underlying plaintexts. We borrow this intuition and summarize forward confidentiality for log messages as follows. Given the log file, the adversary should not learn anything about log messages written to the log file before the time of compromise. Towards a more formal definition, the goal is that, given the distribution of log messages, the adversary cannot tell anything about log messages before the time of compromise besides their distribution. The adversary does not learn anything new about messages. In practice, this means that the adversary cannot learn anything besides the number of messages or their length.

There is one additional caveat: as a secure logging protocol might make use of cryptographic primitives, e.g., encryption, we will have to take into account that these primitives can fail. A standard example of an encryption scheme failing is that the adversary by chance guesses the right encryption key. As this is unlikely, but still possible, our formal definition below will be relaxed and state that an adversary will not learn anything about log messages with high probability.

2.2. Formal Definitions

2.2.1. Preliminaries

In our scenario, log device $\mathcal{L}$ and verifier $\mathcal{V}$ follow a specific logging protocol Π. Log device $\mathcal{L}$ creates a log file $\mathcal{L}$ which is eventually verified by $\mathcal{V}$. More formally, a logging protocol Π comprises the following three algorithms.

1. $\text{Gen}(1^\lambda)$: this algorithm takes as input security parameter λ and outputs the initially empty log file L and system state σ_0. The idea is that σ_0 is the only information shared by the logging device and verifier $\mathcal{V}$.

2. $\text{Log}(\sigma_{i-1}, m_i)$: takes as an input the old system state σ_{i-1} and log message m and outputs an updated system state σ_i together with log *entry* s_i. Log entry s_i is appended to log file L.

   For performance reasons, Log must have constant $O(1)$ time complexity in the total number of log messages.

3. $\text{Verify}(L = (s'_1, \ldots, s'_j), \sigma'_j, \sigma_0)$: takes as input log file L, i.e., a sequence of log entries $(s'_1, \ldots, s'_j)$, state σ'_j, and the initial system state σ_0. This algorithm outputs either 1 (success) together with a sequence of log entries $(m'_1, \ldots, m'_u)$ or 0 (failure). We require correctness, i.e., if $j = n$ and $(s'_1 = \text{Log}(\sigma_0, m_1), \ldots, s'_n = \text{Log}(\sigma_{n-1}, m_n))$, then the output is 1 and $(m_1, \ldots, m_n)$.

   For high efficiency, Verify should be running in not more than $O(n)$ time complexity to verify a log.

The above setup captures prominent sequential logging services such as syslog(-ng) or rsyslog, where each new log entry is simply appended to the log. In practice, $\mathcal{L}$ executes Log, and $\mathcal{V}$ executes Verify.

2.2.2. Forward Integrity (Tamper Evidence)

We formalize security requirements for logging using a standard game-based definition. An adversary $\mathcal{A}$ compromising $\mathcal{L}$ has full access to the log and could, e.g., simply wipe the whole log. In the presence of such an adversary, it is therefore impossible to protect log entries against tampering. Our notion of integrity targets detection of tampering, i.e, *tamper evidence*.

Experiment $\text{Exp}^{\text{FInt}}_{\mathcal{A},\Pi}(\lambda)$ **1:** Forward Integrity

```
1  (σ_0, L) ← Gen(1^λ);
2  st_A ← A(1^λ);
3  for i = 1 to n = poly(λ) do
4      (m_i, st_A) ← A(st_A);
5      (σ_i, s_i) ← Log(σ_{i-1}, m_i);
6      L = L||s_i;
7  end
8  (L' = (s'_1, ..., s'_j), σ') ← A(st_A, σ_n, L);
9  (b, m'_1, ..., m'_u) = Verify(L', σ', σ_0);
10 if [b = 1] ∧ [u < n ∨ ∃i ∈ [1, ..., n] : m_i ≠ m'_i] then
11     output 1
12 else
13     output 0
```

Experiment $\text{Exp}^{\text{FInt}}_{\mathcal{A},\Pi}$ shows a forward integrity game. A protocol $\Pi = (\text{Gen}, \text{Log}, \text{Verify})$ makes use of an internal, secret state σ_i which we allow to evolve over time. Initial state σ_0 serves as shared secret between $\mathcal{L}$ and $\mathcal{V}$. Let $\text{st}_\mathcal{A}$ be the adversary's state.

In $\text{Exp}^{\text{FInt}}_{\mathcal{A},\Pi}$, adversary $\mathcal{A}$ specifies log entries m_i. Each log entry m_i is fed into Log outputting s_i, which is written by $\mathcal{L}$ to log file L. Allowing $\mathcal{A}$ to choose log messages will later imply that a protocol Π is secure regardless of the distribution of log messages. Note that $\mathcal{A}$ does not choose the m_i adaptively as in a chosen plaintext attack.

At the time of compromise n, adversary $\mathcal{A}$ also learns the internal state σ_n of $\mathcal{L}$ and file L. Note that $\mathcal{A}$ can execute Log themselves on arbitrary log entries of their choice, learn the outcome, evolve the state etc. Eventually, $\mathcal{A}$ outputs a log file, containing a sequence of strings s'_i and some system state σ'.

The verification function Verify decides validity of the log file by parsing the sequence of strings s'_i in file L', system state σ', and initial state σ_0. In the real world, Verify will be executed by verifier $\mathcal{V}$ who only shares the initial state σ_0 with the log device.

Definition 1 (Forward Integrity)**.** *A logging protocol* $\Pi = (\text{Gen}, \text{Log}, \text{Verify})$ *provides* forward integrity*,* iff *for all probabilistic polynomial-time (PPT) adversaries* $\mathcal{A}$ *there exists a negligible function* ϵ *such that*

$$Pr[\text{Exp}^{\text{FInt}}_{\mathcal{A},\Pi}(\lambda) = 1] = \epsilon(\lambda).$$

2.2.3. Forward Confidentiality

Along the same lines, we formalize forward confidentiality in Experiment $\text{Exp}^{\text{FConf}}_{\mathcal{A},\Pi}(\lambda)$. Here, adversary $\mathcal{A}$ chooses n pairs of two messages $(m_{i,0}, m_{i,1})$. For an initially chosen bit b, Log creates new log entries s_i on message $m_{i,b}$, and s_i is added to log file L.

Eventually, $\mathcal{A}$ compromises the log device and is given L and current state σ_n, and $\mathcal{A}$ guesses b by outputting a bit b'. If $b' = b$, then $\mathcal{A}$ has guessed b correctly, and the outcome of Experiment $\text{Exp}^{\text{FConf}}_{\mathcal{A},\Pi}(\lambda)$ is 1, otherwise it is 0.

Definition 2 (Forward Confidentiality)**.** *A logging protocol* $\Pi = (\text{Gen}, \text{Log}, \text{Verify})$ *provides* forward confidentiality*,* iff *for all probabilistic polynomial-time (PPT) adversaries* $\mathcal{A}$ *there exists a negligible function* ϵ *such that*

$$Pr[\text{Exp}^{\text{FConf}}_{\mathcal{A},\Pi}(\lambda) = 1] = \epsilon(\lambda).$$

Note that the above definition of forward confidentiality resembles the notion of secure encryption of multiple messages against an eavesdropping adversary (LR-encryption). Specifically in our scenario, $\mathcal{A}$ does not choose messages adaptively as in a chosen plaintext attack. As with any practical definition of confidentiality, lengths of each pair of two messages must be the same, i.e., $|m_{i,0}| = |m_{i,1}|$. In the real world, this reflects the problem that an adversary learns information about a ciphertext just by observing its length.

Experiment $\mathrm{Exp}^{\mathrm{FConf}}_{\mathcal{A},\Pi}(\lambda)$ **2:** Forward Confidentiality

$(\sigma_0, L) \leftarrow \mathrm{Gen}(1^\lambda)$;
$\mathrm{st}_\mathcal{A} \leftarrow \mathcal{A}(1^\lambda)$;
$b \xleftarrow{\$} \{0,1\}$;
for $i = 1$ ***to*** $n = \mathrm{poly}(\lambda)$ **do**
    $(m_{i,0}, m_{i,1}, \mathrm{st}_\mathcal{A}) \leftarrow \mathcal{A}(\mathrm{st}_\mathcal{A})$; // $|m_{i,0}| = |m_{i,1}|$
    $(\sigma_i, s_i) \leftarrow \mathrm{Log}(\sigma_{i-1}, m_{i,b})$;
    $L = L||s_i$;
end
$b' \leftarrow \mathcal{A}(\mathrm{st}_\mathcal{A}, \sigma_n, L)$;
if $b' = b$ **then**
    output 1;
else
    output 0;

As one might expect from forward confidentiality, our definition does not relate to confidentiality of log messages after the time of compromise.

A more detailed description of the threat model and the security properties of the logging protocol used as base of the implementation can be found in [1].

3. Software architecture

Most information systems rely on standards in order to provide logging services. One of the most widely adopted standards is the `syslog` protocol which is specified in RFC 5424. Many implementations of this protocol are available. A popular extensible implementation with additional features is `syslog-ng`, which is an enhanced logging daemon with advanced features for input and output. Furthermore, it features capabilities for log message filtering, rewriting, and routing. It can be used as a drop-in replacement for existing log daemons on UNIX systems.

`syslog-ng` is available for most of the systems currently in use. It contains a plugin mechanism which allows for extending its functionality by supplying additional functionality in form of a suitable plugin. `syslog-ng` is open source software (OSS), thus facilitating the implementation of additional functionality. Therefore, the tamper evident logging service is implemented as a template function within `syslog-ng`.

In order to understand the integration of the secure logging service into `syslog-ng`, terms in table 1 were defined.

Term	Description
Source	Any source of log data, e.g., a file created by a system service, a User Datagram Protocol (UDP) or Transmission Control Protocol (TCP) port etc.
Destination	A receiver of log information, e.g., a file, a network port, etc.
Template	A template is a used to format a log message in a particular way. For example, a template may determine how the date is formatted and in what format the message is sent to a destination
Filter	A filter is a mechanism that decides to which destination a specific log message is dispatched. Internal routing of log messages is performed with filters.
Junction	Junctions make it possible to send the messages to different channels, process the messages differently on each channel, and then join every channel together again. This way, a message can be split up into two different copies that are process differently and joined together afterwards.
Channel	A channel defines a processing entity of a log message. Processing can be done using either filters or templates.

Table 1: `syslog-ng` terminology

Figure 1 shows the principal architecture of `syslog-ng`. Log information is received from a source.

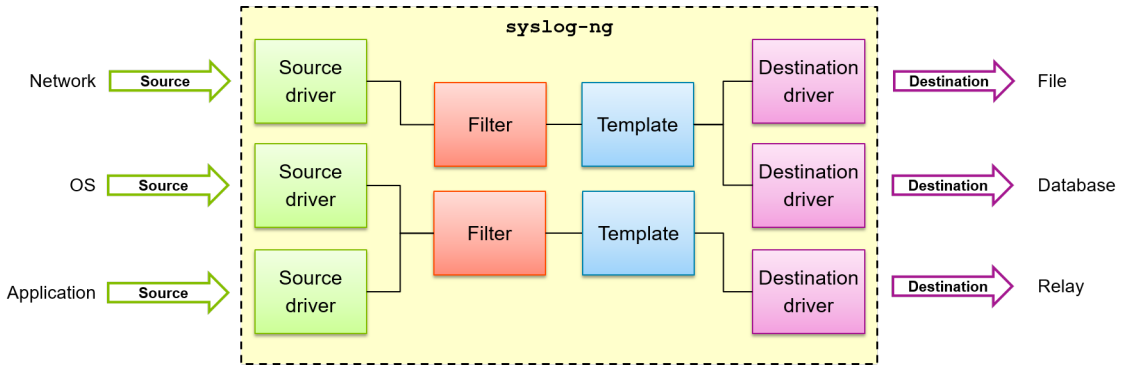

Figure 1: General architecture of `syslog-ng`

A source can be any data source for which a source driver exists. A source driver converts the data being received into the internal format of `syslog-ng`. Once the data is available in the internal format, it can be filtered by a filter, rewritten by a template or routed to a channel. Eventually, the data is written to a destination. Similar to a source, a destination is provided by a destination driver which forwards the data accordingly.

For example, a destination can be a simple file on disk, a network port or a database. These elements can be flexibly combined to allow for adaption of the logging system to specific use case scenarios. As an example, consider a log host that receives log information, processes it with the help of filters and template and forwards the formatted data to another log

host. This log host acts as a log relay, as it does not feature a persistent destination, like a file or database. Such a configuration would allow for the setup of a set of log relays that will all forward their data to a central log collector that features a persistent destination.

`syslog-ng` is configured by means of a configuration file. This file has separate sections for sources, filters, templates, channels and destination. Each of the entities configured can be combined to define a particular log entry. The `syslog-ng` system features an efficient implementation based on internal reference counting that prevents a message from being copied if is to be filtered by different filters or written to different destinations.

The secure logging service uses the template mechanism of `syslog-ng` for implementation, see Figure 2.

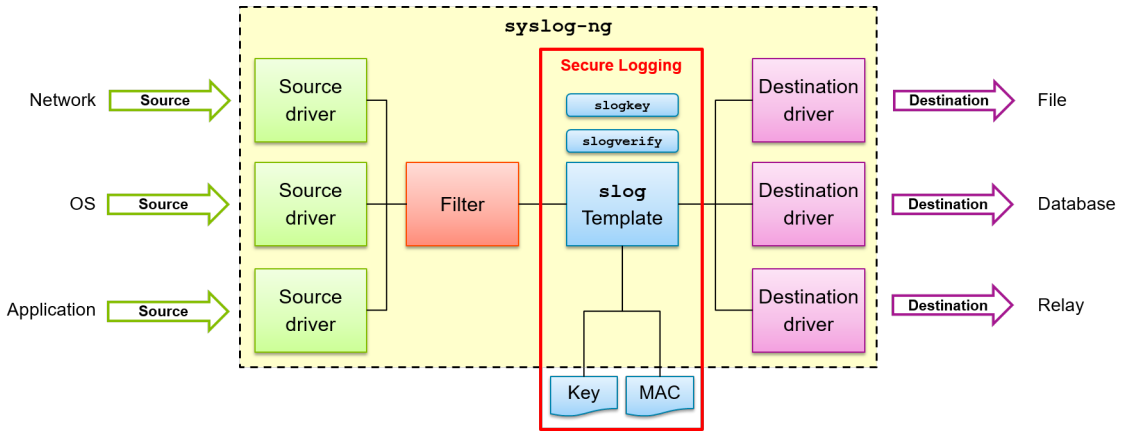

Figure 2: Integration of secure logging template into `syslog-ng`

A log message from any source is passed to the secure logging template. This template takes the raw, i.e. unformatted message as an input and rewrites as an encrypted message with integrity protection. In this form, the message is then passed to the next processing step. This can be either a direct output to a destination or to a channel for routing purposes. Each log message that is passed to the template will be encrypted with its own key. The location of the key file is specified in the template configuration. Additionally, a message authentication code (MAC) for the integrity protection of the complete encrypted log archive is stored in a file. The location of that file is also part of the secure logging configuration.

The following extract from a log archive created with the secure logging service shows how log entries are stored. Listing 3 shows the original log message received at the source.

```
1 Dies ist eine Log Nachricht
2 Und dies auch
3 Hier kommt mal eine laengere Nachricht
```

Listing 3: Original log messsages

Listing 4 shows the log messages after they have been passed through the secure logging template and have been written to the destination. The entries in listing 4 have been

truncated due to space restrictions. The truncation is indicated by the trailing dots which are not part of the actual log record.

```
1 AAAAAAAAAAA=:wp0KVPUTxHGEXMd1jxPPVKqK+aztmaN4tEpqNCstIUuFBBi...
2 AQAAAAAAAAA=:jwYQbBqTun7jcEQ6ns5UIwePwQXI3HKLpPROAZOYtTeNB1i...
3 AgAAAAAAAAA=:XdCte4wSegtu/EnRa7JheQ3AMY9pKT1Gbio/c03RkT3dyuD...
```

Listing 4: Encrypted log archive

Log entries are numbered sequentially. The first field of a log entry contains this number in hexadecimal form followed by a colon. The rest of the line contains the actual log record. After successful verification the log archive would look like depicted in listing 5.

```
1 0000000000000000: Dies ist eine Log Nachricht
2 0000000000000001: Und dies auch
3 0000000000000002: Hier kommt mal eine laengere Nachricht
```

Listing 5: Log archive from listing 4 after successful verification

4. Configuration

The secure logging service uses the standard configuration mechamism of `syslog-ng`. It adds the following template statement to the configuration:

```
1 $(slog -k /var/slog/host.key -m /var/slog/mac.dat $RAWMSG)
```

Listing 6: Secure logging template configuration

In this statement, `slog` is the name of the template function. This name is part of the template implementation and cannot be changed. The parameter `-k` indicates the location of the key file on the host where the instance is running. Parameter `-m` indicates the location of the file containing the message authentication code. `$RAWMSG` represents the original log message without any formatting.

Listing 7 shows an example of a valid configuration file with secure logging functionality enabled.

```
1  source s_network {
2    network(
3      transport("udp")
4      port(514)
5
6      # NOTE: Secure logging requires this flag to be set
7      flags(store-raw-message)
8    );
9  };
10
11 # Secure logging template with key and MAC file locations
12 template t_slog {
13    template("$(slog -k /var/slog/host.key -m /var/slog/mac.dat ↵
          $RAWMSG)\n");
14 };
15
16 # Destination that uses the TELS template
17 destination d_local {
18   file("/var/log/messages.slog" template(t_slog));
19 };
20
21 log {
22   source(s_network);
23   destination(d_local);
24 };
```

Listing 7: `syslog-ng` configuration example

Lines 1–9 define a source in form of a UDP network socket listening on port 514. This is the standard syslog service port using the UDP protocol. Line 7 activates the option for storing the original message without any formatting in addition to the parsed format. This is required to be specified for every source which is in conjunction with the secure logging template. Otherwise, the `$RAWMSG` macro will be empty. Line 13 defines the secure logging template with parameters as defined above. Pay attention to the line feed at the end of the template configuration. If this is missing, the log archive comprises only one single line. If a line feed is specified as in the example, each log record will be written on exactly one line in the log archive. In contrast to the key file, the Message Authentication Code (MAC) is automatically generated by the secure logging service and does not need to be supplied on startup.

5. Key generation

The secure logging service needs to be supplied with an initial cryptographic key K_0. This key will only be used to bootstrap the service, i.e. to encrypt the very first log entry and will be overwritten afterwards. Each log record will be encrypted by its own individual cryptographic key. These keys are generated based on on the previous key by key evolvement function which is part of the core secure logging functionality. The secure logging service features the `slogkey` utility for the generation of cryptographic keys for the service. Key generation is done in two steps. A master key is generated first. Starting from this master key, a host key is generated for each host for which the secure logging service shall be enabled.

5.1. Master key

The master key K_m is generated by supplying the `-m` option to the `slogkey` utility like in the following example:

```
slogkey -m /etc/syslog-ng/master.key
```

The master key K_m is required for subsequent log verification and must should never be stored on the log host.

5.2. Host key

Host key K_0 is generated from master key K_m by supplying the `-d` option to the `slogkey` utility with appropriate arguments like in the following example:

```
slogkey -d /etc/syslog-ng/master.key a08cefa7b520 CAC7119N43 ⤸
    /var/slog/host.key
```

The option arguments designates the file location of the master key K_m which is used for derivation of the host key. The other arguments are the hardware ethernet address of the host system, the serial number of the host hardware and the file receiving the host key K_0, respectively. Instead of the hardware ethernet address or the serial number any other pair of strings that uniquely identifies a particular host can be used for the host key creation. This initial key K_0 is required for subsequent log verification. It can always be generated from the master key K_m. See the manual page supplied with the `slogkey` utility for more information on how to use this utility.

6. Log verification

Log archive verification is performed with the utility `slogverify` which is also part of the secure logging service. For successful log archive verification the host key K_0, the file containing the message authentication code for the log archive and the log archive to be verified must be passed to the utility. Additionally, a buffer size can be passed as optional argument, in case very large log archives. These can only be verified in chunks and the buffer must be sized appropriately in order not to overload the system. If the buffer size is not passed in the default buffer size of 1000 log entries is used. A typical command line for log verification would look like this:

```
slogverify /var/slog/host.key /var/slog/messages.slog ⤸
    /var/slog/mac.dat /var/slog/messages.log 1500
```

See the manual page supplied with the `slogverify` utility for more information on how to use this utility.

7. Source code layout

The listing below shows the location of code for the secure logging service within the `syslog-ng` source tree. The unit tests of `syslog-ng` need as an additional dependency the unit testing framework `Criterion`.

```
<syslog-ng source root>
├── doc
│   └── man ..... Manual pages for secure logging command line utilities
├── lib
│   ├── slog.h
│   └── slog.c ..... Secure logging core functionality
└── modules
    └── cryptofuncs
        ├── cryptofuncs.c ..... Secure logging template in tf_slog_prepare and
        │   tf_slog_call
        ├── slogimport
        │   └── slogimport.c ..... Command line tool for log import
        ├── slogkey
        │   └── slogkey.c ..... Command line tool for initial key generation
        ├── slogverify
        │   └── slogverify.c ..... Command line tool for log verification
        └── tests
            └── test_cryptofuncs.c ..... Secure logging unit tests in
                test_slog_functionality
```

Listing 8: Secure logging source code integration into `syslog-ng`

The Criterion framework is not part of the Extra Packages for Enterprise Linux (EPEL) and must be installed separately in case unit testing is to be performed. It is available at

`https://github.com/Snaipe/Criterion`

The source package needs to be configured accordingly if the unit tests are to be executed. The unit tests for the secure logging service are implemented in file `testcryptofuncs.c`.

References

[1] Erik-Oliver Blass and Guevara Noubir. "Secure Logging with Crash Tolerance". In: *IEEE Conference on Communications and Network Security* (2017). URL: `https://eprint.iacr.org/2017/107.pdf`.

Abbreviations

Notation	Description
EPEL	Extra Packages for Enterprise Linux
EVP	Envelope
GCM	Galois Counter Mode
MAC	Message Authentication Code
TCP	Transmission Control Protocol
UDP	User Datagram Protocol

Table 2: Abbreviations

Glossary

Integrity tag An integrity tag is an additional piece of information that is added to a message to confirm its integrity. A message receiver can then verify whether the received message has been tampered with during transit. Integrity tags are normally generated by message authentication codes.

Message authentication code A message authentication code (MAC) is a cryptographic technique to protect data integrity. When supplied with a message m and a cryptographic key k, a MAC produces an integrity tag t. A sender sends m together with t to a receiver. Knowing key k, the receiver verifies that message m has not been tampered with during transit by computing a MAC of the message received and comparing the resulting integrity tag with t.

A. OpenSSL Documentation

The OpenSSL documentation[1] on how to use the Envelope (EVP) interface and the Galois Counter Mode (GCM) is included here as a reference. An up to date version of this documentation can be found at

`https://wiki.openssl.org/index.php/EVP_Authenticated_Encryption_and_Decryption`

EVP Authenticated Encryption and Decryption

The EVP interface supports the ability to perform authenticated encryption and decryption, as well as the option to attach unencrypted, associated data to the message. Such Authenticated-Encryption with Associated-Data (AEAD) schemes provide confidentiality by encrypting the data, and also provide authenticity assurances by creating a MAC tag over the encrypted data. The MAC tag will ensure the data is not accidentally altered or maliciously tampered during transmission and storage.

There are a number of AEAD modes of operation. The modes include EAX, CCM and GCM mode. Using AEAD modes is nearly identical to using standard symmetric encryption modes like CBC, CFB and OFB modes.

[1] Version from 8 October 2019

As with standard symmetric encryption you will need to know the following:

- Algorithm (currently only AES is supported)
- Mode (currently only GCM and CCM are supported)
- Key
- Initialisation Vector (IV)

In addition you can (optionally) provide some *Additional Authenticated Data* (AAD). The AAD data is not encrypted, and is typically passed to the recipient in plaintext along with the ciphertext. An example of AAD is the IP address and port number in a IP header used with IPsec.

The output from the encryption operation will be the ciphertext, and a tag. The tag is subsequently used during the decryption operation to ensure that the ciphertext and AAD have not been tampered with.

The OpenSSL manual describes the usage of the GCM and CCM modes here: Manual:EVP_EncryptInit(3)#GCM_Mode.

The complete source code of the following examples can be downloaded as evp-gcm-encrypt.c resp. evp-ccm-encrypt.c.

Contents

Authenticated Encryption using GCM mode

Encryption is performed in much the same way as for symmetric encryption as described here. The main differences are:

- You may optionally pass through an IV length using EVP_CIPHER_CTX_ctrl
- AAD data is passed through in zero or more calls to EVP_EncryptUpdate, with the output buffer set to NULL
- Once private data has been added using EVP_EncryptUpdate (non-NULL output buffer), you cannot add AAD data
- After the EVP_EncryptFinal_ex call a new call to EVP_CIPHER_CTX_ctrl retrieves the tag

See the code below for an example:

```
int gcm_encrypt(unsigned char *plaintext, int plaintext_len,
                unsigned char *aad, int aad_len,
                unsigned char *key,
                unsigned char *iv, int iv_len,
                unsigned char *ciphertext,
                unsigned char *tag)
{
    EVP_CIPHER_CTX *ctx;

    int len;
```

```
    int ciphertext_len;


    /* Create and initialise the context */
    if(!(ctx = EVP_CIPHER_CTX_new()))
        handleErrors();

    /* Initialise the encryption operation. */
    if(1~!= EVP_EncryptInit_ex(ctx, EVP_aes_256_gcm(), NULL, NULL, NULL))
        handleErrors();

    /*
     * Set IV length if default 12 bytes (96 bits) is not appropriate
     */
    if(1~!= EVP_CIPHER_CTX_ctrl(ctx, EVP_CTRL_GCM_SET_IVLEN, iv_len, NULL))
        handleErrors();

    /* Initialise key and IV */
    if(1~!= EVP_EncryptInit_ex(ctx, NULL, NULL, key, iv))
        handleErrors();

    /*
     * Provide any AAD data. This can be called zero or more times as
     * required
     */
    if(1~!= EVP_EncryptUpdate(ctx, NULL, &len, aad, aad_len))
        handleErrors();

    /*
     * Provide the message to be encrypted, and obtain the encrypted output.
     * EVP_EncryptUpdate can be called multiple times if necessary
     */
    if(1~!= EVP_EncryptUpdate(ctx, ciphertext, &len, plaintext, plaintext_len))
        handleErrors();
    ciphertext_len = len;

    /*
     * Finalise the encryption. Normally ciphertext bytes may be written at
     * this stage, but this does not occur in GCM mode
     */
    if(1~!= EVP_EncryptFinal_ex(ctx, ciphertext + len, &len))
        handleErrors();
    ciphertext_len += len;

    /* Get the tag */
    if(1~!= EVP_CIPHER_CTX_ctrl(ctx, EVP_CTRL_GCM_GET_TAG, 16, tag))
        handleErrors();

    /* Clean up */
    EVP_CIPHER_CTX_free(ctx);

    return ciphertext_len;
}
```

Authenticated Decryption using GCM mode

Again, the decryption operation is much the same as for normal symmetric decryption as described here. The main differences are:

- You may optionally pass through an IV length using EVP_CIPHER_CTX_ctrl
- AAD data is passed through in zero or more calls to EVP_DecryptUpdate, with the output buffer set to NULL
- Prior to the EVP_DecryptFinal_ex call a new call to EVP_CIPHER_CTX_ctrl provides the tag
- A non positive return value from EVP_DecryptFinal_ex should be considered as a failure to authenticate ciphertext and/or AAD. It does not necessarily indicate a more serious error.

See the code example below:

```
int gcm_decrypt(unsigned char *ciphertext, int ciphertext_len,
                unsigned char *aad, int aad_len,
                unsigned char *tag,
                unsigned char *key,
                unsigned char *iv, int iv_len,
                unsigned char *plaintext)
{
    EVP_CIPHER_CTX *ctx;
    int len;
    int plaintext_len;
    int ret;

    /* Create and initialise the context */
    if(!(ctx = EVP_CIPHER_CTX_new()))
        handleErrors();

    /* Initialise the decryption operation. */
    if(!EVP_DecryptInit_ex(ctx, EVP_aes_256_gcm(), NULL, NULL, NULL))
        handleErrors();

    /* Set IV length. Not necessary if this is 12 bytes (96 bits) */
    if(!EVP_CIPHER_CTX_ctrl(ctx, EVP_CTRL_GCM_SET_IVLEN, iv_len, NULL))
        handleErrors();

    /* Initialise key and IV */
    if(!EVP_DecryptInit_ex(ctx, NULL, NULL, key, iv))
        handleErrors();

    /*
     * Provide any AAD data. This can be called zero or more times as
     * required
     */
    if(!EVP_DecryptUpdate(ctx, NULL, &len, aad, aad_len))
        handleErrors();

    /*
     * Provide the message to be decrypted, and obtain the plaintext output.
     * EVP_DecryptUpdate can be called multiple times if necessary
     */
    if(!EVP_DecryptUpdate(ctx, plaintext, &len, ciphertext, ciphertext_len))
        handleErrors();
    plaintext_len = len;
```

```
    /* Set expected tag value. Works in OpenSSL 1.0.1d and later */
    if(!EVP_CIPHER_CTX_ctrl(ctx, EVP_CTRL_GCM_SET_TAG, 16, tag))
        handleErrors();

    /*
     * Finalise the decryption. A positive return value indicates success,
     * anything else is a failure - the plaintext is not trustworthy.
     */
    ret = EVP_DecryptFinal_ex(ctx, plaintext + len, &len);

    /* Clean up */
    EVP_CIPHER_CTX_free(ctx);

    if(ret > 0) {
        /* Success */
        plaintext_len += len;
        return plaintext_len;
    } else {
        /* Verify failed */
        return -1;
    }
}
```

Authenticated Encryption using CCM mode

Encryption with CCM mode is much the same as for encryption with GCM but with some additional things to bear in mind.

- you can only call EVP_EncryptUpdate once for AAD and once for the plaintext.
- The total plaintext length must be passed to EVP_EncryptUpdate (only needed if AAD is passed)
- Optionally the tag and IV length can also be passed. If they are not then the defaults are used (12 bytes for AES tags, and 7 bytes for AES IVs)

See the code below for an example:

```
int ccm_encrypt(unsigned char *plaintext, int plaintext_len,
                unsigned char *aad, int aad_len,
                unsigned char *key,
                unsigned char *iv,
                unsigned char *ciphertext,
                unsigned char *tag)
{
    EVP_CIPHER_CTX *ctx;

    int len;

    int ciphertext_len;


    /* Create and initialise the context */
    if(!(ctx = EVP_CIPHER_CTX_new()))
        handleErrors();

    /* Initialise the encryption operation. */
    if(1~!= EVP_EncryptInit_ex(ctx, EVP_aes_256_ccm(), NULL, NULL, NULL))
```

```
        handleErrors();

    /*
     * Setting IV len to 7. Not strictly necessary as this is the default
     * but shown here for the purposes of this example.
     */
    if(1~!= EVP_CIPHER_CTX_ctrl(ctx, EVP_CTRL_CCM_SET_IVLEN, 7, NULL))
        handleErrors();

    /* Set tag length */
    EVP_CIPHER_CTX_ctrl(ctx, EVP_CTRL_CCM_SET_TAG, 14, NULL);

    /* Initialise key and IV */
    if(1~!= EVP_EncryptInit_ex(ctx, NULL, NULL, key, iv))
        handleErrors();

    /* Provide the total plaintext length */
    if(1~!= EVP_EncryptUpdate(ctx, NULL, &len, NULL, plaintext_len))
        handleErrors();

    /* Provide any AAD data. This can be called zero or one times as required */
    if(1~!= EVP_EncryptUpdate(ctx, NULL, &len, aad, aad_len))
        handleErrors();

    /*
     * Provide the message to be encrypted, and obtain the encrypted output.
     * EVP_EncryptUpdate can only be called once for this.
     */
    if(1~!= EVP_EncryptUpdate(ctx, ciphertext, &len, plaintext, plaintext_len))
        handleErrors();
    ciphertext_len = len;

    /*
     * Finalise the encryption. Normally ciphertext bytes may be written at
     * this stage, but this does not occur in CCM mode.
     */
    if(1~!= EVP_EncryptFinal_ex(ctx, ciphertext + len, &len))
        handleErrors();
    ciphertext_len += len;

    /* Get the tag */
    if(1~!= EVP_CIPHER_CTX_ctrl(ctx, EVP_CTRL_CCM_GET_TAG, 14, tag))
        handleErrors();

    /* Clean up */
    EVP_CIPHER_CTX_free(ctx);

    return ciphertext_len;
}
```

Authenticated Decryption using CCM mode

Decryption with CCM mode is much the same as for decryption with CCM but with some additional things to bear in mind.

- you can only call EVP_DecryptUpdate once for AAD and once for the plaintext.
- The total ciphertext length must be passed to EVP_DecryptUpdate (only needed if AAD is

passed)

- Optionally the tag and IV length can also be passed. If they are not then the defaults are used (12 bytes for AES tags, and 7 bytes for AES IVs)
- The tag verify is performed when you call the final EVP_DecryptUpdate and is reflected by the return value: there is no call to EVP_DecryptFinal.

See the code below for an example:

```
int ccm_decrypt(unsigned char *ciphertext, int ciphertext_len,
                unsigned char *aad, int aad_len,
                unsigned char *tag,
                unsigned char *key,
                unsigned char *iv,
                unsigned char *plaintext)
{
    EVP_CIPHER_CTX *ctx;
    int len;
    int plaintext_len;
    int ret;

    /* Create and initialise the context */
    if(!(ctx = EVP_CIPHER_CTX_new()))
        handleErrors();

    /* Initialise the decryption operation. */
    if(1~!= EVP_DecryptInit_ex(ctx, EVP_aes_256_ccm(), NULL, NULL, NULL))
        handleErrors();

    /* Setting IV len to 7. Not strictly necessary as this is the default
     * but shown here for the purposes of this example */
    if(1~!= EVP_CIPHER_CTX_ctrl(ctx, EVP_CTRL_CCM_SET_IVLEN, 7, NULL))
        handleErrors();

    /* Set expected tag value. */
    if(1~!= EVP_CIPHER_CTX_ctrl(ctx, EVP_CTRL_CCM_SET_TAG, 14, tag))
        handleErrors();

    /* Initialise key and IV */
    if(1~!= EVP_DecryptInit_ex(ctx, NULL, NULL, key, iv))
        handleErrors();


    /* Provide the total ciphertext length */
    if(1~!= EVP_DecryptUpdate(ctx, NULL, &len, NULL, ciphertext_len))
        handleErrors();

    /* Provide any AAD data. This can be called zero or more times as required */
    if(1~!= EVP_DecryptUpdate(ctx, NULL, &len, aad, aad_len))
        handleErrors();

    /*
     * Provide the message to be decrypted, and obtain the plaintext output.
     * EVP_DecryptUpdate can be called multiple times if necessary
     */
    ret = EVP_DecryptUpdate(ctx, plaintext, &len, ciphertext, ciphertext_len);

    plaintext_len = len;
```

```
    /* Clean up */
    EVP_CIPHER_CTX_free(ctx);

    if(ret > 0) {
        /* Success */
        return plaintext_len;
    } else {
        /* Verify failed */
        return -1;
    }
}
```

Potential Issue in AES/GCM

Early versions of the authenticated encryption interface required using a 0-sized array (not a NULL array) to arrive at the proper authentication tag ***when*** the authentication tag size was *not* a multiple of the block size (for example, an authentication tag size of 20 bytes). For more information on the issue and the work-arounds, see Issue #2859: Possible bug in AES GCM mode and Possible bug in GCM/GMAC with (just) AAD of size unequal to block size.