

# User Space Interface

## Introduction

The concepts of the kernel crypto API visible to kernel space is fully applicable to the user space interface as well. Therefore, the kernel crypto API high level discussion for the in-kernel use cases applies here as well.

The major difference, however, is that user space can only act as a consumer and never as a provider of a transformation or cipher algorithm.

The following covers the user space interface exported by the kernel crypto API. A working example of this description is libkcapi that can be obtained from [1]. That library can be used by user space applications that require cryptographic services from the kernel.

Some details of the in-kernel kernel crypto API aspects do not apply to user space, however. This includes the difference between synchronous and asynchronous invocations. The user space API call is fully synchronous.

[1] <https://www.chronox.de/libkcapi.html>

## User Space API General Remarks

The kernel crypto API is accessible from user space. Currently, the following ciphers are accessible:

- Message digest including keyed message digest (HMAC, CMAC)
- Symmetric ciphers
- AEAD ciphers
- Random Number Generators

The interface is provided via socket type using the type AF\_ALG. In addition, the setsockopt option type is SOL\_ALG. In case the user space header files do not export these flags yet, use the following macros:

```
#ifndef AF_ALG
#define AF_ALG 38
#endif
#ifndef SOL_ALG
#define SOL_ALG 279
#endif
```

A cipher is accessed with the same name as done for the in-kernel API calls. This includes the generic vs. unique naming schema for ciphers as well as the enforcement of priorities for generic names.

To interact with the kernel crypto API, a socket must be created by the user space application. User space invokes the cipher operation with the send()/write() system call family. The result of the cipher operation is obtained with the read()/recv() system call family.

The following API calls assume that the socket descriptor is already opened by the user space application and discusses only the kernel crypto API specific invocations.

To initialize the socket interface, the following sequence has to be performed by the consumer:

1. Create a socket of type AF\_ALG with the struct sockaddr\_alg parameter specified below for the different cipher types.
2. Invoke bind with the socket descriptor
3. Invoke accept with the socket descriptor. The accept system call returns a new file descriptor that is to be used to interact with the particular cipher instance. When invoking send/write or recv/read system calls to send data to the kernel or obtain data from the kernel, the file descriptor returned by accept must be used.

## In-place Cipher operation

Just like the in-kernel operation of the kernel crypto API, the user space interface allows the cipher operation in-place. That means that the input buffer used for the send/write system call and the output buffer used by the read/recv system call may be one and the same. This is of particular interest for symmetric cipher operations where a copying of the output data to its final destination can be avoided.

If a consumer on the other hand wants to maintain the plaintext and the ciphertext in different memory locations, all a consumer needs to do is to provide different memory pointers for the encryption and decryption operation.

## Message Digest API

The message digest type to be used for the cipher operation is selected when invoking the bind syscall. bind requires the caller to provide a filled struct sockaddr data structure. This data structure must be filled as follows:

```

struct sockaddr_alg sa = {
    .salg_family = AF_ALG,
    .salg_type = "hash", /* this selects the hash logic in the kernel */
    .salg_name = "sha1" /* this is the cipher name */
};

```

The `salg_type` value "hash" applies to message digests and keyed message digests. Though, a keyed message digest is referenced by the appropriate `salg_name`. Please see below for the `setsockopt` interface that explains how the key can be set for a keyed message digest.

Using the `send()` system call, the application provides the data that should be processed with the message digest. The `send` system call allows the following flags to be specified:

- **MSG\_MORE**: If this flag is set, the `send` system call acts like a message digest update function where the final hash is not yet calculated. If the flag is not set, the `send` system call calculates the final message digest immediately.

With the `recv()` system call, the application can read the message digest from the kernel crypto API. If the buffer is too small for the message digest, the flag `MSG_TRUNC` is set by the kernel.

In order to set a message digest key, the calling application must use the `setsockopt()` option of `ALG_SET_KEY`. If the key is not set the HMAC operation is performed without the initial HMAC state change caused by the key.

## Symmetric Cipher API

The operation is very similar to the message digest discussion. During initialization, the `struct sockaddr` data structure must be filled as follows:

```

struct sockaddr_alg sa = {
    .salg_family = AF_ALG,
    .salg_type = "skcipher", /* this selects the symmetric cipher */
    .salg_name = "cbc(aes)" /* this is the cipher name */
};

```

Before data can be sent to the kernel using the `write/send` system call family, the consumer must set the key. The key setting is described with the `setsockopt` invocation below.

Using the `sendmsg()` system call, the application provides the data that should be processed for encryption or decryption. In addition, the IV is specified with the data structure provided by the `sendmsg()` system call.

The `sendmsg` system call parameter of `struct msghdr` is embedded into the `struct cmsghdr` data structure. See `recv(2)` and `cmsghdr(3)` for more information on how the `cmsghdr` data structure is used together with the `send/recv` system call family. That `cmsghdr` data structure holds the following information specified with a separate header instances:

- specification of the cipher operation type with one of these flags:
  - `ALG_OP_ENCRYPT` - encryption of data
  - `ALG_OP_DECRYPT` - decryption of data
- specification of the IV information marked with the flag `ALG_SET_IV`

The `send` system call family allows the following flag to be specified:

- **MSG\_MORE**: If this flag is set, the `send` system call acts like a cipher update function where more input data is expected with a subsequent invocation of the `send` system call.

Note: The kernel reports `-EINVAL` for any unexpected data. The caller must make sure that all data matches the constraints given in `/proc/crypto` for the selected cipher.

With the `recv()` system call, the application can read the result of the cipher operation from the kernel crypto API. The output buffer must be at least as large as to hold all blocks of the encrypted or decrypted data. If the output data size is smaller, only as many blocks are returned that fit into that output buffer size.

## AEAD Cipher API

The operation is very similar to the symmetric cipher discussion. During initialization, the `struct sockaddr` data structure must be filled as follows:

```

struct sockaddr_alg sa = {
    .salg_family = AF_ALG,
    .salg_type = "aead", /* this selects the symmetric cipher */
    .salg_name = "gcm(aes)" /* this is the cipher name */
};

```

Before data can be sent to the kernel using the `write/send` system call family, the consumer must set the key. The key setting is described with the `setsockopt` invocation below.

In addition, before data can be sent to the kernel using the `write/send` system call family, the consumer must set the authentication tag size. To set the authentication tag size, the caller must use the `setsockopt` invocation described below.

Using the `sendmsg()` system call, the application provides the data that should be processed for encryption or decryption. In addition, the IV is specified with the data structure provided by the `sendmsg()` system call.

The `sendmsg` system call parameter of `struct msghdr` is embedded into the `struct cmsghdr` data structure. See `recv(2)` and `cmsg(3)` for more information on how the `cmsg` data structure is used together with the `send/recv` system call family. That `cmsg` data structure holds the following information specified with a separate header instances:

- specification of the cipher operation type with one of these flags:
  - `ALG_OP_ENCRYPT` - encryption of data
  - `ALG_OP_DECRYPT` - decryption of data
- specification of the IV information marked with the flag `ALG_SET_IV`
- specification of the associated authentication data (AAD) with the flag `ALG_SET_AEAD_ASSOCLEN`. The AAD is sent to the kernel together with the plaintext / ciphertext. See below for the memory structure.

The `send` system call family allows the following flag to be specified:

- `MSG_MORE`: If this flag is set, the `send` system call acts like a cipher update function where more input data is expected with a subsequent invocation of the `send` system call.

Note: The kernel reports `-EINVAL` for any unexpected data. The caller must make sure that all data matches the constraints given in `/proc/crypto` for the selected cipher.

With the `recv()` system call, the application can read the result of the cipher operation from the kernel crypto API. The output buffer must be at least as large as defined with the memory structure below. If the output data size is smaller, the cipher operation is not performed.

The authenticated decryption operation may indicate an integrity error. Such breach in integrity is marked with the `-EBADMSG` error code.

## AEAD Memory Structure

The AEAD cipher operates with the following information that is communicated between user and kernel space as one data stream:

- plaintext or ciphertext
- associated authentication data (AAD)
- authentication tag

The sizes of the AAD and the authentication tag are provided with the `sendmsg` and `setsockopt` calls (see there). As the kernel knows the size of the entire data stream, the kernel is now able to calculate the right offsets of the data components in the data stream.

The user space caller must arrange the aforementioned information in the following order:

- AEAD encryption input: AAD || plaintext
- AEAD decryption input: AAD || ciphertext || authentication tag

The output buffer the user space caller provides must be at least as large to hold the following data:

- AEAD encryption output: ciphertext || authentication tag
- AEAD decryption output: plaintext

## Random Number Generator API

Again, the operation is very similar to the other APIs. During initialization, the `struct sockaddr` data structure must be filled as follows:

```
struct sockaddr_alg sa = {
    .salg_family = AF_ALG,
    .salg_type = "rng", /* this selects the random number generator */
    .salg_name = "drbg_nopr_sha256" /* this is the RNG name */
};
```

Depending on the RNG type, the RNG must be seeded. The seed is provided using the `setsockopt` interface to set the key. For example, the `ansi_cprng` requires a seed. The DRBGs do not require a seed, but may be seeded. The seed is also known as a *Personalization String* in NIST SP 800-90A standard.

Using the `read()/recvmsg()` system calls, random numbers can be obtained. The kernel generates at most 128 bytes in one call. If user space requires more data, multiple calls to `read()/recvmsg()` must be made.

WARNING: The user space caller may invoke the initially mentioned `accept` system call multiple times. In this case, the returned file descriptors have the same state.

Following CAVP testing interfaces are enabled when kernel is built with `CRYPTO_USER_API_RNG_CAVP` option:

- the concatenation of *Entropy* and *Nonce* can be provided to the RNG via `ALG_SET_DRBG_ENTROPY` `setsockopt` interface. Setting the entropy requires `CAP_SYS_ADMIN` permission.
- *Additional Data* can be provided using the `send()/sendmsg()` system calls, but only after the entropy has been set.

## Zero-Copy Interface

In addition to the send/write/read/recv system call family, the AF\_ALG interface can be accessed with the zero-copy interface of splice/vmsplice. As the name indicates, the kernel tries to avoid a copy operation into kernel space.

The zero-copy operation requires data to be aligned at the page boundary. Non-aligned data can be used as well, but may require more operations of the kernel which would defeat the speed gains obtained from the zero-copy interface.

The system-inherent limit for the size of one zero-copy operation is 16 pages. If more data is to be sent to AF\_ALG, user space must slice the input into segments with a maximum size of 16 pages.

Zero-copy can be used with the following code example (a complete working example is provided with libkcapi):

```
int pipes[2];

pipe(pipes);
/* input data in iov */
vmsplice(pipes[1], iov, iovlen, SPLICE_F_GIFT);
/* opfd is the file descriptor returned from accept() system call */
splice(pipes[0], NULL, opfd, NULL, ret, 0);
read(opfd, out, outlen);
```

## Setsockopt Interface

In addition to the read/recv and send/write system call handling to send and retrieve data subject to the cipher operation, a consumer also needs to set the additional information for the cipher operation. This additional information is set using the setsockopt system call that must be invoked with the file descriptor of the open cipher (i.e. the file descriptor returned by the accept system call).

Each setsockopt invocation must use the level SOL\_ALG.

The setsockopt interface allows setting the following data using the mentioned optname:

- ALG\_SET\_KEY -- Setting the key. Key setting is applicable to:
  - the skcipher cipher type (symmetric ciphers)
  - the hash cipher type (keyed message digests)
  - the AEAD cipher type
  - the RNG cipher type to provide the seed
- ALG\_SET\_AEAD\_AUTHSIZE -- Setting the authentication tag size for AEAD ciphers. For an encryption operation, the authentication tag of the given size will be generated. For a decryption operation, the provided ciphertext is assumed to contain an authentication tag of the given size (see section about AEAD memory layout below).
- ALG\_SET\_DRBG\_ENTROPY -- Setting the entropy of the random number generator. This option is applicable to RNG cipher type only.

## User space API example

Please see [1] for libkcapi which provides an easy-to-use wrapper around the aforementioned Netlink kernel interface. [1] also contains a test application that invokes all libkcapi API calls.

[1] <https://www.chronox.de/libkcapi.html>