Tampereen yliopisto

Miko Ropponen

ON OPENSSL'S EVP API

Faculty of Information Technology and Communication Sciences
Bachelor of Science Thesis
April 2020

ABSTRACT

Miko Ropponen: On OpenSSL's EVP API

Bachelor of Science Thesis

Tampere University

Bachelor's Degree Programme in Computing and Electrical Engineering

April 2020

This paper examines OpenSSL's EVP API, focusing on its usability and how to perform cryptographic operations using it. The usability of cryptographic APIs is an interesting topic, as usability problems will often lead to security problems as well, due to incorrect use of the API. Incorrect usage of security APIs has already led to significant data breaches. Yet, sometimes the security of even a non-security API might require trade-offs in its usability, leading to a contradiction of interests in API design, especially so for security APIs such as EVP.

Other cryptographic library APIs are also evaluated on their usability, namely NaCl, Libsodium and Nettle. The four APIs mentioned in this abstract are then compared to each other, not to find the "best" one out of them on any scale, but rather to analyse their similarities and differences. The usability evaluation relies on documentation available for these APIs. The analysis is done by evaluating the APIs based on Jacob Nielsen's "heuristic evaluation" guidelines for analysing the usability of general user interfaces, mapped to API usability assessment by Brad A. Myers and Jeffrey Stylos, with some further mapping done in this paper to take into account the security nature of the APIs assessed.

Comparison of the usability of the APIs found that all of the APIs satisfy the assessment guidelines rather well, with only minor problems found. Considering the cryptographic nature of the APIs, an argument for many of these guideline contradictions could be made. A big difference could be found in the flexibility of the APIs, that is, the amount of features the API provides. These features include the available cryptographic primitives to perform operations with. A part of this difference could be explained by the level of abstraction the API provides, and also by their different design strategies, where some of the libraries (NaCl and Libsodium) argue that it is better to simply provide the most secure cryptographic primitive. Nettle and EVP take a different approach, providing a wide range of different cryptographic primitives, allowing the libraries to be used in a wider range of contexts. Similarities include the usage of sane defaults in all of the APIs, made evident by the fact that the default usage for each of the APIs works without much set-up of options etc. Another similarity is the fact that due to the libraries being written in C-programming language, the parameter lists for the methods available tend to grow rather large, which goes against one of the evaluation guidelines. This is mainly caused by the fact that the length of the input and output have to be passed along with the actual input and output, which would not be a problem in a higher-level programming language such as C++. This alone is not really an argument for writing the APIs in C++, though.

Further examination of EVP's public-facing API is also done This paper does not go into the specific technical implementation details of EVP. Rather, this paper goes over how to perform the core cryptographic operations of asymmetric encryption, symmetric encryption and message digests using EVP. Despite the fact that EVP is written in C, which does not support object-oriented programming, the library internals are driven by function pointers, making some of the details of EVP resemble objects, such as the context- and method-structures used to perform the operations.

Keywords: OpenSSL, EVP, API, API Usability, Cryptographic library

TABLE OF CONTENTS

1. INTRODUCTION 1
2. WHAT IS EVP 2
    2.1 Application Programming Interfaces 2
    2.2 EVP as an API 2
3. EVP AND OTHER CRYPTOGRAPHIC LIBRARIES 3
    3.1 API Usability 3
    3.2 NaCl 5
    3.3 Libsodium 7
    3.4 Nettle 8
    3.5 EVP 10
    3.6 Comparisons 15
4. FURTHER EXAMINATION OF EVP 17
    4.1 The overall usage 17
    4.2 Message digests 18
    4.3 Symmetric encryption and decryption 19
    4.4 Public key operations 21
5. CONCLUSION 23
SOURCES 24

1. INTRODUCTION

Application Programming Interfaces (API) are an integral part of modern-day programming. In fact, according to Brad A. Myers and Jeffrey Stylos [1], API calls make up most of the code written by most programmers, with many organizations offering their code or data to the public through APIs. This paper focuses on one such API, OpenSSL's EVP (from the word "envelope").

OpenSSL is a commercial-grade, general purpose cryptography and Transport Layer Security/Secure Sockets Layer toolkit, and it is the most used cryptographic library amongst its peers for application development. [2,3] While a programmer that wishes to use the functionality provided by OpenSSL in their application could use the low-level implementations for different cryptographic functions, this approach requires said programmer to be familiar with not only the details of different cryptosystems, but also its implementation in OpenSSL. It is reasonable to argue that a better approach to access the wanted functionality is through the EVP API, which abstracts the low-level implementations behind a high-level API. This offers benefits such as increasing code reusability and adaptability as the same API calls can be used for multiple cryptosystems, on top of being less error-prone due to decreased complexity of code and required knowledge about different cryptosystems and their OpenSSL implementations.

A core point of view for this paper is EVP's usability, as an incorrectly used API can result in major security problems [1]. Considering OpenSSL's nature of being a cryptographic library, an API prone to security problems due to its complexity would be perhaps even more counterproductive than in a more general library. Further examination on what features EVP provides is also done. This examination focuses on OpenSSL's version 1.1.1.

This paper also looks at other security APIs, namely NaCl, Libsodium and Nettle, and examines their similarities and differences with EVP, and evaluates their usability using certain usability attributes and guidelines. The usability evaluation will be done by examining the documentation available for these APIs. The second chapter introduces EVP and its goals as a security API. The third chapter focuses on the usability evaluation for the APIs. In the fourth chapter, EVP's public facing API is further examined, but this paper will not go into detail on EVP's internal workings. Conclusions are presented in the fifth chapter.

2. WHAT IS EVP

2.1 Application Programming Interfaces

An API's purpose in general terms is to provide an interface between a library and an application, between applications, or within different parts of an application (for example, between the graphical user interface and its functionality). This interface works on a higher abstraction level from the actual implementation of the functions "behind" the API, that is, the programmer using the API does not need to know much about the actual low-level implementations. Rather, the programmer calls the API, which then uses the low-level implementations abstracted from the programmer, and returns values/structures etc. accordingly, which the programmer can then either use as is, or continue to manipulate through the API. This not only simplifies programming on the programmer's part, but also makes it so that when the provider of the API updates the low-level code, no additional action is needed from the API user, provided that the API itself does not change. OpenSSL ensures API/ABI (Application Binary Interface) compatibility within a major version, where a version consists of MAJOR.MINOR.PATCH (e.g. version 1.1.1) [2].

2.2 EVP as an API

The purpose of EVP is to make the OpenSSL cryptographic library accessible to developers without expertise in cryptography. Compared to other, more general library APIs such as the POSIX API, much of the complexity is hidden behind EVP, away from the external application. The goal is that the external application can use EVP to simply express what it wants done, e.g. encrypting a file with a certain key. No key specific code needs to be written, EVP will recognize the key type and act accordingly. EVP also mostly removes the need for the external application developer to write code to catch different kinds of signals (e.g. interrupt, error, end of stream), and due to its use of sane defaults, there is no need for complicated set-up or initialisation.

3. EVP AND OTHER CRYPTOGRAPHIC LIBRARIES

OpenSSL is not the only cryptographic library available, and it is worth examining how some of the other cryptographic libraries solve the issue of providing a secure and accessible interface to the library user. When it comes to crypto libraries besides EVP, this paper will not go into much detail on how to perform cryptographic operations using them. Rather, this paper will focus on examining their usability and the features they offer.

The other cryptolibraries are also compared to EVP. The purpose here is not to figure out which cryptographic library is “the best”, but rather just to examine their similarities and differences. Usability is an interesting topic when it comes to examining crypto library APIs, as sometimes increasing the API security might require a trade-off in its usability. However, an API with poor usability will often lead to said API being used incorrectly, which, in the context of crypto libraries especially, can lead to major security problems. Incorrect use of a security API has led to serious data breaches affecting a massive number of end-users worldwide. [1, 11]

3.1 API Usability

This paper examines the usability of select crypto libraries through some of the usability attributes presented in Brad A. Myers’ and Jeffrey Stylos’ article “Improving API Usability” [1]. The examined attributes are simplicity, consistency, and expressiveness. Simplicity measures how simple the API is to use, e.g. the number of steps required to complete some sort of task. Consistency can be measured with how well different parts of the API (such as parameter lists or return values) follow patterns i.e. stay consistent, and expressiveness refers to the range of features or options the API offers. Some of the attributes are omitted because quantifying them would require a user study.

Jacob Nielsen’s [5,6] “heuristic evaluation” guidelines are also used to evaluate the APIs in question. While these guidelines are for evaluating the usability of a user interface, using Brad A. Myers’ and Jeffrey Stylos’ mapping, they can be used to evaluate APIs as well [1]. Some additional conditions and special cases are also explored, as security APIs should (or at least, often must be) arguably be more restrictive compared to general APIs. The guidelines are as follows, along with a brief explanation:

Visibility of system status. The API user should be able to check the system state easily and trying to use operations that are not available for certain system states should provide helpful feedback. This is an area of usability where the security-aspect of a cryptographic library API comes into play, as there's a limit to how transparent these can be to avoid the possibility of cryptographic oracles, where an attacker could deduce information on the system status and gain access to secret information by examining how the system responds to certain inputs;

Match between system and real world. Method and class naming conventions should follow intuitive sense, and methods should be part of the class where the API user expects to find said method;

User control and freedom. Reverting or aborting operations should be simple, and it should be easy for the API user to revert the API to a clean state. It is worth considering however, that in the context of cryptolibrary APIs, reaching the required security might require trade-offs with user control and freedom especially. Some of the cryptographic primitives are not reversible to begin with (e.g. you cannot get the plaintext back from a one-way cryptographic hash function);

Consistency and standards. The design of the API should stay consistent, e.g. calling a feature within the API by multiple names should be avoided. This ties in with another guideline, aesthetic and minimalist design, as well as match between system and real world. In the case of minimalist design, it should be noted that increasing the number of features in the API does not necessarily reduce its usability [1], if the features are clearly named and organized;

Error prevention. Through sane defaults and helpful feedback, the API should assist the user with using the API properly. In case an error does happen, the API should follow another guideline, help users recognize, diagnose, and recover from errors. The API should supply the user with helpful error messages, and provide a possible solution to the problem;

Recognition rather than recall. Method and class names should be easily recognizable by the API user, rather than having to memorize them due to the names being unnecessarily complex or unclear, or easy to confuse with each other;

Flexibility and efficiency of use. Flexibility can be considered to mean expressiveness, but this is another case where the special nature of cryptolibrary APIs should be considered. While it is important that the API can be used efficiently, there is an argument for reducing flexibility in favour of simply using the most secure/compatible options [7].

Efficiency of use is rather self-explanatory, but could be viewed as “how much code does an API user have to write for a certain operation);

Help and documentation. An important aspect of an API’s usability is sufficient documentation. The documentation should of course be well organized and easily searchable. Easy to use documentation specifically has a substantial role in non-security expert API-users being able to write secure code. [1,6,11] These guidelines cover the attributes (simplicity, consistency and expressiveness) presented at the start of this chapter. Note that the usability analysis relies on the documentation available for the APIs.

3.2 NaCl

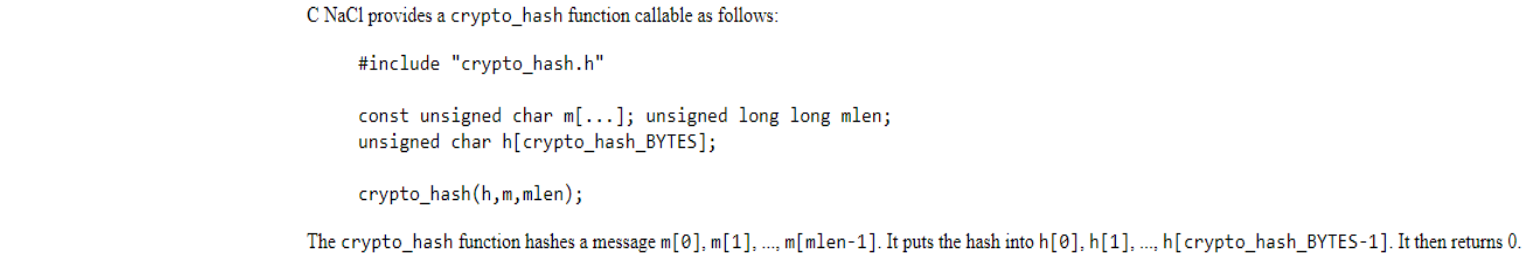
NaCl (Networking and Cryptography library, pronounced “salt”), is a cryptographic library for securing network communication, encryption, and decryption etc. In its design, NaCl has a very clear emphasis on simplicity, and having a reduced feature kit when it comes to the number of different cryptographic primitives. For example, when it comes to message digest algorithms, only Secure Hash Algorithm (SHA) is supported, with two options for digest sizes (SHA-256 and SHA-512). Similarly, only AES and Salsa20 are available for encryption. [8] Being conservative when it comes to choosing which primitives to implement is an intentional strategy of the development team, choosing to omit cryptographic primitives that they consider too low security, i.e. take considerably fewer than 2^{128} operations to break [7]. With all this, NaCl is a high-level cryptographic library.

This paper will focus on the C implementation of NaCl. C++ and Python implementations are available as wrappers for the C implementation, but since EVP is written in C, we will exclude the rest. This does have some small implications when it comes to usability. For example, functions require more parameters in the C implementation than in the C++ implementation, which does have a negative effect on usability [1]. However, this is more of a difference between C and C++, and this more or less affects all the inspected APIs equally. It is worth noting however, that OpenSSL uses function pointers in its library internals, which resemble object-oriented programming not strictly available in C. An example of this would be the context-structures of EVP, which very closely resemble objects.

Performing cryptographic operations with NaCl is very simple. NaCl abstracts the operations at a high-level, requiring only a single function call to perform a certain cryptographic operation, storing the result of the cryptographic primitive in a simple char-type

array. Figure 1 demonstrates hashing a message using NaCl. Using the rest of the operations provided by NaCl is equally simple, and due to its simplicity, not much documentation is needed. Documentation is however available for it, and it is helpful and well organized.

The system status (e.g. is something hashed or not) is obvious to the user from examining the result array, e.g. the h-variable in Figure 1. Most of the functions satisfy the guideline of match between system and real world, and they are easy to recognize and are kept simple, however the function for public-key authenticated encryption is called crypto_box, which doesn't really imply it's functionality in any sort of intuitive fashion.


C NaCl provides a crypto_hash function callable as follows:

```
#include "crypto_hash.h"

const unsigned char m[...]; unsigned long long mlen;
unsigned char h[crypto_hash_BYTES];

crypto_hash(h,m,mlen);
```

The crypto_hash function hashes a message m[0], m[1], ..., m[mlen-1]. It puts the hash into h[0], h[1], ..., h[crypto_hash_BYTES-1]. It then returns 0.

Figure 1: *Hashing a message with NaCl. The image is a screenshot from https://nacl.cr.yp.to/hash.html*

The function names and parameter order do stay consistent. On the topic of parameters, some of the functions do require a rather large amount of them, and they are often of the same datatype, such as in the case of the aforementioned crypto_box. This function requires six parameters, with five of them being of the unsigned char datatype. This is something that could be argued [1] to be detrimental to usability. However, in NaCl's case, splitting crypto_box into multiple functions to reduce the number of parameters in a single function would also violate the high-level of simplicity offered by crypto_box.

Crypto_box is also a good example of the efficiency of use provided by NaCl, as it performs public-key authenticated encryption, which is a multi-step operation consisting of multiple cryptographic primitives, in just one function. It should be considered that all this efficiency and simplicity does come at the cost of flexibility, although this is a conscious design-choice on the development team's part. User control and freedom is a bit of a special case, as none of the operations provided do anything that would ever need to be explicitly reversed or aborted, as none of the original data is destroyed. For example, hashing text does not destroy the original plaintext, so there is no need to reverse the hashing.

NaCl is a rather simple library, somewhat lacking in cryptographic agility. This leads to it being able to store everything on stack, not relying on dynamic memory allocation. This

leads to NaCl not really having internal errors, and thus the return codes are mostly used to indicate the result of a cryptographic operation. For example, when verifying a message with *crypto_box_open()*, a return code of -1 means that the verification fails, indicating that the message's authenticity is compromised, not that NaCl ran into an internal error. Successful operations return a 0.

3.3 Libsodium

Libsodium is a cryptographic library that extends NaCl and claims to further improve the NaCl API's usability. It adds support for a multitude of different programming languages that NaCl does not have an implementation on. Libsodium's design emphasises security and ease of use [9], and it continues NaCl's strategy of being conservative when it comes to choosing what cryptographic primitives to implement. Another similarity is the high level of abstraction provided by the API.

When it comes to the core cryptographic operations, using Libsodium does not much differ from NaCl. For comparison, Figure 2 provides an example of the C implementation for hashing a message using Libsodium. As with NaCl, a message input (MESSAGE) can be hashed to an output (hash) with just one function call. Of note is the increased number of function parameters needed. Compared to NaCl's implementation, specifying the length of the hash as well as a possible key and its length are also required. This optional key can be used for salting the hash, increasing the expressiveness of Libsodium's hash-function, and so this increase in the number of parameters can be viewed as a trade-off between simplicity and expressiveness, although the cost in simplicity is by no means excessive. Other operations follow suit in their similarity of use.

```c
#define MESSAGE ((const unsigned char *) "Arbitrary data to hash")
#define MESSAGE_LEN 22

unsigned char hash[crypto_generichash_BYTES];

crypto_generichash(hash, sizeof hash,
                   MESSAGE, MESSAGE_LEN,
                   NULL, 0);
```

Figure 2: *Hashing a message with Libsodium: The image is a screenshot from https://libsodium.gitbook.io/doc/hashing/generic_hashing*

When it comes to the usability guidelines, most of the analysis for NaCl is valid for Libsodium as well, and so this section does not evaluate libsodium with the heuristic evaluation step-by-step, but rather focuses on differences between libsodium and NaCl. The core differences between them have to do with flexibility (or expressiveness) as well as documentation. As already mentioned, Libsodium provides an extended set of features compared to NaCl, e.g. support for hashing a stream instead of a single message block, and thus its expressiveness is higher compared to NaCl.

The documentation is also improved upon, with increased robustness for all of the cryptographic primitives compared to NaCl's documentation. It also provides additional documentation for topics that are not entirely essential but nevertheless increase usability. For example, the QuickStart- and Frequently Asked Questions -section, which guides the API user in how they can securely accomplish something using libsodium, and also covers some confusion for non-cybersecurity experts, such as explaining the difference between a password and a secret key. This could be argued that a section like this increases the efficiency of use and user-friendliness of the API, as well as eliminates possible security problems caused by inexperience from the API-programmer's part. [11]

3.4 Nettle

Nettle is a low-level cryptographic library that is intended to work in a variety of environments, be it cryptographic tools or kernel space. Because of this, Nettle does not implement anything besides the low-level cryptographic primitives and a straightforward interface to access them. The library is written in C, with bindings available for a few other languages. [10]

Nettle implements multiple different cryptographic primitives, including legacy ones (at least in the case of one-way hash functions) for backwards compatibility. The selection of the algorithm to use falls upon the API programmer, and Nettle requires algorithm specific code, though it does not require any sort of knowledge of the actual inner workings of any of the cryptographic primitives or their specific implementations in Nettle. Figure 3 gives an example of using Nettle by demonstrating hashing with Nettle.

A simple example program that reads a file from standard input and writes its SHA1 check-sum on standard output should give the flavor of Net

```
#include <stdio.h>
#include <stdlib.h>

#include <nettle/sha1.h>

#define BUF_SIZE 1000

static void
display_hex(unsigned length, uint8_t *data)
{
  unsigned i;

  for (i = 0; i<length; i++)
    printf("%02x ", data[i]);

  printf("\n");
}

int
main(int argc, char **argv)
{
  struct sha1_ctx ctx;
  uint8_t buffer[BUF_SIZE];
  uint8_t digest[SHA1_DIGEST_SIZE];

  sha1_init(&ctx);
  for (;;)
  {
    int done = fread(buffer, 1, sizeof(buffer), stdin);
    sha1_update(&ctx, done, buffer);
    if (done < sizeof(buffer))
      break;
  }
  if (ferror(stdin))
    return EXIT_FAILURE;

  sha1_digest(&ctx, SHA1_DIGEST_SIZE, digest);

  display_hex(SHA1_DIGEST_SIZE, digest);
  return EXIT_SUCCESS;
}
```

Figure 3: *Hashing a file with Nettle. The image is a screenshot from http://www.lysator.liu.se/~nisse/nettle/nettle.html*

While the example seems longer than the previous ones from Figures 1 and 2, it does include a lot of extra code not related to Nettle, such as reading a file and printing the resulting hash. The Nettle specific lines are s*truct sha1_ctx ctx, sha1_init(), sha1_update()* and *sha1_digest().* This also demonstrates the need for algorithm-specific code; in this case the algorithm is SHA1, but if a programmer wanted to use for example, SHA-256 then the methods used would also change to *sha256_ctx, sha256_init()* etc. Another difference to the previous examples is that Figure 3 demonstrates hashing a stream instead of a block of data.

Nettle's documentation doesn't list any sort of mechanism to check the system state, but by looking at Figure 3's example, it should be fairly obvious how the programmer could make sure that all of the data given is operated over (by calling the _update()-method enough times for the entire stream/block), and when the operation is finished (which in this case would be when uint8_t digest[] has content). For the purposes of this API, there should not ever be a reason to examine a partially done hash or encryption to begin with. As was already brought up, the usage for other message digest algorithms stays con-

sistent with the example, and this can be generalised to other operations (e.g. encryption): naming conventions and the API usage stays consistent between algorithms and operations, satisfying both match between system and real world as well as consistency and standards. They are also easily recognizable, satisfying recognition rather than recall. Return values also stay consistent whenever a function does have a return value, 1 is used to indicate success, 0 indicating failure. Functions that do not have a return value listed in the documentation cannot fail [10].

When it comes to user control and freedom, aborting an operation is as simple as calling the _init()-function for the specific algorithm you are using, which resets the context-structure given to it as a parameter. Besides that, much of the analysis for the other APIs is valid here too: sometimes you cannot revert operations because they are one-way functions. For something like secret-key encryption, reverting an encryption could be done by simply decrypting the ciphertext, given that the key is still intact and usable.

With error prevention, diagnosis and recovery, there is again a limit to how helpful and specific the API can be so as to not leak private data. However, the documentation does list a return value in case of failure for any function that can fail and explains briefly why such a failure would happen e.g. a key was not valid.

As a low-level general-purpose crypto tool API, Nettle is very expressive and flexible in this context. It offers a great range of different cryptographic primitives, even legacy ones for backwards compatibility. However, as it does require algorithm specific code, a program using Nettle might have to be updated to use newer and more secure algorithms as security standards evolve. This paper could not find criticism in Nettle’s efficiency of use.

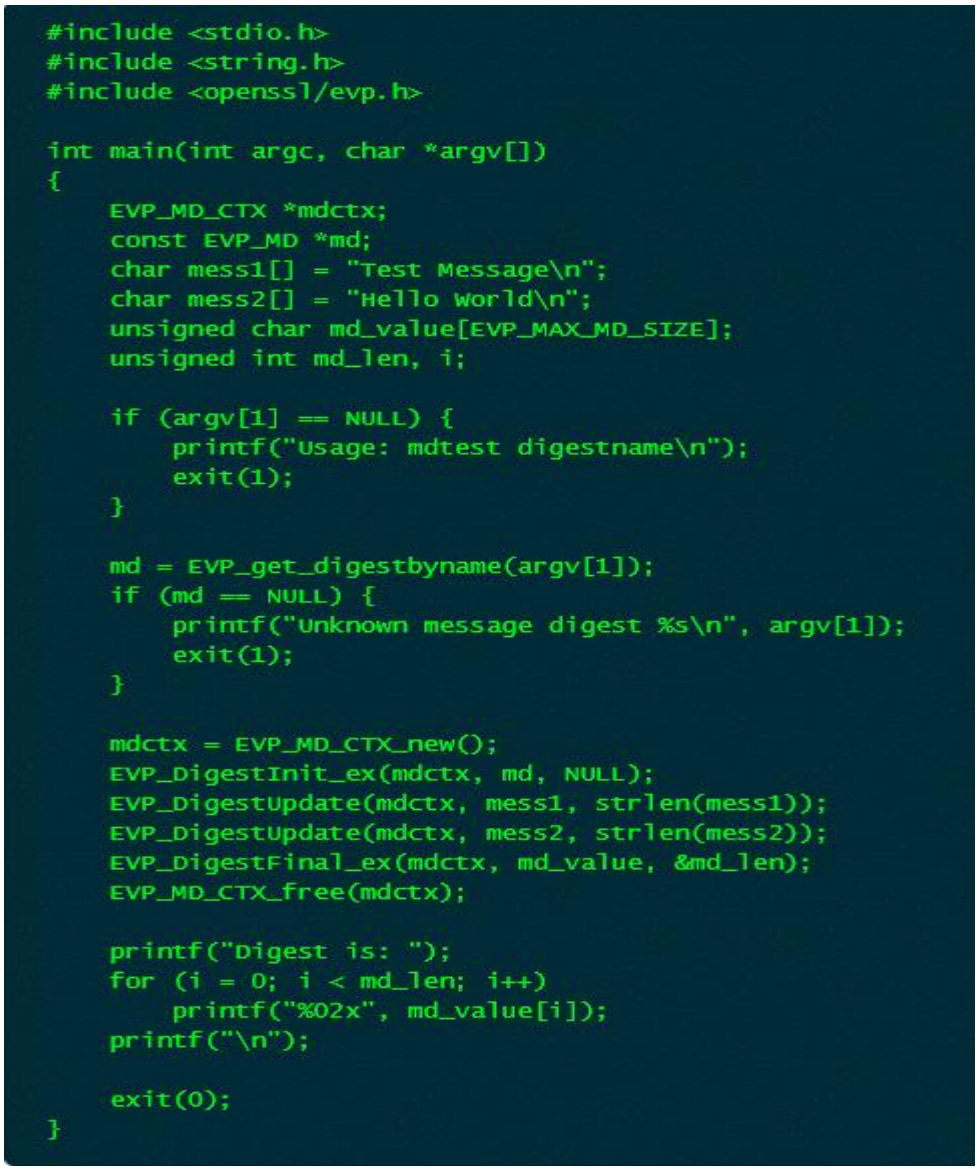
Nettle’s documentation is helpful, even providing context and background for different cryptographic primitives to aid the API user in selecting an appropriate algorithm to use for their application. The documentation also warns users of common pitfalls and security risks associated with them, such as warning the user not to use the legacy hash functions provided for new applications that do not need backwards compatibility. While it is not the most modern looking one, finding what you are looking for is not hard either, or it is well organized into sections.

3.5 EVP

OpenSSL’s EVP API provides an interface to the cryptographic operations provided by OpenSSL or in other words, to the functionality a programmer would expect from a cryptographic toolbox [4]. EVP is split into submodules, with three of them being of particular

interest, called EVP_PKEY, EVP_MD and EVP_CIPHER. These abstract the asymmetric encryption, message digest and symmetric encryption cryptosystems, respectively. Each of these provide their respective abstract context and method structures, for example, the context-structure for message digest cryptosystems is called EVP_MD_CTX. These structures provide the means to manipulate and inspect the structure's parameters or internal status, as well as access to the abstracted functions that implement the cryptographic functions.[3] Figure 4 demonstrates hashing a message with EVP as a comparison to the other APIs examined. Like with Nettle, this example also has non-API-related code, such as printing the resulting hash and some sanity checks for user input. EVP-related methods are obvious in Figure 4, in that they all begin with "EVP_". It is worth mentioning that a wrapper for EVP_DigestInit_ex(), EVP_DigestUpdate() and EVP_DigestFinal_ex() exists as EVP_Digest(), which can alternatively be used to hash a block of data.

Much like Nettle, multiple different algorithms are available through EVP, with the difference being that EVP does not require algorithm-specific code. This is evident from Figure 4, where the algorithm the API user wishes to use is selected through user-input with EVP_get_digestbyname(). This is analogous to performing symmetric encryption, where an algorithm can be selected with EVP_get_cipherbyname(). Public-key-operations can likewise be performed without the need to write key-specific code.



```c
#include <stdio.h>
#include <string.h>
#include <openssl/evp.h>

int main(int argc, char *argv[])
{
    EVP_MD_CTX *mdctx;
    const EVP_MD *md;
    char mess1[] = "Test Message\n";
    char mess2[] = "Hello World\n";
    unsigned char md_value[EVP_MAX_MD_SIZE];
    unsigned int md_len, i;

    if (argv[1] == NULL) {
        printf("Usage: mdtest digestname\n");
        exit(1);
    }

    md = EVP_get_digestbyname(argv[1]);
    if (md == NULL) {
        printf("Unknown message digest %s\n", argv[1]);
        exit(1);
    }

    mdctx = EVP_MD_CTX_new();
    EVP_DigestInit_ex(mdctx, md, NULL);
    EVP_DigestUpdate(mdctx, mess1, strlen(mess1));
    EVP_DigestUpdate(mdctx, mess2, strlen(mess2));
    EVP_DigestFinal_ex(mdctx, md_value, &md_len);
    EVP_MD_CTX_free(mdctx);

    printf("Digest is: ");
    for (i = 0; i < md_len; i++)
        printf("%02x", md_value[i]);
    printf("\n");

    exit(0);
}
```

Figure 4: *Hashing a message with EVP. The image is a screenshot from* https://www.openssl.org/docs/manmaster/man3/EVP_DigestInit.html

As already mentioned, the context-structures provide a means for a programmer to inspect the status of the system within the constraints of a security API. These constraints mean cases like trying to extract a secret key from a CIPHER_CTX, which obviously would be quite insecure. Using unavailable operations for certain system states will result in the operation returning a 0 for failure, which the API user should check for in their code. EVP uses 0 as a return value to indicate a failure in general. While it is not strictly a part of the EVP API, it should be mentioned that OpenSSL does have a mechanism for outputting error-messages. Thus, *visibility of system status*, *error prevention* and *help users recognize, diagnose and recover from errors* are satisfied. Much of the same analysis for the other APIs is valid for EVP as well when it comes to *user control and freedom*: reverting operations is usually infeasible due to security concerns, but methods to reset

the API to a clean state are provided. An additional feature assisting with *error prevention* is the usage of sane defaults, which is evident from Figure 4. The context- and method-structures simply need to be initialised with their respective functions, after which they are ready for use, without the need for setting up any sort of options.

EVP complies with match between system and real world reasonably well. However, as it does offer a rather large number of features to the API user, not all of its methods are necessarily immediately obvious to the user. Using the message digest submodule as an example, the submodule has two similar looking methods available: EVP_DigestInit_ex() and EVP_DigestInit(). The difference between these methods is that the EVP_DigestInit_ex() takes a pointer to an ENGINE-object as a parameter to provide a different implementation for the given message-digest algorithm (such as an updated version that hasn't yet been implemented in the most recent OpenSSL version), and EVP_DigestInit() simply uses the default implementation for the given algorithm. If the ENGINE-parameter is a NULL-pointer, EVP_DigestInit_ex() uses the default implementation as well. This method offers other advantages such as making it possible to reuse a digest context-structure without the need for clean-up and reallocation between calls. Thus, EVP_DigestInit_ex() is the one that is recommended for new applications to use, and EVP_DigestInit() exists mostly for compatibility reasons. This could be a possible source of confusion for an API user and could be argued to go against *recognition rather than recall*, but the documentation for EVP does explain the difference. The naming for methods that an API programmer would expect to use for a certain operation are not overly complicated, and this source of confusion is not a reoccurring case.

EVP's naming conventions stay consistent where possible within the submodules. Indeed when it comes to the message digest and symmetric encryption submodules, the usage of EVP is essentially analogous, with MD changing to CIPHER and Digest changing to Encrypt (e.g. EVP_MD_CTX changes to EVP_CIPHER_CTX and EVP_DigestInit() changes to EVP_EncryptInit() etc.). There are no consistency issues in the public-key submodule either.

EVP is certainly very flexible. It offers a wide range of different cryptographic primitives for all of its submodules that an API user can use efficiently owing to the fact that EVP is algorithm independent. These cryptographic primitives also have multiple different behaviours that the programmer might wish to use, such as setting a cipher mode, manipulating the initialisation vector length etc, which can all be done through EVP. This provides the programmer with a great deal of control should they wish for it.

EVP's documentation leaves a bit to be desired. While it is very robust, explaining every method thoroughly and offers advice to the user, navigating the documentation is a bit cumbersome. Lacking any sort of organized list of documentation pages, the user has to jump between pages through hyperlinks in the "See Also" section of any given page, which makes finding exactly what the user is looking for more difficult than it could, and perhaps should be. The search-bar is also of limited help, as it simply redirects to a google search from the site openssl.org. This is demonstrated in Figure 5, where "digest" was searched for using the search-bar at https://www.openssl.org/docs/man1.1.1/. Note that the first three search results are for OpenSSL branches 1.1.0, master and 1.0.2 respectively, when the search was initiated from the manual page for 1.1.1.

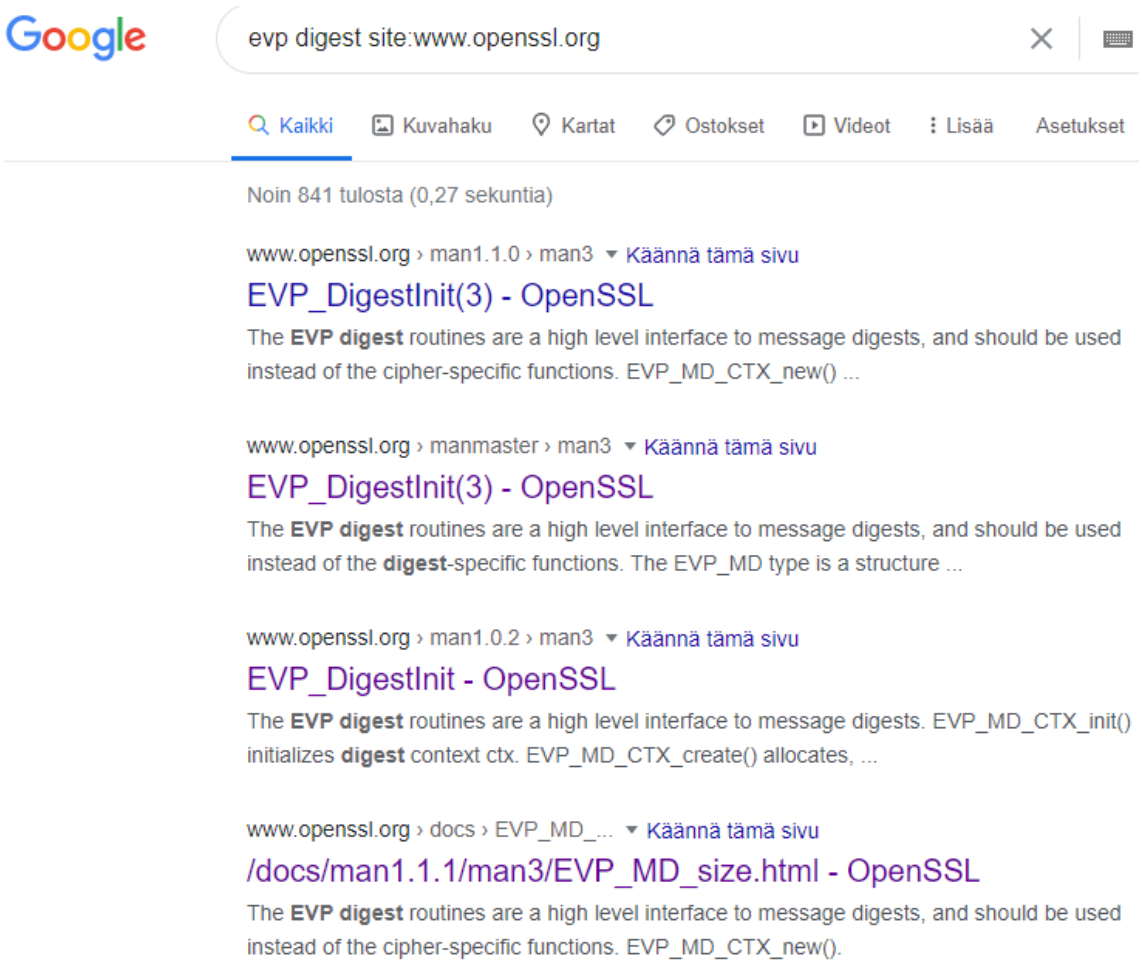


Figure 5: *Using the search bar in EVP's documentation*

The search results are also a bit misleading. The EVP_DigestInit-result(s) will indeed lead to the page (for their respective versions) where a user would find general information about the message digest submodule, but the pages are named after one of the methods in said submodule, *EVP_DigestInit().* The fourth result, despite the name being

entirely different, will lead to the same page, but for version 1.1.1. For someone not already familiar with the structure of the documentation, this is confusing at best. So, while the documentation is robust and encompassing, it could be better organized and more searchable. The current documentation is extracted from *man-page* software documentation for Unix-operating systems.

3.6 Comparisons

All of the APIs analysed provide an interface for the “main” operations expected of a cryptographic toolbox: message digests, symmetric encryption, and asymmetric algorithms (public-key functions). One of the obvious differences is the flexibility of the APIs, in the sense of how many cryptographic primitives are provided for the given operation. NaCl is by far the most restrictive, deliberately choosing to implement only a select few cryptographic primitives, arguing that this is the more secure option compared to leaving the choice of the primitive to the programmer [7]. As Libsodium extends NaCl, it does extend flexibility, but not by adding options for different primitives, more so by adding different modes for the operations (such as operating on data streams, rather than chunks) [9]. Nettle and OpenSSL take an entirely different approach to the number of primitives provided, allowing a great deal of choice to the API user, even offering legacy options for backwards compatibility, that are not recommended for use in new applications [2,10]. Certainly, there are arguments for both approaches.

Another difference is how algorithm-independent the API is, that is, is there a need for primitive-specific code. NaCl’s default functions presented for any operation (e.g. *crypto_stream()* symmetric encryption) do not need any sort of algorithm specific code, but the limited alternative primitives do. For example, an alternative primitive provided for symmetric encryption is AES, and the function for it is *crypto_stream_aes128ctr().* Libsodium follows suit in this, where the default implementation is algorithm-independent, but the alternatives are not. Nettle on the other hand is entirely algorithm-dependent, where each of the primitives have their own interface, but this is deliberate, as Nettle is a low-level interface. Finally, EVP requires zero algorithm-dependent code.

EVP, Nettle and, to an extent, Libsodium have some similarity in their use of context-structures (called *ctx* in Nettle and EVP, *state* in Libsodium), although these structures do have differences in what purpose the structure fills in the abstraction-scheme. To elaborate, a context-structure in EVP holds a method-structure as well, which holds information about the algorithm to use, compared to context-structures in Nettle and Libsodium, which are algorithm-specific. Libsodium also uses these structures only when

operating on a stream of data, and in this context, they are used to hold the unfinished result of the operation.

Each of the APIs also use sane defaults. What this means in practise is that none of the APIs require much set-up or configuration of options. If a context-structure is needed, it can be initialised in one method or just declared (depending on the API), after which they are ready to use.

4. FURTHER EXAMINATION OF EVP

This section goes into further detail on EVP's public-facing API for OpenSSL's version 1.1.1. While an example for how to use EVP for message digests is already provided in Figure 4, and the usage of the other submodules follow suit in their simplicity when used with default options, the submodules provide extended features for when a programmer wants to take further control. The focus is on the three core submodules, EVP_MD, EVP_CIPHER and EVP_PKEY. Other operations, such as base64 encoding and decoding, as well as limited support for password based encryption are available through EVP [4], but won't be covered in detail, as the password based encryption API is rather small, and the usage for base64 encoding is vastly similar to the main submodules.

4.1 The overall usage

As already mentioned, each of the three submodules have their respective method-structures, and a context-structure. A method-structure holds the implementation of the used cryptographic primitive (e.g. the method structure for SHA-256 is EVP_sha256). The method-structure can be manipulated and inspected through methods provided in EVP [3]. These manipulation-functions allow a programmer to provide a new implementation for an algorithm, increasing the cryptographic agility [12] of EVP. The high-level cryptographic agility of a toolkit is measured by how easily the toolkit allows changing an implementation of an algorithm without having to change the toolkit interface or framework. Being able to change the implementation is important because the implementations will deprecate over time due to increase in computing power or cryptanalysis uncovering flaws in the implementation. [12]. If the default implementation for an algorithm is sufficient, a method-structure for it can simply be created by calling the corresponding function that returns the method-structure wanted, e.g. *EVP_sha256()*. Additional flags to alter the behaviour of the implementation can also be set, such as optimizing for one update. An additional point of note is that all of the functions which require a buffer for input or output as parameters also require said buffer's length.

A context-structure stores a method-structure and uses it to perform the operation on the data. It is also possible to change the algorithm implementation through the context-structure. Some context-structures can also hold another context-structure, for example, a public-key context-structure can be assigned to a message digest context-structure for

digital signatures. These structures can also be used to pass flags to change the behaviour of the algorithm.

4.2 Message digests

EVP provides message digest method-structures (called EVP_MD) for the following algorithms: Blake2, MD2, MD4, MD5, MDC-2, RIPEMD-160, SHA-1, SHA-2, SHA-3, SM3 and WHIRLPOOL. EVP provides a few different ways of how to create these structures. The first way is to call the respective function that returns the method structure of the wanted algorithm (e.g. *EVP_sha256(),* which returns the method structure for SHA-256). The second way is to use *EVP_get_digestbyname()*, *EVP_get_digestbynid()* or *EVP_get_digestbyobj()*, which return an EVP_MD-structure based on the passed name, NID or ASN1_OBJECT. A NID is the numeric identifier of an algorithm, while an ASN1_OBJECT-structure represents the ASN1 (Abstract Syntax Notation One) OBJECT IDENTIFIER type [4]. Using the second way allows the programmer to write an application that allows selecting the used algorithm at run-time (see Figure 4). Arguably, in this context, EVP_get_digestbyname() is the most end-user friendly, as the other options (NIDs or ASN1_OBJECTS) are constants defined in OpenSSL. OpenSSL recommends new applications to use SHA-2 or SHA-3. [4]

A custom method-structure can be created with *EVP_MD_meth_new()*, allowing a programmer to create a new implementation. This paper will not cover creating a new implementation in much detail, but a custom implementation requires setting an initialisation-function, an update-function and a finalisation-function with *EVP_MD_meth_set_init()*, *EVP_MD_meth_set_update()* and *EVP_MD_meth_set_final()*, respectively. Some other details must also be specified, such as the block size for the input and the result. EVP_MD-structures that are already implemented in EVP can also be customised with these *_meth_set_*-functions.

The EVP_MD behaviour can be altered through setting optional flags with *EVP_MD_meth_set_flags()*. These additional options include setting the method to only handle one block of input (*EVP_MD_FLAG_ONESHOT)*, or specifying the algorithm to be an extensible-output function (*EVP_MD_FLAG_XOF*), which means that the output-length of the function can be arbitrary, rather than the usual behaviour of the output being of fixed length. The DigestAlgorithmIdentifier can also be altered through flags.

The context-structure for message digest routines is called EVP_MD_CTX, and one can be created with *EVP_MD_CTX_new()*. The CTX needs to be initialised either with *EVP_DigestInit_ex()* or *EVP_DigestInit()*, of which the former is recommended by

OpenSSL [4]. These functions take a method-structure as a parameter, which the CTX will use to perform the message digest operation, using the initialisation, update and finalisation -functions from the method-structure. An optional pointer to an ENGINE-object can be passed to *EVP_DigestInit_ex()*, to provide an alternative implementation for an algorithm. If this pointer is a NULL-pointer, the default implementation will be used. An additional benefit of initialising with the *_ex-function is that the same CTX can be used to perform multiple operations without having to clear and initialise the structure between calls. After initialisation, the digest operation can be performed with *EVP_DigestUpdate()*, which can be called multiple times for the same CTX to hash the data in parts (or to hash an incoming data-stream). The update-function called by *EVP_DigestUpdate()* can also be changed by *EVP_MD_CTX_set_update_fn().* After the operation is done, the hash can be retrieved from the CTX with *EVP_DigestFinal()* or *EVP_DigestFinal_ex()*. *EVP_DigestFinal()* clears up the CTX, meaning that a new CTX must be allocated in case multiple hashing operations are needed. It should be noted that after calling *EVP_DigestFinal_ex()*, *EVP_DigestUpdate()* cannot be called for the same CTX before it is reinitialised with *EVP_DigestInit_ex()*, but as stated, using the *_*ex*-versions for the initialisation and finalisation -functions allows for efficient reuse of the CTX-structure, as *EVP_DigestFinal()* automatically cleans up the CTX, which means that a new one has to be created for every operation. After all the operations are done, the CTX must be cleaned up by calling EVP_MD_CTX_free(), which also frees up the space allocated to the CTX. [4]

Additional controls can be sent to the CTX with *EVP_MD_CTX_ctrl()*, such as setting the length for the output of an extendable-output function. Flags can also be set, cleared or tested by *EVP_MD_CTX_set_flags(), *_clear_flags()* and *_test_flags()* respectively. These flags include optimizing for one update only, instructing the initialisation-functions not to initialise implementation specific data, and setting whether additional data can be provided to a context after calling the finalisation function.

4.3 Symmetric encryption and decryption

The general usage of the symmetric encryption submodule is mostly analogous with the message digest submodule, so it follows the general scheme of: creation of context, initialising the context, calling the update function, calling the finalisation function and finally clearing the context, though the functions do have differences in their specific behaviour due to the differences between the operations. Thus, this subsection focuses mostly on the differences. The method-structure is called EVP_CIPHER, while the context-structure is called EVP_CIPHER_CTX. It should be noted that on top of there being

different method-structures for different cipher algorithms, the block ciphers also have multiple different modes, including block cipher modes and AEAD modes (Authenticated Encryption with Associated Data). For example, the function that returns a method-structure for AES-128 in Cipher Block Chaining mode is *EVP_aes_128_cbc().* Other block cipher modes include but are not limited to Output Feedback mode (OFB) and Galois Counter mode (GCB). The cipher-submodule has both encrypt- and decrypt APIs, as well as a cipher API which can be used for both encryption and decryption, whereas the digest-submodule only has the DigestUpdate API. The encrypt, decrypt and cipher APIs each have their respective initialisation, update and finalisation -function (e.g. *EVP_EncryptInit_ex(), EVP_EncryptUpdate()* and *EVP_EncryptFinal_ex()*). [4]

Like the digest initialisation-function, the symmetric key initialisation-functions require the method-structure and an optional ENGINE-object as parameters. *EVP_CipherInit_ex()* also needs an integer type parameter to specify the mode of the operation (encryption or decryption). They also need the secret key and the initialisation vector (IV) to be used in the operation. Where the digest update-function stores the unfinished hash in the context, from which it can be retrieved with the finalisation-function, the symmetric key update-functions write the encrypted output into a buffer instead. Thus, the finalisation functions do not need to retrieve the result from the context. Their behaviour depends on whether padding is enabled, that is, if the input data is padded to match the block size of the algorithm. If so, the finalisation-functions will encrypt any remaining data and place it in the buffer. If not, and the input data length is not a multiple of the block size, the finalisation-functions will return an error, and they will not encrypt the remaining data. Padding can be enabled or disabled with *EVP_CIPHER_CTX_set_padding().* The input and output -buffer sizes, as well as how many bytes the update-functions will encrypt/decrypt per update-function call have to be specified by the programmer, and it is important to make sure that there is enough space in these buffers. [4]

The submodule provides a number of methods to inspect the context/method-structures. These include functions such as *EVP_CIPHER_block_size()*, which takes an EVP_CIPHER-structure as a parameter and returns the block size of the algorithm used. Similar methods are available for inspecting the key or IV length, or the current block cipher mode. [4]

AEAD modes ensure that the ciphertext is not tampered with, and they are also used to confirm the identity of the one that encrypted the data. Or in other words, confirm the integrity and authenticity of the data. [13] Some special controls are associated with the Authenticated Encryption with Associated Data (AEAD) modes, but this paper will not go

into much detail with them. These controls include setting a non-default IV-length or retrieving the authentication tag generated by the AEAD operation. This tag is used to confirm the integrity and authenticity of the data during decryption. [4]

4.4 Public key operations

A distinction should be made between public key encryption and public key **authenticated** encryption. Public key encryption, or asymmetric encryption, is a slow operation, so for large chunks of data, symmetric encryption is preferred for encrypting the actual message/data. The symmetric key is what is encrypted with asymmetric encryption, along with a hash of the symmetrically encrypted data [4,14]. This is a simplification of what is known as public key authenticated encryption. This subsection first covers the EVP_PKEY-structure, which is used to store private and public keys, and then covers the operations which use said structures. It is worth noting that these operations require the use of the other three submodules.

An empty EVP_PKEY-structure can be created with *EVP_PKEY_new().* Creating a new EVP_PKEY and assigning an existing key to it simultaneously can be done with *EVP_PKEY_new_raw_private_key().* Assigning an existing key to an empty EVP_PKEY can be done with *EVP_PKEY_set1_*()*, where the asterisk can be RSA, DSA, DH and EC_KEY (so for example, *EVP_PKEY_set1_RSA()*). It is also possible to create a new key with *EVP_PKEY_keygen()*. This requires an EVP_PKEY_CTX-structure, which can be created with *EVP_PKEY_CTX_new_id()* or with *EVP_PKEY_CTX_new().* The **new_id*-function creates a context structure without an EVP_PKEY-structure, for the purpose of generating parameters for the key generation operation. After the parameters are set, the CTX is initialised for key generation with *EVP_PKEY_keygen_init()*, after which the *EVP_PKEY_keygen()*-function can be called to generate a key that's stored in the EVP_PKEY-structure it takes as a parameter. [4]

To perform public key encryptions, first, an EVP_PKEY_CTX-structure has to be initialised for encryption with *EVP_PKEY_encrypt_init().* This CTX-structure has to have an EVP_PKEY-structure holding a private key assigned to it. After this the CTX-structure can be used to perform encryptions with EVP_PKEY_encrypt(). However, as mentioned, asymmetric encryption is a slow operation, and thus it is preferable to use the EVP "envelope routines" instead. These routines create a symmetric key and an IV, and then encrypts them with asymmetric encryption, after which this key can be used to encrypt the data. [4]. These envelope operations start by calling *EVP_SealInit()*, which takes an EVP_CIPHER_CTX-structure and an EVP_CIPHER-structure (specifying the symmetric

algorithm to be used) as parameters, along with the EVP_PKEY object holding the private key as well as some requisite buffers to store the resulting asymmetrically encrypted symmetric keys. After this, the EVP_CIPHER_CTX can be used to perform the symmetric encryption with *EVP_SealUpdate()* and *EVP_SealFinal(),* both of which function exactly the same as *EVP_EncryptUpdate()* and *EVP_EncryptFinal().* [4]

Digital signatures can be performed with EVP_DigestSign. This begins with calling the *EVP_DigestSignInit()*-function with an EVP_MD_CTX, an EVP_PKEY_CTX, an EVP_MD (specifying the digest algorithm to be used) and an EVP_PKEY holding the private key as parameters, with an optional ENGINE-pointer. After this, the input data can be hashed with *EVP_DigestSignUpdate(),* storing the hash in the EVP_MD_CTX. Finally, the hash can be signed with *EVP_DigestSignFinal()*, which places the resulting signature into a buffer. Alternatively, these two steps can be done with *EVP_DigestSign(). [4]*

EVP also supports key derivation, which allows deriving a shared secret with a peer. An EVP_PKEY_CTX is initialised for this operation with *EVP_PKEY_derive_init()*. Then, a peer key is set with *EVP_PKEY_derive_set_peer()*, with an EVP_PKEY holding the peer's public key as a parameter. After this, the shared secret can be derived with *EVP_PKEY_derive()*. [4]

5. CONCLUSION

This paper analysed the usability of some of the open-source cryptographic toolkits and more precisely, their APIs. The APIs evaluated were NaCl, Libsodium, Nettle and OpenSSL's EVP, with the main focus of the paper being on EVP. Usability is an important aspect of any API, as an API with usability problems will lead to incorrect use. This can lead, and has led to, security problems affecting millions of end-users world-wide. Thus, it is clear that API designers need to make sure that the API is easy to use correctly. The usability of these APIs was evaluated using Jacob Nielsen's heuristic analysis guidelines for user interfaces, which were mapped to evaluate API usability by Brad A. Myers and Jeffrey Stylos in "Improving API Usability", with further consideration taken in this paper due to the cryptographic nature of the APIs evaluated. This paper found no glaring issues with any of the APIs, but where possible, did offer some perspective on what could possibly be done better, or at least where an API goes against one of the heuristic analysis guidelines. The four APIs were also compared to each other to examine their similarities and differences. One key similarity between them is their usage of sane defaults, an important aspect of usability. An interesting contradiction in the usability of security APIs is that sometimes achieving the required security comes at the cost of usability. Finding this balance where an API is secure yet can be used without errors caused by lacking usability is very important.

This paper also took a closer look into the usage of EVP. EVP provides a high-level abstraction layer to the "core" operations available in OpenSSL: message digests, symmetric encryption and asymmetric encryption. This paper provides insight into how all of these operations are performed and found that the method and context-structures used by EVP resemble objects in object-oriented programming. Another interesting point is that performing public key authenticated encryption relies heavily on not only the public key API of EVP, but also the message digest and symmetric encryption APIs as well.

SOURCES

[1] Brad A. Myers & Jeffrey Stylos. Improving API Usability. Communications of the ACM. 2016 [Online].

[2] OpenSSL Software Foundation. [Online]. Available at: https://www.openssl.org/

[3] Nicola Tuveri & Billy B. Brumley. Start your ENGINEs: dynamically loadable contemporary crypto. 2019 IEEE Cybersecurity Development (SecDev). 2019. [Online]. Available at: https://eprint.iacr.org/2018/354

[4] OpenSSL Software Foundation. Documentation. [Online]. Available at: https://www.openssl.org/docs/man1.1.1/man3/evp.html

[5] Jacob Nielsen. Usability Engineering. Elsevier Science & Technology. 1994. 91-93

[6] Nielsen Norman Group. [Online]. Available at: https://www.nngroup.com/articles/ten-usability-heuristics

[7] Daniel J. Bernstein, Tanja Lange, Peter Schwabe. The security impact of a new cryptographic library. Lecture notes in Computer Science 7533. Springer. 2012. [Online]. Available at: https://cr.yp.to/highspeed/coolnacl-20120725.pdf

[8] NaCl: Networking and Cryptography library. [Online]. Available at: https://nacl.cr.yp.to/index.html

[9] Libsodium. Libsodium documentation. [Online]. Available at: https://libsodium.gitbook.io/doc/

[10] Nettle – a low level cryptographic library. Documentation. [Online]. Available at: https://www.lysator.liu.se/~nisse/nettle/nettle.html

[11] Peter Leo Gorski, Luigi Lo Iacono, Dominik Wemke, Christian Stransky, Sebastian Möller, Yasemin Acar, Sascha Fahl. Developers Deserve Security Warnings, Too: On the Effect of Integrated Security Advice on Cryptographic API Misuse. In *Fourteenth Symposium on Usable Privacy and Security. SOUPS 2018.*

Baltimore, MD. USA. Mary Ellen Zurko and Heather Richter Lipford (Eds.). USENIX Association. *August 12-14, 2018.* [Online]. Available at: https://www.usenix.org/system/files/conference/soups2018/soups2018-gorski.pdf

[12] R. Housley. Guidelines for Cryptographic Algorithm Agility and Selecting Mandatory-to-Implement Algorithms. Internet Requests for Comments. RFC 7696. November 2015. [Online]. Available at: https://tools.ietf.org/html/rfc7696

[13] D. McGrew. An Interface and Algorithms for Authenticated Encryption. Internet Requests for Comments. RFC 5116. January 2008. [Online]. Available at: https://tools.ietf.org/html/rfc5116

[14] Daniel J. Bernstein. Cryptography in NaCl. 2009. [Online]. Available at: https://cr.yp.to/highspeed/naclcrypto-20090310.pdf