Password-based OpenSSL Encryption

Analysis of Key Derivation Protocol

Rita Ainane, Austin Garrett, Cory Johnson, Alexis Vivar

{rainane, agarret7, corymj, alexisv}@mit.edu

May 17, 2018

Contents

1 Introduction

OpenSSL is an opensource toolkit for SSL/TLS cryptographic protocols, used by a majority of the web in some form or another. OpenSSL provides myriad different tools, including native C code libraries, and command line tools for direct encryption of files. As a result of its extensive offerings, it has achieved become ubiquitous for developers intent on establishing secure communication over the Internet.

However, there exist potential issues in specific parts of the OpenSSL project that raise questions about the security of data when using certain features. In particular, secure keys generated using OpenSSL's password based key derivation function (PBKDF) have several questionable properties which potentially jeapordize the security of the procedure. This project analyzes the security of this private key generation protocol, and investigates the impact on the integrity of systems which rely on the security of the project. Finally, we provide recommendations to users to mitigate potential vulnerabilities present in the OpenSSL toolkit.

2 Analyze OpenSSL PBKDF

2.1 OpenSSL Use Cases

The OpenSSL library provides basic cryptographic functionality and allows users to use encryption algorithms and parameters via a simple command-line interface. The supported cipher algorithms are as follows [6]:

- Ciphers: AES, Blowfish, Camellia, SEED, CAST-128, DES, IDEA, RC2, RC4, RC5, Triple, DES, GOST 28147-89
- Cryptographic hash functions: MD5, MD4, MD2, SHA-1, SHA-2, RIPEMD-160, MDC-2, GOST R 34.11-94, BLAKE2, Whirlpool
- Public-key cryptography: RSA, DSA, Diffie-Hellman key exchange, Elliptic curve, GOST R 34.10-2001

An example use case of OpenSSL is as follows: Alice would like to send sensitive information to Bob, but Bob is not a part of Alice's network. This information should not leave Alice's network unsecured due to the risk of an eavesdropper obtaining Alice's sensitive information. Alice would like to ensure the confidentiality of her data as well as guarantee that only a designated party - Bob - can decrypt her data.

OpenSSL provides Alice with the tools to generate keys, encrypt her data, implement public key cryptography and more in order to secure her sensitive information. If Alice is running a Unix-like environment, then OpenSSL is already installed. The process for Alice to secure her communications is as follows [6]:

- **On Bob's (receiver) end:** Bob uses OpenSSL to generate a public and private key pair. The private key should be kept protected by Bob. Bob now communicates the public key to the environment (which includes Alice). This key does not need to be protected. Below is example code for generation of an RSA key pair and extraction of the public key from a file [6]:

  1. Generate and RSA public/private key pair:
     `openssl genrsa -aes256 -out user 2048`
  2. Extract the public key from the file
     `openssl rsa -in user -pubout > user.pub`

- **On Alice's (sender) end:** Alice uses `user.pub public key` to store Bobs public key on her computer. She then uses OpenSSL to generate a secret key for symmetric encryption and using `user.pub public key` she stores the secret key in a file and encrypts it asymmetrically. Below is example code for this procedure [6]:

  1. Generate a random secret key
     `openssl rand 48 -base64 -out password_file`
  2. Encrypt the password file using user.pub public key
     `openssl rsautl -in password_file -out password_encrypted -pubin -inkey user.pub -encrypt`
  3. Encrypt the traffic using a symmetric algorithm (i.e. ae256) with the secret key
     `openssl enc -e -in client.tgz -out client.tgz.enc -aes256 -kfile password_file`

  client.tgz should be protected, and sent alongside the password_encrypted file to Bob. This way, the communicated information is protected.

- **On Bob's end:** Bob can decrypt the secret key file using his private key. He can then use the secret key from the file to decrypt Alice's sensitive communications. Below is example code for this procedure [6]:

  1. Decrypt the secret key file
     `openssl rsautl -decrypt -inkey user -in password_encrypted -out password_file_decrypted`
  2. Decrypt Alice's sensitive information
     `openssl enc -d -in client.tgz.enc -out client.tgz -aes256 -kfile password_file_decrypted`

2.2 OpenSSL encryption

OpenSSL provides a convenient feature to encrypt and decrypt files via the command-line using the command enc. The command allows for password based encryption via various block and stream ciphers, with the current standard being AES256. Via the below command,

a user can specify a ciphername along with various options for the file encryption [8]:

```
$ openssl enc -ciphername [options]
```

The below command allows users to search the available encryption algorithms supported by enc. This command will output a list with the various ciphers, their key sizes as well as modes of operation:

```
$ openssl list-cipher-algorithms
```

Users may choose an option from the outpu of this command to replace the -ciphername option in the enc command. The users may also specify certain optional command for the encryption. As a note, OpenSSL recommends that the -salt parameter always be used for PBKDF since it adds randomness in the key geberation. If that parameter is not enabled, then equal passwords would result in the same encryption keys. This results in an unnecessary adversarial advantage in decrypting the key. Below is a list of options that can be specified:

-in filename : specify the input file

-out filename : specify the output file (created if non-existant, otherwise overwritten)

-e or -d : specify whether to encrypt (-e) or decrypt (-d). The default is -e.

-a, -A, -base64 : specify whether to apply Base64-encoding before or after operation. -a and -base64 are equivalent. Adding the -A flag suppresses line breaks that occur every 64 characters

-bufsize n : specify the buffer size for internal buffers.

-debug : enable debugging output.

-engine id : specify an engine (i.e. to use special hardware)

-iv IV : specify initialization vector as a hex number. Default is to derive initialization vector from the password

-k password, -kfile filename : specify the password or a file containing the password $\rightarrow$ deprecated, and -pass arg should be used instead.

-K key : specify the key used for encryption or decryption. Default is to derive key from password.

-md messagedigest : specify the message digest used for key derivation from md2, Md5, sha, or sha1

-nopad : disables standard padding

-salt, -nosalt, -S salt : switch salting on or off. -S salt allows user to explicitly specify the salt value.

-p, -P : prints the key, initialization vector and salt value. If -P, then these values will be printed and encryption will not occur

-pass arg : specify password or password source. arg can be the password itself, or a filename where the password is stored

-z : enables zlib-compression, which compresses a file via zlib after it is encrypted (and if specified, base64 encoded). When decrypting, zlib will be applied first.

A command that would encrypt a file A.txt to B.enc with the password 6857rocks would look like this:

```
openssl enc -aes-256-cbc -salt -in A.txt -out B.enc -pass pass:6857rocks
```

2.3 Updates to OpenSSL PBKDF

For more than twenty years, RSA Laboratories has published public key cryptography standards, which are revised and released as cryptography research advances. Current stable releases of OpenSSL use a password-based key derivation function compliant with RSA Laboratories' PBKDF2 standard.

2.3.1 OpenSSL 1.0.2 and PBKDF1

Prior to the introduction of OpenSSL 1.1.0, OpenSSL's key derivation process was based on a modified implementation of PBKDF1 from Password-Based Cryptography Standard 5 (PKCS5) - RFC2898 as codified by the Internet Engineering Task Force (IETF).
The function **EVP_BytesToKey** derives a key and initialization vector (IV) as follows:

```
#include <openssl/evp.h>
int EVP_BytesToKey(const EVP_CIPHER *type, const EVP_MD *md,
                   const unsigned char *salt,
                   const unsigned char *data, int datal, int count,
                   unsigned char *key, unsigned char *iv);
```

In 1.0.2, available hashes include MD2, MD5, and SHA1. Iteration count is accepted as a parameter, and a `salt` command is required. `EVP_BytesToKey` concatenates the password

and salt, and hashes the result. Depending on the value specified for `count`, the result will then be hashed again, and the subsequent result hashed again, until count is exhausted. The default count is one (1) for the then-default MD5 (and now-default SHA256), meaning a brute-force attack has only a single layer of hashing to penetrate the given password.

2.3.2 OpenSSL 1.1.0 and PBKDF2

After 1.1.0, OpenSSL was modified to be compliant with the PBKDF2 is updated to include the SHA256 digest. Furthermore the generate

2.4 OpenSSL potential problems

2.4.1 Technical

The encryption tool used by OpenSSL appears to be secure, but implementation and human factors has proven this not to be true in many cases. Prior to OpenSSL 1.1.0 if a messaged digest was not specified, OpenSSL would default the cryptographic hash function to a salted version of MD5 [2]. While MD5 possesses key qualities of a secure cryptographic hash function - like one-wayness and determinism - it has been proven to be non-collision resistant.

MD5 is a hash function that - once given an input - yields an output very quickly. Thus, an adversary can hash a large number of passwords quickly [4]. Since MD5 has been proven to be non collision resistant, it is possible for an adversary to find a collision from two of the hashed passwords. If an adversary is in possession of the hash of a users file, the adversary can also potentially find another password with the same hash via this attack. This renders the users MD5 hashed information insecure, since an adversary can execute such an attack in polynomial time.

MD5 is no longer considered a secure hash function, and before version 1.1.0, OpenSSL only used one iteration of the salted MD5 algorithm - making a brute force attacks on MD5 hashed information even more feasible for an adversary [2]. When implementing a cryptographic hash function, it is usually standard to execute many rounds of the hash. This means that given an input to a hash function and its output, the output is then treated as the new input of yet another iteration of the hash function. This pattern is repeated many times for the purpose of key stretching. Key stretching increases entropy for the encoded password, in order to increase the time that an adversary would need to break the hash via brute force attack [7]. Enc did not allow for multiple iterations of the hash, resulting in much weaker password encryptions.

In version 1.1.0, OpenSSL switched their default digest from MD5 to SHA-256. SHA-256 is currently believed to be a more secure cryptographic hash function than MD5 and therefore adversarial attacks on SHA-256 hashed information should be harder to execute than those on MD5 [3]. Although the default digest changed in version 1.1.0, the syntax used to encrypt did not. All options remain the same, OpenSSL still supports enc -ciphername [options] including all of previous options mentioned in section 2.2.

OpenSSL version 1.0.1 will introduce yet another cryptographic hashing function to improve encryption security - SHA-3. This is pivotal for OpenSSL encryption because both SHA256 and MD5 are members of the Merkle–Damgård family of hash functions. While SHA256 is considered to be more secure than MD5, it may be possible to eventually expose weaknesses in SHA256 using the make similar mathematical assumptions that were used to show non-collision resistance in MD5. SHA-3 on the other hand is a sponge function and thus not a member of the the Merkle–Damgård family of hash function. SHA3's hardness relies on different mathematical assumptions than the previous cryptographic hashing functions and thus is believed to be more computationally difficult to break, as well as not susceptible to attacks that would be performed on sha2 or md5 [5].

2.4.2 Human Error

Cryptographic encryption schemes are constantly being broken, and OpenSSL provides updates to its library to include new encryption algorithms and new parameters in order to account for this. This to allow users to add as many layers of protection they deem fit. Nonetheless, due to human error, encryptions may be much weaker than desired resulting in a false sense of security. Below are some examples on how human error can cause a significantly weaker encryption scheme than desired:

If a user fails to update their version of OpenSSL to the latest version, their encryptions may not be as secure as they may hope. As previously mentioned, the default cryptographic hashing algorithm was changed from MD5 to SHA-256. If a user were to fail to update their version of OpenSSL to 1.1.0, and were still using the default encryption algorithm from the previous version, there encryptions will not be as secure as they expected. This applies to every update to the default hashing function. Once vulnerabilities are uncovered in SHA-256, OpenSSL will once again update its default cryptographic hashing algorithm and thus users should be aware of such changes.

Along with keeping up to date with latest versions of OpenSSL, users should also ensure that they update all other relevant files as well. When using the enc command, the user may specify the file in which the information is to be stored. If a user is using the default hashing algorithm and updates to a newer version of OpenSSL with a different default digest and does not update their files, this may lead to inconsistencies in decryption of the saved files.

Updating OpenSSL may result in new available encryption schemes and parameters. For example, prior to version 1.1.0 the available cryptographic hashing function could not perform key stretching and were thus limited to one iteration no matter the function. However, in the newer versions of OpenSSL, options have been added specifying the following [8]:

> **-iter count** : Use a given number of iterations on the password in deriving the encryption key. High values increase the time required to brute-force the resulting file. This option enables the use of PBKDF2 algorithm to derive the key.
>
> **-pbkdf2** : Use PBKDF2 algorithm with default iteration count unless otherwise specified.

These new parameters greatly improve the security of the encryption scheme and should be used whenever implementing a cryptographic hash function in OpenSSL. However, because the format of the enc command has not changed, users may be used to the old version syntax and not be aware of the availability of these options.

3 Brute Force Attack

As stated before, MD5 is the default hash digest of version 1.0.2 OpenSSL, which is currently in long time support. Many services using this version may be vulnerable to brute-force attacks, which because of the single iteration, becomes incredibly quick. Given a basic unsalted encryption scheme, cracking the password is trivial for low entropy passwords:

```
openssl enc -aes-256-cbc -nosalt -pass pass:X -p -in data.txt -out data.enc
```

Running on an NVIDIA GeForce 8800 Ultra, the MD5 hashing algorithm can be iterated over 200 million times per second. [1] For an unsalted password with 32 bits of entropy, this corresponds to a password that can be cracked in less than a minute. This attack was performed 9 years ago, and since then GPU's have become exponentially stronger. Even with the 8-byte salt appended to the hash, with a low entropy password, it is apparent that brute-force attacks could soon become feasible on encryptions performed with the the OpenSSL `enc` tool.

4 Recommendations

OpenSSL, when used properly, is be a great tool for securing communications over computer networks. It provides up to date security functions as well as a significant amount of options

to strengthen these functions. However when used improperly, a user may believe that their information is more secure than it truly is. Below is a list of suggestions for users of OpenSSL to help maximize the security of their encrypted information.

4.1 Version of OpenSSL

As mentioned in section 2.3.2, newer versions of OpenSSL often come with updates to the cryptographic hash functions available for users, as well as new parameters that strengthen the security of those functions. By having the most up to date version of OpenSSL a user has access to a larger number of measures for strengthened security. SHA3 has recently been offered as an extended digest, which offers fixes to many of the issues present in the MD5 and SHA256 hashing functions.

4.2 *Enc* Parameters and Algorithms

As mentioned, MD5 preceded SHA-256 as the default digest for the enc command. Additionally, by default OpenSSL performs only one iteration of these cryptographic hashing functions. But users have the option to specify parameters beyond just the default. Users can specify certain parameters to greatly strengthen the security of their encryption.

As discussed in the section on Brute Forcing, salts are absolutely necessary to protect against trivial attacks against the security of password-based schemes. The single iteration provides a serious concern, and so to mitigate the potential vulnerabilities, entropy must be maximized to slow down brute force attacks. Along these lines, choosing high entropy passwords could significantly extend the amount of time required to crack the password.

Furthermore, a soon-to-be-released update to OpenSSL will offer SHA3, a more secure cryptographic hashing function than SHA-256 and MD5. It may benefit users to choose this as the PBKDF digest as opposed to SHA256. Additionally, the option -iter could be changed to allow for key stretching in order to increase the time an adversary needs to decrypt the original message. These tools help to restore the insecurity inherent in the default configurations for the OpenSSL command line tool. In general, it is good practice to understand the changes that come with updates to OpenSSL in order for users to understand how they can utilize these updates to increase the security of their encryptions.

References

[1] Md5 crack using gpu.

[2] Cryptosense. Weak key derivation in openssl. https://cryptosense.com/blog/weak-key-derivation-in-openssl/.

[3] Sandeep Kumar and Er Piyush Gupta. A comparative analysis of sha and md5 algorithm. 5:4492 – 4495, 06 2014.

[4] Patrick on. Storing passwords securely. https://patrickmn.com/security/storing-passwords-securely/.

[5] Stack Overflow. Differentiating sha-3 and previous sha algorithms. https://crypto.stackexchange.com/questions/32457/what-is-the-difference-between-sha-3keccak-and-previous-generation-sha-algorit.

[6] Stack Overflow. Protecting sensitive information using openssl. http://ad-technica.com/protecting-sensitive-information-using-openssl/.

[7] Roger Smith. Security 106: The importance of key stretching. http://www.efficientsoftware.co.nz/importance-key-stretching/.

[8] OpenSSL Wiki. Enc. https://wiki.openssl.org/index.php/Enc.