

An Introduction to OpenSSL Part One

Holt Sorenson 2001-08-22

An Introduction to OpenSSL, Part One: Cryptographic Functions

by Holt Sorenson

last updated August 22, 2001

This is the first article in a four-part series on OpenSSL, a library written in the C programming language that provides routines for cryptographic primitives utilized in implementing the Secure Sockets Layer (SSL) protocol. OpenSSL also includes routines for implementing the SSL protocol itself, and it includes an application called `openssl` that provides a command line interface to both sets of routines. This articles introduce some cryptographic tools that the SSL protocol has borrowed from cryptographer's bags of tricks to accomplish its design goals. While readers who are already familiar with cryptographic concepts may be familiar with the content presented in this installment, the following "Brief Overview of Cryptographic Primitives" section should nevertheless be interesting reading and will certainly set up the rest of this discussion.

Brief Overview of Cryptographic Primitives

This section will begin by first introducing symmetric [ciphers](#). Next, asymmetric ciphers will be examined. Following that, hash functions and Message Authentication Codes (MACs) will be explored. Lastly, cryptographic randomness will be explained with an ardent rant/plea to be zealous and vigilant about properly providing cryptographic randomness for cryptosystems.

Symmetric (Single key) ciphers

Symmetric, single [key](#), or secret key ciphers utilize an [algorithm](#), whose inputs are a key and some [plaintext](#). The resultant output is [ciphertext](#). When one wishes to recover the plaintext, one feeds the ciphertext and the key into the algorithm. Such an algorithm has the property that if an attacker knows the ciphertext and the algorithm, neither the plaintext nor the key can be recovered. This makes the key a critical piece of information that should be protected from would be attackers. There are two commonly-used families of symmetric ciphers. One family is called *stream ciphers*, as they are mathematically designed in such a way as to encipher a constant stream of information. The other family is called *block ciphers* because these ciphers operate on blocks of data. One can learn more about both types of ciphers by consulting [\[RSAFAQ2\]](#). A common block cipher implemented by OpenSSL is Triple DES (3DES) [\[3DESCite\]](#)

that is based on the Data Encryption Standard (DES). Another is Blowfish [BFcite]. Both 3DES and Blowfish will be used in the examples later in the article. 3DES's progenitor, DES, is now obsolesced by the relentless march of technology. Its successor is called the Advanced Encryption Standard [AESCite]. The nascent AES has yet to be included in OpenSSL, however. An example of a common stream cipher implemented by OpenSSL is RC4 [RC4cite].

Asymmetric (Dual key) ciphers

Asymmetric, dual key, or public key ciphers are a bit more complex in their design and implementation. Alice and Bob are two persons who only have a public channel over which to communicate. They wish the content of their messages to be known only by each other. This requirement keeps Alice and Bob from transmitting a secret key to one another across the public channel, because if eavesdropping Eve was monitoring the channel she could use the key she learned from Bob and Alice's transmission to decrypt all transmissions enciphered with that key.

Bob has information for his friend, Alice, that he wishes no one else to know. He also wishes to receive an acknowledgement of the message that he sends. His requirement for this acknowledgment is that it have the property that only Alice could create the acknowledgment. Such a property allows Bob to authenticate that Alice created the acknowledgement message. Both Bob and Alice know that their nosey adversary, Eve, is eavesdropping on their communication. They decide utilize asymmetric encryption because of their security requirements for this transaction.

Bob and Alice agree upon an asymmetric encryption algorithm and Alice creates a key pair that contains two keys. One is known by both Bob and Eve because Alice sends the key to Bob, and Eve eavesdrops on the transmission. This key is called the public key. The other key, known only to Alice, is called the private key.

Bob feeds the Alice's public key and plaintext to the asymmetric algorithm and the resultant output is ciphertext. Bob sends the ciphertext to Alice. Alice recovers the plaintext by feeding the ciphertext and the private key to the algorithm. The algorithm is termed asymmetric because there are two keys involved; there is the public key, used to encrypt the plaintext, and private key, used to decrypt the ciphertext. The algorithm has the property that if Eve knows the ciphertext, the algorithm, and the public key, neither the plaintext nor the public key's complement, the private key, can be recovered. This makes the private key a critical piece of

information that should be protected from Eve and her cohorts.

Alice then crafts her acknowledgment message and feeds the asymmetric algorithm the private key and the acknowledgement message. The result of the operation is the signature. The signature and the message are published. Bob who has the public key corresponding to the private key held by the Alice, can verify the signature with Alice's public key. Since only Alice has the private key, only Alice could have performed the initial signing operation that yielded the published signature that is linked to the message by the private key and the algorithm. Although Eve knows the algorithm, the public key, the message, and its signature, Eve can't recover Alice's private key. Bob is assured that Alice really received the message, because only she could craft the acknowledgement that he verified she created. A common asymmetric cipher is RSA (named for its creators, Rivest, Shamir, and Adleman) [\[RSACite\]](#). A common asymmetric signature scheme is DSA (Digital Signature Algorithm) [\[DSSCite\]](#). The Diffie-Hellman algorithm is often used in conjunction with DSS when a use of DSS also needs asymmetric encryption [\[DHCite\]](#).

Whether using a symmetric or asymmetric cipher, care should be taken to use keys of a sufficient length that will protect your transaction for the amount of time that its data is sensitive. In general, the longer you wish to protect the data the longer the key should be. [\[KeyLengthCite\]](#) discusses this in greater detail. Also, the scenario explained above only protects against passive attackers such as Eve. A malicious attacker, Mallory, could actively attack Alice and Bob's transmissions by posing as Alice to Bob, and as Bob to Alice. Such an attack is called a man-in-the-middle attack and more steps such as out-of-band verification are needed in a protocol to thwart such attacks.

Hash functions and Message Authentication Codes (MAC) are two cryptographic mechanisms that allow parties to verify the integrity of data that has been transmitted or has been stored for a period of time.

Hash Functions

A cryptographic hash, also known as a message digest or a modification detection code, is an algorithm that, when fed a set of data, computes a unique identifier. This identifier cannot be replicated by feeding any other data to the algorithm except that particular set of data. This property is called *collision-resistant*. Also, if an attacker knows the algorithm and the unique identifier, the attacker cannot recover the set of data by feeding the algorithm the unique

identifier. This property is called *one-way*. One example of a hash function is the Secure Hash Algorithm (SHA) [SHACite]. Another hash function that is often used is Message Digest 5 (MD5) [MD5Cite]. SHA is being updated to be usable with the new block sizes of AES [SHAupdate]. MD5 is faster than SHA, but a recent paper has shown that it is not as collision resistant as originally conjectured [Dobbertin1996].

Message Authentication Code (MAC)

A message authentication code, or MAC, feeds a key and plaintext to an algorithm to create the MAC. Both stream and block ciphers can be used as MACs [RSAfaq4]. Hash functions can also be used as MACs in conjunction with a shared secret, or key. Since MACs use keys as part of their algorithm, one should take the same care with these keys as one would take with keys used in any other cryptographic operation.

An example: Alice and Bob's brother, Bobcheck, have decided they want to exchange data. They want assurance that that the data hasn't changed while in transit. They aren't worried about the data remaining confidential. Bobcheck and Alice have previously agreed on a shared secret, and have agreed to use the HMAC [Krawczyk1997] function.

Alice takes the key and message and runs them through the HMAC algorithm. She transmits the message and the HMAC result to Bobcheck. Bobcheck runs the key and the message through the HMAC algorithm. He verifies that the result of the HMAC is the same as that which Alice transmitted. Since it is, he knows the message has traversed the hostile environment without modification. Bobcheck can trust that Alice was the person that sent the message since she is the only other person in possession of the shared secret.

Cryptographic Randomness

One of the easiest places to break a cryptographic system is to use values for keys that aren't cryptographically random. [Eastlake1994] more colorfully articulates this idea by stating:

"For the present, the lack of generally available facilities for generating such unpredictable numbers is an open wound in the design of cryptographic software... the only safe strategy so far has been to force the local installation to supply a suitable routine to generate random numbers. To say the least, this is an awkward, error-prone and unpalatable solution".

A cryptographically "random number is one that the adversary has to guess, [and for which] there is no strategy for determining [the number] that is better than brute force" [\[Callas1996\]](#).

When random numbers are needed, one should utilize a source of randomness that, if identically replicated by an attacker, would not generate the same numbers for your application as it does for the attacker. Sources of randomness are said to be truly random if the attacker has infinite computing resources and the difficulty of computing the random numbers that you are generating remains intractable. Sources of randomness are said to be pseudo random if they are only safe against an attacker with limited computational resources [\[Ellison1995\]](#).

Unfortunately many people implementing cryptography have failed to grasp the concepts elucidated above.

Recently a minor change was made to OpenSSL to check the random value seeding the OpenSSL pseudo random number generator. This resulted in persons asking the OpenSSL mailing list how to skirt the check so that they could utilize OpenSSL without providing an unguessable seed. They received recommendations such as:

- use a constant text string
- use a DSA public key
- seed the OpenSSL random number generator with bytes generated by `rand(3)`
- read `/etc/passwd`
- read the OpenSSL executable off the disk
- hash files in the current directory
- use a dummy file to trick the check

The worst advice given was to rip the check code out of the OpenSSL library [\[Guttman2000\]](#). This would cause the prng to be seeded with nothing, resulting in consistently predictable output. The query has been put forth enough by users of OpenSSL that it is now a FAQ item [\[OpenSSLFAQRNG\]](#). One last example of poor suggestions on seeding a PRNG is to "use several files as random seed enhancers which will help to make the key more secure. Text files that have been compressed with a utility such as gzip are good choices." [\[Slacksite\]](#). Random Seed Enhancers ? Bollocks. gzip has a regular file format that allows manipulation of the file. This format is predicatable. It is also trivial for an attacker to create the same seed by gzipping the same files that you have. How many files are on your system ? I found a little over 45,000 on the system on which I am writing this article. 45,000 is a small enough number that a

determined, if bored, attacker can gzip each file and try the result as a seed to the PRNG that generated your SSL enabled server's private key. Even using multiple files with this method to seed OpenSSL's prng is silly given the number of off the shelf solutions to generating cryptographically random seeds referenced below.

I imploringly beseech that you not subjugate yourself to such putrescent refuse! Such statements are great for fertilizing farmland occupying hundreds of acres, but don't cut the proverbial mustard as adequate recommendations for how to seed a PRNG. Remember that PRNGs are just that, pseudo. Assisting PRNGS in having a proper start is a human problem because PRNGs don't know better. Innumerable are the triage group therapy sessions cryptographers have been forced to convene because they had to console traumatized PRNGs that had been abused by being fed predicable input [Goldberg1996]. Have you ever seen a PRNG get it's stomach pumped ? It's not pretty. If the algorithms used by gzip really were good enough for this purpose then said algorithms would alone be used as PRNGs.

Some possible user land remedies, in the form of software packages, that are thought to be acceptable for generating randomness suitable for cryptographic use on most Unices are the Entropy Gathering Daemon (EGD) [egd] and the Pseudo Random Number Generator Daemon (PRNGD) [prngd]. Kernel-based devices are available in Linux [LinuxRandom], Solaris [MaierAndi] or [SUNWski], and OpenBSD [OpenBSDRandom]. FreeBSD offers a /dev/random. Intel Pentium III CPUs or newer come with a hardware based random number generator [IntelRNG]. Linux 2.2.18 and newer makes it accessible as /dev/intel_rng [Garzik2000] with properly configured kernels.

Please, please, ***PLEASE*** go to the trouble of making sure that you are using sources of randomness that consensus has deemed worthy of generating key material. If you use broken randomness I can guarantee that your crypto will be broken no matter how well designed the algorithm and protocol is, or what the length of your keys are. If you are going to the trouble of using crypto, why compromise the system for the attacker ? Securing systems is about securing systems, not pretending to secure systems..

As I've brushed over some cryptographic primitives above, I've made many statements which imply that the science of cryptography is absolute and that breaking ciphers is impossible. This is not the case. Many aspects of cryptography are only conjectured, not proven, to work. Cryptography utilizes complex algorithms which use BFNs (really big numbers) to maintain the integrity of data, to keep it confidential, or to authenticate what key was used to sign a

message for a LFT (a really long time). It's also important to catch the nuance of the last phrase in the previous sentence. Keys are what are used in cryptographic operations, not people or computers. You can't really tell if a person encrypted something because humans use computers to perform the operations on their behalf. To make matters worse, keys are used on systems that you can't trust, systems which are notorious for having their security (if the designers even attempted to include security) obliterated regularly. So the reality is that we presume a key was used to perform an operation; We presume that a given operation was performed; We presume that a person was somehow involved and that this key is theirs.

Continuing to delve into all the details of this quandary is outside of the scope of this article, but a recent publication [[Schneier2000](#)], explicates these issues in easy to understand terms. It's even probable that your pointy-haired-boss can be somewhat security literate after reading it. I will resist saying something like there is a limit that your PHB would approach in trying to understand security/cryptography because we have been discussing concepts that are built on math and anyone who takes Calculus 101 has dealt with limits and knows how inexcusably gruesome a pun that would be.

Holt Sorenson works for [Counterpane Internet Security](#) where he wrangles tuples of bits so his colleagues can get their work done. He's always serious and never jokes or laughs. When he is not surgically attached to computers he likes to hang out with his family and engage in frivolous pursuits that purportedly keep him out of trouble [[Iterata1999](#)].

To read **An Introduction to OpenSSL, Part Two: Cryptographic Functions Continued**, click [here](#).

Relevant Links

"Using and Creating Cryptographic-Quality Random Numbers"

Jon Callas

"Cryptanalysis of MD5 Compress"

Hans Dobbertin

"RFC 1750: Randomness Recommendations for Security"

D. Eastlake

"Cryptographic Random Numbers"

Carl Ellison

"Hardware driver for Intel i810 Random Number Generator (RNG)"

Jeff Garzik and Philipp Rumpf

"Randomness and the Netscape Browser"

Ian Goldberg and David Wagner

"Software Generation of Practically Strong Random Numbers"

Peter Guttman

[Privacy Statement](#)

Copyright 2006, SecurityFocus