

An Introduction to OpenSSL, Part Two: Cryptographic Functions Continued

Holt Sorenson 2001-09-05

An Introduction to OpenSSL, Part Two: Cryptographic Functions Continued

by Holt Sorenson

last updated September 5, 2001

This is the second article in a 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. In the [first article](#) in the series, we discussed some of the basics of cryptography. This article will cover acquiring and compiling OpenSSL and explore some commands that facilitate encryption and decryption.

Acquiring and Compiling OpenSSL

If you are using a recent Debian package or RPM based Linux distribution, you can usually install OpenSSL without compiling it. Debian users can probably type `apt-get -f update` followed by `apt-get -f install openssl` as root to install OpenSSL. Of course, this assumes that the package source is defined in the `/etc/apt/sources.list` file. Users of RPM-based distributions can use `rpmfind` or browse to <http://www.rpmfind.net>, and then use `rpm -ivh` to install the RPM. If these instructions don't help or don't represent proper procedures for your Linux system, you will need to consult the documentation that came with your distribution.

Solaris users can browse to <http://www.sunfreeware.com> and download OpenSSL if it is available for their architecture (`uname -p`) and version (`uname -r`). After downloading it, use `pkgadd(1M)` to install the package.

Readers who do not have the luxury of installing OpenSSL without compiling it, should download the OpenSSL source from openssl.org. At the time of publication of this article, the most recent version of OpenSSL was 0.9.6b. Using an earlier version is not recommended. If users have an earlier version, upgrading is **highly** recommended, as there were some security bugs in earlier versions that have since been fixed [[OSSUpdate](#)]. Readers who have a CryptoSwift, Compaq Atalla, or nCipher CHIL crypto device can download the engine source to experiment with the device. Note that the engine software is said to be at "a pretty experimental status and [have] almost no documentation" [[ENGREADME](#)].

Un-gzip and de-tar the source and change directory into the newly created OpenSSL source tree. One possible command to accomplish this would be:

```
gzip -dc openssl-0.9.6b.tar.gz | tar xvf -
```

If you are compiling for a production system use:

```
./config --prefix= --openssldir= -DSSL_FORBID_ENULL
```

For the values and , I prefer to use `/usr/local/openssl-`. This way I can have multiple OpenSSL versions around, with the flavor of the week symlinked to `openssl`. Note that if readers use this scheme, they will likely have to use `-I` and `-L` to show the compiler where to find OpenSSL when compiling and linking OpenSSL into other applications.

If having multiple versions available is not a priority, readers can use:

```
./config --prefix=/usr/local --openssldir=/usr/local/openssl
```

If OpenSSL is being compiled for a development system in which SSL will be debugged at the protocol level, omitting the command `-DSSL_FORBID_ENULL` is acceptable. `-DSSL_FORBID_ENULL` causes OpenSSL to omit null ciphers in the SSL cipher suite. Null ciphers permit cleartext (unencrypted information) to traverse the wire. Null ciphers provide no confidentiality and aren't encouraged for use on production systems.

Note that by default, shared objects are not created. If they are desired, users will have to append the `'shared'` argument to the above config command.

At this point, users should now run `make`. When that is completed, `make test` should be run. If `make test` fails, try reconfiguring with `no-asm`, `no-threads`, or `386`, depending on what failed and what architecture is being used. If you have more troubles, consult the files `INSTALL` or `FAQ` that come with the distribution, or browse to [OpenSSL's support page](#).

Now run `make install` and the OpenSSL binaries and supporting files will be installed in the default directories or the directories specified with the above config command.

Finally, add the OpenSSL binary directory to your path and you are good to go.

If readers are lazy like me and like to have the OpenSSL commands available from the command line without having to prefix them with `openssl`, they can change directory to the OpenSSL binary directory and use one of the following kludges to create symlinks:

```
./openssl -? 2>&1|perl -ne '/^$|[A-Z]/|push(@a,split());END{foreach(@a){do
{symlink"openssl",$_;}unless-l$_}}'
```

Readers who don't have perl can use:

```
for i in `./openssl -? 2>&1|egrep -v '^$|[A-Z]`;do [ -h ${i} ] || ln -s openssl ${i};done
```

Note that there will be a symlink called `passwd` which generates `crypt(3)` passwords. Users should change its name to something else so that it doesn't collide with the system `passwd` binary in the user's path, delete the link, or place the OpenSSL directory after `/bin` or `/usr/bin` in the path. If the binary directory for OpenSSL is owned by a user other than you, this operation could require privileges.

OpenSSL can be run in command line mode or interactive mode. To begin interactive mode, run `openssl` at the prompt and then type `help`. If readers prefer the command line interface to interactive mode, they can type `openssl help` and the prompt will return. The commands that are then displayed are all the commands that OpenSSL supports. Readers can find information about the commands by typing the command followed by the `'-?'` option. `openssl` will then output the usage of that command. Readers can also consult the man pages placed in newly installed OpenSSL directory hierarchy.

If these operations were not being conducted as experiments, following reasonable security measures such as setting `umask` to a conservative value like `077` is necessary. Additionally, one should follow best security practices to prevent compromise, as cryptographic methods are only one link in the proverbial security chain.

OpenSSL Commands

Congratulations, you have successfully installed OpenSSL! Now, let's explore some of the commands that are now available.

Go to a temporary directory and type:

```
man gcc > gcc_man_temp;man gcc >> gcc_man_temp;man gcc >> gcc_man_temp
```

First we will experiment with a couple of OpenSSL's hash function commands so that we can use their output to verify cryptographic operations later. If the reader has used the symlink kludges above, it is unnecessary to preface the commands below with `openssl`. If readers are so inclined, it may be illustrative to run each of the cryptographic operations below using the `time` command. This will help to show the different amounts of time that different cryptographic primitives take. It is also useful to compare the performance of different algorithms that come from the same family of cryptographic primitives, and to compare the performance of different operations with the same algorithm. If the reader is using a newer version of Solaris, `/usr/proc/bin/ptime` will time with greater resolution.

Type `openssl md5 gcc_man_temp`. On the right hand side of the equals sign (=) in the output the reader should see a string of hexadecimal numbers. This is the hash of the `gcc_man_temp` file.

Repeat these steps for the `openssl sha1` command.

Save the hashes for future reference.

Symmetric Encryption

Now we will explore symmetric encryption. First we need a couple of keys. Since we are testing, we'll use `/dev/urandom`. When encrypting data that really needs to be safe, using `/dev/random` is generally a better bet than `/dev/urandom`; this is because the latter produces randomness that is lower quality as it is generated by a PRNG once it runs out of the higher quality randomness that can also be fetch from `/dev/random` [Callas1996]. It is also possible that because of performance constraints or concern about Denial of Service, the users of the cryptography application would want to use `/dev/urandom` even though it is not as secure as `/dev/random`. Sticking to `/dev/random` is certainly the most conservative stance one can take, unless the user has a hardware-based random device.

We can use the following commands:

```
dd bs=16 count=1 if=/dev/urandom of=bf_key
dd bs=21 count=1 if=/dev/urandom of=3des_key
```

If you do not have a random device installed consult the [randomness](#) section of the previous article and then use the `egc` command that comes with [\[egd\]](#) as follows:

```
egc.pl read 16|perl -ne '{s/(.*:\s)?([a-zA-F\d]{2})+?/$2 /g;foreach $e(split(/\s/))
{print chr($e)}}' > bf_key
```

```
egc.pl read 21|perl -ne '{s/(.*:\s)?([a-zA-F\d]{2})+?/$2 /g;foreach $(split(/\s/))
{print chr($e)}}' > 3des_key
```

These keys will be used for blowfish and triple-des, respectively.

Now encrypt the temporary file with blowfish:

```
openssl bf -e -a -kfile bf_key -in gcc_man_temp -out gcc_man_temp_bf_enc
```

In order to verify that the data is now in ciphertext form, readers can use their favorite file viewer to view `gcc_man_temp_bf_enc`.

The data can then be decrypted with blowfish:

```
openssl bf -d -a -kfile bf_key -in gcc_man_temp_bf_enc -out gcc_man_temp_bf_dec
```

Now we run `openssl sha1` and `openssl md5` on `gcc_man_temp_bf_dec` and if the hashes match then we have successfully encrypted and decrypted using blowfish and the experimental key.

Now repeat the above steps with `openssl des3`.

Assymmetric Cryptography Using OpenSSL

Now that we have covered symmetric cryptography, let's take a brief look at asymmetric cryptography with OpenSSL's implementation of RSA.

First we need a seed for OpenSSL's pseudo-random number generator. We can acquire this with the following command:

```
dd bs=256 count=3 if=/dev/urandom of=random_seed
```

If the reader's OS lacks a random device, he or she can run the following until `random_seed` contains more than 4096 bits:

```
egc.pl read 255|perl -ne '{s/(.*:\s)?([a-zA-F\d]{2})+?/$2 /g;foreach $(split(/\s/))
{print chr($e)}}' >> random_seed
```

This should take about three tries with `egc`. Readers can check to see how much is in the `random_seed` file by using:

```
echo `wc -c < random_seed` ` " * 8" |bc
```

Next we need to generate our RSA keypair using:

```
openssl genrsa -out rsa_key_private -rand random_seed 4096
```

If we were not testing it is to include the parameter `-des3` or `-idea` so that the private key is encrypted with a symmetric cipher before being stored on disk.

Next, extract the public key into a separate file using the command:

```
openssl rsa -pubout -in rsa_key_private -out rsa_key_public
```

When using asymmetric key pairs on production systems, allowing access to the public key is necessary. However access to the private key should be tightly controlled and available only to the entities that need to utilize it.

We can examine the numbers that constitute the keys by using:

```
openssl rsa -text -in rsa_key_private or openssl rsa -text -pubin -in rsa_key_public
```

Now we need some material to use for encryption and signing tests. The material should be less than 4096 bits (512 bytes), as we are using a 4096 bit key. The mathematics of asymmetric ciphers do not permit one to encrypt a set of data that is larger than the keysize of the asymmetric keypair. The next few examples will use the result of:

```
man wc | head -15 > wc_man_temp
```

We can encrypt to the private key with the command:

```
openssl rsautl -pubin -inkey rsa_key_public -encrypt -in wc_man_temp -out wc_man_temp_rsa_enc
```

Decrypting is then accomplished with:

```
openssl rsautl -inkey rsa_key_private -decrypt -in wc_man_temp_rsa_enc -out wc_man_temp_rsa_dec
```

Now verify that the decryption operation was successful by using a hash function command on `wc_man_temp_rsa_dec` and `wc_man_temp` (or the appropriate files).

If readers have been using `time` (or equivalent) as they execute the above commands, it is interesting to notice that the public key cryptographic operations take longer than the symmetric key operations and that RSA decryption takes at least an order of magnitude longer (at least on the author's machine, anyway.)

Readers should now use RSA to sign and verify the signature of some data. Generate the signature with:

```
openssl rsautl -inkey rsa_key_private -sign -in wc_man_temp -out wc_man_temp_rsa_sign
```

We can then verify the signature with:

```
openssl rsautl -pubin -inkey rsa_key_public -verify -in wc_man_temp_rsa_sign -out wc_man_temp_rsa_verify
```

Next, we can use a hash function to verify that `wc_man_temp_rsa_verify` and `wc_man_temp` are the same.

Readers who used `time` will see that the RSA signing operation took about the same amount of time that the RSA decryption operation did. That is because when users sign with RSA, they sign plaintext with the private key instead of with the public key so that the recipients of the message can verify with the public key.

That concludes the second article of our discussion of OpenSSL and wraps up the overview of cryptographic primitives. In the next installment of this series we will begin our discussion of OpenSSL and PKI.

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.

To read **An Introduction to OpenSSL, Part Three: PKI - Public Key Infrastructure**, click [here](#).

Relevant Links

[An Introduction to OpenSSL, Part One: Cryptographic Functions](#)

Holt Sorenson

[OpenSSL FAQ](#)

OpenSSL

[Privacy Statement](#)

Copyright 2006, SecurityFocus