

GNU/Linux Administration - Support #307

OpenSSL Command-Line Howto

01/15/2014 09:29 PM - Daniel Curtis

Status:	Closed	Start date:	01/15/2014
Priority:	Normal	Due date:	
Assignee:	Daniel Curtis	% Done:	100%
Category:		Estimated time:	1.00 hour
Target version:		Spent time:	1.00 hour

Description

The openssl application that ships with the OpenSSL libraries can perform a wide range of crypto operations. This HOWTO provides some cookbook-style recipes for using it.

Introduction

The openssl command-line binary that ships with the OpenSSL libraries can perform a wide range of cryptographic operations. It can come in handy in scripts or for accomplishing one-time command-line tasks.

Documentation for using the openssl application is somewhat scattered, however, so this article aims to provide some practical examples of its use. I assume that you've already got a functional OpenSSL installation and that the openssl binary is in your shell's PATH.

Just to be clear, this article is strictly practical; it does not concern cryptographic theory and concepts. If you don't know what an MD5 sum is, this article won't enlighten you one bit—but if all you need to know is how to use openssl to generate a file sum, you're in luck.

The nature of this article is that I'll be adding new examples incrementally. Check back at a later date if I haven't gotten to the information you need.

How do I find out what OpenSSL version I'm running?

Use the version option:

```
openssl version
```

```
OpenSSL 0.9.8b 04 May 2006
```

You can get much more information with the version -a option.

```
openssl version -a
```

```
OpenSSL 0.9.8b 04 May 2006
built on: Fri Sep 29 18:45:58 UTC 2006
platform: debian-i386-i686/cmov
options: bn(64,32) md2(int) rc4(idx,int) des(ptr,risc1,16,long) blowfish(idx)
compiler: gcc -fPIC -DOPENSSL_PIC -DZLIB -DOPENSSL_THREADS -D_REENTRANT
-DDSO_DLFCN -DHAVE_DLFCN_H -DL_ENDIAN -DTERMIO -O3 -march=i686
-Wa,-noexecstack -g -Wall -DOPENSSL_BN_ASM_PART_WORDS -DOPENSSL_IA32_SSE2
-DSHA1_ASM -DMD5_ASM -DRMD160_ASM -DAES_ASM
OPENSSLDIR: "/usr/lib/ssl"
```

How do I get a list of the available commands?

There are three built-in options for getting lists of available commands, but none of them provide what I consider useful output. The best thing to do is provide an invalid command (help or -h will do nicely) to get a readable answer.

```
openssl help
```

openssl:Error: 'help' is an invalid command.

Standard commands

```
asn1parse  ca          ciphers  crl       crl2pkcs7
dgst       dh          dhparam  dsa       dsaparam
ec         ecparam  enc      engine   errstr
gendh     gendsa   genrsa   nseq     ocspl
passwd    pkcs12   pkcs7    pkcs8    prime
rand      req      rsa      rsautl   s_client
s_server  s_time   sess_id  smime    speed
spkac     verify   version  x509
```

Message Digest commands (see the `dgst' command for more details)

```
md2      md4      md5      rmd160   sha
sha1
```

Cipher commands (see the `enc' command for more details)

```
aes-128-cbc aes-128-ecb aes-192-cbc aes-192-ecb aes-256-cbc
aes-256-ecb base64      bf          bf-cbc      bf-cfb
bf-ecb      bf-ofb     cast       cast-cbc    cast5-cbc
cast5-cfb  cast5-ecb cast5-ofb  des         des-cbc
des-cfb    des-ecb   des-ede   des-ede-cbc des-ede-cfb
des-ede-ofb des-ede3  des-ede3-cbc des-ede3-cfb des-ede3-ofb
des-ofb    des3     desx     rc2        rc2-40-cbc
rc2-64-cbc rc2-cbc  rc2-cfb  rc2-ecb   rc2-ofb
rc4        rc4-40
```

What the shell calls “Standard commands” are the main top-level options.

You can use the same trick with any of the subcommands.

```
openssl dgst -h
```

```
unknown option '-h'
options are
-c          to output the digest with separating colons
-d          to output debug info
-hex       output as hex dump
-binary    output in binary form
-sign file sign digest using private key in file
-verify file verify a signature using public key in file
-prverify file verify a signature using private key in file
-keyform arg key file format (PEM or ENGINE)
-signature file signature to verify
-binary    output in binary form
-engine e   use engine e, possibly a hardware device.
-md5 to use the md5 message digest algorithm (default)
-md4 to use the md4 message digest algorithm
-md2 to use the md2 message digest algorithm
-sha1 to use the sha1 message digest algorithm
-sha to use the sha message digest algorithm
-sha256 to use the sha256 message digest algorithm
-sha512 to use the sha512 message digest algorithm
-mdc2 to use the mdc2 message digest algorithm
-ripemd160 to use the ripemd160 message digest algorithm
```

In more boring fashion, you can consult the OpenSSL man pages.

How do I get a list of available ciphers?

Use the ciphers option. The ciphers(1) man page is quite helpful.

- list all available ciphers

```
openssl ciphers -v
```

- list only TLSv1 ciphers

```
openssl ciphers -v -tls1
```

- list only high encryption ciphers (keys larger than 128 bits)

```
openssl ciphers -v 'HIGH'
```

- list only high encryption ciphers using the AES algorithm

```
openssl ciphers -v 'AES+HIGH'
```

Benchmarking

How do I benchmark my system's performance?

The OpenSSL developers have built a benchmarking suite directly into the openssl binary. It's accessible via the speed option. It tests how many operations it can perform in a given time, rather than how long it takes to perform a given number of operations. This strikes me a quite sane, because the benchmarks don't take significantly longer to run on a slow system than on a fast one.

To run a catchall benchmark, run it without any further options.

```
openssl speed
```

There are two sets of results. The first reports how many bytes per second can be processed for each algorithm, the second the times needed for sign/verify cycles. Here are the results on an 2.16GHz Intel Core 2.

The 'numbers' are in 1000s of bytes per second processed.

type	16 bytes	64 bytes	256 bytes	1024 bytes	8192 bytes
md2	1736.10k	3726.08k	5165.04k	5692.28k	5917.35k
mdc2	0.00	0.00	0.00	0.00	0.00
md4	18799.87k	65848.23k	187776.43k	352258.73k	474622.63k
md5	16807.01k	58256.45k	160439.13k	287183.53k	375220.91k
hmac(md5)	23601.24k	74405.08k	189993.05k	309777.75k	379431.59k
sha1	16774.59k	55500.39k	142628.69k	233247.74k	288382.98k
rmd160	13854.71k	40271.23k	87613.95k	124333.06k	141781.67k
rc4	227935.60k	253366.06k	261236.94k	259858.09k	194928.50k
des cbc	48478.10k	49616.16k	49765.21k	50106.71k	50034.01k
des ede3	18387.39k	18631.02k	18699.26k	18738.18k	18718.72k
idea cbc	0.00	0.00	0.00	0.00	0.00
rc2 cbc	19247.24k	19838.12k	19904.51k	19925.33k	19834.98k
rc5-32/12 cbc	0.00	0.00	0.00	0.00	0.00
blowfish cbc	79577.50k	83067.03k	84676.78k	84850.01k	85063.00k
cast cbc	45362.14k	48343.34k	49007.36k	49202.52k	49225.73k
aes-128 cbc	58751.94k	94443.86k	111424.09k	116704.26k	117997.57k
aes-192 cbc	53451.79k	82076.22k	94609.83k	98496.85k	99150.51k
aes-256 cbc	49225.21k	72779.84k	82266.88k	85054.81k	85762.05k
sha256	9359.24k	22510.83k	40963.75k	51710.29k	56014.17k
sha512	7026.78k	28121.32k	54330.79k	86190.76k	104270.51k
sign verify	sign/s	verify/s			
rsa 512 bits	0.000522s	0.000042s	1915.8	23969.9	
rsa 1024 bits	0.002321s	0.000109s	430.8	9191.1	
rsa 2048 bits	0.012883s	0.000329s	77.6	3039.6	
rsa 4096 bits	0.079055s	0.001074s	12.6	931.3	
sign verify	sign/s	verify/s			
dsa 512 bits	0.000380s	0.000472s	2629.3	2117.9	

```
dsa 1024 bits 0.001031s 0.001240s 969.6 806.2
dsa 2048 bits 0.003175s 0.003744s 314.9 267.1
```

You can run any of the algorithm-specific subtests directly.

- test rsa speeds

```
openssl speed rsa
```

- do the same test on a two-way SMP system

```
openssl speed rsa -multi 2
```

How do I benchmark remote connections?

The `s_time` option lets you test connection performance. The most simple invocation will run for 30 seconds, use any cipher, and use SSL handshaking to determine number of connections per second, using both new and reused sessions:

```
openssl s_time -connect remote.host:443
```

Beyond that most simple invocation, `s_time` gives you a wide variety of testing options.

- retrieve remote `test.html` page using only new sessions

```
openssl s_time -connect remote.host:443 -www /test.html -new
```

- similar, using only SSL v3 and high encryption (see `ciphers(1)` man page for cipher strings)

```
openssl s_time \  
-connect remote.host:443 -www /test.html -new \  
-ssl3 -cipher HIGH
```

- compare relative performance of various ciphers in 10-second tests

```
IFS=":"  
for c in $(openssl ciphers -ssl3 RSA); do  
  echo $c  
  openssl s_time -connect remote.host:443 \  
    -www / -new -time 10 -cipher $c 2>&1 | \  
    grep bytes  
  echo  
done
```

If you don't have an SSL-enabled web server available for your use, you can emulate one using the `s_server` option.

- on one host, set up the server (using default port 4433)

```
openssl s_server -cert mycert.pem -www
```

- on second host (or even the same one), run `s_time`

```
openssl s_time -connect myhost:4433 -www / -new -ssl3
```

Certificates

How do I generate a self-signed certificate?

You'll first need to decide whether or not you want to encrypt your key. Doing so means that the key is protected by a passphrase.

On the plus side, adding a passphrase to a key makes it more secure, so the key is less likely to be useful to someone who steals it. The downside, however, is that you'll have to either store the passphrase in a file or type it manually every time you want to start your web or ldap server.

It violates my normally paranoid nature to say it, but I prefer unencrypted keys, so I don't have to manually type a passphrase each time a secure daemon is started. (It's not terribly difficult to decrypt your key if you later tire of typing a passphrase.)

This example will produce a file called mycert.pem which will contain both the private key and the public certificate based on it. The certificate will be valid for 365 days, and the key (thanks to the -nodes option) is unencrypted:

```
openssl req \  
-x509 -nodes -days 365 \  
-newkey rsa:1024 -keyout mycert.pem -out mycert.pem
```

Using this command-line invocation, you'll have to answer a lot of questions: Country Name, State, City, and so on. The tricky question is Common Name. You'll want to answer with the hostname or CNAME by which people will address the server. **This is very important**. If your web server's real hostname is mybox.mydomain.com but people will be using www.mydomain.com to address the box, then use the latter name to answer the "Common Name" question.

Once you're comfortable with the answers you provide to those questions, you can script the whole thing by adding the -subj option. I've included some information about location into the example that follows, but the only thing you really need to include for the certificate to be useful is the hostname (CN):

```
openssl req \  
-x509 -nodes -days 365 \  
-subj '/C=US/ST=Oregon/L=Portland/CN=www.madboa.com' \  
-newkey rsa:1024 -keyout mycert.pem -out mycert.pem
```

How do I generate a certificate request for VeriSign?

Applying for a certificate signed by a recognized certificate authority like VeriSign is a complex bureaucratic process. You've got to perform all the requisite paperwork before creating a certificate request.

As in the recipe for creating a self-signed certificate, you'll have to decide whether or not you want a passphrase on your private key. The recipe below assumes you don't. You'll end up with two files: a new private key called mykey.pem and a certificate request called myreq.pem:

```
openssl req \  
-new -newkey rsa:1024 -nodes \  
-keyout mykey.pem -out myreq.pem
```

If you've already got a key and would like to use it for generating the request, the syntax is a bit simpler.

```
openssl req -new -key mykey.pem -out myreq.pem
```

Similarly, you can also provide subject information on the command line.

```
openssl req \  
-new -newkey rsa:1024 -nodes \  
-subj '/CN=www.mydom.com/O=My Dom, Inc./C=US/ST=Oregon/L=Portland' \  
-keyout mykey.pem -out myreq.pem
```

When dealing with an institution like VeriSign, you need to take special care to make sure that the information you provide during the creation of the certificate request is exactly correct. I know from personal experience that even a difference as trivial as substituting

“and” for “&” in the Organization Name will stall the process.

If you'd like, you can double check the signature and information provided in the certificate request.

- verify signature

```
openssl req -in myreq.pem -noout -verify -key mykey.pem
```

- check info

```
openssl req -in myreq.pem -noout -text
```

Save the key file in a secure location. You'll need it in order to use the certificate VeriSign sends you. The certificate request will typically be pasted into VeriSign's online application form.

How do I test a new certificate?

The `s_server` option provides a simple but effective testing method. The example below assumes you've combined your key and certificate into one file called `mycert.pem`.

First, launch the test server on the machine on which the certificate will be used. By default, the server will listen on port 4433; you can alter that using the `-accept` option.

```
openssl s_server -cert mycert.pem -www
```

If the server launches without complaint, then chances are good that the certificate is ready for production use.

You can also point your web browser at the test server, e.g., <https://yourserver:4433/>. Don't forget to specify the “https” protocol; plain-old “http” won't work. You should see a page listing the various ciphers available and some statistics about your connection. Most modern browsers allow you to examine the certificate as well.

How do I retrieve a remote certificate?

If you combine `openssl` and `sed`, you can retrieve remote certificates via a shell one-liner or a simple script.

```
#!/bin/sh
#
# usage: retrieve-cert.sh remote.host.name [port]
#
REMHOST=$1
REMPORT=${2:-443}

echo |\
openssl s_client -connect ${REMHOST}:${REMPORT} 2>&1 |\
sed -ne '/-BEGIN CERTIFICATE-/,/-END CERTIFICATE-/p'
```

You can, in turn, pipe that information back to `openssl` to do things like check the dates on all your active certificates.

```
#!/bin/sh
#
for CERT in \
  www.yourdomain.com:443 \
  ldap.yourdomain.com:636 \
  imap.yourdomain.com:993
do
  echo |\
  openssl s_client -connect ${CERT} 2>/dev/null |\
  sed -ne '/-BEGIN CERTIFICATE-/,/-END CERTIFICATE-/p' |\
  openssl x509 -noout -subject -dates
done
```

How do I extract information from a certificate?

An SSL certificate contains a wide range of information: issuer, valid dates, subject, and some hardcore crypto stuff. The x509 subcommand is the entry point for retrieving this information. The examples below all assume that the certificate you want to examine is stored in a file named cert.pem.

Using the -text option will give you the full breadth of information.

```
openssl x509 -text -in cert.pem
```

Other options will provide more targeted sets of data:

- who issued the cert?

```
openssl x509 -noout -in cert.pem -issuer
```

- to whom was it issued?

```
openssl x509 -noout -in cert.pem -subject
```

- for what dates is it valid?

```
openssl x509 -noout -in cert.pem -dates
```

- the above, all at once

```
openssl x509 -noout -in cert.pem -issuer -subject -dates
```

- what is its hash value?

```
openssl x509 -noout -in cert.pem -hash
```

- what is its MD5 fingerprint?

```
openssl x509 -noout -in cert.pem -fingerprint
```

How do I export or import a PKCS#12 certificate?

PKCS#12 files can be imported and exported by a number of applications, including Microsoft IIS. They are often associated with the file extension .pfx.

To create a PKCS#12 certificate, you'll need a private key and a certificate. During the conversion process, you'll be given an opportunity to put an "Export Password" (which can be empty, if you choose) on the certificate.

- create a file containing key and self-signed certificate

```
openssl req \  
-x509 -nodes -days 365 \  
-newkey rsa:1024 -keyout mycert.pem -out mycert.pem
```

- export mycert.pem as PKCS#12 file, mycert.pfx

```
openssl pkcs12 -export \  
-out mycert.pfx -in mycert.pem \  
-name "My Certificate"
```

If someone sends you a PKCS#12 and any passwords needed to work with it, you can export it into standard PEM format.

- export certificate and passphrase-less key

```
openssl pkcs12 -in mycert.pfx -out mycert.pem -nodes
```

- same as above, but you'll be prompted for a passphrase for the private key

```
openssl pkcs12 -in mycert.pfx -out mycert.pem
```

Certificate Verification

Applications linked against the OpenSSL libraries can verify certificates signed by a recognized certificate authority (CA).

How do I verify a certificate?

Use the verify option to verify certificates

```
openssl verify cert.pem
```

If your local OpenSSL installation recognizes the certificate or its signing authority and everything else (dates, signing chain, etc.) checks out, you'll get a simple OK message.

```
openssl verify remote.site.pem
```

```
remote.site.pem: OK
```

If anything is amiss, you'll see some error messages with short descriptions of the problem:

```
error 10 at 0 depth lookup:certificate has expired. Certificates are typically issued for a limited period of time—usually just one year—and openssl will complain if a certificate has expired.
```

```
error 18 at 0 depth lookup:self signed certificate. Unless you make an exception, OpenSSL won't verify a self-signed certificate.
```

What certificate authorities does OpenSSL recognize?

When OpenSSL was built for your system, it was configured with a "Directory for OpenSSL files." (That's the `--openssldir` option passed to the configure script, for you hands-on types.) This is the directory that typically holds information about certificate authorities your system trusts.

The default location for this directory is `/usr/local/ssl`, but most vendors put it elsewhere:

- `/usr/share/ssl` - **Red Hat/Fedora**
- `/etc/ssl` - **Gentoo**
- `/usr/lib/ssl` - **Debian**
- `/System/Library/OpenSSL` - **Macintosh OS X**

Use the version option to identify which directory (labeled `OPENSSLDIR`) your installation uses

```
openssl version -d
```


Within that directory and a subdirectory called certs, you're likely to find one or more of three different kinds of files.

1. A large file called cert.pem, an omnibus collection of many certificates from recognized certificate authorities like VeriSign and Thawte.
2. Some small files in the certs subdirectory named with a .pem file extension, each of which contains a certificate from a single CA.
3. Some symlinks in the certs subdirectory with obscure filenames like 052eae11.0. There is typically one of these links for each .pem file.

The first part of obscure filename is actually a hash value based on the certificate within the .pem file to which it points. The file extension is just an iterator, since it's theoretically possible that multiple certificates can generate identical hashes.

On my Gentoo system, for example, there's a symlink named f73e89fd.0 that points to a file named vsignss.pem. Sure enough, the certificate in that file generates a hash the equates to the name of the symlink:

```
openssl x509 -noout -hash -in vsignss.pem

f73e89fd
```

When an application encounters a remote certificate, it will typically check to see if the cert can be found in cert.pem or, if not, in a file named after the certificate's hash value. If found, the certificate is considered verified.

It's interesting to note that some applications, like Sendmail, allow you to specify at runtime the location of the certificates you trust, while others, like Pine, do not.

How do I get OpenSSL to recognize/verify a certificate?

Put the file that contains the certificate you'd like to trust into the certs directory discussed above. Then create the hash-based symlink. Here's a little script that'll do just that.

```
#!/bin/sh
#
# usage: certlink.sh filename [filename ...]

for CERTFILE in $*; do
  # make sure file exists and is a valid cert
  test -f "$CERTFILE" || continue
  HASH=$(openssl x509 -noout -hash -in "$CERTFILE")
  test -n "$HASH" || continue

  # use lowest available iterator for symlink
  for ITER in 0 1 2 3 4 5 6 7 8 9; do
    test -f "${HASH}.${ITER}" && continue
    ln -s "$CERTFILE" "${HASH}.${ITER}"
    test -L "${HASH}.${ITER}" && break
  done
done
```

Command-line clients and servers

The `s_client` and `s_server` options provide a way to launch SSL-enabled command-line clients and servers. There are other examples of their use scattered around this document, but this section is dedicated solely to them.

In this section, I assume you are familiar with the specific protocols at issue: SMTP, HTTP, etc. Explaining them is out of the scope of this article.

How do I connect to a secure SMTP server?

You can test, or even use, an SSL-enabled SMTP server from the command line using the `s_client` option.

Secure SMTP servers offer secure connections on up to three ports: 25 (TLS), 465 (SSL), and 587 (TLS). Some time around the 0.9.7 release, the `openssl` binary was given the ability to use STARTTLS when talking to SMTP servers.

- port 25/TLS; use same syntax for port 587

```
openssl s_client -connect remote.host:25 -starttls smtp
```

- port 465/SSL

```
openssl s_client -connect remote.host:465
```

RFC821 suggests (although it falls short of explicitly specifying) the two characters "<CRLF>" as line-terminator. Most mail agents do not care about this and accept either "<LF>" or "<CRLF>" as line-terminators, but Qmail does not. If you want to comply to the letter with RFC821 and/or communicate with Qmail, use also the `-crflf` option:

```
openssl s_client -connect remote.host:25 -crflf -starttls smtp
```

How do I connect to a secure [whatever] server?

Connecting to a different type of SSL-enabled server is essentially the same operation as outlined above. As of the date of this writing, openssl only supports command-line TLS with SMTP servers, so you have to use straightforward SSL connections with any other protocol.

- https: HTTP over SSL

```
openssl s_client -connect remote.host:443
```

- ldaps: LDAP over SSL

```
openssl s_client -connect remote.host:636
```

- imaps: IMAP over SSL

```
openssl s_client -connect remote.host:993
```

- pop3s: POP-3 over SSL

```
openssl s_client -connect remote.host:995
```

How do I set up an SSL server from the command line?

The `s_server` option allows you to set up an SSL-enabled server from the command line, but it's I wouldn't recommend using it for anything other than testing or debugging. If you need a production-quality wrapper around an otherwise insecure server, check out Stunnel instead.

The `s_server` option works best when you have a certificate; it's fairly limited without one.

- the `-www` option will sent back an HTML-formatted status page to any HTTP clients that request a page

```
openssl s_server -cert mycert.pem -www
```

- the `-WWW` option "emulates a simple web server. Pages will be resolved relative to the current directory." This example is listening on the https port, rather than the default port 4433

```
openssl s_server -accept 443 -cert mycert.pem -WWW
```

Digests

Generating digests with the `dgst` option is one of the more straightforward tasks you can accomplish with the `openssl` binary. Producing digests is done so often, as a matter of fact, that you can find special-use binaries for doing the same thing. How do I create an MD5 or SHA1 digest of a file?

Digests are created using the `dgst` option.

- MD5 digest

```
openssl dgst -md5 filename
```

- SHA1 digest

```
openssl dgst -sha1 filename
```

The MD5 digests are identical to those created with the widely available `md5sum` command, though the output formats differ.

```
openssl dgst -md5 foo-2.23.tar.gz
```

```
MD5= 81eda7985e99d28acd6d286aa0e13e07
```

```
md5sum foo-2.23.tar.gz
```

```
81eda7985e99d28acd6d286aa0e13e07 foo-2.23.tar.gz
```

The same is true for SHA1 digests and the output of the `sha1sum` application.

```
openssl dgst -sha1 foo-2.23.tar.gz
```

```
SHA1= e4eabc78894e2c204d788521812497e021f45c08
```

```
sha1sum foo-2.23.tar.gz
```

```
e4eabc78894e2c204d788521812497e021f45c08 foo-2.23.tar.gz
```

How do I sign a digest?

If you want to ensure that the digest you create doesn't get modified without your permission, you can sign it using your private key. The following example assumes that you want to sign the SHA1 sum of a file called `foo-1.23.tar.gz`.

- signed digest will be `foo-1.23.tar.gz.sha1`

```
openssl dgst -sha1 \  
-sign mykey.pem \  
-out foo-1.23.tar.gz.sha1 \  
foo-1.23.tar.gz
```

How do I verify a signed digest?

To verify a signed digest you'll need the file from which the digest was derived, the signed digest, and the signer's public key.

- to verify `foo-1.23.tar.gz` using `foo-1.23.tar.gz.sha1` and `pubkey.pem`

```
openssl dgst -sha1 \  
-verify pubkey.pem \  
-signature foo-1.23.tar.gz.sha1 \  
foo-1.23.tar.gz
```

How do I create an Apache digest password entry?

Apache's HTTP digest authentication feature requires a special password format. Apache ships with the `htdigest` utility, but it will only write to a file, not to standard output. When working with remote users, it's sometimes nice for them to be able to generate a password hash on a machine they trust and then mail it for inclusion in your local password database.

The format of the password database is relatively simple: a colon-separated list of the username, authorization realm (specified by the Apache `AuthName` directive), and an MD5 digest of those two items and the password. Below is a script that duplicates the output of `htdigest`, except that the output is written to standard output. It takes advantage of the `dgst` option's ability to read from standard input.

```
#!/bin/bash  
  
echo "Create an Apache-friendly Digest Password Entry"  
echo "-----"  
  
# get user input, disabling tty echoing for password  
read -p "Enter username: " UNAME  
read -p "Enter Apache AuthName: " AUTHNAME  
read -s -p "Enter password: " PWORD; echo  
  
printf "\n%s:%s:%s\n" \  
"$UNAME" \  
"$AUTHNAME" \  
$(printf "${UNAME}:${AUTHNAME}:${PWORD}" | openssl dgst -md5)
```

What other kinds of digests are available?

Use the built-in `list-message-digest-commands` option to get a list of the digest types available to your local OpenSSL installation:

```
openssl list-message-digest-commands
```

Encryption/Decryption

How do I base64-encode something?

Use the `enc -base64` option.

- send encoded contents of `file.txt` to stdout

```
openssl enc -base64 -in file.txt
```

- same, but write contents to `file.txt.enc`

```
openssl enc -base64 -in file.txt -out file.txt.enc
```

It's also possible to do a quick command-line encoding of a string value:

```
echo "encode me" | openssl enc -base64  
  
ZW5jb2RIIG1lCg==
```

Note that echo will silently attach a newline character to your string. Consider using its `-n` option if you want to avoid that situation, which could be important if you're trying to encode a password or authentication string:

```
echo -n "encode me" | openssl enc -base64
```

```
ZW5jb2RIIG11
```

Use the `-d` (decode) option to reverse the process:

```
echo "ZW5jb2RlIG11Cg==" | openssl enc -base64 -d
```

```
encode me
```

How do I simply encrypt a file?

Simple file encryption is probably better done using a tool like GPG. Still, you may have occasion to want to encrypt a file without having to build or use a key/certificate structure. All you want to have to remember is a password. It can nearly be that simple—if you can also remember the cipher you employed for encryption.

To choose a cipher, consult the `enc(1)` man page. More simply (and perhaps more accurately), you can ask `openssl` for a list in one of two ways.

see the list under the 'Cipher commands' heading

```
openssl -h
```

- or get a long list, one cipher per line

```
openssl list-cipher-commands
```

After you choose a cipher, you'll also have to decide if you want to base64-encode the data. Doing so will mean the encrypted data can be, say, pasted into an email message. Otherwise, the output will be a binary file.

- encrypt `file.txt` to `file.enc` using 256-bit AES in CBC mode

```
openssl enc -aes-256-cbc -salt -in file.txt -out file.enc
```

- the same, only the output is base64 encoded for, e.g., e-mail

```
openssl enc -aes-256-cbc -a -salt -in file.txt -out file.enc
```

To decrypt `file.enc` you or the file's recipient will need to remember the cipher and the passphrase.

- decrypt binary `file.enc`

```
openssl enc -d -aes-256-cbc -in file.enc
```

- decrypt base64-encoded version

```
openssl enc -d -aes-256-cbc -a -in file.enc
```

If you'd like to avoid typing a passphrase every time you encrypt or decrypt a file, the openssl(1) man page provides the details under the heading "PASS PHRASE ARGUMENTS." The format of the password argument is fairly simple.

- provide password on command line

```
openssl enc -aes-256-cbc -salt -in file.txt \  
-out file.enc -pass pass:mySillyPassword
```

- provide password in a file

```
openssl enc -aes-256-cbc -salt -in file.txt \  
-out file.enc -pass file:/path/to/secret/password.txt
```

Errors

How do I interpret SSL error messages?

Poking through your system logs, you see some error messages that are evidently related to OpenSSL or crypto:

```
sshd31784: error: RSA_public_decrypt failed: error:0407006A:lib(4):func(112):reason(106)  
sshd770: error: RSA_public_decrypt failed: error:0407006A:lib(4):func(112):reason(106)
```

The first step to figure out what's going wrong is to use the `errstr` option to interpret the error code. The code number is found between "error:" and "lib". In this case, it's 0407006A.

```
openssl errstr 0407006A
```

```
error:0407006A:rsa routines:RSA_padding_check_PKCS1_type_1:block type is not 01
```

If you've got a full OpenSSL installation, including all the development documentation, you can start your investigation there. In this example, the `RSA_padding_add_PKCS1_type_1(3)` man page will inform you that PKCS #1 involves block methods for signatures. After that, of course, you'd need to pore through your application's source code to identify when it would expect be receiving those sorts of packets.

Keys

How do I generate an RSA key?

Use the `genrsa` option.

- default 512-bit key, sent to standard output

```
openssl genrsa
```

- 1024-bit key, saved to file named `mykey.pem`

```
openssl genrsa -out mykey.pem 1024
```

- same as above, but encrypted with a passphrase

```
openssl genrsa -des3 -out mykey.pem 1024
```

How do I generate a public RSA key?

Use the `rsa` option to produce a public version of your private RSA key.

```
openssl rsa -in mykey.pem -pubout
```

How do I generate a DSA key?

Building DSA keys requires a parameter file, and DSA verify operations are slower than their RSA counterparts, so they aren't as widely used as RSA keys.

If you're only going to build a single DSA key, you can do so in just one step using the `dsaparam` subcommand.

- key will be called `dsakey.pem`

```
openssl dsaparam -noout -out dsakey.pem -genkey 1024
```

If, on the other hand, you'll be creating several DSA keys, you'll probably want to build a shared parameter file before generating the keys. It can take a while to build the parameters, but once built, key generation is done quickly.

- create parameters in `dsaparam.pem`

```
openssl dsaparam -out dsaparam.pem 1024
```

- create first key

```
openssl gendsa -out key1.pem dsaparam.pem
```

- and second ...

```
openssl gendsa -out key2.pem dsaparam.pem
```

How do I create an elliptic curve key?

Routines for working with elliptic curve cryptography were added to OpenSSL in version 0.9.8. Generating an EC key involves the `eccparam` option.

```
openssl eparam -out key.pem -name prime256v1 -genkey
```

- openssl can provide full list of EC parameter names suitable for passing to the `-name` option above:

```
openssl eparam -list_curves
```

How do I remove a passphrase from a key?

Perhaps you've grown tired of typing your passphrase every time your secure daemon starts. You can decrypt your key, removing the passphrase requirement, using the `rsa` or `dsa` option, depending on the signature algorithm you chose when creating your private key.

If you created an RSA key and it is stored in a standalone file called `key.pem`, then here's how to output a decrypted version of the same key to a file called `newkey.pem`.

- you'll be prompted for your passphrase one last time

```
openssl rsa -in key.pem -out newkey.pem
```

Often, you'll have your private key and public certificate stored in the same file. If they are stored in a file called mycert.pem, you can construct a decrypted version called newcert.pem in two steps.

- you'll need to type your passphrase once more

```
openssl rsa -in mycert.pem -out newcert.pem
openssl x509 -in mycert.pem >>newcert.pem
```

Password hashes

Using the passwd option, you can generate password hashes that interoperate with traditional /etc/passwd files, newer-style /etc/shadow files, and Apache password files.

How do I generate a crypt-style password hash?

You can generate a new hash quite simply:

```
openssl passwd MySecret
```

```
8E4vqBR4UOYF.
```

If you know an existing password's "salt," you can duplicate the hash:

```
openssl passwd -salt 8E MySecret
```

```
8E4vqBR4UOYF.
```

How do I generate a shadow-style password hash?

Newer Unix systems use a more secure MD5-based hashing mechanism that uses an eight-character salt (as compared to the two-character salt in traditional crypt()-style hashes). Generating them is still straightforward using the -1 option:

```
openssl passwd -1 MySecret
```

```
$1$sXiKzkus$haDZ9JpVrRHBznY5OxB82.
```

The salt in this format consists of the eight characters between the second and third dollar signs, in this case sXiKzkus. So you can also duplicate a hash with a known salt and password.

```
openssl passwd -1 -salt sXiKzkus MySecret
```

```
$1$sXiKzkus$haDZ9JpVrRHBznY5OxB82.
```

Prime numbers

Current cryptographic techniques rely heavily on the generation and testing of prime numbers, so it's no surprise that the OpenSSL libraries contain several routines dealing with primes. Beginning with version 0.9.7e (or so), the prime option was added to the openssl binary.

How do I test whether a number is prime?

Pass the number to the prime option. Note that the number returned by openssl will be in hex, not decimal, format:


```
openssl prime 119054759245460753
```

```
1A6F7AC39A53511 is not prime
```

You can also pass hex numbers directly.

```
openssl prime -hex 2f
```

```
2F is prime
```

How do I generate a set of prime numbers?

Pass a bunch of numbers to openssl and see what sticks. The seq utility is useful in this capacity.

- define start and ending points

```
AQUO=10000
ADQUEM=10100
for N in $(seq $AQUO $ADQUEM); do
  # use bc to convert hex to decimal
  openssl prime $N | awk '/is prime/ {print "ibase=16;"$1}' | bc
done
```

Random data

How do I generate random data?

Use the rand option to generate binary or base64-encoded data.

- write 128 random bytes of base64-encoded data to stdout
openssl rand -base64 128
- write 1024 bytes of binary random data to a file
openssl rand -out random-data.bin 1024
- seed openssl with semi-random bytes from browser cache

```
cd $(find ~/.mozilla/firefox -type d -name Cache)
openssl rand -rand $(find . -type f -printf '%f:') -base64 1024
```

On a Unix box with a /dev/urandom device and a copy of GNU head, or a recent version of BSD head, you can achieve a similar effect, often with better entropy:

- get 32 bytes from /dev/urandom and base64 encode them

```
head -c 32 /dev/urandom | openssl enc -base64
```

You can get a wider variety of characters than what's offered using Base64 encoding by using strings:

1. get 32 bytes from /dev/random, grab printable characters, and
2. strip whitespace. using echo and the shell's command substitution
3. will nicely strip out newlines.

```
echo $(head -c 32 /dev/random | strings -1) | sed 's/[[:space:]]//g'
```

Make sure you know the trade-offs between the random and urandom devices before relying on them for truly critical entropy. Consult the random(4) man page on Linux and BSD systems, or random(7D) on Solaris, for further information.

S/MIME

S/MIME is a standard for sending and receiving secure MIME data, especially in e-mail messages. Automated S/MIME capabilities have been added to quite a few e-mail clients, though openssl can provide command-line S/MIME services using the smime option.

Note that the documentation in the smime(1) man page includes a number of good examples. How do I verify a signed S/MIME message?

It's pretty easy to verify a signed message. Use your mail client to save the signed message to a file. In this example, I assume that the file is named msg.txt:

```
openssl smime -verify -in msg.txt
```

If the sender's certificate is signed by a certificate authority trusted by your OpenSSL infrastructure, you'll see some mail headers, a copy of the message, and a concluding line that says Verification successful.

If the messages has been modified by an unauthorized party, the output will conclude with a failure message indicating that the digest and/or the signature doesn't match what you received:

```
Verification failure
23016:error:21071065:PKCS7 routines:PKCS7_signatureVerify:digest
failure:pk7_doit.c:804:
23016:error:21075069:PKCS7 routines:PKCS7_verify:signature
failure:pk7_smime.c:265:
```

Likewise, if the sender's certificate isn't recognized by your OpenSSL infrastructure, you'll get a similar error:

```
Verification failure
9544:error:21075075:PKCS7 routines:PKCS7_verify:certificate verify
error:pk7_smime.c:222:Verify error:self signed certificate
```

Most e-mail clients send a copy of the public certificate in the signature attached to the message. From the command line, you can view the certificate data yourself. You'll use the smime -pk7out option to pipe a copy of the PKCS#7 certificate back into the pkcs7 option. It's oddly cumbersome but it works.

```
openssl smime -pk7out -in msg.txt | \
openssl pkcs7 -text -noout -print_certs
```

If you'd like to extract a copy of your correspondent's certificate for long-term use, use just the first part of that pipe.

```
openssl smime -pk7out -in msg.txt -out her-cert.pem
```

At that point, you can either integrate it into your OpenSSL infrastructure or you can save it off somewhere for special use.

```
openssl smime -verify -in msg.txt -CAfile /path/to/her-cert.pem
```

How do I encrypt a S/MIME message?

Let's say that someone sends you her public certificate and asks that you encrypt some message to her. You've saved her certificate as her-cert.pem. You've saved your reply as my-message.txt.

To get the default—though fairly weak—RC2-40 encryption, you just tell openssl where the message and the certificate are located.

```
openssl smime her-cert.pem -encrypt -in my-message.txt
```

If you're pretty sure your remote correspondent has a robust SSL toolkit, you can specify a stronger encryption algorithm like triple

DES:

```
openssl smime her-cert.pem -encrypt -des3 -in my-message.txt
```

By default, the encrypted message, including the mail headers, is sent to standard output. Use the `-out` option or your shell to redirect it to a file. Or, much trickier, pipe the output directly to `sendmail`.

```
openssl smime her-cert.pem \  
-encrypt \  
-des3 \  
-in my-message.txt \  
-from 'Your Fullname <you@youraddress.com>' \  
-to 'Her Fullname <her@heraddress.com>' \  
-subject 'My encrypted reply' |\  
sendmail her@heraddress.com
```

How do I sign a S/MIME message?

If you don't need to encrypt the entire message, but you do want to sign it so that your recipient can be assured of the message's integrity, the recipe is similar to that for encryption. The main difference is that you need to have your own key and certificate, since you can't sign anything with the recipient's cert.

```
openssl smime \  
-sign \  
-signer /path/to/your-cert.pem \  
-in my-message.txt \  
-from 'Your Fullname <you@youraddress.com>' \  
-to 'Her Fullname <her@heraddress.com>' \  
-subject 'My signed reply' |\  
sendmail her@heraddress.com
```

Resources

- <http://www.madboa.com/geek/openssl/#cert-exam>

History

#1 - 01/15/2014 09:31 PM - Daniel Curtis

- Description updated

#2 - 08/09/2014 09:14 PM - Daniel Curtis

- Project changed from Automation Laboratory Technology to GNU/Linux Administration

#3 - 12/26/2014 10:09 AM - Daniel Curtis

- Description updated