stunnel basics and pki standards

Article Number: 375 | Rating: Unrated | Last Updated: Fri, Sep 28, 2018 7:39 PM

stunnel basics and pki standards

# REFERENCES

[1] http://www.sslshopper.com/article-most-common-openssl-commands.html <---- a great collection of openssl commands

[2] http://www.madboa.com/geek/openssl/ <---- a detailed "how-to" for most of the openssl questions that I had

[3] http://shib.kuleuven.be/docs/ssl_commands.shtml <---- another great collection of openssl commands

[4] http://www.seas.ucla.edu/~mkampe/cs111.wq11/docs/ssl.html <---- detailed data on the ciphers used with ssl

[5] http://www.bacula.org/5.1.x-manuals/en/misc/misc/Using_Stunnel_Encrypt_Commu.html <---- a more complicated scenario than what is presented below including a great description

[6] http://www.stunnel.org/static/stunnel.html#name <---- stunnel project page

# PURPOSE

There was a specific data stream that was unencrypted in the native application. There were no configurable options to provide encryption. The vendor had advised that encrypting this data would negatively affect performance and stability. The customer's security team would not budge on this finding. Enter Stunnel. "The stunnel program is designed to work as SSL encryption wrapper between remote clients and local (inetd-startable) or remote servers. The concept is that having non-SSL aware daemons running on your system you can easily set them up to communicate with clients over secure SSL channels." [6] The stunnel project was created and is currently maintained by Michal Trojnara.

# OVERVIEW

Stunnel provides an ssl wrapper for services, applications, and daemons that have not been built to support the ssl/tls specification. The wrapper provides each of the security controls that would be inherited if you had built the application to the ssl specification. These controls include confidentiality, integrity, and authentication. My understanding is that this was the original purpose of the ssl standard; to provide a suite of security controls that could be consumed by any application developer. This eliminated the need to develop home grown solution for encryption, authentication, and data integrity.

## BASIC OPERATION

Each server will need the same number of stunnel clients as the number of unique client/server relationships it supports.

----> On the client side, stunnel will set up a listener on local host using the same port that was configured to listen on the target server. This is a local listener on the client. The client sends data to this local listener in precisely the same way that it would have sent it to the remote server. All of the traffic the local stunnel listener receives will be ssl/tls encrypted and pushed out over the wire. *** CAVEAT*** There will be other small configuration changes that need to be made on the the client. For example, if the clients local application configuration points to the target server at 1.2.3.4:9103, this would need to be modified to point to 127.0.0.1:9103, which is where the stunnel client is listening.

----> The stunnel listener on the client will send this encrypted traffic to another stunnel listener on the server.

---->The stunnel listener on the server will be using an unallocated high port. For example, 19103.

----> The server side stunnel listener will unencrypt the traffic received and forward it to the original destination port configured for application. ie. 1903.

The beauty of this solution is that, aside from the local configuration changes, the client and the server are both unaware on the encrypted tunnel shared between the listeners. Stunnel is a neat little wrapper that cleans up and locks down a previously unencrypted data stream.

# SAMPLE IMPLEMENTATION

## CERTIFICATE STANDARDS: A number of x.509 certificate standards and openssl commands are included below. Public key cryptography is used with stunnel in very much the same way it is used in the standard ssl suite. The certificates and certificate locations are critical components for the stunnel configurations which are included below. At 30k feet it's important to remember that it's the public key crypto that is used to exchange the shared secret key (aka symmetric key) which encrypts the entire data stream once the tunnel is established.

----> x.509 digital certificates
- documents that bind a public key to an individual or company
- provided in DER and PEM formats
- based on ASN-1 notation
- can be encoded in binary form (DER) == distinguished encoding rules
- can be encoded in ascii form (PEM) == Privacy Enhanced Mail. This format includes the "Begin Certificate" and "End Certificate" anchors.

----> Public Key Cryptography Standards (PKCS) - partial list. My understanding is that each of these standards is used to define how the x.509 private keys are stored. The PKCS standards would appear to be a subset of the larger x.509 family of standards.
-- pkcs#10: standard for certificate generation request submitted to CA
-- pkcs#12: standard for storing and transporting a password protected private key. This may also include the corresponding public certificate

----> file types: there are multiple file types that correspond to the DER (binary), PEM (ascii), and PKCS formats. A partial list is included below.
- .p12
- .pfx
- .cer
- .der
- .crt
- .pem

## USEFUL OPENSSL COMMANDS [3]

convert DER (.crt .cer .der) to PEM [3]

```
---> openssl x509 -inform der -in MYCERT.cer -out MYCERT.pem
```

convert PEM to DER [3]

```
---> openssl x509 -outform der -in MYCERT.pem -out MYCERT.der
```

convert PKCS#12 (.pfx .p12) to PEM containing both private key and certificates.
add -nocerts for private key only; add -nokeys for certificates only. [3]

```
---> openssl pkcs12 -in KEYSTORE.pfx -out KEYSTORE.pem -nodes
```

convert (add) a seperate key and certificate to a new keystore of type PKCS#12. [3]

```
---> openssl pkcs12 -export -in MYCERT.crt -inkey MYKEY.key -out KEYSTORE.p12 -name "tomcat"
```

convert (add) a seperate key and certificate to a new keystore of type PKCS#12 for use with a server that should send the chain along too(eg Tomcat). you can repeat the combination of "-CAfile" and "-caname" for each intermediate certificate. [3]

```
---> openssl pkcs12 -export -in MYCERT.crt -inkey MYKEY.key -out KEYSTORE.p12 -name "tomcat"
-CAfile MY-CA-CERT.crt -caname myCA -chain
```

## STUNNEL PRE-REQUISITES

- stunnel.exe installed on the client and server machines
- Libeay32.dll installed on the client and server machines
- Libssl32.dll installed on the client and server machines
- CACert.pem this is the certificate authorities public/private key pair as a single ascii file
- Server.pem this is the server's public/private key pair as a single ascii file
- stunnel.conf
- client hash files

## INSTALLATION AND CONFIGURATION

```
-------------------------------------------------------------------------------
INITIAL STUNNEL INSTALL
-------------------------------------------------------------------------------
# double click on stunnel installer
# install to C:program filesstunnel
# double click on Win32OpenSSL-1_0_0d installer
## there is an error stating that C++ components will be needed for proper functioning. click through it.
## dlls are placed in the windows system32 folder
## libssl32.dll was copied and placed in the C:Program Filesstunnel folder
## libeay32.ddl was already resident in the stunnel folder following the stunnel install. It was not overwritten with the dll provided by the openssl install.

--------------------------------------------------------------------------------
CREATE THE CA
--------------------------------------------------------------------------------

# for the purposes of the initial install we will be creating our own CA for the creation and signing of digital certificates and private keys.

## create the digital certificate and private key for the CA
### follow the prompts and enter the password for the private key
### accept the defaults for all other values

> openssl -req -out CACert.pem -new -x509

## convert digital certificate and private key to a single .pfx (pkcs#12) file
### provide the password for the private key export <same as above>

> openssl pkcs12 -export -out CACert.pfx -inkey privkey.pem -in CACert.pem

## convert the .pfx file to a single .pem file with both the public and private keys
### provide the password for the private key import <same as above>
### verify that the digital certificate and private key are in the file

> openssl pkcs12 -in CACert.pfx -out CACertf.pem -nodes
> cat CACertf.pem

--------------------------------------------------------------------------------
CREATE THE SERVER CERTIFICATES [server 1]
--------------------------------------------------------------------------------

# create the server certificates using the newly created CA certificates.

## server
### create the private key for the sever
### "genrsa" specifies the algorithm and "1024" specifies the length of the key in bits
> openssl genrsa -out server1.key 1024
```

```
### create a certificate signing request for the servers digital certificate
> openssl req -key server1.key -new -out server1.req
```

```
### create file.srl
> touch file.srl
> vi file.srl
```

The contents of this file should be an even numbered digit that begins with "00".
The CA will count off from and assign serial numbers to the certificates it issues.
To view the certificate issued and the serial number used > openssl x509 -in server1.pem -text -noout

```
### sign the server's digital certificate using the private key for the CA created in earlier steps.
> openssl x509 -req -in server1.req -CA CACert.pem -CAkey privkey.pem -CAserial file.srl -out server1.pem
```

```
# convert digital certificate and private key to a single .pfx (pkcs#12) file
### provide the password for the private key export <same as above>

> openssl pkcs12 -export -out server1.pfx -inkey server1.key -in server1.pem
```

```
## convert the .pfx file to a single .pem file with both the public and private keys
### provide the password for the private key import <same as above>
### verify that the digital certificate and private key are in the file

> openssl pkcs12 -in server1.pfx -out server1.pem -nodes
> cat vmgrf.pem
```

CREATE THE CLIENT CERTIFICATES [server2]

```
# create the client certificates using the newly created CA certificates.
```

```
## client
### create the private key for the client
### "genrsa" specifies the algorithm and "1024" specifies the length of the key in bits
> openssl genrsa -out server2.key 1024
```

```
### create a certificate signing request for the servers digital certificate
> openssl req -key server2.key -new -out server2.req
```

```
### create file.srl
> touch file.srl
> vi file.srl
```

The contents of this file should be an even numbered digit that begins with "00".
The CA will count off from and assign serial numbers to the certificates it issues.
To view the certificate issued and the serial number used > openssl x509 -in server2.pem -text -noout

```
### sign the server's digital certificate using the private key for the CA created in earlier steps.
> openssl x509 -req -in server2.req -CA CACert.pem -CAkey privkey.pem -CAserial file.srl -out server2.pem
```

```
# convert digital certificate and private key to a single .pfx (pkcs#12) file
### provide the password for the private key export <same as above>

> openssl pkcs12 -export -out server2.pfx -inkey vmsec.key -in server2.pem
```

```
## convert the .pfx file to a single .pem file with both the public and private keys
### provide the password for the private key import <same as above>
### verify that the digital certificate and private key are in the file

> openssl pkcs12 -in server2.pfx -out server2.pem -nodes
> cat server2.pem
```

FINAL CERTIFICATE SET

----

server1.pem [server digital certificate and private key]
server2.pem [client digital certificate and private key]
CACertf.pem [CA digital certificate and private key]

CONSTRUCT THE CONFIGURATION FILES

Server1 is listening on *2* ports that I need my client [server2] to connect to: 3000 and 4000. Note that the client server [server2] is now listening locally on those ports [localhost:3000 and localhost:4000] which would normally be accessed remotely at the server [server1]. The stunnel listeners on the client [server2] picks the packets off the localhost interface, encrypts, and fires them across the wire to the stunnel listeners on server1. The stunnel listeners on server1 take the encrypted packets that arrive at the arbitrarily chosen high ports [10.10.10.1:30001 and 10.10.10.1:40001], unencrypts, and passes them on to ports 3000 and 4000 that are listening on localhost [server1].

server2 [client] Stunnel Configuration File

```
CAfile = CAcertF.pem
CApath = d:program filesstunnel
verify = 3
cert = server2.pem
client = yes
output = d:program filesstunnelstunnel.log
# uncomment for possible performance fix
# socket = r:TCP_NODELAY=1

[vnc-1]
accept = 127.0.0.1:3000
connect = 10.10.10.1:3000
sslVersion = TLSv1
ciphers = AES256-SHA

[vnc-2]
accept = 127.0.0.1:4000
connect = 10.10.10.1:4000
sslVersion = TLSv1
ciphers = AES256-SHA
# AES-128 is also FIPS-compliant
```

Server1 [server] Stunnel Configuration File

```
CAfile = CAcertf.pem
CApath = d:program filesstunnel
```

```
verify = 3
cert = server1.pem
client = no
output = d:program filesstunnellogsstunnel.log
# uncomment for possible performance fix
# socket = l:TCP_NODELAY=1

[vnc 1]
accept = 10.10.10.1:3000
connect = 127.0.0.1:3000
sslVersion = TLSv1
ciphers = AES256-SHA
# AES-128 is also FIPS-compliant

[vnc 2]
accept = 10.10.10.1:4000
connect = 127.0.0.1:4000
sslVersion = TLSv1
ciphers = AES256-SHA
# AES-128 is also FIPS-compliant


## Now restart server 1 and server 2 and away you go!!
```

Posted - Fri, Sep 28, 2018 7:39 PM. This article has been viewed 9591 times.

Online URL: http://kb.ictbanking.net/article.php?id=375