NAME

IO::Socket::SSL::Intercept −− SSL interception (man in the middle)

SYNOPSIS

```
use IO::Socket::SSL::Intercept;
# create interceptor with proxy certificates
my $mitm = IO::Socket::SSL::Intercept->new(
    proxy_cert_file => 'proxy_cert.pem',
    proxy_key_file  => 'proxy_key.pem',
    ...
);
my $listen = IO::Socket::INET->new( LocalAddr => .., Listen => .. );
while (1) {
    # TCP accept new client
    my $client = $listen->accept or next;
    # SSL connect to server
    my $server = IO::Socket::SSL->new(
        PeerAddr => ..,
        SSL_verify_mode => ...,
        ...
    ) or die "ssl connect failed: $!,$SSL_ERROR";
    # clone server certificate
    my ($cert,$key) = $mitm->clone_cert( $server->peer_certificate );
    # and upgrade client side to SSL with cloned certificate
    IO::Socket::SSL->start_SSL($client,
        SSL_server => 1,
        SSL_cert => $cert,
        SSL_key => $key
    ) or die "upgrade failed: $SSL_ERROR";
    # now transfer data between $client and $server and analyze
    # the unencrypted data
    ...
}
```

DESCRIPTION

This module provides functionality to clone certificates and sign them with a proxy certificate, thus making it easy to intercept SSL connections (man in the middle). It also manages a cache of the generated certificates.

How Intercepting SSL Works

Intercepting SSL connections is useful for analyzing encrypted traffic for security reasons or for testing. It does not break the end-to-end security of SSL, e.g. a properly written client will notice the interception unless you explicitly configure the client to trust your interceptor. Intercepting SSL works the following way:

- Create a new CA certificate, which will be used to sign the cloned certificates. This proxy CA certificate should be trusted by the client, or (a properly written client) will throw error messages or deny the connections because it detected a man in the middle attack. Due to the way the interception works there no support for client side certificates is possible.

  Using openssl such a proxy CA certificate and private key can be created with:

  ```
  openssl genrsa −out proxy_key.pem 1024
  openssl req −new −x509 −extensions v3_ca −key proxy_key.pem −out proxy_cert.
  # export as PKCS12 for import into browser
  openssl pkcs12 −export −in proxy_cert.pem −inkey proxy_key.pem −out proxy_ce
  ```

- Configure client to connect to use intercepting proxy or somehow redirect connections from client to the proxy (e.g. packet filter redirects, ARP or DNS spoofing etc).
- Accept the TCP connection from the client, e.g. don't do any SSL handshakes with the client yet.
- Establish the SSL connection to the server and verify the servers certificate as usually. Then create a new certificate based on the original servers certificate, but signed by your proxy CA. This is the step where IO::Socket::SSL::Intercept helps.
- Upgrade the TCP connection to the client to SSL using the cloned certificate from the server. If the client trusts your proxy CA it will accept the upgrade to SSL.
- Transfer data between client and server. While the connections to client and server are both encrypted with SSL you will read/write the unencrypted data in your proxy application.

METHODS

IO::Socket::SSL::Intercept helps creating the cloned certificate with the following methods:

$mitm = IO::Socket::SSL::Intercept->new(%args)

This creates a new interceptor object. `%args` should be

proxy_cert X509 | proxy_cert_file filename

This is the proxy certificate. It can be either given by an X509 object from Net::SSLeays internal representation, or using a file in PEM format.

proxy_key EVP_PKEY | proxy_key_file filename

This is the key for the proxy certificate. It can be either given by an EVP_PKEY object from Net::SSLeays internal representation, or using a file in PEM format. The key should not have a passphrase.

pubkey EVP_PKEY | pubkey_file filename

This optional argument specifies the public key used for the cloned certificate. It can be either given by an EVP_PKEY object from Net::SSLeays internal representation, or using a file in PEM format. If not given it will create a new public key on each call of `new`.

serial INTEGER|CODE

This optional argument gives the starting point for the serial numbers of the newly created certificates. If not set the serial number will be created based on the digest of the original certificate. If the value is code it will be called with `serial(original_cert,CERT_asHash(original_cert))` and should return the new serial number.

cache HASH | SUBROUTINE

This optional argument gives a way to cache created certificates, so that they don't get recreated on future accesses to the same host. If the argument ist not given an internal HASH ist used.

If the argument is a hash it will store for each generated certificate a hash reference with `cert` and `atime` in the hash, where `atime` is the time of last access (to expire unused entries) and `cert` is the certificate. Please note, that the certificate is in Net::SSLeays internal X509 format and can thus not be simply dumped and restored. The key for the hash is an `ident` either given to `clone_cert` or generated from the original certificate.

If the argument is a subroutine it will be called as `$cache->(ident,sub)`. This call should return either an existing (cached) `(cert,key)` or call `sub` without arguments to create a new `(cert,key)`, store it and return it. If called with `$cache->('type')` the function should just return 1 to signal that it supports the current type of cache. If it reutrns nothing instead the older cache interface is assumed for compatibility reasons.

($clone_cert,$key) = $mitm->clone_cert($original_cert,[$ident])

This clones the given certificate. An ident as the key into the cache can be given (like `host:port`), if not it will be created from the properties of the original certificate. It returns the cloned certificate and its key (which is the same for alle created certificates).

$string = $mitm->serialize

This creates a serialized version of the object (e.g. a string) which can then be used to persistantly store created certificates over restarts of the application. The cache will only be serialized if it is a HASH. To work together with Storable the `STORABLE_freeze` function is defined to call `serialize`.

$mitm = IO::Socket::SSL::Intercept->unserialize($string)

This restores an Intercept object from a serialized string. To work together with Storable the `STORABLE_thaw` function is defined to call `unserialize`.

AUTHOR

Steffen Ullrich