Using TLS to Improve NFS Security

BY RICK MACKLEM

Traditionally, NFS has provided very limited security based on the IP address/DNS host name of the client using exports(5). This can be subverted by IP address spoofing and simply does not work for mobile clients without any fixed, well-known IP address or DNS host name. Also, all data normally travels in clear text on the wire and, as such, can easily be sniffed.

RFC2203 was published September 1997 and provided a mechanism to alleviate at least some of the above issues via the use of GSSAPI with Kerberos mechanism, commonly referred to as Kerberized NFS. When used via the "`sec=krb5p`" (KerberosV with privacy), the RPC message's arguments and results are encrypted on the wire. Kerberos works well for user authentication but is less convenient for machine authentication. Unlike NFSv3, NFSv4 requires a "system principal" which is used to maintain the `Open/Byte_range` lock state on the server. Kerberos has the concept of a host-based Kerberos principal of the form "`host/<FQDN-of-machine>@REALM`", for which a keytab entry can be created and copied onto a client to be used as a "system principal". The "`<FQDN-of-machine>`" instance should protect the keytab entry from being used by another client, if compromised. However, this makes such a Kerberos principal useless for a mobile client without a fixed, well-known DNS host name. Also, for "`sec=krb5p`", only the data payloads of the RPCs are encrypted, exposing the RPC headers and making it impractical to offload the encryption/decryption to specialized hardware. In summary, when combined with the administrative effort involved in implementing a Kerberos environment, "`sec=krb5p`" has not been widely adopted and does not work well for mobile clients without a fixed, well-known DNS host name.

In an effort to improve NFS security, an Internet Draft titled "Towards Remote Procedure Call Encryption By Default" has been written, which describes the use of Transport Layer Security (TLS) to encrypt RPC message traffic on the wire along with the use of X.509 certificates for machine authentication. Since TLS is widely adopted, there are already specialized hardware offload solutions, not to mention efficient software implementations. This article describes the FreeBSD 13 implementation of this NFS over TLS, plus presents an example use case for mobile clients, such as laptops.

Implementation

Although I refer to it as NFS over TLS, it is more correctly Sun RPC over TLS, since the implementation is done in the kernel RPC (krpc) and is largely transparent to the NFS layer. OpenSSL's libraries provide a comprehensive implementation of TLS and handling of X.509 certificates in

user space. However, NFS is implemented in the kernel, and passing all NFS RPC messages up into user address space so that they can be handled by the OpenSSL libraries seemed impractical. Fortunately, FreeBSD 13's kernel added kernel TLS (KTLS) [TLS Offload in the Kernel, John Baldwin, *FreeBSD Journal*,May/June 2020 https://issue.freebsdfoundation.org/publication/?m=33057&i=667002&p=12&ver=html5] that performs efficient handling of TLS Application Data Records, including encryption/decryption, within the network stack in the kernel.

This provided the basic mechanism to encapsulate/encrypt the RPC messages in TLS Application Data Records and decrypt/de-encapsulate those RPC messages on the receive end. It does not, however, handle the non-Application Data Records, such as those used for the TLS handshake. To handle non-Application Data Records, rpc.tlsclntd(8) and rpc.tlsservd(8) were implemented in user space for the client and server respectively. These daemons handle upcalls from the kernel done via custom RPCs using the krpc over an AF_LOCAL socket, in a manner similar to what the gssd(8) daemon does for Kerberized NFS. To handle these upcalls, the daemons perform OpenSSL library calls to do the heavy lifting of handling non-Application Data Records, including handling of the TLS handshake. The daemons also use a custom system call to register with the krpc in the kernel, plus the odd case of needing to associate a file descriptor with an already extant socket in the kernel.

When a client wishes to do NFS over TLS, it performs a STARTTLS Null RPC. A Null RPC is an RPC with no arguments or results and is normally assigned RPC Number 0. To do a STARTTLS, the Null RPC request uses a new RPC credential type of AUTH_TLS. For the NFS service in FreeBSD, if rpc.tlsservd is running, the krpc replies with a credential verifier made up of the eight ASCII bytes "STARTTLS". This STARTTLS probe done by the NFS client triggers a TLS handshake to set up TLS on the TCP connection being used for RPC message transport.

The sequence of actions in the server at this point is:

- Block krpc reception on the TCP socket.
- Send the Null RPC reply with the credential verifier of "STARTTLS".
- Do a handshake upcall to the `rpc.tlsservd`.
  In the rpc.tlsservd to handle the handshake:
  - Acquire a file descriptor for the TCP socket.
    At this point the `krpc` has a TCP socket for the client's NFS connection
    but there is no file descriptor reference for it.
    This was one of the more challenging corners of the implementation.
    My solution was to use the daemon's custom syscall to associate a file descriptor with the socket.
    Once done, closure of the socket is relegated to the daemon instead of the krpc.
  - Add a structure to a linked list for the socket file descriptor,
    keyed on a unique 64bit reference number.
  - Call SSL_set_fd() to associate the socket with an SSL context.
  - Call SSL_accept() to do the actual handshake.
  - If the handshake succeeds, do `BIO_get_ktls_send()` and `BIO_get_ktls_recv()`
    calls to check that KTLS is now enabled on the socket.
    If either of these return zero, the handshake is considered failed.
  - Depending upon what command line options were specified for the daemon,
    any X.509 certificate provided by the client is verified and any
    user mapping specified by the certificate is used to create POSIX
    `<uid, gidlist>` credentials for the user.

- Replies to the upcall RPC with a set of flags indicating whether the handshake succeeded, if a verified certificate was received and POSIX user credentials mapped from the certificate, if any. Included in the reply is the unique 64bit reference number for the socket, along with the startup date/time for the daemon, so that the kernel can refer to the socket in subsequent upcalls. The startup date/time differentiates the reference number from the same reference number that might be used by a previous or subsequent instance of the daemon.
  - If the handshake succeeded, mark the krpc socket as using TLS, along with the flags and credentials, if any, in the upcall's reply.
  - Unblock kernel RPC reception on the socket.

The socket should now be ready to handle RPC messages, with the KTLS handling the Application Data Record encapsulation/encryption below the sosend() calls and the decryption/de-encapsulation below the soreceive() calls used by the krpc, if the handshake succeeded.

If a non-Application Data Record is at the head of the socket's receive queue, a new MSG_TLSAPPDATA flag for the soreceive() call indicates that the call should return ENXIO so that the non-Application Data Record will remain at the head of the socket's receive queue. The ENXIO return triggers an upcall to rpc.tlsservd to handle the non-Application Data Record. The kernel code blocks reception on the socket by the krpc and then does the handle record upcall to the daemon. The 64bit reference number, along with the daemon's start date/time are passed up in arguments, so that the daemon can identify the correct socket.

- This upcall simply does a SSL_read() with a length argument equal zero. This call always fails, but processes non-Application Data Records at the head of the socket's receive queue before failing.

The third upcall to the daemon is done to shut down and close the TCP socket, with the 64bit reference number and daemon start date/time as arguments.

- This upcall closes the socket and removes the socket's element from the linked list. If not already done, as indicated by `SSL_get_shutdown()`, this upcall also does a `SSL_shutdown()` before closing the socket, to send a Peer Reset TLS record to the client.

Although all of the above is handled by the krpc, the NFS server does use new flags related to TLS that are passed to the NFS server by the krpc for an RPC to determine if the RPC is permitted, based on the following `exports(5)` options.

There are three new `exports(5)` options:

tls - Indicates that the client must use NFS over TLS, but is not required to present any X.509 certificate to the server during TLS handshake.

tlscert - Indicates that the client must use TLS and must have provided a X.509 certificate during TLS handshake that verified.

tlscertuser - Indicates that the client must use TLS, must have provided a X.509 certificate during TLS handshake that verifier and that this certificate must have successfully mapped to a POSIX user

credential (<uid, gid_list>).
This mapping is generated from a login name found in the otherName component of subjectAltName with a "@domain", where "domain" matches the one the server uses.
This mapping is only generated if rpc.tlsservd is started with the -u/--certuser command line option.

If none of the above exports(5) options were specified, TLS is permitted, but not required. There is also a command line option for rpc.tlsservd that specifies that the daemon require that the rDNS name for the client's IP address match the "DNS" component of subjectAltName in the client's X.509 certificate. This is analogous to what RFC 6125 recommends that a client do to verify the identity of a domain-named application service. Since this option is intended to subvert client IP address spoofing, exports(5) cannot be used, since it is keyed on the client's IP address. As such, this option specifies that all clients doing NFS over TLS satisfy this criterion and failures result in handshake failures. It is the strongest client host identity check but requires that all clients have X.509 certificates that verify and where the DNS component of subjectAltName is correct. All clients must also have fixed, well-known DNS addresses when this option is specified.

The client daemon functions is a similar manner, but with some differences. Unlike rpc.tlsservd, rpc.tlscltnd only requires a certificate if the -m/--mutualverf command line option is specified. The client can also handle multiple certificates stored in different files, in case different NFS over TLS servers require different certificates.

When an NFS mount establishes a new TCP connection to the server, where the "tls" mount option has been specified, the krpc will do the following:

- Send the Null RPC request with the credential of type AUTH_TLS.
- If a Null RPC reply with a credential verifier consisting of the ASCII bytes "STARTTLS" is received.
  - Block kernel RPC reception on the new TCP socket.
  - Do a handshake upcall to the rpc.tlsclntd.
    In rpc.tlsclntd to handle the handshake:
    - Acquire a file descriptor for the TCP socket.
      At this point the krpc has a TCP socket for the client's NFS connection but there is no file descriptor reference for it.
      This is done by the daemon's custom system call, similar to rpc.tlsservd.
    - Call SSL_set_fd() to associate the socket with an SSL context.
    - If the daemon was started with the command line option -m/--mutualverf, SSL_[ctx_]use_certificate_file()/SSL_[ctx_]use_PrivateKey_file() are called to provide a certificate during the handshake.
      An argument for the upcall may override the default names for the certificate/key files.
      The default names are "cert.pem" and "certkey.pem", but may be overridden on a per-mount basis via the "tlscertname" mount option, in case different NFS servers require different certificates.
    - Call SSL_connect() to do the actual handshake.
    - If the handshake succeeds, do BIO_get_ktls_send() and

`BIO_get_ktls_recv()` calls to check that KTLS is now enabled on the socket.
If either of these return zero, the handshake is considered failed.
If the handshake is successful:
- Add a structure to a linked list for the socket file descriptor,
  keyed on a unique 64bit reference number.
- Reply to the upcall RPC indicating the handshake succeeded.
  Included in the reply is the unique 64bit reference number for the
  socket, along with the startup date/time for the daemon, so that the
  kernel can refer to the socket in subsequent upcalls.
  else:
  - close the socket.
  - Reply failure to the kernel, which will result in all subsequent
    NFS RPCs failing with EACCES.
- Upon receiving the upcall reply, the krpc sets a flag if the handshake
  succeeded and unblocks krpc reception on the socket.

For the client, if either the STARTTLS Null RPC or TLS handshake fails for a mount when the "tls" option has been specified, all RPCs will fail with EACCES. This is done so that NFS mounts with the "tls" option specified will not function unless TLS is working for the mount.

Mobile Devices Such as Laptops as a Use Case

A mobile device, such as a laptop, typically accesses the Internet from anywhere with no fixed, well-known IP address. To allow a laptop to mount an NFSv4 file server from anywhere on the Internet requires some reasonable security mechanism. Although NFS over TLS can be used for NFSv3 mounts, enabling NFSv3 mounts from anywhere is awkward, since the Mount protocol uses a dynamically assigned port number via rpcbind whereas NFSv4 mounts only use port #2049. A such, this example will use NFSv4 mounts.

It is possible to use Kerberized NFS with privacy to do a mount from a FreeBSD laptop. The laptop user would need to do commands such as:

```
# sysctl vfs.usermount=1   - done as su/root
# service gssd onestart - done as su/root
% kinit        - to acquire a TGT for the user
% mount -t nfs -o sec=krb5p,nfsv4,minorversion=1 nfsv4-server.uoguelph.
ca:/ /mnt
            - done as the non-root user
```

- Use the mount point.

```
% umount /mnt
```

By using the mount option "`sec=krb5p`", but not the "`gssname`" mount option, the FreeBSD client will use the "user principal" as the "system principal". This mount breaks badly if the user's TGT expires before the "umount".

As far as I know, this has not been widely adopted, possibly due to the effort required to install Kerberos and maintain a KDC accessible from anywhere on the Internet.

To do this using NFS over TLS requires the generation of a X.509 certificate for the client. Although there are many ways to create/sign X.509 certificates, this can easily be accomplished by a site local CA, managed by the openssl(1) command in FreeBSD 13.

- Create a certificate for the laptop, signed by the site local CA.
  Using `openssl(1)` the commands might be:
  ```
  # openssl genrsa -aes256 -out certkey.pem
  # openssl req -new -key certkey.pem -addext
  "subjectAltName=otherName:1.3.6.1.4.1.2238.1.1.1;UTF8:rmacklem@uoguelph.ca"
  -out req.pem
  # openssl ca -in req.pem -out cert.pem
  ```
- Copy `cert.pem` and `certkey.pem` into a directory named `/etc/rpc.tlsclntd`
  on the laptop in some secure manner.
- Enable the client daemon, using the certificate.
  - edit /etc/rc.conf and add:
  tlsclntd_flags="-m"
  - Due to the "-aes256" option, rpc.tlsclntd will query for the
    passphrase when starting, so it may be preferred to start the
    daemon manually before doing the mount instead of at boot time.
  - to start at boot time, add to `/etc/rc.conf: tlsclntd_enable="YES"`
  - or start it manually before doing the mount via: `# service tlsclntd onestart`
  - Once the laptop is connected to the Internet, the mount can be done as `su/root:`
  ```
  # mount -t nfs -o nfsv4,minorversion=1,tls nfsv4-server.uoguelph.ca:/ /mnt
  ```

Since the client presents a certificate signed by the site local CA, the server can be reasonably assured that the client has a certificate created by the site local CA administrator. The "-aes256" option used when creating the client's private key forces the rpc.tlsclntd to query for a passphrase to be entered for the key when rpc.tlsclntd is started. This subverts a trivial compromise where the laptop is stolen, or the certificate/key files are copied to another client. For the certificate to be used on an unauthorized client, the passphrase would have to somehow be captured/cracked.

It is also possible to revoke a certificate and add it to a CRL if for any reason the laptop should no longer be allowed to do the mount.

For the above example, all RPCs done on the server will be performed using the POSIX credentials (`<uid, gid_list>`) of the login name "rmacklem" on the NFSv4 server. This avoids any need for the laptop to have a uniform uid, gid space with respect to the server. It also limits the risk due to a compromised laptop to files accessible by "rmacklem". This optional configuration may not conform to the Internet draft. A co-author of the draft agrees that mapping client RPC credentials to a specific user based on the X.509 certificate presented during the TLS handshake is useful and has in fact coined the term "TLS Identity Squashing" for it. However, this individual would prefer a database that maps the certificate's <issuerName, serialNumber> tuple to the "user". He argues that putting the "user" in the certificate conflagrates "machine" vs "user" credentials. Being a pragmatist, I feel that putting the "user" in the certificate is just an easy way to implement this. The rpc.tlsclntd could be modified to use a flat file/database to implement this, if that were to become preferred practice.

The above is just one example use case. The command line options on the daemons allow a range of configurations, ranging from only requiring TLS to encrypt RPC messages on the wire to requiring all clients to present verifiable X.509 certificates where the DNS component of subjectAltName must match the rDNS name for the client's IP host address. The client may also verify the authenticity of the NFS server in the manner recommended by RFC 6125 for TLS domain-named application servers.

To set up NFS over TLS on FreeBSD 13, see:
https://people.freebsd.org/~rmacklem/nfs-over-tls-setup.txt
plus the man pages for `rpc.tlsclntd(8)`, `rpc.tlsservd(8)`, `exports(5)`, `ktls(4)`and `mount_nfs(8)`.

RICK MACKLEM For over 30 years, starting in 1980, Rick was the system administrator for a CS department, running BSD systems among others. When the VAX 11/780 running 4.3BSD was being replaced by MicroVAXII systems, there was a need for NFS on 4.3BSD, so Rick implemented that and contributed it to CSRG. A Usenix paper titled "Lessons Learned Tuning the 4.3BSD NFS Implementation" described the first implementation of NFS over TCP done as a part of this work. Now retired, Rick continues to work on NFS implementation for FreeBSD.