NFSv4
Internet-Draft
Intended status: Experimental
Expires: April 21, 2014
C. Lever
Oracle
October 18, 2013

End-to-end Data Integrity For NFSv4
draft-cel-nfsv4-end2end-data-protection-01

Abstract

End-to-end data integrity protection provides a strong guarantee that data an application reads from durable storage is exactly the same data it wrote previously to durable storage. This document specifies possible additions to the NFSv4 protocol enabling it to convey end-to-end data integrity information between client and server.

Requirements Language

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [RFC2119].

Status of this Memo

This Internet-Draft will expire on April 21, 2014.

Copyright Notice

Table of Contents

1. Introduction

1.1. Scope Of This Document

This document specifies a protocol based on NFSv4 minor version 2 [PROVISIONAL-NFSV42] that enables per-I/O data integrity information to be conveyed between an NFS client and an NFS server.

A key requirement is that data integrity verification is possible from application write to read. This does not mean that a single protection envelope must exist from application to storage. However, it must be possible to perform integrity checking during each step of an I/O request’s journey from application to storage and back.

Therefore, the authors will not address how an NFSv4 client handles integrity-protected read and write requests from applications, nor with how an NFSv4 server manages protection information on its durable storage. We specify only a generic mechanism for transmitting integrity-protected read and write requests via the NFSv4 protocol, which client and server implementors may use as they see fit.

A key interest in specifying and prototyping an integrity protection feature is exploring how I/O error handling and state recovery mechanisms in NFSv4 must be strengthened to guarantee the integrity of protected data.

Additionally, we want to identify exactly what modes of corruption are faced in environments where applications run on nodes separated from physical data storage. Do we expect corruption that has never been seen in SAN or DAS environments, particularly failure modes in NAS clients that cannot be detected by traditional means (such as looking for misplaced block writes)?

Finally, do we already have appropriate integrity protection mechanisms in the current protocol? Network-layer integrity mechanisms such as an integrity-protecting RPCSEC_GSS service have been around for years, and might be adequate. But do these mechanisms protect against CPU and memory corruption and application bugs, as well as malicious changes to data-at-rest?

1.2. Causes of Data Corruption

Data can be corrupted during transmission, during the act of recording, or during the act of retrieval. Data can become corrupt while at rest on durable storage. Either active corruption (e.g. data is accidentally or maliciously overwritten) or passive corruption (e.g. storage device failure) can occur.

Data storage systems must handle an increasingly large amount of data. If the rate of corruption stays fixed while the amount of data stored increases, we expect corruption to become more common.

To reduce failure rate and increase performance, data storage system complexity has increased. Complexity itself introduces the risk of corruption, since complexity can introduce bugs and make test coverage unacceptably sparse. Diagnosing a failure in complex systems is an everyday challenge.

Data corruption can be "detected" or "undetected" (silent). The goal of data integrity protection is not to make corruption impossible, but rather to ensure corruption is detected before it can no longer be corrected, or at least before corrupt data is used by an application.

1.3. End-to-end Data Integrity

End-to-end data integrity is a class of operating system, file system, storage controller, and storage device features that provide broad protection against unwanted changes to or loss of data that resides on data storage devices.

Typically, data integrity is verified at individual steps in a data flow using techniques such as parity. This provides isolated protection during particular transfer operations or at best between adjacent nodes in an I/O path.

In contrast, end-to-end protection guarantees data can be verified at every step as data flows from an application through a file system and storage controllers, via a variety of communication protocols, as it is stored on storage devices, and when it is read back from storage.

1.4. The Case For End-To-End Data Integrity Management

A modern NFSv4 deployment may already provide some degree of data protection to in-transit data.

- o The use of RPCSEC GSS Kerberos 5i and 5p [RFC2203] can protect NFSv4 requests from tampering or corruption during network transfer.

- o An NFSv4 fileserver can employ RAID or block devices that store additional checksum data per logical block, in order to detect media failure.

- o An advanced file system on an NFSv4 fileserver may protect data integrity by storing multiple copies of data or by separately storing additional checksums.

To demonstrate why end-to-end data integrity protection provides a stronger integrity guarantee than protection provided by the single-domain mechanisms above, consider the following cases:

- o On an NFSv4 fileserver, suppose a device driver bug causes a write operation to DMA the wrong memory pages to durable storage. The written data is incorrect, but the DMA transport checksum matches it. The DMA operation completes without reporting an error, and upper layers discard the original copy of the data.

- o Suppose an operating system or file system bug allows modifications to a page after it has been prepared for I/O and a checksum has been generated. The page and checksum are then written to storage. The written data does not represent the data originally by the application, and the accompanying stored checksum does not match it. The write operation completes without reporting an error, and upper layers discard the original copy of the data.

- o Suppose a RAID array on an NFSv4 server receives incorrect data for some reason. The array will generate RAID parity blocks from the incorrect data. The data is incorrect, but the accompanying parity matches it. The write operation completes without reporting an error, and upper layers discard the original copy of the data.

- o Suppose an application is writing data repeatedly to the same area of a file stored on an NFSv4 fileserver. Retransmits of an old write request become indistinguishable from new write requests to the same region. The written data always matches its appliction-generated checksum, but a replayed retransmission can overwrite newer data, and upper layers discard the original copy of the data.

- o Suppose a middle box is caching NFSv4 write requests on behalf of a number of NFSv4 clients. The wsize in effect for the clients does not have to match the wsize in effect between the middle box and the NFSv4 server. If the middle box fragments and reassembles the write requests incorrectly, the write requests appear to complete, but incorrect data is written to the NFSv4 server, and the clients discard the original copy of the data.

In none of these cases is corruption identified while the original data remains available to correct the situation. An end-to-end solution could have caught and reported each of these, allowing the data's originator to retry or report failure before the data loss is compounded.

1.5. Terminology

Buffer separation: Protection information and the data it protects is contained in distinct buffers which have independent paths to durable storage.

Checksum: A value which is used to detect corruption in a collection of data. It is usually computed by applying a simple operation (such as addition) to each element of the collection. Computing a checksum is a low-overhead operation, but is less effective at helping detect and correct errors than a CRC.

Cyclic Redundancy Check: A value which is used to detect corruption in a collection of data. It is based on a linear block error-correcting code. The hash function's generator polynomial is chosen to maximize error detection, and is typically more successful than either simple parity or a checksum. A CRC is efficient to compute with dedicated hardware, but can be expensive to compute in software.

Data corruption: Any undesired alteration of data. Data corruption can be "detected" or "undetected" (silent).

Data integrity: A database term used here to mean that a collection of data is exactly the same before and after processing, transmission, or storage.

Data integrity verification failure: A node in an I/O path has failed to verify protection information associated with some data. This can be because the data or the protection information has been corrupted, or the node is malfunctioning.

Integrity metadata: See "Protection information."

Latent corruption: Data corruption that is discovered long after data was originally recorded on a storage device.

Lost write: A write operation to a storage device which behaves as if the target data is stored durably, but in fact the data is never recorded.

Misdirected write: A write operation that causes the target data to be written to a different location on a storage device than was intended.

Parity: A single bit which represents the evenness or oddness of a collection of data. Checking a parity bit can reveal and help correct data corruption. Parity is easy to compute and requires little space to store, but is generally less effective than other methods of error correction. "Parity" can also refer to checksum data in a RAID.

Protection envelope: A set of nodes in an I/O system which together guarantee data integrity from input to output.

Protection information: Information about a collection of application data that allows detection and possibly correction of corruption. This can take the form of parity, a checksum, a CRC value, or something more complex. Also the formal name of an end-to-end data integrity mechanism adopted by T10 for SCSI block storage devices.

Protection interval: A collection of application data that is protected from corruption. The collection must be no larger or smaller than what can be written atomically to durable storage. Typically there is a one-to-one mapping between a protection interval and a logical block on a storage device. However, a device with a large sector size may store multiple protection intervals per sector, to maintain adequate protection with limited protection information.

Protection type: An enumerated value that indicates the the size, contents, and interpretation of fields containing protection information.

2. Protocol

This section prescribes changes to the NFSv4 XDR specification [PROVISIONAL-NFSV42-XDR] to enable the conveyance of Protection Information via NFSv4. Therefore, an NFSv4.2 implementation is a necessary starting point. These changes are compatible with the NFSv4 minor versioning rules described in the NFSv4.2 specification.

The RPC protocol used by NFSv4 is ONC RPC [RFC5531]. The data structures used for the parameters and return values of these procedures are expressed in this document in XDR [RFC4506].

2.1. Protection types

A new fixed-size structure is defined that encodes the format and content of Protection Information. This includes the meaning of tags, the size of the protection interval, and so on.

For NFS, we need to go beyond existing SCSI protection types and consider cryptographic integrity types (i.e. the ability to guarantee integrity of data-at-rest over time by means of digital signature).

To begin, we provide NFSv4 equivalents for a few typical T10 PI protection types [T10-SBC2], in addition to a few new protection types:

```
enum nfs_protection_type4 {
        NFS_PI_TYPE1     = 1,
        NFS_PI_TYPE2     = 2,
        NFS_PI_TYPE3     = 3,
        NFS_PI_TYPE4     = 4,
        NFS_PI_TYPE4     = 5,
};

struct nfs_protection_info4 {
        nfs_protection_type4 pi_type;
        uint32_t             pi_intvl_size;
        uint64_t             pi_other_data;
};
```

The pi_type field reports the protection type. The pi_intvl_size field reports the supported protection interval size, in octets. The meaning of the content of the pi_other_data field depends on the protection type.

2.1.1. Protection Type Table

The following table specifies tag sizes and contents, and other features of each protection type.

NFS Protection Type	Description	pi_other_data	Comment
1	PI field is application-owned; 8-byte protection information field containing a SHA-1 hash of the protection interval	Always zero	NFS "native" PI
2	PI field is application-owned; 8-byte protection information field containing a hash of the protection interval signed by a private key. A public key is provided separately so the server can verify incoming protection intervals	Zero means the RSASSA-PKCS1-v1_5 signing scheme [RFC3447] is used	NFS "native" PI
3	8-byte protection information field containing 2-byte guard tag (CRC-16 checksum of protection interval), 2-byte application tag (user defined), and 4-byte reference tag (LO 32-bits of LBA)	1 if the PI field is application-owned; otherwise zero	T10 PI Type 1

4	8-byte protection information field containing 2-byte guard tag (CRC-16 checksum of protection interval), 2-byte application tag (user defined), and 4-byte reference tag (*)	1 if the PI field is application-owned; otherwise zero	T10 PI Type 2
5	PI field is application-owned; 8-byte protection information field containing 2-byte guard tag (CRC-16 checksum of protection interval), 2-byte application tag (user defined), and 4-byte reference tag (user defined)	1 if the PI field is application-owned; otherwise zero	T10 PI Type 3

The protection type enumerator is key to the extensibility of the NFSv4 end-to-end data integrity feature. A future specification can introduce new protection types that support Advanced Format drives, or types for storage that does not support application-owned Protection Information fields, for example. To manage this ongoing process, the contents of this table should be administered by IANA.

[*] Protection Type 2 uses an indirect LBA in its reference tag. In this case, the I/O operation passes the reference tag value for the first protection interval in a separate operation. The reference tag in the first protection field must match this value. The reference tags in subsequent fields are this value plus (n-1).

It’s still not clear to me how type 2 works without chaining read and write requests. When an application writes a series of unrelated blocks, what should the reference LBNs be? When an application reads randomly, what reference LBNs should it expect?

2.2. GETATTR

A new read-only per-FSID GETATTR attribute is defined to request the list of protection types supported on a particular FSID.

```
const FATTR4_PROTECTION_TYPES = 82;
```

The reply data type follows.

```
typedef nfs_protection_info4 fattr4_protection_info<>;
```

2.3. INIT_PROT_INFO - Initialize Protection Information

Some protection types require additional data in order for the storage to perform integrity verification. This data is transmitted by a new operation.

2.3.1. ARGUMENTS

```
struct INITPROTINFO4args {
        nfs_protection_type4 ipi_type;
        opaque               ipi_data;
};
```

2.3.2. RESULTS

```
struct INITPROTINFO4res {
        nfsstat4             status;
};
```

2.3.3. DESCRIPTION

This operation is used to transmit initialization data in preparation for a stream of integrity-protected I/O requests. The exact content of the ipi_data field depends on the protection type specified in the ipi_type field.

For example, for NFS_PI_TYPE2, the ipi_data field might contain a binary format public key that can be used to validate the signature of incoming protection intervals.

2.4. New data content type

NFSv4.2 introduces a mechanism that can be used to extend the types of data that can be read and written by a client. To convey protection information we extend the data_content4 enum.

```
enum data_content4 {
        NFS4_CONTENT_DATA            = 0,
        NFS4_CONTENT_APP_DATA_HOLE   = 1,
        NFS4_CONTENT_HOLE            = 2,
        NFS4_CONTENT_PROTECTED_DATA  = 3,
};

struct data_protected4 {
        nfs_protection_info4 pd_type;
        offset4              pd_offset;
        bool                 pd_allocated;
        opaque               pd_info<>;
        opaque               pd_data<>;
};
```

The pd_offset field specifies the byte offset where data should be read or written. The number of bytes to write is specified by the size of the pd_data array.

The pd_allocated field is equivalent to the d_allocated field in the data4 type specified in [PROVISIONAL-NFSV42].

The opaque pd_info field contains a packed array of fixed-size protection fields. The length of the array must be consistent with the pd_offset and count arguments specified for the data range of the operation. The size and format of the contents of each field in the array is determined by the value of the pd_type field.

The opaque pd_data field contains the normal data being conveyed in this operation.

2.5. READ_PLUS

The READ_PLUS operation reads protection information using the NFS4_CONTENT_PROTECTED_DATA content type.

```
   union read_plus_content switch (data_content4 rpc_content) {
   case NFS4_CONTENT_DATA:
           data4                rpc_data;
   case NFS4_CONTENT_APP_DATA_HOLE:
           app_data_hole4       rpc_adh;
   case NFS4_CONTENT_HOLE:
           data_info4           rpc_hole;
   case NFS4_CONTENT_PROTECTED_DATA:
           data_prot_fields4    rpc_pdata;
   default:
           void;
};
```

The offset and length arguments of the READ_PLUS operation (rpa_offset and rpa_count) determine the data byte range covered by the protection information and normal data returned in each request.

For example, suppose the protection type mandated 8-byte protection fields and a 512-byte protection interval. A READ_PLUS requesting protection information for a 4096-byte range of a file would receive an array of eight 8-byte protection fields, or 64 bytes.

2.6. WRITE_PLUS

The WRITE_PLUS operation writes protection information using the NFS4_CONTENT_PROTECTED_DATA content type.

```
union write_plus_arg4 switch (data_content4 wpa_content) {
case NFS4_CONTENT_DATA:
        data4               wpa_data;
case NFS4_CONTENT_APP_DATA_HOLE:
        app_data_hole4      wpa_adh;
case NFS4_CONTENT_HOLE:
        data_info4          wpa_hole;
case NFS4_CONTENT_PROTECTED_DATA:
        data_prot_fields4   wpa_pdata;
default:
        void;
};
```

The offset and length arguments of the WRITE_PLUS operation (pd_offset and the size of pd_data) determine the data byte range covered by the protection information.

For example, suppose the protection type mandated 8-byte protection fields and a 512-byte protection interval. A WRITE_PLUS writing protection information to a 4096-byte range of a file would send an array of eight 8-byte protection fields, or 64 bytes.

2.7. Error codes

New error codes are introduced to allow an NFSv4 server to convey integrity-related failure modes to clients. These new codes include (but are not limited to) the following:

```
enum nfsstat4 {
...
        NFS4ERR_PROT_NOTSUPP = 10200,
        NFS4ERR_PROT_INVAL   = 10201,
        NFS4ERR_PROT_FAIL    = 10202,
        NFS4ERR_PROT_LATFAIL = 10203,
```

```
};
```

NFS4ERR_PROT_NOTSUPP: The protection type specified in an operation is not supported for the FSID upon which the file resides.

NFS4ERR_PROT_INVAL: The protection information passed as an argument is garbled (cf. BADXDR). This error code MUST be returned if the offset and length of read or written data does not align with the protection interval specified by the protection type.

NFS4ERR_PROT_FAIL: During a WRITE_PLUS operation, the protection information does not verify the written data. If this was an UNSTABLE WRITE_PLUS, the client should retry the operation using FILE_SYNC so the server can report precisely where the data writes are failing.

NFS4ERR_PROT_LATFAIL: During a READ_PLUS operation, the protection information does not verify the read data. This error code reports a verification that occurred before the data arrives at an NFSv4 client. The client is not required to read protection information to see this error.

If data integrity verification fails while a server is pre-fetching data, the failure cannot be reported until the client reads the section of the file where the failure occurs. Pre-fetched data might never be read by a client, therefore a data integrity verification failure that occured while pre-fetching may never be reported to an NFS client or an application.

3. Protocol Design Considerations

3.1. Protection Envelopes

We explore protection envelopes that might appear in a typical NFSv4 deployment, and design an architecture that guarantees unbroken data integrity protection through each of these envelopes.

In addition, it is useful to permit varying degrees of server, client, and application participation in a data protection scheme. We can define protection envelopes of varying circumference that allow implementations and deployments to choose a level of complexity, data protection, and performance impact that suits their applications.

The following are presented in order of smallest to largest circumference. To enable end-to-end protection, each protection envelope in this list depends on having the previous envelope in place.

Server storage: The storage subsystem on an NFSv4 server is below the physical filesystems on that server. If a data integrity mechanism is available on the block storage, the physical filesystem may or may not choose to use it. Data integrity verification failures are reflected to NFS clients as simple I/O errors.

Server filesystem: The physical filesystem on an NFSv4 server may provide a data integrity mechanism based on its own checksumming scheme, or by using a standard block storage mechanism such as T10 PI/DIX [DIX]. The NFSv4 service on that system may or may not choose to use the filesystem's integrity service. Data integrity verification failures are reflected to NFS clients as simple I/O errors.

Server: An NFSv4 server may choose to use the local filesystem's data integrity mechanism, but not to advertise a data integrity mechanism via NFSv4. Data integrity verification failures are reflected to NFS clients as simple I/O errors.

Client-server: If an NFSv4 server advertises data integrity mechanisms via NFSv4, an NFSv4 client may choose to use NFSv4 data integrity protection without advertising the capability to applications running on it. It may also choose not to use NFSv4 data integrity protection at all. Data integrity verification failures are reflected to applications as simple I/O errors.

Application-client-server: Suppose that an NFSv4 client chooses to use data integrity protection via NFSv4 and that. the capability is advertised to applications. Applications may or may not choose to use the capability. An NFSv4 client uses on-the-wire data integrity when an application chooses to use the capability, but may or may not use it when the application chooses not to use it. Data integrity verification failures are reflected to applications as is. This is full end-to-end data integrity protection via NFSv4.

Note that the "server" envelope is not externally distinguishable from a server that does not support data integrity protection at all (other than that it provides somewhat better data integrity guarantees than one that does not support data integrity protection). This is a way to introduce stronger data integrity without requiring a large deployment of NFSv4 clients capable of integrity verification. Or, stronger data integrity can be introduced to legacy NFS environments that have no protocol mechanisms for extending the protection envelop past the server.

The "application-client-server" envelope illustrates that, on a protection-enabled file system, data integrity verification can be used on a per-file basis. Applications may choose to use protection for some files and not others. Some applications may choose to use protection, and some applications may choose not to use it.

Note that in each case, data integrity protection is available to the edge of the farthest protection envelope. Data integrity is protected only after the data arrives at a protection envelope boundary, and before it leaves that boundary. Legacy NFS clients continue to access protected data on a server, but are unaware of data integrity verification failures except as generic I/O errors.

The client-cache-server case is considered separately. The "cache" node in this case may be a dedicated NFSv4 cache, a caching peer-to-peer NFSv4 client, or a pNFS metadata server. A separate protection envelope exists between an NFSv4 client and an intermediate cache, and that cache and the NFSv4 server where the protected data resides.

3.2. Protecting Holes

NFSv4 minor version 2 [PROVISIONAL-NFSV42] exposes clients to certain mechanics of the underlying file systems on servers which allow more direct control of the storage space utilized by files. The goal of these new features is to economize the transfer and storage of file data. These new features include support for reading sparse files efficiently, space reservation, and punching holes (similar to a TRIM or DISCARD operation on a block device) in files.

A hole is an area of a file that can be represented in a file system by having no backing storage. By definition any read of that region of the file returns a range of bytes containing zero. Any write to that region allocates fresh backing storage normally.

NFSv4.2 extends this notion to allow NFSv4 clients to specify a pattern containing non-zero bytes to be returned when reading that region of a file. The protocol feature is independent of how an NFSv4 server's file system chooses to store this data. In fact a server's file system is free to simply store zeroes or a byte pattern on disk as raw data rather than in some optimized fashion.

If an NFSv4 server's file system does use an optimized storage method, a decision must be made about whether accompanying PI is needed. For a plain hole (where zero is always returned by a raw data read operation) the intention is that there is no backing storage there, thus PI is not meaningful. However a read operation that requests protection information must return something meaningful. For protection types that mandate only a checksum guard tag (and do not store either reference or or application tag data), a checksum for each protection interval can be generated on the server during a normal read operation, or on the client if a sparse read is used.

For a data hole (where some non-zero pattern is returned by a raw read operation), storing PI is optional, and depends on whether the protection type requires the storage to return an intact application tag. Without the requirement of storing the application tag, the file system could discard the PI after a write operation, and recompute it from the pattern on a read operation. Or, it could store the PI information as part of the pattern metadata.

3.3. Multi-server Considerations

The NFSv4 protocol provides several mechanisms for NFSv4 servers to co-operate in ways that enhance performance scalability and data availability. An NFSv4 client can access the same data serially on single NFSv4 servers when a file system is replicated. A file system can be migrated between NFSv4 servers transparently to clients. Or a file system can be constructed from files that reside in parts on several NFSv4 servers.

To allow coherent use of a data integrity mechanism:

- o Each NFSv4 Data Server hosting a particular file system MUST support the same protection types.

- Each replica of a file system MUST support the same protection types.

- The destination of a file system migration MUST support all protection types supported by the source, and the transitioned file system MUST use the same protection type it did on the source server.

Enforcing these mandates is likely outside the purview of the NFSv4 protocol, particularly because no mechanism for transitioning file systems is set out by any NFSv4 protocol specification. However, enforcing such mandates could be built into administrative tools.

3.3.1. pNFS and Protection Information

There has been some uncertainty about whether Protection Information should be considered metadata or data. pNFS has a convenient operational definition of data and metadata: if it’s data, it goes to the Data Server; if it’s metadata, it goes to the Metadata Server.

Protection Information belongs with the data it protects, which is written to Data Servers. Therefore Protection Information is data. If a client ever writes Protection Information to a Metadata Server, such Protection Information will be forwarded to an appropriate Data Server for storage.

For the file layout type, which uses NFSv4 when communicating with Data Servers, all protection types have protocol support for Protection Information. For other layout types, support may or may not be available in their respective data protocols. Layout implementations are not guaranteed to support every protection type.

3.3.2. Server-to-server copy

NFSv4 minor version 2 [PROVISIONAL-NFSV42] introduces a new multi-server feature known as server-to-server copy. Clients can offload the data copy portion of copying part or all of a file. The destination file is recognized as a separate entity (ie. has a unique file handle), not as a replica of the original file.

As such, the destination file may be stored in a file system that has a different protection type than the source file, or may not be protected at all. If the destination filesystem supports the same protection type as the source filesystem, the copy offload operation MUST copy Protection Information associated with the source file to the destination file.

Server implementors MAY provide data integrity verification on both ends of the offloaded copy operation. A server MUST report data integrity verification failures that occur during an offloaded copy operation.

4. Security Considerations

A man-in-the-middle attack can replace both the data and integrity metadata in any NFSv4 request that is sent in the clear. Therefore, when a data integrity protection mechanism is deployed on an untrusted network, it is strongly urged that a cryptographically secure integrity-checking RPC transport, such as RPCSEC GSS Kerberos 5i [RFC2203], is used to convey NFSv4 traffic on open networks.

5. IANA Considerations

This document currently does not require actions by IANA. However, see Section 2.1.

6. Acknowledgements

The author of this document gratefully acknowledges the contributions of Martin K. Petersen, David Noveck, and Spencer Shepler. Bill Baker, Chris Mason, and Tom Haynes also provided guidance and suggestions.

7. References

7.1. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

[RFC2203] Eisler, M., Chiu, A., and L. Ling, "RPCSEC_GSS Protocol Specification", RFC 2203, September 1997.

[RFC3447] Jonsson, J. and B. Kaliski, "Public-Key Cryptography Standards (PKCS) #1: RSA Cryptography Specifications Version 2.1", RFC 3447, February 2003.

[RFC4506] Eisler, M., "XDR: External Data Representation Standard", STD 67, RFC 4506, May 2006.

[RFC5531] Thurlow, R., "RPC: Remote Procedure Call Protocol Specification Version 2", RFC 5531, May 2009.

7.2. Informative References

[DIX] Petersen, M., "I/O Controller Data Integrity Extensions", November 2009, <http://oss.oracle.com/~mkp/docs/dif.pdf>.

[PROVISIONAL-NFSV42] Haynes, T., Ed., "NFS Version 4 Minor Version 2", March 2013, <http://datatracker.ietf.org/doc/draft-ietf-nfsv4-minorversion2>.

[PROVISIONAL-NFSV42-XDR] Haynes, T., Ed., "NFS Version 4 Minor Version 2 Protocol External Representation Standard (XDR) Description", March 2013, <https://datatracker.ietf.org/doc/draft-ietf-nfsv4-minorversion2-dot-x>.

[T10-SBC2] Elliott, R., Ed., "ANSI INCITS 405-2005, Information Technology - SCSI Block Commands - 2 (SBC-2)", November 2004.

Author's Address

Charles Lever
Oracle Corporation
1015 Granger Avenue
Ann Arbor, MI 48104
US

Phone: +1 734 274 2396
Email: chuck.lever@oracle.com