

# Security audit of NFSv4 implementation on GNU/Linux

Jonathan Lyard {[jonathan.lyard@bull.net](mailto:jonathan.lyard@bull.net)}  
Tony Reix {[tony.reix@bull.net](mailto:tony.reix@bull.net)}  
<http://nfsv4.bullopen-source.org>

August 24, 2006

## Version history

Version	Date	Status
1.0	08/23/2006	Submitted paper
0.1	07/19/2006	Draft

## Audited versions of NFSv4 :

**NFSv4 kernel version** : 2.6.17-rc2 patched with CITI  
linux-2.6.17-rc1-CITLNFS4\_ALL-1.diff

**RPCSEC\_GSS** : librpcsecgss-0.8

## Abstract

On Linux, NFSv4 comes to be more and more stable and starts being integrated in several distros (Red Hat and Novell). It is now time for administrators to replace their old NFS versions by the version 4. Since Linux is widely distributed across the world, NFSv4 will eventually be installed and used on a huge number of boxes. This means that the protocol should be efficient, robust and **secure**.

First of all, NFSv4 was designed to operate in WAN/Internet context : that introduces new security issues since the NFS traffic is now leaving the LAN. It will have to go through untrusted areas where the risk of being attacked is high. Then, lots of administrators will appreciate the support for Kerberos 5 to secure NFSv4 exchanges on their LAN. If the specification of the Kerberos is considered secure, there was an important need to audit how NFSv4 implementation makes use of Kerberos 5.

When I joined NFSv4 project, little work had been done on the security audit. The source code had already been audited but it was not clearly stated which parts of the source code were analyzed nor which vulnerabilities had been looked for. It was necessary to have someone working full-time on this project in order to initiate the security audit. I was not experienced with the security audit

of network protocols but I had a background in network and systems security.

This paper describes my contribution to the security analysis of NFSv4 and gives some hints for one willing to achieve this work. In Part I, we describe the adversary model and the scenarios to attack NFSv4. In part II, we explain the security protocols used by NFSv4 and in part III, we provide the methodology and the techniques used for the source audit. Finally, in part IV, we present innovative methods to dynamically audit NFSv4 : fuzzing and fault injection and their applications in SPIKE framework.

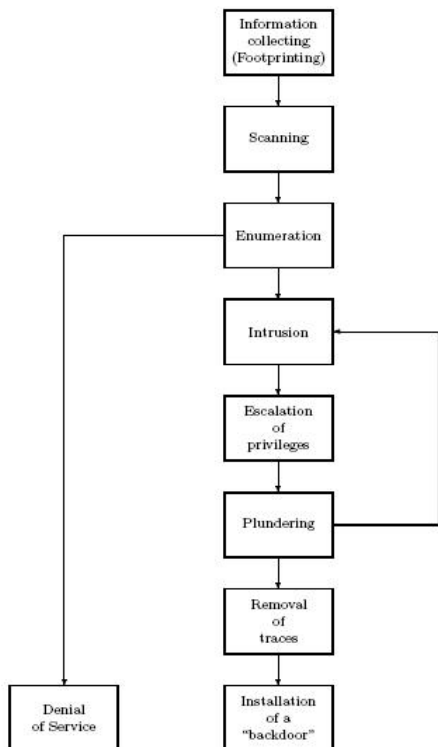
## Part I

# Adversary model and scenarios to attack NFSv4

We start this security analysis with a preliminary study of the adversary model so as to identify which threats our software could be facing. Contrary to older versions that were limited to LAN, NFSv4 will be used in WAN/Internet networks and that's why a security analysis is critical.

The systematic attack process is depicted on figure 1. First, the attacker collects information about the target and scans the TCP/UDP ports, the services running behind the open ports, the versions in use. Then, the attacker searches for the underlying vulnerabilities and try to exploit one to penetrate the remote machine. If he can't hack into the system, he can also try to launch DoS<sup>1</sup> attacks to prevent legitimate clients from connecting to the server.

Figure 1: Systematic attack of a target



We first show that NFSv4 server can be easily detected on a network and that one can easily gather information about those servers. Then, we depict the three main attack scenarios against NFSv4.

## 1 Data-gathering

We consider an attacker who is aware of some weaknesses in NFSv4 implementation and wants to exploit them to corrupt exported file systems or even to execute arbitrary shellcode. Identifying the NFSv4 on a network and gathering information about them is trivial :

- `rpcinfo -p` allows to detect all RPC services running on a machine (including nfs service). Port is SUN RPC reserved port (TCP 111). This port cannot be blocked since the portmapper is used by NFSv4 so far<sup>2</sup> (though NFSv4 listens on a well-defined port).

<sup>1</sup>DoS : Deny of Service, prevents legitimate use of a system instead of trying to break into it. Risks should not be neglected since NFSv4 servers availability can be primordial.

<sup>2</sup>The portmapper should eventually not be used in NFSv4 since the new version use a well-defined TCP port.

- A well-defined port (2049) eases detection with port-scanning tools (nmap [1] and others...)
- NULL procedure which - according to old RFC 2623 [2] - is used by NFS client to determine if an NFSv4 server is operating and responding to requests. For instance, mounting of a remote filesystem starts with a first exchange of NULL request/reply. Writing a scanner, which establishes a TCP connexion on port 2049 and makes a NULL exchange to discover operating NFSv4 servers on an IP address range, is really trivial. We have put this into practice using SPIKE framework [3].
- Minor version negotiation allows an attacker to determine the list of supported versions on a NFSv4 server.

---

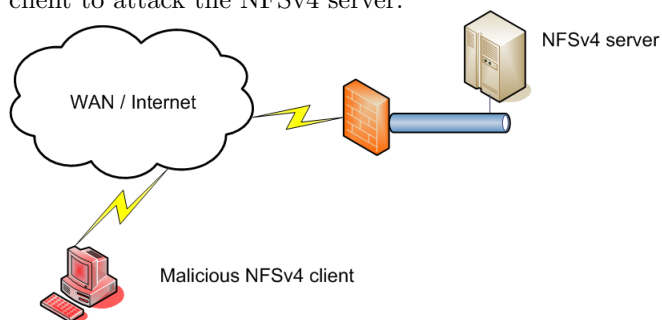
**Probability of attempt** : High - **Probability of success** : High - **Risks** : Low - **Complexity** : Very Low

**Counter measures** :

- Firewalls in front of NFSv4 server should allow Sun RPC's (TCP/UDP 111) and NFSv4's (i.e. TCP 2049). All other ports should be blocked if unused.
  - Mandate authentication (RPCSEC\_GSS) to make data-gathering harder.
- 

## 2 Attacks from a modified NFSv4 client

Figure 2: An adversary setting-up a malicious NFSv4 client to attack the NFSv4 server.



An attacker can try to exploit vulnerabilities by sending malicious requests to the NFSv4 server. This can trigger faults like overflows thus giving the possibility to launch DoS attacks or shellcode injection<sup>3</sup>.

<sup>3</sup>Shellcode is a set of instructions injected and then executed by an exploited program. The term shellcode is derived from its original purpose : it was the specific portion of an exploit used to spawn a root shell.

He can try to exploit known vulnerabilities that are not patched in the version in use. He can also use fuzzing techniques (i.e. fault injection) so as to discover some holes and hopefully find a way to exploit them.

Fuzzing is explained in the present paper ; NFSv4 security auditors should use such techniques to discover and patch vulnerabilities before an evil-minded person tries to exploit them.

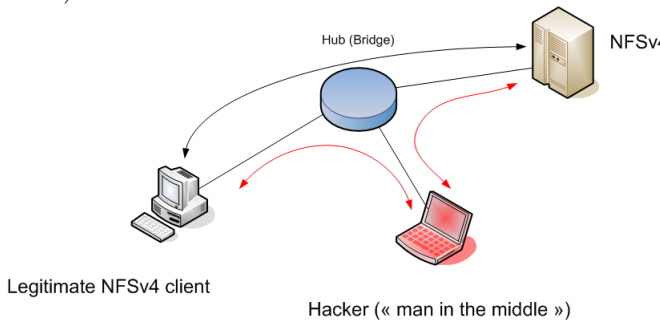
**Exposure level :** High

**Counter measures :**

- When clients can be trusted, the best is to authenticate them with a fit security protocol (Kerberos 5, SPKM-3 and LIPKEY so far). Like this, unauthorized user will reach a smaller part of potentially-vulnerable code ; typically a hacker will not be able to exploit vulnerabilities in server’s components that require authentication beforehand.
- NFSv4 should be regularly checked against vulnerabilities so as to hopefully discover them before an attacker try to exploit them.
- Like for any distributed software, NFSv4 administrators must regularly upgrade to latest stable version and install patches to correct bugs and vulnerabilities. The best place to get the latest NFSv4 versions and patches is on the NFSv4 developer’s Website [4]. The reader can also find the list of recommended and latest stable versions on our Website [5].

### 3 Attacks against NFSv4 requests/responses

Figure 3: An adversary setting-up a “man-in-the-middle” attack against NFSv4 on the LAN (here a bridged network)



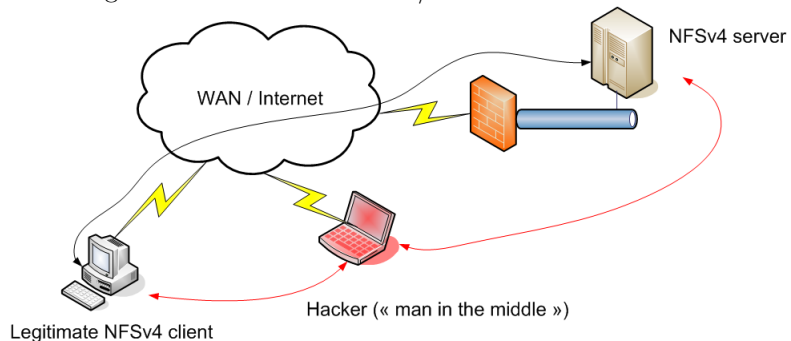
In this scenario, a hacker sniffs and possibly corrupts the request/responses exchanged between a client and the NFSv4 server. This is certainly the most common scheme when it is about attacking distributed software.

The attacker may be located somewhere on the network path between the client and the server ; setting eve’s dropping attacks is thus trivial. Else, the attacker should try to redirect the traffic to a machine under its control. Let us now give an overview of how the attacks can turned into practice.

On Ethernet networks, anyone connected on the same LAN can passively spy every frames. On a shared LAN (repeaters - one collision domain), a hacker just has to set his network interface in the so-called promiscuous mode (that is retrieve all frames, not only those intended to the interface). On a switched LAN (bridges - one collision domain per interface), one can still spy all frames thanks to techniques like ARP Spoofing [6], ARP Cache Poisoning [7], ICMP redirect [8], corruption of proxy server [9]...

EtterCap [10] is a good swiss knife to attack LAN networks and can be used to put the latter techniques to work.

Figure 4: An adversary setting-up a “man-in-the-middle” attack against NFSv4 on the WAN/Internet



On wider networks - where the attacker is not nearby the victim - the task is more tricky. But advanced techniques such as DNS Spoofing [11] can solve the issue. Achieving a DNS Spoofing attack entails compromising a server in the DNS hierarchy or intercepting DNS requests (“man-in-the-middle” attack). Then, an attacker replies to incoming requests with fake responses like “IP address for nfs-serv.foo.com is 170.128.2.1” ; 170.128.2.1 being of course a machine under attacker’s control.

Another techniques often used by hacker on networks where routing advertisements are exchanged without authentication is send fake routing updates. The updates sent by the adversary will make the traffic go through a machine under his control.

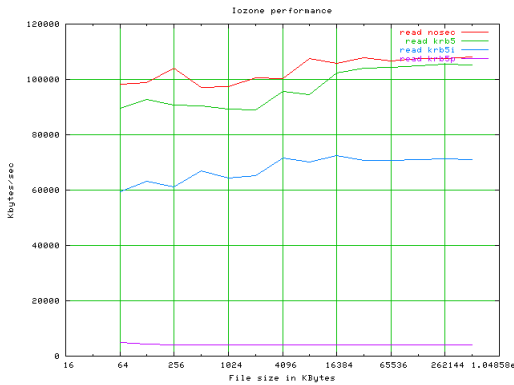
## A classical application

NFS is often used for centralized home directories. The user's home directories are saved on a server and they mount it with NFS from their workstation. A user can thus work on any workstation he want in the network. A major issue is that each home directory contain a sub-directory called `.ssh` which notably contain the RSA private key.

Therefore, when a user use SSH on his workstation, he thinks the traffic is secure even though the RSA private key was sent in-the-clear few minutes ago !

That is why home directories should be shared with `RPCSEC_GSS` in privacy mode - even if the performances are divided by 5 or 6.

Figure 5: The performances issue when authenticating (krb5) , protecting the integrity (krb5i) and especially encrypting (krb5p) the NFSv4 request. Here, a test on synchronous reads made with Iozone.



**Exposure level :** High

**Counter measures :**

- Setup authentication and encryption of NFSv4 traffic, via `RPCSEC_GSS` or an external protocol (SSH tunneling, IPsec...)
- Control who is connected on the LAN (wireless networks, laptops on Ethernet networks)
- Authenticate and protect integrity of routing advertisements (BGP and OSPF can be configured to mandate authentication)

## 4 Taking advantage of a NFSv4 server

We will depict two scenarios of how an adversary can take advantage of a NFSv4 server. The first one is when a non-privileged user on an NFSv4 server wants to take advantage of this account, either for privileges escalation or for attacking legitimate clients. The second scenario explains how a hacker can setup a malicious NFSv4 client in order to attack unfortunate clients.

### 4.1 The attacker owns a user account on an NFSv4 server

Here, the hacker can be an employee owning a user account on a server which runs a NFSv4 server. He may try to (from less to more risky) :

- Gathers information about the exported file-systems, the clients accessing the file systems... This can help him to setup attacks ; he can also discover weaknesses in the virtual export tree (each sub-tree can be exported to different hosts with different security flavors).
- Extracts private information (i.e. private keys, certificates) and possibly corrupts them.
- Exploit vulnerabilities in the NFSv4 commands to root the machine or read/write configuration files he has the right to access.

**Exposure level :** Low

**Counter measures :**

- Users owning account on servers running NFSv4 should be controlled. Such accesses should be logged to a remote log system.
- Rights on the configuration files and the commands should be checked. We checked that by default rights are fine.
- NFSv4 commands reachable by non-privilege users should be scanned for vulnerabilities by the community so as to prevent them from being used by a hacker to root the servers.

## 4.2 The attacker sets up a malicious NFSv4 server

Consider an attacker who can setup a NFSv4 server and who try to use it in order to attack the clients who connects on this server. This scenario applies to scenarios where NFSv4 servers are used as public files servers (like public FTP servers) and can be accessed by non-trusted users. In such a scenario, clients should be protected against malicious server.

The scenario also applied to attacks where the hacker redirect the client on a malicious NFSv4 server. Migration and replication capabilities will be integrated to GNU/Linux implementation of NFSv4 soon ; they will possibly give the possibility for an adversary to redirect a legitimate user to a malicious NFSv4 server.

The hacker can then exploit vulnerabilities on the client implementation to run a shellcode which spawns a root shell. He will then be able to run any root commands and install some backdoors for future access on the machine.

---

**Exposure level :** Low

**Counter measures :**

- Authentication of NFSv4 server (to avoid being redirected on a malicious one). Currently NFSv4 supports SPKM3 to authenticate both the client and the server with certificates.
  - Audit of NFSv4 client's source code by the community to patch the vulnerabilities that could be exploited by a malicious server.
- 

## Part II

# Security protocols in NFSv4

First of all, a WAN/Internet context implies that we cannot trust NFSv4 clients since the machine cannot be securely identified (IP address can easily be spoofed (i.e. IP spoofing [12], ARP spoofing, [6], ICMP redirect [8]...). Even when authentication of the machine is possible, it is still difficult to identify the physical user connected on the machine. Biometric authentication can help to solve this issue but it is not widely used so far. Moreover, we never know the intents of the user connected into the company's network, in particular when he is not employed by the company itself (i.e. WAN connecting business partners and customers).

NFSv4 comes with a mechanism for authentication, integrity protection and encryption of data ; this protocol works at the RPC level and is thus called RPCSEC\_GSS [13]. It is a common RPC security mechanism which uses a set of security flavors ; so far, Kerberos 5, SPKM-3 and LIPKEY are supported in the GNU/Linux implementation of NFSv4.

## 5 Unix/SYS authentication

### 5.1 Authentication scheme

With AUTHSYS authentication (RPC value for the security flavor is 1), a client has to send the following credential :

```
struct authsys_cred {
    uint32 stamp; /* an arbitrary value generated by the caller
(generally a timestamp) */
    string<255> hostname; /* name of the caller's machine */
    uint32 uid; /* caller's effective user ID */
    uint32 gid; /* caller's effective group ID */
    uint32<16> gids; /* a counted array of groups to which the caller
belongs */
```

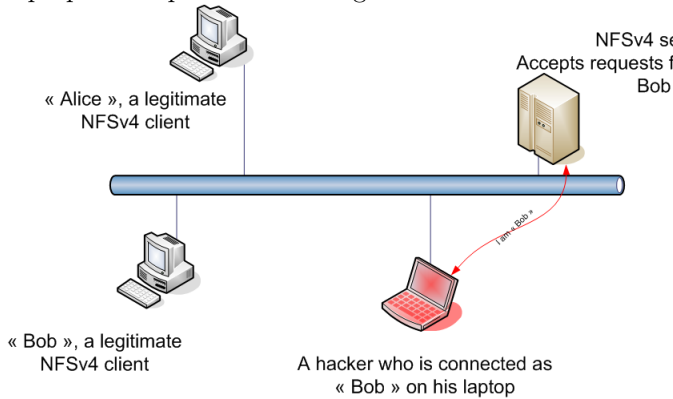
Figure 6: Credential used in SYS authentication

The hostname is not used for any security purpose ; the server obtains the hostname via the source IP address of the request. Spoofing the IP address is thus sufficient for a hacker willing to impersonate a machine.

The verifier accompanying the credential has AUTH\_NONE flavor. Since this verifier is null, a client can forge any values for UID/GID and the server has no way of knowing whether the values are legitimate or not.

## 5.2 Attacks

Figure 7: A hacker who is connected as “Bob” on his laptop can impersonate the legitimate user “Bob”



Unix authentication is in fact a very low security scheme which can easily be defeated by any experienced hacker. Among all user accounts on the machine, a password can loosely be cracked in few hours (brute force, dictionary, time-memory tradeoff attacks).

Then, if an attacker already owns an account on the client, he will certainly be able to switch to another account (attacks against password hashes, vulnerabilities in local programs...). When it is about network authentication (NIS [14], LDAP [15]...), some attacks can be launched by listening to the network (passive sniffing [16] [17], man-in-the-middle) or taking benefit from trust relations (via spoofing attacks).

A hacker can also generate its own NFSv4 request filling the UID, GID and GIDS fields with arbitrary values. We put this method to work using the ONC RPC module in SPIKE [3].

Last but not least, a network which is not well-protected against physical intrusions (insertion of a laptop, wireless networks) give the possibility for an adversary to plug his own machine and usurp anyone’s identity.

In brief, Unix authentication is not sufficient unless the network is well closed and guarded (Firewalls, IDS/IPS<sup>4</sup> like Snort [18]), and physical accesses to the desktop machines are well controlled.

*Attention should be given to “Security considerations” in RFC 3530 [19] stating that AUTH\_SYS flavor is neither mandatory nor recommended. This authentication is implemented in CITI’s NFSv4 for Linux but interoperability with other systems is not insured.*

<sup>4</sup>IDS : Intrusion Detection System - IPS : Intrusion Prevention System

Fortunately, NFSv4 can use stronger authentication protocols thanks to the RPCSEC\_GSS protocol [13]. Mandatory security flavors that should be included in RPCSEC\_GSS implementations are : Kerberos 5 , SPKM-3 and LIPKEY.

**Probability of attempt : High - Probability of success : High - Risks : High - Complexity : Low**

**Counter measures :**

- Mandate use of a more secure authentication.
- Apply protection to the network against intrusions, eve’s dropping, spoofing... This is limited to LAN networks and reaching an acceptable security level can be costly.
- Do both to fulfil the in-depth protection principles.

## 6 A brief overview of RPCSEC\_GSS

RPCSEC\_GSS [13] is a generic security protocol which aims at securing exchanges of RPC requests/responses. It defines three security modes : authentication (against impersonation), integrity (authentication as well as protection against malicious modification), privacy (integrity as well as protection against spying).

```
struct rpc_gss_cred_t {
    rpc_gss_proc_t gss_proc; /* control procedure (DATA, INIT,
CONTINUE_INIT or DESTROY) */
    unsigned int seq_num; /* sequence number up to 0x80000000 */
    rpc_gss_service_t service; /* service used : none (authentication),
integrity, privacy */
    opaque handle<>; /* context handle */
}
```

Figure 8: Credential of RPCSEC\_GSS authentication (version 1)

## 7 Kerberos to authenticate inside corporate networks

### 7.1 Principles

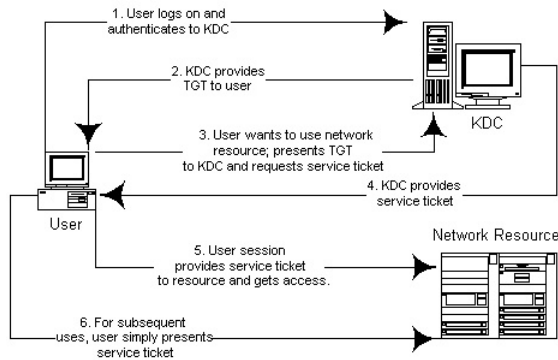
Kerberos [20] [21] makes use of a trusted third party, a Key Distribution Center (KDC), which consists of two logically separated parts: an Authentication Server (AS) and a Ticket Granting Server (TGS). Kerberos works on the basis of “tickets” which serve to prove the identity of users.

Kerberos maintains a database of secret keys; each entity on the network shares a secret key known only by itself and by Kerberos. Knowledge of this key serves to prove an entity's identity. For communication between two entities (a NFSv4 client/server), Kerberos generates a session key which they can use to secure their interactions.

The set of all nodes identified in a Kerberos domain is called a "realm". This is the main problem of Kerberos authentication. Indeed, the protocol in itself is considered to be secure, even if some weaknesses were found and first published in 1991 at the Usenix conference [22].

Nevertheless, there are some difficulties for scaling Kerberos to large networks [23]. As the number of nodes is growing, a single realm is not sufficient. When interconnecting different companies, each one having their own KDC, it becomes hard to establish the trusted relations.

Figure 9: Principles of Kerberos security architecture



There is no doubt that Kerberos will be largely used to authenticate NFSv4 peers in a corporate network. But since it does not scale well, NFSv4 should also make use of other security protocols which are better suited to WAN/Internet context (LIPKEY under SPKM-3).

## 8 SPKM-3 and LIPKEY to authenticate on interconnected networks

NFSv4 requires mandatory support for SPKM-3 and LIPKEY. At the time writing, NFSv4 support for those protocols has not been validated enough ; it is not to be used on operational files servers.

SPKM-3 describes a method for setting-up a secure channel with mutual authentication where both user and server authenticate with public-key certificates. SPKM-3 also describes a method for creating a secure channel where only the server authenticates with a public-key certificate, and the user is anonymous. LIPKEY then uses the SPKM-3 anonymous secure channel to authenticate a user with a password, completing the mutual authentication [24] [25].

Usage of public-key certificates and password will allow NFSv4 client and server to authenticate on Internet and WAN networks. As implementation becomes stable, a security analysis on the implementation will be carried out.

Each administrator should analyze how certificates are distributed among NFSv4. Indeed, initial distribution of certificates requires a network serving authentication and integrity properties. Else, an attacker could launch a man-in-the-middle attack and eventually substitutes a certificate by his own one.

On each node, storage of certificates should guarantee integrity (i.e. using software like Tripwire [26]). Otherwise an attacker could modify/replace a certificate thus breaking the authentication mechanism. This issue is not directly connected to NFSv4 itself, but one has to remember that the security level of an entire system is equal to the security level of its weakest component. No need for the implementation of SPKM-3 in NFSv4 to be highly secured if a hacker can easily corrupt a certificate !

**Probability of attempt :** Medium - **Probability of success :** Medium - **Risks :** High - **Complexity :** High

**Counter measures :**

- Exchange certificates on a secure channel or verify them with a secure channel (IPSec channel, face-to-face, phone...)
- Protect integrity of the certificates and monitor any undesirable modifications (Tripwire [26]...)
- As a general counter-measure, protect against intrusions on trusted NFSv4 clients (firewall, IPS...)

## 9 Side channel attacks

A definition of side channel attacks in cryptography is :

*A side channel attack is any attack based on information gained from the implementation of a cryptosystem, rather than theoretical weaknesses in the algorithms ("compare*

*cryptanalysis*”). For example, timing information, power consumption, electromagnetic emanations or even sound can provide an extra source of information which can be exploited to break the system.

Applied to our security researches on NFSv4, we can extend this definition to attacks based on any information we can gain from the environment which is not linked to theoretical weaknesses in the algorithms.

We can divide the way attackers retrieve side-channel information in three cases :

- **The attacker has a physical or non-privileged access** to the machine and can probe : CPU utilization of `nfsd` processes, memory usages, accessible files related to NFSv4... For instance, he can try to measure duration of CPU utilization peaks while providing arbitrary requests as input. Such measures can be used to evaluate the path followed in the open-source code. To be realistic, this is really unlikely to happen as it is hard to get such pertinent information in time-shared operating systems. That is why this class of attacks is mostly launched against embedded systems (some attacks are really awesome [27]). However, NFSv4 may be used on embedded systems and those attacks would need to be considered.
- **The attacker is able to spy exchanges between a client and the NFSv4 server** and, for instance, probe the response-time upon reception of a NFSv4 request. There are some examples of attacks [27] demonstrating that response-times possibly turn out to be exploitable information. However, there are little chance that such attacks can act up with NFSv4.
- **Attacker can send arbitrary requests to NFSv4 server and get the responses in return.** Errors sent back on the network when authentication, integrity or privacy fails can constitute a relevant source of information. Regarding authentication, a basic application is to use NFSv4 server as an oracle which informs us if an arbitrary authentication key is valid. Indeed, **there is no delay in the transmission of an error following a bad authentication.** This can be considered as a weakness in `RPCSEC_GSS` authentication.

## 10 Attacks against protocol concepts

### 10.1 Minor versioning

If versions are badly managed by NFSv4 administrator, an attacker could force the server to downgrade to an unsecure minor version ; he could thus negotiate a version which has known security vulnerabilities.

Concerning minor version management, the more secure - but more restrictive - rule is to limit negotiation to latest versions with no discovered security vulnerabilities. Of course, each client should support the latest versions.

### 10.2 Security negotiation (SECINFO)

RFC 3530 [19] warns of a weakness in security negotiation with the SECINFO procedure - and the new `SECINFO_NO_NAME` as defined in M.Eisler's draft [28]. Without security protection encapsulating SECINFO and its results, an attacker in the middle could modify results such that the client might select a weaker algorithm in the set allowed by the server, making the client and/or server vulnerable to further attacks.

### 10.3 Migration / Replication (attack on `fs_locations`)

A security analysis will need to be done when the migration and replication capabilities will be implemented in Linux NFSv4 implementation.

# Theoretical audit of RPCSEC\_GSS

RPCSEC\_GSS guarantees authentication, integrity and/or privacy of RPC requests and responses. More details can be found in RFC 2203 [29] and for use with Kerberos 5 in RFC 1964 [21].

Our study is focused on RPCSEC\_GSS with Kerberos 5 but same concepts can be applied to other security flavors.

Exchanges between the client and the KDC<sup>5</sup> are out of the scope of this study.

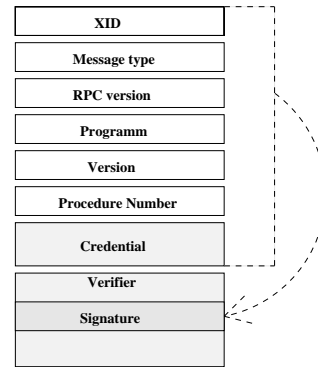
Likewise, we consider attacks on NFSv4 requests/responses protected with RPCSEC\_GSS per-message protection, once the context has been setup. Study of possible attacks on the context setup is important but out of scope of the present paper.

Therefore, we assume that each NFSv4 peers share the same context and in particular the private encryption key. They want to exchange data on a insecure channel and use their shared secret to protect them against several adversary classes, given a limited computing power.

## 11 Security modes in RPCSEC\_GSS

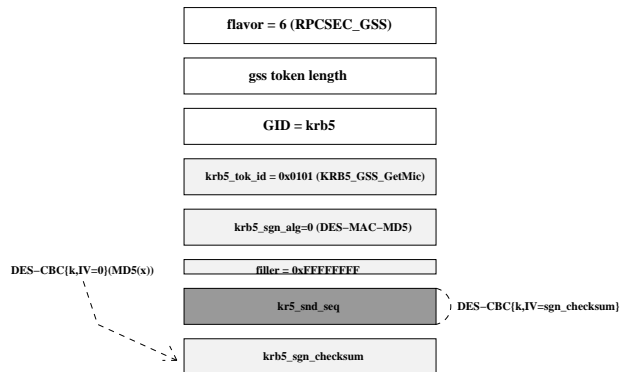
RPCSEC\_GSS defines three modes of protection called : authentication, integrity and privacy.

Figure 10: Authentication mode (signature covers the RPC header)



Authentication mode consists of a MAC computed on the RPC header (from the XID field to the credential). So it just guarantees that the message was originally sent by the legitimate user. It does not prevent against any modification of the NFSv4 data : if an attacker drops the request and prevents its delivery, he can send arbitrary NFSv4 operations using this valid MAC.

Figure 11: Kerberos 5 - RPCSEC\_GSS verifier



Authentication mode makes use of a verifier mainly composed of :

- an encrypted sequence number to prevent replay attacks

<sup>5</sup>In cryptography, a key distribution center (KDC) is part of a cryptosystem intended to reduce the risks inherent in exchanging keys. KDCs often operate in systems within which some users may have permission to use certain services at some times and not at others.

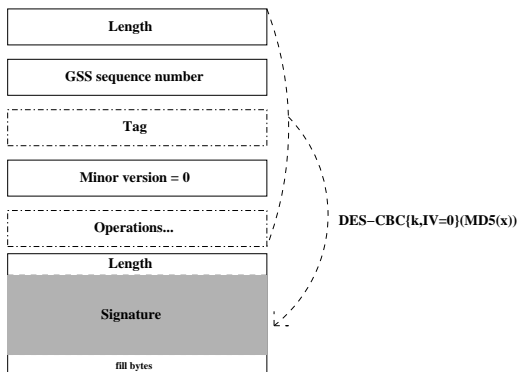
- a MAC improperly called checksum in the specification. Generally, a checksum is a redundancy check for protecting the integrity of data by detecting errors in the data that is sent through a channel. If it really was a checksum, an attacker could compute the checksum of any arbitrary message. The difference is that checksums protect against modifications by a random noise whereas MAC protects from *malicious* modifications.

### Threats

- Attacks against the MAC (detailed later on) : an adversary wants to forge a valid MAC for a new message  $M$  or wants to find a valid pair  $(M, c)$  with some given properties on  $M$ .
- Replay attacks : an adversary has seen a pair  $(M, c)$  where  $c$  is the MAC for  $M$ . He wants to replay  $(M, c)$  and hopefully execute the NFSv4 request a second time. This makes sense for all non-idempotent requests (create, remove, rename...).
- Swap : an adversary swap  $(M_1, c_1)$  and  $(M_2, c_2)$ . The server first receives  $(M_2, c_2)$  and then  $(M_1, c_1)$ .
- Erase : an adversary erases  $(M, c)$

## 11.2 Integrity

Figure 12: RPCSEC\_GSS integrity mode (GSS data + GSS checksum)



Integrity mode comprises the RPC credential/verifier as for authentication plus a MAC computed on NFSv4 data and some GSS values. If an attacker corrupts NFSv4 data, the MAC becomes invalid.

Here also, a sequence number is injected into the MAC's input to prevent replay attacks.

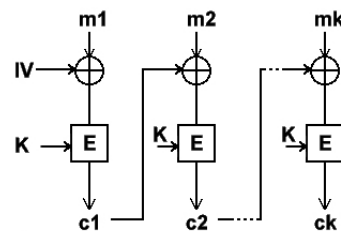
**Threats** Same as for authentication.

## 11.3 Privacy

Privacy mode keeps the RPC credential/verifier for authentication purpose. In addition, NFSv4 data along with associated integrity check quantities are encrypted so as to guarantee integrity and confidentiality of the NFSv4 requests/responses. Only DES-CBC encryption (with a zero IV) is available so far in RPCSEC\_GSS with Kerberos 5. Plaintext is padded to the next highest multiple of 8 bytes.

Figure 13: CBC mode

$$M = m_1 + m_2 + \dots + m_k$$



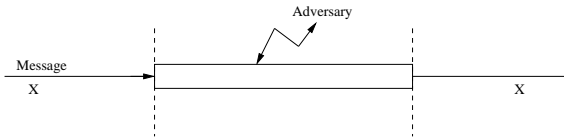
We notice that RPC headers are not encrypted : sending the headers in-the-clear does not lack sensible information and encryption of RPC headers would have required revising the RPC protocol.

**Threats** There are some weaknesses in the CBC mode :

- Information leakage by first block collisions : if for two different plaintexts the first blocks are the same and the IV is fixed (here it is zero), there is a leakage of the equality of these blocks. The first block will be the length of the token and the GSS\_Sequence\_Number. As the sequence number is incremented at each request, there is no risk of first block collision.
- Integrity issue : a third party can replace ciphertext blocks so that all but a few will decrypt well. There is no risk since a decryption error on a single block invalidates the whole NFSv4 request.

### 11.3.1 Cryptographic model

We recall definition of authentication, integrity and privacy in cryptography :



- Message authentication : we make sure about who sent the message
- Message integrity : we make sure that the received message is equal to the sent one
- Confidentially (privacy) : the adversary should not get any information on  $X$

In cryptography, message authentication is stronger than message integrity. Therefore, cryptographic definition differs from RPCSEC\_GSS terminology where authentication mode is weaker than integrity mode. RPCSEC\_GSS's authentication make sure that the message was **originally** sent by the right person - not necessarily the **received** message.

In fact, authentication in RPCSEC\_GSS is like a regular message authentication if we consider  $X$  as the RPC header excluding the payload (i.e. NFSv4 data). One should keep this in mind when choosing to use this mode of security.

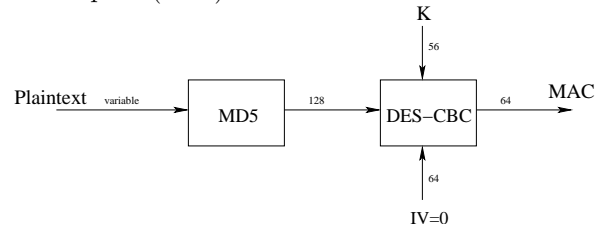
### 11.4 Message Authentication Code (MAC)

In the present paper, this cryptographic function is written  $c = MAC_K(M)$  where  $c$  is the  $MAC$  of  $M$  and  $K$  is the secret key, shared by the client and the server.

Currently, there is only one MAC algorithm in RPCSEC\_GSS. It is computed with DES MAC MD5 algorithm which means that the plaintext is first hashed with MD5 and then encrypted with DES in CBC mode employing the secret key (i.e. the Kerberos session-key) and a zero IV<sup>6</sup>.

<sup>6</sup>In cryptography, an initialization vector (IV) is a block of bits that is required to allow a stream cipher or a block cipher executed in any of several streaming modes of operation to produce a unique stream independent from other streams produced by the same encryption key, without having to go through a re-keying process. Here, the signature algorithm does not make use of the IV.

Figure 14: A MAC based on a hash function (MD5) and a block cipher (DES).



The MAC scheme is fairly simple and works as follows : the client compute the encrypted hash of the header  $x$ , that is  $c = MAC_k(x)$  and sends  $x|d|c$  to the server. When the latter receives the request  $x_r|d_r|c_r$ , he computes  $c'_r = MAC_k(x_r)$  and checks that  $c'_r = c_r$ .

### 11.5 Attacks against MAC (authentication - integrity)

Authentication guarantees that the user who sends the data is really the user he pretends to be. This is generally done with MAC. MAC prevents from impersonation : since IP addresses can easily be spoofed, we cannot rely on this information contrary to what was done in latest NFS versions ("Security considerations" from RFC 3530 [19]). Authentication also prevents an attacker from hijacking the TCP session used by NFSv4 to carry the RPC requests/responses.

To attack a MAC algorithm, an attacker has several axes of research. First of all, he can try to retrieve the private key (Kerberos session key) shared between the client and the server. That requires an access on either the client or the server to extract the session key (which is loaded in memory).

Weaknesses in the MAC scheme can be used to deduce information about the key thanks to several spied NFSv4 headers and their MAC.

An attacker can also exploit collisions on the hash function. A quantitative analysis is done later on to evaluate the risks of collisions exploitation.

#### 11.5.1 Description of hash-function attacks

We explain here the possible attacks against a hash function, which is the first block of our signature algorithm. The generic attacking model is described with the authentication scheme (MAC on the RPC header) though they also apply to the integrity scheme (MAC of RPC procedure arguments, that is NFSv4 data).

We consider only the first part of the MAC scheme : the hash function. Let  $x$  be the RPC header, we call  $y = H(x)$  the hashed value.

They are three basic attacks on a hash function :

- **collision** : find  $x, x'$  s.t.  $H(x) = y$  and  $H(x') = y$ , that is, find two NFSv4 headers which result in the same signature. Let  $x$  be an arbitrary byte stream, one can show - thanks to the Birthday Paradox- that a collision can be found in  $\sqrt{\text{card}(Y)}$  hashes. The great difficulty here is to find  $x, x'$  s.t.  $x$  and  $x'$  are two valid RPC headers.
- **first preimage attack** : given  $y$ , find  $x$  s.t.  $H(x) = y$ .

For a one-way function with no weaknesses, this attack requires computation of  $\text{card}(Y)$  hashes. Usually,  $\text{card}(X) > \text{card}(Y)$ , and there is a good chance that several preimages exist for  $Y$ .

Applied to RPC header, the complexity would be higher than  $\text{card}(Y)$  since the set  $X$  of  $x$  is restricted to valid RPC headers. But since  $x$  is here the header of the RPC request, it is not of interest : the RPC header is sent on-the-clear with all services (none, integrity, privacy).

- **second preimage attack** : given  $x$  s.t.  $H(x) = y$ , find  $x'$  s.t.  $H(x') = y$ . An attacker sniffing a RPC request/response  $x|d|y$  can thus substitute it by  $x'|d|y$ . The receiver computes  $H(x') = y = H(x)$ , hence accepting the authentication. For a secure one-way function, the attack requires  $\text{card}(Y)$  hashes. In practice, it may be hard to master a second preimage attack because the set  $X'$  of  $x'$  is limited to valid RPC headers.

Currently, the hash function used in RPCSEC\_GSS Krb5 is MD5. There are known weaknesses as detailed in {Improved collision Attack on MD5} which allows an attacker to find collisions complexity

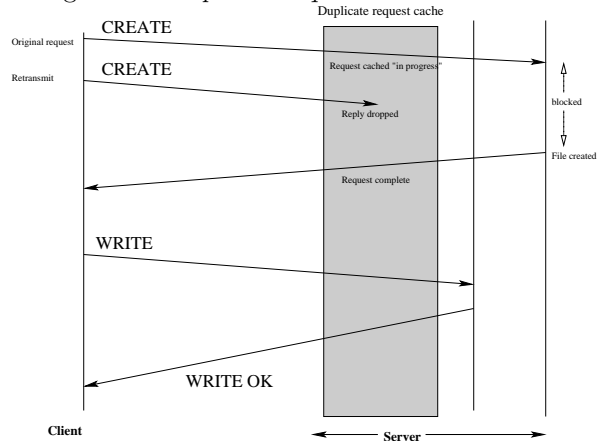
### 11.5.2 Attacks against authentication mode

When authentication mode is used to authenticate NFSv4 peers, the receiver knows for sure who has sent this request since the RPC header is hash-signed with the session-key. The private key is shared by the client and the server and has to be setup beforehand. For example, with Kerberos 5, the session-key is exchanged within the Kerberos ticket.

**Replay-attacks and on-the-fly corruption** Authentication mode is pretty weak : an attacker can still intercept the request, change NFSv4 payload and relay it to the server. In fact, if he just keeps the same RPC header the signature remains valid. Keeping the same RPC header is not a constraint since, except for the credential and the procedure (NULL - COMPOUND), all headers used by NFSv4 are always the same.

However, a mechanism of *duplicate request cache* on the server, prevents a request which has already been received to be executed a second time. The *duplicate request cache* eliminates the risk of non-idempotent<sup>7</sup> requests being executed several times (due to lost replies). The cache is keyed by the XID and that is why a replay attack will not work as-is since the duplicated request will either be dropped or the response to the first request will be sent back again.

Figure 15: *Duplicate request cache mechanism*

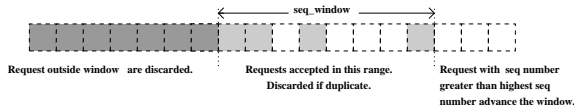


The authentication scheme limits the scope of replay attacks with a sequence number in the credential : **GSS sequence number**. A valid couple (RPC header, MAC) cannot be used outside the server's window. It should be within  $[N - \text{seqwindow} + 1; N]$  ; where  $N$  is the last sequence number and **seqwindow** the size of the acceptance window. A request is only accepted once with a given GSS sequence number.

In brief, if an attacker intercepts a request and prevents its delivery, he will have a short time to modify NFSv4 data and relay it to the server (before a retransmission timeout is triggered on the client).

<sup>7</sup>An idempotent operation is one that returns the same result if it is repeated.

Figure 16: RPCSEC\_GSS sequence window on the server



### Exploitation of collisions on the hash function

About collisions on the hash function, the search of a second preimage for a valid signature is very unlikely to success. Moreover, exploitation can only be done when the GSS sequence number of this second preimage is within the server's window as well as the RPC sequence number (XID). However, the GSS sequence number are easily predictable : they are sent in-the-clear as part of the credential and are simply incremented by one for each new request.

Among the RPC fields feeding the hash function, the only variable field is XID. We can also consider the credential's GSS sequence number to be a variable field with care that it is incremented from a deterministic initial value <sup>8</sup>.

RFC 1831 [30] does not specify a domain value for XID, so it goes from 0x0 to 0xFFFFFFFF. RFC 2203 [29] specifies that the credential's GSS sequence number is bounded to 0x80000000. There are 63 variable bits on 192 (RPC fields)+ 256 (credential) which makes a set of  $2^{63}$  valid RPC headers.

### Application to current Krb5 - RPCSEC\_GSS implementation

Let us apply the collision search to the current implementation. The first step of the MAC is a MD5 hashing function  $H : X \rightarrow Y$  mapping 448 bits to 128 bits where  $card(X) = 2^{63}$  and  $card(Y) = 2^{128}$ . The second step is a DES-CBC encryption function  $E : Y, K \rightarrow Z$  mapping 128 bits of plaintext and a 56-bit key to 64 bits of ciphertext. We can approximate this scheme by a global function  $MAC : (X, K) \rightarrow Z$  where  $MAC = H \circ E$  uniformly mapping 448 bits to 64 bits with  $card(X) = 2^{63}$  and  $card(Z) = 2^{64}$ .

There are three possible collisions search :

1. Collision search on the hash function  $H$
2. Collision search on the encryption function  $E_k$
3. Collision search on an entire  $MAC_k$  function (a collision on either  $H$  or  $E_k$ )

<sup>8</sup>The initial GSS sequence number is the four-byte sequence number from the sender followed by 4 bytes (0x0 or 0xFF) which is DES-CBC encrypted with the context key and an IV (formed from the first 8 bytes of sgn\_cksum).

A first trivial statement is that for a given key  $k$  the probability of collision on  $E_k$  (2) is :

$$\Pr\{\exists(y, y') \in Y / E_k(y) = E_k(y')\} = 1 \text{ since } card(Z) > card(Y)$$

This proves that collisions exist on the encryption function. But a collision search requires knowledge of the secret key  $k$ . Therefore, an adversary willing to find a collision on  $E_k$  should spy some RPC headers and their associated MAC. He stores all the MACs in a table and when two are equals, he computes the hashes. If there are equals, he has discovered a collision on  $E_k$ .

For non-trivial cases, when the input space is smaller than the output space, we can compute the probability that there is at least one collision on a uniform function, let's say  $f$ . Let's start computing the probability that there is no collision on  $f$  :

$$\begin{aligned} & \Pr\{\neg\exists(x, x') \in X / f(x) = f(x')\} \\ &= \frac{|Y|}{|Y|} \cdot \frac{|Y|-1}{|Y|} \dots \frac{|Y|-|X|+1}{|Y|} \\ &= \prod_{i=0}^{|X|-1} \frac{|Y|-i}{|Y|} \\ &= \prod_{i=0}^{|X|-1} 1 - \frac{i}{|Y|} \end{aligned}$$

We can use the Taylor series expansion of the exponential function :

$$\exp(x) = 1 + x + \frac{x^2}{2!} + \dots$$

Thus :

$$\begin{aligned} & \Pr\{\neg\exists(x, x') \in X / f(x) = f(x')\} \\ & \approx e^{-\frac{1}{|Y|}} \cdot e^{-\frac{2}{|Y|}} \dots e^{-\frac{|X|-1}{|Y|}} \\ &= e^{-\frac{1 \cdot |X| \cdot (|X|-1)}{2 \cdot |Y|}} \\ & \approx e^{-\frac{|X|^2}{2 \cdot |Y|}} \end{aligned}$$

And so :

$$\Pr\{\exists(x, x') \in X / f(x) = f(x')\} \approx 1 - e^{-\frac{|X|^2}{2 \cdot |Y|}}$$

We can now compute the average number of tries (n) we need to get a collision with probability at least  $\alpha$ .

$$e^{-\frac{n(n-1)}{2 \cdot |Y|}} > \alpha$$

$$n^2 - n < 2|Y| \cdot \ln(1/\alpha)$$

Applied to the MAC function (3),  $MAC_k$ , which outputs 64 bits ( $|Y|=2^{64}$ ). With a probability  $\alpha=1/2$  to get a collision, the lower bound for n satisfies :

$$n^2 - n = 2 \cdot 2^{64} \cdot \ln(2)$$

$$n^2 - n - 2^{65} \cdot \ln(2) = 0$$

$$\Delta = 1 - 4 \cdot 2^{65} \ln(2)$$

$$n = \frac{-1 + \sqrt{1 - 4 \cdot 2^{65} \ln(2)}}{2} \approx 2^{32.236}$$

And so, a collision on G can be found in  $2^{32}$  MAC operations.

Applied to the hash function, H, which outputs 128 bits, we need about  $2^{64}$  hashes to find a collision. Since we can only change  $2^{63}$  bits on the input, the probability of getting a collision with this set of  $2^{63}$  headers is :

$$\Pr\{\exists(x, x') \in X / H(x) = H(x')\}$$

$$\approx 1 - e^{-\frac{|X|^2}{2 \cdot |Y|}}$$

$$\approx 1 - e^{-\frac{2^{126}}{2 \cdot 2^{128}}} \approx 0.118$$

Two headers which collides on the hash function will results in two equal  $MAC_k$  for any secret key  $k$ , which is of course more interesting than a collision on a key-dependent function.

In brief :

- If an attacker spies  $2^{32}$  requests, he has one chance out of two to discover a collision on G. The amount of needed requests is much too large since the Kerberos session key would have expired previously ; thus G will not remain the same.
- If an attacker generates  $\sqrt{2^{128}} = 2^{64}$  requests and their hashes, he has one chance out of two to find a collision on the hash function. This is hard but feasible and **really interesting** : the collision can be exploited during any NFSv4 session and, with chances, there will be a time when the **GSS sequence number** and the **XID** will be accepted by the server.

### 11.5.3 Attacks against integrity mode

In addition to authenticated the RPC header, a client using integrity mode also computes a MAC on the procedure parameters, which are in fact NFSv4 data. Therefore, the request carries two MACs : one covering RPC header and the other covering NFSv4 data. This prevents corruption of NFSv4 data ; if some fields are modified the signature will not remain valid.

Any request modifying the filesystem (files and meta-data) on the server should not be corrupted ; integrity is essential for most read-write file systems accessed outside a well-controlled network.

In a smaller extent, requests targeting read-only file systems should be protected against corruption in specific cases. For instance, a hacker can try to modify authenticated NFSv4 requests and eventually read some specific files on the remote file system. Typically, if users' home directories are exported with NFSv4, an attacker may try to read the file in which the user's private RSA key is saved (.ssh/id\_rsa).

**Replay attacks** During a session, a MD5 hash of NFSv4 data is always DES-CBC encrypted with the same private key K and with a fixed IV (zero). Fortunately, the GSS sequence number play the part of an IV : the ciphertext will be changed even if the NFSv4 operations and its parameters remain the same.

Therefore, a replay attack cannot be set up directly.

**Collision search** As in authentication mode, we first have to enumerate the variable fields in NFSv4 data so as to estimate the probability for a collision attack to success.

## 11.6 Attacks against encryption (privacy mode)

### 11.6.1 Attacks against DES-encrypted traffic

RFC 3530 [19] (NFSv4) specifies that the encryption should be 56-bit DES **which is not considered secure**. Indeed, a 56-bit private key is not enough regarding current computing powers (an exhaustive search is feasible). State of art in terms of cryptography recommends usage of 96-bit or preferably 128-bit keys. In NFSv4 for GNU/Linux implementation, only 56-bit DES is available so far.

With a 56-bit private key, one can retrieve the key in at worst  $2^{56}$  encryptions with an exhaustive search. Is the complexity high enough to be well-protected ? Not really. In 1998, using custom-designed chips and a personal computer, the Electronic Frontier Foundation built "DES Cracker" [31]. Costing less than \$250,000 and taking less than one year to build, DES Cracker broke a DES-encrypted message in 56 hours. In fact, this machine can search 88-billion keys per second. And the machine scales : doubling the number of chips, DES private key can be retrieved in half the time. There is no doubt that 8 years later (2006), a hacker should now crack DES within few hours.

Moreover, some weaknesses have been found in DES allowing to crack it faster than an exhaustive search [32]

; the private key can be retrieved with  $2^{47}$  known plain-texts using differential cryptanalysis or even  $2^{43}$  using linear cryptanalysis. However, known plain-text attacks are mainly theoretical : it is easier to run an exhaustive search on the key knowing one NFSv4 request and its encrypted value than sniffing  $2^{43}$  requests !

---

**Algorithm 1** Exhaustive search of DES key

---

**Input** :  $y$ , an encrypted NFSv4 request/response,  $m$  the plain-text  
a set of possible keys  $\mathcal{K} = \{k_1..k_N\}$   
**Output** : the secret DES key  
pick a random permutation  $\sigma$  of  $1..N$   
for all  $i=1$  to  $2^{56}$   
    if  $DES^{-1}_{k_{\sigma(i)}}(y) = m$  then yield( $k_{\sigma(i)}$ )  
end for

---

**Setting up a brute force attack with guessed parts of plain-text** So, how to setup a bruteforce attack on DES key ? The exhaustive search (*Algorithm 1*) cannot be applied directly since it requires to have a plain-text and a cipher-text request. Though one can only sniff the encrypted NFSv4 request, he can guess some fields in the plain-text (*Algorithm 3*). With the cipher-text and those chunks of plain-text, we demonstrate it is enough to run an exhaustive search (*Algorithm 2*) and find the right DES private key with very high probability.

---

**Algorithm 2** Exhaustive search of DES key when the plaintext is unknown

---

**Input** :  $y$ , an encrypted NFSv4 request/response,  $m$  the plain-text  
a set of possible keys  $\mathcal{K} = \{k_1..k_N\}$   
**Output** : the secret DES key  
pick a random permutation  $\sigma$  of  $1..N$   
for all  $i=1$  to  $2^{56}$   
     $\mathcal{O}(DES^{-1}_{k_{\sigma(i)}}(y))$  then yield( $k_{\sigma(i)}$ )  
end for

---

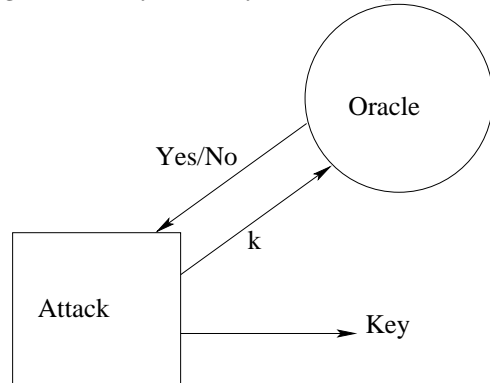
Some GSSRPC\_SEC and NFSv4 fields are fixed or bounded to a small set of values. Following reasoning is done on COMPOUND requests, which are the most frequent procedures used by NFSv4.

- length (32 bits) : equal to the NFSv4 data length
- GSS Sequence Number : bounded to 0x80000000
- tag : always set to 0
- minor version (32 bits) : currently, it is always set to 0

- operations count (32 bits) : the number of operations compounded will generally vary from 1 to 5. To be sure, let's say from 0x00000000 to 0x00000008 which makes 28 fixed bits instead of 32.
- opcode (32 bits) : there are one or several opcodes and maybe arguments in-between. To be sure, we only consider the first opcode. Opcodes go from 0 to 39 (0b00100111), which makes 26 fixed bits.

In total, there is  $32 + 32 + 28 + 26 = 118$  guessable bits.  $\mathcal{O}$  will return whether length, minor version, operations count and opcode fields match.

Figure 17: Key recovery with a stop test oracle




---

**Algorithm 3** an oracle  $\mathcal{O}$

---

input :  $x[0..l]$ , the NFSv4 request/response  
output : true if  $x$  is valid, false otherwise  
if  $x[0..32] \neq \frac{l}{8}$  return false;  
elseif  $x[64..96] \neq 0$  return false;  
elseif  $x[96..128] \neq 0$  return false;  
elseif  $x[128..160] > 8$  return false;  
elseif  $x[160..192] > 39$  return false;  
return true;

---

There is still to prove that if the oracle outputs true, there are good chances we have found the right DES key.

Let us call  $E$  the DES encryption function and assume that  $E$  uniformly maps on the output space.

Let  $x_1|x_2 \in (X_1, X_2) = X$  and  $x_1|x'_2 \in (X_1, X_2) = X$ .

$x_1$  is the known part of the plain-text and  $y$  is the cipher-text. The probability that there exists a key  $k'$  s.t.  $E_{k'}(x_1 + x_2) = E_{k'}(x_1 + x'_2) = y$  with  $x_2 \neq x'_2$  is :

$$\Pr\{\exists k' \in K, \exists x'_2 \in X_2 / E_{k'}(x_1 + x'_2) = y\}$$

$$= \Pr\{(\exists k' \in K / E_{k'}(x) = y) (\exists x'_2 \in X_2 / x = x_1 + x'_2)\}$$

We have :

$$Pr\{\exists k' \in K/E_{k'}(x) = y\}$$

$$= \sum_{i \in K} Pr\{E_i(x) = y\}$$

$$= \frac{|K|}{|Y|}$$

And for a random x :

$$Pr(\exists x'_2/x = x_1 + x'_2)$$

$$= Pr(\exists x'_2/x \in [x_1, x_1 + \max(X_2)])$$

$$= \frac{|X_2|}{|X_1|+|X_2|}$$

Hence :

$$Pr\{\exists k' \in K, \exists x'_2 \in X_2/E_{k'}(x_1|x'_2) = y\}$$

$$= \frac{|K|}{|Y|} \cdot \frac{|X_2|}{|X_1|+|X_2|}$$

$$= \frac{|K|}{|Y|} \cdot \frac{|X_2|}{|X|}$$

Therefore, we need to have  $|K| \ll \text{size}(x_1)$  that is we need to know more than 56 bits in the plain-text.

Applied to  $|K| = 2^{56}$ ,  $|Y| = |X| = 2^{512}$  (512 as an average size),  $|X_1| = 2^{118}$ ,  $|X_2| = 2^{118}$ ,

$$Pr\{\exists k' \in K, \exists x'_2 \in X_2/E_{k'}(x_1|x'_2) = y\} = 2^{-850} \approx 0$$

In at worst  $\theta(2^{56})$  and in average  $\theta(2^{55})$  tries, an attacker will retrieve the correct key ; an exhaustive search will always work even with those only chunks of plain-text.

## 11.7 Increasing the security level

RFC 3530 [19] recommends implementors and users to migrate to AES [33]. AES is safer and private-keys have a length of 128, 192 or 256 bits.

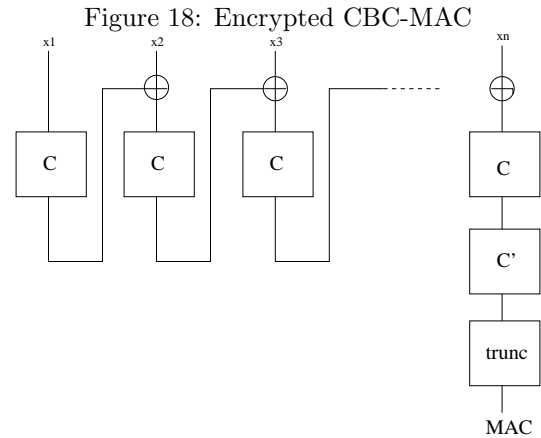
In CITI's NFSv4 implementation for Linux, the only encryption available so far is DES-CBC. It is planned to add support in RPCSEC\_GSS for SHA1 (instead of MD5) and 3-DES (instead of single DES) - as supported in the GSS API [34]. The checksum size will be increased to 160 bits ; the complexity of a collision attack will go up to  $\theta(2^{80})$  which is reasonably secure.

As for the MAC scheme, a fundamental concept in cryptography states that hashing encrypted data is more secure than encrypting hashed data. NFSv4 adopted the less secure solution (but also the less costly in terms of computation time) : first hashing the data with MD5 and then encrypting the hash with DES in CBC mode. Therefore, an attacker can find collisions on MD5 hash

functions without knowledge of the private key. Moreover, possible collisions on the hash function can be exploited in any NFSv4 session : they does not depend on the private key.

It would be better to use a MAC algorithm which takes the key into account (a CBC-MAC based on AES for instance). Computation cost will be increased since it requires to encrypt the whole data instead of the 128 hashed-bits but one will not be able to find collisions without the private key.

The scheme should be chosen with care since a simple CBC mode would be weak. The ISO/IEC 9797 standard defines a secure way to build a MAC from a block cipher (such as DES or AES). This scheme could be used in next versions of RPCSEC\_GSS.



## 11.8 Other attacks

### 11.8.1 Deny of service using sequence number

A deny of service attack using RPCSEC\_GSS sequence numbers above the server's window is documented in RFC 2203 [29] (7.2 "Sequence number"). The sequence numbers mechanism is briefly explained in the present report *on page 13*. With Kerberos 5 GSS-API, one can consider that the risk is low since, without the private key, the attacker cannot generate arbitrary sequence numbers. In fact, a Deny of Service (DoS) attack can still work : the attacker simply generates random sequence numbers and a part of them will be above the window, thus being accepted by the server and consuming some time for verification of RPC header's MAC.

### 11.8.2 Message stealing attacks

RPCSEC\_GSS does not address attacks where an attacker can block or steal messages without being detected by the server. To implement such a protection, the RPC protocol would not be stateless anymore.

Consequently, an attacker can :

- Steal requests : the legitimate client will retransmit after a timeout. The server will not detect the loss of the first request. Excluding an increase of the delay, the attack is painless.
- Steal responses : the legitimate client will retransmit the request after a timeout. Thanks to its duplicate requests cache, the server will normally send back the response without executing the request. This is particularly important for non-idempotent request.

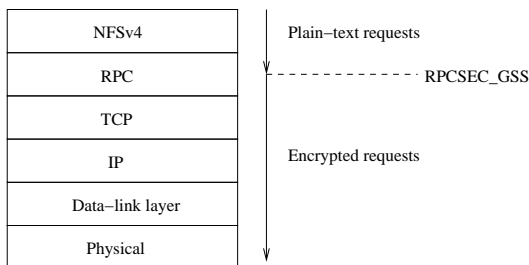
## 11.9 Security at IP or TCP layer

We compare use of RPCSEC\_GSS for authentication and encryption of RPC traffic with protocol like SSH or IPSec which authenticates and encrypts respectively TCP or IP traffic.

### 11.9.1 RPCSEC\_GSS

Except setting up the key architecture, using RPCSEC\_GSS is setting an option when mounting or exporting filesystems. Level of protection is satisfactory but - so far - there is no possibility to choose better security algorithms ; this lacks when it is about exchanging sensitive data. A major advantage of this option is that there is one secure connection per user.

Figure 19: Encryption with RPCSEC\_GSS



### 11.9.2 SSH Tunnel

Using SSH Tunneling is simple ; we build a SSH tunnel and redirect NFSv4 port (i.e. 2049) into the tunnel. Security can be high thanks to algorithm negotiation. A major drawback is that there is no way for a client to access several NFSv4 servers, except if they are all behind the other side of the tunnel. Then, when several users on a machine share the same SSH tunnel, RPCSEC\_GSS authentication is recommended : SSH authenticates the machines, not the hosts.

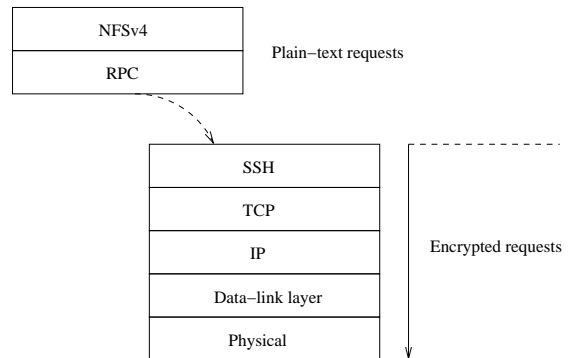
To setup a SSH tunnel with OpenSSH and redirect NFSv4 traffic, one can issue the following command on the client :

```
ssh -L 2049:<nfsv4_server>:2049 -l <user> -N <nfsv4_server>
```

where `user` is your login and `nfsv4_server` is the NFSv4 server's name or IP address.

Then, take care that instead of mounting from the server, you must mount from localhost.

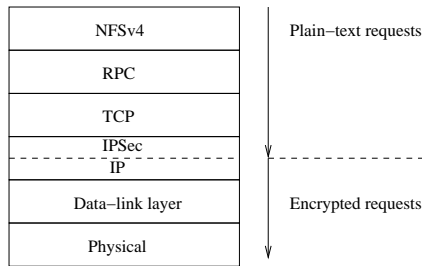
Figure 20: Encryption with SSH



### 11.9.3 IPSec

Another way to secure RPC exchanges is to encapsulate the underlying TCP connexion in an IPSec tunnel. Algorithms can be negotiated so as to reach the desired security level. Likewise SSH Tunnel, drawbacks are that several users sharing the same IPSec tunnel cannot be distinguished. But contrary to SSH Tunelling, a client can access several NFSv4 servers since sessions are identified by their IP destination address.

Figure 21: Encryption with IPSec



## 11.10 Comparison with Samba

### 11.10.1 Samba

Samba is a free software implementation of Microsoft's networking system released under the GNU General Public License. As of version 3, Samba not only provides file and print services for various Microsoft Windows clients but can also integrate with a Windows Server domain, either as a Primary Domain Controller (PDC) or as a Domain Member. It can also be part of an Active Directory domain.

Samba runs on most Unix and Unix-like systems, such as GNU/Linux, Solaris, and the BSD variants, including Apple's Mac OS X Server (it was added to the OS X workstation edition with version 10.2). It is standard on virtually all distributions of Linux and is commonly included as a basic system service on other Unix-based systems as well.

### 11.10.2 Security in Samba

There are several approaches for securing Samba :

**User-level security** : In User Level Security, the client sends a session setup request directly following protocol negotiation. This request provides a username and password. The server can either accept or reject that username/password combination. At this stage the server has no idea which share the client will eventually try to connect to, so it can't base the accept/reject on anything other than:

1. the username/password
2. the name of the client machine.

If the server accepts the username/password then the client expects to be able to mount shares (using a tree connection) without specifying a password. It expects that all access rights will be as the username/password specified in the session setup.

**Share-level security** : In Share Level security, the client authenticates itself separately for each share. It sends a password along with each tree connection (share mount). It does not explicitly send a username with this operation. The client expects a password to be associated with each share, independent of the user. This means that Samba has to work out which username the client probably wants to use. It is never explicitly sent the username.

*There is no mechanism like that in NFSv4.*

**Domain security** : provides a mechanism for storing all user and group accounts in a central, shared, account repository. Repository is shared between domain (security) controllers. Servers that act as domain controllers provide authentication and validation services to all machines that participate in the security context for the domain.

*This is now obsoleted in favor of ADS.*

**ADS security (Active Directory)** : Samba can join an Active Directory domain using NT4 style RPC based security, if the domain is run in native mode. Microsoft ADS allows to use a Kerberos-based security architecture.

*This is similar to what is done in NFSv4 when using `umich_ldap` for name-to-ID mapping and Kerberos 5 as `RPCSEC_GSS` flavor.*

### 11.10.3 User-level security v.s. Unix authentication

One should realize that, with the SMB protocol, the password is passed over the network either in plain-text or encrypted, but not both in the same authentication request.

When encrypted passwords are used, a password that has been entered by the user is encrypted in two ways:

- An MD4 hash of the unicode of the password string. This is known as the NT hash. *I have no idea why it was decided to use MD4 (which is considered weak) instead of MD5 (which is not the best but still more secure).*
- The password is converted to upper case, and then padded or truncated to 14 bytes. This string is then appended with 5 bytes of NULL characters and split to form two 56-bit DES keys to encrypt a "magic" 8-byte value. The resulting 16 bytes form the LanMan hash. *This is really weak to split the password in two parts and not to use an initialisation vector for each*

*password ! One can see if two passwords are equal on their 7 first (or last) characters. This also allows to use precomputed tables in order to run time-memory tradeoff attacks and finally to crack most LanMan passwords in few minutes.*

User-level security in Samba is not as weak as “Unix authentication” mode in NFSv4 since usage of passwords increases a bit the level of security. However, when using LanMan hashes, this does not make a great additional difficulty to attack Samba shares.

#### 11.10.4 ADS security v.s. Kerberos 5 in RPC-SEC\_GSS

In Kerberos mode, the level of security seems to be the same in Samba 3 than in NFSv4. The Windows 2000 Kerberos Security Support Provider (SSP) implements the General Security Service Application Program Interface (GSS API) Kerberos Mechanism Token formats defined in RFC 1964. Only DES-CBC-MD5 and DES-CBC-CRC encryption types are available for MIT interoperability.

Recall that NFSv4 uses DES-CBC-MD5 in RPC-SEC\_GSS ; security level is thus similar between Samba 3 and NFSv4 in Kerberos 5 mode.

#### 11.10.5 Conclusion

In conclusion of this comparison between NFSv4 and Samba 3, the security level is similar in each.

Without Kerberos, Samba 3 does better than NFSv4 since a login/password - which is MD4-hashed in current Windows versions - is required to access to a share file system. Recall that NFSv4 with Unix authentication trusts authentication on the client’s workstation since it only requires an UID/GID.

With Kerberos, cryptographic algorithms are the same between Samba 3 and NFSv4.

## Part IV

# Audit of the source code

A part of our security research on NFSv4 Linux implementations will consist in auditing the source code. NFSv4 is open-source software, written in C which will make the task easier.

## 12 Methodology

We first depict three major approaches to manually audit software :

- **top-down approach** : The source will be audited for well-known vulnerabilities without a good understanding of how NFSv4 protocol and implementation work. For example, we can search entire source tree for formatting vulnerabilities affecting the \*printf functions without reading the program line-by-line. This method will be good as a first audit, since it does not require an in-depth understanding of the code. The drawback is that any vulnerabilities that require deep knowledge of the context of the program will probably be missed. Basically, we will find vulnerabilities that can be located by reading one line of code. Anything requiring advanced skills has to be found in another way.

- **bottom-up approach** : In this approach, the auditor attempts to gain a very deep understanding of the software by reading a large portion of the code. This method is time consuming but allows to more easily discover more subtle bugs. NFSv4 source code is too large and complex for such an approach given the time available for our audit. However, a good understanding of the specification (RFC 3530 + drafts) will be necessary. Some parts of the code such as those on the path from the reception of a request to the transmission of the response need to be understood.

- **selective approach** : Since both the previous approaches have problems that prevent them from being really effective, a combination of the two can be more successful. Most of the time, a significant percentage of any source code is dead code when looking for security vulnerabilities. To save time and effort, we should focus auditing sections of code most likely to contain security issues that would be exploitable. That means, we have to locate code that can be reached by attacker inputs. This will mainly be parts of code which are reading fields of the RPC request. Code accessing data structures computed

from the latter fields is also candidate for vulnerabilities since an attacker can "indirectly" inject arbitrary values.

Starting with a top-down approach using automated source code analysis, we will quickly move to a selective approach with an instrumented manual investigation of selected parts of the code.

## 13 Automated source code analysis

As a starting point, a first level of security can be reached thanks to automated source code analysis tools. Automated analysis is generally useful to get a quick start on a large and relatively un-audited source tree. NFSv4 source code seems to be regularly audited with such tools (i.e. Coverity). Anyway, we will run different auditing tools on the source tree to remove potential simple vulnerabilities which remain undetected so far.

Candidate tools are :

- Recommended by NFSv4 developers :
  - **Coverity** [35] : a commercial source-code analysis tool. It is used by NFSv4 developers to regularly audit their code. An in-depth analysis can be run with this tool to check that all the code is well-audited.
  - **FlawFinder** [36] : scans C/C++ source code and reports potential security flaws.
  - **Sparse** [37] : Sparse is a static type-checking program written specifically for the Linux kernel, written by Linus Torvalds. Support for running sparse is in the kernel build system.
  - **SMATCH** [38] : a C source checker which mainly focuses checking the Linux kernel code. It is based on the papers about the Stanford Checker.
- Additional tools :
  - **DUMA** [39] : an open-source library to detect buffer overruns and under-runs in C and C++ programs. Redirect memory allocation system calls (`malloc`, `calloc`, `memalign`, `strdup`...) and deallocation (`free`). This can be used to detect vulnerabilities on user-land code under stress (`fsx`, `fstress`) and will be adapted to audit libraries.
  - **Splint** [40] : a static-analysis tool designed to detect security problems within C programs.

Splint has the ability of relatively strong security checking but requires to add special comments in the source code ([www.splint.org](http://www.splint.org)). A part of the code (i.e. server's kernel code) can be commented and checked with Splint.

- **Cqual** [41] : another analysis tool which detects simple vulnerabilities, like format strings.
- **RATS** [42] : offered by Secure Software, also detects simplistic vulnerabilities. There are lots of false negatives (undetected vulnerabilities) with this tool and this is critical when auditing a real-world software like NFSv4.

Unfortunately, all those tools are lacking when it comes to detecting complicated vulnerabilities. That's why they will never replace a manual audit.

## 14 Manual investigation

Regarding the amount of code written for or used by NFSv4, we will only investigate selected parts. We will first enumerate chunks of code that are reachable by user-defined input and that are likely to be attacked. We will then try to acquire an in-depth understanding of those parts.

Reading the code, we may -by chance- identify some vulnerabilities. But there is no doubt that we will also need specialized tools (i.e. `cscope` [43]) to locate calls to functions which are likely to introduce vulnerabilities.

Functions we want to locate and investigate the use have to be defined. The list will include functions such as `strcpy`, `strcat`, `memset`, `memcpy`, `gets`, `*scanf`, `*printf`...

Some other vulnerabilities can be found only with a careful read of the source code. For instance, bugs in loop constructs, integer overflows, signed comparison, uninitialized variable usage, use after free...

This will considerably increase the cost our manual investigation. That is why we should better concentrate on some well-defined parts of the code and analyze them in depth for a large number of vulnerabilities. This will ensure that those parts have a good security level.

We will let less sensitive parts to further analysis...

### 14.1 Defining the area to audit

Basically, NFSv4 is divided in two parts : client and server. Each side is itself divided in kernel and user-land code. Files involved are :

- Client / user-land :
  - `nfs-utils`
  - `util-linux` : patched for `mount/umount`
- Client / kernel :
  - `fs/nfs` *25,468 lines*
  - `fs/nfscommon` (for ACL mapping) *290 lines*
  - `net/sunrpc` *13,107 lines*
  - `net/sunrpc/auth_gss` *5,716 lines*
- Server / user-land :
  - `nfs-utils` :
    - `nfsd` *103 lines*
    - `mountd` *1763 lines*
    - There are also `exportfs`, `nfsstat`, `showmount` but since they are triggered by user-commands and not via the network, level of security risk is low.
- Server / kernel :
  - `fs/nfsd` *18,602 lines*
  - `fs/nfscommon` (for ACL mapping) *290 lines*
  - `net/sunrpc` *13,107 lines* and `/net/sunrpc/auth_gss` *5,716 lines*

NFSv4 also make use of different C libraries :

- Widely used and - hopefully - well analysed : `glibc`, `kerberos 5`
- `libnfsidmap` : Bull had run tests against the `nfsidmap` library. Specific security tests should be done, for instance long user/group names...
- `libacl` : Used for the NFSv4 ACL implementation.
- `libgssapi` : General library to deal with security flavors, hopefully well-audited.
- `librpcsec_gss` : Implementation of the RPC's security protocol.
- `lib-ti-rpc` : RPCbind service should be audited

Ideally an audit should be performed on the whole code. As a first audit, we concentrated on the server side and mainly the kernel code (reachable by attacker inputs coming from the network).

User-land commands on server side should only be accessible to a root user, so there is no need to check if some local vulnerabilities can be exploited.

## 15 Vulnerability classes

A list of vulnerability classes that are commonly or not so commonly found in C applications will give us an overview of what we should look for in the source. We took inspiration from [44] to draw the following list of vulnerability classes.

### 15.1 Generic logic errors

#### 15.1.1 Description

This is the most non-specific class but the root cause of many issues. The goal is to well understand the application in order to find flaws that can lead to security vulnerabilities.

#### 15.1.2 Application

As NFSv4 code is reasonably secure and well-audited, vulnerabilities are likely to be found in this class.

### 15.2 Almost extinct bug classes

#### 15.2.1 Description

A few vulnerabilities were commonly found in open source software some years ago, now they are seen more rarely but they can still exist. They generally take the form of well-known unbounded memory copy functions such as `strcpy`, `sprintf`, `strcat`, `gets...` Those type of functions have no idea of the the size of their destination buffers. If the destination buffer has been properly allocated , or verification of the input size is done before any copying, these functions can be used without danger.

However, when no checking is performed, these functions introduce security risks and are well-known to be subject of attacks such as stack overflows [45].

#### 15.2.2 Application

Dangers associated to the use of those functions is well-known to experienced programmers and the risks are documented in man pages. There is thus little chance to find this type of bugs in NFSv4 code.

NFSv4 server being divided in user and kernel code, we should verify that bounds-checking is done at the border between user and kernel land [wiki - f].

NFSv4 is mainly manipulating files and file attributes, so a particular care should be given to large names of files,

large ACLs (limited to 35 rules), large named attributes ... [wiki - g,h,i]

This is a rough list of potentially-dangerous functions [wiki - d] :

- `gets`, `*scanf`, `*printf`, `strcat`, `strcpy`, `mktemp`, `system`...

## 15.3 Format strings

### 15.3.1 Description

The format string class of vulnerabilities surfaced in the year 2000 [46] [47] and has resulted in the discovery of significant vulnerabilities in the past few years.

The bug is based on the attacker being able to control the format string passed to functions that accept printf-style arguments (`*printf`, `syslog`, `*scanf`,...). Exploitation of these vulnerabilities has largely been based on the use of the previously obscure `%n` directive in which the number of bytes already printed is written to an integer pointer argument.

### 15.3.2 Application

Fortunately, format string bugs are really easy to find in an audit since there is only a limited number of functions accepting printf-arguments. NFSv4 source code contains a certain number of calls to those functions that will have to be checked.

## 15.4 Generic incorrect bounds-checking

### 15.4.1 Description

To prevent from vulnerabilities such as the one described in 2.2, applications will make an attempt at bounds-checking. However, it is reasonably common that these attempts are done incorrectly. Spotting such vulnerabilities requires an in-depth analysis of bound checkings. Therefore, we should not consider that a piece of code is not vulnerable because it makes some attempts at bound-checking. These attempts also have to be verified before moving on.

### 15.4.2 Application

Bounds-checking should be done when manipulating strings, dynamic memory chunks or static arrays [wiki - e]. Lots of bounds-checkings are done in NFSv4 source code and we will thus check if they are all done correctly.

## 15.5 Loop constructs

### 15.5.1 Description

Loops are very common place to find buffer overflow vulnerabilities, possibly because their behavior is a bit more complicated than linear code. The more complex the loop, the more likely it will introduce a new vulnerability.

There is no doubt that checking all loops in the source will be painful and time-consuming, that's why we should better concentrate on the more complex ones as a first analysis.

Usage of some functions in loop context are well-known to be dangerous, for instance :

`*getc`, `getchar`...

### 15.5.2 Application

We will audit complex loops in the NFSv4 source code.

## 15.6 Off-by-One / Off-by-a-few vulnerabilities

### 15.6.1 Description

Off-by-one or off-by-a-few vulnerabilities are common coding errors in which one or a very limited number of bytes are written outside the bounds of allocated memory. Those vulnerabilities often result from incorrect null-termination of strings.

Exploitation can be hard (limited space to inject the shellcode) but is often possible. Exploitation has been demonstrated in some widely deployed applications in the past.

### 15.6.2 Application

We will have a closer look at this type of vulnerabilities when auditing NFSv4.

## 15.7 Non-null termination issues

### 15.7.1 Description

For strings to be handled securely, they must generally be properly null-terminated so that their boundaries can be easily determined. Strings not properly terminated

can lead to exploitable security issue later in the program execution. For example, if a string is not properly terminated, adjacent memory may be considered to be part of the same string and an attacker may thus be able to write arbitrary data.

A common source of non-null terminated strings is `strncpy`. Indeed, if the latter function runs out of space in the destination buffer, it will not null-terminate the string it writes.

### 15.7.2 Application

When auditing the source code against unsecure string functions usage we will consider non-null termination issue.

## 15.8 Signed comparison vulnerabilities

### 15.8.1 Description

Many coders want to prevent from overflow vulnerabilities by performing length checks on user input, but this is often done incorrectly when signed-length specifiers are used. If two signed integers are compared, a length check may not take into account the possibility of an integer being less than zero, especially when compared to a constant value.

For instance, the following comparison will be unsigned :

```
if( (int) left < (unsigned int) right )
```

Operators such as `sizeof()` are unsigned, and the resulting comparison will thus be unsigned :

```
if( (int) left < sizeof(buf) )
```

### 15.8.2 Application

This type of vulnerabilities can be hard to detect in the source, since it implies verifying the types in all integer comparisons. However, a checkout will be done each time user-inputs can influence an integer value latter used in comparison.

## 15.9 Integer overflows

### 15.9.1 Description

Integer overflows are well known and documented vulnerabilities in the world of security research. The first

speech introducing integer overflows was given at Back-Hat USA 2002 in [48].

Integer overflows occur when an integer increases beyond its maximum value or decreases below its minimum value. The minimum and maximum is defined by its type and size. Addition, subtraction or multiplication overflows cause the result to wrap over the maximum/minimum boundary for the integer type.

Integer overflows are useful for an attacker to bypass size checks or to cause buffers to be allocated at a size too small to contain the data copied into them.

### 15.9.2 Application

Discovering integer overflows thanks to a static audit is not an easy task. A more efficient way to discover them is thanks to fuzzing techniques - used later on in this audit.

## 15.10 Different-sized integer conversions

### 15.10.1 Description

Conversion between integers of different sizes can have interesting and unexpected results. They can lead to truncation of values, cause the sign of integer to change or cause values to be sign extended. Thus, they can sometimes lead to exploitable security conditions.

For example, a 16-bit signed integer value of -1 (0xffff) will be extended to 4294967295 (0xffffffff) when converted to a 32-bit unsigned value.

### 15.11 Application

As integer overflows, those types of issue are not always easy to discover and fuzzing techniques are more likely to point out such insecure conversions. Our NFSv4 fuzzer will thus be an invaluable tool to check the code against integer conversion bugs.

## 15.12 Double free vulnerabilities

### 15.12.1 Description

Although the mistake of freeing the same memory chunk twice may seem benign, it can lead to memory corruption and arbitrary code execution. Most programmers do not make the mistake of freeing a local variable twice but global pointers are, on the other hand, likely to introduce such security mistakes.

Therefore, a good practice when freeing memory is to set the associated pointer to null afterward. If not done, we can potentially find a double free vulnerability later on in the code and such vulnerabilities will thus have to be taken into account during the audit.

### 15.12.2 Application

We will search the sources for double usage of `free` or `realloc` with size of zero.

## 15.13 Out-of-scope memory usage vulnerabilities

### 15.13.1 Description

Certain memory regions in an application have a scope and lifetime for which they are valid. Any use of these regions before they are valid or after they become invalid can be considered a security risk. A potential result is memory corruption which can lead to arbitrary code execution.

### 15.13.2 Application

Vulnerabilities of this class will not be easy to locate since they require a good understanding of the source code to identify when pointers become valid or invalid.

## 15.14 Uninitialized variable usage

### 15.14.1 Description

While it is relatively uncommon to see the use of uninitialized variables, these vulnerabilities can lead to real exploitable conditions in applications. They are rare because they can lead to immediate program crashes and most commonly found in code that is not commonly exercised, such as code blocks triggered by uncommon error conditions.

Some development tools warn about access to uninitialized data but only in simple cases. But usage of uninitialized data often comes from initialization in complex test structure, which cannot be detected at compilation time.

### 15.14.2 Application

Manual audit against this class of vulnerabilities will be painful and inefficient since the task of checking all variable initializations is unrealistic. On the other hand,

NFSv4 fuzzing will be likely to trigger unusual exceptions hence testing chunks of code which are rarely exercised in regular uses.

## 15.15 Use after free vulnerabilities

### 15.15.1 Description

Heap buffers are valid for a lifetime, from the time they are allocated (`malloc`) to the time they are deallocated (`free`, `realloc` of size zero). Any attempts to write to a heap buffer after its deallocation can lead to memory corruption and eventually arbitrary code execution.

Use after free vulnerabilities are most likely to occur when several pointers to a heap buffer are stored in different places and one of them is freed. It can also happen in cases where pointers to different offsets into a heap buffers are stored and the original buffer is freed.

## Part V

# Fuzzing and fault injection methods

Fuzzing is a term that encapsulates the activity that surrounds the discovery of most security bugs found. One method of fuzzing involves the technique of fault injection which generally involves sending bad data into an application by means of manipulating various API calls or by sending by network requests as for NFSv4.

Unlike static analysis (such as source code audit), when a fuzzer finds a security hole, it has typically given the user the set of input that was used to find it. Indeed, a major drawback of static analysis is that the security researcher tends to find an enormous wealth of bug that may or may not be reachable by the input sent to the application externally. Tracking down each bug found during the analysis to see if it can actually be triggered is not efficient nor scalable. In the static analysis of NFSv4, it was indeed really difficult to determine what bugs outputted by Flawfinder were real security vulnerabilities.

As security researchers, we are not interested in non-exploitable bugs or bugs that cannot be reached. That is why fuzzer are most likely to be useful to discover real security vulnerabilities. However, there is a limitation in fuzzer : the input space covered by the fuzzer grows exponentially with the number of fields in the request.

I used SPIKE, a fuzzing framework, to put the dynamic audit to work. Practically all network protocols can be modeled in SPIKE and in particular, it is pretty easy to model RPC and NFSv4 request. Moreover, SPIKE supports XDR format and that's why it turned out to be an efficient framework for RPC and NFSv4 fuzzing. Once the request modeled in SPIKE, faults can be injected like :

- out-of-bound value (i.e. the maximum value for an integer field)
- faulty XDR format (i.e a the data part an XDR field that does not correspond to what is specified in the length part)
- value that does not correspond to the specification (i.e. bad file handle)
- operations that does not respect NFSv4 state (i.e. WRITE before a file is opened)

RPC fuzzing is not hard to put in practice. But fuzzing NFSv4 is much harder since the protocol is stateful. I

have drafted a implementation in SPIKE that covers simple NFSv4 operations. However, there are still a lot of work to do to achieve this fuzzer so as to be able to cover all code in as many conditions as possible.

## Part VI

# Conclusion

## What has been done

- Possible attacks against NFSv4 and the underlying risks have enumerated and explained.
- A theoretical audit of RPCSEC\_GSS has been completed. It has shown that the algorithms in use for authentication, integrity and privacy are not enough secure regarding today's computing power. A new version of RPCSEC\_GSS is being developed by CITI.
- A small part of the source code has been audited and documented.
- A draft of a fuzzer to audit NFSv4 dynamically has been developed. It is based on SPIKE framework. The goal is to inject faulty requests into NFSv4 server and to observe the response in return. Hopefully, it will help to discover vulnerabilities but there are still a lot of work to complete this fuzzer.

## Perspectives

The security audit is not completed and I hope someone will finish it. Work has to be done on the source code audit since the area of NFSv4-related source code is really large. It may take several months to audit seriously all the code. My prototype of a modified client to run a dynamic audit on NFSv4 server should also be completed to implement every NFSv4 operation.

## References

- [1] Nmap (Network Mapper) - <http://www.insecure.org/nmap/>.  
*Nmap is a free open source utility for network exploration or security auditing. It is well-known and used in the hacking community.*
- [2] RFC 2623 - <http://www.ietf.org/rfc/rfc2623.txt>.  
*NFS Version 2 and Version 3 Security Issues and the NFS Protocol's - Use of RPCSEC\_GSS and Kerberos V5*, June 1999.
- [3] SPIKE Proxy - <http://www.immunitysec.com/resources-freesoftware.shtml>.
- [4] CITI University of Michigan - <http://www.citi.umich.edu/projects/nfsv4>.  
*NFSv4 developers' Website.*
- [5] Bull GNU/Linux NFSv4 project - <http://nfsv4.bullopen-source.org>.  
*Our team's Website.*
- [6] ARP Spoofing references to be defined.
- [7] ARP Cache Poisoning references to be defined.
- [8] ICMP redirect references to be defined.
- [9] DHCP's proxy corruption references to be defined.
- [10] Ettercap - <http://ettercap.sourceforge.net>.  
*Ettercap is a suite for man in the middle attacks on LAN. It features sniffing of live connections, content filtering on the fly and many other interesting tricks.*
- [11] DNS Spoofing references to be defined.
- [12] IP-spoofing demystified by daemon9, route, infinity - <http://www.zone-h.org/files/49/Information-on-IP-Spoofing.txt>. June 1996.
- [13] RFC 2203 - <http://www.ietf.org/rfc/rfc2203.txt>.  
*RPCSEC GSS Protocol Specification*, September 1997.
- [14] Managing NFS and NIS (2nd Edition Review) by Hal Stern, Mike Eisler, Ricardo Labiaga - O'Reilly. July 2001.
- [15] LDAP System Administration by Gerald Carter - O'Reilly. March 2003.
- [16] Ethereal - <http://www.ethereal.com>.  
*A widely-used GUI network sniffer and analyzer.*
- [17] Tcpdump - <http://www.tcpdump.org>.  
*A widely-used network sniffer and analyzer.*
- [18] Snort - <http://www.snort.org/>.  
*An open source network intrusion prevention and detection system (IPS/IDS).*
- [19] RFC 3530 - <http://www.ietf.org/rfc/rfc3530.txt>.  
*Network File System (NFS) version 4 Protocol*, April 2003.
- [20] RFC 1510 - <http://www.ietf.org/rfc/rfc1510.txt>.  
*The Kerberos Network Authentication Service (V5)*, September 1993.
- [21] RFC 1964 - <http://www.ietf.org/rfc/rfc1964.txt>.  
*The Kerberos Version 5 GSS-API Mechanism*, June 1996.
- [22] Limitations of the Kerberos Authentication System by Steven M. Bellovin and Michael Merritt - Winter 1991 Usenix Conference - <http://www.cs.columbia.edu/smb/papers/kerblimit.usenix.pdf>. January 1991.
- [23] Kerberos Scaling issue references to be defined.
- [24] RFC 2847 - <http://www.citi.umich.edu/projects/nfsv4/rfc/rfc2847.txt>.  
*A Low Infrastructure Public Key Mechanism Using SPKM*, June 2000.
- [25] Draft Adamson RFC 2847 bis 00 - <http://www.ietf.org/internet-drafts/draft-adamson-rfc2847-bis-00.txt>.  
*Low Infrastructure Public Key Mechanisms: SPKM-3 and LIPKEY*, March 2006.
- [26] Tripwire - <http://www.tripwire.com>.  
*Tripwire solutions detect and alert to the changes in system files and device configurations.*
- [27] Side channel references to be defined.
- [28] Draft Nfsv4 Secinfo by M. Eisler - <http://www.citi.umich.edu/projects/nfsv4/rfc/draft-ietf-nfsv4-secinfo-00.txt>.  
*A commercial source-code analysis product.*
- [29] RFC 2623 - <http://www.ietf.org/rfc/rfc2203.txt>.  
*RPCSEC\_GSS Protocol Specification*, September 1997.
- [30] RFC 1831 - <http://www.ietf.org/rfc/rfc1831.txt>.  
*ONC RPC: Remote Procedure Call Protocol specification*, August 1995.

- [31] DES cracker - [http://www.eff.org/Privacy/Crypto/Crypto\\_misc/DESCracker/](http://www.eff.org/Privacy/Crypto/Crypto_misc/DESCracker/).  
*Electronic Frontier Foundation's DES cracker who cracked DES within 56 hours in 1998, 1998-1999.*
- [32] Standing the test of time : the Data Encryption Standard by Susan Landau.  
*An article which sums up attacks on DES, March 2000.*
- [33] Announcing the Advanced Encryption Standard (AES) - FIPS -  
<http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf>.  
*The reference on AES, a block cipher improving DES, November 2001.*
- [34] Draft to RFC 1964 -  
<http://mirrors.isc.org/pub/www.watersprings.org/pub/id/draft-raeburn-cat-gssapi-krb5-3des-00.txt>.  
*Triple-DES support for the Kerberos 5 GSSAPI Mechanism, July 2000.*
- [35] Coverity - <http://www.coverity.com>.  
*A commercial source-code analysis product.*
- [36] FlawFinder - <http://www.dwheeler.com/flawfinder>.  
*Program that scans C/C++ source code and reports potential security flaws. By default, it sorts its reports by risk level (the riskiest operations in the code are listed first).*
- [37] Sparse -  
<http://tree.celinuxforum.org/CelfPubWiki/Sparse>.  
*Sparse is a static type-checking program written specifically for the Linux kernel, written by Linus Torvalds.*
- [38] Smatch - <http://smatch.sourceforge.net>.  
*Smatch is C source checker but mainly focused checking the Linux kernel code. It is based on the papers about the Stanford Checker.*
- [39] DUMA - <http://duma.sourceforge.net>.  
*An open-source library (under GNU General Public License) to detect buffer overruns and under-runs in C and C++ programs.*
- [40] Splint - <http://www.splint.org>.  
*Splint is a tool for statically checking C programs for security vulnerabilities and coding mistakes.*
- [41] Cqual - <http://www.cs.umd.edu/~jfoster/cqual>.  
*Cqual is a type-based analysis tool that provides a lightweight, practical mechanism for specifying and checking properties of C programs.*
- [42] RATS ( Rough Auditing Tool for Security) by Secure Software -  
[http://www.securesoftware.com/resources/download\\_rats.html](http://www.securesoftware.com/resources/download_rats.html).  
*RATS is a tool for scanning C, C++, Perl, PHP and Python source code and flagging common security related programming errors such as buffer overflows and TOCTOU (Time Of Check, Time Of Use) race conditions.*
- [43] Cscope - <http://cscope.sourceforge.net>.  
*Cscope is a developer's tool for browsing source code, particularly adapted to large projects.*
- [44] The shellcoder's handbook, Discovering and Exploiting security holes by Jack Koziol et al. - Wiley. 2004.
- [45] Smashing the stack for fun and profit by Aleph One - <http://www.phrack.org/show.php?p=49&a=14>.  
*The most famous paper introducing stack overflows., 1996.*
- [46] Format String Attacks by Tim Newsham (Guardent, Inc.) -  
<http://www.lava.net/~newsham/format-string-attacks.pdf>.  
*A first insight of format string bugs.*
- [47] Advances in format string exploitation by gera and riq (Corest) -  
<http://www.phrack.org/phrack/59/p59-0x07.txt>.  
*A good example of advanced exploitation of format string bugs on SPARC architectures.*
- [48] Professional Source Code Auditing - Presentation at BackHat USA 2002 -  
<http://www.blackhat.com/presentations/bh-usa-02/bh-us-02-iss-sourceaudit.ppt>.  
*The first presentation of integer overflows exploitation.*