

Dokumentation

zum Projekt

Integration eines Security Service in Flexinet

von

Michael Jung

TU Berlin

Wintersemester 2001 / 2002

Berlin, den 14.04.2002

1 Einführung

Flexinet ist ein Projekt, bei dem es um die Erstellung einer *aktiven bzw. programmierbaren Netzinfrastruktur zur schnellen und flexiblen Realisierung neuer Dienste* geht (z.B. MPEG-Filter und ähnliche Dienste). Zur Umsetzung des Flexinet-Konzeptes wird das System *AMnet* verwendet. Hier wird ein angeforderter Dienst als Object-Code von einem LDAP-Server geladen und lokal (auf dem Netzknoten) ausgeführt. Ebenfalls zu AMnet gehören Mechanismen bzw. Skripte, mit denen solche Dienste auf dem LDAP-Server abgelegt werden können.

2 Aufgabenstellung

Das Ziel des Projektes bestand in der Einführung eines *Security Service* in das System AMnet. Im Detail soll hier als Security Goal die *Data Integrity* für den Object-Code eines Dienstes gewährleistet werden, und zwar vom Zeitpunkt seiner Ablage auf dem LDAP-Server bis zu seiner Ausführung auf dem aktiven Netzknoten. Data Integrity bedeutet, dass Daten zwar von unauthorisierten Entities gelesen, nicht jedoch manipuliert werden können. Letzteres muss etwas genauer betrachtet werden. Hier wird der Ausschluss der Manipulation auf indirektem Weg sichergestellt, und zwar derart, dass eine solche Manipulation zwar möglich ist, jedoch mit Sicherheit (oder sehr hoher Wahrscheinlichkeit) von autorisierter Seite festgestellt werden kann. In der bisherigen AMnet-Version wäre es z.B. möglich, dass ein Angreifer den Object-Code auf dem LDAP-Server durch einen eigenen, wohl möglich schädlichen Object-Code ersetzt, welcher dann bei der nächsten Anforderung des Dienstes ausgeführt wird.

Da lediglich die Data Integrity gewährleistet werden soll, nicht jedoch weitere Security Goals, wie z.B. Confidentiality (Daten sind nur autorisierten Entities zugänglich), bietet sich als Security Service eine *Signierung* der Dienste und deren spätere *Verifikation* an. Das Resultat einer Signierung ist eine dem signierten Dienst zugeordnete Signatur, welche selbstverständlich zusammen mit dem Dienst auf dem LDAP-Server gespeichert werden muss, um später eine Verifikation durchführen zu können. AMnet muss also bei der Anforderung eines Dienstes zusätzlich deren Signatur vom LDAP-Server laden und dem Anwender entsprechende Meldungen ausgeben, abhängig davon, ob die Verifikation erfolgreich war oder nicht.

3 Realisierung

Aus der Aufgabenstellung lassen sich insgesamt vier Teilaufgaben separieren :

- *Signierung eines Dienstes vor dessen Ablage auf dem LDAP-Server*
- *Erweiterung der Datenstruktur des LDAP-Servers um Signatur und ggfs. weitere Informationen*
- *Anpassung von AMnet an Datenstruktur des LDAP-Servers und Verifikationsmechanismus*
- *Verifikation eines Dienstes vor dessen Ausführung*

Eine kurze Analyse dieser Teilaufgaben zeigt die grundlegenden Lösungskonzepte auf :

Die Signierung stellt bzgl. eines Dienstes einen *einmaligen Vorgang* dar. Der Dienstanbieter kann seinen Dienst von Hand signieren, da hier Zeit keine wesentliche Rolle spielt. D.h. hierfür ist ein einfaches *Signierungstool* sinnvoll, welches von der Kommandozeile gestartet wird.

Nach der Signierung muss die erzeugte Signatur zusammen mit dem Dienst auf dem LDAP-Server abgelegt werden. Auch dies ist analog zur Signierung ein einmaliger Vorgang, welcher keiner Automation bedarf. Die Struktur der Dienste auf dem LDAP-Server wird durch Konfigurationsdateien festgelegt; entsprechend müssen hier also nur die Signatur und ggfs. weitere Informationen in eine solche Datei eingetragen werden, was ebenfalls von Hand geschehen kann.

Die Anpassung von AMnet an die Datenstruktur des LDAP-Servers ist selbstverständlich ein integraler Bestandteil des gesamten Projektes, war jedoch nicht Teil meiner Arbeit. Hier verweise ich auf die Dokumentation von Mathias Bohge, der im selben Zeitraum diesen Projektteil erarbeitete.

Die Verifikation muss bei jeder Anforderung eines Dienstes durchgeführt werden, auch dann, wenn ein und derselbe Dienst mehrmals hintereinander angefordert wird. Hier ist also im Gegensatz zur Signierung eine automatische Verarbeitung von Vorteil. Da AMnet in Java geschrieben ist und für Signierung und Verifikation eine frei verfügbare C++-Bibliothek verwendet wird, konnte die Verifikation nicht direkt in AMnet realisiert werden. Stattdessen wird ein Client/Server-Mechanismus verwendet, d.h. die Verifikation erfolgt durch einen in C++ geschriebenen Server.

In den folgenden Abschnitten sollen die drei zu meinem Projektteil gehörenden Aufgaben und deren Lösung näher beschrieben werden. Anschliessend folgen eine Einführung in die Projektumgebung sowie Hinweise zur Anpassung des AMnet-Systems an den Signierungsmechanismus !

3.1 Signierung eines Dienstes

Die Signierung eines Dienstes erfolgt durch den Dienstanbieter. Wie bereits erwähnt, bestand der Plan für diesen Projektteil in der Programmierung eines Signierungstools, welches von der Kommandozeile gestartet wird. Auf Grund der verwendeten Crypto-Bibliothek wurde als Programmiersprache C++ gewählt.

Signierung bedeutet im Gegensatz zur reinen Verschlüsselung, dass als Security Goal lediglich die Data Integrity gewährleistet wird, d.h. ein potentieller Angreifer hat zwar Einblick in den Inhalt des *Plaintext* (des zu signierenden Textes), jedoch wird eine Manipulation des Plaintext nicht den vom Angreifer gewünschten Effekt erzielen. Letzteres hängt damit zusammen, dass der Empfänger aufgrund der beigefügten Signatur jegliche Manipulation des Plaintext feststellen und damit dessen Weiterverarbeitung (vor allem seine Ausführung) verhindern kann.

Auf eine leicht zu übersehene Tatsache soll gleich zu Beginn hingewiesen werden. Das Ziel des Projektes bestand in der Schaffung eines Security Service, welcher die Data Integrity für Dienste gewährleistet. Dieses Feature sollte dazu dienen, den Schutz des aktiven Knoten (auf dem AMnet läuft) um eine mächtige Komponente zu erweitern, welche das Einschleusen von manipuliertem und ggfs. gefährlichem Code verhindert. Hier muss festgestellt werden, dass dieses Projekt nicht der Einführung eines Mechanismus zur Code-Analyse dient, um gefährlichen von ungefährlichem Code zu unterscheiden (dies wäre auch nicht möglich, was durch grundlegende Theorien der Informatik festgelegt ist). Das bedeutet, dass ein bereits vom Dienstanbieter mit gefährlichen Anweisungen versehener Dienst-Code selbstverständlich „ohne Probleme“ auf den aktiven Knoten gelangen und dort ausgeführt werden kann. Die Konsequenz daraus muss sein, dass man nur Dienste von vertrauenswürdigen Anbietern zulässt.

Doch nun zur konkreten Umsetzung des Signierungsteils; im Wesentlichen besteht eine Signierung aus zwei Schritten :

- *Bildung eines dem Plaintext zugeordneten Hash-Wertes*
- *Verschlüsselung des Hash-Wertes*

Der erste Schritt wird durch eine *Hash-Funktion* realisiert; ich entschied mich für die Funktion *SHA*. Für den zweiten Schritt musste ein Verschlüsselungsalgorithmus gewählt werden. Es bot sich der weit verbreitete *RSA*-Algorithmus an, für den bereits eine Menge Tools verfügbar sind. Bei *RSA* handelt es sich um eine asynchrone Verschlüsselung. Das bedeutet, dass für Ver- und Entschlüsselung zwei verschiedene Schlüssel verwendet werden – der *private-key* und der *public-key* (in dieser Reihenfolge).

Obwohl die Verifikation auf der Seite des Dienstanbieters (und damit für das Signierungstool) nicht von wesentlicher Bedeutung ist, müssen hier trotzdem schon einige Dinge diesbzgl. beachtet werden. Zum einen muss dafür gesorgt werden, dass bei der späteren Verifikation des Plaintext der korrekte *public-key* verwendet wird. Da es in der praktischen Anwendung von AMnet mit Sicherheit mehr als einen Dienstanbieter geben wird, müssen auch mehrere *public-keys* verwaltet werden. Das bedeutet, dass neben Dienst und Signatur zusätzlich eine Beschreibung des Dienstanbieters auf dem LDAP-Server abgelegt werden muss. Bei jeder späteren Anforderung des Dienstes kann anhand dieser Information der (weiter unten beschriebene) Verifikationsserver den passenden *public-key* aus einer Datenbank auswählen. Um eine möglichst hohe Sicherheit zu gewährleisten, sollten alle relevanten Informationen vor der Ablage auf dem LDAP-Server signiert werden. Es bietet sich also an, neben dem Dienst auch zusätzlich die Beschreibung des Dienstanbieters zu signieren. Da man als Resultat nur eine einzige Signatur erhalten will, müssen Dienst und Dienstanbieterbeschreibung zu einem Plaintext zusammengefasst, d.h. in eine Datei geschrieben, und danach signiert werden.

Mit fortschreitender Projektarbeit wurde mir klar, dass für die Signierung ein von AMnet unabhängiges Tool von Vorteil ist. Als Resultat soll das Tool eine Signatur liefern, welche bzgl. ihres Formats auch von anderen *RSA*-Tools erstellt werden kann. So bot es sich an, zunächst alle für die gesamte Projektarbeit notwendigen Signierungs- und Verifikationsmechanismen in einem Tool zu implementieren, um sich mit der Funktionalität der Crypto-Bibliothek vertraut zu machen.

Es entstand das Tool *RSAmagic*, welches folgende Features besitzt :

- *Generierung eines Schlüsselpaares (private-key und public-key)*
- *Erzeugung des Plaintext durch Konkatenation seiner Bestandteile*
- *Signierung des Plaintext mittels eines private-key*
- *Verifikation des Plaintext mittels eines public-key*
- *Konvertierung diverser Dateiformate (ASCII-, Hexadezimal- und Base64-Format)*

Die einzelnen Features sollen nun in dieser Reihenfolge vorgestellt werden.

3.1.1 Generierung von private-key und public-key

Zur Erzeugung eines Schlüsselpaars wird folgender Aufruf verwendet :

```
RSAmagic -k <signature length> {<key-memo> | <private-key> <public-key>}
```

Ein Beispielaufruf ist :

```
RSAmagic -k 1024 mykey
```

Der Parameter *signature length* legt fest, welche Länge (in Bits) jede zukünftige, mit dem private-key erzeugte Signatur besitzt. Das bedeutet, dass die Signaturlänge im private-key selbst gespeichert wird (Standard PKCS1v15). Entsprechend können auch mit dem public-key nur Signaturen dieser Länge verifiziert werden.

Hinter der Angabe der Signaturlänge folgen optionale Parameter zur Festlegung der Dateinamen für private-key und public-key. Wird lediglich ein zusätzlicher Parameter verwendet, so gibt dieser den Namensrumpf für beide Schlüssel an (oben als *key-memo* bezeichnet), wobei das Tool diesen selbständig für den private-key um das Suffix *.pvk* und für den public-key um das Suffix *.pbk* erweitert. Werden stattdessen zwei zusätzliche Parameter angegeben, so definieren diese eindeutig beide Dateinamen für das Schlüsselpaar (Reihenfolge siehe oben). Werden keinerlei Angaben bzgl. des Dateinamens gemacht, so verwendet das Tool standardmässig den Namensrumpf *RSAmagic*, d.h. es werden zwei Dateien erstellt : *RSAmagic.pvk* und *RSAmagic.pbk* !

Die Speicherung sämtlicher Schlüssel erfolgt in hexadezimaler Form, wodurch jede Schlüsseldatei mit einem herkömmlichen ASCII-Texteditor darstellbar ist.

An die konkrete Erzeugung eines Schlüsselpaars für AMnet sind keine speziellen Bedingungen geknüpft, d.h. es gibt z.B. keine Beschränkung bzgl. der Schlüssellänge. (Allerdings müssen die Schlüssel natürlich wie oben beschrieben für den RSA-SHA-Algorithmus bestimmt sein, was von Bedeutung ist, wenn die Schlüssel mit einem anderen Tool erstellt werden.)

3.1.2 Erzeugung des zu signierenden Plaintext

Wie später noch gezeigt wird, kann *RSAmagic* nur eine Datei als Ganzes signieren. Soll ein bestimmter Teil einer Datei signiert werden, so muss dieser zunächst durch den Anwender in einer eigenen Datei gespeichert werden. Auf der anderen Seite muss eine aus mehreren Dateien bestehende Datenbasis für eine Signierung zu einer einzigen Datei zusammengefasst werden. Für letzteren Fall (welcher von beiden sicherlich am häufigsten auftritt) bietet *RSAmagic* mehrere Funktionen an. Alle basieren auf dem Mechanismus, dass am Ende einer Datei weitere Daten angehängt werden. Unterschiede treten hier in zwei Punkten auf. Zum einen geht es um die Frage, ob die anzuhängenden Daten aus einer Datei gelesen oder direkt im Parameter angegeben werden. Die zweite Frage besteht darin, ob nach der Zusammenstellung des Plaintext dieser später auch wieder sicher in seine ursprünglichen Bestandteile zerlegt werden soll. Diese Funktionalität wird derart gewährleistet, dass am Ende eines jeden angehangenen Datenblocks dessen Länge zusätzlich als 32-Bit-Integer in die Gesamtdatei aufgenommen wird. Die spätere Zerlegung des Plaintext kann dann in umgekehrter Reihenfolge, d.h. von hinten nach vorne erfolgen. Diese Funktionalität wurde in einem frühen Entwicklungsstadium des Projektes benötigt, später jedoch verworfen (aufgrund getrennter Ablage von Dienst-Objectcode, Signatur und Dienstanbieterbeschreibung auf dem LDAP-Server). Trotzdem ist das Feature in *RSAmagic* enthalten und kann für andere Anwendungen genutzt werden. Damit ergeben sich insgesamt vier mögliche Aufrufe :

```
RSAmagic -a <main file> <to be appended file>  
RSAmagic -at <main file> <to be appended text>
```

```
RSAmagic -al <main file> <to be appended file>  
RSAmagic -atl <main file> <to be appended text>
```

Hier einige Beispielaufufe :

```
RSAmagic -a libmpeg.so owner.txt  
RSAmagic -atl libdbl.so SysTecCo
```

Alle Aufrufe beginnen mit der Option *-a* (für *append*). Diese Option kann erweitert werden durch das Zeichen *t* für direkte Texteingabe in der Kommandozeile (anstatt des Namens der anzuhängenden Datei) und durch das Zeichen *l* für Speicherung der Länge des angehängten Datenblocks. Hinter der Optionsangabe folgt als nächster Parameter der Name der Datei (*main file*), an welche der Datenblock angehängt werden soll. Der letzte Parameter muss - abhängig vom Vorhandensein der Option *t* - entweder einen Dateinamen oder eine direkte Texteingabe repräsentieren !

Besteht der Plaintext aus mehr als zwei separaten Dateien (welche es zusammenzufügen gilt), so muss ein adäquater Aufruf des Tools entsprechend oft wiederholt werden. Als Resultat liegt eine verlängerte Datei vor, welche den kompletten zu signierenden Plaintext enthält.

Der Sinn der Option *l* bestand darin, später den Plaintext wieder in seine Bestandteile zerlegen zu können. Die dafür notwendigen Aufrufe sollen hier kurz aufgeführt werden :

```
RSAmagic -c <main file> <number of bytes> <cut file>
RSAmagic -cl <main file> <cut file>
```

Ein Beispielaufruf ist :

```
RSAmagic -c plaintext.asc 12 owner.txt
```

Die Option *-c* spezifiziert die *cut*-Funktion. Diese kann in zwei Ausprägungen vollzogen werden. Es kann entweder die konkrete Anzahl von Bytes, die es vom *main file* abzuschneiden gilt, angegeben werden oder man veranlasst mittels der zusätzlichen Option *l* das Tool, zuvor gespeicherte Längenangaben zur Bestimmung der abzuschneidenden Datenmenge zu verwenden. Beide Features haben die Eigenschaft, dass die abgeschnittenen Daten in einer eigenen Datei abgelegt werden - also nicht verloren gehen. Zu beachten ist hierbei, dass *RSAmagic* nicht erkennen kann, ob eine Datei tatsächlich vorher mit der spezifischen Längenspeicherung aus mehreren Dateien erzeugt wurde. Der Anwender muss selbst dafür sorgen, dass *RSAmagic* mit der passenden *cut*-Funktion ausgeführt wird, ansonsten können durch das Tool unvorhersagbare Aktionen ausgelöst werden (denn die letzten 4 Bytes der Datei werden bei der Option *-cl* stets als 32-Bit-codierte Länge interpretiert, unabhängig davon, wie diese 4 Bytes zu Stande kamen bzw. woher sie stammen).

In der konkreten Anwendung für AMnet wird der Plaintext mittels der Option *-a* oder *-at* erstellt. D.h. es werden nicht die Längen sämtlicher zusammenzufügender Datenbestandteile im Plaintext abgelegt. Der Plaintext besteht hier aus dem Dienst-Objectcode, welcher um die Dienstanbieterbeschreibung erweitert wird. Ein entsprechender Aufruf könnte wie folgt aussehen :

```
RSAmagic -at libmpeg.so teleSystemsCo
```

Es wurde die Option *-at* verwendet, d.h. der letzte Parameter (hier: Dienstanbieterbeschreibung) wird als Texteingabe interpretiert und direkt an das Ende der Datei *libmpeg.so* (hier: Dienst-Objectcode) angehängt. Handelt es sich bei der Dienstanbieterbeschreibung um eine komplizierte oder relativ grosse Zeichenfolge, so kann diese zur Vereinfachung des gesamten Prozesses in einer Datei gespeichert und später mittels der Option *-a* hinter den Dienst-Objectcode geschrieben werden. Allerdings ist hierbei unbedingt zu beachten, dass am Ende der Dienstanbieterbeschreibung kein Zeilenumbruchzeichen (Newline) stehen darf. Dies ist mittels der Option *-at* automatisch gewährleistet. Das Erzeugen einer separaten Datei, welche die Dienstanbieterbeschreibung entsprechend eben beschriebener Form enthält, kann bei UNIX/Linux-Texteditoren unter Umständen zu unlösbaren Problemen führen, da diese (meist) selbständig einen Zeilenumbruch hinzufügen. Aus diesem Grund soll darauf hingewiesen werden, dass mittels der Option *-at* nicht nur der gesamte Plaintext, sondern auch mit einem „Trick“ die Datei für die Dienstanbieterbeschreibung erstellt werden kann - nämlich in dem Fall, dass ein Text an eine nicht vorhandene oder mit Null-Länge versehene Datei gehängt wird. Ist die für *RSAmagic* spezifizierte Datei nicht vorhanden, so wird sie automatisch mit Null-Länge erstellt und um den angegebenen Text erweitert.

Der erste der beiden folgenden Aufrufe erspart also für zukünftige Signierungen viel Schreibarbeit (und muss demnach für weitere Signierungen nicht wiederholt werden) :

```
RSAmagic -at owner.txt teleSystemsCoServiceDescriptionForAMnet0E25A149C2
RSAmagic -a libmpeg.so owner.txt
```

Es sei noch zu erwähnen, dass die verwendete Dienstanbieterbeschreibung nicht nur mit Dienst-Objectcode und Signatur auf dem LDAP-Server abgelegt werden muss (siehe 3.2), sondern selbstverständlich auch auf der Seite des Verifikationsservers (siehe 3.3) bekannt sein, also in der zugehörigen Datenbank eingetragen werden muss ! In dieser Datenbank, deren Format später noch erläutert wird, werden Dienstanbieterbeschreibungen eindeutige public-keys zugeordnet. Damit kann der Verifikationsserver herausfinden, welcher public-key für die Verifikation eines Dienstes verwendet werden soll.

Auf einen konzeptionellen Ansatz soll hier noch kurz hingewiesen werden : Theoretisch hätte man das gesamte System auch so entwerfen können, dass der public-key direkt zusammen mit Dienst-Objectcode und Signatur auf dem LDAP-Server abgelegt wird und dadurch die Dienstanbieterbeschreibung komplett entfällt. Das RSA-Signierungskonzept mit seinem Schlüsselpaar sieht auch ausdrücklich in seiner Sicherheitsanalyse vor, dass ein Angreifer Plaintext, Signatur und public-key besitzt - und trotzdem daraus nicht oder nur in unangemessener Rechenzeit auf den private-key schliessen kann (denn der wäre notwendig, um eine Manipulation des Plaintext durch Neusignierung zu verschleiern). Trotzdem wurde also die Dienstanbieterbeschreibung eingeführt, um auf der einen Seite einem potentiellen Angreifer die Arbeit so schwer wie möglich zu machen, und um auf der anderen Seite die Zugehörigkeit eines Dienstes zu einem Anbieter auf einfache Weise sichtbar zu machen (für statistische Angaben usw.).

3.1.3 Signierung des Plaintext

Für den Signierungsvorgang werden der Plaintext (siehe 3.1.2) und der private-key (siehe 3.1.1) benötigt. Als Resultat erhält man eine dem Plaintext zugeordnete *Signatur*, durch welche später mittels des (zum private-key korrespondierenden) public-key der originäre Plaintext verifiziert werden kann.

Die für die Signierung auszuführende Anweisung lautet :

```
RSAmagic -s <plaintext file> {<private-key file> {<signature file>}}
```

Ein Beispielaufruf ist :

```
RSAmagic -s plaintext.asc mykey.pvk signature.hex
```

Nach der Option *-s* folgen die Dateinamen für Plaintext, private-key und Signatur. Die ersten beiden Dateien repräsentieren Datenquellen, müssen also bereits vor dem Signierungsaufruf vorhanden sein. Die Angabe der Dateinamen für private-key und Signatur sind optional. Fehlt letzterer, so schreibt *RSAmagic* die erzeugte Signatur in eine Datei mit dem Namen des Plaintext erweitert um die Endung *.sig* ! Fehlt auch die Angabe für den private-key, so wird standardmässig der Dateiname *RSAmagic.pvk* verwendet. Der Plaintext muss kein spezielles Format aufweisen; *RSAmagic* akzeptiert jede beliebige Datei.

Die Signatur liegt nach Ausführung der Anweisung in hexadezimaler Form vor - analog zur Speicherung der Schlüssel. Letztere dürfen vom Anwender nicht verändert oder konvertiert werden, sondern müssen für die Signierung in der Form, wie sie von *RSAmagic* erzeugt wurden, gelesen werden können.

Der Grund dafür, warum die Signatur in hexadezimaler Form gespeichert wird, besteht darin, dass sie für die Ablage auf dem LDAP-Server in einer AMnet-Konfigurationsdatei eingefügt werden muss. Solche Dateien sind in einfachem ASCII-Text geschrieben, dürfen also keine Sonderzeichen enthalten, damit sie mittels eines herkömmlichen Text-Editors bearbeitet werden können.

Die konkrete Umsetzung der Signierung für AMnet bedarf keiner besonderen Behandlung. Der zuvor entsprechend 3.1.2 erstellte Plaintext, bestehend aus Dienst-Objectcode und Dienstanbieterbeschreibung, kann direkt mit oben erwähntem Aufruf signiert werden. Sicherheitshalber sollte noch erwähnt werden, dass nicht die Base64-kodierte Form des Dienst-Objectcode signiert wird, sondern tatsächlich der originäre Code. Das ändert aber nichts daran, dass der Base64-kodierte Code wie gewohnt, d.h. wie es bei den bisherigen Versionen von AMnet der Fall war, auf dem LDAP-Server abgelegt wird.

3.1.4 Verifikation des Plaintext

Die Verifikation trifft eine Aussage darüber, ob der Plaintext seit seiner Signierung verändert wurde. Im Gegensatz zur Signierung müssen hier drei (statt zwei) Datenquellen vorhanden sein : Plaintext, public-key und Signatur. Das Resultat ist eine Meldung nach STDOUT, ob die Verifikation erfolgreich war oder nicht. Der Aufruf lautet :

```
RSAmagic -v <plaintext file> {<public-key file> {<signature file>}}
```

Ein Beispielaufruf ist :

```
RSAmagic -v plaintext.asc mykey.pbk signature.hex
```

Analog zum Signierungsaufruf sind auch hier die letzten beiden Parameter optional. Eine fehlende Signaturangabe wird von *RSAmagic* selbständig durch die Angabe für den Plaintext zuzüglich der Endung *.sig* ersetzt. Der public-key wird bei fehlender Angabe durch den Benutzer auf *RSAmagic.pbk* gesetzt.

Bzgl. AMnet wird die Verifikation nicht durch *RSAmagic*, sondern durch den später im Detail beschriebenen Verifikationsserver realisiert. Trotzdem kann die Verifikationsfunktionalität von *RSAmagic* von Nutzen sein, wenn man z.B. die Gültigkeit von Signatur und/oder public-key für einen Dienst überprüfen will. Der Mechanismus für die Verifikation ist hier derselbe wie der des Verifikationsservers, denn beide verwenden sie gleiche Crypto-Bibliothek. So ist es empfehlenswert, nach jeder Signierung die erzeugte Signatur mittels des beschriebenen Aufrufs von *RSAmagic* zu überprüfen, bevor diese zusammen mit Dienst-Objectcode und Dienstanbieterbeschreibung auf dem LDAP-Server abgelegt wird !

3.1.5 Konvertierung diverser Dateiformate

RSAmagic kann zwischen normalen ASCII-Format und zwei weiteren Formaten hin- und zurückkonvertieren. Bei letzteren handelt es sich um hexadezimale und Base64-Kodierung. Im Detail stehen folgende Aufrufe zur Verfügung :

```
RSAmagic -e16 <ascii file> {<hexadecimal file>}  
RSAmagic -e64 <ascii file> {<base64 file>}  
RSAmagic -e64l <ascii file> {<base64 file>}  
RSAmagic -d16 <hexadecimal file> {<ascii file>}  
RSAmagic -d64 <base64 file> {<ascii file>}
```

Die Option *-e* (für *encode*) führt stets eine Konvertierung ausgehend von einer ASCII-Datei hin zu einer Datei anderen Formats, welches durch eine zusätzliche (direkt hinter *-e* folgende) Zahl 16 für hexadezimales Format oder 64 für Base64-Format festgelegt wird. Der Optionsangabe (z.B. *-e16*) folgt als nächster Parameter der Name der ASCII-Datei, welche es zu konvertieren gilt. Wird als letzter Parameter keine Zielformat spezifiziert, so setzt *RSAmagic* diesen Dateinamen selbständig auf den der Quelldatei (ASCII-Datei) erweitert um die Endung *.hex* für hexadezimale Kodierung oder *.64* für Base64-Kodierung.

Bei Base64-Kodierung gibt es zusätzlich die Option *-e64l*, welche dafür sorgt, dass pro erzeugtem Base64-Zeichen stets ein Zeilenumbruch eingefügt wird (*l* für *line*). Diese Option ist jedoch nur für die Hinkonvertierung zum Base64-Format verfügbar (und sinnvoll), nicht jedoch für die Dekodierung.

Die Option *-d* (für *decode*) stellt die Umkehrung der Option *-e* dar. Hier wird also abhängig von der gewählten Option *-d16* oder *-d64* eine hexadezimale oder eine Base64-Datei in das ASCII-Format umgewandelt. Wird als letzter Parameter keine Zielformat (ASCII-Datei) spezifiziert, so wird automatisch der Name der Quelldatei zuzüglich der Endung *.asc* gewählt. Bei der Option *-d64* spielt es keine Rolle, ob die Quelldatei (hier also Base64-Datei) mit Zeilenumbrüchen formatiert ist oder nicht (siehe oben Option *-e64l*), d.h. *RSAmagic* akzeptiert und erkennt automatisch den vorliegenden Typ des Base64-Formats.

Um Dateien von hexadezimaler zu Base64-Kodierung (oder umgekehrt) zu transformieren, müssen zwei Aufrufe von *RSAmagic* für den Umweg über die ASCII-Kodierung erfolgen, da eine direkte Transformation nicht möglich ist.

3.2 Erweiterung der Datenstruktur des LDAP-Servers

Neben dem Base64-kodierten Dienst-Objectcode müssen nun auf dem LDAP-Server zusätzlich die Signatur und die Dienstanbieterbeschreibung abgelegt werden. Die Datenstruktur auf dem LDAP-Server wird in der Datei *amnet.schema* festgelegt, welche Bestandteil des AMnet-Systems ist (Pfad *user/ldap/*). Diese Datei wurde im Laufe des Projektes an die neuen Bedingungen angepasst, wobei im Detail folgende Änderungen an der Definition für den Aufbau eines Dienst-Moduls vorgenommen wurden (es ist nur der relevante Teil der Datei dargestellt) :

```
#####
## ServiceModule ##
#####

## attribute amnetModuleName
attributetype ( 1.3.6.1.4.1.87.2.4 NAME 'amnetModuleName'
                DESC 'name of a module'
                SUP description
                SINGLE-VALUE )

## attribute amnetModuleData
attributetype ( 1.3.6.1.4.1.87.2.5 NAME 'amnetModuleData'
                DESC 'represents the modules data'
                SYNTAX 1.3.6.1.4.1.1466.115.121.1.40
                SINGLE-VALUE )
# 1.3.6.1.4.1.1466.115.121.1.40 stands for arbitrary octets

## attribute amnetModuleType
attributetype ( 1.3.6.1.4.1.87.2.14 NAME 'amnetModuleType'
                DESC 'modules type'
                SUP description
                SINGLE-VALUE )

## attribute amnetModuleFormat
attributetype ( 1.3.6.1.4.1.87.2.15 NAME 'amnetModuleFormat'
                DESC 'modules format'
                SUP description
                SINGLE-VALUE )

## attribute amnetModuleSignature
attributetype ( 1.3.6.1.4.1.87.2.16 NAME 'amnetModuleSignature'
                DESC 'modules signature'
                SUP description
                SINGLE-VALUE )

## attribute amnetModuleOwner
attributetype ( 1.3.6.1.4.1.87.2.17 NAME 'amnetModuleOwner'
                DESC 'modules owner'
                SUP description
                SINGLE-VALUE )

## objectclass amnetServiceModule
objectclass ( 1.3.6.1.4.1.87.1.2
              NAME 'amnetServiceModule'
              SUP top
              MUST ( amnetModuleName $ amnetModuleData )
              MAY ( amnetModuleType $ amnetModuleFormat $ amnetModuleSignature $
                    amnetModuleOwner ) )
```

Die blau markierten Stellen wurden in die Datei *amnet.schema* eingefügt, um zwei zusätzliche Attribute für ein Dienstmodul zu definieren : *amnetModuleSignature* für die durch *RSAmagic* erzeugte Signatur (siehe 3.1.3) und *amnetModuleOwner* für die Dienstanbieterbeschreibung, welche für die Signierung temporär an den Dienst-Objectcode angehängt wurde (siehe 3.1.2).

Hingewiesen werden soll hier auf die Tatsache, dass beide neu eingeführten Attribute für Signatur und Dienstanbieterbeschreibung optionale Komponenten darstellen (siehe *MAY*-Definition in den letzten Zeilen des oben dargestellten Dateiausschnittes). Die Bedeutung dessen wird ersichtlich, wenn man in Betrachtung zieht, dass Dienste für bisherige AMnet-Versionen nicht signiert wurden und in dieser unsignierten Form auch noch auf LDAP-Servern gespeichert sind. Durch die Optionalität der Signierung können diese Dienste auch unter der neuen AMnet-Version verwendet werden, ohne sie nachträglich durch die jeweiligen Dienstanbieter signieren lassen zu müssen. Auf diese Weise ist die Kompatibilität zu früheren AMnet-Versionen sichergestellt !

Ein Dienst wird mittels des AMnet-Skriptes *add* auf den LDAP-Server geladen, wobei eine speziell für den Dienst angepasste Konfigurationsdatei (*ldif*-Datei) erstellt werden muss. Diesbzgl. hat sich in der neuen AMnet-Version nicht viel geändert - man muss lediglich die Signatur und die Dienstanbieterbeschreibung zusätzlich in die Konfigurationsdatei einfügen (siehe blaue Markierungen) :

```
## service description
dn: amnetServiceName=MPEG-Filter,dc=tm,dc=uka,dc=de
objectClass: AMnetService
amnetServiceName: MPEG-Filter
amnetDescription: filter for scaling MPEG streams
amnetCheckClass:< file:///.../amnet/user/signalling/CLASSES/Check.class.64
amnetOS: Unix/Linux

## first (and only) module
dn: amnetModuleName=libmpeg.so,amnetServiceName=MPEG-Filter,
    dc=tm,dc=uka,dc=de
objectClass: amnetServiceModule
amnetModuleName: libmpeg.so
amnetModuleData:< file:///.../amnet/user/modules/LIB-Linux/libmpeg.so.64
amnetModuleType: shared library
amnetModuleSignature: 50D018CD79D103CC7E23FBBFC0FD90963FDA36772BEDB6B3D893B
amnetModuleOwner: mSysCom

## first parameter: quality
dn: amnetParamName=Quality,amnetModuleName=libmpeg.so,
    amnetServiceName=MPEG-Filter,dc=tm,dc=uka,dc=de
objectClass: amnetParameter
amnetParamName: Quality
amnetParamType: int
amnetParamValue: 3
```

Die (im Vergleich zur Dienstanbieterbeschreibung) relativ lange Signatur kann mit der Drag&Drop-Funktion der Maus eingefügt werden - in der Form, wie sie durch *RSAmagic* erzeugt wurde, d.h. in hexadezimaler Kodierung ! Die Dienstanbieterbeschreibung wird in der Form geschrieben, wie sie bei der Signierung des Dienstes an den Dienst-Objectcode angefügt wurde. Dementsprechend muss der Anwender selbst dafür sorgen, dass hier keine Sonderzeichen enthalten sind.

Wichtig ist hier zu beachten, dass beide Attribute auch tatsächlich direkt an die Definition des jeweiligen Moduls angehängt werden - und damit keine eigenen Modul-Parameter darstellen (es besteht ein Unterschied zwischen Attributen und Parametern, denn Parameter besitzen selbst Attribute - siehe *quality*-Parameter) ! Die komplette *ldif*-Datei findet man im Projektverzeichnis *amnet/*. Werden mehrere Module für einen Dienst definiert, so müssen entsprechend für jedes Modul dessen Signatur und Dienstanbieterbeschreibung eingetragen werden. Bisher wurde immer davon gesprochen, dass Dienste signiert werden, jedoch handelt es sich bei genauerer Betrachtung stets um ein Modul eines Dienstes. In den meisten Fällen ist es jedoch so, dass ein Dienst aus genau einem Modul besteht, so dass in der Praxis keine Unterscheidung zwischen Dienst und Modul getroffen wird. Nimmt man es genau, so ist das Thema des Projektes nicht Dienst- sondern Dienstmodulsignierung ! Ein Dienstmodul stellt eine konkrete Datei, den Objectcode dar (z.B. *libmpeg.so*), deren Dateiname ebenso als Attribut in der Moduldefinition aufgeführt wird.

Mit dem Skript *del* werden Dienste vom LDAP-Server entfernt. Auch das *del*-Skript benötigt analog zum *add*-Skript eine speziell auf den zu behandelnden Dienst zugeschnittene Konfigurationsdatei. Da jedoch sowohl Signatur als auch Dienstanbieterbeschreibung als Attribute definiert wurden, müssen diese hier nicht explizit dem *del*-Skript mitgeteilt werden (Parameter sind die niedrigste Instanz, welche es noch in der *del*-Konfigurationsdatei zu erwähnen gilt). D.h. bzgl. des Entfernens eines Dienstes vom LDAP-Server hat sich im Vergleich zu früheren AMnet-Versionen nichts geändert !

3.3 Verifikation eines Dienstes

Die Dienstverifikation wird durch einen Server gewährleistet, welcher Anfragen vom AMnet-System entgegennimmt. Der Server selbst wird hier der Einfachheit halber nicht zum AMnet-System gehörig gerechnet, obwohl ohne ihn signierte Dienste nicht ausgeführt werden können ! Der Grund dafür besteht darin, dass zwar AMnet im geschilderten Fall nicht ohne den Server auskommt, jedoch der Server allein auch für andere Anwendungen genutzt werden kann, denn seine Anfrageschnittstelle ist sehr allgemein gehalten - also nicht AMnet-spezifisch. (Genauso verhält es sich mit dem Signierungstool *RSAmagic*.)

Der Verifikationsserver wird wie folgt gestartet :

```
verify {-p <server-port>} {-d <database>} {-v}
```

Ein Beispielaufruf ist :

```
verify -p 5120 -d public_keys.txt -v
```

Alle Parameter sind optional und können in beliebiger Reihenfolge angegeben werden.

Durch die Option *-p* wird der TCP-Port des Servers festgelegt. Erfolgt keine Angabe diesbzgl. durch den Anwender, so wird standardmässig der Wert 3490 gewählt. Hier muss beachtet werden, dass die Wahl des Server-Ports mit der Konfiguration des AMnet-Systems abgestimmt werden muss ! Das AMnet-System repräsentiert bzgl. des Verifikationsserver einen Client, welcher den Server-Port kennen muss. Auf der AMnet-Seite wird der Server-Port in einer Konfigurationsdatei festgelegt. Die dazu notwendigen Konfigurationsbefehle werden in der Dokumentation von Mathias Bohge detailliert beschrieben.

Die Option *-d* spezifiziert die public-key-Datenbank. Bei jedem Verifikationsvorgang liest der Server aus der Datenbank den zur Dienstanbieterbeschreibung passenden public-key.

Das Format der Datenbank sieht wie folgt aus (für jede Zeile): *<owner> <public-key>*

Sowohl Dienstanbieterbeschreibung (*Owner*) als auch public-key müssen in der Form angegeben werden, wie sie beim Signierungsvorgang vom Anwender gewählt bzw. von *RSAmagic* erzeugt wurden. (Der public-key muss also in hexadezimaler Form angegeben werden !)

Ein Beispiel-Eintrag in der Datenbank könnte wie folgt aussehen :

```
mSysCom 50D018CD79D103CC7E23FBBFC0FD90963FDA36772BEDB6B3D893BBEE706FA3F
```

Im Projektwurzverzeichnis ist bereits eine Datenbank *verify.keys* vorhanden, welche obigen Eintrag enthält.

Es soll hier noch ein wichtiger Umstand erwähnt werden : Eine Änderung der Datenbank muss nicht notwendig ein Neustarten des Servers zur Folge haben ! Bei jeder Verifikation eines Dienstes wird der passende public-key aus der Datenbank gelesen (sofern dieser dort vorhanden ist), d.h. beim Start des Servers wird der zu diesem Zeitpunkt existierende Datenbankzustand nicht in den Speicher gelesen, um mit ihm später Verifikationsanfragen zu bearbeiten. Somit hat eine Änderung der Datenbank unmittelbaren Effekt auf den laufenden Server !

Der dritte Parameter *-v* (für *verbose*) ermöglicht die Ausgabe von Zustandsmeldungen während der Laufzeit des Servers (empfehlenswert).

Da der Verifikationsserver *verify* auch unabhängig von AMnet verwendet werden kann, soll hier kurz die Anfrageschnittstelle beschrieben werden. Als Eingabe erwartet der Server vom Client folgende Daten (in dieser Reihenfolge) :

- Länge der zu verifizierenden Daten (32-Bit-Integer)
- zu verifizierende Daten (bei AMnet durch den Dienst-Objectcode repräsentiert)
- Länge der Signatur (32-Bit-Integer)
- Signatur (in hexadezimaler Kodierung)
- Länge des Datenbesitzers bzw. -anbieters (32-Bit-Integer)
- Datenbesitzer bzw. -anbieter (bei AMnet durch die Dienstanbieterbeschreibung gegeben)

Es ist zu beachten, dass die Signatur durch Signierung eines Plaintext entstand, welcher aus den eigentlichen Anwendungsdaten (oben als zu verifizierende Daten bezeichnet) erweitert um die Angabe des Datenbesitzers bzw. anbieters gebildet wurde (siehe 3.1.2). Der zu verifizierende Plaintext wird hier also nicht als ein Block, sondern in zwei getrennten Teilen zum Server gesendet, welcher daraus temporär den Plaintext neu aufbaut.

Als Rückgabewert (1 Byte) kommen vier Alternativen in Frage :

- 0 : *Verifikation erfolgreich*
- 1 : *Verifikation fehlgeschlagen*
- 2 : *Dienstanbieter unbekannt* (es wurde kein passender Eintrag in der Datenbank gefunden)
- 3 : *interner Server-Fehler*

Bzgl. des Alternative des internen Fehlers muss darauf hingewiesen werden, dass ein solcher vom Server nicht im Falle eines fehlerhaften public-key oder einer fehlerhaften Signatur angezeigt wird ! Unter internen Fehlern werden hier vielmehr solche Fehler verstanden, welche nicht aus den Funktionen der Crypto-Bibliothek resultieren (genau das wäre bei den genannten Fehler-Beispielen der Fall), sondern vielmehr aus Dingen wie z.B. unzureichender Speicher oder Fehler beim Dateizugriff. Auf den tatsächlichen Grund eines internen Fehlers kann anhand des Rückgabewertes nicht weiter geschlossen werden.

Die oben erwähnten Fehler - fehlerhafter Schlüssel oder fehlerhafte Signatur - können leider aufgrund der Funktionalität der Crypto-Bibliothek nicht ausgewertet werden. Vielmehr tritt in solchen Fällen ein unvorhersagbarer Zustand ein (z.B. Aufhängen oder Terminieren des Server-Prozesses oder gar Rückgabe eines unbestimmten Wertes an den Client). Ähnliches gilt übrigens auch für das Signierungstool *RSAmagic*. Es ist also wichtig, nur solche Zeichenketten als Schlüssel zu verwenden, welche auch tatsächlich Schlüssel sind, und in Bezug auf die Signatur darauf zu achten, dass diese nur mit solchen Schlüsseln verifiziert wird, welche für Signaturen dieser Länge erstellt wurden (die Länge einer Signatur wird im Schlüssel spezifiziert - siehe 3.1.1).

3.4 Einführung in die Projektumgebung

Die komplette Projektarbeit befindet sich im Archiv *amnet_project.zip*, welches zunächst entpackt werden muss. Dafür muss kein eigenes Verzeichnis erstellt werden, da ein solches Hauptverzeichnis bereits im Archiv enthalten ist (dessen Name ist *amnet_project*).

Entpacken Sie also das Archiv wie folgt : `unzip amnet_project.zip`

Wechseln Sie danach in das erstellte Verzeichnis *amnet_project*.

Führen Sie jetzt alle nachfolgenden Anweisungen aus, um Erfahrungen mit der Funktionalität der in diesem Projekt entwickelten Anwendungen zu sammeln ! Das eigentliche AMnet-System ist hierfür nicht notwendig, da sich im Projektverzeichnis ein Client befindet, welcher AMnet simulieren kann - zumindest was die Kommunikation mit dem Server betrifft. Im Verzeichnis befinden sich bereits vorkompilierte Binaries aller im Projekt entwickelten Anwendungen. Diese wurden unter Linux 2.4.14 erstellt, d.h. entsprechend dem von Ihnen verwendeten System müssen diese Anwendungen evtl. neu kompiliert werden. Die Quellcodes befinden sich im Unterverzeichnis *src/*, einschliesslich der Crypto-Bibliothek *Crypto++ 4.1* ! (Auf den Kompilierungsvorgang soll hier nicht eingegangen werden - für alle Quellcodes existieren Makefiles bzw. für die Crypto-Bibliothek ein GNU-Makefile.)

3.4.1 Signierung mit dem Tool *RSAmagic*

Zunächst ist es notwendig, ein Schlüsselpaar zu generieren. Danach kann mittels des erzeugten private-key eine beliebige Datei signiert werden. Da wir später jedoch auch den Verifikationsserver *verify* testen wollen, bietet es sich an, vor der Signierung noch eine Dienstanbieterbeschreibung an die zu signierende Datei anzuhängen. Als Beispiel verwenden wir *RSAmagic* selbst bzw. eine Kopie dieser Datei, d.h. bei späterer Dienstsignierung muss hier der Dienst-Objectcode verwendet werden. Führen Sie also bitte folgende Anweisungen aus (Sie befinden sich im Verzeichnis *amnet_project*) :

```
RSAmagic -k 1024 mykey
cp RSAmagic plaintext.asc
RSAmagic -at plaintext.asc testProvider
RSAmagic -s plaintext.asc mykey.pvk mysig.hex
```

In der ersten Zeile werden zwei Schlüssel *mykey.pvk* und *mykey.pbk* erzeugt - beide für Signaturen der Länge 1024 Bits bestimmt. In den nächsten zwei Zeilen wird der Plaintext vorbereitet. Man hätte auch die Originaldatei *RSAmagic* erweitert um die Dienstanbieterbeschreibung *testProvider* signieren können, allerdings hätte man danach letzteres wieder entfernen müssen (ebenfalls mittels der *RSAmagic*-Funktionalität) und ausserdem würde sich durch diese Verfahrensweise das Datum des letzten Schreibzugriffs der Originaldatei ändern, was unter Umständen nicht erwünscht ist. Dies sollte für spätere Dienstsignierungen beachtet werden.

Als Resultat liegt jetzt die Signatur *mysig.hex* vor, welche noch zur Kontrolle mittels *RSAmagic* verifiziert werden soll :

```
RSAmagic -v plaintext.asc mykey.pbk mysig.hex
```

Der Aufruf sollte folgende Ausgabe zur Folge haben : *verification was successful...*
Im Falle einer anderen Ausgabe sollte der gesamte Vorgang wiederholt werden, da sehr wahrscheinlich bestimmte Dateiinhalte unbewusst verändert und dadurch von *RSAmagic* falsch interpretiert wurden.

3.4.2 Verifikation mit dem Server *verify*

Nun soll eine Anfrage des AMnet-Systems simuliert werden, welche vom Server *verify* zu bearbeiten ist. Da der *verify*-Server anhand der Dienstanbieterbeschreibung den passenden public-key aus der Datenbank (hier werden wir sie *verify.keys* nennen) lesen muss, ist es zunächst notwendig, den vorhin erzeugten public-key *mykey.pbk* in diese Datenbank einzutragen. Dies kann mit der Drag&Drop-Funktion der Maus geschehen. Ein häufiger Fehler besteht darin, dass man statt des public-key die erzeugte Signatur in der Datenbank ablegt, da beide in hexadezimaler Kodierung vorliegen und dadurch eine Verwechslung nachträglich nur schwer auffällt.

Im Projektverzeichnis existiert bereits eine Datenbank *verify.keys* mit einem Beispieleintrag. Analog dazu muss jetzt ein zweiter Eintrag hinzugefügt werden :

```
testProvider 50D018CD79D103CC7E23FBBFC0FD90963FDA36772BEDB6B3D893BBEE706FA
```

Der hier aufgeführte public-key ist nur ein zufälliger Schlüssel, an dessen Stelle der von Ihnen erzeugte eingefügt werden muss (die Dienstanbieterbeschreibung *testProvider* muss jedoch exakt, d.h. mit Gross-/Kleinschreibung übernommen werden). Zwischen Dienstanbieterbeschreibung und public-key wird ein Leerzeichen eingefügt. Achten Sie beim Einfügen des public-key darauf, dass er nicht auf mehrere Zeilen verteilt werden darf. (Oft ist der public-key in der hexadezimalen Kodierung länger als eine Zeile im Texteditor und wird durch die Drag&Drop-Funktion der Maus mit Zeilenumbrüchen eingefügt.)

Als nächster Schritt wird der Server gestartet :

```
verify -p 3490 -d verify.keys -v
```

Der Start des Servers hätte auch vor dem Einfügen des Datenbankeintrags erfolgen können, da der Server bei jeder Verifikation eines Dienstes erneut auf die Datenbank zugreift !

Um eine Anfrage des AMnet-Systems zu simulieren, verwenden wir das Tool *testclient*.

Führen Sie also bitte folgende Anweisung aus :

```
testclient localhost:3490 RSMagic mysig.hex testProvider
```

Hier werden als Parameter die Originaldatei *RSAmagic* und die Dienstanbieterbeschreibung getrennt von einander aufgeführt, da die Anfrageschnittstelle des Servers dies auch so verlangt (siehe 3.3). Sowohl auf der Client- wie auch auf der Server-Seite müssten nun Zustandsmeldungen erscheinen. Wurden alle Anweisungen von Ihnen korrekt ausgeführt, so müssten auf beiden Seiten jeweils die letzten Meldungen eine erfolgreiche Verifikation bestätigen. Der Server hat somit den Wert 0 an den Client geschickt.

3.4.3 Anpassen des AMnet-Systems

Nachdem Sie den *verify*-Server erfolgreich getestet haben, können Sie nun das AMnet-System, insbesondere die Konfiguration des LDAP-Servers, an den Signierungsmechanismus anpassen. Es wird vorausgesetzt, dass Sie die in diesem Projekt entwickelte AMnet-Version installiert haben. Anleitungen dazu finden Sie in der Dokumentation von Mathias Bohge.

Die Anpassung besteht lediglich darin, dass Sie die Datei *amnet.schema* aus dem Projektverzeichnis *amnet/* in das Verzeichnis *user/ldap/* des AMnet-Systems kopieren, d.h. die dort bereits vorhandene Datei gleichen Namens überschreiben.

Da somit die Definition des Aufbaus eines Dienst-Moduls geändert wurde, müssen entsprechend bei jeder Ablage eines Dienstes auf dem LDAP-Server die zusätzlichen Attribute Signatur und Dienstanbieterbeschreibung in die *ldif*-Datei für das *add*-Skript eingefügt werden (siehe 3.2). Ferner muss die Konfiguration des AMnet-Systems an den verwendeten TCP-Port des Verifikationsservers angepasst werden. Hierzu muss wieder auf die Dokumentation von Mathias Bohge verwiesen werden.

Viel Glück !