AnonDienst - Design und Implementierung

Stefan Köpsell

9. Januar 2004

1 Ziel und Motivation

Ziel ist die Entwicklung eines benutzbaren und sicheren Systems, welches es einem Internetnutzer ermöglicht, anonym bzw. unbeobachtbar Dienste im Internet zu nutzen. Dabei soll Schutz selbst gegen starke Angreifer bestehen, die z.B. in der Lage sind, große Teile des Internets bzw. des Systems selbst zu kontrollieren. Als einziges benutzbares derartiges System existierte zu Beginn der Arbeit das Freedom-Netz der kanadischen Firma Zero-Knowledge. Freedom war auf den amerikanischen Kontinent konzentriert, so daß schlechte Verfügbarkeit und mangelnder Durchsatz zu eingeschränkter Benutzbarkeit in Europa führten [ct-Artikel]. Mittlerweile hat Zero-Knowledge den Betrieb von Freedom aus wirtschaftlichen Gründen eingestellt.[link]

2 Grundlegende Architektur-Entscheidungen

Die Basis des System bilden die von David Chaum entwickelten Mixe [Chau81], da zu dieser Anonymisierungstechnik bereits umfangreiche Forschungs- und Entwicklungsarbeit durch die Arbeitsgruppe geleistet wurde, auf deren Ergebnisse zurückgegriffen wird. Als Verknüpfungsform der Mixe wurde die Kaskade gewählt. Sie besitzt gegenüber dem Mixnetz (freie Folge) eine geringere Komplexität, was zu einfacherer Analyse bezüglich Sicherheit und zu erwartetem geringeren Aufwand in der Implementierung führt. Es hat sich gezeigt, daß die freie Folge eine Reihe von Sicherheitsproblem birgt, die bei Kaskaden nicht existieren [paper].

Um synchrone Echtzeitkommunikation zu ermöglichen, kommt eine Erweiterung zum Einsatz, die auf den in [ISDM-Mixe] beschriebenen Ideen basiert. Dabei werden sogenannte *Kanäle* aufgebaut (im folgenden als *MixKanal* bezeichnet. Diese Kanäle realisieren zuverlässige verbindungsorientierte Byteströme zwischen Sender und Empfänger.

Der Anonymisierungsdienst bietet nur die Möglichkeit, Klassen von Proxies zu adressieren, d.h. ein Nutzer kann nur entscheiden, daß seine Daten z.B. an einen HTTP-Proxy geschickt werden. Die Proxies sind für die weitere Verarbeitung der Daten gemäß dem jeweiligen Proxy-Protokoll zuständig. Ein Vorteil ist, daß die Verarbeitung des Proxy-Protokolls auf erprobte und ausgereifte Komponenten ausgelagert werden kann, die oft noch zusätzliche Funktionalität bieten (Zugriffskontrolle, Ressourcenbegrenzung, Caching etc.)

Neben den Mix-Servern, die den grundlegenden Anonymisierungsdienst bilden, wird eine Client-Komponente benötigt. Diese muß auf den Endsystemen (z.B. Rechner der Nutzer) installiert sein. Sie ist für den Transfer der anonym zu übertragenden Daten zuständig und bereitet diese dem Protokoll des zugrundeliegenden Anonymisierungsdienstes gemäß auf.

Um die Benutzung des Anonymisierungsdienstes zu erleichtern und dem Nutzer eine Rückmeldung über sein aktuelles Schutzniveau zu geben, wurde ein dritter Bestandteil in

das Gesamtsystem aufgenommen – der sogenannte *InfoService*. Dieser ist mit einer Datenbank vergleichbar und hält abrufbar Informationen über die aktuell verfügbaren Mix-Kaskaden, deren Auslastung etc. bereit. Die Client-Komponente kann mit Hilfe der beim InfoService vorliegenden Daten dem Nutzer eine Vorstellung über seine momentane „Anonymität" vermitteln.

3 Grundlegende Entwurfs-Entscheidungen

Das System soll sowohl benutzbar als auch sicher sein. Dabei läßt sich keine generelle Priorität von Benutzbarkeit gegenüber Sicherheit festlegen. Ziel ist eine sinnvolle Verhältnismäßigkeit, da weder ein unbenutzbares aber sicheres noch ein unsicheres aber benutzbares System hilfreich sind.

Benutzbarkeit aus Sicht des Endanwenders beinhaltet, daß Installation und Konfiguration der Client-Komponente einfach durchzuführen sind. Außerdem darf nicht das Gefühl aufkommen, daß die Nutzung des Anonymisierungsdienstes den Zugriff auf das Internet stark einschränkt. Dies bedeutet z.B. das Durchsatz und Latenzzeit akzeptabel sind, was ein wichtiges Kriterium beim Entwurf des Systems ist.

Um möglichst vielen Menschen den Zugang zum Anonymisierungsdienst zu ermöglichen, ist es notwendig, daß die Client-Komponente auf vielen verschiedenen Hardware- und Betriebsystemplattformen ausgeführt werden kann. Daher wurde als Programmiersprache und Ausführungsumgebung Java™ gewählt. Speziell fiel die Entscheidung zu Gunsten von Java 1.1. Neuere Versionen von Java bieten zwar einen deutlich größeren Funktionsumfang, der insbesondere in Hinblick auf die kryptographischen Primitive sehr hilfreich wäre. Andererseits wurde zu Beginn des Projektes unter MacOS und zahlreichen Unix-Varianten nur Java 1.1 unterstützt (siehe Tabelle).

Ein wichtiger Aspekt der Benutzbarkeit ist die Benutzungsschnittstelle. Die Client-Komponente soll eine graphische Oberfläche besitzen, die die typischen Anforderungen wie z.B. intuitive Bedienbarkeit, Ergonomie, Aufgabenangemessenheit etc. erfüllt. Java 1.1 bietet zwei verschiedene Ansätze, um Oberflächen zu erzeugen. Zum einen kann mit Hilfe der Funktionen des Abstract Windows Toolkit (AWT) auf die vom Betriebssystem bereitgestellten Elemente wie Fenster, Buttons, Textfelder etc. zurückgegriffen werden. Zum anderen bietet sich mit den Java Foundation Classes (JFC oder auch Swing) die Möglichkeit, betriebssystemunabhängig Oberflächen zu gestalten. Dies wird dadurch erreicht, daß Swing die einzelnen Bestandteile selbst zeichnet. Jetzt fehlt noch eine Menge...

Die Mixe bilden die zentralen Verarbeitungseinheiten des Gesamtsystems. Ein Mix muß die Datenströme sehr vieler Nutzer verarbeiten. Deshalb wird beim Design und der Implementierung sehr großer Wert auf Optimierung bzgl. Ausführungsgeschwindigkeit gelegt. Als Programmiersprache wurde daher C++ gewählt. Es ist ausdrücklich kein Ziel wiederverwertbaren Code zu schaffen. Innerhalb des Projektes erfolgt eine Wiederverwendung bei ähnlichen Problemen (z.B. mittels Generaliserung und Vererbung etc.) nur, wenn sich dadurch keine Verschlechterung der Performance ergibt. Andernfalls wird jeweils eine eigene, spezialisierte Klasse entwickelt. Die Robustheit ist gegenüber der Performance ebenso zweitrangig. Dies bedeutet beispielsweise, daß beim Entwurf einer Methode mehrere Vorbedingungen definiert werden, deren Einhaltung die Methode selbst jedoch nicht überprüft (also z.B. ob Parameter Elemente des zulässigen Wertebereiches sind etc.) (TODO: Benutzbarkeit, Plattformunabhängigkeit, keine neuen C++-Features)

So jetzt fehlt noch eine ganze Menge....

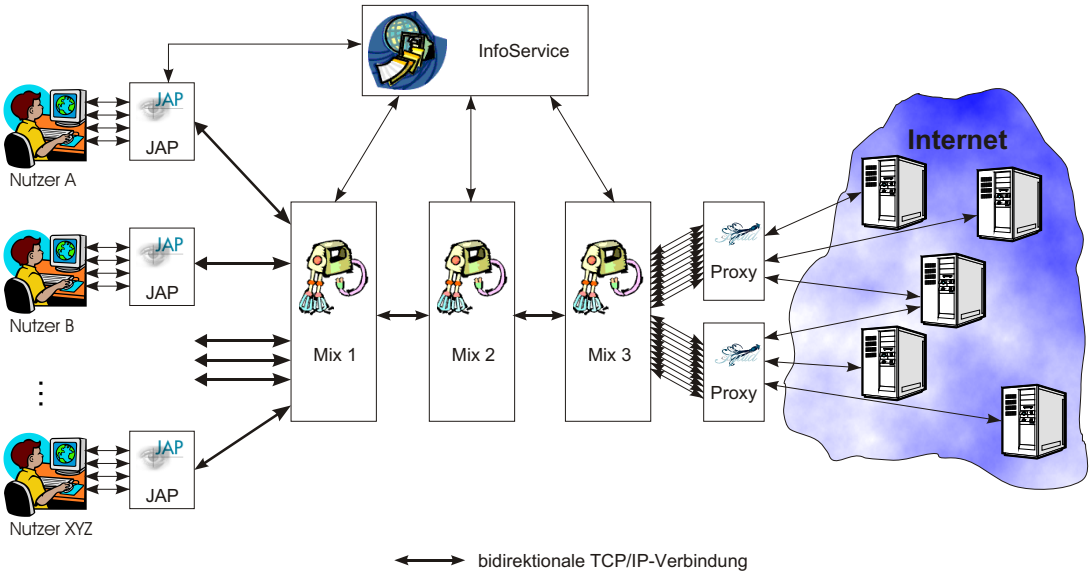

Abbildung 1: Architektur des Systems

4 Beschreibung des implementierten Mix-Systems und des Kommunikationsprotokolls

Wie oben erwähnt besteht das gesamte System aus dem JAP als Client-Komponente, dem InfoService und den Mixen (Abb. 1). Für die nachfolgenden Erläuterungen wird als Beispiel die Anonymiserung von Web-Zugriffen gewählt.

Der JAP arbeitet als lokaler Proxy für einen Browser, d.h. er nimmt die Anfragen des Browsers entgegen, sendet sie über den Anonymisierungsdienst, empfängt vom Anonymisierungsdienst die Antwort des Web-Servers und leitet diese an den Browser weiter.

Im folgenden wird als Mix ein Prozeß verstanden, der auf einem Rechner ausgeführt wird. JAP und die Mixe kommunizieren über TCP/IP-Verbindungen. Die Mixe sind zu Kaskaden zusammengeschaltet. Dabei ist jeder Mix Bestandteil höchstens einer Kaskade. Ist ein Mix nicht Teil einer Kaskade, so wird er als *freier Mix* bezeichnet. Als *erster Mix* wird ein Mix bezeichnet, der mit den JAP's und genau einem anderen Mix kommuniziert. Als *mittlerer Mix* wird ein Mix bezeichnet, der mit genau zwei anderen Mixen kommuniziert und als *letzter Mix* wird ein Mix bezeichnet, der mit genau einem anderen Mix und verschiedenen Proxies kommuniziert. Die Verbindung zum eigentlichen Ziel (Kommunikationspartner, Internet-Dienst) wird durch die Proxies hergestellt. Zwischen den Mixen einer Kaskade besteht genau eine TCP/IP-Verbindung. Zwischen den JAP's und einem ersten Mix besteht pro JAP genau eine TCP/IP-Verbindung.

Die grundlegende Kommunikationseinheit zwischen JAP und den Mixen ist das *MixPaket*. Ein MixPaket gehört dabei zu genau einem *MixKanal*. Ein MixKanal (kurz: Kanal) ist eine virtuelle Zusammenfassung mehrerer MixPakete. Im letzten Mix ist mit jedem MixKanal jeweils genau eine TCP/IP-Verbindung (mit einem Proxy) assoziiert. Ein MixPacket ist 998 Bytes groß. Es besitzt einen Header von 6 Bytes und einen Datenteil von 992 Bytes. Die ersten vier Bytes des Headers bilden eine Kanal-ID, die restlichen 2 Bytes sind für Flags reserviert (Abb. 2). Über die Kanal-ID erfolgt die Zuordnung des MixPaketes zum MixKa-

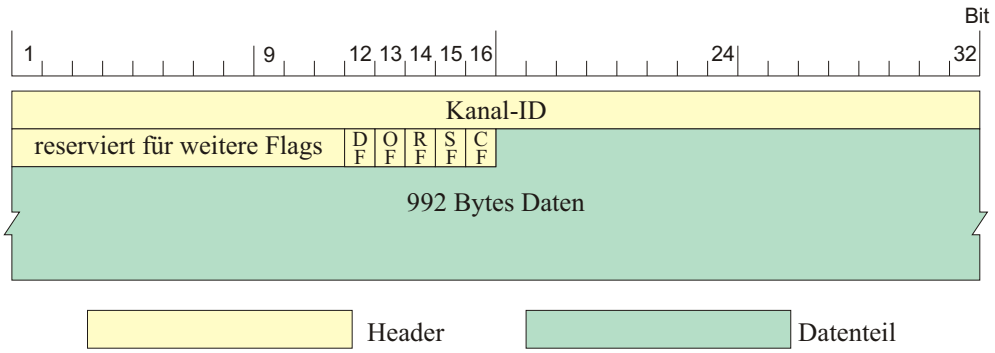

Abbildung 2: Allgemeiner Aufbau eines MixPaketes

nal. Zwischen den Mixen werden die ersten 16 Bytes jedes MixPaketes mit AES im OFB Modus verschlüsselt, um für einen Außenstehenden die Zuordnung MixPacket ↔ MixKanal zu erschweren.

Zur Zeit sind folgende Flags definiert:

- das *Open-Flag* (OF),
- das *Close-Flag* (CF),
- das *Dummy-Flag* (DF),
- das *Channel-Suspend-Flag* (SF)
- und das *Channel-Resume-Flag* (RF).

Ist das Open-Flag gesetzt, so ist der Datenteil hybrid verschlüsselt andernfalls rein symmetrisch. Im ersten Fall sind die ersten 128 Byte asymmetrisch für den jeweiligen Mix verschlüsselt. Als Verschlüsselungsverfahren wird Plain-RSA verwendet. Die folgenden 864 Bytes sind symmetrisch mit AES im OFB Modus verschlüsselt. Die ersten 16 Byte des (entschlüsselten) Datenteils bilden den Schlüssel für das symmetrisches Verschlüsselungsverfahren.

Der Datenteil besitzt im letzten Mix bei nicht gesetztem Open-Flag folgende Struktur (Abb. 3):

- 3 Byte Header
- 989 Bytes Nutzdatenteil (Payloadteil)

Ist das Open-Flag gesetzt, so befindet sich vor dem Payloadheader noch der symmetrische Schlüssel mit 16 Byte Länge, so daß der Payloadteil maximal 973 Bytes groß ist[1]. Die ersten zwei Bytes des Payloadheaders bilden einen 16 Bit Integer (Network Byte Order) und beschreiben die tatsächliche Länge $len_{Payload}$ der Nutzdaten (Payload). Das dritte Byte bestimmt den Typ des Payload. An Hand des Typs wird entschieden, zu welchem Proxy die Nutzdaten geschickt werden.

[1]Die tatsächliche Größe hängt von Länge der Mix-Kaskade ab. Sie verringert sich für jeden zuvor durchlaufenen Mix um weitere 16 Byte.

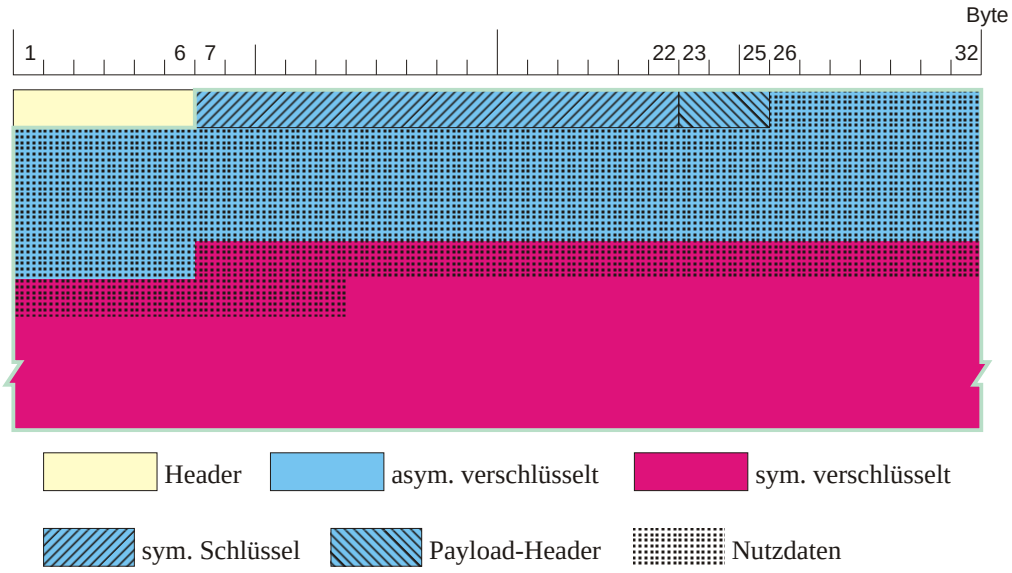

Abbildung 3: Aufbau eines MixPaketes im letzten Mix bei gesetztem Open-Flag

Auf Grund der Größe des Datenteils ergibt sich, daß $0 \leq len_{Payload} \leq 989$. Die ersten 6 Bit der Payload-Längen-Angabe sind demzufolge immer Null[2].

Neue Kanäle werden nur durch den JAP geöffnet. Der Beginn eines neuen Kanals wird durch das Setzen des Open-Flags signalisiert. Das Ende eines Kanals wird durch das Setzen des Close-Flags signalisiert. Ein Kanal wird nie durch einen mittleren Mix geschlossen.

Ist das Close-Flag nicht gesetzt, so enthält der Payloadteil tatsächlich Nutzdaten.

Channel-Suspend- und Channel-Resume-Flag werden nur durch den ersten Mix gesetzt und nur duch den letzten Mix ausgewertet. Ein gesetztes Channel-Suspend-Flag signalisiert dabei, daß vom letzten Mix keine Daten mehr für diesen Kanal gesendet werden sollen. Ein Channel-Resume-Flag signalisiert, daß die Datenübertragung fortgesetzt werden kann.

4.1 Funktionsweise am Beispiel eines Web-Seiten Abrufs

Im folgenden soll zur Verdeutlichung das Mix-Protokolls an Hand eines Beispiels erläutert werden. Dabei wird von einer Mix-Kaskade bestehend aus zwei Mixen (Mix_1 und Mix_2) ausgegangen. Ferner soll im Browser die Web-Seite `http://anon.inf.tu-dresden.de/` abgerufen werden.

Der Browser baut dazu zunächst eine TCP/IP-Verbindung zum JAP auf und sendet folgende Anfrage *req*:

```
GET http://anon.inf.tu-dresden.de/ HTTP/1.0
```

Der JAP empfängt diese 45 Bytes und konstruiert daraus ein MixPaket wie folgt:

1. Er erzeugt eine 16 Byte große Zufallsbitfolge, die als symmetrischer Schlüssel k_2 mit Mix_2 für den neu zu etablierenden MixKanal genutzt werden.

[2]Diese Bits könnten zum Beispiel bei einer (temporären) Protokollerweiterung der Signalisierung zwischen JAP und letztem Mix dienen.

2. Er erzeugt den Payload-Header ph_{Req} bestehend aus der Länge der zu sendenden Anfrage und dem Typ (Proxy-Art) der Daten: $ph_{\text{Req}} = \texttt{0x00|0x2D|0x01}$
3. Er erzeugt eine Byte-Array $temp_1$ bestehend aus dem Schlüssel k_2, dem Payload-Header und der zu sendenden Anfrage aufgefüllt mit Zufallsbytes auf die Länge des Datenteils: $temp_1 = k_2|ph_{\text{Req}}|req|rand(989 - 16 - 3 - 45)$.
4. Er verschlüsselt die ersten 128 Byte von $temp_1$ mit dem öffentlichen Schlüssel von Mix_2.
5. Er verschlüsselt die letzten 992-128 Byte von $temp_1$ mit k_2.
6. Er erzeugt eine 16 Byte große Zufallsbitfolge, die als symmetrischer Schlüssel k_1 mit Mix_1 für den neu zu etablierenden MixKanal genutzt wird.
7. Er bildet ein Byte-Array $temp_2 = k_1|$Ergebnis von 4|Ergebnis von 5.
8. Er verschlüsselt die ersten 128 Byte von $temp_2$ mit dem öffentlichen Schlüssel von Mix_1.
9. Er verschlüsselt 992-128 Byte beginnend ab dem 129. Byte von $temp_2$ mit k_1.
10. Er erzeugt eine zufällige Kanal-ID id_1, die momentan nicht verwendet wird.
11. Er erzeugt ein MixPaket m_1 bestehend aus der Kanal-ID, den Flag-Bytes (Open-Flag gesetzt), dem Ergebnis aus 8 und 9: $m_1 = id1_|\texttt{0x00|0x01}|$Ergebnis aus 8|Ergebnis aus 9.
12. Er speichert das Tripel (id_1, k_1, k_2) in seiner Kanal-Tabelle.
13. Er schickt das MixPaket an den ersten Mix.

Der erste Mix empfängt dieses MixPaket und führt folgende Schritte aus:

1. Er überprüft, ob das Open-Flag gesetzt ist.
2. Er entschlüsselt die ersten 128 Byte des Datenteils mittels seines geheimen Schlüssels.
3. Er entschlüsselt die letzten 992-128 Byte des Datenteils mittels des aus der vorrangegangenen Entschlüsselung gewonnenen symmetrischen Schlüssels k_1.
4. Er erzeugt eine zufällige, momentan nicht verwendete Kanal-ID id_2.
5. Er erzeugt ein MixPaket m_2 bestehend aus id_2, den Flag-Bytes (Open-Flag gesetzt), den letzten 128-16 Byte aus dem Ergebnis von 2, dem Ergebnis aus 3 und 16 zufälligen Bytes: $m_2 = id_2|\texttt{0x00|0x01}|$Ergebnis 2|Ergebnis 3|$rand(16)$.
6. Er fügt das Tripel (id_1, id_2, k_1) seiner Kanal-Tabelle hinzu.
7. Er sendet m_2 an Mix_2.

Hat Mix_2 dieses MixPaket empfangen so verarbeitet er es wie folgt:

1. Er überprüft ob das Open-Flag gesetzt ist.
2. Er entschlüsselt die ersten 128 Byte des Datenteils mittels seines geheimen Schlüssels.
3. Er entschlüsselt die letzten 992-128 Byte des Datenteils mittels des aus der vorrangegangenen Entschlüsselung gewonnenen symmetrischen Schlüssels k_2

4. Er ermittelt den Payload-Header ph_{Req} und extrahiert die Payload-Länge und den Payload-Typ.
5. Er extrahiert die Anfrage req aus den Payload-Daten.
6. Er baut eine TCP/IP Verbindung zu einem HTTP-Proxy auf und sendet req an diesen.
7. Er empfängt die 437 Byte lange Antwort rsp des HTTP-Proxy[3].
8. Er konstruiert den Payload-Header ph_{Rsp} bestehend aus der Länge von rsp und dem Typ: $ph_{\text{Rsp}} = \texttt{0x01|0xB5|0x01}$
9. Er erzeugt den Datenteil $temp_3 = ph_{\text{Rsp}}|rsp|rand(989 - 3 - 437)$.
10. Er verschlüsselt $temp_3$ mit k_2.
11. Er konstruiert des MixPaket m_3 bestehend aus id_2, dem Flag-Bytes (kein Flag gesetzt)und dem verschlüsselten Datenteil.
12. Er sendet m_3 an Mix_1.
13. Er konstruiert das MixPaket m_4 bestehend aus id_2, den Flag-Bytes (Close-Flag gesetzt) und einem zufälligen Datenteil.
14. Er sendet m_4 an Mix_1.

Der erste Mix empfängt die Pakete m_3 und m_4 von Mix_2 und verarbeitet sie wie folgt:

1. Er erkennt, daß in m_3 das Close-Flag nicht gesetzt ist und sucht in seiner Kanal-Tabelle nach dem Tripel, das zu id_2 gehört.
2. Er verschlüsselt den Datenteil mit k_1
3. Er konstruiert das MixPaket m_5 bestehend aus id_1, den Flag-Bytes (kein Flag gesetzt) und dem Ergebnis aus 2.
4. Er sendet das MixPaket m_5 an den Nutzer.
5. Er erkennt, daß in m_4 das Close-Flag gesetzt ist und konstruiert das MixPaket m_6 bestehend aus id_1, den Flag-Bytes (Close-Flag gesetzt) und 992 zufälligen Bytes.
6. Er sendet m_6 an den JAP.
7. Er löscht das Tripel (id_1, id_2, k_1) aus seiner Kanal-Tabelle

Der JAP empfängt die MixPakete m_5 und m_6 und verfährt wie folgt:

1. Er erkennt, daß in m_5 das Close-Flag nicht gesetzt ist und sucht in seiener Kanal-Tabelle nach dem Tripel, das zu id_1 gehört.
2. Er entschlüsselt den Datenteil zunächst mit k_1 und das Ergebnis dann mit k_2.
3. Er bestimmt aus dem entschlüsselten Datenteil den Payload-Header, die Payload-Länge und die Payload-Daten (rsp).
4. Er sendet rsp an den Browser.
5. Er empfängt m_6 und erkennt, daß das Close-Flag gesetzt ist.
6. Er schliesst die TCP/IP-Verbindung zum Browser.

[3]Die Länge der Antwort ist für dieses Beispiel willkürlich gewählt.