UMA PROPOSTA DE EXTENSÃO DA CAMADA SEGURA DE TRANSPORTE (TLS) PARA USO SOBRE DATAGRAMAS DE USUÁRIOS (UDP)

Alessandro Levachof Berim

TESE SUBMETIDA AO CORPO DOCENTE DA COORDENAÇÃO DOS PROGRAMAS DE PÓS-GRADUAÇÃO DE ENGENHARIA DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIAS EM ENGENHARIA DE SISTEMAS E COMPUTAÇÃO.

Aprovada por:

____

Prof. Luís Felipe Magalhães de Moraes, Ph.D.

____

Prof. Jorge Lopes de Souza Leão, Dr.Ing.

____

Prof. Marcio Portes de Albuquerque, Ph.D.

RIO DE JANEIRO, RJ – BRASIL
MARÇO DE 2004

BERIM, ALESSANDRO LEVACHOF

Uma Proposta de Extensão da Camada Segura de Transporte (TLS) Para Uso Sobre Datagramas de Usuários (UDP) [Rio de Janeiro] 2004

V, 196 p. 29,7 cm (COPPE/UFRJ, M.Sc., Engenharia de Sistemas e Computação, 2004)

Tese – Universidade Federal do Rio de Janeiro, COPPE

1. Criptografia
2. *User Datagram Protocol*
3. *Transport Layer Security*
4. *Internet Protocol Security*
5. Linguagem de Programação Java

I. COPPE/UFRJ   II. Título (série)

Dedicatória

A minha mãe, Kalpa.

Agradecimentos

Ao Princípio Supremo, Deus.

Ao meu orientador, o professor Luís Felipe, por tudo que fez por mim desde os tempos de bolsista de iniciação científica do Laboratório RAVEL.

Aos professores Leão e Márcio que aceitaram prontamente participar da minha Banca Examinadora e pelas contribuições ao trabalho.

A todos que ajudaram na realização deste trabalho, a citar alguns: os antigos e novos colegas ravelianos, Nelson Fernandes e Pinaffi, aos amigos João Alexandre e Solimar e a namorada Patrícia.

E finalmente agradeço também aos que se ofereçam para colaborar e não tiveram a oportunidade, foram tantos que não tenho como nominar e nem como lembrar de todos. São nessas oportunidades que percebemos quantos amigos temos nesta jornada de passagem pela Terra.

Resumo da Tese apresentada à COPPE/UFRJ como parte dos requisitos necessários para a obtenção do grau de Mestre em Ciências (M.Sc.)

UMA PROPOSTA DE EXTENSÃO DA CAMADA SEGURA DE TRANSPORTE (TLS) PARA USO SOBRE DATAGRAMAS DE USUÁRIOS (UDP)

Alessandro Levachof Berim

Março/2004

Orientador: Luís Felipe Magalhães de Moraes

Programa: Engenharia de Sistemas e Computação

A Internet está consolidada como instrumento fundamental para comunicação pessoal e empresarial. E sem dúvida, cada vez mais a preocupação com a segurança na troca de informações está presente na agenda de todos. Seja qual for a necessidade de segurança: privacidade, autenticação dos usuários, garantia da integridade da informação veiculada ou outra qualquer, a criptografia dos dados que trafegam na rede de comunicação IP é uma necessidade, vide o comércio web.

Esta tese tem como objetivo propor um novo protocolo de segurança de rede para fornecer o suporte necessário e atualmente ausente para a troca segura de datagramas UDP. Este protocolo fornecerá basicamente as mesmas opções de segurança disponíveis para o protocolo de transporte TCP existentes pelo uso do TLS. Para validar o novo protocolo foi implementada uma API de referência disponibilizada na linguagem de programação Java e testes com uma aplicação cliente/servidor foram realizados. Além disso, foi realizada também uma análise comparativa deste novo protocolo de segurança de rede com outros.

Abstract of Thesis presented to COPPE/UFRJ as a partial fulfillment of the requirements for the degree of Master of Science (M.Sc.)

A PROPOSAL OF EXTENSION OF THE SECURE TRANSPORT LAYER (TLS) FOR USE OVER USER DATAGRAMS (UDP)

Alessandro Levachof Berim

March/2004

Advisor: Luís Felipe Magalhães de Moraes

Department: Systems and Computer Engineering

The Internet is consolidated as fundamental instrument for personal and corporated communication. Undoubtedly, more and more the concern with the security in the exchange of information is present in the calendar of everybody. Whichever the need of security: privacy, the users authentication, assurance of the integrity of the transmitted information or any other one, the cryptography of the data that travels in the network IP communication is a need, see the web commerce.

This thesis has as objective to present a new network security protocol to supply the necessary and now absent support for the secure exchange of UDP datagrams. This protocol will supply basically the same available options of security existent for the TCP transport protocol with the use of TLS. To validate the new protocol a reference API was implemented in the Java programming language and tests with an client/server application were accomplished. Besides, it was also accomplished a comparative analysis of this new network security protocol with others.

Índice

Capítulo 1 - Introdução **1**

1.1 Motivação ... 1

1.2 Objetivos do Trabalho ... 4

1.3 Organização do Texto ... 5

Capítulo 2 - Protocolos de Segurança de Rede **6**

2.1 Transport Layer Security - TLS ... 6

2.2 Wireless Transport Layer Security – WTLS ... 11

2.3 Internet Protocol Security – IPSec ... 16

2.4 Secure UDP ... 25

Capítulo 3 - TLS sobre UDP **29**

3.1 Especificação do Novo Protocolo ... 30

3.1.1 Visão Geral ... 30

3.1.2 Protocolo TLS/UDP *Record* ... 31

3.1.3 Protocolo *Change Cipher Spec* ... 44

3.1.4 Protocolo Alert ... 45

3.1.5 Protocolo *Handshake* ... 51

3.1.6 Computação Criptográfica ... 74

3.1.7 Definição das Combinações Criptográficas ... 79

3.1.8 Notas sobre a Implementação ... 82

3.1.9 Análise de Segurança ... 84

3.2 Implementação da API Referência ... 91

3.2.1 TLS/TCP no Java 2 SDK ... 91

3.2.2 Interface de Programação para o TLS/UDP ... 100

Capítulo 4 - Estudo Comparativo **110**

4.1 Ambiente Operacional ... 110

4.2 Aplicação Cliente/Servidor ... 111

4.3 Casos de Estudo ... 112

4.3.1 Aplicação Sem Criptografia ... 113

4.3.2 Aplicação Com IPSec ... 113

4.3.3 Aplicação Com TLS 114
4.4 Resultados: Análise e Comparação 115
4.5 Tunelamento 117
4.5.1 Port Forwarding 117
4.6 Comparação e Análise Crítica 119

Capítulo 5 - Conclusão **123**
5.1 Considerações finais 123
5.2 Propostas de Trabalhos Futuros 124

Referências Bibliográficas **127**

Apêndice A – Glossário **129**

Apêndice B - Modelo de Referência TCP/IP **131**
B.1 - Introdução 131
B.2- Protocolo TCP 132
B.3 - Protocolo UDP 144
B.4 - Protocolo SCTP 148
B.5 - Protocolo IP 151

Apêndice C – Blocos Criptográficos **155**
C.1- Função Hash 155
C.1.1 - SHA e SHA-1 157
C.1.2 - MD2, MD4 e MD5 157
C.1.3 - Ataques as Funções de *Hash* 158
C.2 - Algoritmos Simétricos 160
C.2.1 - Algoritmos para Cifragem de Fluxo 161
C.2.2 - Algoritmos para Cifragem de Bloco 165
C.2.3 - Message Authentication Code - MAC 177
C.3 - Algoritmos Assimétricos 178
C.3.1 - Rivest-Shamir-Adleman - RSA 180
C.4 - Assinaturas Digitais 181
C.4.1 - Rivest-Shamir-Adleman - RSA 183
C.4.2 - Digital Signature Algorithm - DSA 184
C.5 - Algoritmos para Troca de Chave 186
C.5.1 - Diffie-Hellman 186

Apêndice D - Syslog Sign **189**

Apêndice E – SNMP versão 3 **189**

Apêndice F – DNSSEC **190**

Apêndice G – NTP versão 2 **191**

Apêndice H – NFS versão 4 **192**

Apêndice I – RIP versão 2 **193**

Apêndice J – PGPFone **194**

Apêndice K - Connectionless LDAP **195**

Índice de Figuras

Figura 1 - Alternativas de segurança para o TCP 2
Figura 2 - Alternativas de segurança para o UDP..3
Figura 3 - Pilha dos protocolos TLS 7
Figura 4 – Handshake TLS 9
Figura 5 – Gateway Wap 12
Figura 6 - Modo Túnel x Modo Transporte (IPSec) 17
Figura 7 - Datagrama IP protegido com AH..17
Figura 8 - Cabeçalho AH 18
Figura 9 - Datagrama IP protegido com ESP...18
Figura 10 - Cabeçalho ESP 19
Figura 11 - Janela deslizante...23
Figura 12 - Secure UDP 25
Figura 13 - Formato do pacote Secure UDP 27
Figura 14 - Camadas TLS/UDP..30
Figura 15 - Fluxo de mensagens para o handshake completo 53
Figura 16 - Fluxo de mensagens para o handshake abreviado....................................54
Figura 17 - Relacionamento das principais classes JSSE 95
Figura 18 - Instruções de uso da API TLS/UDP..105
Figura 19 - Ambiente operacional de teste 110
Figura 20 - Tipo de Segurança..112
Figura 21 - Modelo TCP/IP 131
Figura 22 - Reconhecimento positivo e retransmissão 135
Figura 23 - Janela Deslizante..136
Figura 24 - Largura de faixa em TCP 138
Figura 25 - Cabeçalho TCP...139
Figura 26- Pseudo-cabeçalho TCP...141
Figura 27 - Cabeçalho UDP..145
Figura 28 - Pseudo-cabeçalho UDP 145
Figura 29 - Cabeçalho IP 151
Figura 30 - Estrutura iterativa de Damgard/Merkle para funções de hash 156

Figura 31 - Modo *Electronic Code Book* 167
Figura 32 - Modo *Cipher Block Chaining* 168
Figura 33 - Modo *Cipher Feedback* 169
Figura 34 - Modo *Output Feedback* 170

Índice de Tabelas

Tabela 1 - Algoritmo para troca de chaves X Certificado 63
Tabela 2 - Conjuntos Criptográficos 80
Tabela 3 - Algoritmo para Troca de Chaves e Assinatura 81
Tabela 4 - Características dos Algoritmos Simétricos 81
Tabela 5 - Resultado da Medida: IPSec x TLS 116
Tabela 6 - Flags TCP 140

Lista de Abreviaturas e Símbolos

AH	*Authentication Header*
API	*Application Programming Interface*
BIND	*Berkeley Internet Name Domain*
CA	*Certification Authority*
CBC	*Cipher Block Chaining*
CFB	*Cipher-FeedBack*
DES	*Data Encryption Standard*
DH	*Diffie-Hellman*
DHE	*Diffie-Hellman Ephemerous*
DNS	*Domain Name System*
DSA	*Digital Signature Algorithm*
DSS	*Digital Signature Standard*
EC	*Elliptic Curve*
ECB	*Eletronic CodeBook*
ECC	*Elliptic Curve Cryptography*
ECDH	*Elliptic Curve Diffie-Hellman*
ECDSA	*Elliptic Curve Digital Signature Algorithm*
ESP	*Encapsulation Security Payload*
FBI	*Federal Bureau Investigation*
FTP	*File Transfer Protocol*
HMAC	*Hash Message Authentication Code*
HTTP	*HyperText Transfer Protocol*
IEEE	*Institute of Electrical and Electronics Engineers*
IDEA	*International Data Encryption Algorithm*
IETF	*Internet Engineering Task Force*
IKE	*Internet Key Exchange*
IP	*Internet Protocol*
IPSec	*Internet Protocol Security*
ISAKMP	*Internet Security Association and Key Management Protocol*
IV	*Initialisation Vector*

JDK	*Java Development Kit*
J2SE	*Java 2 Platform, Standard Edition*
JLS	*Java Language Specification*
JSSE	*Java Secure Socket Extension*
JVM	*Java Virtual Machine*
LPOO	Linguagem de Programação Orientada a Objeto
MAC	*Message Authentication Code*
MD5	*Message-Digest Algorithm 5*
MSS	*Maximum Segment Size*
MTU	*Maximum Transfer Unit*
NAT	*Network Address Translation*
NFS	*Network File System*
OFB	*Output-FeedBack*
OO	*Object-oriented*
OOP	*Object-oriented Programming*
OSI	*Open System Interconnection*
PDU	*Protocol Data Unit*
PFS	*Perfect Forward Secrecy*
PGP	*Pretty Good Privacy*
PMTU	*Path Maximum Transfer Unit*
POP	*Post Office Protocol*
PRF	*Pseudo-Random Function*
PRNG	*Pseudo Random Number Generator*
RC2	*Rivest Cipher #2*
RC4	*Rivest Cipher #4*
RC5	*Rivest Cipher #5*
RC6	*Rivest Cipher #6*
RFC	*Request For Comments*
RSA	*Rivest-Shamir-Adleman*
RTP	*Real Time Protocol*
RTSP	*Real Time Streaming Protocol*
RTT	*Round Trip Time*
SA	*Security Association*

SADB	*Security Association DataBase*
SANS	*SysAdmin, Audit, Network, Security*
SDU	*Service Data Unit*
SHA	*Secure Hash Algorithm*
S-HTTP	*Secure HyperText Transfer Protocol*
SMS	*Short Message Service*
SMTP	*Simple Mail Transfer Protocol*
SNMP	*Simple Network Management Protocol*
SPI	*Security Parameter Index*
SSH	*Secure Shell*
SSL	*Secure Socket Layer*
TCP	*Transmission Control Protocol*
TFTP	*Trivial File Transfer Protocol*
TLS	*Transport Layer Security*
UDP	*User Datagram Protocol*
URL	*Uniform Resource Locator*
VoIP	*Voice over IP*
XML	*Extensible Markup Language*
WAP	*Wireless Application Protocol*
WDP	*Wireless Datagram Protocol*
WSP	*Wireless Session Protocol*
WTLS	*Wireless Transport Layer Security*
WTP	*Wireless Transaction Protocol*

Capítulo 1 - Introdução

1.1 Motivação

De acordo com estatísticas realizadas em diversas Instituições de pesquisa especializadas em Segurança da Informação [1,2], um número cada vez mais crescente de ataques de *hackers* são realizados na Internet e com requinte cada vez maior. Os ataques estão tão comuns e danosos que são notícias da mídia, como em jornais de veiculação diária e revistas semanais. Em fevereiro de 1999, o FBI e a Câmara de Comércio Americana anunciaram que as companhias americanas perdem aproximadamente US$ 2 bilhões por mês causada pela espionagem corporativa. Sendo que 95% destas perdas não foram detectadas ou foram omitidas pelas companhias [3].

O tema segurança de rede está em evidência e qualquer projeto de uma rede de comunicação ou de inclusão de um novo aplicativo em uma infra-estrutura de rede de comunicação IP exige que sejam realizadas as devidas análises de requisitos de segurança para estabelecer um nível adequado para o qual a comunicação de dados na rede será efetuada.

Quando se pensa em troca de informações em um ambiente de rede de computadores distribuído, logo surge como uma das primeiras perguntas: "Qual o nível de segurança exigido para o intercâmbio de informações?". E como resposta a esta indagação podem advir alguns requisitos específicos, como por exemplo:

- Exigência de autenticação do usuário
- Não repúdio das mensagens pelo cliente
- Garantia da integridade da informação
- Privacidade via encriptação dos dados

Devido a natureza insegura da Internet, baseada no protocolo IP (Apêndice B) diversos protocolos de segurança foram e estão sendo desenvolvidos para inserir a devida segurança na comunicação de dados na grande rede. Cada protocolo opera em uma diferente camada e fornece segurança para a sua camada e a todas acima.

A pilha de protocolos TCP/IP (Apêndice B) na sua versão mais simples (figura 1A) não fornece nenhuma segurança. Correntemente nas aplicações TCP/IP, os dados são protegidos (observar as camadas sombreadas) na camada de rede (figura 1B), transporte (figura 1C) ou de aplicação (figura 1D).

O mecanismo cada vez mais utilizado em aplicações cliente/servidor que utilizam o protocolo de transporte orientado a conexão TCP para adicionar os requisitos de segurança mencionados anteriormente é o *Transport Layer Security* (TLS) (figura 1C). Este protocolo opera na camada de transporte. Todas as aplicações que utilizam o modo de transporte TCP podem adicionar a camada extra do TLS de modo quase transparente [4]. Este protocolo é a versão IETF [5] aperfeiçoada do protocolo SSL [6] desenvolvido pela empresa Netscape Communications, muito difundido principalmente pelo uso em aplicações de *home banking* e comércio eletrônico via web (*web commerce*). Este protocolo será detalhado posteriormente no capítulo 2.

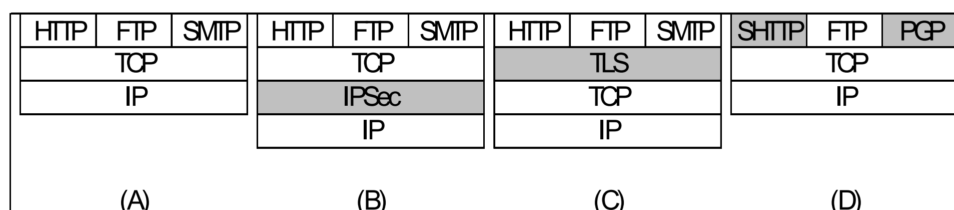


Figura 1 - Alternativas de segurança para o TCP

Correntemente nas aplicações UDP/IP, os dados são protegidos somente na camada de rede (figura 2B) e na camada de aplicação (figura 2D). Não existia ainda um protocolo de criptografia de rede similar ao TLS para ser usado com o protocolo de transporte não orientado a conexão como o UDP (figura 2C). As aplicações que utilizam UDP não tinham um mecanismo padrão, seguro, aberto e genérico para troca de mensagens com a mesma segurança fornecida pelo TLS ao TCP. O único protocolo aberto, seguro e padronizado pelo IETF disponível para o UDP é o IPSec (posteriormente detalhado no capítulo 2) que opera na camada de rede e que exige uma configuração manual específica antes da sua utilização. Além disso, não é equivalente ao TLS/TCP na sua funcionalidade.

Na maioria dos casos, as aplicações que utilizam o protocolo UDP criam e incorporam seu próprio mecanismo de segurança (figura 2D). Isto, por si mesmo, pode gerar potencialmente um problema de segurança. Quando é homologado um padrão pelo IETF ou qualquer outro órgão regulador, este já foi amplamente discutido e avaliado pela comunidade acadêmica e científica de forma que a chance de existir uma falha de segurança é muito menor que em um mecanismo proprietário criado por uma empresa para suprir suas necessidades de segurança. Aliás, esta é a grande vantagem dos protocolos abertos e dos aplicativos de domínio público [7] *(open source)* na Internet.

Até mesmo as versões mais recentes dos protocolos de rede desenvolvidos pelo IETF que fazem uso do UDP, como o SNMP, Syslog e DNS, estão sendo obrigados a desenvolverem os seus próprios mecanismos de segurança. Este problema é causado principalmente pela ausência de um protocolo padrão que adicione segurança ao UDP de maneira prática e genérica.

O protocolo de transporte UDP é largamente utilizado também em aplicações em tempo real, como as multimídias: VoIP, Video sob Demanda, Videoconferência, Telemedicina, entre outros motivos, pelo fato de gerar pouco *overhead* no tráfego dos dados, tendo assim uma alta eficiência.

Uma comunicação via VoIP, por exemplo, pelo fato de utilizar o UDP apresenta os mesmos problemas apresentados anteriormente. Desta forma, os grampos "telefônicos" na Internet são facilmente passíveis de ocorrerem caso não seja agregado algum mecanismo de criptografia.

(A)	(B)	(C)	(D)
SNMPv1 RTSP DNS	SNMPv1 RTSP DNS	SNMPv1 RTSP DNS	SNMPv3 RTSP DNSSEC
UDP	UDP	TLS/UDP	UDP
IP	IPSec	UDP	IP
	IP	IP	

Figura 2 - Alternativas de segurança para o UDP

1.2 Objetivos do Trabalho

Como explicado anteriormente, o protocolo de transporte não orientado a conexão e não confiável UDP não dispõe um protocolo de criptografia como o TLS que opere na camada de transporte, tal como o TCP. Todas as aplicações UDP necessitam criar mecanismos proprietários para autenticar os usuários, encriptar os dados que trafegam na rede, assegurar a integridade das mensagens entre outros requisitos de segurança ou fazer uso do IPSec (caso possível).

Assim sendo, o objetivo deste trabalho de Mestrado é a proposta de um **novo** protocolo de segurança similar ao TLS/TCP para o protocolo de transporte não confiável UDP. E assim criar um mecanismo universal para que todas as aplicações que fazem uso do protocolo UDP também possam agregar a devida segurança a sua funcionalidade de maneira praticamente transparente, necessitando apenas realizar um mínimo de alterações no código fonte da aplicação. Este novo protocolo será referenciado daqui em diante como **TLS/UDP**.

Foi definida também uma API de referência para servir como modelo às futuras implementações deste protocolo. Foi utilizada como linguagem de programação padrão para uma implementação de referência do novo protocolo a linguagem Java. Na criação da API do TLS/UDP foi utilizado o código fonte do compilador JDK/Sun, mais especificamente a codificação do TLS/TCP.

Para verificar a implementação Java do TLS/UDP foram realizados *benchmarks* de forma a avaliar o desempenho do protocolo TLS em relação ao IPSec tanto sobre a camada de transporte TCP como a UDP. E assim, validar a viabilidade da utilização deste novo protocolo de segurança de rede.

Foi realizado também uma análise e comparação deste novo protocolo TLS/UDP com outros atualmente disponíveis em termos de funcionalidade, desempenho e segurança.

1.3 Organização do Texto

Os próximos capítulos desta dissertação de tese de Mestrado estão organizados conforme descrito a seguir:

O capítulo dois analisa os protocolos de segurança de rede disponíveis para UDP e TCP que serviram como fonte de inspiração para o projeto do novo protocolo de segurança de rede TLS/UDP.

O capítulo três define a especificação do novo protocolo proposta como solução para o problema apresentado. Faz a definição do modo de funcionamento e suas formas de utilização. E também mostra como foi realizada a implementação da API de referência do novo protocolo na linguagem Java [8].

No capítulo quatro é descrita uma implementação de duas aplicações cliente/servidor com a finalidade de avaliar o protocolo (*benchmark*). Também será discutido e avaliado o comportamento da aplicação sem criptografia, com o novo protocolo TLS/UDP e com IPSec.

No capítulo cinco é realizada a conclusão sobre o trabalho, sendo composta de uma análise crítica do novo protocolo de segurança de rede e comparação com outros vigentes. E além disso, são descritas algumas propostas de continuação do trabalho realizado e as contribuições disponibilizadas à comunidade acadêmica.

Ao final de todo o trabalho estão as referências bibliográficas que foram utilizadas durante o desenvolvimento deste trabalho de tese.

No apêndice A temos um glossário com a definição básica de alguns termos importantes para o perfeito entendimento do texto.

No apêndice B e C respectivamente será realizado um estudo dos protocolos da camada de transporte (SCTP, TCP e UDP) e rede (IP) e também dos blocos de criptografia utilizados praticamente em todos os protocolos de segurança de rede visando assim subsidiar o entendimento dos próximos capítulos da dissertação e por conseguinte do novo protocolo de criptografia de rede apresentado.

Nos apêndices D a K são apresentados algumas aplicações que utilizam a camada de transporte UDP e adicionaram segurança ao seu funcionamento através da incorporação de mecanismos proprietários na camada de aplicação (quase todos).

Capítulo 2 - Protocolos de Segurança de Rede

Neste capítulo são analisados os protocolos de segurança de rede disponíveis atualmente para serem utilizados pelas aplicações que utilizam o protocolo de transporte UDP, como o IPSec, WTLS e Secure UDP, de forma a agregar segurança ao seu funcionamento. Além disso, também serão analisados alguns protocolos de segurança de rede que podem ser utilizados pelas aplicações que utilizam o protocolo de transporte TCP, como o IPSec, TLS e o WTLS, mas que foram aproveitados neste trabalho como suporte para a criação e especificação do novo protocolo de segurança de rede proposto.

Na análise detalhada dos diversos protocolos de segurança de rede estão contempladas as características mais relevantes e as potenciais falhas de segurança e vulnerabilidades encontradas em cada um deles.

2.1 Transport Layer Security - TLS

O protocolo TLS fornece um serviço seguro e confiável fim a fim para as camadas superiores ao TCP. Foi baseado no protocolo *Secure Socket Layer* (SSL) desenvolvido pela empresa Netscape Corporation. O TLS é uma atualização da versão atual do SSL 3.0, sendo referenciado, às vezes, como SSL versão 3.1 e tem compatibilidade *backward* com o SSL versão 3.0.

Este protocolo foi desenvolvido por um grupo de trabalho do IETF e a definição do protocolo está descrita na RFC 2246 e foi atualizada em outra publicada recentemente, a RFC 3546. Outras RFCs complementam a especificação atual, são as RFC 2712, 2817, 2818 e 3268 [5].

O protocolo TLS opera nas camadas de sessão e transporte, entre a camada de aplicação (por exemplo: HTTP) e o protocolo de transporte TCP (figura 3). E fornece as seguintes opções de segurança: dupla autenticação, confidencialidade, garantia de integridade na troca das mensagens e não repúdio.

O protocolo TLS tem duas camadas internas (figura 3). O protocolo *TLS Record* é a camada mais inferior do protocolo. É usado para transmitir todos os dados das camadas superiores (camada de aplicação e os da camada superiores do TLS).

A camada superior do protocolo TLS consiste em múltiplos sub-protocolos que são utilizados no estabelecimento e na administração das conexões seguras entre as partes comunicantes. Estes protocolos são: *TLS Handshake*, *TLS Change Cipher Spec* e o *TLS Alert*.

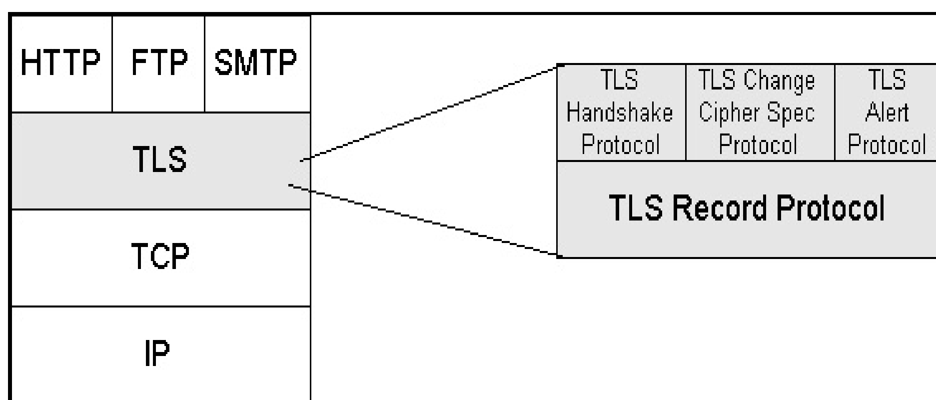


Figura 3 - Pilha dos protocolos TLS

O protocolo TLS pode utilizar, por exemplo, certificados digitais X.509 versão 3 para autenticar a identidade do cliente e/ou servidor e assegurar a confidencialidade dos dados trocados. Cada certificado contém uma chave pública e o dono do certificado mantém uma chave privada associada com a chave pública do certificado. O certificado é assinado por uma entidade de confiança, conhecida como Autoridade Certificadora (AC).

Devido a grande quantidade de processamento matemático necessário para processar as mensagens encriptadas com qualquer algoritmo criptográfico de chave pública, o protocolo TLS utiliza para a encriptação das mensagens um algoritmo de criptografia de chave simétrica e a chave da sessão TLS (*master secret*) utilizada é baseada em valores aleatórios trocados entre as partes nas primeiras mensagens no protocolo *TLS Handshake*. E o algoritmo de criptografia de chave pública é utilizado quando o cliente está enviando um número aleatório (segredo) conhecido como *premaster secret*. Com o uso da chave pública é assegurado que somente o

destinatário conseguirá descobrir o valor deste segredo. E este número aleatório será usado na criação da chave de sessão (*master secret*).

Vários algoritmos simétricos estão previstos para uso no protocolo TLS, por exemplo: DES 40 bits, DES 56 bits, 3DES EDE, RC4 40 bits, RC4 128 bits, RC2 40 bits, IDEA, AES 128 e AES 256. E o modo de operação indicado para os algoritmos de criptografia simétrica de bloco é o CBC. E para cada mensagem com os dados da aplicação é calculado o Hash-MAC, e as funções de hash que podem ser utilizadas para tal são o MD5 e/ou SHA-1.

O protocolo TLS utiliza a chave de sessão gerada por um período de tempo limitado ou um número limitado de mensagens. Uma vez que expira a validade da chave de sessão, as partes comunicantes renegociam uma chave de sessão nova. Além disso, qualquer das partes comunicantes pode solicitar a renegociação a qualquer momento de uma nova chave de sessão. A parte interessada deveria enviar uma mensagem de *hello request* (servidor) ou *client hello* (cliente) para indicar o desejo de renegociar os parâmetros da sessão.

Como algoritmo para troca das chaves pode ser utilizado o RSA ou *Diffie-Hellman*. Caso seja requerida que a troca de chaves não seja anônima, o certificado do servidor (e talvez do cliente) deverá ser transmitido e os algoritmos de assinatura que podem ser utilizados são o RSA e o DSA.

Para cada lado da comunicação é definido um estado da conexão que especifica o algoritmo de compressão, o algoritmo de encriptação e o algoritmo de MAC. Além disso, outros parâmetros como o *MAC Secret*, as chaves de encriptação e os vetores de inicialização (IV) também são especificados em ambos os sentidos.

TLS Handshake

No processo de *handshake* é realizado todo o procedimento necessário para o estabelecimento do acordo de quais algoritmos criptográficos serão utilizados na comunicação. E assim todos os parâmetros necessários para que a comunicação segura possa ser efetuada são definidos, como exemplo: o algoritmo para a troca de chaves, o algoritmo simétrico, a chave simétrica e tamanho, o vetor de inicialização (IV), o algoritmo de assinatura (opcional), *MAC* e o *MAC Secret*.

Uma vez que o procedimento de *handskake* é completado, as duas partes podem transmitir os dados da aplicação através de um canal de comunicação seguro.

A sequência de *handskake* pode variar dependendo do método usado para a troca de chaves (RSA ou *Diffie-Hellman*), da forma de autenticação (simples ou dupla) e do tamanho da chave usada para assinatura. O procedimento de *handshake* requerido para estabelecer uma sessão TLS é mostrado na figura 4 onde estão relacionados as mensagens trocadas entre o cliente e o servidor. O cliente inicia a comunicação transmitindo a mensagem *client hello* para o servidor desejado.

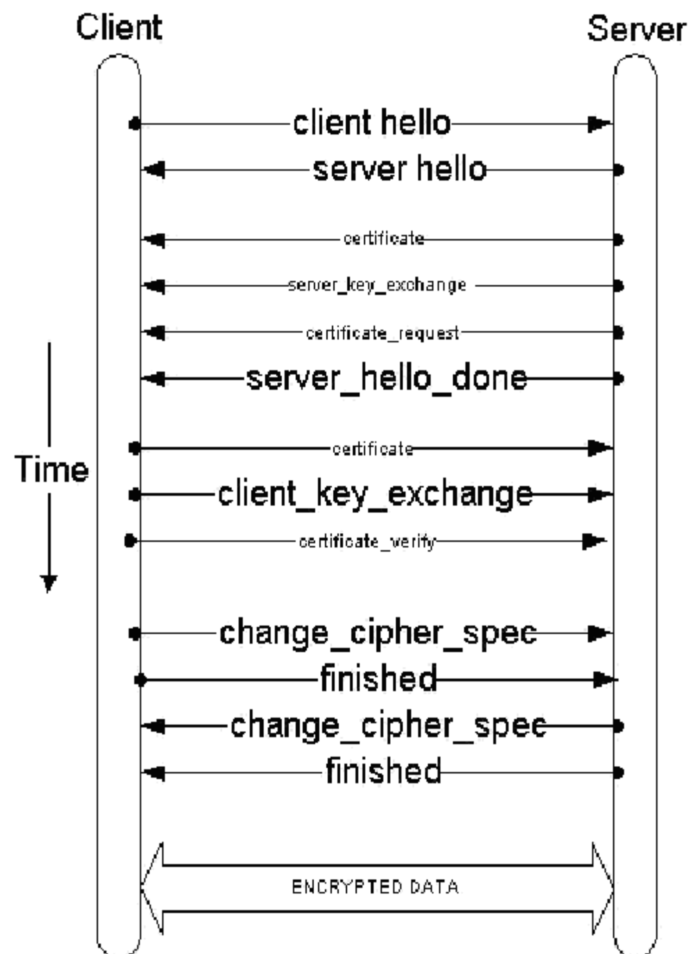


Figura 4 – Handshake TLS

Nesta mensagem está incluído um número aleatório (*ClientHello.random*) que é usado para evitar o ataque por *reprodução*. Em resposta, o servidor envia uma mensagem de *server hello*, que inclui um número aleatório também (*ServerHello.random*) e em seguida envia um certificado que contém a sua chave pública. O cliente verifica o certificado atráves da validação da assinatura da

Autoridade Certificadora. O cliente então gera uma p*remaster key* e encripta com a chave pública do servidor e envia este segredo ao servidor na mensagem *client key exchange*. O servidor por sua vez, decripta a mensagem com a sua chave privada e obtém a *premaster key*. Esta chave é usada para gerar a chave simétrica da sessão (*master secret*) que será usada tanto pelo cliente como pelo servidor para encriptação e autenticação das mensagens.

O processo de geração da chave secreta (*master secret*) é realizado com a ajuda de uma função pseudo-aleatória (PRF, ver anexo):

master secret = PRF (*premaster secret*, “master secret”,
ClientHello.random + *ServerHello.random*)

A sessão criada entre as partes tem uma identificação associada (*session ID*) que é gerada pelo servidor e enviado ao cliente na mensagem de *server hello*. Este ID também pode ser utilizado para indicar o desejo de reutilizar os parâmetros de segurança em uma próxima sessão entre os participantes. E para isso o cliente deve enviar o *session ID* armazenado na próxima mensagem de *client hello*. O servidor pode ignorar o *session ID* e não reutilizar os parâmetros de segurança.

TLS Record

O protocolo *TLS Record* é o responsável pela fragmentação e empacotamento dos dados advindos das camadas superiores. Primeiramente, os dados são fragmentados em *chunks* de tamanho apropriado, comprimidos e então um *Message Authenticating Code* (MAC) é adicionado a mensagem e a mensagem resultante é encriptada. Finalmente o cabeçalho TLS record é adicionado para ajudar o destinatário a identificar a mensagem e também para reverter as operações realizadas pelo remetente da mensagem.

TLS Changer Cipher

O protocolo *Change Cipher* é um protocolo muito simples. Ele é usado na etapa de estabelecimento da sessão TLS para impedir as situações de *deadlock*.

TLS Alert

O protocolo *Alert* é utilizado para enviar alertas entre as partes comunicantes de uma conexão TLS. Os alertas têm dois níveis diferentes: advertência e fatal. Uma vez que um alerta fatal é sinalizado, a conexão é encerrada obrigatoriamente.

2.2 Wireless Transport Layer Security – WTLS

O *Wireless Application Protocol* (WAP) [9] é um resultado de trabalho contínuo de desenvolvimento para definir uma especificação para aplicações que operam sobre redes de comunicação sem fio. O objetivo do Fórum WAP é definir um conjunto de especificações para ser usado nos serviços de aplicações sem fio. O mercado sem fio está crescendo muito rapidamente e alcançando novos clientes e gerando uma gama diferenciada de serviços. Para permitir os operadores e fabricantes enfrentarem os novos desafios destes serviços avançados foi criado o Fórum WAP que define um conjunto de protocolos de transporte, segurança, transação, sessão e a camada de aplicação.

O protocolo da camada de segurança na arquitetura WAP é chamado de *Wireless Transport Layer Security* (WTLS). A camada WTLS opera sobre o protocolo da camada de transporte. A camada WTLS é modular e seu funcionamento depende do nível de segurança exigido pela aplicação. Até mesmo se esta camada será usada ou não, quem define é a aplicação. O WTLS proporciona a camada de nível superior do WAP uma interface de serviço de transporte segura que preserva a interface com o serviço de transporte abaixo dela.

A meta primária da camada WTLS é prover privacidade, integridade de dados e autenticação entre as duas aplicações que comunicam entre si. O WTLS fornece funcionalidade semelhante ao TLS 1.0 e incorpora novas características como suporte a datagrama, *handshaking* otimizado e atualização da chave de forma dinâmica. O protocolo WTLS é aperfeiçoado para redes de pequena largura de faixa com uma latência relativamente longa.

Um modelo de camadas dos protocolos da arquitetura WAP é ilustrado na figura 5. O funcionamento das camadas dos protocolos WAP e as suas funções são semelhantes ao modelo OSI de referência ISO (ISO7498) para as camadas superiores.

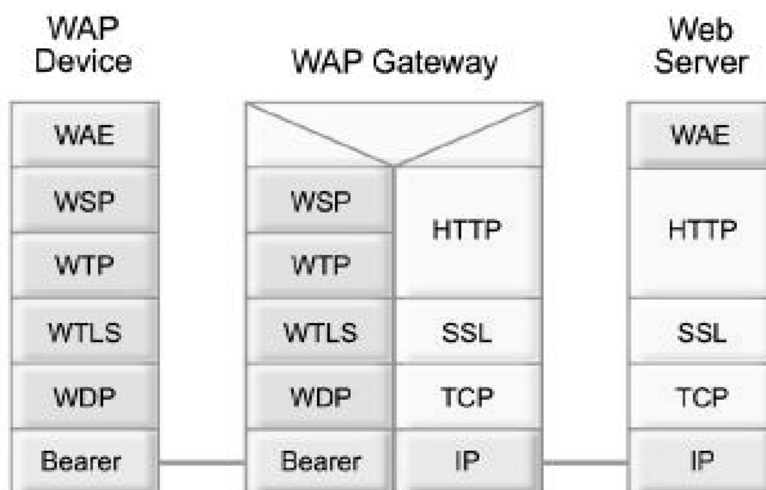


Figura 5 – Gateway Wap

A seguir, serão descritas as principais diferenças do protocolo WTLS (atual versão: 06-Abril-2001) em relação ao *Transport Layer Security* versão 1.0:

- O WTLS é projetado para funcionar com protocolo de transporte confiável ou não (datagrama). Caso seja utilizado sobre um protocolo de transporte não confiável (similar ao UDP) é necessário utilizar o modo de numeração de sequência explícita, onde um campo "número de sequência" é colocado em texto puro no cabeçalho WTLS. E este campo é utilizado na técnica de janela deslizante (IPSec também utiliza) para evitar o ataque por *reprodução*.

- A atualização do material criptográfico (chaves, IVs, *MAC secrets*) é realizada de forma automática no WTLS e o intervalo para cada *refresh* é negociado no processo de *handshake*. O novo material é calculado a cada n mensagens, onde $n = 2^{key_refresh}$. Isto é, o material será atualizado quando o número de sequência for igual a n, 2n, 4n e assim por diante. O parâmetro *key_refresh* é negociado entre as partes.

- O WTLS oferece suporte aos algoritmos de criptografia baseados em curvas elípticas (*Elliptic Curves* - EC). Para a troca de chaves, EC *Diffie-Hellman*, e para assinatura, EC DSA. Porém não disponibilizou na versão atual do protocolo, o uso dos algoritmos de encriptação simétricos de fluxo, como o RC4.

- No protocolo *Alert* do WTLS foi adicionado um campo *checksum* de 4 bytes. É calculado a partir do último *WTLSCipherText* recebido da outra parte. O receptor do alerta tem que verificar se o *checksum* é relativo a uma mensagem enviada anteriormente por ele.

- Novas mensagens de alerta foram adicionadas no WTLS. Por exemplo: (i) alerta *time_required*, um aviso enviado pelo servidor com o objetivo de informar ao cliente para reiniciar o seu *timer* de retransmissão pois requer mais tempo para realizar alguma operação do processo de *handshake*, (ii) e alerta *duplicated_finished_received*, geralmente um aviso indicando que o cliente retransmitiu a mensagem *finished*.

- O WTLS suporta um *handshake* completo otimizado onde o Servidor pode calcular a chave pré-master (e consequentemente a master) caso tenha como obter o certificado do cliente, seja utilizado o método para troca de chaves *EC Diffie-Hellman* e que os parâmetros necessários sejam fornecidos nos certificados das partes. Assim sendo, o cliente envia a mensagem *ClientHello*, o servidor responde com as mensagens *ServerHello*, *Certificate*, *ChangeCipherSpec* e *Finished* e finalmente o cliente responde com as mensagens *ChangeCipherSpec* e *Finished* e a seguir, a aplicação cliente/servidor já pode comunicar de forma segura.

- Como o protocolo WTLS suporta protocolos de transporte não confiáveis, foi adicionado um mecanismo para retransmissão das mensagens de *Handshake* para adicionar confiabilidade no processo de estabelecimento do canal de comunicação seguro. Por exemplo, para o *Handshake* completo, o cliente tem que retransmitir as mensagens

ClientHello e *Finished* se a esperada mensagem de resposta não for recebida do servidor em um tempo de expiração (*time-out*) predefinido. E caso o número máximo de retransmissões exceder o número máximo predefinido, o cliente termina o procedimento de *Handshake*.

- A mensagem *Server Certificate* e *Client Certificate* no protocolo WTLS suporta uma *Certificate URL* (uma URL onde o cliente ou o servidor podem buscar o certificado). Foi adicionado ao protocolo TLS na última atualização em Junho de 2003, RFC 3546 (*Transport Layer Security Extensions*), a possibilidade do cliente enviar a *Certificate URL* mas não o servidor.

- A chave master do WTLS tem 20 bytes de tamanho enquanto a chave master do TLS tem 48 bytes. Além disso, suporta algoritmos HMAC com tamanho da saída de 40 bits (utiliza apenas parte da saída do *hash*, MD5_40 e SHA_40) e suporta também algoritmos simétricos com chave de 40 bits (RC5_CBC_40 e DES_CBC_40).

- No WTLS, para o cálculo do material criptográfico derivado da chave master é utilizado também como parâmetro o número de sequência. O primeiro conjunto é calculado com número de sequência igual a zero e as atualizações de acordo com número de sequência do momento.

  key_block = PRF (SecurityParameters.master_secret,
  expansion label, ***seq_number*** *+*
  SecurityParameters.server_random
  SecurityParameters.client_random)

- No protocolo WTLS, apenas um algoritmo de *hash* é utilizado para a realização da PRF. Esta alteração (e outras já citadas anteriormente) foi realizada com objetivo de economizar recursos do dispositivo WAP, principalmente energia, memória e CPU.

- O vetor de inicialização IV para cada registro é calculado do seguinte modo: IV_S = IV XOR S, onde IV é o vetor de inicialização original (*client write IV* ou *server write IV*) gerado pela PRF e S é obtido através da concatenação do número de sequência (2 bytes) do registro tantas vezes quantas forem necessárias para alcançar o mesmo tamanho do vetor de inicialização. Foi mostrado como este método para geração do vetor de inicialização pode ser usado para inferir dados sobre a comunicação (vulnerabilidade).

2.3 Internet Protocol Security – IPSec

O *Internet Protocol Security* (IPSec) fornece um padrão aberto para adicionar segurança a qualquer comunicação IP em um ambiente não confiável, como a Internet. Insere segurança na camada IP da pilha TCP/IP, sendo assim tanto utilizado pelo protocolo de transporte TCP como UDP. A arquitetura IPSec é composta da integração de algoritmos de encriptação e para troca de chaves, protocolos de segurança de rede e de infra-estrutura de segurança [10].

A arquitetura IPSec fornece os seguintes serviços de segurança: confidencialidade dos dados (encriptação), autenticação das partes (não repúdio) e assegura a integridade dos dados. Fornece também controle de acesso, proteção contra ataques por *reprodução* e tunelamento (*Virtual Private Networks* - VPN).

A arquitetura IPSec é especificada em mais de 25 RFCs (*Request For Comments*). A especificação do núcleo da arquitetura IPSec estão detalhadas nas RFCs 2401 até 2412. Existe um grupo de trabalho do IETF responsável pelo desenvolvimento e aperfeiçoamento desta arquitetura.

A proteção dos datagramas IPs é realizada através dos seguintes protocolos: *Authentication Header* (AH) e *Encapsulation Security Payload* (ESP). AH é usado para garantir a integridade dos dados, a autenticação do *host* e oferece opcional segurança ao ataque por *reprodução*. ESP fornece as mesmas opções do AH e mais a opção de encriptar os dados (confidencialidade).

Modos de transporte

O IPsec, independentemente do fato de utilizar AH ou ESP, suporta dois modos de transporte diferentes: transporte e túnel.

O modo transporte é usado normalmente para proteger quando as partes finais são também as partes a serem criptografadas. No modo túnel, a parte que realiza a criptografia normalmente é um *gateway* que fornece segurança a favor de alguma rede (normalmente utilizado para realizar *router-to-router VPN*). No modo transporte, o cabeçalho IPSec é inserido entre o cabeçalho IP do datagrama e os demais dados das camadas superiores. No modo túnel, o datagrama IP inteiro é encapsulado em um

outro datagrama IP e o cabeçalho IPSec é inserido entre os dois cabeçalhos IPs (ver figura 6).

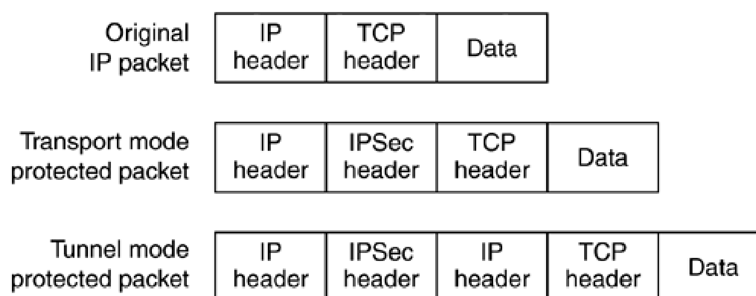


Figura 6 - Modo Túnel x Modo Transporte (IPSec)

Authentication Header

A especificação do AH foi descrita na RFC 2402 e define que a integridade dos dados transmitidos (inclusive o cabeçalho) é realizada através do *keyed hashing* (HMAC). O AH calcula o *hash* do cabeçalho IP e dos dados, mas não inclui os campos que provavelmente se modificam como *hop count* (figura 7).

Figura 7 - Datagrama IP protegido com AH

O AH pode ser usado para proteger o protocolo da camada superior (modo de transporte) ou o datagrama IP inteiro (modo túnel) da mesma forma que o ESP. Em ambos os casos, o cabeçalho AH é inserido imeadiatamente após o cabeçalho IP. O datagrama protegido AH é apenas um outro datagrama IP e portanto o AH pode ser usado sozinho ou em conjunto com o ESP.

A figura 8 mostra o cabeçalho AH (os campos SPI, *Sequence Number* e *Next header* serão descritos posteriormente no decorrer desta seção).

O campo *Payload Length* tem valor igual ao tamanho dos dados úteis do cabeçalho em palavras de 32 bits menos 2. AH é um cabeçalho de extensão IPv6 e sendo assim, este campo recebe o valor acima conforme especificado na RFC 1883 (observando que AH é definido em palavras de 32 bits e a RFC define este campo em palavras de 64).

O campo *Authentication data* é um campo de tamanho variável que contém o resultado da função de verificação da integridade. AH não define um mecanismo único, mas obriga a implementação dos seguintes autenticadores: HMAC-SHA-96 e HMAC-MD5-96. Estas funções *keyed* MAC são truncadas em 96 bits.

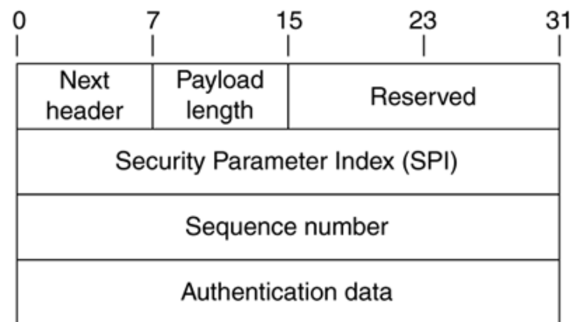


Figura 8 - Cabeçalho AH

Encapsulation Security Payload

A especificação do ESP foi descrita na RFC 2406 e define os algoritmos de encriptação suportados pelo ESP, exemplo: 3DES-CBC. O ESP também fornece verificação da integridade através dos mesmos mecanismos utilizados pelo AH (HMAC MD5 e HMAC SHA). Para realizar estas tarefas, ele insere um novo cabeçalho (*ESP Header*) depois do cabeçalho IP e antes dos dados a serem protegidos (um protocolo da camada superior ou um datagrama IP inteiro) e insere também um *ESP trailer*. A localização do cabeçalho e do *trailer ESP* é mostrado na figura 9.

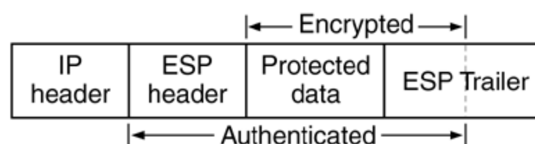


Figura 9 - Datagrama IP protegido com ESP

Todos os algoritmos simétricos de bloco usados com ESP têm que operar em modo CBC. O modo CBC necessita que a quantia de dados a ser encriptada seja um múltiplo do tamanho do bloco de dados do algoritmo de encriptação. Para esta exigência ser satisfeita é realizado um *padding* ao final dos dados quando for

necessário encriptar. O *pad* torna-se parte do texto cifrado do pacote e será retirado pelo receptor depois do processamento do IPSec.

O modo CBC também requer um vetor de inicialização (IV) para começar o processo de encriptação. Este vetor é inserido nos primeiros bytes do *payload*. Cada algoritmo deve especificar o tamanho e o método para obter o vetor. Por exemplo, quando for utilizado 3DES-CBC, o vetor IV será composto dos primeiros 8 bytes do campo protegido de dados. Em cada datagrama IPSec enviado é necessário enviar este vetor pois caso houvesse perda (ou desordenamento) dos datagramas IP na rede não seria possível decriptar os dados. A figura 10 mostra o cabeçalho ESP (o campo SPI e *Sequence Number* serão descritos no decorrer desta seção).

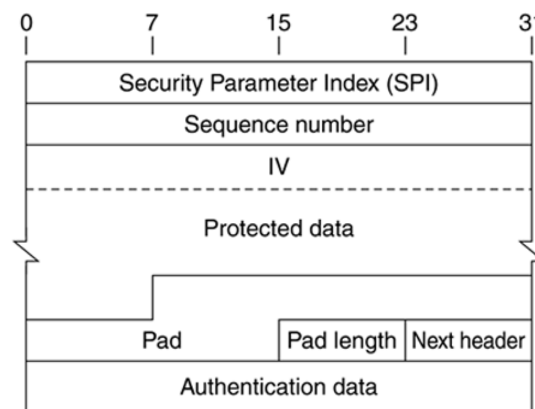


Figura 10 - Cabeçalho ESP

Security Association

A *Security Association*, mais conhecida como *SA* na terminologia IPSec, representa o contrato de segurança estabelecido entre as partes comunicantes. Ela determina os protocolos IPSec usados para fornecer segurança a comunicação. Indica os algoritmos de encriptação e autenticação, as chaves, o tempo de vida das chaves e outros atributos.

As associações SAs são unidirecionais, isto é, *simplex*. Se dois *hosts* A e B estão comunicando entre si usando ESP, então o *host* A terá um *SAout* para o processamento dos pacotes de saída e terá um *SAin* para o processamento dos pacotes de entrada. O *host* B também criará os dois SAs para o processamento dos pacotes. O *SAout* do *host* A e o *SAin* do *host* B terão os mesmos atributos criptográficos (por exemplo: chaves) e vice-versa.

As SAs são específicas para cada protocolo (AH e ESP). Há uma associação para cada protocolo. Cada *host* mantém uma tabela para cada protocolo e para cada sentido da comunicação. Toda implementação IPSec sempre constrói uma banco de dados de *SAs* (*SADB*) que mantém as SAs atualizadas. As SAs são identificadas por um SPI, *Security Index Parameter*, que está incluído tanto no cabeçalho AH como no ESP (campo de 32 bits). O receptor utiliza o SPI para indexar o banco de dados (*SADB*) para saber qual SA utilizar no processamento do pacote de entrada (usado para identificar unicamente o *SA* no receptor). O *host* receptor utiliza a tupla <SPI, IP Destino, Protocolo> para identificar unicamente a *SA*. O endereço IP destino é necessário caso o *host* tenha mais de uma interface de rede.

Internet Key Exchange

O IETF definiu uma infra-estrutura genérica para o gerenciamento das *SAs* e das chaves conhecida como *Internet Security Association and Key Management Protocol*, ISAKMP, e foi especificada na RFC 2408. A infra-estrutura ISAKMP define o formato dos pacotes, tempos de retransmissão e requerimentos para construção das mensagens. E assim, o ISAKMP isola o gerenciamento das SAs e das chaves, do protocolo utilizado para efetivamente trocar as chaves. Existem diferentes protocolos para a troca de chaves, como o protocolo IKE apresentado a seguir.

O *Internet Key Exchange*, IKE, é um dos protocolos que podem ser utilizados na infra-estrutura ISAKMP. O protocolo IKE foi especificado na RFC 2409 e realiza a administração das *SAs* (criação, remoção, dentre outras) e faz o gerenciamento das chaves usadas na comunicação. Este protocolo pode ser utilizado para negociar chaves para qualquer outro protocolo que precise de chaves, não somente o IPSec. O protocolo IKE é um híbrido dos protocolos Oakley e SKEME e sempre utiliza o algoritmo *Diffie-Hellman* para as partes combinarem a chave simétrica de forma segura e dinâmica. O IPSec também exige que o gerenciamento manual das chaves seja suportado e este mecanismo foi usado extensivamente durante a fase inicial de desenvolvimento e testes da arquitetura.

O protocolo IKE é normalmente executado em *userlevel* (camada de aplicação) é responsável por preencher dinamicamente a tabela *SADB* (*kernel*). A comunicação do IKE com o processo que cuida desta tabela depende inteiramente da capacidade de comunicação entre processos (*Inter-Process Communications* - IPC, como a fila de mensagens ou *sockets*) da plataforma. O IKE também precisa disponibilizar uma interface com o *host* IKE remoto para estabelecer a SA e tem definido para uso, a porta lógica 500 e o protocolo de transporte UDP.

O protocolo IKE usa o conceito de associação de segurança - SA - mas a forma de criação de um IKE SA é diferente que de IPSec SA. O IKE SA define o modo no qual os dois *hosts* comunicam-se. Especifica, por exemplo, o algoritmo utilizado para encriptação do tráfego IKE, o método para autenticar o *host* remoto, entre outras definições. O IKE SA pode ser utilizado para produzir um número qualquer de IPSec SAs entre as partes (um para cada aplicação, por exemplo).

As IPSec SAs estabelecidas pelo IKE podem opcionalmente ter a propriedade *Perfect Forward Secrecy* (PFS) das chaves, e se desejado, a identificação do host remoto. Após a troca Diffie-Hellman, os dois hosts compartilham um segredo mas este não é autenticado. Eles podem usar este segredo para proteger a comunicação, mas eles não têm nenhuma garantia que o *host* remoto é, de fato, alguém no qual eles confiam. O próximo passo na troca IKE é a autenticação do segredo compartilhado Diffie-Hellman, e portanto a autenticação do próprio IKE SA. Há cinco métodos de autenticação definidos no IKE: chaves pré-compartilhadas, assinatura digital DSS, assinatura digital RSA, uma troca de valor aleatório encriptado usando RSA e

finalmente um método revisado de autenticação com valores aleatórios encriptados que é sutilmente diferente do outro método anterior.

A criação do IKE SA é referenciada nas RFCs como Fase 1. Uma vez que a Fase 1 é completada, a Fase 2 para criação das IPSec SAs pode começar. Há dois modos de trocas que podem ser realizados na Fase 1, o modo de troca principal (*Main mode*) ou modo de troca agressivo (*Aggresive mode*). O modo agressivo é mais rápido mas modo principal é mais flexível e seguro. Há um único tipo de Fase 2 conhecido como modo rápido (*Quick mode*). A troca realizada na Fase 2 negocia os IPSec SAs sob a proteção do IKE SA que foi criado na Fase 1.

As chaves usadas para o IPSec SAs são, por *default*, derivadas do estado secreto IKE. São trocados valores aleatórios (*nonces*) em *Quick mode* e *hashed* com este estado secreto para gerar as chaves e garantir que cada SAs tenha uma chave única. Todas estas chaves não têm a propriedade PFS pois elas são derivadas da mesma chave raiz, o segredo IKE. Para possibilitar o PFS, novos valores públicos Diffie-Hellman, e o grupo do qual eles são derivados, são trocados junto com os *nonces* e os parâmetros de negociação IPSec SA . O segredo resultante é usado para gerar as chaves IPSec SA, garantindo assim o PFS.

As duas primeiras mensagens na troca da Fase 1 (modo principal ou agressivo) também trocam *cookies*. Estes valores assemelham-se a números aleatórios mas são valores temporais e relacionado com o endereço IP do host. A criação do *cookie* é feita através de um *hash* dos seguintes parâmetros: o segredo, a identidade do *host* e contador baseado na hora atual. Para um observador casual, o resultado deste *hash* será um número aleatório mas o receptor do *cookie* pode determinar rapidamente se foi gerado o *cookie* ou não pela análise do *hash*. Isto amarra o *cookie* ao *host* e fornece uma limitada proteção ao ataque de negação de serviço pois a troca Diffie-Hellman não é executada até uma completa ida e volta (*round trip*) e uma troca de *cookies* sejam realizadas.

Escrever uma rotina em que seriam construídas falsas mensagens IKE e enviá-las a um destino com um endereço fonte forjado poderia ser trivial. Para evitar tal possibilidade, no modo principal, o receptor (*responder*) não faz qualquer cálculo Diffie-Hellman até que ele receba uma segunda mensagem do transmissor (*initiator*) e verifique que a mensagem contém o *cookie* que ele gerou para o transmissor. O modo agressivo não tem tal proteção contra o ataque de negação de serviço, as partes

completam a troca em três mensagens (ao invés de seis, no modo principal) e passam mais informação em cada mensagem. Após a recepção da primeira mensagem no modo agressivo, o *responder* tem que fazer uma exponenciação Diffie-Hellman, isto antes de ter a chance de verificar o *cookie* na próxima mensagem que ele receberá.

Janela Deslizante

O protocolo IP não é orientado a conexão e sem confiabilidade. A entrega dos datagramas IP não é garantida e nem a ordem de chegada não está assegurada também. O protocolo IPSec trata o problema da entrega fora-de-ordem e dos pacotes perdidos utilizando uma técnica de janela deslizante (figura 11).

Para proteger o protocolo IPSec de ataques por *reprodução*, cada pacote AH ou ESP enviado tem um *Security Parameter Index* (SPI) e um número de sequência. O SPI é um identificador único entre as partes comunicantes e é determinado antes de qualquer pacote ser transmitido entre eles. O SPI indexa uma tabela onde os algoritmos de criptografia e parâmetros usados entre as partes comunicantes estão definidos. O número de sequência inicia em 1 e continua sendo incrementado até 2^{32}-1 cada vez que o SA é utilizado para transmitir um pacote. Quando o número de sequência alcança seu valor máximo, o SPI tem que ser renegociado e o número de sequência é reinicializado em 1. Esta técnica assegura que nunca haverá mais de um pacote com o mesmo SPI e número de sequência que são válidos.

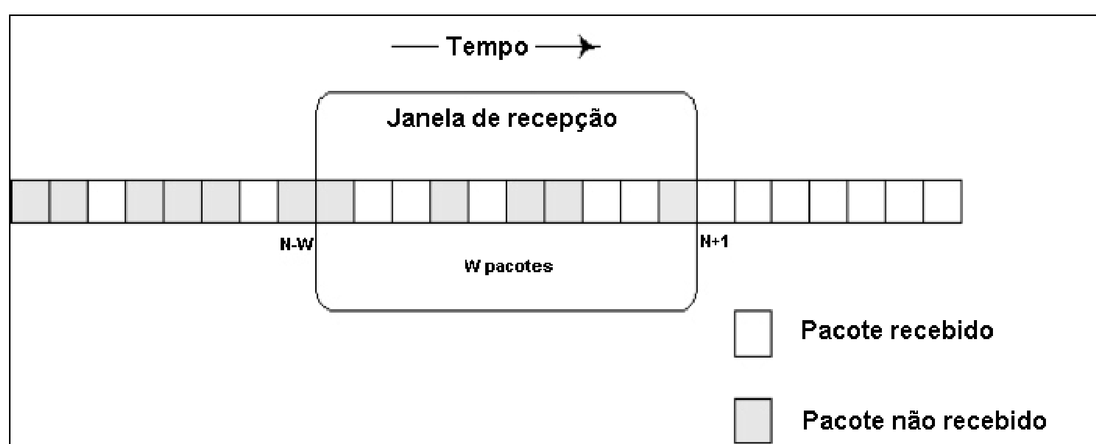


Figura 11 - Janela deslizante

O remetente de um pacote simplesmente incrementa o número de sequência e envia o pacote, a menos que o número de sequência tenha alcançado o valor máximo.

Por outro lado, o receptor mantém uma janela de recepção de pacotes válidos o qual está disposto a aceitar. A janela de recepção é de tamanho fixo W. A janela é definida baseada no pacote recebido com número de sequência mais elevado (limite superior) e pelo tamanho W. Uma vez que um pacote de número de sequência maior é recebido a janela é avançada (figura 11).

Se um pacote é recebido e tem um número de sequência menor que o menor número de sequência esperado na janela de recepção, ele é descartado. Se o pacote está dentro da janela então o pacote é processado. Se o pacote tem um número de sequência maior que o pacote com maior número de sequência na janela de recepção, o pacote é processado e a janela é avançada.

Fragmentação

O IPSec não fragmenta nem remonta os datagramas. No processamento de saída, a carga útil para transporte (*transport payload*) é processada e então passada para a camada IP. No processamento de entrada, a camada IPSec obtém o pacote remontado da camada IP.

No processamento de saída, o IPSec adiciona o cabeçalho IPSec e este impacta no tamanho do PMTU (*Path Maximum Transmission Unit*). Se o IPSec não participar da descoberta do PMTU, a camada IP poderá ser forçada a fragmentar pacotes pois o cabeçalho IPSec pode aumentar o tamanho do datagrama IP para um valor maior que o PMTU.

Tempo de vida de uma associação

Existe um tempo de vida máximo associado a cada SA. Este tempo pode ser especificado em termos do número de bytes que foram processados com esta SA ou pelo intervalo de tempo em segundos que esta SA foi utilizada (*default* 8 horas) ou ambas as formas.

2.4 Secure UDP

O protocolo Secure UDP [3] fornece serviço de confidencialidade e autenticação em uma comunicação de dados em tempo real em uma rede não segura. O projeto do protocolo não altera nenhuma das características do protocolo UDP. Apenas modificações secundárias do código fonte das aplicações UDP atuais são requeridas para torná-las seguras através do uso deste protocolo.

O protocolo *Secure UDP* funciona utilizando dois diferentes canais de comunicação, o *Key channel* e o *Data channel* (figura 12).

- O *key channel* é utilizado para a negociação dos protocolos de criptografia, troca de chaves e a renegociação das chaves (*re-keying*). O *key channel* utiliza o protocolo TLS (e por conseguinte, protocolo de transporte TCP).

- O *data channel* é utilizado para, de forma segura, transmitir e receber os datagramas UDP. O *data channel* utiliza o protocolo de transporte UDP.

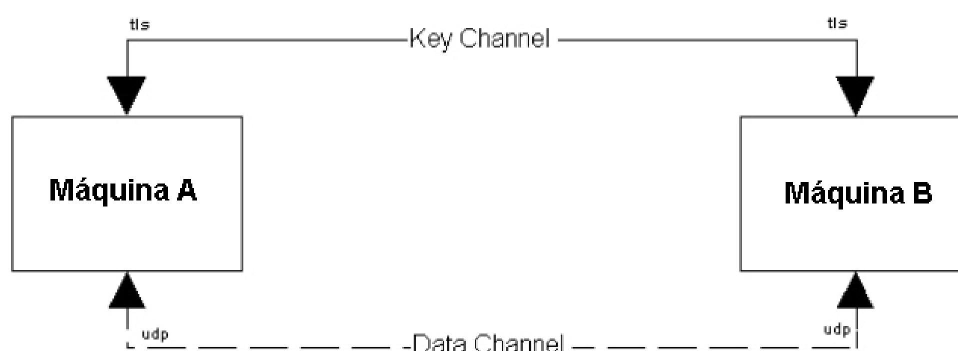


Figura 12 - Secure UDP

Processo de *Handshake*

Devido ao custo da criptografia de chave pública, o Secure UDP utiliza também o conceito de chave de sessão obtida através de uma operação de chave pública. Quando duas máquinas precisam comunicar-se de forma segura, uma das máquinas abre uma conexão TLS com a outra máquina. Uma vez que a conexão TLS

foi estabelecida, um SPI (*Security Parameter Index*) é negociado entre ambas máquinas. O SPI é um identificador único para ambas as máquinas. O SPI define os protocolos de criptografia usados por ambas as máquinas. Uma vez que o SPI é definido, ambas as máquinas reinicializam o número de sequência delas para 1 e começam a transmissão dos dados no *channel data*.

Os dados são transmitidos usando o mesmo SPI, até que o número de sequência alcance seu valor máximo. Quando o número de sequência alcança seu valor máximo, o remetente envia uma mensagem (*server_hello* ou *client_hello*) de renegociação (*re-keying*) no *key channel*. Uma vez recebida, é gerado um novo SPI e uma nova chave de sessão. Para prevenir que o procedimento de *re-keying* atrase o fluxo dos dados, as partes comunicantes deveriam iniciar a operação de *re-keying* antes que o número de sequência alcancem seu valor máximo.

Os dados são recebidos utilizando a mesma técnica de janela deslizante do IPSec. A técnica de janela deslizante permite ao lado receptor decriptar as mensagens assim que elas chegam e ainda impede os ataques por *reprodução*. Uma vez que a mensagem é recebida, seu MAC é conferido e se o mesmo estiver correto, a mensagem é decifrada e é entregue a camada superior.

Cada pacote é tratado separadamente para permitir a decriptação do pacote independentemente dos outros pacotes. Ataques por análise de tráfego dos dados são possíveis. Para dificultar tal ataque a cada mensagem são acrescentados dados aleatórios. Estes dados aleatórios impedem o atacante de detectar mensagens de conteúdo semelhante.

Embora o protocolo *Secure UDP* tente evitar o ataque por análise de tráfego, não pode impedir o atacante descobrir uma atividade de rede em grande escala que é normalmente associada à transmissão de dados multimídia. Em outras palavras, o *Secure UDP* pode esconder o fato de que dois quadros de um filme são os mesmos mas não pode esconder o fato que o usuário está assistindo um filme ou não.

Formato do Pacote Secure UDP

Para habilitar a comunicação segura através deste protocolo e assegurar a confidencialidade e a autenticidade do pacote recebido, o *Secure UDP* utiliza um formato de pacote especial (figura 13).

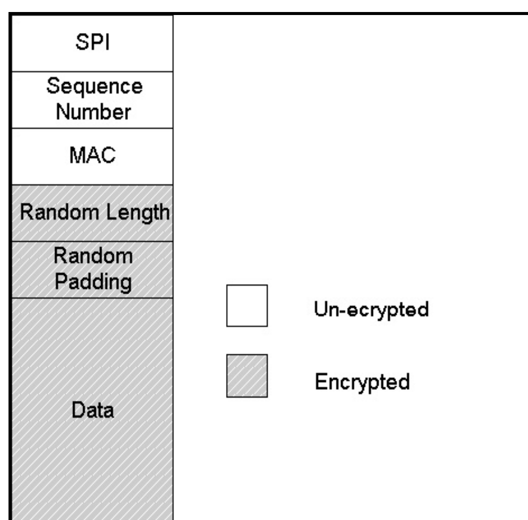


Figura 13 - Formato do pacote Secure UDP

A seguir, uma descrição dos diferentes campos presentes neste pacote:

- SPI: O campo SPI é um identificador único em uma comunicação entre duas máquinas. O SPI é estabelecido e negociado pelo *key channel*. O remetente sempre insere o SPI no pacote e o receptor usa o SPI para ajustar os protocolos de criptografia necessários para decifrar e autenticar a mensagem.

- *Sequence Number*: Este campo contém o número de sequência que é usado para prevenir os ataques por *reprodução*. Este campo é fixado em zero no cálculo do MAC.

- MAC: O campo MAC (*Message Authentication Code*) é o código de autenticação da mensagem. É usado para verificar se a mensagem não foi falsificada durante a transmissão.

- *Random length & Random padding*: São respectivamente a quantidade de dados aleatórios e os dados aleatórios inseridos pelo remetente para prevenir o ataque por análise do tráfego dos dados. Estes campos são encriptados.

- Data: Os dados encriptados da aplicação através de criptografia simétrica.

Funcionamento do Protocolo

Caso a máquina A deseje enviar de modo seguro os datagramas UDP para a máquina B, serão executados os seguintes passos para o estabelecimento de uma comunicação segura:

1. A máquina A abre uma conexão TLS (*key channel*) para máquina B.

2. Uma vez estabelecida a conexão TLS, a máquina A e máquina B estabelecem uma chave de sessão comum e uma sessão ID.

3. A máquina A inicializa o seu número de sequência para zero e começa a enviar os datagramas UDP para a máquina B que usa a sessão ID como o SPI e a chave de sessão para encriptação dos pacotes e para a geração do MAC.

4. Uma vez que o número de sequência da máquina A alcança o valor máximo, a máquina A envia uma mensagem de *client_hello* a máquina B pelo *key channel*.

5. A máquina A e a B renegociam a chave de sessão. Logo que as chaves novas são estabelecidas, a máquina A retorna a enviar os datagramas UDP.

Capítulo 3 - TLS sobre UDP

Conforme explicado no capítulo um, este novo protocolo surgiu da lacuna existente em relação a protocolos de segurança de rede disponíveis para adicionar segurança ao modo de transporte UDP. Atualmente não existe um protocolo padrão, seguro, aberto e genérico disponível para as aplicações que utilizam o UDP adicionarem segurança ao seu funcionamento (ao contrário das aplicações que utilizam o TCP). E assim, cada aplicação tem que desenvolver a sua própria solução proprietária acarretando os riscos inerentes advindos desta iniciativa.

Então, neste capítulo será apresentado o novo protocolo de criptografia de rede TLS/UDP que visa disponibilizar para comunidade em geral uma solução para o problema existente atualmente. As características do protocolo e o modo de funcionamento serão devidamente detalhados a seguir.

E para que seja prontamente utilizado o protocolo pela comunidade *open-source* será disponibilizado uma interface de programação (API) para que futuras implementações a utilizem como referência e também para que sejam assim evitadas potenciais incompatibilidades entre as mesmas. Além disso, uma implementação de referência foi codificada na linguagem Java para possibilitar o uso do protocolo em diferentes plataformas automaticamente.

3.1 Especificação do Novo Protocolo

3.1.1 Visão Geral

O objetivo principal do protocolo TLS/UDP é fornecer privacidade, garantia da integridade dos dados e autenticação na comunicação entre duas aplicações que utilizam o protocolo de transporte não confiável UDP. Ele fornece funcionalidade semelhante ao protocolo TLS e também incorpora algumas características dos protocolos WTLS e IPSec, como o uso da técnica de janela deslizante, a atualização automática da chave e suporte a comunicação via datagrama de usuário.

O protocolo TLS/UDP é composto de duas camadas: (i) Protocolo *Record* utilizado no encapsulamento e transmissão segura dos dados advindos das camadas superiores, (ii) e a camada utilizada para estabelecimento e administração das sessões seguras (Protocolo *Hanshake*, *Alert* e *Change Cipher Spec*) conforme figura 14.

O protocolo *Record* assegura que a troca de mensagens será confidencial (através do uso associado da criptografia simétrica e assimétrica) e íntegra (através do uso de MAC). A chave para a encriptação simétrica é única para cada sessão e é negociada pelo Protocolo *Handshake*. A camada *Record* encapsula, por exemplo, o protocolo *Handshake*.

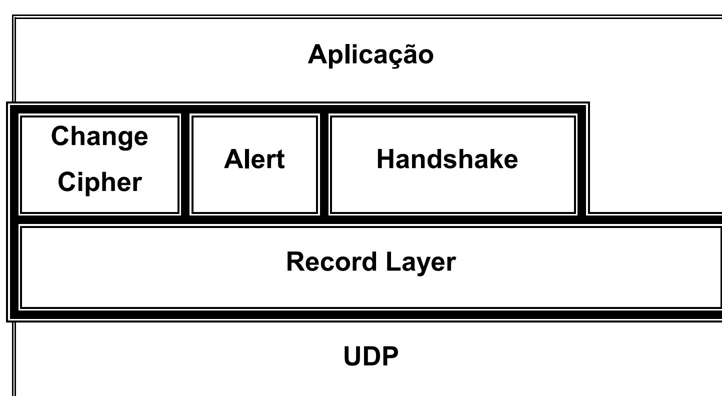


Figura 14 - Camadas TLS/UDP

O protocolo *Handshake* permite ao cliente e ao servidor autenticarem um ao outro usando algum algoritmo de encriptação assimétrico (por exemplo, RSA ou DSS) e negociarem o algoritmo de encriptação simétrico (por exemplo, IDEA ou AES) e as chaves criptográficas antes que a aplicação inicie a transmissão ou recepção

do primeiro byte de dados. A geração da chave é feita a partir do compartilhamento de um segredo comum impossível de ser espionado por terceiros (segurança).

Os algoritmos criptográficos (simétricos, assimétricos, assinatura e MAC) utilizados na comunicação serão definidos pela aplicação de acordo com a sua necessidade específica a partir do conjunto de opções disponibilizadas pelo novo protocolo e assim as aplicações podem trocar os parâmetros de segurança sem o conhecimento do código da outra parte (interoperabilidade).

O protocolo visa também fornecer uma estrutura modular onde novos algoritmos de encriptação possam ser incorporados (extensibilidade). Não é necessário modificar o protocolo para acrescentar, por exemplo, um novo algoritmo simétrico e assim a vida útil deste protocolo é prolongada.

O protocolo utiliza um mecanismo para cache de sessão com o objetivo de reduzir o atraso do início da troca de mensagens da aplicação entre os pares que já se comunicaram anteriormente (otimização).

3.1.2 Protocolo TLS/UDP *Record*

O Protocolo Record recebe as mensagens (da camada superior) a serem transmitidas, fragmenta em blocos de dados de tamanho apropriado, opcionalmente comprime os dados, aplica um MAC, encripta e transmite o resultado para a camada inferior. E o processo inverso é realizado no momento do recebimento dos dados: a mensagem é decifrada, verificada a integridade, descomprimida, remontada e a seguir a mensagem é entregue aos clientes da camada superior.

Quatro protocolos clientes são descritos neste documento: o protocolo *Handshake*, o protocolo *Alert*, o protocolo *Change Cipher Spec* e o protocolo de dados da aplicação. Para permitir a extensão do protocolo TLS/UDP, outros protocolos clientes podem ser suportados pelo protocolo *Record*. O novo protocolo deve alocar um número para o seu tipo (*ContentType*) que não esteja ainda alocado. Se uma implementação TLS/UDP recebe um tipo de protocolo que não entende deveria ignorar o datagrama.

A linguagem de apresentação utilizada neste capítulo para descrever as estruturas de dados e os atributos está definida na RFC 2246 ("The TLS Protocol").

3.1.2.1 Estados do Cliente e Servidor

Uma sessão TLS/UDP tem definido um estado que define o ambiente operacional do Protocolo *Record*. No estado da sessão são especificados: o algoritmo de compressão, o algoritmo de encriptação e algoritmo de MAC. Além destes, os parâmetros para estes algoritmos são também contemplados: o MAC *secret* e as chaves de encriptação de bloco e os IVs para a sessão em ambos os sentidos (escrita e leitura). Todos os registros são processados sob o estado corrente de leitura e escrita. O estado inicial sempre especifica que nenhuma encriptação, compressão ou MAC serão usados. Novos métodos de compressão, algoritmos simétricos e de MAC podem ser adicionados nas futuras versões deste protocolo. Os parâmetros de segurança para o estado de leitura e de escrita de uma sessão TLS/UDP são definidos pelos seguintes atributos da estrutura abaixo:

```
struct {
    SessionEnd              entity;
    BulkCipherAlgorithm     bulk_cipher_algorithm;
    CipherType              cipher_type;
    uint8                   key_size;
    uint8                   key_material_length;
    MACAlgorithm            mac_algorithm;
    uint8                   mac_size;
    CompressionMethod       compression_algorithm;
    opaque                  master_secret[48];
    opaque                  client_random[32];
    opaque                  server_random[32];
    Boolean                 Automatic key refresh;
} SecurityParameters
```

enum { *server*, *client* } *SessionEnd*;

enum { *null*, DES_CBC, ... } *BulkCipherAlgorithm*;

enum { *stream*, *block* } *CipherType*;

enum { *null*, MD5, SHA } *MACAlgorithm*;

enum { *null* (0), (255) } *CompressionMethod*;

enum { *false (0), true (1)* } *Boolean;*

A seguir, os atributos da estrutura *SecurityParameters* são explicados:

Session End

Indica se a entidade em questão é considerada como cliente ou servidor na sessão em destaque.

Bulk Encryption Algorithm

O algoritmo simétrico utilizado para encriptação. Esta especificação inclui o tamanho da chave do algoritmo, o tipo do algoritmo e o tamanho do bloco, se necessário.

MAC Algorithm

O algoritmo utilizado para autenticação da mensagem. Esta especificação inclui o tamanho do *hash* o qual é retornado pelo algoritmo MAC.

Compression Algorithm

O algoritmo utilizado para compressão dos dados. Esta especificação tem que incluir todas as informações necessárias para o funcionamento do algoritmo.

Master Secret

Um segredo de 48 bytes compartilhado entre as duas partes da sessão.

Client Random

Um valor aleatório de 32 bytes fornecido pelo cliente.

Server Random

Um valor aleatório de 32 bytes fornecido pelo servidor.

Automatic Key Refresh

Um valor booleano (*true* ou *false*) que indica se haverá ou não atualização automática do material criptográfico. *Default*: *true*.

Refresh Time

O tempo em segundos no qual a atualização automática do material criptográfico será realizada e repetida. O tempo *default* são 28800 segundos (oito horas).

A camada Record utiliza alguns parâmetros de segurança acima para derivar os seguintes atributos da sessão tanto para o servidor como para o cliente:

client write MAC secret

O valor do segredo utilizado no cálculo e na verificação dos MACs para cada registro transmitido pelo cliente. Este segredo será atualizado se a flag *Automatic Key Refresh* estiver ativa.

server write MAC secret

O valor do segredo utilizado no cálculo e na verificação dos MACs para cada registro transmitido pelo servidor. Este segredo será atualizado se a flag *Automatic Key Refresh* estiver ativa.

client write encryption key

A chave simétrica utilizada para encriptação e decriptação dos registros transmitidos pelo cliente. Esta chave será atualizado se a *flag Automatic Key Refresh* estiver ativa.

server write encryption key

A chave simétrica utilizada para encriptação e decriptação dos registros transmitidos pelo servidor. Esta chave será atualizado se a flag *Automatic Key Refresh* estiver ativa.

client write IV

O vetor de inicialização IV utilizado no processamento dos algoritmos de bloco no modo CBC para os registros transmitidos pelo cliente. Este vetor será atualizado a cada datagrama encriptado transmitido e tem que ter o mesmo tamanho que o tamanho do bloco do algoritmo selecionado.

server write IV

O vetor de inicialização IV utilizado no processamento dos algoritmos de bloco no modo CBC para os registros transmitidos pelo servidor. Este vetor será atualizado a cada datagrama encriptado transmitido e tem que ter o mesmo tamanho que o tamanho do bloco do algoritmo selecionado.

Os parâmetros de escrita do cliente (*client write*) são usados pelo servidor quando está recebendo e processando os dados e vice-versa (*server write*). O algoritmo usado para derivar estes parâmetros de segurança é descrito na seção 3.1.6.4.

Uma vez que foram inicializados os parâmetros de segurança e as chaves foram geradas, os parâmetros do estado da sessão tem que ser atualizados para cada registro processado. Cada estado da sessão inclue ainda os seguintes atributos:

cipher state

O estado atual do algoritmo de encriptação. Consiste da chave estabelecida para a sessão. E adicionalmente para os algoritmos de bloco no modo CBC, conterá o IV da sessão que é atualizado e enviado explicitamente no cabeçalho TLS/UDP a cada datagrama enviado. Para os algoritmos de fluxo, irá conter a informação necessária para permitir que os dados do *stream* continuem sendo encriptados ou decriptados.

client write sequence number

O número de sequência utilizado nos registros transmitidos pelo cliente. O número de sequência é do tipo unit64 e não pode exceder o valor $2^{64} - 1$. O número de sequência é colocado em zero quando o estado da sessão é inicializado (após o término do processo de *Handshake*). Este número é incrementado após o processamento de cada datagrama.

server write sequence number

O número de sequência utilizado nos registros transmitidos pelo servidor. O número de sequência é do tipo unit64 e não pode exceder o valor $2^{64} - 1$. O número de sequência é colocado em zero quando o estado da sessão é inicializado (após o envio da mensagem ChangeCipherSpec). Este número é incrementado após o processamento de cada datagrama.

O vetor de inicialização (IV) inicial é gerado de forma aleatória obrigatoriamente e precisa ser atualizado a cada datagrama enviado. Um método comumente utilizado é designar o último bloco de dados encriptados do processo de encriptação como IV para o próximo processo de encriptação. Isto estende o CBC através dos datagramas e também tem a vantagem de limitar o vazamento de informação sobre o gerador de números aleatórios (outra opção, muito utilizada para a geração dos IVs).

3.1.2.1.1 Número de sequência

Como mencionado no Apêndice B, os datagramas UDP podem ser perdidos, duplicados ou recebidos fora de ordem. Assim sendo, os números de sequência são utilizados para descartar os datagramas duplicados e assim evitar o *replay attack* e diminuir os efeitos de um ataque de negação de serviço. Esta tarefa é realizada através do uso da técnica de janela deslizante de forma equivalente a utilizada no IPSec.

Utilizando por exemplo, uma janela de tamanho igual a 32 unidades, o receptor pode manter uma lista dos datagramas recebidos na faixa de [n-32 +1] até n. Onde n é o número de sequência corrente e também o maior número dentro da janela. Os datagramas com números de sequência menor ou igual a [n-32] são descartados. Se o datagrama é validado e tem número maior que n, então o valor n tem que ser atualizado com este valor e a janela efetivamente avança. Caso o datagrama tenha o número de sequência dentro da janela de recepção e seja duplicado, este será descartado e caso contrário será armazenado na lista caso seja validado.

O tamanho da janela recomendado é de 128 ou maior, sendo um atributo definido na implementação.

Quando o *Handshake* inicia com a troca de mensagens em texto puro, o número de sequência é inicializado em zero e é incrementado a cada mensagem de *Handshake*. E o número de sequência é reinicializado para zero após a mensagem *ChangeCipherSpec* em ambas as direções.

Em retransmissões, o número de sequência tem que ser mantido o mesmo da mensagem original. Quando o *Handshake* é iniciado dentro de uma sessão segura, o número de sequência continua sendo incrementado para cada mensagem enviada.

3.1.2.2 Camada *Record*

A camada *Record* recebe os dados das camadas superiores em blocos não vazios de tamanho máximo de $2^{14} - 1$ bytes.

3.1.2.2.1 Fragmentação

A camada *Record* fragmenta os dados da informação em registros conhecidos como *TLS/UDP_Plaintext* de tamanho máximo de $2^{14} - 1$ bytes. Os limites da mensagem do cliente não são preservados na camada *Record*, isto é, múltiplas mensagens do cliente com mesmo *ContentType* podem ser combinadas em um simples registro *TLS/UDP_PlainText* ou uma simples mensagem pode ser fragmentada em vários registros. A seguir, são descritos os vários campos de uma estrutura *TLS/UDP_PlainText*:

```
struct {
        uint8 major, minor;
} ProtocolVersion;
enum {
        change_cipher_spec (20), alert (21), handshake (22),
        application_data (23), (255)
} ContentType;

struct {
        ContentType         type;
        ProtocolVersion     version;
        uint64              sequence_number
        uint16              length;
        opaque              fragment[TLS/UDP_Plaintext.length];
} TLS/UDP_Plaintext;
```

type

O protocolo da camada superior usado no fragmento empacotado.

version

A versão do protocolo sendo empregado. Este documento descreve o TLS/UDP versão 1.0.

length

O tamanho em bytes do fragmento TLS/UDP_Plaintext (*TLS/UDP_Plaintext.fragment*). Este tamanho não pode exceder 2^{14} bytes.

sequence_number

O número de sequência é transmitido em texto puro (não encriptado) e é usado como uma das entradas no cálculo do MAC. É utilizado principalmente para proteção contra ataques por *reprodução* e de negação de serviço. Este valor ser obtido opcionalmente pela camada superior para conhecimento sobre a ordenação dos datagramas.

fragment

Os dados da aplicação (ou da camada superior). Os dados são transparentes e tratados como um bloco independente a ser cuidado pelo protocolo da camada superior especificado pelo campo *type*.

3.1.2.2.2 Compressão e Descompressão

Todos os registros podem ser comprimidos usando algum algoritmo de compressão definido pelo estado da sessão. Há sempre um algoritmo de compressão ativo, sendo inicialmente definido como *CompressionMethod.null*. Observar que um algoritmo de compressão *stateful* (ver glossário) não pode ser utilizado pelo fato do UDP não ser orientado a conexão e sem garantia de entrega dos datagramas.

O algoritmo de compressão transforma a estrutura *TLS/UDP_Plaintext* em uma estrutura *TLS/UDP_Compressed*. A compressão tem que ser sem perda e não pode incrementar mais de 1024 bytes o fragmento. Se a função de descompressão gera um fragmento de tamanho superior ao permitido (2^{14} bytes), deve ser reportado um alerta fatal *"decompression failure error"*.

```
struct {
        ContentType          type;
        ProtocolVersion      version;
        uint64               sequence_number
        uint16               length;
        opaque               fragment[TLS/UDP_Compressed.length];
} TLS/UDP_Compressed;
```

length

O tamanho em bytes do fragmento comprimido (*TLS/UDP_Compressed.fragment).* O tamanho não pode exceder (2^{14} + 1024) bytes. Somente este campo é atualizado quando da transformação da estrutura *TLS/UDP_Plaintext* em *TLS/UDP_Compressed,* os outros campos são idênticos ao da estrutura. *TLS/UDP_Plaintext*.

fragment

A forma comprimida do fragmento *TLS/UDP_Plaintext*.

3.1.2.2.3 Proteção dos Dados

As funções de encriptação e MAC transformam uma estrutura *TLS/UDP_Compressed* em uma estrutura *TLS/UDP_Ciphertext*. A função de decriptação reverte o processo.

```
struct {
        ContentType          type;
        ProtocolVersion      version;
        uint64               sequence_number
        uint16               length;
        select (SecurityParameters.cipher_type) {
                case stream:   NotImplemented;
                case block:    IV;
        } ExplicitVector;
```

```
        select (SecurityParameters.cipher_type) {
                case stream:    GenericStreamCipher;
                case block:     GenericBlockCipher;
        } fragment;
} TLS/UDP_Ciphertext;
```

type

O campo *type* é idêntico ao *TLS/UDP_Compressed.type*.

version

O campo *version* é idêntico ao *TLS/UDP_Compressed.version*.

sequence_number

O campo *sequence_number* é idêntico ao *TLS/UDP_Compressed.sequence_number*.

length

O tamanho em bytes do *TLS/UDP_Ciphertext.fragment*. O tamanho não pode exceder (2^{14} + 2048) bytes.

ExplicitVector

Para os algoritmos de bloco em modo CBC é o vetor de inicialização (IV) enviado explicitamente no cabeçalho do datagrama TLS/UDP para tornar a decriptação de cada datagrama independente mesmo que alguns datagramas sejam perdidos ou reordenados no caminho. Para os algoritmos de fluxo está característica não foi implementada nesta versão do protocolo.

fragment

A forma encriptada do *MAC* e *TLS/UDP_Compressed.fragment* e *paddings* se necessário.

3.1.2.2.3.1 Algoritmo de Fluxo (*Stream Cipher*)

Os algoritmos simétricos de fluxo convertem a estrutura *TLS/UDP_Compressed.fragment* de/para a estrutura *TLS/UDP_Ciphertext.fragment*.

stream-ciphered struct {
opaque content [TLS/UDP_Compressed.length];
opaque MAC[SecurityParameters.hash_size];
} GenericStreamCipher;

O *MAC* é computado da seguinte maneira:

HMAC_hash (MAC_write_secret, TLS/UDP_Compressed.sequence_number + TLS/UDP_Compressed.type + TLS/UDP_Compressed.version + TLS/UDP_Compressed.length + TLS/UDP_Ciphertext.ExplicitVector + TLS/UDP_Compressed.fragment)

O sinal + representa a operação lógica de concatenação. E o algoritmo de *HMAC_hash* utilizado será o especificado pelo *SecurityParameters.mac_algorithm*. Observe que o MAC é computado antes da encriptação.

O *Stream Cipher* encripta o bloco inteiro, incluindo o MAC. Para *Stream Ciphers* que não utilizam um vetor de sincronização (como o RC4), o estado do final de um registro é diretamente usado no datagrama subsequente (porém não pode ser usado desta forma no TLS/UDP).

Não está previsto o uso de algoritmos simétricos de fluxo para encriptação dos fragmentos nesta versão do protocolo TLS/UDP.

3.1.2.2.3.2 CBC block cipher

Para os algoritmos simétricos de bloco como DES e 3DES, as funções de encriptação e o MAC convertem a estrutura *TLS/UDP_Compressed.fragment* na estrutura *TLS/UDP_Ciphertext.fragment*.

block-ciphered struct {

opaque content[TLSCompressed.length];

opaque MAC[SecurityParameters.hash_size];

uint8 padding[GenericBlockCipher.padding_length];

uint8 padding_length;

} GenericBlockCipher;

O MAC é computado conforme descrito na seção anterior.

padding

Padding são bits que são adicionados ao fragmento para tornar o tamanho do texto puro múltiplo do tamanho do bloco do algoritmo de encriptação. O tamanho do *padding* pode ser maior que o tamanho de um bloco, porém está limitado ao máximo de 255 bytes. O *padding* dificulta os ataques baseados na análise dos tamanhos das mensagens trocadas.

padding_length

O tamanho do *padding* deveria ser tal que o tamanho total da estrutura *GenericBlockCipher* seja múltiplo do tamanho do bloco do algoritmo de bloco. O tamanho pode variar de 0 a 255, inclusive.

O tamanho dos dados encriptados (*TLS/UDP_Ciphertext.length*) é igual a soma do [*TLS/UDP_Compressed.length* + S*ecurityParameters.hash_size* + *padding_length* + 1]

Exemplo: Se o tamanho do bloco do algoritmo é 8 bytes, o tamanho do conteúdo dos dados é (*TLSCompressed.length*) é 61 bytes e o tamanho do MAC é 20 bytes, o tamanho antes do *padding* será de 82 bytes. Assim, o tamanho do [*padding* módulo 8] tem que ser igual a 6 para tornar o tamanho total do bloco múltiplo de 8 bytes. O tamanho do *padding* pode ser 6, 14, 22 e assim por diante, até 254. Se o padding adicionado tiver o tamanho mínimo necessário (6), cada byte do

padding irá conter o valor 6. Assim, os últimos 8 octetos do *GenericBlockCipher* antes da encriptação do bloco serão xx 06 06 06 06 06 06 06, onde xx é o último octeto do MAC.

3.1.3 Protocolo *Change Cipher Spec*

O protocolo *Change Cipher Spec* tem como função sinalizar mudança na estratégia de encriptação. O protocolo consiste de uma única mensagem, a qual é comprimida e encriptada sob o estado da sessão corrente. A mensagem consiste de um único byte de valor 1.

```
struct {
    enum { change_cipher_spec(1), (255) } type;
} ChangeCipherSpec;
```

A mensagem *Change Cipher Spec* pode ser transmitida tanto pelo cliente e pelo servidor para notificar a parte que receber que os subsequentes registros serão protegidos com o novo conjunto criptográfico (*CipherSpec* e chaves) negociado. O envio (recebimento) desta mensagem gera a modificação imediata do estado da sessão no transmissor (receptor). A mensagem *Change Cipher Spec* é transmitida durante o *handshake* após os parâmetros de segurança terem sido negociados e acordados, e antes da mensagem *finished* de verificação ser transmitida.

3.1.4 Protocolo Alert

Um dos tipos de conteúdos suportados pela camada *Record* é a informação enviada pelo protocolo *Alert*. As mensagens de alerta carregam a severidade e a descrição da mensagem. O recebimento de uma mensagem com um grau de severidade fatal resulta no término imediato da sessão, e além disso, o identificador da sessão (*Session ID*), as chaves e os segredos associados são invalidados também. Assim como as outras, as mensagens de alerta são comprimidas e encriptadas, conforme especificado no estado da sessão corrente.

```
enum { warning(1), fatal(2), (255) } AlertLevel;

enum {
        close_notify(0),
        unexpected_message(10),
        bad_record_mac(20),
        decryption_failed(21),
        record_overflow(22),
        decompression_failure(30),
        handshake_failure(40),
        bad_certificate(42),
        unsupported_certificate(43),
        certificate_revoked(44),
        certificate_expired(45),
        certificate_unknown(46),
        illegal_parameter(47),
        unknown_ca(48),
        access_denied(49),
        decode_error(50),
        decrypt_error(51),
        duplicate_finished_received (57),
        export_restriction(60),
        protocol_version(70),
```

insufficient_security(71),

internal_error(80),

user_canceled(90),

no_renegotiation(100),

(255)

} AlertDescription;

struct {

AlertLevel level;

AlertDescription description;

} Alert;

3.1.4.1 Alertas de Encerramento de Sessão

O cliente e o servidor tem que compartilhar o conhecimento de que uma sessão esteja terminada para impedir um *truncation attack*. Ambas as partes podem iniciar a troca de mensagens para terminar uma sessão.

close_notify

Esta mensagem notifica o receptor que o transmissor não irá mais transmitir mensagens nesta sessão. A sessão torna-se não reutilizável (*unresumable*) se a comunicação é terminada sem o envio desta mensagem com nível de severidade *warning*. Ambas as partes podem realizar o fechamento enviando o alerta *close_notify*. Qualquer dado recebido após o fechamento será ignorado.

O receptor da mensagem tem que responder com o mesmo alerta *close_notify* e fechar o canal de comunicação imediatamente, descartando qualquer dado pendente. Não é requerido que o solicitante do processo de término da comunicação espere a resposta para fechar o seu canal de leitura da sessão correspondente. A implementação do TLS/UDP tem que avisar a aplicação que a sessão foi terminada.

3.1.4.2 Alertas de Erros

O tratamento dos erros no protocolo TLS/UDP é muito simples. Quando um erro é detectado, a parte que detectou transmite uma mensagem para a outra. Os seguintes alertas de erro estão definidos (e outros erros podem ser adicionados em uma próxima versão do protocolo):

unexpected_message

Uma mensagem não esperada foi recebida. Este alerta é normalmente fatal e nunca deveria acontecer em uma comunicação entre implementações do protocolo realizadas de acordo com a especificação. Ver tópico referente a confiabilidade do *Handshake* (página 54).

bad_record_mac

Este alerta é recebido se um registro é recebido com um MAC incorreto. Esta mensagem é sempre fatal.

decryption_failed

Uma decriptação foi realizada de um modo inválido. O TLS/UDP_Ciphertext não era múltiplo do tamanho do bloco ou os valores de *padding* não estavam corretos quando checados (*Ciphertext* mal formado). Essa mensagem é sempre fatal.

record_overflow

O registro *TLS/UDP_Ciphertext* recebido tinha um tamanho maior que (2^{14} + 2048) bytes ou o registro decriptado para *TLS/UDP_Compressed* tinha mais que (2^{14} + 1024) bytes. Esta mensagem é sempre fatal.

decompression_failure

A função de descompressão recebeu uma entrada inapropriada. Esta mensagem é sempre fatal.

handshake_failure

A recepção deste alerta indica que o transmissor não estava habilitado a negociar um conjunto de parâmetros de segurança aceitáveis a partir das opções disponíveis. É uma mensagem de erro fatal.

bad_certificate

O certificado recebido estava corrompido. Exemplo: As assinaturas contidas não foram validadas.

unsupported_certificate

O tipo de certificado recebido não é suportado.

certificate_revoked

O certificado foi revogado pelo assinantes.

certificate_expired

O certificado está expirado ou não está atualmente válido.

certificate_unknown

Algum problema surgiu no processamento do certificado.

illegal_parameter

Algum campo no processo de handshake estava fora da faixa de valores possíveis ou inconsistente com os outros campos. Este alerta é sempre fatal.

unknown_ca

O certificado da Autoridade Certificadora (CA) não pode ser localizado ou não é uma autoridade certificadora confiável. Esta mensagem é sempre fatal.

access_denied

Um certificado válido foi recebido mas o controle de acesso decidiu que a negociação não será continuada. Este alerta é sempre fatal.

decode_error

Uma mensagem não pode ser estratificada (*decoded*) porque algum campo estava fora da faixa especificada ou o tamanho da mensagem estava incorreto. Este alerta é sempre fatal.

decrypt_error

Uma operação criptográfica no processo de comunicação falhou. Não se pode verificar a assinatura ou decriptar a chave pré-master ou validar a mensagem *finished*.

duplicate_finished_received

Em um *Handshake* otimizado, o cliente transmitiu uma segunda (retransmissão) mensagem *Finished.* Este alerta é geralmente um *warning* (não é fatal).

protocol_version

O cliente ou o servidor tentou negociar uma versão de protocolo que não é suportada. Este alerta é sempre fatal.

insufficient_security

Enviado no lugar de *handshake_failure* quando uma negociação falhou especificamente porque o servidor requisitou cifras mais seguras que o cliente suportava. Este alerta é sempre fatal.

internal_error

Um erro interno não relacionado com o protocolo torna impossível a continuação da comunicação (por exemplo: uma falha de alocação de memória). Este alerta é sempre fatal.

user_canceled

O handshake foi cancelado por alguma razão não relacionada com o protocolo. Se o usuário cancela a operação após o handshake estar completo, o término da comunicação através de um *close_notify* é mais apropriado. Este alerta deveria ser seguido por um *close_notify*. Esta mensagem é geralmente um *warning*.

no_renegotiation

Este alerta é transmitido pelo cliente em resposta a um *hello request* ou pelo servidor em resposta a um *client hello* após o inicial *handshaking*. Quando um dos lados verifica que a renegociação é inapropriada, ele envia este alerta. O originador da renegociação pode decidir se continua com a sessão atual ou não. Este alerta é sempre um *warning*.

Para todos os erros onde o nível de severidade do alerta não está explicitamente especificado, a parte que envia o alerta pode determinar a seu critério se o alerta será um erro fatal ou não. Se um alerta com nível de *warning* é recebido, a parte que recebe a mensagem pode decidir a seu critério se tratará este alerta como fatal ou não. Contudo todos os alertas transmitidos com um nível de severidade fatal tem que ser tratados como tal.

3.1.5 Protocolo *Handshake*

O protocolo *Handshake* é utilizado pelas partes da comunicação para: (i) a combinação segura dos parâmetros de criptografia utilizados na camada Record e (ii) para a autenticação das fontes de informação. O protocolo é responsável pela negociação e estabelecimento de uma sessão segura de comunicação, a qual consiste nos seguintes atributos descritos a seguir:

session identifier

É uma sequência de bytes arbitrária escolhida pelo servidor para identificar uma sessão ativa ou reutilizável.

peer certificate

Ë o certificado X509v3 [13] da entidade. Este elemento pode ser nulo.

compression method

O algoritmo uutilizado para comprimir os dados antes da encriptação.

cipher spec

Especifica o algoritmo de encriptação simétrico (null, DES, IDEA, ...) e um algoritmo de MAC (como MD5 ou SHA-1). Ele também define atributos criptográficos como o tamanho do *hash* (tabela 2).

master secret

Um segredo de 48 bytes compartilhado entre o cliente e o servidor.

is resumable

Um *flag* indicando se a sessão pode ser reutilizada para iniciar novas comunicações entre as partes.

O protocolo *Handshake* estabelece os seguintes passos operacionais:

- Troca de mensagens de hello para combinar os algoritmos criptográficos e os valores aleatórios e checagem da viabilidade de reutilização da sessão.
- Troca dos parâmetros criptográficos necessários para permitir o cliente e o servidor combinarem uma *premaster secret*.
- Troca de certificados e outras informações criptográficas para permitir o cliente e o servidor autenticarem um ao outro.
- Gerar uma chave *master secret* a partir da *premaster secret*.
- Fornecer os parâmetros de segurança necessários a camada *Record*.
- Possibilitar ao cliente e ao servidor verificar a integridade de todo o processo de *handshake* (que não houve adulteração ou falsificação das mensagens)

Processo de handshake

O cliente envia um *client hello* para o qual o servidor tem que responder com uma mensagem *server hello*. O *client hello* e o *server hello* estabelecem os seguintes atributos: *Protocol Version, Session ID, Cipher Suite* e *Compression Method*. Além disso, há uma troca de números aleatórios gerados tanto pelo cliente (*ClientHello.random*) como pelo Servidor (*ServerHello.random*).

Após a troca de mensagens de *hello*, o servidor envia o seu certificado, caso seja requisitada a sua autenticação. Além disso, uma mensagem *server key exchange* pode ser transmitida, se necessário (exemplo: caso o servidor não tenha um certificado ou se o certificado é apenas de assinatura somente). Se o servidor é autenticado, ele pode requisitar um certificado do cliente. Agora o servidor envia uma mensagem *server hello done* indicando que a fase de *hello* do handshake foi completada. O servidor irá então esperar por uma resposta do cliente. Se o servidor enviou uma mensagem *certificate request*, o cliente tem que enviar a mensagem *certificate*. A mensagem *client key exchange* é agora transmitida e o conteúdo da mensagem dependerá do algoritmo de chave pública selecionado nas mensagens de *client hello* e o *server hello*. Se o cliente transmitiu um certificado com a facilidade da assinatura,

uma mensagem *certificate verify* digitalmente assinada deve ser transmitida para verificar o certificado explicitamente.

A troca de chaves utiliza até quatro mensagens: *server certificate*, *server key exchange*, *client certificate* e *client key exchange*. Novos métodos de troca de chaves podem ser criados através da especificação de um formato para estas mensagens e definindo o uso das mensagens para permitir ao cliente e ao servidor combinar um segredo. Atualmente, os métodos atuais trocam segredos na faixa de 48 a 128 bytes de tamanho.

Após a definição das chaves, a mensagem *change cipher spec* é transmitida pelo cliente e este torna o *Cipher Spec* (conjunto criptográfico) pendente no corrente. O cliente então imediatamente transmite a mensagem *finished* já utilizando os novos algoritmos, chaves e segredos. Em resposta, o servidor transmite a sua própria mensagem de *change cipher spec* e torna o *Cipher Spec* pendente no corrente e transmite a mensagem *finished* já utilizando o novo *Cipher Spec*. Neste ponto, o *handshake* está completo e o cliente e o servidor podem começar a trocar dados da camada de aplicação de forma segura. Ver diagrama de fluxo na figura 15.

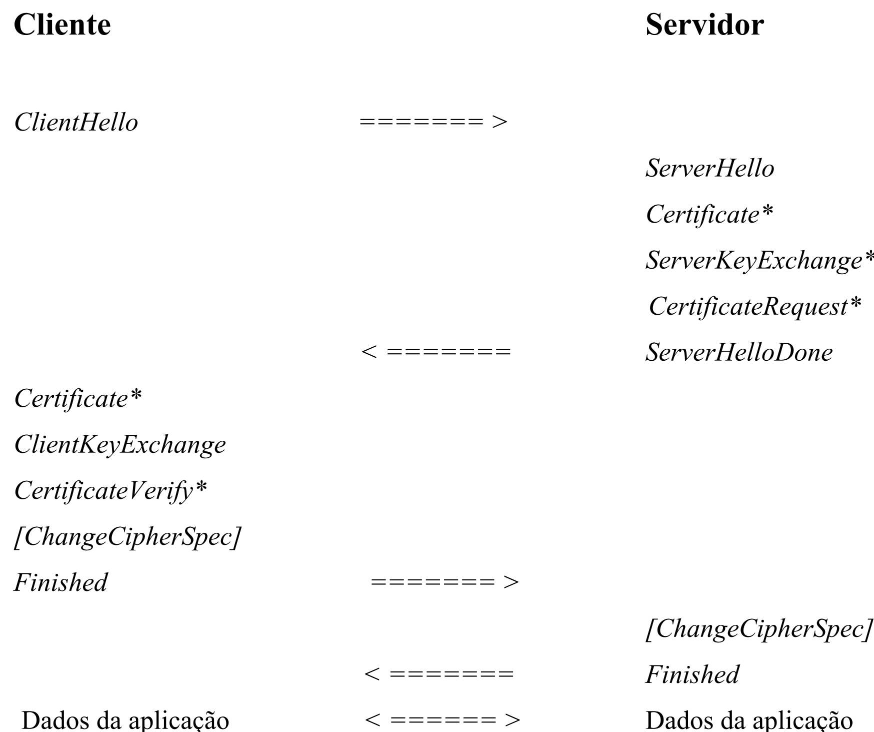


Figura 15 - Fluxo de mensagens para o handshake completo

* Indica que a mensagem é opcional ou dependente da situação (nem sempre é enviada).

Quando o cliente e o servidor decidem retomar uma sessão anterior ou duplicar uma sessão existente (ao invés de negociar novos parâmetros de segurança), o fluxo das mensagens é abreviado conforme explicado a seguir.

O cliente envia um *ClientHello* usando a *session ID* da sessão a ser retomada. O servidor então checa o seu cache de sessões para encontrar o ID referenciado, caso encontre estará pronto para restabelecer a sessão, e então enviará um *ServerHello* com o mesmo valor de *Session ID*. A seguir, tanto o cliente como o servidor enviam a mensagem de *Change Cipher Spec* e depois enviam a mensagem *finished*. Uma vez restabelecida a sessão, os dados da camada de aplicação podem ser trocados. Ver figura 16. Se a *session ID* não é encontrada no cache do servidor, o servidor gera uma nova *session ID* e assim cliente e servidor executam um handshake completo.

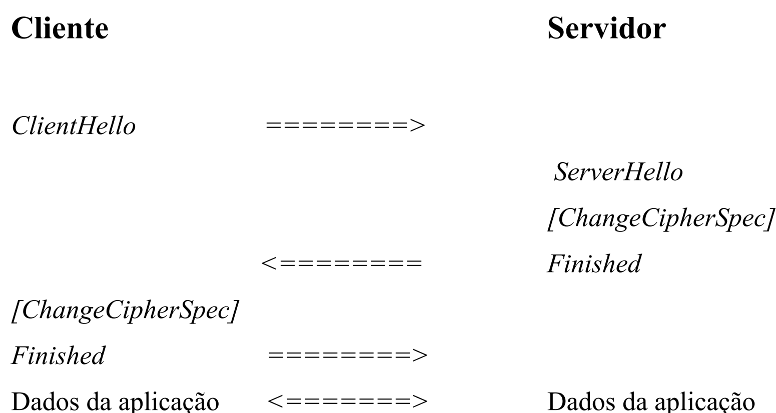


Figura 16 - Fluxo de mensagens para o handshake abreviado

Confiabilidade do Handshake

Como as mensagens de *Handshake* podem ser perdidas, duplicadas ou entregues desordenadas. Para tornar o processo de *Handshake* confiável, este protocolo requer que as mensagens de *Handshake* dirigidas em um mesmo sentido devem ser concatenadas em uma única unidade de serviço de dados (SDU, *Service Data Unit*) para serem transmitidas. E o cliente pode retransmitir as mensagens de *Handshake* se necessário e o servidor tem que responder apropriadamente as mensagens retransmitidas do cliente.

Em certos momentos, o *Handshake* pode consistir de múltiplas mensagens enviadas em um mesmo sentido antes de qualquer resposta ser requerida da outra parte. E estas mensagens devem ser concatenadas em um único SDU para garantir que todas as mensagens enviadas neste SDU serão entregues na ordem correta e além disso, diminui o número de mensagens que podem ter problemas no caminho de rede. Por exemplo, as mensagens *ServerHello*, *ChangeCipherSpec* e *Finished* podem ser transmitidas em um único SDU no *Handshake* abreviado. O tamanho máximo do SDU suportado nas camadas inferiores deve ter tamanho suficiente para conter as mensagens concatenadas de *Handshake*.

Para o *Handshake* completo, o cliente tem que retransmitir as mensagens *ClientHello* e *Finished* se a esperada mensagem de resposta não for recebida do servidor em um tempo de expiração (*time-out*) predefinido. Observe que o SDU completo que contém a mensagem *Finished* tem que ser retransmitido. Se o número máximo de retransmissões exceder o número máximo predefinido, o cliente termina o procedimento de *Handshake*.

Para um *Handshake* abreviado, assim como o completo, o cliente retransmite a mensagem *ClientHello* caso necessário. E tem que concatenar e retransmitir as mensagens *ChangeCipherSpec* e *Finished* junto com a mensagem da camada da aplicação até que uma mensagem da camada da aplicação seja recebida do servidor e decriptada corretamente ou que um alerta *duplicate_finished_received* seja recebido do servidor. Contudo, as mensagens *ChangeCipherSpec* e *Finished* concatenadas na primeira vez, não precisam ser transmitidas juntamente com a mensagem da camada de aplicação, se assim quiser.

Para o *Handshake* completo, o servidor tem que retransmitir o SDU que continha a mensagem *ServerHello* caso receba uma mensagem *ClientHello* duplicada. Contudo, se o *ClientHello* é novo, o servidor tem que iniciar um novo processo de *Handshake*. O servidor tem também que retransmitir o SDU que continha a mensagem *ChangeCipherSpec* e *Finished* caso receba uma mensagem *Finished* duplicada do cliente.

No *Handshake* abreviado, o servidor tem o mesmo comportamento que no *Handshake* completo no tratamento de mensagens ClientHello novas ou duplicadas. E além disso, tem que ignorar mensagens *Finished* duplicadas. Se o servidor não tem

nenhum dado de aplicação para transmitir ao cliente, ele deveria enviar um alerta *duplicated_finished_received* (*warning*).

A estrutura geral das mensagens de *handshake* está apresentada a seguir.

```
enum {
   hello_request(0), client_hello(1), server_hello(2),
   certificate(11), server_key_exchange (12),
   certificate_request(13), server_hello_done(14),
   certificate_verify(15), client_key_exchange(16),
   finished(20), (255)
} HandshakeType;

struct {
   HandshakeType msg_type;
   uint24 length;
   select (HandshakeType) {
      case hello_request:          HelloRequest;
      case client_hello:           ClientHello;
      case server_hello:           ServerHello;
      case certificate:            Certificate;
      case server_key_exchange:    ServerKeyExchange;
      case certificate_request:    CertificateRequest;
      case server_hello_done:      ServerHelloDone;
      case certificate_verify:     CertificateVerify;
      case client_key_exchange:    ClientKeyExchange;
      case finished:               Finished;
   } body;
} Handshake;
```

A seguir, as mensagens do protocolo *Handshake* são descritas na ordem que são transmitidas. Mensagens de *Handshake* transmitidas em uma ordem não esperada

resulta em um erro fatal. O formato e o conteúdo de cada mensagem será apresentado em detalhes nas subseções seguintes.

3.1.5.1 Hello Messages

As mensagens *hello* (*Hello Request, Client Hello* e *Server Hello*) são usadas para iniciar a comunicação entre as partes. Entre outras tarefas, combinam o conjunto criptográfico a ser utilizado na sessão entre o servidor e o cliente e trocam números aleatórios.

Quando uma nova sessão começa, os algoritmos de encriptação, *hash* e compressão são inicializados em *null*. O estado da sessão corrente é utilizado nas mensagens de renegociação.

3.1.5.1.1 Hello Request

A mensagem *Hello Request* pode ser transmitida pelo servidor a qualquer tempo. Esta mensagem é uma simples notificação ao cliente para que inicie o processo de negociação novamente com o envio de uma mensgaem *Client Hello* Esta mensagem pode ser ignorada pelo cliente se não desejar renegociar a sessão ou então o cliente também pode enviar um alerta *no_renegotiation*. Se o servidor enviar um *Hello Request* mas não receber um *Client Hello* em resposta, ele pode fechar a sessão com um alerta fatal.

Estrutura da mensagem:

struct { } HelloRequest;

Esta mensagem não deveria ser incluída no cálculo do *hash* realizado para a mensagem *finished* e para a mensagem *certificate verify*.

3.1.5.1.2 Client Hello

Esta mensagem é enviada quando um cliente deseja iniciar uma comunicação (*handshake*) com um servidor. O cliente também pode enviar esta mensagem em resposta a um *Hello Request* ou a sua própria iniciativa com o objetivo de renegociar os parâmetros de segurança de uma sessão em andamento.

Estrutura da mensagem:

A mensagem *Cliente Hello* inclui uma estrutura *random*, a qual é utilizada pelo protocolo posteriormente.

```
struct {
        uint32 gmt_unix_time;
        opaque random_bytes[28];
} Random;
```

gmt_unix_time

A data e hora atual no formato padrão UNIX 32 bits (o número de segundos desde a meia-noite de 01 de Janeiro de 1970, GMT) de acordo com o relógio do transmissor. Não é necessário que os relógios estejam ajustados para o protocolo TLS/UDP operar corretamente.

random_bytes

Um número de 28 bytes obtido atráves de um gerador de números aleatórios confiável.

A mensagem de *Client Hello* inclui um *session ID* de tamanho variável. Se não estiver vazio, este valor identifica uma sessão a qual o cliente deseja reutilizar os parâmetros de segurança e neste caso o processo de *handshake* é abreviado. Quando o cliente envia esta mensagem com o *session ID* atual, significa que deseja atualizar somente os valores da estrutura *random* e derivados.

O *session ID* torna-se válido apenas quando o processo de *handshake* é efetivado e é removido do cache quando expira o tempo da sessão ou por um alerta fatal ou quando a comunicação cessa por um determinado tempo. O valor do *session ID* é definido pelo servidor na mensagem *Server Hello*.

```
opaque SessionID<0..32>;
```

O *session ID* é transmitido sem encriptação e sem proteção direta da sua integridade. O servidor deve precaver-se contra eventuais mensagens falsificadas com *session IDs* válidos. Observar que o conteúdo de todas as mensagens trocadas no processo de *handshake* como um todo, incluindo o *session ID*, é protegido pela troca de mensagens *finished* ao final do processo de *handshake*.

A lista dos conjuntos criptográficos (*CipherSuite*) passada do cliente para o servidor na mensagem de *Cliente Hello* contém as combinações dos blocos criptográficos suportados pelo cliente e enviadas na ordem de preferência. Cada combinação define um algoritmo para troca de chaves, um algoritmo simétrico (incluindo o tamanho da chave secreta) e um algoritmo MAC. O servidor selecionará um destes conjuntos, ou caso não encontre uma opção aceitável responde com um alerta *handshake failure* e termina o processo de *handshake*.

```
uint8 CipherSuite[2];
```

A mensagem *ClientHello* inclui também uma lista dos algoritmos de compressão suportados pelo cliente, ordenados de acordo com a sua preferência.

```
enum { null(0), (255) } CompressionMethod;
struct {
        ProtocolVersion client_version;
        Random random;
        SessionID session_id;
        CipherSuite cipher_suites<2..2^16-1>;
        CompressionMethod compression_methods<1..2^8-1>;
} ClientHello;
```

client_version

A versão do protocolo TLS/UDP a qual o cliente deseja comunicar com o servidor durante a sessão. Este valor deveria conter a versão mais recente suportada. A versão atual e única é a 1.0.

random

Uma estrutura *random* gerada pelo cliente.

session_id

O ID da sessão a qual o cliente deseja utilizar nesta comunicação. Este campo deve ser nulo caso o cliente deseje gerar novos parâmetros de segurança.

cipher_suites

A lista das opções dos conjuntos criptográficos suportados pelo cliente na ordem de preferência (descendente). Se o campo *session_id* não está vazio (implicando na requisição de reutilização de alguma sessão anterior), este vetor tem que incluir no mínimo o conjunto criptográfico da sessão anterior referenciada (tabela 2).

compression_methods

A lista dos métodos de compressão suportados pelo cliente, na ordem de preferência. Se o campo *session_id* não está vazio (implicando na requisição de reutilização de alguma sessão anterior), tem que ser incluído o método da sessão anterior referenciada. Este vetor tem que conter a opção *CompressionMethod.null* e todas as implementações têm que suportar. Assim, o cliente e o servidor sempre estarão habilitados a combinar um método de compressão.

Após enviar a mensagem de *Client Hello*, o cliente espera por uma mensagem *Server Hello*. Qualquer outra mensagem retornada pelo servidor, exceto um *Hello Request*, será tratada como um erro fatal.

3.1.5.1.3 Server Hello

O servidor envia esta mensagem em resposta a uma mensagem *Client Hello* quando estiver de acordo com alguma opção da lista dos conjuntos criptográficos que o cliente disponibilizou. Se não encontrar um conjunto aceitável, ele responderá com um alerta *handshake failure*.

Estrutura da mensagem

```
struct {
        ProtocolVersion server_version;
        Random random;
        SessionID session_id;
        CipherSuite cipher_suite;
        CompressionMethod compression_method;
} ServerHello;
```

server_version

Este campo contém a versão mais antiga sugerida pelo cliente na mensagem Client Hello e a mais alta suportada pelo servidor. A versão atual e única desta especificação é 1.0.

random

Esta estrutura é gerada pelo servidor e tem que ser diferente (e independente) do *ClientHello.random*.

session_id

Este é o identificador da sessão correspondente a comunicação. Se o *ClientHello.session_id* não estava vazio, o servidor irá procurar este ID no seu cache. Se encontrar, o servidor estará pronto para estabelecer uma nova comunicação utilizando usando o estado da sessão referenciada, o servidor então responde com o mesmo valor que foi enviado pelo cliente e assim a retomada de sessão é realizada onde os

participantes passam diretamente para as mensagens *change cipher spec* e *finished*. Caso o servidor não encontre o ID, enviará um valor de ID diferente indicando o início de uma nova sessão. O servidor pode responder com um *session_id* vazio para indicar que a sessão não estava no cache e portanto não pode ser reutilizada. Se a sessão for retomada, o mesmo conjunto criptográfico (*CipherSuite*) negociado originalmente será reutilizado.

cipher_suite

Um único conjunto criptográfico selecionado pelo servidor da lista de opções disponibilizadas pelo cliente em ClientHello.cipher_suites. Para sessões retomadas, este campo será alocado com o estado da sessão anterior referenciada.

compression_method

Um único algoritmo de compressão selecionado pelo servidor da lista de opções disponibilizadas pelo cliente. Para sessões retomadas, este campo será alocado com o estado da sessão anterior referenciada.

3.1.5.2 Server Certificate

O servidor tem que enviar um certificado digital quando o método para troca de chaves combinado não é do tipo anônimo. Esta mensagem é sempre enviada imediatamente após a mensagem *Server Hello*.

O tipo de certificado tem que estar de acordo com o algoritmo para troca de chaves selecionado. Geralmente é um certificado X.509v3 e contém a chave a qual combina com o método de troca de chaves combinado. A menos que especificado de outra forma, o algoritmo para assinatura do certificado tem que ser o mesmo utilizado para a chave pública do certificado. E se não for especificado, a chave pública pode ter qualquer tamanho.

Tabela 1 - Algoritmo para troca de chaves X Certificado

Algoritmo para troca de chaves	Tipo da Chave do Certificado
RSA	Chave pública RSA (o certificado tem que permitir que a chave seja usada para encriptação)
DHE_DSS	Chave pública DSS
DHE_RSA	Chave pública RSA que pode ser usada para assinatura
DH_DSS	Chave Diffie-Hellman. O algoritmo usado para assinar o certificado deve ser o DSS
DH_RSA	Chave Diffie-Hellman. O algoritmo usado para assinar o certificado deve ser o RSA

Todos os atributos (*profiles*) do certificado, chaves e formato são definidos pelo grupo de trabalho IETF PKIX [14]. O bit *digitalSignature* tem que estar ativado (*set*) para a chave ser elegível de assinatura e o bit *keyEncipherment* tem que estar ativado para permitir encriptação. O bit *keyAgreement* tem que estar ativado nos certificados *Diffie-Hellman*.

Estrutura da mensagem:

opaque ASN.1Cert<1..2^{24}*-1>;*

struct {

ASN.1Cert certificate_list<0..2^{24}*-1>;*

} Certificate;

certificate_list

Uma sequência (cadeia) de certificados X.509v3. O certificado do emissor tem que ser o primeiro na lista e cada certificado seguinte tem que diretamente autenticar o precedente. Como a validação do certificado requer que as *root keys* sejam distribuídas independentemente, certificados auto-assinados (*self-signed*) que especificam a autoridade do certificado raiz podem opcionalmente ser omitidos da cadeia, sob a suposição que a outra entidade remota já o possui (para possibilitar a validação neste caso).

A mesma estrutura e o mesmo tipo de mensagem será utilizado pelo cliente para responder a uma mensagem *certificate request*. Observe que o cliente pode não enviar o certificado caso não tenha um certificado apropriado para enviar em resposta a requisição de autenticação feita pelo servidor.

3.1.5.3 Server Key Exchange Message

Esta mensagem pode ser enviada imediatamente após a mensagem *Server Certificate* (ou *Server Hello*, caso seja uma negociação anônima). Esta mensagem somente é transmitida quando a mensagem *Server Certificate* (caso transmitida) não contenha os dados suficientes para permitir ao cliente combinar de forma segura a chave *premaster* com o servidor. E esta situação é verdadeira para os seguintes algoritmos para troca de chave:

- DHE_DSS
- DHE_RSA
- DH_anon

Não é válido enviar a mensagem *Server Key Exchange* para os seguintes algoritmos para troca de chave:

- RSA

- DH_DSS
- DH_RSA

Esta mensagem transporta as informações criptográficas que permitem ao cliente combinar de forma segura a chave *premaster*. Sendo uma chave pública RSA para encriptar a chave *premaster* ou uma chave pública *Diffie-Hellman* com a qual o cliente poderá completar a troca de chaves.

Estrutura da mensagem

enum { rsa, diffie_hellman } KeyExchangeAlgorithm;

struct {

opaque rsa_modulus<1..2^{16}-1>;

opaque rsa_exponent<1..2^{16}-1>;

} ServerRSAParams;

rsa_modulus

O módulo da chave RSA temporária do servidor.

rsa_exponent

O expoente público da chave RSA temporária do servidor.

struct {

opaque dh_p<1..2^{16}-1>;

opaque dh_g<1..2^{16}-1>;

opaque dh_Ys<1..2^{16}-1>;

} ServerDHParams; /* Parâmetros *Ephemeral DH* */

dh_p

O módulo primo usado no processamento do *Diffie-Hellman*.

dh_g

O número gerador usado no processamento do *Diffie-Hellman*.

dh_Ys

O valor público *Diffie-Hellman* do servidor (g^X mod p).

```
struct {
    select (KeyExchangeAlgorithm) {
        case diffie_hellman:
            ServerDHParams params;
            Signature signed_params;
        case rsa:
            ServerRSAParams params;
            Signature signed_params;
    };
} ServerKeyExchange;
```

params

Os parâmetros do servidor para a troca de chaves.

signed_params

Um *hash* assinado do campo *params* para as trocas de chave não anônimas.

md5_hash

MD5(*ClientHello.random* + *ServerHello.random* + *ServerParams*);

sha_hash

SHA(*ClientHello.random* + *ServerHello.random* + *ServerParams*);

enum { anonymous, rsa, dsa } SignatureAlgorithm;

```
select (SignatureAlgorithm)    {
    case anonymous: struct { };
    case rsa:
        digitally-signed struct {
            opaque md5_hash[16];
            opaque sha_hash[20];
        };
    case dsa:
        digitally-signed struct {
            opaque sha_hash[20];
        };
} Signature;
```

3.1.5.4 Certificate Request

Um servidor autenticado pode opcionalmente requisitar um certificado do cliente. Esta mensagem, caso transmitida, seguirá imediatamente a mensagem *Server Key Exchange* (ou Server Certificate, caso a *Server Key Exchange* seja desnecessária).

Estrutura da mensagem:

```
enum {
    rsa_sign(1), dss_sign(2), rsa_fixed_dh(3), dss_fixed_dh(4),
    (255)
} ClientCertificateType;

opaque DistinguishedName<1..2^16-1>;

struct {
    ClientCertificateType certificate_types<1..2^8-1>;
    DistinguishedName certificate_authorities<3..2^16-1>;
} CertificateRequest;
```

certificate_types

Uma lista dos tipos de certificados possíveis ordenados de acordo com a preferência do servidor.

certificate_authorities

Uma lista dos *distinguished names* [14] das autoridades certificadoras aceitáveis. Estes *distinguished names* podem especificar a AC raiz desejada ou uma AC subordinada. Assim, esta mensagem pode ser usada para descrever tanto as raízes conhecidas como o espaço de autorização desejado.

Caso um servidor anônimo requisite uma identificação do cliente será gerado um alerta fatal *handshake_failure*.

3.1.5.5 Server Hello Done

A mensagem é transmitida pelo servidor para indicar a finalização da fase de *Hello* e mensagens associadas (todas as informações necessárias para a troca de chaves foram trocadas e o cliente pode prosseguir com a etapa de geração da chave *premaster*).

Depois de transmitir esta mensagem, o servidor fica esperando por uma resposta do cliente. Com a recepção da mensagem *server hello done*, o cliente deveria verificar que o servidor forneceu um certificado válido (se requerido) e checar se os parâmetros do *server hello* são aceitáveis.

Estrutura da mensagem:

struct { } ServerHelloDone;

3.1.5.6 Client Certificate

Esta mensagem é transmitida pelo cliente após o recebimento de uma mensagem *server hello done*. É enviada somente caso o servidor requisite o

certificado do cliente. Se nenhum certificado está disponível, o cliente deve enviar a mensagem *Client Certificate* contendo nenhum certificado. Se a autenticação do cliente é requerida pelo servidor, este pode responder com um alerta fatal *handshake failure*. Os certificados dos clientes são transmitidos usando a estrutura Certificate definida na seção 3.1.5.2.

Quando for utilizada a troca de chaves baseada no *Diffie-Hellman* estático (DH_DSS ou DH_RSA) e caso a autenticação do cliente seja requisitada, os parâmetros *Diffie-Hellman* grupo e gerador (*group/generator*) codificados no certificado do cliente tem que combinar com os parâmetros *Diffie-Hellman* especificados pelo servidor, caso os parâmetros do cliente sejam usados para a troca de chaves.

3.1.5.7 Client Key Exchange Message

O cliente transmite esta mensagem imediatamente após a mensagem *Client Certificate*, caso tenha sido transmitida. Caso contrário, será a primeira mensagem transmitida pelo cliente após receber uma mensagem *Server Hello Done*.

Com esta mensagem a chave *premaster* é estabelecida: (i) ou através de uma transmissão direta do segredo atráves de uma encriptação RSA, (ii) ou pela transmissão dos parâmetros *Diffie-Hellman* que possibilita cada lado gerar um mesmo segredo (chave *premaster*). Quando o método para a troca de chave é DH_RSA ou DH_DSS, a certificação do cliente é requerida e assim ele envia um certificado que contém a chave pública *Diffie-Hellman* cujos parâmetros (*group* e *generator*) combinam com os valores especificados pelo servidor no seu certificado, e neste caso a mensagem não conterá nenhum dado.

Estrutura da mensagem:

O conteúdo da mensagem dependerá do método para troca de chave que foi selecionado.Veja a seção 3.1.5.3 que define a estrutura *KeyExchangeAlgorithm*.

```
struct {
        select (KeyExchangeAlgorithm) {
                case rsa: EncryptedPreMasterSecret;
                case diffie_hellman: ClientDiffieHellmanPublic;
        } exchange_keys;
} ClientKeyExchange;
```

3.1.5.7.1 Estrutura *EncryptedPreMasterSecret*

Caso o algoritmo RSA seja utilizado para autenticação e acordo da chave, o cliente gera uma chave (segredo) premaster de 48 bytes, encripta com a chave pública do servidor (disponível no certificado) ou com uma chave temporária RSA fornecida na mensagem *Server Key Exchange* e transmite o resultado na mensagem *ClientKeyExchange*.

Estrutura da mensagem:

```
struct {
        ProtocolVersion client_version;
        opaque random[46];
} PreMasterSecret;
```

client_version

A versão mais recente suportada pelo cliente. É usado para detectar ataques de *roll-back*. Na recepção desta mensagem, o servidor deve verificar se este valor é o mesmo que foi transmitido pelo cliente na mensagem *Client Hello*.

random

São 46 bytes aleatórios gerados de forma segura.

```
struct {
        public-key-encrypted PreMasterSecret pre_master_secret;
```

} EncryptedPreMasterSecret;

Um ataque descoberto pelo Daniel Bleichenbacher pode ser usado para atacar um servidor TLS/UDP que esteja usando PKCS#1/RSA. O ataque leva vantagem no fato que o servidor pode ser forçado a revelar se uma mensagem particular, quando decriptada, está devidamente formatada pelo padrão PKCS#1. O melhor modo para impedir tal vulnerabilidade é tratar as mensagens formatadas incorretamente como se estivesse correta. E assim quando receber uma mensagem incorreta, o servidor deveria gerar um valor aleatório de 48 bytes e utilizá-lo como *premaster secret*.

pre_master_secret

Um valor aleatório gerado pelo cliente e usado para derivar a chave master (*master secret*).

3.1.5.7.2 Estrutura ClientDiffieHellmanPublic

Esta estrutura transporta o valor público *Diffie-Hellman* do cliente (Yc) caso já não esteja incluído no certificado do cliente. A codificação usada para o Yc é determinada pelo valor *PublicValueEncoding*.

Estrutura da mensagem:

enum { implicit, explicit } PublicValueEncoding;

implicit

Caso o certificado do cliente já contenha a chave *Diffie-Hellman*, então Yc está implícito e não precisa ser transmitida novamente. Neste caso, a mensagem *Client Key Exchange* será transmitida com o conteúdo vazio.

explicit

O valor Yc necessita ser transmitido.

struct {

select (PublicValueEncoding) {

case implicit: struct { };

case explicit: opaque dh_Yc<1..2^{16}-1>;

} dh_public;

} ClientDiffieHellmanPublic;

dh_Yc

O valor público *Diffie-Hellman* do cliente (Yc).

3.1.5.8 Certificate Verify

Esta mensagem é enviada para possibilitar uma explícita verificação do certificado do cliente. Esta mensagem somente é transmitida depois da mensagem *Client Certificate* e caso o certificado tenha capacidade de assinatura (todos os certificados exceto aqueles que contém parâmetros *Diffie-Hellman* estáticos)

Estrutura da mensagem:

struct {

Signature signature;

} CertificateVerify;

A estrutura *Signature* está definida na seção 3.1.5.3.

CertificateVerify.signature.md5_hash

MD5(handshake_messages);

CertificateVerify.signature.sha_hash

SHA(handshake_messages);

A entrada da função de *hash* é *handshake_messages* e refere-se a todas as mensagens transmitidas e recebidas começando pelo *Client Hello* até esta mensagem (exclusive), incluindo os campos *type* e *length* das mensagens de *handshake*. A entrada é composta da concatenação de todas as estruturas de *Handshake*.

3.1.5.9 Finished

Esta mensagem é sempre transmitida imediatamente após a mensagem *Change Cipher Spec* para verificar se todo o procedimento para troca de chave e autenticação foi realizado com sucesso.

A mensagem *finished* é a primeira mensagem protegida com os algoritmos, chaves e segredos negociados anteriormente. Os receptores desta mensagem tem que verificar se o conteúdo desta mensagem está correto. Uma vez que um lado da comunicação enviou a sua mensagem *finished* e recebeu e validou a mensagem *finished* da outra parte, ela pode começar a transmitir e receber os dados da camada de aplicação.

Estrutura da mensagem:

```
struct {
        opaque verify_data[12];
} Finished;
```

verify_data

PRF(master_secret, finished_label, MD5(handshake_messages) + SHA-1(handshake_messages)) [0..11];

finished_label

Para as mensagens *finished* enviadas pelo cliente, a *string* "*client finished*" é utilizada como *label*. Para o servidor, utiliza-se a *string* "*server finished*".

handshake_messages

Todos os dados de todas as mensagens de *Handshake* até esta (exclusive). São todos os dados visíveis na camada *Handshake*. Não inclui os cabeçalhos da camada *Record*. É composta da concatenação de todas as estruturas de *Handshake*.

Caso a mensagem *finished* não seja precedida pela mensagem *Change Cipher Spec* será gerado um alerta de erro fatal.

Este valor *handshake_messages* é diferente do *handshake_messages* utilizado na seção 3.1.5.8. No valor calculado para esta seção está incluída a mensagem *Certificate Verify* (caso seja transmitida). E também, a *string handshake_messages* do servidor será diferente do cliente pois a mensagem *finished* do servidor inclui também a mensagem *finished* do cliente. As mensagens *Hello Request*, *Change Cipher Spec*, alertas e qualquer outro tipo de *record* não são considerados mensagens de *Handshake* e portanto não são incluídas no cálculo do *hash*.

3.1.6 Computação Criptográfica

3.1.6.1 Cálculo da Chave *master secret*

Em todos os métodos para troca de chaves, o mesmo algoritmo é usado para converter a chave *premaster* (*pre_master_secret*) na chave *master* (*master_secret*). A chave *premaster* deve ser apagada da memória quando a chave master for computada.

master_secret = PRF(pre_master_secret, "master secret",
ClientHello.random + ServerHello.random) [0..47];

A chave master tem sempre o tamanho de 48 bytes. O tamanho da chave *premaster* dependerá do método para troca de chave empregado.

3.1.6.1.1 RSA

Quando o algoritmo RSA é utilizado para autenticação do servidor e para troca de chave, o cliente gera um segredo (valor aleatório) de 48 bytes (chave *premaster*), encripta com a chave pública do servidor e transmite o resultado ao servidor. O servidor decripta a chave *premaster* com a sua chave privada. E agora ambas as partes compartilham o segredo (chave *premaster*) e convertem na chave *master* conforme especificado na seção 3.1.6.1.

O algoritmo de chave pública RSA é executado usando PKCS #1 (bloco tipo 2). Na assinatura digital RSA, uma estrutura de 36 bytes de 2 *hashs* (SHA e MD5) é assinada (encriptada com a chave privada) e a codificação é efetuada usando PKCS #1 (bloco tipo 0 ou 1) [14].

3.1.6.1.2 Diffie-Hellman

O cálculo convencional do *Diffie-Hellman* é realizado e a chave gerada (Z) é utilizada como chave *premaster* e depois é convertida na chave *master* conforme especificado na seção 3.1.6.1.

Os parâmetros *Diffie-Hellman* (*dh_p* e *dh_g*) podem ser efêmeros (*ephemeral*) ou obtidos através do certificado do servidor.

3.1.6.2 Cálculo das Chaves Secundárias

O protocolo Record requer um algoritmo para gerar chaves, IVs e o *MAC secrets* a partir dos parâmetros de segurança fornecidos pelo protocolo *Handshake*.

A chave *master* é *hashed* em uma sequência de bytes seguros e deste *hash* são assinalados os *MAC secrets* e as chaves necessárias para possibilitar a comunicação (seção 3.1.6.4). Os conjuntos criptográficos necessitam dos seguintes parâmetros adicionais de segurança: *client write MAC secret*, *server write MAC secret*, *client write key*, *server write key*, *client write IV* e *server write IV*. E estes parâmetros são gerados a partir da chave *master* na ordem mencionada (com exceção dos IVs).

No processo de geração do material criptográfico, a chave *master* é utilizada como uma fonte de entropia e os valores aleatórios (*client_random* e *server_random*) também são utilizados neste processo.

Para derivar todo o material criptográfico é realizada a seguinte computação até que se obtenha todos os bytes necessários.

key_block = PRF(SecurityParameters.master_secret, "key expansion",
SecurityParameters.server_random +
SecurityParameters.client_random);

E o *key_block* é dividido conforme mostrado a seguir:

client_write_MAC_secret[SecurityParameters.hash_size]
server_write_MAC_secret[SecurityParameters.hash_size]
client_write_key[SecurityParameters.key_material_length]
server_write_key[SecurityParameters.key_material_length]

O conjunto criptográfico especificado neste documento o qual requer mais material é o 3DES_EDE_CBC_SHA. Requer 2 chaves x 24 bytes, 2 *MAC secrets* x 20 bytes e 2 IVs x 8 bytes, perfazendo um total de 104 bytes de material.

3.1.6.3 HMAC

Um variado número de operações das camadas *Record* e *Handshake* requerem um *keyed MAC* (*hash* de algum dado protegido por um segredo). A falsificação do *MAC* é impraticável sem o conhecimento do *MAC secret*. A técnica utilizada para esta operação é conhecida como HMAC (Apêndice C).

O *HMAC* pode ser utilizado com uma variedade de algoritmos de hash. O TLS/UDP executa esta operação no *Handshake* com dois diferentes algoritmos: MD5 e SHA-1, designando respectivamente *HMAC_MD5* (*secret, data*) e *HMAC_SHA* (*secret, data*). Algoritmos de hash adicionais podem ser definidos pelos conjuntos

criptográficos e usados para proteger os dados, porém MD5 e SHA-1 estão designados especificamente (*hard coded*) na descrição do processo de *Handshaking* para esta versão do protocolo.

3.1.6.4 PRF

Uma técnica é requerida para realizar a expansão da chave *master* em blocos de dados com o objetivo de geração de chaves e outros materiais utilizados na criptografia (seção 3.1.6.2). Esta função pseudo-aleatória (*PRF - Pseudo-Random Function*) recebe como entrada um segredo, uma semente e um *label* identificador e produz como saída uma quantidade arbitrária de bytes.

Para tornar o *PRF* tão seguro quanto possível, o protocolo TLS/UDP utiliza dois algoritmos de *hash* independentes de modo a garantir sua segurança caso algum dos algoritmos utilizados torne-se inseguro.

Primeiro, define-se uma função de expansão dos dados, P_hash(*secret, data*), a qual usa uma única função de *hash* para expandir o segredo e a semente em uma quantidade arbitrária de bytes de saída:

P_hash(segredo, semente) = HMAC_hash(segredo*, A(1)* + semente) +
HMAC_hash(segredo*, A(2)* + semente) +
HMAC_hash(segredo*, A(3)* + semente) + ...

Onde + indica concatenação.

A(i) é definido como:

A(0) = semente;

A(i) = *HMAC_hash*(segredo, A(i-1))

A função P_hash pode ser repetida (iteração) quantas vezes forem necessárias para produzir a quantidade de bytes necessárias. Por exemplo, se P_SHA-1 está sendo utilizado para derivar 64 bytes de dados, ela deveria ser repetida 4 vezes (até A(4)),

criando 80 bytes de dados. Os últimos 16 bytes da iteração final deveriam ser descartados, restando na saída os 64 bytes de dados.

A função PRF utilizada no protocolo TLS/UDP é criada dividindo o segredo em duas metades e usando cada metade para gerar dados com P_MD5 e a outra metade para gerar dados com P_SHA-1, e então fazendo uma operação XOR das saídas destas duas funções de expansão juntas.

Considere S1 e S2 as duas metades do segredo e ambas tem o mesmo tamanho. S1 é a primeira metade do segredo e S2 a segunda. O tamanho de S1 e S2 é definido pelo arredondamento para mais (*rounding up - ceil*) do tamanho do segredo dividido por dois. Assim se o segredo original tem um número ímpar de bytes, o último byte de S1 será o primeiro byte de S2.

L_S = Tamanho em bytes do segredo;
L_S1 = L_S2 = *ceil*(L_S / 2);

O segredo é dividido em duas metades, sendo S1 os primeiros L_S1 bytes do segredo e S2 os últimos L_S2 bytes do segredo.

A função PRF é definida então como o resultado da combinação de dois *streams* pseudo-aleatórios através da operação XOR

PRF(segredo, *label*, semente) = P_MD5(S1, *label* + semente) XOR
P_SHA-1(S2, *label* + semente);

O *label* é uma *string* ASCII. E deve ser incluída na forma exata que for definida (sem byte de tamanho ou caractere *null*). Por exemplo, o *label* "lab ravel" deveria ser processado através do *hashing* dos seguintes bytes (em hexadecimal):

6C 61 62 20 72 61 76 65 6C

Como o MD5 produz uma saída de 16 bytes e SHA-1 uma de 20 bytes, as fronteiras das iterações internas não estarão alinhadas. Para gerar 80 bytes de saída implica que P_MD5 será executado até a operação A(5) enquanto P_SHA-1 será executado somente até A(4).

3.1.7 Definição das Combinações Criptográficas

Os seguintes valores foram definidos para cada conjunto criptográfico (*CipherSuite*) utilizado nas mensagens *Client Hello* e *Server Hello* nesta versão do protocolo.

O *CipherSuite* TLS_UDP_NULL_WITH_NULL_NULL é especificado como o estado inicial de uma sessão TLS/UDP durante o primeiro *handshake* naquele canal e não pode ser negociado pois não fornece proteção adicional a comunicação.

TLS_UDP_NULL_WITH_NULL_NULL = { 0x00,0x00 };

Os seguintes conjuntos criptográficos requerem que o servidor forneça um certificado RSA que será utilizado para a troca de chaves. O servidor pode requisitar um certificado *signature-capable* DSS ou RSA na mensagem *Certificate Request*.

TLS_UDP_RSA_WITH_NULL_MD5 = { 0x00,0x01 };
TLS_UDP_RSA_WITH_NULL_SHA = { 0x00,0x02 };
TLS_UDP_RSA_WITH_IDEA_CBC_SHA = { 0x00,0x07 };
TLS_UDP_RSA_WITH_DES_CBC_SHA = { 0x00,0x09 };
TLS_UDP_RSA_WITH_3DES_EDE_CBC_SHA = { 0x00,0x0A };

Os seguintes conjuntos criptográficos são utilizados para servidores autenticados (e opcionalmente em clientes autenticados) via *Diffie-Hellman*.

TLS_UDP_DH_DSS_WITH_DES_CBC_SHA = { 0x00,0x0C };
TLS_UDP_DH_DSS_WITH_3DES_EDE_CBC_SHA = { 0x00,0x0D };
TLS_UDP_DH_RSA_WITH_DES_CBC_SHA = { 0x00,0x0F };
TLS_UDP_DH_RSA_WITH_3DES_EDE_CBC_SHA = { 0x00,0x10 };
TLS_UDP_DHE_DSS_WITH_DES_CBC_SHA = { 0x00,0x12 };
TLS_UDP_DHE_DSS_WITH_3DES_EDE_CBC_SHA = { 0x00,0x13 };
TLS_UDP_DHE_RSA_WITH_DES_CBC_SHA = { 0x00,0x15 };
TLS_UDP_DHE_RSA_WITH_3DES_EDE_CBC_SHA = { 0x00,0x16 };

DH indica conjuntos criptográficos os quais o certificado do servidor contém parâmetros *Diffie-Hellman* assinados pela autoridade certificadora (CA). DHE indica *ephemeral Diffie-Hellman*, onde os parâmetros *Diffie-Hellman* são assinados por um certificado DSS ou RSA, o qual foi assinado por uma CA. O algoritmo de assinatura utilizado está especificado após os parâmetros DH ou DHE. O servidor pode requisitar um certificado *signature-capable* do cliente para a devida autenticação do mesmo ou pode requisitar um certificado *Diffie-Hellman*. Todo certificado *Diffie-Hellman* fornecido pelo cliente tem que usar os parâmetros (*group* e *generator*) descritos pelo servidor.

Os seguintes conjuntos criptográficos são usados para comunicações *Diffie-Hellman* completamente anônimas nas quais nenhuma parte é autenticada. Observar que este modo de comunicação é vulnerável ao ataque *man-in-the-middle*.

TLS_UDP_DH_anon_WITH_DES_CBC_SHA = { 0x00,0x1A };
TLS_UDP_DH_anon_WITH_3DES_EDE_CBC_SHA = { 0x00,0x1B };

Todos os conjuntos criptográficos que tenham como primeiro byte 0xFF são considerados privados e podem ser utilizados para algoritmos experimentais ou locais.

Tabela 2 - Conjuntos Criptográficos

CipherSuite	**Troca de Chave**	**Algoritmo Simétrico**	**Hash**
TLS_UDP_NULL_WITH_NULL_NULL	NULL	NULL	NULL
TLS_ UDP_RSA_WITH_NULL_MD5	RSA	NULL	MD5
TLS_ UDP_RSA_WITH_NULL_SHA	RSA	NULL	SHA
TLS_ UDP_RSA_WITH_IDEA_CBC_SHA	RSA	IDEA_CBC	SHA
TLS_ UDP_RSA_WITH_DES_CBC_SHA	RSA	DES_CBC	SHA
TLS_ UDP_RSA_WITH_3DES_EDE_CBC_SHA	RSA	3DES_EDE_CBC	SHA
TLS_ UDP_DH_DSS_WITH_DES_CBC_SHA	DH_DSS	DES_CBC	SHA
TLS_ UDP_DH_RSA_WITH_DES_CBC_SHA	DH_RSA	DES_CBC	SHA
TLS_ UDP_DH_RSA_WITH_3DES_EDE_CBC_SHA	DH_RSA	3DES_EDE_CBC	SHA
TLS_ UDP_DHE_DSS_WITH_DES_CBC_SHA	DHE_DSS	DES_CBC	SHA
TLS_ UDP_DHE_DSS_WITH_3DES_EDE_CBC_SHA	DHE_DSS	3DES_EDE_CBC	SHA
TLS_ UDP_DHE_RSA_WITH_DES_CBC_SHA	DHE_RSA	DES_CBC	SHA
TLS_ UDP_DHE_RSA_WITH_3DES_EDE_CBC_SHA	DHE_RSA	3DES_EDE_CBC	SHA
TLS_ UDP_DH_anon_WITH_DES_CBC_SHA	DH_anon	DES_CBC	SHA
TLS_ UDP_DH_anon_WITH_3DES_EDE_CBC_SHA	DH_anon	3DES_EDE_CBC	SHA

Tabela 3 - Algoritmo para Troca de Chaves e Assinatura

Troca de Chaves	Descrição
DHE_DSS	Ephemeral DH com assinatura DSS
DHE_RSA	Ephemeral DH com assinatura RSA
DH_anon	DH Anônimo sem assinatura
DH_DSS	DH com certificado DSS
DH_RSA	DH com certificado RSA
NULL	Nenhuma troca de chave
RSA	Troca de chaves RSA

Tabela 4 - Características dos Algoritmos Simétricos

Algoritmo Simétrico	Tipo	*Key Material (bytes)*	*Expanded Material Key (bytes)*	*Effective Key Bits (bits)*	*IV size (bytes)*	*Block Size (bytes)*
NULL	N/A	0	0	0	0	N/A
IDEA_CBC	Block	16	16	128	8	8
DES_CBC	Block	8	8	56	8	8
3DES_EDE_CBC	Block	24	24	168	8	8

Tipo

Indica se o algoritmo simétrico é de fluxo ou bloco (modo CBC).

Key Material

O número de bytes do *key_block* que são utilizados para a geração das chaves de escrita (*write keys*).

Expanded Key Material

O número de bytes efetivos utilizados no algoritmo de encriptação.

Effective Key Bits

A quantidade de material entrópico (em bits) que está presente no *Key Material* e sendo utilizado nas rotinas de encriptação.

IV Size

A quantidade de dados (em bytes) que precisa ser gerada para o vetor de inicialização. Sendo zero para os algoritmos de fluxo e igual ao tamanho do bloco (*block size*) para os algoritmos de bloco.

Block Size

A quantidade de dados (em bytes) que um algoritmo de bloco encripta de uma vez. Um algoritmo de bloco em modo CBC pode somente encriptar uma quantidade de bytes múltipla do tamanho do seu bloco (*block size*).

3.1.8 Notas sobre a Implementação

Nesta seção serão fornecidas algumas recomendações e práticas de segurança para ajudar na realização com sucesso de futuras implemetações deste protocolo:

- Compatibilidade:

  Qualquer aplicação compatível com o protocolo TLS/UDP tem que implementar um conjunto criptográfico (*CipherSuite*) para assegurar a interoperabilidade e para esta versão do protocolo foi designado para tal: TLS_UDP_DHE_DSS_WITH_3DES_EDE_CBC_SHA.

- Números Aleatórios e Semente:

  O protocolo TLS/UDP requer um gerador de números pseudo-aleatórios criptograficamente seguro (*Pseudo Random Number Generator - PRNG*). O projeto do *PRNG* e a semente utilizada devem ser cuidadosamente analisados. Os *PRNGs* baseados em operações de *hash* seguras como o MD5 e/ou SHA são aceitáveis, mas não podem fornecer mais segurança que a quantidade de estados possíveis do gerador de

números aleatórios. Por exemplo, *PRNGs* baseados no MD5 normalmente fornecem 128 bits de estado.

- Certificados e Autenticação

  As implementações deste protocolo são responsáveis por verificar a integridade e validade do certificado e devem suportar o entendimento das mensagens de revogação de certificado. Os certificados deveriam sempre serem verificados para assegurar que este foi devidamente assinado por uma Autoridade Certificadora. A seleção e inclusão de ACs confiáveis deve ser efetuada cuidadosamente. Os usuários deveriam estar habilitados a visualizarem as informações sobre o certificado e a AC raiz.

- Conjunto criptográfico

  O protocolo TLS/UDP suporta uma variedade de tamanhos de chaves e níveis diferenciados de segurança, incluindo alguns que fornecem nenhuma ou mínima segurança. Uma implementação adequada provavelmente não suporta muitos conjuntos criptográficos fracos. Por exemplo, uma encriptação de 40 bits é facilmente quebrável, de modo que as implementações não deveriam permitir chaves de 40 bits. Da mesma maneira, a implementação do *Diffie-Hellman* anônimo é fortemente desencorajada porque é vulnerável ao ataque *man-in-the-middle*. As aplicações deveriam também forçar o uso de um tamanho adequado para as chaves. Por exemplo: certificados contendo assinaturas ou chaves RSA de 512 bits não são adequadas para um nível de segurança elevado.

- *Automatic Key Refresh* e *Refresh Time*

  Este valor booleano (*true* ou *false*) que indica se haverá ou não atualização automática do material criptográfico a cada *Refresh Time* segundo. Por *default*, esta *flag* está ativa (*true*), no entanto a aplicação pode desativar esta *flag* caso queira escolher o momento adequado para atualizar o material criptográfico (chaves e *MAC secrets*) via troca de mensagens de *Handshake*. Caso a aplicação deseje administrar a atualização, deve tomar os devidos cuidados para não gerar possíveis furos

de segurança como por exemplo utilizar um algoritmo de chave simétrica com tamanho de chave pequeno e não atualizar o material criptográfico. O valor do *Refresh Time default* de oito horas foi obtido da RFC 2407 (página 12) referente ao tempo de vida de uma SA (*Security Association*) do IPSec mas a aplicação pode alterar este valor sempre que efetuar um novo *Handshake*.

3.1.9 Análise de Segurança

As camadas superiores não devem confiar que o protocolo TLS/UDP negociará um conjunto criptográfico seguro para a comunicação. Há uma variedade de técnicas que um atacante *man in the middle* pode utilizar para que um conjunto criptográfico menos seguro seja utilizado.

A regra fundamental para o estabelecimento de uma comunicação segura é definida quando as camadas mais altas (aplicação notadamente) estão cientes dos requisitos de segurança exigidos e que nunca transmitam informação sobre um canal menos seguro que o mínimo requerido.

Se foi negociado 3DES com uma chave RSA de 1024 bits com um *host* cujo certificado foi verificado, é esperado que a comunicação seja segura. Contudo, uma comunicação com uma chave simétrica de 40 bits não deveria nunca ser realizada a menos que a informação veiculada não seja tão confidencial assim e/ou o custo para quebrar a encriptação não valha o esforço.

Este documento está baseado em algumas suposições relacionadas tradicionalmente com as características de um atacante: (i) tem recursos computacionais fartos, (ii) não pode obter informação privilegiada de fontes fora do canal de comunicação, (iii) tem a habilidade de capturar, modificar, destruir, reproduzir ou qualquer outro tipo de adulteração nas mensagens transmitidas entre as partes oficiais da comunicação.

A seguir, é descrito como o protocolo TLS/UDP foi projetado para resistir a uma variedade de ataques.

3.1.9.1 Handshake

O protocolo *Handshake* é responsável pela seleção do conjunto criptográfico (*CipherSpec*) adequado e pela geração da chave *master*, os quais juntos são os parâmetros primários a serem associados com a sessão segura. O protocolo também pode opcionalmente autenticar ambos os participantes da comunicação que tenham certificados assinados por uma Autoridade Certificadora.

3.1.9.1.1 Autenticação e Troca de Chaves

O protocolo TLS/UDP suporta três modos de autenticação: autenticação do servidor com cliente não autenticado (simples), autenticação de ambas as partes (dupla) e anonimato total (nenhuma). Sempre que o servidor é autenticado, o canal estará protegido ataques *man in the middle* mas as sessões completamente anônimas são inerentementes vulneráveis a tal ataque. Servidores anônimos não podem requisitar a autenticação dos clientes. Se o servidor é autenticado, sua mensagem *Certificate* tem que fornecer uma cadeia de certificados válida que conduza a uma Autoridade Certificadora a qual o cliente tenha confiança. De forma similar, os clientes autenticados têm que fornecer um certificado aceitável para o servidor, caso este seja solicitado. Cada parte é responsável por verificar que o certificado da outra parte está válido e que não esteja expirado ou revogado.

O objetivo geral do processo de troca de chave é gerar uma chave secreta *premaster* conhecida apenas por ambas as partes da comunicação. A chave secreta *premaster* é utilizada para gerar a chave secreta *master*. A chave *master* é necessária para derivar todo o material criptográfico utilizado (chaves simétricas, *MAC secrets* e IVs). Com o envio da mensagem *Finished*, e validada a integridade desta pela outra parte, fica comprovado assim que ambas as partes conhecem a verdadeira chave *premaster* e que não houve adulteração das mensagens de *handshake*.

Para impedir a chave *premaster* de permanecer em memória mais tempo que o necessário, ela deveria ser convertida o mais cedo possível na chave *master*.

3.1.9.1.1.1 Troca de Chaves Anônima

As sessões completamente anônimas podem ser estabelecidas usando tanto o algoritmo para troca de chave RSA como *Diffie-Hellman*.

Com o algoritmo RSA anônimo, o cliente encripta a chave *premaster* com a chave pública do servidor não autenticado extraída da mensagem *Server Key Exchange*. O resultado é enviado na mensagem *Client Key Exchange*. Considerando que os bisbilhoteiros (*eavesdroppers*) não conhecem a chave privada do servidor, será impraticável eles decodificarem a chave *premaster*. Para esta versão do protocolo TLS/UDP nenhum algoritmo RSA anônimo foi definido neste documento.

Com o algoritmo *Diffie-Hellman*, os parâmetros públicos do servidor estão contidos na mensagem *Server Key Exchange* e os do cliente são enviados na mensagem *Client Key Exchange*. Os bisbilhoteiros que não conhecem os valores privados não devem estar habilitados a encontrar o resultado do *Diffie-Hellman* (*chave premaster*).

As conexões completamente anônimas somente fornecem proteção contra espionagem passiva. A menos que um canal independente e a prova de falsificações seja usado para assegurar que as mensagens *Finished* não foram substituídas por um atacante, a autenticação de servidor é necessária em ambientes onde o ataque *man in the middle* ativo seja possível.

3.1.9.1.1.2 Troca de Chave RSA com Autenticação

No algoritmo RSA, a troca de chave e a autenticação do servidor são combinadas. A chave pública pode estar contida no certificado do servidor ou pode ser uma chave RSA temporária enviada na mensagem *Server Key Exchange*. Quando chaves RSA temporárias são usadas, elas são assinadas pelo certificado RSA ou DSS do servidor. A assinatura inclui o atributo *ClientHello.random*, assim as assinaturas e chaves temporárias velhas não podem ser reutilizadas. Os servidores podem usar uma única chave RSA temporária em múltiplas negociações de sessões.

Depois de verificar o certificado do servidor, o cliente encripta a chave *premaster* com a chave pública do servidor. Com a decriptação da chave *premaster*

bem sucedida e com o envio da mensagem *Finished* correta, o servidor demonstra que conhece a chave privada correspondente ao seu próprio certificado e que não houve violação das mensagens de *Handshake*.

Quando o algoritmo RSA é usado para troca de chave, os clientes são autenticados usando a mensagem *Certificate Verify* (ver seção 5.4.8). O cliente assina um valor derivado da chave *master* e todas as mensagens de *Handshake* anteriores. Nas mensagens de *Handshake* estão contidos o certificado do servidor, o qual vincula a assinatura ao servidor, e o atributo *ServerHello.random*, o qual vincula a assinatura ao processo de *Handshake* atual.

3.1.9.1.1.3 Troca de Chave *Diffie-Hellman* com Autenticação

Quando a troca de chaves *Diffie-Hellman* é usada, o servidor pode fornecer um certificado contendo os parâmetros *Diffie-Hellman* fixos ou pode usar a mensagem *Server Key Exchange* para enviar um conjunto efêmero (temporário) de parâmetros *Diffie-Hellman* assinados com um certificado RSA ou DSS.

Estes parâmetros efêmeros antes de serem assinados são concatenados com os valores *ClientHello.random* e *ServerHello.Random* e então é calculado o *hash* do resultado para assegurar que os atacantes não estão reutilizando parâmetros anteriores. Em qualquer caso, o cliente pode verificar o certificado ou a assinatura para assegurar que os parâmetros pertencem ao servidor.

Se o cliente tem um certificado contendo parâmetros *Diffie-Hellman* fixos, seu certificado contém a informação requerida para completar a troca da chave. Observe que neste caso o cliente e servidor gerarão o mesmo resultado *Diffie-Hellman* (*chave premaster*) toda vez eles comunicarem.

Os parâmetros *Diffie-Hellman* do cliente devem ser compatíveis com aqueles fornecidos pelo servidor para a troca de chaves funcionar corretamente.

Caso o cliente tenha enviado um certificado RSA ou DSS (com capacidade de assinatura), ele enviará em seguida também a mensagem *Certificate Verify* para possibilitar a sua autenticação através da verificação explícita do seu certificado.

3.1.9.1.2 Detecção de Ataques Contra o Protocolo *Handshake*

Um atacante poderia tentar influenciar o processo de *Handshake* para fazer as partes selecionarem um algoritmo de encriptação diferente do qual normalmente escolheriam. Como as implementações podem suportar encriptação com chave simétrica de 40 bits e algumas podem até mesmo permitir algoritmos de encriptação ou MAC nulos, este ataque é particularmente preocupante.

Neste ataque, o adversário tem que modificar uma ou mais mensagens de *Handshake*. Se isto acontecer, o cliente e servidor irão computar valores diferentes para os *hashs* das mensagem de *Handshake*. Como resultado, cada parte não aceitará as mensagens *Finished* da outra. E sem a chave *master*, o atacante não pode consertar as mensagens *Finished* advindas de cada lado e assim o ataque será descoberto.

3.1.9.1.3 Reutilização de Sessão

Quando uma comunicação é estabelecida através da retomada de uma sessão, os novos valores de *ClientHello.random* e *ServerHello.random* são utilizados na geração da chave *master* da sessão. Garantindo que a chave *master* não foi comprometida e que as operações de *hash* usadas para produzir as chaves de encriptação e *MAC secrets* são seguras, consequentemente a comunicação deve estar assegurada e efetivamente independente das comunicações prévias. Os atacantes não podem usar conhecidas chaves de encriptação ou *MAC secrets* para descobrir a chave *master* sem quebrar as operações de *hash* (SHA e MD5 simultaneamente).

Não podem ser retomadas sessões a menos que o cliente e servidor concordem. Se qualquer parte suspeita que a sessão possa ter sido comprometida ou que o certificado esteja expirado ou revogado, deve forçar um *Handshake* completo. Um tempo limite máximo de 8 horas é sugerido para a expiração da sessão identificada pelo seu ID pois o atacante que obtém a chave *master* pode, por exemplo, personificar a parte comprometida até que a sessão correspondente seja eliminada. As aplicações que são executadas em ambientes relativamente inseguros não deveriam armazenar os IDs das sessões em dispositivos de armazenamento compartilhados.

3.1.9.1.4 MD5 e SHA

O protocolo TLS/UDP utiliza as funções de *hash* de forma conservadora. Onde possível, o MD5 e SHA são usados em conjunto para assegurar que falhas em algum dos algoritmos não quebrem totalmente o protocolo.

3.1.9.2 Proteção dos Dados da Aplicação

A chave master é *hashed* com as variáveis *ClientHello.random* e *ServerHello.random* para produzir de forma exclusiva o material criptográfico (chaves simétricas e *MAC secrets*) utilizado em cada comunicação.

Os dados de saída são protegidos com um MAC antes da transmissão. Para impedir ataques por *reprodução* ou de modificação das mensagens, o MAC é computado a partir do *MAC secret*, do número de sequência, do tamanho da mensagem, do conteúdo da mensagem e de duas *strings* de caracteres fixos. O campo de tipo (*type*) da mensagem é necessário para assegurar que as mensagens intencionadas para uma camada *Record* TLS/UDP cliente não sejam redirecionadas a um outro cliente. Os números de sequência asseguram que as tentativas de reprodução, apagamento ou reordenação das mensagens serão descobertas. O número de sequência tem 64 bits de tamanho, e na prática, nunca será alcançado o seu limite.

Se um atacante quebrar a chave de encriptação simétrica, todas as mensagens encriptadas com ela poderão ser lidas. O comprometimento da chave MAC pode tornar o ataque de modificação das mensagens possível. Como os MACs também são encriptados, os ataques de modificação das mensagens (*message-alteration*) geralmente necessitam quebrar o algoritmo de encriptação também.

Os *MAC secrets* podem ser maiores que as chaves de encriptação de modo que as mensagens podem permanecer resistentes as falsificações até mesmo quando as chaves de encriptação forem quebradas.

3.1.9.3 Notas finais

Para o protocolo TLS/UDP disponibilizar uma comunicação segura é necessário que o sistema final cliente e servidor, as chaves e as aplicações sejam seguras. E além disso, as implementações do protocolo devem estar livres de erros de segurança gerados principalmente na modelagem e codificação em uma linguagem de programação qualquer. Exemplo de uma falha de segurança comum inserida no momento da codificação em uma linguagem de programação: *buffer overflow*.

As chaves públicas e simétricas de pequeno tamanho e servidores anônimos devem ser usados com grande precaução. As implementações do protocolo e os usuários da aplicação devem ter cuidado quando decidir quais certificados e Autoridades Certificadoras são válidas. Uma Autoridade Certificadora desonesta pode trazer um tremendo prejuízo a segurança da comunicação.

Quando receber a mensagem *HelloRequest* em texto puro, o cliente TLS/UDP tem que implementar um mecanismo de proteção. A mensagem *ClientHello* não pode ser transmitida sem uma verificação da validade do endereço IP do transmissor. Os clientes deveriam ignorar mensagens *HelloRequest* caso sejam recebidas muito frequentemente de um mesmo endereço.

3.2 Implementação da API Referência

O objetivo desta seção é descrever a interface de programação API implementada em Java do novo protocolo TLS/UDP desenvolvida neste trabalho. No processo de criação desta interface foi utilizada o código fonte do protocolo TLS/TCP integrado ao Java 2 JDK. A meta principal foi especificar e implementar uma API similar as já existentes para o protocolo TLS/TCP como a JSSE para Java (descrita na próxima seção) e a OPENSSL para a linguagem de programação C.

As seguintes razões motivaram o desenvolvimento desta implementação na linguagem de programação Java: código fonte aberto do TLS/TCP do JDK/Sun possibilitando a modificação do código original, a portabilidade automática para outras plataformas, a API bem documentada, a orientação a objetos que facilita a reutilização de código e o prévio conhecimento da linguagem. E um dos principais motivos do descarte da linguagem de programação C deve-se ao fato do OPENSSL (melhor implementação nesta linguagem) disponibilizar uma API com uma documentação incompleta.

Foram utilizados no processo de codificação do protocolo na linguagem de programação Java, a versão binária e os fontes do JDK 1.4 (*build* 1.4.1_01-b01).

3.2.1 TLS/TCP no Java 2 SDK

O *Java Secure Socket Extension* - JSSE [15] disponibiliza uma implementação para a linguagem Java do protocolo SSL e TLS. Os desenvolvedores podem incluir segurança na comunicação para qualquer aplicação cliente/servidor TCP/IP, como HTTP, Telnet e FTP através do uso desta biblioteca.

Com a abstração dos algoritmos de criptografia e do processo de *handshaking*, o JSSE minimiza o risco da criação de vulnerabilidades na segurança. Além disso, ele simplifica o desenvolvimento das aplicações através da disponibilização dos blocos construtores os quais os desenvolvedores podem diretamente integrar em suas aplicações.

Anteriormente o JSSE era um pacote opcional (*standard extension*) ao Java 2 SDK, Standard Edition versões 1.2 e 1.3. E agora ele foi integrado a versão 1.4 (atual) do JDK.

O JSSE fornece uma interface de programação com o usuário e também uma implementação desta API. Ela suplementa os serviços de criptografia definidos no Java 2 JDK, versão 1.4 para os pacotes *java.security* e *java.net* fornecendo uma classe de *sockets* extendida, gerenciadores de confiança (*trust managers*), gerenciadores de chave, contextos SSL e uma fábrica de *sockets* para encapsular perfis de criação. Fornece também uma limitada API para gerenciamento de certificados de chave pública (*javax.security.cert*) utilizada apenas para compatibilidade com o JSSE 1.0.x e atualmente quando necessitar dos recursos, utilize o pacote padrão *java.security.cert*.

A implementação JSSE na versão 1.4 do JDK suporta o protocolo SSL 3.0 (não implementa a versão 2.0) e o TLS 1.0. A interface de programação com o usuário do JSSE, pacotes *javax.net*, *javax.net.ssl* e *javax.security.cert*, disponibiliza:

- *Sockets* simples e *sockets* servidores (*server sockets*) seguros (SSL/TLS);

- Fábrica (*factories*) para a criação de *sockets* simples e servidores comuns e seguros. Usando esta facilidade pode-se encapsular a criação e configuração dos *sockets*;

- Uma classe representando o contexto de um *socket* seguro que pode atuar como uma *factory* para a criação de *sockets* seguros;

- Interfaces para gerenciamento de chaves e confiança e *factories* para criação deles;

- Uma classe para conexões HTTP seguras;

- Uma API para gereciamento de certificados de chave pública. Os certificados X509 podem ser gerenciados pela ferramenta *keytool* disponibilizada junto com o JDK.

Os seguintes conjuntos criptográficos (*Cipher Suite*) são suportados pela atual versão do JSSE:

SSL_RSA_WITH_RC4_128_SHA
SSL_RSA_WITH_RC4_128_MD5
SSL_RSA_WITH_DES_CBC_SHA
SSL_RSA_WITH_3DES_EDE_CBC_SHA
SSL_DHE_DSS_WITH_DES_CBC_SHA
SSL_DHE_DSS_WITH_3DES_EDE_CBC_SHA
SSL_RSA_EXPORT_WITH_RC4_40_MD5
SSL_DHE_DSS_EXPORT_WITH_DES40_CBC_SHA
SSL_RSA_WITH_NULL_MD5
SSL_RSA_WITH_NULL_SHA
SSL_DH_anon_WITH_RC4_128_MD5
SSL_DH_anon_WITH_DES_CBC_SHA
SSL_DH_anon_WITH_3DES_EDE_CBC_SHA
SSL_DH_anon_EXPORT_WITH_RC4_40_MD5
SSL_DH_anon_EXPORT_WITH_DES40_CBC_SHA

E os seguintes conjuntos criptográficos estão habilitados por *default*, listados na ordem de preferência:

SSL_RSA_WITH_RC4_128_SHA
SSL_RSA_WITH_RC4_128_MD5
SSL_RSA_WITH_DES_CBC_SHA
SSL_RSA_WITH_3DES_EDE_CBC_SHA
SSL_DHE_DSS_WITH_DES_CBC_SHA
SSL_DHE_DSS_WITH_3DES_EDE_CBC_SHA
SSL_RSA_EXPORT_WITH_RC4_40_MD5
SSL_DHE_DSS_EXPORT_WITH_DES40_CBC_SHA

Administração das chaves e de confiança

Os gerenciadores de chaves (*Key managers*) e de confiança (*Trust managers*) usam *keystores* para o armazenamento das chaves e afins. Um *key manager* gerencia um *keystore* e fornece chave públicas para utilizar na autenticação dos usuários. Um *trust manager* faz decisões sobre quem confiar baseado em uma *truststore* a qual administra. Estes gerenciadores são abstrações utilizadas no JDK para execução das funcionalidades mencionadas acima.

Keystores e *Truststores*

Um *keystore* é uma base de dados para as chaves. É usado por uma variedade de funções, incluindo autenticação e integridade dos dados. Existe vários tipos de *keystores* disponíveis, incluindo PKCS12 e JKS (Sun).

De modo geral, a informação da *keystore* pode ser agrupada em duas diferentes categorias: entrada de chaves e a entrada dos certificados confiados. A entrada de chaves consiste da identidade da entidade e a sua respectiva chave privada. E a entrada dos certificados confiados contém somente a identidade e a sua respectiva chave pública. Na implementação JDK do JKS, uma *keystore* pode conter ambas as entradas (chaves e certificados).

Uma *truststore* é uma *keystore* usada quando estiver sendo executadas decisões sobre quem confiar. Se você recebeu algum dado de uma entidade que já confia, e se você verificou que a entidade é de quem reclama ser, então pode assumir que os dados vieram realmente daquela entidade.

Uma entrada somente deveria ser adicionada a *truststore* se o usuário realmente fez uma decisão de confiar naquela entidade. Através da geração do par de chaves ou pela importação de um certificado, o usuário passa a confiar naquela entidade e assim a entrada na *keystore* é considerada uma entrada confiável.

Pode ser útil ter dois diferentes arquivos de *keystore*: uma somente contendo as entradas de chaves e uma outra contendo as entradas dos certificados confiados, incluindo os certificados das CAs. A primeira entrada contém a informação privada enquanto a última não. Usando dois diferentes arquivos ao invés de um único arquivo de *keystore* fornece uma separação lógica mais clara entre seus próprios certificados

(e as correspondentes chaves privadas) e os certificados dos outros. E assim é possível fornecer mais proteção as chaves privadas aplicando restrição de acesso a esta *keystore* específica. E o arquivo com os certificados confiáveis pode ter permissão de acesso mais liberal, se necessário.

Relacionamento entre as classes

Para a comunicação ser segura, ambos os lados têm que estar habilitados para tal (*SSL enabled*). Na API JSSE, a classe *endpoint* da comunicação é a SSLSocket. No diagrama abaixo (figura 17), as classes principais usadas para criar os *SSLSockets* estão desenhadas conforme a lógica de funcionamento.

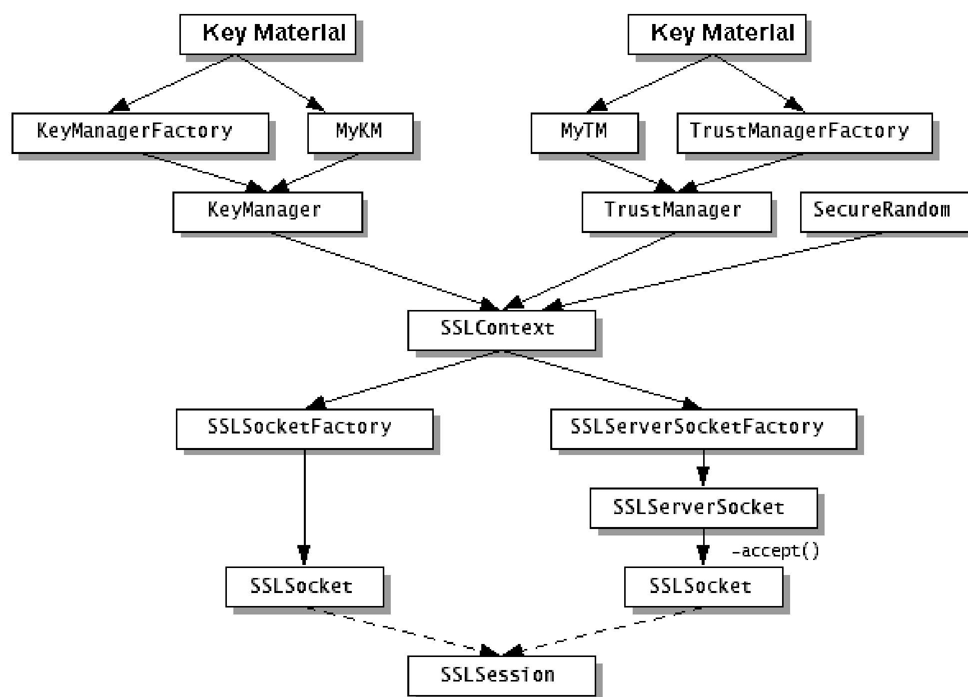


Figura 17 - Relacionamento das principais classes JSSE

Um *SSLSocket* é criado através de um *SSLSocketFactory* ou quando um *SSLServerSocket* aceita um pedido de conexão de entrada. E por sua vez, um *SSLServerSocket* é criado por um *SSLServerSocketFactory*. Tanto o objeto *SSLSocketFactory* como o *SSLServerSocketFactory* são criados por um SSLContext.

Há dois modos para criar e inicializar um *SSLContext*:

- O modo mais simples é através da chamada do método estático getDefault da classe *SSLSocketFactory* ou *SSLServerSocketFactory*. Estes métodos criam um *SSLContext default* com um *default KeyManager*, *TrustManager*

e um gerador de números aleatórios seguro. As chaves e afins estão armazenados no *default keystore/truststore* como determinado pelas propriedades do sistema (*system properties*).

- O modo que fornece maior controle sobre o comportamento do contexto criado é através do método estático *getInstance* da classe *SSLContext* e então inicializar o contexto chamando o método *init*. O método *init* recebe três argumentos: um *array* de objetos *KeyManager*, um *array* de objetos *TrustManager* e um gerador de números aleatórios seguro. Os objetos *KeyManager* e *TrustManager* são criados pela implementação das interfaces apropriadas ou usando as classes *KeyManagerFactory* e *TrustManagerFactory*. E estas *factories* podem ser inicializadas com o material contido na *KeyStore* passado como argumento para o método *init* das classes *TrustManagerFactory* e *KeyManagerFactory*.

Logo que a conexão SSL é estabelecida, uma sessão SSL (*SSLSession*) é criada, a qual contém várias informações tal como: a identidade das partes, *cipher suite* selecionado, dentre outras. A *SSLSession* é então utilizada para descrever o andamento do relacionamento e a informação do estado entre as duas entidades. Cada conexão SSL ocupa uma sessão por vez e por sua vez, esta sessão pode ser usada em inúmeras conexões entre as entidades, simultaneamente ou sequencialmente.

Convertendo um *socket* inseguro em seguro

A seguir, será mostrado como o JSSE pode ser usado para tornar segura uma comunicação com os *sockets* tradicionais. O código fonte abaixo foi extraído do livro *Java 2 Network Security*, de Marco Pistoia et al [16].

O primeiro exemplo "*Socket* sem SSL" lista uma amostra de código que pode ser usada para estabelecer uma comunicação entre o cliente (item b) e o servidor (item a) usando os *sockets* tradicionais (inseguros). E depois este código é modificado em "*Socket* com SSL" para estabelecer uma comunicação segura usando o JSSE.

1- *Socket* sem SSL

(a) Trecho de código da **aplicação servidora** sem SSL:

```
import java.io.*;
import java.net.*;

int port = availablePortNumber;
ServerSocket s;

try {
   s = new ServerSocket(port);
   Socket c = s.accept();
   OutputStream out = c.getOutputStream();
   InputStream in = c.getInputStream();
   // Send messages to the client through the OutputStream
   // Receive messages from the client through the InputStream
}
catch (IOException e) {
}
```

(b) Trecho de código da **aplicação cliente** sem SSL:

```
import java.io.*;
import java.net.*;

int port = availablePortNumber;
String host = "hostname";

try {
   s = new Socket(host, port);
   OutputStream out = s.getOutputStream();
   InputStream in = s.getInputStream();
   // Send messages to the server through the OutputStream
   // Receive messages from the server through the InputStream
}
catch (IOException e) {
}
```

2 - *Socket* com SSL

(a) Trecho de código da **aplicação servidora** com SSL. Observar que as diferenças deste código com SSL (seguro) para o outro (inseguro) estão destacadas em negrito:

```
import java.io.*;
import javax.net.ssl.*;

int port = availablePortNumber;
SSLServerSocket s;

try {
   SSLServerSocketFactory sslSrvFact =
      (SSLServerSocketFactory) SSLServerSocketFactory.getDefault();
   s =(SSLServerSocket)sslSrvFact.createServerSocket(port);
   SSLSocket c = (SSLSocket)s.accept();

   OutputStream out = c.getOutputStream();
   InputStream in = c.getInputStream();
   // Send messages to the client through the OutputStream
   // Receive messages from the client through the InputStream
}
catch (IOException e) {
}
```

(b) Trecho de código da **aplicação cliente** com SSL. Observar que as diferenças deste código com SSL (seguro) para o outro (inseguro) estão destacadas em negrito:

```
import java.io.*;
import javax.net.ssl.*;

int port = availablePortNumber;
String host = "hostname";

try {
   SSLSocketFactory sslFact =
     (SSLSocketFactory)SSLSocketFactory.getDefault();
   SSLSocket s = (SSLSocket)sslFact.createSocket(host, port);

   OutputStream out = s.getOutputStream();
   InputStream in = s.getInputStream();
   // Send messages to the server through the OutputStream
   // Receive messages from the server through the InputStream
}
catch (IOException e) {
}
```

3.2.2 Interface de Programação para o TLS/UDP

Nesta seção será apresentada a API do novo protocolo TLS/UDP com a descrição dos novos pacotes, classes, métodos e atributos que foram criados ou modificados em relação ao JSSE. E para diferenciar a implementação do protocolo TLS/TCP do JDK/Sun (JSSE) da implementação do novo protocolo TLS/UDP, foi definido um nome para esta nova API denominada "*Java Secure DatagramSocket Extension*" (JSDE).

Para facilitar a criação dos pacotes do JSDE foi reutilizado todo o código fonte das classes incluídas nos pacotes originais do JSSE (javax.net.ssl, javax.net.cert e javax.net) embora nem todas as classes tenham sido alteradas e/ou utilizadas.

No processo de modificação do código originalmente desenvolvido para o protocolo TCP para o UDP foi utilizada a nova biblioteca Java NIO incluída somente a partir da atual versão 1.4 do Java. Esta API (pacotes java.nio.*) fornece aos desenvolvedores diversos mecanismos inteligentes usados no tratamento de arquivos e *sockets*. *Non-blocking I/O* (NIO) é muito interessante porque torna a aplicação muito mais escalável, portável e simples de programar. Até pouco tempo atrás, os programadores Java tinham que tratar múltiplas conexões para um *socket* através da inicialização de uma *thread* para cada conexão. E inevitavelmente encontravam problemas tais como a limitação do sistema operacional, *deadlocks* ou outras violações de segurança relacionadas as *threads*. Agora, o desenvolvedor pode usar seletores para administrar múltiplas conexões simultâneas para um *socket* em uma única *thread*.

Na distribuição binária do JDK está incluído o código licenciado da empresa RSA Data Security para o algoritmo de chave pública RSA. Não foi disponibilizado na distribuição do código fonte do JSSE a parte de código licenciado da RSA. Houve a tentativa de obter a licença de uso para fins de pesquisa deste trabalho mas não foi obtido êxito no contato com a empresa. E por este motivo não foi implementado nesta versão do código TLS/UDP este algoritmo de criptografia. Atráves de pesquisas na Internet, foi verificado que existem outras versões *open source* do algoritmo RSA que poderão ser utilizadas nas futuras implementações do novo protocolo TLS/UDP.

E assim, os seguintes conjuntos criptográficos (*Cipher Suite*) são suportados pela atual versão do JSDE:

TLS_UDP_DHE_DSS_WITH_DES_CBC_SHA
TLS_UDP_DHE_DSS_WITH_3DES_EDE_CBC_SHA
TLS_UDP_DH_anon_WITH_DES_CBC_SHA
TLS_UDP_DH_anon_WITH_3DES_EDE_CBC_SHA

E os seguintes conjuntos criptográficos estão habilitados por *default*, listados na ordem de preferência:

TLS_UDP_DHE_DSS_WITH_3DES_EDE_CBC_SHA
TLS_UDP_DHE_DSS_WITH_DES_CBC_SHA

Os pacotes do JSDE são similares aos JSSE e a descrição detalhada seria muito extensa e fora de contexto. Desta forma, a seguir foi realizada uma descrição resumida dos pacotes JSDE e das respectivas classes, interfaces e exceções (e para maiores informações, favor consultar [15]).

Pacote br.ufrj.ravel.net.tlsudp (corresponde ao pacote javax.net.ssl):

Fornece as classes necessárias para a criação e manipulação dos datagramas *sockets* seguros através do TLS/UDP.

1- Sumário das interfaces:

HandshakeCompletedListener: Esta interface é implementada por qualquer classe que queira receber notificações sobre a finalização do processo de *handshake* do protocolo TLS/UDP em um dado canal.

HostnameVerifier: Esta classe é a interface base para verificação do *hostname* do servidor.

KeyManager: Esta é a interface base para gerenciadores das chaves JSDE.

ManagerFactoryParameters: Esta classe é a interface base para o fornecimento de informações relacionadas ao algoritmo para uma KeyManagerFactory ou TrustManagerFactory.

TLSUDPSession: No TLS/UDP, as sessões são usadas para descrever um relacionamento em andamento entre duas entidades.

TLSUDPSessionBindingListener: Esta interface é implementada pelos objetos que desejam conhecer quando estão ligados ou não a uma TLSUDPSession.

TLSUDPSessionContext: Esta classe representa um conjunto de TLSUDPSessions associadas a uma entidade.

TrustManager: Esta classe é a interface base para os gerenciadores de confiança JSDE.

X509KeyManager: Uma instância desta interface administra pares de chaves baseadas nos certificados X509, as quais são usadas para autenticar a entidade local do datagrama *socket* seguro.

X509TrustManager: Uma instância desta interface administra os certificados X509 que podem ser usados para autenticar a entidade remota do datagrama *socket* seguro.

2- Sumário das classes:

HandshakeCompletedEvent: Este evento indica que um *handshake* TLS completou para uma dada comunicação segura.

KeyManagerFactory: Esta classe atua como uma *factory* para administradores de chaves baseados em uma fonte para o material da chave.

KeyManagerFactorySpi: Esta classe define um *Service Provider Interface* (SPI) para a classe KeyManagerFactory.

TLSUDPContext: Uma instância desta classe representa uma implementação do protocolo TLS/UDP a qual atua como uma *factory* para a *factory* de datagramas *sockets* seguros.

TLSUDPContextSpi: Esta classe define um *Service Provider Interface* (SPI) para a classe TLSUDPContext.

TLSUDPPermission: Esta classe define algumas permissões de rede.

TLSServerDatagramSocket: Esta classe estende a classe *DatagramSocket* e fornece um Datagrama *Socket* seguro através do uso do protocolo TLS/UDP.

TLSServerDatagramSocketFactory: Esta classe tem como objetivo instanciar TLSServerDatagramSockets.

TLSUDPSessionBindingEvent: Este evento é propagado para a classe TLSUDPSessionBindingListener.

TLSDatagramSocket: Esta classe estende a classe DatagramSocket e disponibiliza Datagramas *Sockets* seguros através do uso do protocolo TLS/UDP.

TLSDatagramSocketFactory: Esta classe tem como objetivo criar TLSDatagramSockets.

TrustManagerFactory: Esta classe atua como uma factory para gerenciadores de confiança baseados em uma fonte de material confiável.

TrustManagerFactorySpi: Esta classe define um *Service Provider Interface* (SPI) para a classe TrustManagerFactory.

3- Sumário das exceções:

TLSUDPException: Indica que algum tipo de erro foi detectado.

TLSUDPHandshakeException: Indica que o cliente e o servidor não conseguiram negociar o nível desejado de segurança.

TLSUDPKeyException: Informa o uso de uma chave inválida.

TLSUDPPeerUnverifiedException: Indica que a identidade do *host* não foi verificada (sem autenticação).

TLSUDPProtocolException: Informa um erro na operação do protocolo TLS/UDP.

Pacote br.ufrj.ravel.net (corresponde ao pacote javax.net)

Fornece as classes utilizadas nas aplicações de rede. Estas classes incluem *factories* para a criação dos datagramas *sockets*. Usando as *factories* é possível encapsular a criação e o comportamento dos datagramas *sockets*.

1- Sumário das classes:

ServerDatagramSocketFactory: Esta classe cria datagramas *sockets* servidores.
DatagramSocketFactory: Esta classe cria datagramas *sockets*.

Pacote br.ufrj.ravel.security.cert (corresponde ao pacote javax.security.cert)

Fornece as classes utilizadas para os certificados de chave pública. E estas classes são uma versão simplificada do pacote java.security.cert para uso do JSDE.

1- Sumário das classes:

Certificate: Classe abstrata para gerenciamento de uma variedade de certificados de identidade.
X509Certificate: Classe abstrata para o tratamento dos certificados X.509v1.

2- Sumário das exceções:

CertificateEncodingException: Indica um erro na codificação do certificado.
CertificateException: Indica uma variedade de problemas com o certificado.
CertificateExpiredException: Indica que o certificado expirou.
CertificateNotYetValidException: Indica que o certificado não está válido ainda.
CertificateParsingException: Indica que houve algum erro na decodificação (*parsing*) do certificado.

Instruções para uso da API:

Para utilizar as facilidades deste novo protocolo é necessário proceder com as seguintes tarefas:

1- Instalar o Java 2 JDK versão 1.4 para a plataforma correspondente.
2- Instalar o arquivo jsde.jar no diretório <java-dir>\jre\lib\.
3- Adicionar a seguinte linha ao arquivo java.security localizado no diretório <java-dir>\jre\lib\security:

   security.provider.6= br.ufrj.ravel.net.tlsudp.internal.tlsudp.Provider

4- Compilar o código que utiliza o TLS/UDP.
5- Criar *keystores* e *truststores* para possibilitar a autenticação.
6- Executar a aplicação.

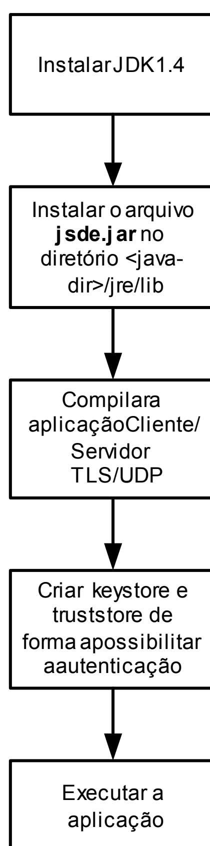


Figura 18 - Instruções de uso da API TLS/UDP

Convertendo um datagrama *socket* inseguro em seguro.

A seguir, será mostrado como o JSDE pode ser usado para tornar segura uma comunicação com os datagramas *sockets* tradicionais.

O primeiro exemplo "Datagrama *Socket* sem TLS/UDP" lista uma amostra de código que pode ser usada para estabelecer uma comunicação entre o cliente (item b) e o servidor (item a) usando os datagramas *sockets* tradicionais (inseguros). E depois este código é modificado em "Datagrama *Socket* com TLS/UDP" para estabelecer uma comunicação segura usando o JSDE.

1- Datagrama *Socket* sem TLS/UDP

(a) Trecho de código da **aplicação servidora** sem TLS/UDP:

```
import java.net.*;

int port = availablePortNumber;

try {
        DatagramSocket sds = new DatagramSocket(port);
        sds.setReuseAddress(true);

        byte[] buffer = new byte[32768];
        DatagramPacket packet = new DatagramPacket(buffer, buffer.length);
        sds.receive(packet);
        int len = packet.getLength();

} catch (Exception e) {
        System.out.println("Exception" + e);
 }
```

(b) Trecho de código da **aplicação cliente** sem TLS/UDP:

```
import java.net.*;

int port = availablePortNumber;
String host = “hostname”;

try {
            InetAddress ad = InetAddress.getByName(host);
            SocketAddress sa = new InetSocketAddress(ad, port);
            DatagramSocket ds = new DatagramSocket(null);
            ds.connect(sa);

            String frase = "Esta mensagem foi enviada pelo CLIENTE ";
            byte[] msg;
            msg = frase.getBytes();

            DatagramPacket packet = new DatagramPacket(msg, msg.length, host,
port);
            ds.send(packet);

} catch (Exception e) {
           System.out.println("Exception " + e);
}
```

2- Datagrama *Socket* com TLS/UDP

(a) Trecho de código da **aplicação servidora** com TLS/UDP. Observar que as diferenças deste código com TLS/UDP (seguro) para o outro (inseguro) estão destacadas em negrito:

```
import br.ufrj.ravel.net.tlsudp.*;

int port = availablePortNumber;

try {
        Security.addProvider(
                new br.ufrj.ravel.net.tlsudp.internal.tlsudp.Provider());
        TLSServerDatagramSocketFactory sslSrvFact =
                (TLSServerDatagramSocketFactory)
                TLSServerDatagramSocketFactory.getDefault();
        sds =(TLSServerDatagramSocket)sslSrvFact.createServerSocket(port);

        TLSDatagramSocket in = (TLSDatagramSocket)sds.accept();
        //  não será necessário na próxima versão

        sds.setReuseAddress(true);

        byte[] buffer = new byte[32768];
        DatagramPacket packet = new DatagramPacket(buffer, buffer.length);
        in.receive(packet);
        int len = packet.getLength();

} catch (Exception e) {
        System.out.println("Exception" + e);
 }
```

(b) Trecho de código da **aplicação cliente** com TLS/UDP. Observar que as diferenças deste código com TLS/UDP (seguro) para o outro (inseguro) estão destacadas em negrito:

```
import br.ufrj.ravel.net.tlsudp.*;

int port = availablePortNumber;
String host = “hostname”;

try {
        Security.addProvider(
                new br.ufrj.ravel.net.tlsudp.internal.tlsudp.Provider());
        TLSDatagramSocketFactory tlsudpFact =
        (TLSDatagramSocketFactory) TLSDatagramSocketFactory.getDefault();

        TLSDatagramSocket ds =
        (TLSDatagramSocket)tlsudp.createSocket();

        InetAddress ad = InetAddress.getByName(host);
        SocketAddress sa = new InetSocketAddress(ad, port);
        ds.connect(sa);

        String frase = "Esta mensagem foi enviada pelo CLIENTE ";
        byte[] msg;
        msg = frase.getBytes();

        DatagramPacket packet = new DatagramPacket(msg, msg.length, host,
port);
        ds.send(packet);

} catch (Exception e) {
        System.out.println("Exception " + e);
}
```

Capítulo 4 - Estudo Comparativo

Neste capítulo será explicada a infra-estrutura de teste empregada para validar a funcionalidade do novo protocolo de segurança de rede proposto TLS/UDP. Um ambiente de homologação foi criado para viabilizar o estudo e comparação do comportamento de aplicações cliente/servidor com as seguintes opções de segurança: (i) sem criptografia, (ii) com IPSec ou (iii) com TLS, e todas estas opções foram testadas no protocolo de transporte UDP e TCP. E para tal, neste capítulo foram detalhadas as aplicações, os algoritmos de criptografia utilizados em cada um dos protocolos de segurança de rede analisados e o ambiente operacional de teste.

4.1 Ambiente Operacional

Para executar as aplicações e realizar os testes desejados foi montado um ambiente operacional de rede com foram definidos os recursos necessários de infra-estrutura computacional adequados ao objetivo proposto.

Sendo assim, para viabilizar os testes foram utilizados os seguintes recursos:

- Computador AMD Duron 1GHz, 256MB RAM e HD 40GB IDE
- Computador AMD K6 II 500MHz, 128MB RAM e HD 20GB IDE
- Duas placas de rede 10BASE-T/100BASE-TX e um cabo *crossover*
- Sistema Operacional Windows 2000 Professional com *Service Pack* 4.

O diagrama simplificado de rede utilizado para descrever a interconexão entre a máquina cliente e a servidora pode ser visualizado na figura 19.

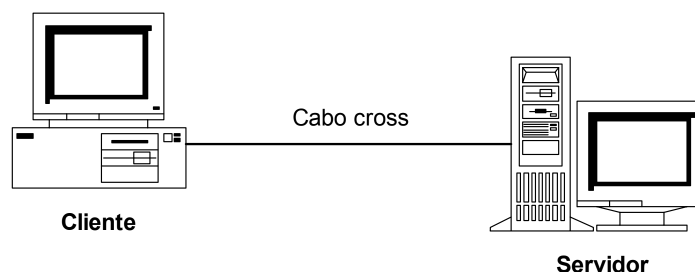


Figura 19 - Ambiente operacional de teste

4.2 Aplicação Cliente/Servidor

As ferramentas tradicionais *open source* de *benchmarking* normalmente utilizadas para medida do desempenho de rede são NetPerf, NetSpec e TTCP Utility. E os tipos de medidas comumente avaliadas em uma comunicação UDP/IP são:

- *Request/Response*: Este tipo de medida normalmente é cotada em 'transações por segundo' para um dado tipo de requisição e resposta. Uma transação abrange o envio de uma requisição simples e a sua respectiva resposta. A partir desta taxa é possível deduzir a latência média de ida e volta (*round-trip average latency*).

- *Throughput*: Este tipo de medida tem como objetivo a obtenção da taxa a qual um sistema pode transmitir (receber) dados para (de) um outro sistema sem erro. Às vezes, referenciada como a medida do desempenho de fluxo unidirecional. Normalmente cotado em 'bytes por segundo'.

Como todas estas ferramentas *open source* foram desenvolvidas na linguagem de programação C/C++ não foi possível utilizá-las na avaliação do protocolo TLS/UDP pois a única versão implementada deste novo protocolo foi codificada na linguagem Java. Desta forma, para avaliação e comparação dos protocolos TLS/UDP e IPSec foram desenvolvidos dois tipos diferentes de aplicação cliente/servidor em Java. E cada uma das aplicações foram testadas sob três modos de operação: sem criptografia nenhuma (figura 20a), com o protocolo TLS (figura 20b) e com o IPSec (figura 20c) para ambos protocolos da camada de transporte, UDP e TCP.

Foi realizada a avaliação entre o protocolo TLS e o IPSec observando que ambos os protocolos usaram conjuntos criptográficos similares para possibilitar uma comparação mais justa (embora tenha inúmeros fatores que não contribuam para tal, citados na análise dos resultados obtidos).

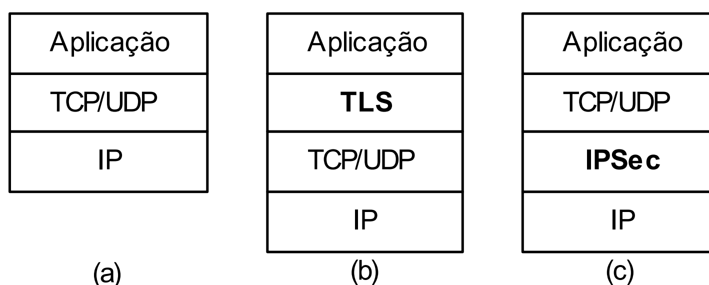


Figura 20 - Tipo de Segurança

A primeira aplicação visa a obtenção da latência, em milisegundos, medida a partir do instante que o cliente enviou uma requisição em texto puro ao servidor até o momento do recebimento da resposta em texto puro pela camada de aplicação (*request/response*). Desta forma, será verificado principalmente a influência do processo de *handshake* do TLS e do mecanismo de troca de chaves IKE (e negociação das SAs) do IPSec. No caso específico, tanto a requisição como a resposta tiveram o tamanho reduzido de 10 bytes.

A segunda aplicação visa a obtenção da latência, em milisegundos, medida a partir do instante que o cliente enviou uma mensagem ao servidor solicitando uma grande quantidade de bytes (*throughput*) até o momento em que todos os dados sejam recebidos em texto puro (decriptado) pela camada de aplicação. Desta forma, será verificada principalmente a influência do processo de encriptação e decriptação via algoritmo simétrico no atraso de entrega dos dados à camada de aplicação para cada um dos protocolos de segurança avaliados (TLS e IPSec). No caso específico, a quantidade de dados solicitada foi de 26 MBytes.

4.3 Casos de Estudo

As aplicações cliente/servidor anteriormente detalhadas foram testadas sob três condições diferentes de operação (casos de estudo) conforme explicado a seguir:

- No primeiro caso de estudo, cada aplicação cliente/servidor foi executada no modo de operação sem segurança.

- No segundo caso, cada aplicação foi executada no mesmo modo anterior porém foi configurado um túnel IPSec adicionando segurança a comunicação TCP/UDP/IP.
- E no terceiro e último caso, em cada aplicação foi realizada uma pequena adaptação para suportar o protocolo TLS e por conseguinte, adicionar segurança similar a fornecida pelo IPSec.

Neste capítulo, a palavra segurança tem o seguinte significado: autenticação das entidades participantes da comunicação, confidencialidade (privacidade) e integridade dos dados trafegados durante a sessão de comunicação.

Para a comparação entre os protocolos TLS e o IPSec ser justa foram utilizados os mesmos algoritmos de criptografia em cada um deles.

4.3.1 Aplicação Sem Criptografia

Neste caso de estudo, ambas as aplicações de teste terão o seu melhor desempenho pois nenhum processamento adicional é realizado. Este caso foi criado para servir como medida base e demonstrar de forma clara a perda de desempenho causado pela inserção da segurança na comunicação.

4.3.2 Aplicação Com IPSec

Foi utilizada a implementação IPSec do sistema operacional Windows 2000 Professional. E esta implementação suporta as seguintes características:

- Para a troca de chaves via ISAKMP/Oakley (IKE): *Diffie-Hellman* de 768 e 1024 bits, algoritmos simétricos: DES e 3DES, verificação de integridade: SHA e MD5 e *Perfect Forward Secrecy*.
- Para a autenticação podem ser usados certificados digitais, chaves pré-compartilhadas ou Kerberos versão 5.
- Segurança AH com verificação de integridade dos dados e cabeçalho: SHA e MD5.

- Segurança ESP com algoritmos simétricos: DES e 3DES e para verificação de integridade dos dados: SHA e MD5.
- Modo transporte ou túnel.

Para o ambiente operacional de teste e medição foram empregadas e configuradas, tanto no cliente como servidor, as seguintes opções de segurança:

- Troca de chaves via IKE com *Diffie-Hellman* de 1024 bits, algoritmos simétricos: DES e 3DES, verificação de integridade: SHA e *Perfect Forward Secrecy*.
- Autenticação via chaves simétricas pré-compartilhadas.
- Segurança ESP com algoritmos simétricos DES (qdo ESP/DES) e 3DES (qdo ESP/3DES) e para verificação de integridade dos dados, SHA.
- Modo túnel.

4.3.3 Aplicação Com TLS

Foi utilizada a implementação Java do novo protocolo TLS/UDP desenvolvida durante este trabalho. E o conjunto criptográfico utilizado para adicionar segurança às aplicações de teste foi o TLS_UDP_DHE_DSS_WITH_DES_EDE_CBC_SHA e posteriormente o TLS_UDP_DHE_DSS_WITH_3DES_EDE_CBC_SHA. E um conjunto criptográfico equivalente foi utilizado para o protocolo TLS/TCP.

Sendo assim, para a negociação da chave simétrica foi usado o *Diffie-Hellman* de 1024 bits, para encriptação via algoritmo simétrico, DES e 3DES, para autenticação foram utilizados certificados DSA auto-assinados (1024 bits) e finalmente para verificação da integridade dos dados, SHA.

4.4 Resultados: Análise e Comparação

Os resultados descritos na tabela 05 comprovam alguns fatos, a citar:

- O IPSec tem melhor desempenho que o TLS. Um dos motivos é que o IPSec é implementado no *kernel* do sistema operacional (*kernel mode*) e o TLS, no espaço de usuário (*user mode*).
- O algoritmo simétrico de bloco DES requer menos recursos computacionais que o 3DES, porém tem sua segurança prejudicada.
- O desempenho do UDP foi superior ao equivalente TCP. O *overhead* do TCP é criado pelas vantagens disponibilizados por este protocolo de transporte em relação ao UDP, como a entrega sequenciada e assegurada dos pacotes (ver apêndice B).
- O processo de troca de chaves, tanto no IPSec como no TLS, acarreta um atraso inicial na comunicação. E a negociação dos parâmetros de segurança no IPSec é muito mais rápida que no TLS.
- Os tempos de processamento medidos dependem de vários fatores, entre eles: os computadores, os algoritmos, a linguagem de programação e do tipo de rede utilizada.
- A implementação em Java do protocolo TLS prejudicou o seu desempenho.

Tabela 5 - Resultado da Medida: IPSec x TLS

	Sem Segurança UDP	Sem Segurança TCP	IPSec/UDP	IPSec/TCP	TLS/UDP	TLS/TCP	
Tempo Médio Total de uma Transação 10B/10B (ms)	10	20	350	360	901	1012	DES
			611	621	911	1032	3DES
Tempo Médio Total de Transferência de 26 MB (ms)	4261	4992	11417	23318	27019	46016	DES
			18917	37374	50943	93388	3DES

4.5 Tunelamento

Foi desenvolvido também um aplicativo cliente/servidor, com funcionalidade similar ao *port forwarding* do OpenSSH e ao *SSL Wrappers*, que permite encriptar as comunicações UDP/IP de qualquer aplicação cliente/servidor através da criação de um túnel TLS/UDP sem a necessidade de modificação do código da aplicação. Alguns exemplos de *SSL Wrappers* de domínio público: stunnel e sslwrap.

Esta facilidade acarreta em uma grande benefício ao usuário final que deseja adicionar segurança a comunicação pois com este mecanismo não há necessidade do código fonte da aplicação. E poderia ser comparado a uma equivalente configuração de um túnel IPSec.

4.5.1 Port Forwarding

A seguir, é mostrado o modo simples de funcionamento da aplicação túnel cliente/servidor desenvolvida e as etapas necessárias para a interação com a aplicação a ser assegurada:

No servidor:

1- Redirecionar a aplicação servidora original para ouvir em um endereço IP local, como 127.0.0.1 em uma porta UDP Z qualquer (não utilizada).

2- Colocar o programa túnel servidor para ouvir as requisições da aplicação túnel cliente em um endereço IP local válido na rede, como o 146.164.32.67 em uma porta UDP Y qualquer. Este programa repassará os dados (após a decriptação) da aplicação cliente original para o endereço local da aplicação servidora original, por exemplo: 127.0.0.1, na porta Z, previamente conhecida e configurada no passo 1.

No cliente:

1- Colocar o programa túnel cliente para ouvir as requisições da aplicação cliente original em um endereço local, como o IP 127.0.0.1 em uma porta UDP X qualquer. Este programa repassará os dados da aplicação cliente original para o endereço remoto do programa túnel servidor em uma porta na qual esteja ouvindo, previamente conhecida e configurada (no caso, a porta Y).

2- Redirecionar a aplicação cliente original de forma a apontar para o endereço IP local, como o 127.0.0.1 em porta UDP X previamente conhecida e configurada no passo 1 acima.

4.6 Comparação e Análise Crítica

Esta seção fornece uma análise comparativa do novo protocolo TLS/UDP com os outros protocolos de segurança de rede apresentados, como o IPSec, WTLS e o *Secure UDP*. A forma de apresentação desta análise foi através da descrição das vantagens e desvantagens de cada protocolo.

(a) Protocolo TLS/UDP

Vantagens:

- Não depende de suporte pelo *kernel* do sistema operacional. Pode ser implementado totalmente em *userland.*
- A aplicação tem a possibilidade de negociar os parâmetros de segurança caso assim se faça desejável.
- Somente é necessário um mínimo de modificações na aplicação para obter todas as facilidades disponíveis pelo novo protocolo, tornando o trabalho para o desenvolvedor praticamente nulo e de fácil implementação.
- Um padrão de domínio público disponível para a comunidade acadêmica e para a Internet de um modo geral.
- É o primeiro protocolo de criptografia de rede que opera na camada de transporte disponibilizado para todas as aplicações que utilizam o protocolo de transporte não confiável UDP.
- Este protocolo reutiliza muitas técnicas já disponíveis em padrões bem conhecidos e de domínio público. Assim sendo, a pesquisa e a experiência obtida no desenvolvimento dos protocolos (padrões) anteriores foram transferidas automaticamente para este novo protocolo.

Desvantagens:

- Como todo protocolo novo será necessário passar pelo crivo da comunidade acadêmica para validar e assegurar que este protocolo é seguro e que não existam falhas de segurança não previstas pelos autores.
- O certificado digital do cliente é transmitido sem encriptação.

(b) Protocolo IPSec

Vantagens:

- O protocolo está integrado ao protocolo IP e ao sistema operacional. E pelo fato de executar em *kernel mode* tem mais privilégios de execução do que se estivesse em *user mode*.
- Não é necessária nenhuma modificação nas aplicações, sendo totalmente transparente a qualquer tipo de aplicação.
- Suporta diferentes tipos de aplicações indiferentemente da utilização do protocolo de transporte TCP ou UDP.
- Os fabricantes de placas de rede Ethernet, 3Com e Intel, disponibilizam suporte nativo em hardware (em alguns tipos de placas) para os algoritmos de criptografia DES, 3DES, MD5 e SHA e assim o desempenho do IPSec pode ser alavancado de forma abrupta.
- O IPSec impede os ataques de "Port Scanner" e "Spoof source IP Address".
- Suporte ao *Perfect Forward Secrecy*.

Desvantagens:

- O sistema operacional necessita fornecer o suporte ao IPSec (ou precisa existir uma solução de algum fabricante de software para tal). Isto pode não ser possível quando uma das partes está conectada através de um computador de uma localização pública como um quarto de hotel,

cybercafé ou aeroporto, por exemplo. Assim, o IPSec provê uma solução para o problema mas a infra-estrutura necessária para utilizar o IPSec pode ser ainda um obstáculo.

- O usuário necessita do suporte do administrador de rede para habilitar o IPSec em ambas as máquinas utilizadas na comunicação.
- A camada de aplicação fica isolada do protocolo e sem possibilidade de interferir na negociação dos parâmetros de segurança.
- Os fabricantes de soluções de segurança IPSec geram normalmente uma implementação não otimizada e com desempenho inferior àquela codificada diretamente no sistema operacional.
- Dependência total do algoritmo *Diffie-Hellman* para a combinação da chave simétrica (secreta). Caso se descubra um modo de quebrá-lo, o IPSec (IKE, na verdade) estará quebrado.
- Autenticação somente de computadores. O TLS suporta autenticação de usuários possibilitando o controle de acesso ao nível de usuário.
- Problemas de funcionamento do ESP e IKE com *Network Address Translation* (NAT). E inviabilidade da utilização do AH com NAT.

(c) Protocolo *Secure UDP* [3]

Vantagens:

- Não depende de suporte pelo *kernel* do sistema operacional. Pode ser implementado totalmente em *userland.*
- A aplicação tem a possibilidade de negociar os parâmetros de segurança caso assim se faça desejável.
- Este protocolo reutiliza muitas técnicas já disponíveis em padrões bem conhecidos e de domínio público. Assim, a pesquisa e a experiência obtida dos protocolos (padrões) anteriores foram transferidas automaticamente ao *Secure UDP*.

Desvantagens:

- Este protocolo foi especificado teoricamente mas nenhuma implementação do protocolo foi realizada. O autor mencionou que trabalhos futuros envolveriam a implementação do protocolo e a execução de *benchmarks*.
- Utiliza o protocolo TLS, e consequentemente o TCP, para negociação dos parâmetros de segurança.
- O protocolo não tem uma especificação bem definida acarretando em possíveis implementações incompatíveis.

(d) Protocolo WTLS

Vantagens:

- Suporte a ambos os tipos de comunicação: orientado a conexão ou a datagramas.
- Suporte aos algoritmos baseados em curvas elípticas que mesmo com um tamanho de chave menor que os tradicionais, tem desempenho igual ou superior. E assim, economiza recursos (memória, cpu e energia) do dispositivo de comunicação móvel (celular).

Desvantagens:

- Suporta tamanho de chaves pequenos tornando a comunicação vulnerável, caso os conjuntos criptográficos fracos sejam utilizados e caso a atualização do material criptográfico demore a ser realizada.
- O mecanismo para atualização do vetor de inicialização tem uma falha de segurança conforme mencionado no capítulo 2.
- O certificado digital do cliente é transmitido sem encriptação.

Capítulo 5 - Conclusão

Neste capítulo são apresentadas as conclusões obtidas a partir da pesquisa previamente realizada. E para finalizar são descritas as propostas e sugestões de trabalhos futuros que vicejaram durante o desenvolvimento deste trabalho.

5.1 Considerações finais

Este trabalho faz uma análise das soluções de segurança disponíveis para o protocolo de transporte UDP. E em seguida, apresenta o novo protocolo TLS/UDP.

Este protocolo endereça os problemas de segurança concernentes as várias aplicações que fazem uso do protocolo de transporte UDP. Diversos protocolos da camada de aplicação podem fazer uso deste novo protocolo, como exemplo inicial temos aqueles mencionados nos apêndices B, D até o K.

Foi especificado um protocolo que demonstra ser possível adicionar segurança as comunicações UDP/IP mesmo com o descarte e a falta de ordenação dos datagramas de usuário apenas adicionando algumas funcionalidades ao protocolo TLS original, como a retransmissão das mensagens de *handshake*, possibilidade de decriptação independente de cada datagrama, uso de uma janela deslizante para evitar ataques de reprodução e negação de serviço viabilizada pela inserção de um número de sequência no cabeçalho TLS/UDP.

A implementação Java do protocolo TLS/UDP mostrou ser superior em relação a equivalente implementação do TLS/TCP do JDK/Sun embora nem todas as funcionalidades tenham sido implementadas em código e pode servir como uma implementação de referência do novo protocolo.

As aplicações cliente/servidor de *benchmark* desenvolvidas neste trabalho foram baseadas nos programas de *benchmarks* comumente utilizados para avaliação do desempenho da rede e serviram para mensurar e comparar os protocolos de segurança de rede conforme detalhado no capítulo quatro.

A facilidade de tunelamento das comunicações não seguras acarreta em um grande benefício ao usuário que não tem o código fonte da aplicação para modificá-la

e torná-la segura. Independente de ter o código ou não, toda aplicação cliente/servidor pode se tornar segura através desta facilidade disponibilizada pelo protocolo TLS/UDP. Este túnel pode ser criado através de uma simples aplicação cliente/servidor que faz uso da técnica de “port forwarding” (desde que se conheça previamente a(s) porta(s) utilizada(s) pela aplicação).

Conforme analisado e comparado tanto o IPSec como o TLS terão ainda um grande tempo de convivência juntos pois oferecem algumas funcionalidades diferentes pelo menos neste estágio de desenvolvimento. Anteriormente as opções para adicionar segurança as aplicações UDP/IP eram: IPSec ou algum mecanismo proprietário dentro da própria aplicação. E agora com este novo protocolo TLS/UDP, uma nova facilidade foi disponibilizada e com a segurança equivalente ao IPSec.

Curiosidade: de acordo com a lista de discussão oficial do grupo que desenvolveu o protocolo TLS foi discutido no momento da definição do escopo de abrangência do TLS que este não funcionaria sobre o protocolo de transporte UDP e que esta funcionalidade seria postergada para a próxima versão do TLS. Porém desde a data da publicação da RFC em Janeiro de 1999 até hoje não foi feito nenhum esforço nesse sentido e nem aparece com uma das tarefas do grupo de trabalho do IETF responsável pelo desenvolvimento do TLS.

5.2 Propostas de Trabalhos Futuros

- A submissão deste novo protocolo ao grupo de trabalho do IETF responsável pelo TLS para uma possível criação de uma *Internet-Draft* e futuramente de uma RFC.
- A pesquisa e definição dos algoritmos de chave pública e de assinatura baseados nas curvas elípticas a serem adicionados ao protocolo TLS/UDP.
- A inserção do número de sequência no cabeçalho TLS/UDP na atual implementação Java para dificultar os ataques por *reprodução* e por conseguinte, a negação de serviço.
- A implementação em Java do mecanismo de retransmissão das mensagens de *handshake* para tornar mais confiável o protocolo TLS/UDP conforme explicado no capítulo três.

- A implementação em Java, dos algoritmos de criptografia não implantados na versão atual, como RSA e AES.
- A codificação do campo IV no cabeçalho do TLS/UDP para tornar a decriptação de cada datagrama IP independente do anterior na atual implementação Java e a devida atualização no cálculo do MAC referente a esta modificação.
- Implementação da API referência do novo protocolo TLS/UDP na linguagem de programação C e/ou C++ comumente utilizadas para ampliar potencialmente o uso do protocolo na comunidade acadêmica e científica.
- Implementação do TLS/UDP em alguns softwares de domínio público (*open source*) amplamente conhecidos que implementam protocolos de rede baseados no modo de transporte orientado a datagrama UDP. Como exemplo: o software BIND (*Berkeley Internet Name Domain*) para o protocolo DNS.
- Estudo do tamanho da janela deslizante ótimo ou criação de uma algoritmo para alocação dinâmica do tamanho da janela deslizante de acordo com alguns parâmetros da rede, tais como: vazão, atraso e a perda de datagramas no meio de transmissão.
- Análise da interferência do protocolo na transmissão de fluxos de dados multimídia (voz e vídeo) com a realização de testes de um programa cliente/servidor que implemente o protocolo RTP/RTSP devidamente acoplado com o protocolo TLS/UDP.
- Pesquisar algoritmos simétricos com tempo de encriptação e decriptação menores para viabilizar a utilização deste protocolo com as aplicações em tempo real.
- Verificar a viabilidade do uso de hardware específico para realizar o processamento criptográfico e assim melhorar o desempenho das aplicações que utilizam o protocolo TLS/UDP.
- A necessidade de largura de faixa para as aplicações multimídia é muito grande. Embora o TLS/UDP assegure a confidencialidade dos dados transmitidos, um adversário pode descobrir a duração dos arquivos das mídias transmitidas e poderia usar os padrões de utilização da largura de faixa para obter alguma informação privilegiada. A análise de padrão de

tráfego é possível nas redes telefônicas convencionais e é mais fácil na Internet. O ataque por análise dos padrões de dados é um problema muito difícil de contornar especialmente para arquivos multimídia devido ao grande tamanho dos arquivos transmitidos e pesquisas devem ser realizadas para dificultar este ataque.

- Outra possível direção para pesquisa é a melhoria no desempenho da encriptação para fluxos multimídia. Normalmente, um servidor estaria servindo múltiplos fluxos encriptados para múltiplos clientes. O servidor poderia ter placas de encriptação dedicadas *onboard*, mas estas placas não serão suficientes para lidar com todos os seus clientes (não necessariamente). Um estudo da viabilização do uso simultâneo da mesma chave simétrica pelo servidor para um mesmo vídeo/áudio transmitido para diferentes clientes (*multicasting*) poderia ser realizado.
- Medir e comparar o desempenho do TLS/UDP em relação ao IPSec em outros sistemas operacionais como Conectiva Linux e OpenBSD.
- Especificar uma proposta de *benchmarking* para as implementações TLS/UDP similar a proposta desenvolvida ("*Terminology for Benchmarking IPsec Devices", Internet-Draft*) para *benchmarking* de dispositivos IPSec pelo grupo de trabalho *Benchmarking Methodology* do IETF.
- Estudo das técnicas de compressão de dados que poderiam ser utilizadas no protocolo TLS/UDP.
- Estudar a viabilidade da utilização do protocolo ISAKMP, Oakley ou IKE no *Handshake* TLS/UDP para negociação dos parâmetros de segurança.
- Estudar a possibilidade do fornecimento de uma URL para obtenção do certificado do cliente e/ou servidor no protocolo TLS/UDP conforme previsto na RFC 3546, "*Transport Layer Security Extensions*", para o TLS. E também estudar a viabilidade de suporte a *Online Certificate Status Protocol* (OCSP) e de *Certificate Revocation List* (CRL) para validação dos certificados digitais.
- Verificar a possibilidade de aumentar o desempenho da implementação TLS/UDP em Java através da utilização de compiladores para código nativo e *Just In Time* (JIT) de performance superior.

Referências Bibliográficas

[1] CERT Coordination Center, http://www.cert.org/

[2] SANS Institute, http://www.sans.org/

[3] HASSAN, Ahmed, "Secure UDP", Universidade de Waterloo, 2000. http://plg.uwaterloo.ca/~aeehassa/home/papers/crypto/secureUDP.htm

[4] OPENSSL, http://www.openssl.org/related/apps.html

[5] Fórum IETF/TLS, http://www.ietf.org/html.charters/tls-charter.html

[6] FRIER, P., KARLTON, P., KOCHER, P., "The SSL 3.0 Protocol", Netscape Communications Corp., Nov 18, 1996. http://www.netscape.com/eng/ssl3/

[7] OPENSOURCE, Organização não comercial dedicada a promover o OpenSource. http://www.opensource.org

[8] SUN Microsystems, "Java Technology", http://java.sun.com/

[9] Open Mobile Alliance, "WAP Fórum", http://www.wapforum.org/

[10] Fórum IETF/IPSec, http://www.ietf.org/html.charters/ipsec-charter.html

[11] ATKINSON, R, "IP Authentication Header", RFC 1826, 1995.

[12] ATKINSON, R, "IP Encapsulating Security Payload", RFC 1827, 1995.

[13] HOUSLEY, R., FORD, W., POLK, W. et al., "Internet Public Key Infrastructure: Part I: X.509 Certificate and CRL Profile", RFC 2459, Janeiro 1999.

[14] Fórum IETF/PKIX, "Infra-Estrutura de Chaves Públicas", http://www.ietf.org/html.charters/pkix-charter.html

[15] SUN Microsystems, "Java Secure Socket Extension", http://java.sun.com/products/jsse/index.jsp

[16] PISTOIA, Marco et al, "JAVA 2 Network Security", Prentice Hall, Agosto de 1999.

[17] Defense Advanced Research Projects Agency, "Transmission Control Protocol", RFC 793, Setembro de 1981.

[18] POSTEL, J., "User Datagram Protocol", RFC 768, Agosto de 1980.

[19] POSTEL, J., "Internet Protocol", RFC 791, Setembro de 1981.

[20] DAMGARD, I., "A design principle for hash functions", Advances in Cryptology - Crypto '89, pp. 416-427, Springer-Verlag, 1990.

[21] MERKLE, R.C., “One way hash functions and DES”, Advances in Cryptology -- Crypto ’89, pp. 428-446, Springer-Verlag, 1990.

[22] NIST FIPS PUB 180-1, "Secure Hash Standard", National Institute of Standards and Technology, U.S. Department of Commerce, Maio de 1994.

[23] NIST, FIPS PUB 197, “Advanced Encryption Standard (AES)”, Novembro de 2001.

[24] KELSEY, J., SCHNEIER, B., WAGNER, D., “Key-Schedule Cryptanalysis of 3-WAY, IDEA, G-DES, RC4, SAFER, and Triple-DES”, Advances in Cryptology--CRYPTO ’96 Proceedings, pp. 237-251, Springer-Verlag, 1996.

[25] MENEZES, A.J., OORSCHOT, P. C., VANSTONE, S.A., “Handbook of Applied Cryptography”, CRC Press New York, 1997.

[26] RSA Corporation, RSA FAQ.

[27] DIFFIE, W., HELLMAN, M., "New Directions in Cryptography", IEEE Transactions on Information Theory, V. IT-22, n. 6, pp. 74-84, Jun 1977.

[28] RIVEST, R., SHAMIR, A., ADLEMAN, L., "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems,"Communications of the ACM, v. 21, n. 2, pp. 120-126, Fevereiro de 1978.

[29] NIST, FIPS PUB 186, "Digital Signature Standard," National Institute of Standards and Technology, U.S. Department of Commerce, 18 de Maio, 1994.

Apêndice A – Glossário

IETF: *Internet Enginnering Task Force* é um órgão ligado ao ISOC (Internet Society) que tem como objetivo estabelecer grupos de trabalho em assuntos diversos da Internet, estes grupos de trabalho criam recomendações e padrões publicados através das RFCs;

MTU: *Maximum Transfer Unit* é designado em bytes para indicar qual é a máxima quantidade de dados possíveis de serem enviados por vez pelo meio de transporte;

NAT: *Network Address Translator* é utilizado para designar um processo de trocas de IPs utilizados em redes com endereçamento diferente do utilizado no encaminhamento normal (ex: redes privadas prefixo 10.0.0.0/8);

RFC: *Request For Comments*, utilizado para descrever uma recomendação, um protocolo ou uma funcionalidade utilizada(o) na Internet;

Buffers: Espaço em memória de dados para armazenamento temporário ou memória dedicada para armazenamento temporário;

Cache: Termo utilizado para designar uma memória mais rápida dedicada para o acesso a informações que são freqüentemente consultadas na memória principal. Durante o funcionamento a memória cache fica com uma cópia do espaço em memória principal que é mais acessado;

Checksum: Termo utilizado para indicar uma verificação de erro através do somatório dos valores;

Default: Termo utilizado para indicar uma configuração, procedimento ou valor padrão, ou para designar uma configuração original;

Ethernet: Tecnologia utilizada para comunicação entre computadores em redes locais;

Header: Cabeçalho. Geralmente para designar o formato de dados iniciais em um pacote ou arquivo;

Reset: Significa o término de um processo ou o seu desligamento para depois reiniciá-lo do estado inicial;

Registro: O PDU (Protocol Data Unit) da camada referente ao protocolo *Record*.

Segmento: É o nome dado à Unidade de Dados do Protocolo ou PDU (Protocol Data Unit) da camada de transporte.

Timeout: Expressão utilizada para indicar o tempo em que um certo processo ou procedimento irá esperar até receber algum resultado ou resposta. Caso não exista uma resposta até um certo *timeout* este é considerado expirado;

Stateful Compression Algorithm: Um algoritmo de compressão de dados que utiliza informação sobre dados previamente comprimidos para uso nos dados a serem processados.

Apêndice B - Modelo de Referência TCP/IP

Neste apêndice será realizado uma revisão sobre as características mais importantes referentes a camada de transporte e de rede do modelo TCP/IP de modo a subsidiar o leitor com informação suficiente para o entendimento dos capítulos da dissertação nas questões pertinentes a estas camadas.

Conforme veremos serão detalhadas as diferenças entre o protocolo de transporte TCP e o UDP (camada quatro). E também será feita uma apresentação do protocolo de rede IP (camada três).

B.1 - Introdução

O modelo TCP/IP é similar ao modelo OSI com algumas simplificações, como a ausência da camada de apresentação e sessão presentes no modelo OSI ou incorporação destas na própria aplicação. Na figura 21 temos uma representação do modelo TCP/IP, onde verificamos alguns protocolos que operam na camada de aplicação como Telnet (*shell* remoto), FTP (transferência de arquivos), SMTP (*email*), HTTP (web) e RealAudio (multimídia). Na camada de transporte, temos os protocolos TCP e UDP e na camada de rede, o protocolo IP propriamente dito.

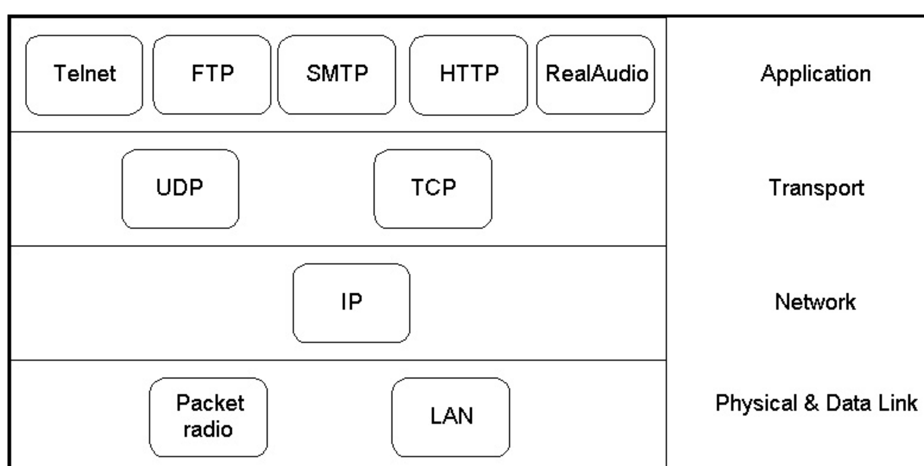


Figura 21 - Modelo TCP/IP

B.2- Protocolo TCP

O protocolo TCP (*Transmission Control Procotol*) [17] é um protocolo de transporte orientado a conexão designado como STD número 7 e a especificação está descrita na RFC 793. Os pacotes são transmitidos e recebidos na ordem correta e livres de erros.

O protocolo TCP é bastante robusto, sendo utilizado em diferentes tipos de redes com topologia, velocidade e retardo de transmissão variados. A importância deste protocolo está refletida na presença do seu nome na própria arquitetura, TCP/IP.

O protocolo através do esquema de reconhecimento positivo e retransmissão, garante que todos os segmentos sejam entregues ao nó destino. Além de prover este serviço confiável podemos citar outras características importantes do protocolo TCP:

- Orientado à conexão: antes das duas aplicações iniciarem a transmissão elas devem estabelecer uma conexão, assim como em telefonia. Durante o estabelecimento da conexão, alguns parâmetros referentes à transmissão dos dados são determinados. A conexão é terminada quando uma das partes decide fazê-lo. É importante ressaltar que as conexões são *full-duplex*, o que permite que dados de controle da transmissão em um determinado sentido possam ser enviados através de dados que trafegam no sentido oposto (*piggybacking*). Vale ressaltar que o uso do *piggybacking* não é obrigatório porque, além de poder não haver transmissão de dados em um dos sentidos, as transmissões de dados nos dois sentidos são independentes.

- Orientado a *streams* de *bytes* não estruturados: os dados são enviados como uma sequência de *bytes*, não contendo nenhum tipo de estrutura adicional. O destino se preocupa somente em entregar os bytes para a aplicação na mesma ordem em que eles foram transmitidos.

- Utilização de *buffers*: os dados da aplicação transmissora são enviados para o software TCP em quantidades de bytes escolhidas da forma mais conveniente, variando portanto de aplicação para aplicação. O

software TCP por sua vez, ao receber os dados da aplicação transmissora, acumula em um *buffer* de transmissão, enviando-os em blocos cujo tamanho geralmente difere daquele com que ele recebe os dados da aplicação e de forma a otimizar a taxa de transferência dos dados. O mesmo raciocínio vale para o nó destino, ou seja, os dados são armazenados em um *buffer* de recepção e entregues para a aplicação necessariamente em blocos com o mesmo tamanho utilizado na transmissão através da rede.

A princípio, não é óbvio perceber como um protocolo que utiliza serviços do protocolo IP pode prover um serviço confiável, já que este último é totalmente não confiável. Para atingir este objetivo, o protocolo TCP executa as seguintes tarefas:

- Divide os dados passados pela aplicação em blocos de tamanho conveniente. Estes blocos são denominados segmentos.

- Quando cada segmento é enviado, um temporizador para aquele segmento é disparado com um determinado valor para o intervalo de tempo até que este se esgote. O software TCP então espera pela confirmação do recebimento (*acknowledgment ticket*) do segmento pelo nó destino. Caso o tempo se esgote sem que o reconhecimento tenha chegado, o segmento é retransmitido.

- Quando o software TCP recebe dados do outro ponto da conexão, ele envia um reconhecimento. Na prática, reconhecimentos não são enviados imediatamente mas sim após algum atraso devidamente escolhido. Este procedimento é conhecido como mecanismo de reconhecimentos atrasados (*delayed acks*) e visa aumentar a eficiência da conexão utilizando apenas um pacote para reconhecer uma quantidade maior de dados.
- Mantém uma soma de checagem no cabeçalho (*checksum*) de forma a detectar qualquer alteração no conteúdo da informação transmitida.

Caso haja algum erro, o pacote é descartado no destino e espera-se por retransmissão.

- Por serem transmitidos (encapsulados) em datagramas IP, os segmentos podem chegar ao destino fora de ordem. O software TCP deve efetuar então a reordenação dos dados a fim de garantir a entrega destes em ordem para a aplicação. Ainda pelo mesmo motivo acima, os segmentos podem ser duplicados, fazendo com que o software TCP tenha que descartar dados duplicados, não os entregando à aplicação.

- Controle de fluxo fim a fim fazendo com que o nó destino permita que o nó fonte envie no máximo a quantidade de *bytes* correspondente ao espaço livre no *buffer* de recepção.

Dentre as tarefas supracitadas, no que se refere à confiabilidade, pode-se dizer que as mais importantes são a segunda e terceira, as quais em conjunto formam o conceito de reconhecimento positivo e retransmissão. Positivo aqui se refere ao fato do receptor enviar reconhecimentos correspondentes à segmentos recebidos, em oposição ao caso onde o destino implementaria os temporizadores e enviaria reconhecimentos negativos com relação a segmentos que não tivessem sido recebidos até a expiração do temporizador.

A figura 22 mostra como este esquema garante que todos os segmentos serão recebidos em algum momento, inclusive em casos onde ocorre a expiração do temporizador. Pode-se notar que nesta figura os segmentos e os reconhecimentos estão numerados. A fim de que seja possível para o receptor detectar segmentos duplicados é necessário numerar os segmentos com um número de sequência.

Da mesma forma, o nó fonte precisa distinguir a qual segmento se refere cada reconhecimento, o que também demanda a numeração dos mesmos.

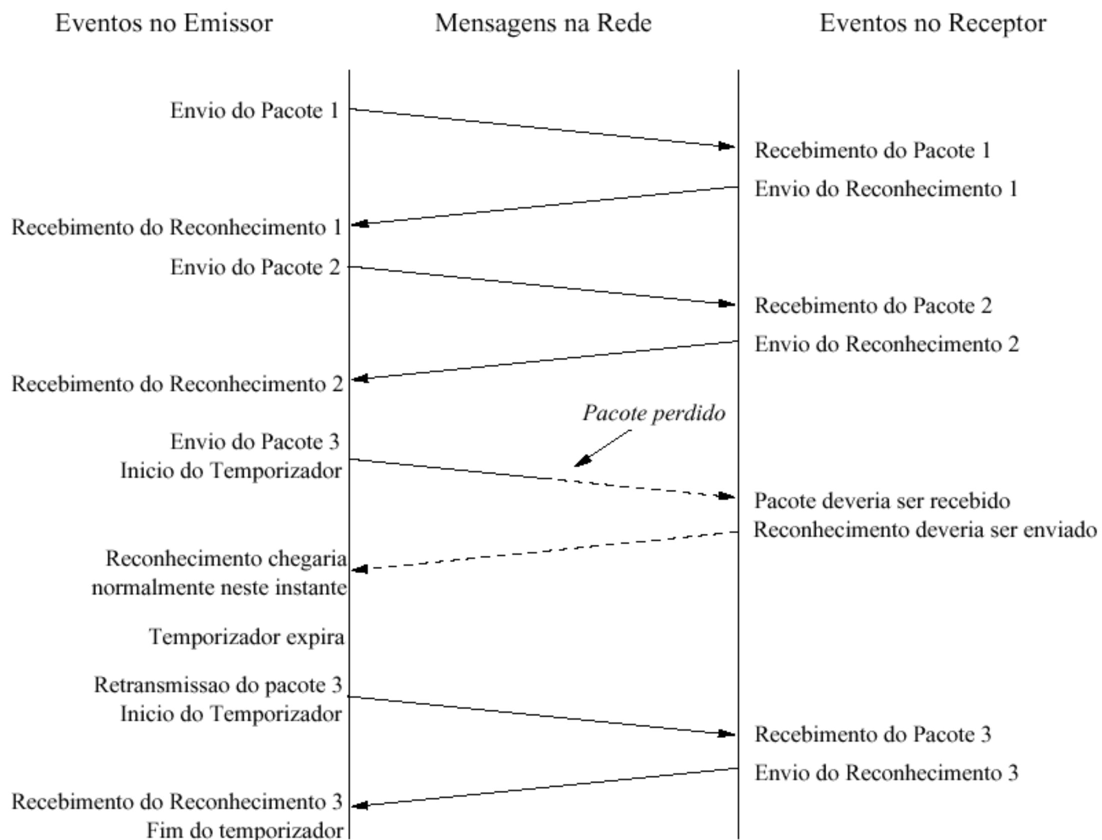


Figura 22 - Reconhecimento positivo e retransmissão

Janela Deslizante

De acordo com o exposto anteriormente, no caso de se transmitir um segmento por vez e esperar o reconhecimento deste para a transmissão do próximo, obtêm-se uma vazão máxima (desconsiderando retransmissões e tempos adicionais de processamento) igual a [MSS * 8] / RTTméd (bits/s), onde RTTméd é o tempo médio de ida do segmento e volta (*Round Trip Time*) de seu reconhecimento e MSS é o tamanho máximo do segmento em bytes (*Maximum Segment Size*). É evidente que o caso de reconhecimento positivo simples desperdiça uma quantidade substancial de largura de faixa da rede na medida em que atrasa o envio de um novo pacote até o recebimento do reconhecimento do pacote anterior.

Para tornar o mecanismo mais eficiente, introduz-se o mecanismo de janelas deslizantes. A idéia consiste em permitir que o nó fonte transmita vários pacotes sem esperar por um reconhecimento. Para isso, um protocolo com janelas deslizantes

utiliza uma janela com um determinado tamanho e transmite todos os dados dentro desta janela sem esperar por reconhecimento algum. Sendo assim, o número de pacotes não reconhecidos (transmitidos mas cujo reconhecimento ainda não foi recebido) fica limitado ao tamanho da janela.

O desempenho de protocolos com janelas deslizantes depende do tamanho da janela e da velocidade com que a rede pode aceitar pacotes, sendo esta última determinada por velocidades de transmissão, retardo de propagação, tempo gasto em filas e processamento. Aumentando-se o tamanho da janela e supondo um RTT fixo, chega-se a uma situação onde a rede se torna completamente ocupada, o que acarreta um aumento na vazão (figura 23).

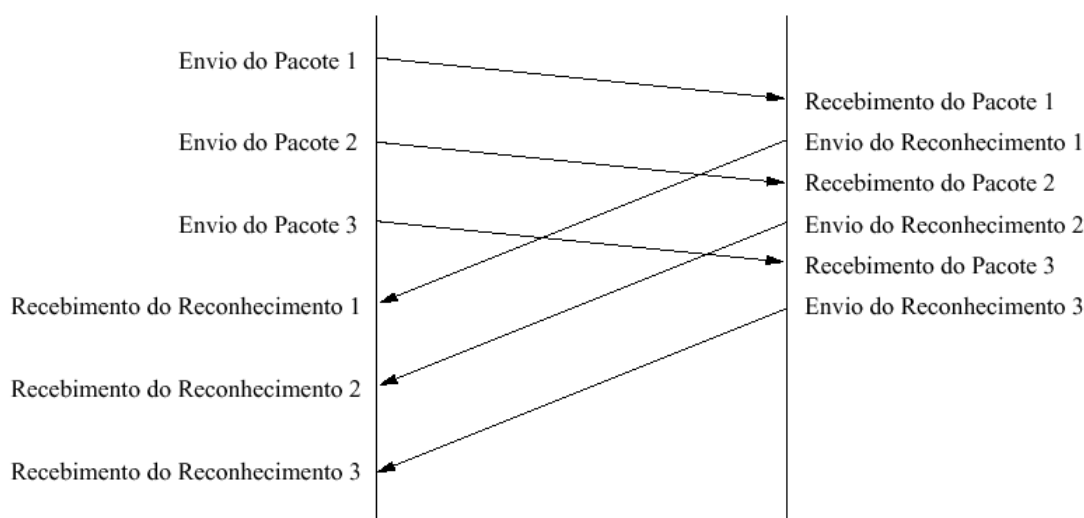


Figura 23 - Janela Deslizante

Conceitualmente neste tipo de protocolo o nó fonte sabe quais pacotes foram reconhecidos dentro da janela e mantém um temporizador para cada pacote em separado, retransmitindo cada pacote cujo temporizador se esgotar. Ao obter o reconhecimento de um pacote inicial, o protocolo desloca a sua janela (deslizante), incluindo novos pacotes a serem transmitidos. No nó destino, de forma análoga, uma janela de reconhecimentos á mantida.

Pode-se concluir então que a janela divide os pacotes em três conjuntos distintos. Os pacotes que foram "transmitidos, recebidos e reconhecidos com sucesso", os quais não pertencem mais à janela e situam-se à esquerda desta; os pacotes que ainda não foram transmitidos e que situam-se à direita da janela e, finalmente, os pacotes que pertencem à própria janela e estão sendo transmitidos.

No caso do protocolo TCP, este mecanismo opera em termos de bytes, e não em termos de segmentos ou pacotes. Para tal, todos os bytes em um *stream* de dados são numerados sequencialmente, e o nó fonte guarda ponteiros associados a cada conexão. O primeiro ponteiro delimita a extremidade esquerda da janela, separando os bytes que foram transmitidos e reconhecidos dos bytes que podem ser transmitidos sem reconhecimento ou transmitidos e ainda não reconhecidos. O segundo delimita a extremidade direita da janela, separando os bytes que podem ser transmitidos sem reconhecimento dos bytes que ainda não podem ser transmitidos. O terceiro e último ponteiro separa os bytes dentro da janela que já foram transmitidos dos que ainda estão para serem transmitidos. Consequentemente, o terceiro ponteiro move-se mais rapidamente do que os demais.

Outra peculiaridade do protocolo TCP neste aspecto é o fato do tamanho da janela poder variar com o tempo. Isto ocorre porque o destino, ao enviar os reconhecimentos, indica a quantidade máxima de bytes que pode receber no seu *buffer*. Este valor é denominado janela anunciada (*advertised window*).

Sendo assim, o nó fonte pode aumentar ou diminuir o tamanho da sua janela de transmissão de acordo com o valor anunciado pelo nó destino, o qual nunca será maior do que o espaço disponível no *buffer* de recepção.

A grande vantagem deste mecanismo é que fornece um controle de fluxo fim a fim na medida em que o nó que transmite fica impedido de enviar dados a uma taxa maior do que a suportada pelo nó receptor, aumentando a confiabilidade da transmissão e evitando que pacotes sejam descartados no destino. Eventualmente a janela pode atingir o valor zero, significando que o *buffer* de recepção está completamente cheio. Neste caso a transmissão é interrompida até que o receptor anuncie um valor de janela diferente de zero.

Um ponto muito importante sobre controle de fluxo é o fato de que prover o controle fim a fim não é suficiente para evitar perdas através da rede. A Internet é um meio altamente heterogêneo em termos de velocidade de processamento dos equipamentos assim como capacidade da rede e retardo dos enlaces. É necessário portanto, além do controle fim a fim, um mecanismo que previna o esgotamento de recursos ao longo do caminho entre fonte e destino, ou seja, deve-se evitar que qualquer fonte envie mais tráfego do que um nó pode suportar. Quando isto ocorre, há descarte de pacotes em um determinado nó, e a situação é denominada

congestionamento. Através da utilização de janelas deslizantes com tamanho de janela variável, o protocolo TCP resolve o problema de controle de fluxo fim a fim.

Porém, não há um mecanismo explícito para resolver o controle de congestionamento. Além disso, dependendo da forma de como o TCP é implementado, a reação em situações de congestionamento será muito diferente em cada caso, levando ou a redução do problema ou até ao agravamento da situação.

Foram pesquisados vários mecanismos para o controle de congestionamento e durante anos foram sugeridos ao protocolo TCP. Ainda continua sendo uma área de pesquisa ativa e contínua. As modernas implementações TCP utilizam quatro algoritmos combinados para realizar o controle: *Slow start*, *Congestion avoidance*, *Fast retransmit* e *Fast recovery*.

A figura 24 mostra um típico gráfico exemplo de uma conexão de rede TCP onde se verifica o aumento e a diminuição da janela deslizante e por consequência a respectiva influência na largura de faixa. Normalmente é um gráfico no formato dente-de-serra. Esta flutuação na largura de faixa é um grande problema para as aplicações multimídia, as quais requerem uma largura de faixa consistente.

O protocolo TCP é inapropriado para a transmissão de *streams* de dados *time-sensitive* através da rede, como por exemplo na transmissão de voz e vídeo onde um grande atraso ou mesmo a variação do atraso na entrega dos pacotes, conhecido como *jitter* e uma baixa eficiência (rendimento) do protocolo acarreta em uma perda de desempenho nas aplicações multimídia.

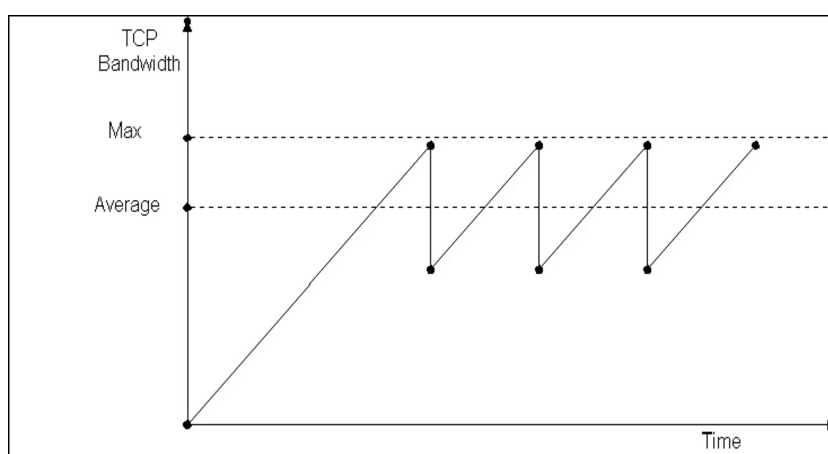


Figura 24 - Largura de faixa em TCP

O cabeçalho do segmento TCP é composto de diversos campos conforme mostrado na figura 25. Para identificar a qual conexão pertencem os dados, o protocolo TCP utiliza os campos *Source Port* (porta fonte) e *Destination Port* (porta destino) presentes no segmento TCP e os endereços IP presentes no cabeçalho do pacote IP. Podemos observar uma violação no relacionamento entre as camadas na medida em que a informação da camada de rede é utilizada na camada de transporte.

Continuando a análise do cabeçalho TCP, temos o campo *Sequence Number* (número de sequência) identifica a posição, dentro do *stream* de dados, relativa ao primeiro *byte* que está sendo enviado ao outro lado da conexão. Já o campo *Acknowledgement Number* (número de reconhecimento) identifica o número do próximo byte que o nó destino espera receber.

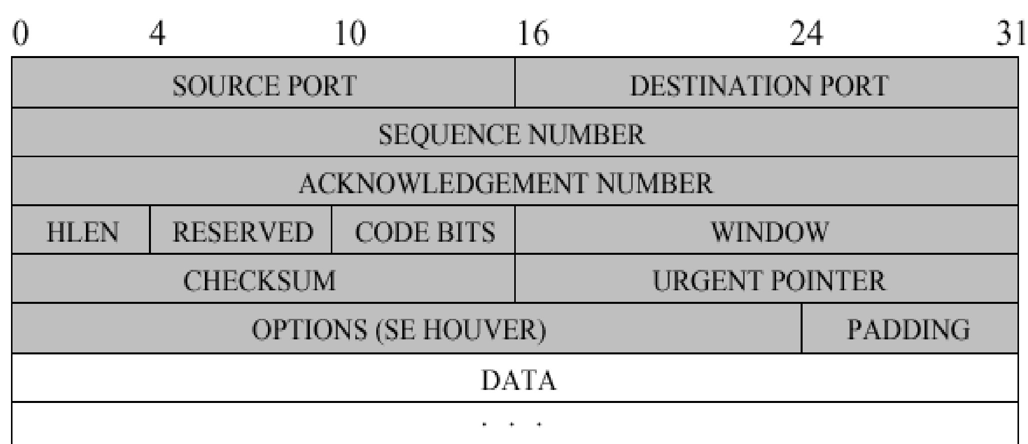


Figura 25 - Cabeçalho TCP

O esquema de reconhecimento do protocolo TCP é cumulativo porque ele anuncia o quanto foi acumulado do *stream* de dados de forma contínua. Sendo assim, se algum segmento for perdido, pode ser que haja “buracos" no *buffer* de recepção, fazendo com que os dados com os bytes de número sequência maiores do que os referentes a lacuna não sejam reconhecidos, apesar de serem armazenados.

Este esquema apresenta vantagens e desvantagens. Dentre as vantagens estão a facilidade de geração (um único número) e o fato de que nem sempre a perda de um reconhecimento provoca retransmissão, pois se um próximo segmento chegar antes que o temporizador se esgote, ele reconhecerá também todos os dados que seriam reconhecidos anteriormente. A grande desvantagem deste esquema reside no fato de que o nó fonte não tem o controle de todas as transmissões realizadas com sucesso. Isto pode acarretar retransmissões desnecessárias ou espera de próximos

reconhecimentos para saber se o nó destino conseguiu receber e armazenar os dados transmitidos. Ambos os esquemas são ineficientes. Este é um dos motivos que reforçam a idéia de introduzir reconhecimentos seletivos no protocolo TCP (*SACK – Selective Acknowledgement*).

O campo HLEN (*Header Length*) contém 4 bits e indica o tamanho do cabeçalho em blocos múltiplos de 32 bits. Isto é preciso porque o tamanho do campo *Options* (opções) pode variar dependendo das opções do protocolo TCP. O campo *Reserved* contém 6 bits e está reservado para uso futuro. O campo *Code Bits* (ou *flags*) contém os bits de código para identificar o propósito dos dados que estão sendo enviados através dos segmentos. A tabela 6 resume o significado destes bits cuja utilização será detalhada a seguir.

Tabela 6 - Flags TCP

URG	Indica que os dados enviados são prioritários e o campo *Urgent Pointer* informa onde terminam os dados urgentes.
ACK	Indica o recebimento com sucesso de um segmento previamente transmitido de acordo com o campo *Acknowledgement Number*.
PSH	Este segmento é repassadado imediatamente para a aplicação, não sendo armazenado no *buffer*.
RST	Finaliza abruptamente a conexão ou a tentativa da mesma.
SYN	Sincroniza os números de sequência.
FIN	Sinaliza o fim da transmissão dos dados (fim da conexão).

O bit URG (urgente) faz com seja possível para o sofware TCP no nó destino sinalizar para a aplicação sobre a presença de dados prioritários. A aplicação então pode entrar em modo urgente e fazer com que os dados enviados no segmento em questão sejam processados tão logo recebidos, e antes de qualquer dado que porventura já tenha sido recebido e esteja armazenado no *buffer* de recepção. Este mecanismo é útil principalmente em aplicações de terminais remotos, onde o usuário pode querer interromper um processo após o envio de muitos bytes, mas sem querer esperar99 pelo processamento de todos estes bytes para que haja a interrupção. O bit URG quando colocado em 1 indica portanto o envio de dados urgentes e é utilizado em conjunto com o campo *Urgent Pointer* (ponteiro urgente), o qual indica o byte onde os dados urgentes terminam.

O bit ACK (reconhecimento) indica que o campo *Acknowledgement Number* é válido, ou seja, o segmento está sendo utilizado para reconhecer dados recebidos referentes a uma conexão.

O bit PSH (*push* | empurrar) faz com que os dados sejam entregues à aplicação assim que sejam recebidos sem esperar pelo enchimento do *buffer* de recepção. Este esquema é muito útil em aplicações interativas, onde o usuário requer um tempo de resposta muito pequeno. Num terminal remoto por exemplo, a aplicação na fonte enviaria um segmento com apenas um byte referente a cada tecla pressionada pelo usuário colocando o valor 1 no bit PSH.

Continuando a análise dos campos do segmento TCP, temos o campo *Window* (janela) que contém 16 bits representando um número que indica o tamanho da janela em bytes. O tamanho máximo possível da janela é de 65536 bytes.

E por último, o campo *Checksum* (soma de checagem) é utilizado para verificar a integridade tanto dos dados quanto do cabeçalho. O processo é feito da mesma forma que no protocolo UDP, onde um pseudo-cabeçalho e zeros são adicionados ao segmento para que todo ele seja múltiplo de 16 bits. Vale frisar que nem o pseudo-cabeçalho nem os zeros adicionados são transmitidos. Após este passo, o software TCP computa o complemento a 1 da soma dos complementos a 1 de todos os blocos de 16 bits do segmento alterado. Neste passo o campo *Checksum* será considerado como contendo somente zeros. No nó destino o software TCP executa o mesmo processo e descarta o pacote em caso de erro (valores diferentes de *checksum*).

O formato do pseudo-cabeçalho de 12 bytes (figura 26) contém os endereços IP fonte e destino (campos *Source IP Address* e *Destination IP Address*) que servem para verificar se o segmento chegou ao destino correto. O campo *Protocol* indica qual o valor do protocolo que a camada inferior irá utilizar no campo de tipo de protocolo (campo *Protocol Type* no datagrama IP) e vale 6 (seis) para o TCP. O campo *TCP Length* especifica o tamanho total do segmento, incluindo o cabeçalho.

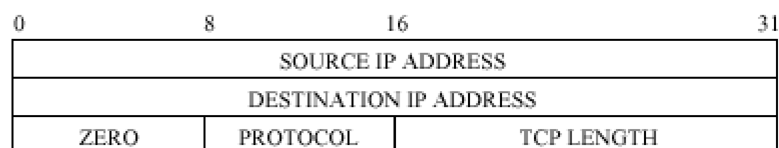


Figura 26- Pseudo-cabeçalho TCP

O processo de abertura de uma conexão TCP utiliza um mecanismo denominado *three-way handshake* contendo 3 passos descritos a seguir:

1. O nó que executou a função de abertura ativa envia um segmento com o bit SYN (*synchronization*) do campo *Code Bits* com valor 1, número da porta destino igual à porta do outro nó remoto com a qual deseja conectar e o número de sequência inicial (*ISN - InitialSequence Number*) x.
2. O nó que espera pela abertura da conexão passivamente responde então com um segmento com o bit SYN com valor 1, número da porta destino igual à porta que o outro ponto informou como porta fonte no segmento recebido, número de sequência inicial y e número de reconhecimento igual a x + 1.
3. O nó que iniciou o processo (passo 1) envia um segmento com número de reconhecimento igual a y+1, reconhecendo portanto o segmento recebido no passo 2. Este segmento não possui o bit SYN com valor 1. Este passo é utilizado apenas para indicar que os dois lados concordam que a conexão foi estabelecida.

Conforme explicado, nada impede que o tamanho de um segmento TCP possa variar ao longo de uma conexão. Porém, deve haver um acordo sobre o tamanho máximo que um segmento pode assumir durante a existência de uma conexão. Para isso, o software TCP em ambos os extremos de uma conexão mantém um valor denominado MSS (*Maximum Segment Size*) limitando o tamanho do segmento.

A escolha do valor MSS tem influência direta no desempenho do protocolo TCP na medida em que valores muito pequenos acarretarão um uso ineficiente da largura de faixa da rede, e valores muito altos podem fazer com que o datagrama IP seja fragmentado ao longo do percurso. Esta fragmentação obrigará a espera de todos os fragmentos para que o software TCP possa enviar o reconhecimento (fragmentos não podem ser reconhecidos em separado). Logo deve-se procurar um ponto ótimo.

Embora importante, a determinação deste valor ótimo nem sempre é simples. Dentro de uma rede local este problema se torna mais simples, sendo possível usar um valor que resulte em um datagrama IP com tamanho igual ao MTU *(Maximum Transfer Unit*) da rede. Caso o caminho envolva várias redes heterogêneas pode-se

tentar descobrir o tamanho ideal ou escolher o valor 536 bytes, o qual resultará em um datagrama IP de tamanho padrão (576 bytes).

Alguns exemplos de aplicações padrões de rede definidas em RFC que utilizam o protocolo de transporte TCP são o FTP, HTTP, SMTP, DNS (transferência de zona), Telnet, SSH e POP.

B.3 - Protocolo UDP

O protocolo UDP (*User Datagram Protocol*) [18] é um protocolo não orientado a conexão e não confiável designado como STD número 6 e a especificação está descrita na RFC 768.

No protocolo TCP, os dados transmitidos são segmentados em pacotes de pequeno tamanho e estes são numerados e transmitidos (e recebidos) em ordem para cada conexão. Em contraposição, no protocolo UDP os pacotes conhecidos como datagramas (ou datagramas de usuário) não são numerados e a entrega na ordem correta dos datagramas às camadas superiores não é assegurada devido aos possíveis caminhos diferentes que os pacotes IP possam utilizar devido ao roteamento e a falta de numeração dos datagramas. Além disso, o recebimento pelo destino (serviço de entrega) não é assegurado, os datagramas podem ser descartados no caminho ao destino que a camada de transporte não terá consciência da falta do mesmo. Os datagramas recebidos são livres de erros mas quem realiza a checagem de erro é a camada de rede IP normalmente.

O protocolo UDP segue o princípio do melhor esforço (*best-effort*), onde os segmentos apenas serão descartados caso haja esgotamento de recursos. Caso haja perda de informação não haverá retransmissão.

O protocolo UDP é basicamente uma interface de aplicação para o protocolo IP que não adiciona nenhuma confiabilidade, controle de fluxo dos dados ou recuperação de erro. Estas funcionalidades devem ser providas pela aplicação do usuário caso assim necessite.

A aplicação que necessita enviar datagramas para um destino final precisa identificar o processo alvo de modo preciso não apenas com o endereço IP pois os datagramas normalmente são direcionados a processos específicos do host e não para o host como um todo. O protocolo UDP, assim como o TCP, fornece esta característica através do uso de portas lógicas.

O protocolo UDP tem como propósito principal simplesmente servir como um *multiplexer/demultiplexer* para enviar e receber datagramas usando as portas para direcionar os datagramas do usuário para o processo específico. O cabeçalho UDP mostrado na figura 27 ratifica o exposto anteriormente.

Cada datagrama UDP é enviado dentro de um único datagrama IP. Embora o datagrama IP possa ser fragmentado durante transmissão, o receptor reorganizará antes de apresentar os datagramas para a camada UDP. Como todas as implementações do protocolo IP são obrigadas a aceitar datagramas de 576 bytes, e que o tamanho máximo do cabeçalho IP é de 60 bytes, um datagrama UDP de 516 bytes tem que ser aceito em todas as implementações. Muitas implementações irão aceitar datagramas maiores mas não está garantido. O datagrama UDP tem um cabeçalho de 16 bytes, o qual está descrito na figura 27.

Source Port	Destination Port
Length	Checksum
Data...	

Figura 27 - Cabeçalho UDP

• *Source Port*: Indica a porta do processo transmissor. Esta porta será utilizada na resposta pelo nó remoto.

• *Destination Port*: Especifica a porta do processo destino no nó remoto.

• *Length*: O tamanho em bytes do datagrama do usuário incluindo o cabeçalho.

• *Checksum*: Um campo opcional de 16 bits do complemento a um da soma do pseudo-cabeçalho, o cabeçalho UDP e os dados. O pseudo-cabeçalho contém o endereço IP fonte e destino, o protocolo e o tamanho do datagrama UDP (figura 28).

Source IP address		
Destination IP address		
Zero	Protocol	UDP Length

Figura 28 - Pseudo-cabeçalho UDP

O pseudo-cabeçalho IP estende o *checksum* efetivamente para incluir o original (não fragmentado) datagrama IP.

A interface de programação da aplicação com o protocolo UDP fornece as seguintes funcionalidades:

- A criação de novas portas de recepção

- A operação de receber dados que retorna os bytes de dados e uma indicação da porta origem e endereço IP origem.
- A operação de transmissão tem como parâmetros: os dados e os endereços IPs e as portas origem e destino.

O protocolo UDP fornece simplesmente um mecanismo para uma aplicação enviar um datagrama para uma outra. A camada UDP pode ser considerada como sendo extremamente leve e por conseguinte tem baixo *overhead*, mas exige que a aplicação tenha a responsabilidade para recuperação em caso de erro conforme mencionado anteriormente.

O protocolo UDP é um protocolo s*tateless* onde não há nenhuma conexão envolvida. Uma máquina envia um pacote a outra e esquece disto, não espera por uma resposta ou algo para informar que o pacote chegou, conseqüentemente é o protocolo "Envia e Reze". O protocolo UDP foi projetado para ser um protocolo de *messaging.* Ele não mantém informação sobre estados mas é completamente possível implementar estados usando UDP, porém o estado deve ser mantido pelo aplicativo no espaço de usuário. Através do espaço de usuário, está referindo-se a tudo fora do *kernel* e da pilha TCP/IP. Os protocolos NFS e TFTP são exemplos excelentes de serviços baseados em UDP que tem informação de estado e realizam verificação de erro.

O protocolo UDP é muito mais adequado para utilização em aplicações multimídia que o protocolo TCP. Ele não faz uso de *acknowledgment tickets* e não retransmite os pacotes. Com UDP, os erros causados pela rede de comunicação não propagam durante a execução de um *stream* multimídia. Por exemplo, se alguns pacotes chegam atrasados a execução do filme não deveria ser atrasada. Ao invés disso, os correspondentes quadros são descartados e a execução não é atrasada. Isto é uma característica muito importante para permitir que conexões de pequena largura de faixa permaneçam em sincronismo com um *stream* de dados conforme ocorra variações na largura de faixa.

A seguir, temos alguns exemplos de aplicações padrões definidas em RFCs que utilizam o protocolo da camada de transporte UDP:

- *Trivial File Transfer Protocol* (TFTP)
- *Domain Name System* (DNS)

- *Remote Procedure Call* (RPC)
- *Network File System* (NFS)
- *Simple Network Management Protocol* (SNMP)
- *Syslog Protocol*
- *Connectionless Lightweight Directory Acess Protocol* (CLDAP)
- *Real Time Streaming Protocol* (RTSP)
- *Real Time Transport Protocol* (RTP)
- *Instant Messaging and Presence Protocol* (IMPP)
- *Peer To Peer* (P2P)
- *Session Initiation Protocol* (SIP - VoIP)
- H-323 (VoIP)
- *Network Time Protocol* (NTP)
- *Simple Network Time Protocol* (SNTP)

B.4 - Protocolo SCTP

Introdução

O protocolo SCTP, *Stream Control Transmission Protocol*, é um novo protocolo de transporte da pilha IP. E pode ser comparado ao tradicionais UDP e TCP, que disponibilizam funções na camada de transporte para muitas aplicações Internet. SCTP foi aprovado como uma RFC "*Proposed Standard*", número 2960 e depois foram acrescentadas algumas atualizações e correções nas RFCs 3286, 3309, 3436 e 3554. Existe um grupo de trabalho IETF, *Transport Area Working Group*, preocupado em aperfeiçoar este e outros protocolos da camada de transporte.

De forma similar ao TCP, o protocolo SCTP fornece um serviço de transporte confiável, assegurando que os dados são transportados através da rede sem erro e em sequência. E também é orientado a sessão tal como o TCP.

O protocolo SCTP, diferentemente do TCP, disponibiliza funções que são críticas para o transporte da sinalização telefônica e ao mesmo tempo pode beneficiar outras aplicações que necessitam de uma camada de transporte confiável e com diferencial de desempenho.

Características básicas:

O protocolo SCTP é *unicast*. Suporta a troca de dados entre somente dois *endpoints*, embora estes possam ser representados através de múltiplos endereços IP. O protocolo disponibiliza uma transmissão confiável, detectando quando dados são descartados, reordenados, duplicados ou corrompidos, e retransmite os dados corrompidos conforme necessário. E a transmissão SCTP é *full-duplex*. O protocolo SCTP é orientado a mensagem e suporta enquadramento dos limites de uma mensagem individual. Em comparação, o protocolo TCP é orientado a byte e não preserva nenhuma estrutura implícita dentro de um *stream* de bytes transmitidos. SCTP suporta taxa de transmissão adaptativa semelhante para TCP a qual depende das condições de carga na rede.

A função multi-streaming

O nome *Stream Control Transmission Protocol* (SCTP) é derivado da função *multi-streaming* disponibilizada pelo SCTP. Esta característica permite particionar os dados da aplicação em múltiplos fluxos os quais têm a propriedade da independência na entrega sequenciada, isto é, a perda de uma mensagem em qualquer um dos fluxos afetará inicialmente somente a entrega das mensagens daquele fluxo e não entrega para os outros fluxos.

Em contraste, o TCP assume um único fluxo de dados e assegura que a entrega deste fluxo acontecerá com a preservação do sequenciamento original dos bytes. Embora isto seja desejável para a entrega de um arquivo ou registro, ela causa um atraso adicional quando acontece uma perda de mensagem ou erro no sequenciamento dos pacotes na rede. Quando isto acontece, o TCP tem atrasar a entrega de dados até que o correto sequenciamento seja restabelecido através do recebimento de uma mensagem fora da sequência ou pela retransmissão da mensagem perdida.

Para várias aplicações, a característica de preservação rígida do sequenciamento não é verdadeiramente necessária. Na sinalização telefônica, é só necessário manter o sequenciamento das mensagens que afetam o mesmo recurso (por exemplo: a mesma chamada ou o mesmo canal). Outras mensagens fracamente correlacionadas ou sem correlação podem ser entregues sem a necessidade de manter a integridade da sequência global.

Outro exemplo de um possível uso do *multi-streaming* é a entrega de documentos multimídia, como uma página web, quando realizado sobre uma única sessão. Considerando que documentos multimídia consistem de objetos de tipos e tamanhos diferentes, o *multi-streaming* permite que o transporte destes componentes seja parcialmente ordenado em lugar do estritamente ordenado, e pode resultar em uma melhor percepção de usuário em relação ao desempenho da camada de transporte e por conseguinte da aplicação.

O transporte é realizado dentro de uma única associação SCTP, de forma que todos os fluxos são sujeitos a um fluxo comum e ao mecanismo de controle de congestionamento, reduzindo o *overhead* requerido na camada de transporte.

O protocolo SCTP realiza *multi-streaming* através da criação de uma relação de independência entre a transmissão e a entrega dos dados. Em particular, cada

chunk de dados no protocolo usa dois conjuntos de números de sequência: um número de sequência de transmissão que controla a transmissão das mensagens e a descoberta da perda de mensagens, e o par *Stream ID*/Número do *Stream* na Sequência que é usado para verificar o sequenciamento da remessa dos dados recebidos.

Esta independência dos mecanismos permite ao receptor determinar imediatamente quando uma lacuna na sequência de transmissão ocorre (por exemplo: devido a perda de uma mensagem) e também permite verificar se as mensagens recebidas após a interrupção pertencem a um fluxo afetado. Se uma mensagem é recebida dentro de um fluxo afetado, haverá uma correspondente lacuna no número de sequência do fluxo, enquanto as mensagens dos outros fluxos não mostrarão a lacuna. O receptor pode continuar a entregar as mensagens aos fluxos não afetados enquanto serão armazenadas em *buffers* (*buffering*) as mensagens para o fluxo afetado até que a retransmissão aconteça.

Segurança

Em adição aos mecanismos de *verification tag* e *cookie*, o protocolo especifica o uso do IPSec caso uma forte segurança e proteção a integridade seja requerida. A especificação do protocolo SCTP não define nenhum novo procedimento ou protocolo de segurança. **Existe também trabalhos em progresso para fazer uso do TLS, *Transport Layer Security*, sobre SCTP.**

B.5 - Protocolo IP

O protocolo de rede IP (Internet Protocol) [19], atualmente na versão 4, é um dos componentes do STD número 5 e tem sua especificação descrita em várias RFCs, entre elas: 791, 950, 919, 922 e atualizada na RFC 2474. É o protocolo principal da pilha de protocolos TCP/IP.

O protocolo IP está situado na camada 3 (rede) da pilha TCP/IP e não é orientado a conexão. Tem como funcionalidade principal possibilitar o roteamento dos pacotes entre *hosts* através da inserção do campo de endereçamento da fonte e do destino, e assim fornecendo um serviço de entrega *best-effort* dos datagramas através de uma nuvem IP. Executa também a fragmentação e remontagem da unidade de informação (datagrama IP) com diferentes tamanhos máximos (MTU) em redes heterogêneas.

O protocolo IP, assim como o UDP, não fornece serviço de entrega assegurado, nem controle de fluxo dos dados ou mesmo mecanismo para recuperação de erros de transmissão.

O datagrama IP contém vários tipos de informação conforme ilustrado na figura 29. A seguir estão descritos os diversos campos presentes no cabeçalho IP.

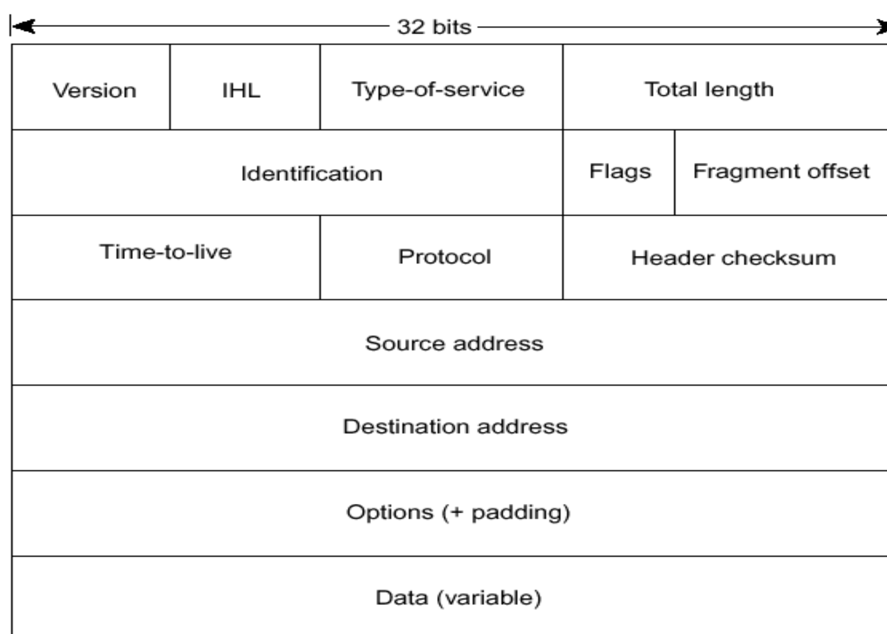


Figura 29 - Cabeçalho IP

- *Version*: Indica a versão do protocolo IP utilizado.
- *IP Header Length (IHL)*: Indica o tamanho do cabeçalho do datagrama em palavras de 32 bits.
- *Type-of-Service*: Especifica como o protocolo de camada superior recomenda que fosse tratado o datagrama e assinala vários níveis de importância ao mesmo.
- *Total Length*: Especifica o tamanho em bytes do datagraam IP inteiro, incluindo os dados e o cabeçalho.
- *Identification*: Contém um número inteiro que identifica o datagrama. Este campo auxilia a remontar os fragmentos de um datagrama.
- *Flags*: Consiste de um campo de 3 bits dos quais os dois bits menos significativos são para controle da fragmentação. O bit menos significativo especifica se o datagrama foi fragmentado. O bit intermediário especifica se o datagrama é o último fragmento da série de datagramas fragmentados. O bit mais significativo não é usado.
- *Fragment Offse*t: Indica a posição relativa dos dados do fragmento em relação ao ínicio dos dados no datagrama original, o qual permite ao processo IP destino apropriadamente reconstruir o datagrama original.
- *Time-to-Live:* Mantém um contador que decrementa de 1 a cada hop que o datagrama atravessa até chegar a zero, ponto no qual o datagrama é descartado. Este campo é utlizado para evitar loops de roteamento infinitos.
- *Protoco*l: Indica qual camada superior receberá os datagramas após o processamento IP ser completado.
- *Header Checksu*m: Utilizado para assegurar a integridade dos dados presentes no cabeçalho IP.
- *Source Address:* Especifica o nó transmissor.
- *Destination Addres*s: Especifica o nó receptor.
- *Option*s: Permite ao protocolo suportar várias opções, tal como segurança.
- *Dat*a: Contém os dados da camada superior.

Processo de Fragmentação

Quando um datagrama IP é encaminhado de um host a outro, pode atravessar diferentes redes físicas. Cada rede física tem um tamanho máximo de quadro conhecido como a unidade máxima de transmissão (MTU). Ele limita o tamanho do datagrama que pode ser colocado em uma rede física.

O protocolo IP implementa um processo para fragmentar datagramas que excedem o MTU. O processo cria um conjunto de datagramas dentro do tamanho máximo permitido. O host receptor remonta o datagrama original a partir dos fragmentos. O protocolo IP requer que cada enlace suporte o MTU mínimo de 68 octetos. Este valor é a soma do tamanho máximo do cabeçalho IP (60 octetos) e a quantidade mínima de dados em um fragmento (8 octetos). Se alguma rede tiver um valor mais baixo que este, a fragmentação e a remontagem deve ser implementada nas camadas inferiores (física e enlace). E tem que ser transparente a camada IP.

Não são exigidas que implementações do protocolo IP tratem datagramas não fragmentados maiores que 576 bytes. Na prática, a maioria das implementações suportam valores maiores de tamanho do datagrama.

Um datagrama não fragmentado tem colocado em zero todos os bits dos campos referentes a informação de fragmentação. Isto é, o bit de sinalização (*flag*) de fragmentação é colocado em zero e o *fragment offset* é colocado em zero também. Os seguintes passos são executados para fragmentar um datagrama:

- O flag bit DF é conferido para verificar se a fragmentação é permitida. Se o bit está em 1, o datagrama será descartado e uma mensagem de erro ICMP é retornada a origem.
- Baseado no valor da MTU, o campo de dados é dividido em duas ou mais partes. Todas as porções de dados recentemente criadas têm que ter um tamanho que seja múltiplo de 8 octetos, com a exceção da última porção de dados.
- Cada porção de dados é encapsulada em um datagrama IP. O cabeçalho destes novos datagramas são praticamente igual ao original apenas com algumas modificações necessárias, descritas a seguir:

  1- O bit *flag* de fragmentação é colocado em 1 (*set*) em todos os fragmentos com exceção do último.

2- No campo *fragment offset* de cada fragmento é assinalado a localização desta porção de dados ocupada no datagrama original, relativo ao começo do datagrama original não fragmentado. O *offset* é definido em unidades de 8 octetos.

3- Se foram incluídas opções no datagrama original, o bit mais significativo do campo *options* determina se esta informação é copiada a todos os fragmentos ou somente ao primeiro. Por exemplo, as opções de "*source route*" são copiadas em todos os fragmentos.

4- O campo de tamanho do cabeçalho e o campo de tamanho total do novo datagrama são ajustados.

5- E o campo de checksum do cabeçalho é recalculado.

Cada um deste datagramas fragmentados são encaminhados agora como um datagrama IP normal. O protocolo IP trata cada fragmento independentemente. Os fragmentos podem atravessar diferentes roteadores até chegar ao destino pretendido. Eles podem ser inclusive fragmentados novamente se atravessarem redes com MTU menor que o tamanho do fragmento.

No host destino, os fragmentos são remontados no datagrama original. O campo de identificação fixado pelo host transmissor é usado juntamente com o endereço IP fonte e destino no datagrama para ajudar a remontagem. A fragmentação não altera estes campos.

Para remontar os fragmentos, o host receptor aloca um buffer quando o primeiro fragmento chega. O host também inicializa um temporizador. Quando fragmentos subseqüentes do datagrama chegam, os dados são copiados para o buffer na localização indicada pelo campo *fragment offset*. Quando todos os fragmentos chegam, o datagrama original é restabelecido e o processamento continua como se fosse um datagrama não fragmentado.

Se o temporizador exceder o tempo máximo definido e ainda existam fragmentos pendentes, o datagrama é descartado. O valor inicial definido para o temporizador é chamado de tempo de vida (*time-to-live,* TTL) do datagrama IP. É dependente da implementação. Algumas implementações permitem que seja configurado este valor.

Apêndice C – Blocos Criptográficos

Neste apêndice serão estudados os diversos blocos criptográficos utilizados no projeto de um protocolo de criptografia como exemplo temos TLS e o IPSec.

Uma análise básica dos diferentes blocos de criptografia será realizada para também ser utilizada como uma ferramenta de apoio na discussão dos protocolos de segurança de rede estudados nesta dissertação de Mestrado.

C.1- Função Hash

Uma função hash H é uma transformação onde uma entrada m é processada e retorna uma string de tamanho fixo, o qual é chamada de hash h, assim sendo: h = H(m). Funções de hash com esta propriedade tem uma variedade de usos computationais, mas quando empregadas em criptografia as funções de hash normalmente tem algumas propriedades adicionais.

As exigências básicas para uma função de hash criptográfica são as seguintes:

- A entrada da função pode ter qualquer tamanho.
- A saída da função tem um tamanho fixo.
- H(x) é relativamente fácil de computar para qualquer valor m.
- H(x) é uma função de mão única (*one-way*).
- H(x) é livre de colisão.

Uma função de hash é de mão única se a sua inversão é difícil, isto é, significando que a partir de um valor h de hash seja impraticável computacionalmente encontrar alguma entrada m tal que H(m) = h. Dada uma mensagem m, se for computacionalmente possível achar uma mensagem n diferente de m tal aquele H(m) = H(n), então é dito que H é uma função de hash fracamente livre de colisão. Uma função de hash fortemente livre de colisão H é aquela que é computacionalmente improvável achar duas mensagens m e n, tal aquele H(m) = H(n).

O valor de hash representa concisamente uma mensagem ou documento mais longo do qual ele foi computado. Este valor é chamado de sumário (ou resumo) da

mensagem (*message digest*). Podemos pensar em um sumário de mensagem como uma impressão digital de um documento maior. Exemplos de funções de hash comumente conhecidas são MD5 e SHA.

A função de hash criptográfica tem duas principais funcionalidades: possibilitar a checagem da integridade de uma mensagem e facilitar a implementação de assinaturas digitais. Como as funções de hash são geralmente mais rápidas que encriptação ou algoritmos de assinatura digitais, é comum computar a assinatura digital ou checagem de integridade para algum documento aplicando o procedimento criptográfico desejado ao valor de hash do documento o qual é pequeno comparado ao próprio documento. Adicionalmente, um sumário pode ser público sem revelar os conteúdos do documento do qual foi derivado. Isto é importante em *digital timestamping*, onde usando funções de hash, a pessoa pode adquirir *document timestamped* sem revelar seu conteúdo ao serviço de *timestamping*.

Damgard e Merkle [20, 21] influenciaram grandemente a criação das funções de hash criptográficas definindo uma função de hash em termos do que é chamado uma função de compressão. Uma função de compressão recebe uma entrada de tamanho fixo e retorna uma saída de tamanho fixo de menor tamanho. Dado uma função de compressão, uma função de hash pode ser definida como repetidas execuções da função de compressão até que a mensagem inteira seja processada. Neste processo, uma mensagem de tamanho arbitrário é dividida em blocos de tamanho dependente da função de compressão e também são *padded* (por razões de segurança) de maneira que o tamanho da mensagem seja múltiplo do tamanho de bloco. Então os blocos são processados consecutivamente e recebem como entrada o resultado do hash parcial anterior e o bloco da mensagem atual, tendo como saída final o valor do hash (veja figura 30, F é uma função de compressão).

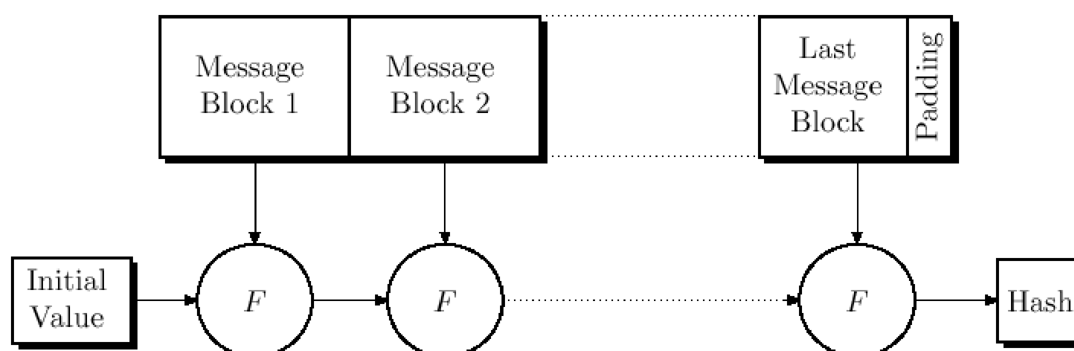


Figura 30 - Estrutura iterativa de Damgard/Merkle para funções de hash

C.1.1 - SHA e SHA-1

O algoritmo de hash seguro (Secure Hash Algorithm - SHA) foi especificado no Secure Hash Standard (SHS, FIPS180) e foi desenvolvido pelo NIST [22]. O algoritmo de hash SHA-1 é uma revisão do SHA original que foi publicado em 1994. A revisão corrigiu uma falha não publicada do SHA.

O projeto deste algoritmo foi muito semelhante ao MD4, uma função de hash desenvolvida por Rivest da RSA Labs. SHA-1 também está descrito no padrão ANSI X9.30 (parte 2). O algoritmo recebe como entrada uma mensagem de tamanho menor que 2^{64} bits e produz um sumário de mensagem de 160 bits.

O algoritmo é um pouco mais lento que o MD5, mas como o sumário da mensagem é maior, este algoritmo é menos vulnerável ao ataque por colisão via força-bruta e ao ataque de inversão. SHA é parte integrante do projeto Capstone.

C.1.2 - MD2, MD4 e MD5

MD2, MD4 e MD5 são algoritmos de hash (*message digest*) desenvolvidos por Rivest. Eles são destinados para aplicações de assinatura digitais onde uma mensagem grande tem que ser comprimida de uma maneira segura antes de ser assinada com a chave privada. Todos os três algoritmos recebem como entrada uma mensagem de tamanho arbitrário e produzem um sumário de mensagem de 128 bits. Enquanto as estruturas destes algoritmos são um pouco semelhantes, o projeto do MD2 é bastante diferente do MD4 e do MD5.

O MD2 foi otimizado para máquinas de 8 bits e o MD4 e MD5 foram projetados para máquinas 32 bit. A especificação e código fonte dos três algoritmos podem ser encontradas na Internet, RFCs 1319-1321.

O MD2 foi desenvolvido por Rivest em 1989. A mensagem é primeiramente *padded* de modo que o tamanho em bytes seja divisível por 16. Um *checksum* de 16 bytes é acrescentado a mensagem e o valor do hash é computado a partir da mensagem resultante. Rogier e Chauvaud informaram que podem ser construídas colisões para o MD2 caso o cálculo do *checksum* for omitido. Esta é a única criptoanálise conhecida para MD2.

O MD4 foi desenvolvido por Rivest em 1990. A mensagem é *padded* para assegurar que seu tamanho em bits mais 64 seja divisível por 512. Uma representação binária de 64 bits do tamanho original da mensagem é então concatenada à mensagem. A mensagem é processada em blocos de 512 bits na estrutura iterativa de Damgard/Merkle e cada bloco é processado em três *rounds* distintos. Ataques em versões de MD4 que não tinham o primeiro ou o último *round* foram desenvolvidos muito rapidamente por Den Boer, Bosselaers e outros. Dobbertin mostrou como colisões para a versão completa do MD4 podem ser encontradas em apenas um minuto em um computador PC típico. Em recente trabalho, Dobbertin (Fast Software Encryption, 1998) mostrou que uma versão reduzida do MD4 na qual o terceiro round da função de compressão não é executado mas todo o resto permanece o mesmo, não é uma função de mão única. Claramente, o MD4 deve ser considerado quebrado (*broken*).

O MD5 foi desenvolvido por Rivest em 1991. É basicamente o MD4 com "cinto de segurança" e embora seja um pouco mais lento que o MD4, é mais seguro. O algoritmo consiste em quatro *rounds* distintos com o projeto um pouco diferente do MD4. O tamanho do sumário, como também as exigências de *padded* permaneceram os mesmos. Den Boer e Bosselaers acharam pseudo-colisões para o MD5. Mais recente trabalho realizado por Dobbertin estendeu as técnicas usadas tão efetivamente na análise de MD4 para achar colisões para a função de compressão do MD5.

Van Oorschot e Wiener criaram um mecanismo de procura de colisões por força bruta em funções de hash, e eles estimaram uma máquina de procura por colisão especificamente projetada para o MD5 (valendo $10 milhões em 1994) poderia achar uma colisão para MD5 em 24 dias em média. As técnicas podem ser aplicadas a outras funções de hash.

C.1.3 - Ataques as Funções de *Hash*

As propriedades criptográficas essenciais de uma função de hash como descrito anteriormente são a ausência de colisão e a irreversibilidade. O ataque mais simples que pode ser feito em uma função de hash é escolher entradas ao acaso até que se encontre um valor de saída desejado (contrariando a propriedade de

irreversibilidade) ou encontrarmos duas entradas que produzem a mesma saída (contrariando a propriedade de ser livre de colisão).

Suponha que a função de hash produza uma saída longa de n bits. Se estamos tentando achar algumas entradas que produzam um determinado valor de saída y, e desde que cada saída seja igualmente provável, teremos então um número de entradas possíveis na ordem de 2^n.

Um ataque de aniversário é o nome usado para se referir a uma classe de ataques por força bruta. Se alguma função, quando recebe uma entrada aleatória, devolve um dos k valores igualmente prováveis, então avaliando repetidamente a função para diferentes entradas, esperamos obter a mesma saída após aproximadamente $1.2k^{1/2}$ tentativas.

Se estamos tentando achar uma colisão, pelo paradoxo do aniversário, obteremos alguma colisão após $1.2(2^{n/2})$ tentativas possíveis de entrada. Van Oorschot e Wiener mostraram como um ataque por força bruta poderia ser implementado.

Com respeito ao uso de funções de hash na provisão de assinaturas digitais, Yuval propôs a seguinte estratégia baseada no paradoxo de aniversário onde n é o tamanho do sumário de mensagem:

- O adversário seleciona duas mensagens: a mensagem alvo a ser assinada e uma mensagem inócua que a Alice queria assinar.
- O adversário gera $2^{n/2}$ variações da mensagem inócua (que tenham o mesmo significado) e os correspondentes sumários de mensagem. E gera um número igual de variações da mensagem alvo e seus sumários.
- A probabilidade de que uma das variações da mensagem inócua tenha o mesmo sumário que uma das variações da mensagem alvo são maiores que 1/2 de acordo com o paradoxo de aniversário.
- Então o adversário obtém a assinatura de Alice através da variação da mensagem inócua.
- A assinatura da mensagem inócua é removida e substituída pela mensagem alvo que gera o mesmo sumário de mensagem. E assim o adversário forjou com sucesso a mensagem sem descobrir a chave de encriptação.

As pseudo-colisões são colisões obtidas na função de compressão que é o componente principal de uma função de hash. Embora colisões em uma função de compressão de uma função de hash possam ser utilizadas na obtenção de colisões para a própria função de hash, isto normalmente não é verdade. Embora as pseudo-colisões possam ser vistas como uma propriedade de cunho negativo em uma função de hash, uma pseudo-colisão não é equivalente a uma colisão na função de hash e a função de hash ainda pode ser considerada ainda como razoavelmente segura, entretanto seu uso para novas aplicações tende a ser desencorajado a favor de funções de hash livres de pseudo-colisões. MD5 é um tal exemplo de função de hash livre de pseudo-colisões.

C.2 - Algoritmos Simétricos

Os algoritmos simétricos, ou de chave secreta, utilizam uma única chave que é compartilhada entre ambos os lados da conexão para encriptar e desencriptar as mensagens trocadas na comunicação.

Os algoritmos simétricos não são utilizados somente para encriptação dos dados mas também em mecanismos de autenticação e como um exemplo desta técnica temos *Message Authentication Code* (MAC).

O processamento computacional é executado muito mais rapidamente quando comparado com os algoritmos assimétricos. Esta é uma propriedade desejável para um desempenho satisfatório de sistemas em tempo real. A velocidade de encriptação possibilita ao servidor tratar múltiplas conexões e a velocidade de decriptação permite ao cliente a obtenção dos dados em tempo real. Além disso, a velocidade de decriptação poderia ser diminuída em troca de uma encriptação mais rápida, quando por exemplo a largura de faixa é o gargalo de um cliente e não a decifração do código pela CPU.

O principal problema com os algoritmos de chave simétricos reside no mecanismo de distribuição da chave secreta entre o receptor e o transmissor de modo que ninguém mais tenha acesso a chave.

Para os algoritmos simétricos existem as seguintes técnicas para cifragem dos dados: a cifragem de blocos, a cifragem de fluxo (*stream*) e o *Message Authentication Code* (MAC).

C.2.1 - Algoritmos para Cifragem de Fluxo

Embora se utilizem as cifras para blocos de dados de tamanho variáveis, fluxos (*stream*) de dados podem ser codificados de maneira diferenciada. As cifragens de bloco operam em blocos de grande tamanho enquanto as cifragens de fluxo operam em blocos de dados de pequeno tamanho. As cifragens de fluxo não sofrem de problemas de análise de tráfego como as cifragens de bloco de tamanho fixo.

O algoritmo de cifragem de fluxo (ou algoritmo de fluxo) produz o que é chamado de *keystream*, o qual representa uma sequência de bits usado como uma chave simétrica. A encriptação é executada pela combinação do *keystream* com o texto puro, usualmente com a operação XOR bit a bit.

Há dois tipos de cifragem de fluxo: as síncronas e as auto-sincronizáveis. Os diferentes tipos são diferenciadas pela técnica utilizada para gerar o *keystream* usado para encriptar o fluxo de dados.

- Síncronas: o *keystream* gerado é independente do texto encriptado e do texto puro.
- Auto-sincronizáveis: o *keystream* gerado depende do texto encriptado e/ou do texto puro.

Os algoritmos de fluxo são muito mais rápidos que os algoritmos de blocos. Enquanto os algoritmos de blocos operam em blocos de dados de tamanho grande, os de fluxo tipicamente operam em unidades menores de texto puro.

Como exemplo de alguns algoritmos de fluxo temos: RC4, SEAL e VRA. O algoritmo de fluxo mais amplamente utilizado é o RC4.

C.2.1.1 - One-Time Pad

É um algoritmo de chave privada na qual a chave é uma sequência verdadeiramente aleatória de bits (*keystream*) tão longa quanto a própria mensagem, e a encriptação é executada por uma operação lógica XOR da mensagem com a chave. É teoricamente inquebrável. O interesse atual em algoritmos de cifragem de fluxo é atribuído comumente ao apelo teórico das propriedades do One-time Pad.

Um One-time Pad, às vezes chamado de algoritmo de Vernam, utiliza um *string* de bits que são gerados completamente ao acaso. O *keystream* tem o mesmo tamanho da mensagem de texto puro (mensagem). O *string* aleatório (*keystream*) é combinado através da operação lógica XOR bit a bit com o texto puro para produzir o texto cifrado. Considerando que o *keystream* inteiro é aleatório, até mesmo um oponente com recursos computacionais infinitos não poderá obter o texto puro original mesmo tendo o texto cifrado completo. Diz-se que tal técnica fornece segredo perfeito e a análise do One-time Pad é vista como um das bases de criptografia moderna. O One-time Pad foi utilizado durante o tempo da guerra nas comunicações diplomáticas que requeriam segurança excepcionalmente alta.

Como a chave secreta (que pode somente ser usada uma vez) tem o mesmo tamanho da mensagem a ser transmitida, este algoritmo introduz problemas graves de administração das chaves. Embora seja perfeitamente segura, One-time Pad é de modo geral inviável. Foram desenvolvidos algoritmos de fluxo que são uma aproximação a ação do One-time Pad. Embora os algoritmos de fluxo atuais estejam impossibilitadas de fornecer a segurança teórica satisfatória do One-time Pad, eles são pelo menos práticos.

C.2.1.2 - RC4

O algoritmo de fluxo RC4 foi projetado por Rivest para RSA Data Security (agora RSA Security). É um algoritmo de fluxo com tamanho de chave variável e operações orientadas a byte. O algoritmo está baseado no uso de permutações aleatórias. Análises demonstram que o período da cifragem no qual o algoritmo permanece provavelmente forte é maior que 10^{100}. São requeridas de oito a dezesseis operações de máquina para cada byte de saída e é esperado que o algoritmo execute muito rapidamente em software. É considerado um algoritmo seguro.

O RC4 é usado para encriptação de arquivos em produtos da própria empresa RSA. Normalmente é uma das opções comumente disponíveis nas comunicações seguras, como exemplo: na encriptação do tráfego de/para sites *webs* seguros que utilizam o protocolo SSL/TLS.

C.2.1.3 - Ataques aos Algoritmos Simétricos de Fluxo

O uso mais comum de um algoritmo de fluxo para encriptação é gerar um *keystream* de modo que isso dependa da chave secreta e então combiná-lo (usando XOR) com a mensagem a ser codificada. É crucial que o *keystream* pareça aleatório, isto é, depois de verificar quantidades crescentes do *keystream*, um adversário não deveria ter nenhuma vantagem adicional que possibilite predizer qualquer bit subseqüente da sequência. Enquanto há algumas tentativas para garantir esta propriedade de modo analítico, a maioria dos algoritmos de fluxo confiam em uma análise *ad hoc*.

Uma condição necessária para um algoritmo de fluxo ser seguro é que passe em uma bateria de testes estatísticos que avaliem, entre outras coisas, as freqüências com que padrões sucessivos de bits de diferentes tamanhos ocorrem. Estes testes também podem aferir a correlação entre bits na sequência em algum momento. Claramente a quantidade de testes estatísticos dependerá da perfeição do projetista. É muito raro (e muito fraco) o algoritmo de fluxo que não passe na maioria dos conjuntos de testes estatísticos.

Um *keystream* pode ter fraquezas estruturais potenciais que permitam um adversário deduzir o *keystream*. Obviamente, se o período de um *keystream*, isto é, o número de bits no *keystream* antes que este comece a repetir novamente, é muito pequeno, o adversário pode utilizar as partes reveladas do *keystream* para ajudar na decriptação de outras partes do texto cifrado. O projeto de um algoritmo de fluxo deveria ser acompanhado por uma garantia de um período mínimo para o *keystream* gerado ou por uma evidência teórica boa para o valor mínimo do período. Sem isto, não pode ser assegurado ao usuário do criptosistema que um determinado *keystream* não repetirá tão logo de modo a tornar vulnerável a segurança da criptografia utilizada.

Um conjunto de fraquezas estruturais mais interessantes pode gerar a oportunidade de encontrar modos alternativos para gerar parte ou até mesmo todo o *keystream*. Uma das técnicas mais importante entre estas aproximações é a que usa um registrador de deslocamento com realimentação linear para reproduzir parte da sequência. A motivação para usar o registrador de deslocamento com realimentação linear é devida ao algoritmo de Berlekamp e Massey que recebe como entrada uma

sequência finita de bits e gera como saída um registrador de deslocamento com realimentação linear que poderia ser usado para gerar aquela sequência. Isto originou uma medida de segurança conhecida como a Complexidade Linear de uma Sequência. Para uma determinada sequência, a complexidade linear é o tamanho do registrador de deslocamento com realimentação linear que precisa ser usado para reproduzir a sequência. Claramente uma condição necessária para a segurança do algoritmo de fluxo é que a sequência que produza tenha uma alta complexidade linear.

Outro tipo de ataque tenta recuperar parte da chave secreta que foi usada. Desconsiderando o mais óbvio ataque de procura da chave por força bruta, uma classe poderosa de ataques pode ser descrita pelo termo "Divida e conquiste". Durante uma análise *offline* o criptoanalista identifica alguma parte da chave que tem efeito direto e imediato em algum aspecto ou componente do *keystream* gerado. Executando uma procura por força bruta sobre esta parte menor da chave secreta e observando como as sequências geradas casam com o verdadeiro *keystream*, o criptoanalista pode deduzir potencialmente o valor correto desta pequena fração da chave secreta. Esta correlação entre o *keystream* produzido (depois de fazer alguma suposição para dividir a chave) e o *keystream* interceptado é denominada como Ataque de Correlação e é muito rápido e eficiente.

Algumas considerações sobre a implementação: (i) um algoritmo de fluxo síncrono permite um adversário mudar bits do texto puro sem qualquer propagação de erro para o resto da mensagem. Se a autenticação da mensagem que está encriptada é requerida, a utilização de um MAC é aconselhável; (ii) uma outra questão importante é a sincronização entre emissor e receptor que às vezes pode ser perdida em algum momento da execução do algoritmo de fluxo e algum método é necessário para assegurar que o *keystream* possa ser colocado em sincronismo novamente. Um modo típico de realizar isto é fazendo com que o emissor da mensagem intercale marcadores de sincronização na transmissão de maneira que somente a parte da transmissão que está posicionada entre os marcadores de sincronização será perdida. Contudo este procedimento pode trazer alguma vulnerabilidade na segurança do algoritmo.

C.2.2 - Algoritmos para Cifragem de Bloco

Um algoritmo de bloco é um tipo de algoritmo de encriptação de chave simétrica que transforma um bloco de tamanho fixo de texto puro (texto não encriptado) de dados em um bloco encriptado de dados de mesmo tamanho.

Para muitos algoritmos de bloco, o tamanho de bloco é de 64 bits. Conforme os processadores estão ficando mais sofisticados, o tamanho de bloco está aumentando proporcionalmente, inclusive já caminhamos para blocos de 128 bits.

Como blocos de texto puro diferentes são mapeados para blocos de texto encriptado diferentes (para permitir única decriptação), um algoritmo de bloco efetivamente provê uma permutação (um para um) do conjunto de todas as possíveis mensagens. A permutação efetuada durante qualquer encriptação é obviamente secreta, desde que é uma função da chave secreta.

Cifragem de Blocos de Tamanho Fixo

Estas cifras operam em blocos de tamanho fixo. A chave de encriptação e o texto puro são divididos em partes de tamanho diferentes e são passados por múltiplos circuitos de operações reversíveis. Por exemplo, a cifra poderia ser uma operação XOR da metade superior da chave com a metade superior do texto puro. E o mesmo procedimento seria realizado para as metades inferiores e após executaria alguma operação de deslocamento de bits (*bit shifting*) e repetiria esta operação múltiplas vezes (*rounds*).

Este tipo de cifra fornece uma função 1-1 entre o texto puro e o texto codificado. Então, por exemplo, "AAAA" será sempre encriptado para o mesmo bloco, digamos "BDEF". E isto é uma falha de segurança porque se consegue facilmente o inverso da função, uma vez que sejam descobertos alguns pares de mapeamento do texto puro para cifrado. E esta cifra sofre do ataque de análise de tráfego por causa da propriedade 1-1. Por exemplo, em uma gravação da voz terá sempre muitas instâncias de silêncio, tais instâncias terão o mesmo texto cifrado e serão fáceis de descobrir.

Cifragem de Blocos de Tamanho Arbitrário

Para que a cifragem de bloco de tamanho fixo possam operar em blocos de diferentes tamanhos, o texto puro é acrescentado de bits (*padding*) para alcançar o tamanho mínimo ou é dividido em múltiplos blocos, caso exceda o tamanho máximo. Assim, os blocos são encriptados usando uma cifragem para blocos de tamanho fixo. Este método (*padding*) ajuda a evitar o ataque por análise de tráfego mencionado no item anterior que tratava da cifragem de blocos de tamanho fixo.

Uma técnica utilizada para possibilitar a operação com blocos de tamanho arbitrário e principalmente superar as fraquezas da cifragem de blocos de tamanho fixo é o encadeamento (*chaining*) das cifras de bloco fixo usando diferentes modos de operação. O principal objetivo desejado com estes modos de operação é aumentar a difusão (diminuir a correlação) das estatísticas do texto puro. Estes modos fazem uso de alguns dados históricos (prévio bloco de texto puro ou prévio bloco de texto cifrado ou contador) para prevenir a repetição de padrões de texto cifrado. E assim ajudar a evitar o ataque por análise de tráfego.

Os modos de operação do padrão DES foram publicados no FIPS 81 e também no ANSI X3.106. Uma versão mais geral do padrão (ISO92b) generalizou os quatro modos de DES para ser aplicável a qualquer algoritmo de bloco de tamanho de bloco arbitrário. Os modos padronizados são *Electronic Code Book*, *Cipher Block Chaining*, *Cipher Feedback* e *Output Feedback*. Além deste modos utilizados amplamente, outros também são conhecidos, entre eles: *Counter mode* e o *Propagating Cipher Block Chaining*.

C.2.2.1 - Modos de Operação

C.2.2.1.1 - Eletronic CodeBook - ECB

No modo ECB, cada bloco de texto puro é codificado independentemente dos outros blocos pelo algoritmo de encriptação (ver figura 31). A segurança do modo ECB é definida pela segurança do algoritmo de encriptação utilizado. Caso exista padrões de texto puro, estes serão facilmente detectados.

Cada bloco de texto puro idêntico gera um bloco de texto cifrado idêntico. O texto puro pode ser facilmente manipulado pela remoção, repetição ou troca de blocos. A velocidade de cada operação de encriptação é somente dependente do algoritmo de bloco utilizado. O modo ECB possibilita uma fácil paralelização o qual ocasiona em um desempenho melhor quando comparado com os outros modos.

Se ocorrerem um ou mais erros na transmissão dos bits de um único bloco de texto cifrado, a decriptação somente daquele bloco é prejudicada, não há propagação de erro para os blocos seguintes.

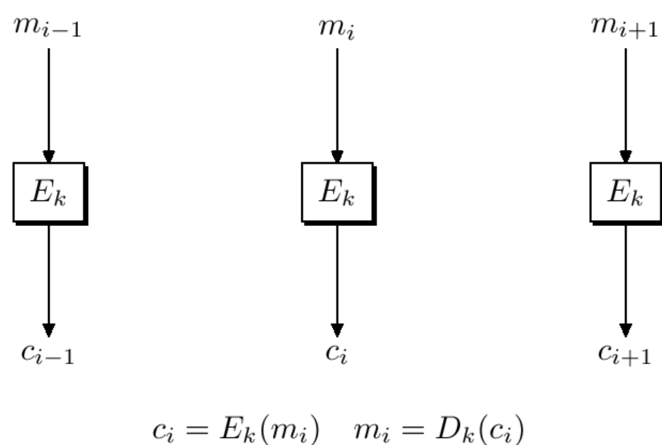


Figura 31 - Modo *Electronic Code Book*

C.2.2.1.2 - Cipher Block Chaining - CBC

No modo CBC, em cada bloco de texto puro é realizado a operação XOR com o prévio bloco de texto cifrado e somente então será encriptado. Um vetor de inicialização c_0 é usado como uma semente para o processo (veja figura 32).

Este modo é tão seguro contra ataques padrão quanto o algoritmo de bloco utilizado. Além disso, qualquer padrão existente no texto puro é dissimulado pela operação XOR do bloco de texto cifrado prévio com o bloco de texto puro. Note também que o texto puro não pode ser manipulado diretamente exceto através de remoção de blocos do ínicio ou fim do texto cifrado. O vetor de inicialização deveria ser diferente para qualquer duas mensagem encriptadas com a mesma chave e preferentemente que seja escolhido um valor aleatório. Ele não precisa ser codificado e pode ser transmitido com o texto cifrado.

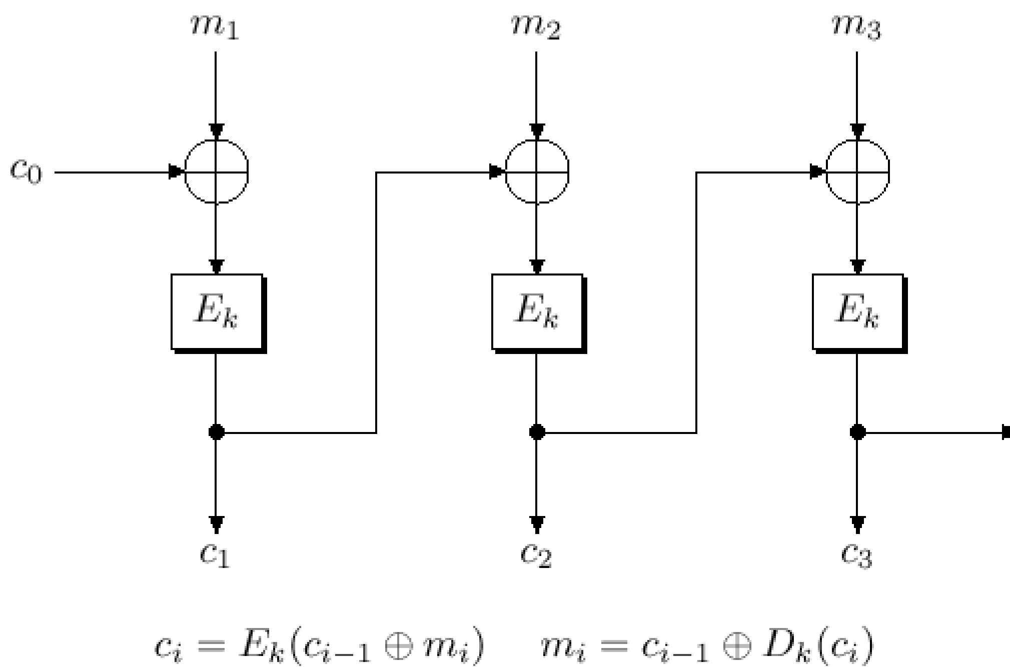
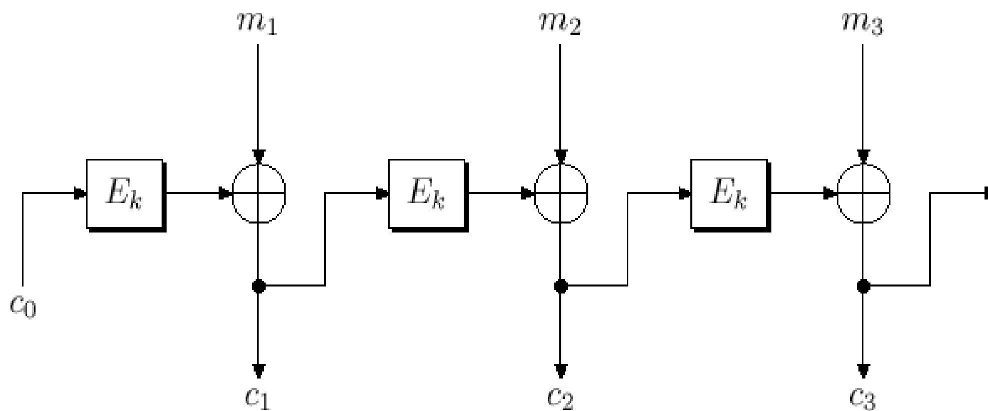


Figura 32 - Modo *Cipher Block Chaining*

A velocidade de encriptação é idêntica ao do algoritmo de bloco utilizado, mas o processo de encriptação não pode ser facilmente paralelizado, embora o processo de decriptação possa ser.

C.2.2.1.3 - Cipher Feedback - CFB

No modo CFB, o bloco anterior de texto cifrado é encriptado e a saída produzida é combinada com o bloco de texto puro usando XOR para produzir o bloco de texto cifrado atual. É possível definir o modo CFB de forma que use uma realimentação menor que um bloco de dados completo. Um vetor de inicialização c_0 é usado como semente para o processo (veja figura 33).



$$c_i = E_k(c_{i-1}) \oplus m_i \qquad m_i = E_k(c_{i-1}) \oplus c_i$$

Figura 33 - Modo *Cipher Feedback*

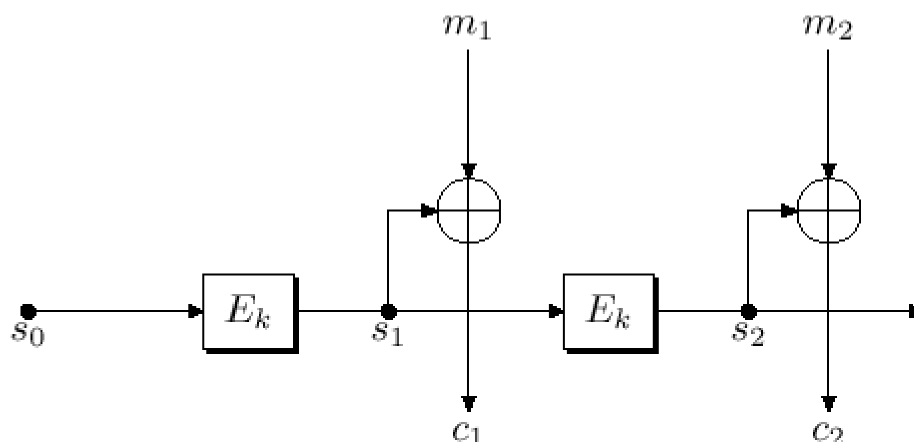
O modo CFB é tão seguro quanto o algoritmo de encriptação subjacente e os padrões de texto puro são dissimulados no texto cifrado pelo uso da operação XOR. O texto puro não pode ser manipulado diretamente exceto pela eliminação dos bloco do início ou do fim do texto cifrado. No modo CFB e com a realimentação de um bloco completo, quando dois blocos de texto cifrado são idênticos, as saídas das operações de encriptação de bloco são idênticas no próximo passo também. Isto permite que a informação sobre o bloco de texto puro sejam vazadas. As considerações de segurança para o vetor de inicialização são as mesmas do modo CBC, exceto pelo ataque descrito na próxima seção. Ao invés disso, o último bloco de texto cifrado pode ser atacado.

Quando usando a realimentação completa, a velocidade de encriptação é idêntica a do algoritmo de bloco utilizado. O processo de encriptação não pode ser facilmente paralelizado.

C.2.2.1.4 - Output Feedback - OFB

O modo OFB é semelhante ao modo CFB exceto que a operação XOR com cada bloco de texto puro é gerada independentemente do texto puro e do texto cifrado. Um vetor de inicialização s_0 é usado como semente para uma sequência de blocos de dados s_i, e cada bloco de dados s_i é derivado da encriptação do bloco de dados prévio

s_{i-1}. A encriptacão de um bloco de texto puro é obtido do XOR do bloco de texto puro com o bloco de dados pertinente (veja figura 34).



$$c_i = m_i \oplus s_i \qquad m_i = c_i \oplus s_i \qquad s_i = E_k(s_{i-1})$$

Figura 34 - Modo *Output Feedback*

A realimentação parcial (menos de um bloco completo) não é recomendada por motivos de segurança. O modo OFB tem uma vantagem sobre o modo CFB onde qualquer bit errado que possa ocorrer durante a transmissão não é propagado de forma a afetar a decriptação dos blocos subseqüentes. As considerações de segurança para o vetor de inicialização são as mesmas do modo CFB.

Um problema com o modo OFB é que o texto puro é manipulado facilmente. Isto é, um atacante que conhece um bloco de texto puro m_i pode substituí-lo com um falso bloco de texto puro x através da operação XOR $m_i \oplus x$ para o bloco de texto cifrado correspondente c_i. Há ataques semelhantes nos modos CBC e CFB mas nestes ataques o bloco de texto puro é modificado de forma imprevisível pelo atacante. Ainda, o primeiro bloco de texto cifrado (quer dizer, o vetor de inicialização) em modo CBC e o último no modo CFB são da mesma maneira vulneráveis ao ataque como os blocos no modo OFB. Ataques deste tipo podem ser prevenidos usando um esquema de assinatura digital, por exemplo.

A velocidade de encriptação é idêntica ao do algoritmo de bloco utilizado. Embora o processo não possa facilmente ser paralelizado, algum tempo pode ser economizado gerando o *keystream* antes dos dados estarem disponíveis para encriptação.

Devido às negligências no modo OFB, Diffie propôs um modo adicional de operação, conhecido como modo contador (*counter mode*). Difere do modo OFB na forma que os sucessivos blocos de dados são gerados para as encriptações subseqüentes. Em vez de obter um bloco de dados com a encriptação do bloco de dados prévio, Diffie propôs a encriptação da quantidade i + IV mod 2^{64} para o bloco de dados iésimo, onde IV é algum vetor de inicialização.

Certos modos de operação do algoritmo de bloco transformam este efetivamente em um gerador de *keystream* e deste modo qualquer algoritmo de bloco pode ser usado como um algoritmo de fluxo. Como exemplo temos o DES em modo CFB ou OFB. Porém, os algoritmos de fluxo com um projeto dedicado são tipicamente muito mais rápidos.

C.2.2.2 - Data Encryption Standard – DES e 3DES

O algoritmo de blocos DES, um acrônimo para *Data Encryption Standard*, é o nome da *Federal Information Processing Standard* (FIPS) 46-3 que descreve o algoritmo de encriptação de dados (DEA). O DEA também é definido no ANSI padrão X9.32.

O DEA é uma melhoria do algoritmo Lucifer desenvolvido pela IBM no ínicio de 1970. Embora o algoritmo tenha sido essencialmente projetado pela IBM, o NSA e NBS (agora NIST) representaram um papel importante nas fases finais de desenvolvimento. O DEA, freqüentemente chamado DES, foi estudado extensivamente desde sua publicação e é o algoritmo simétrico mais conhecido e utilizado no mundo.

O DEA tem um bloco de dados de tamanho de 64 bit e usa uma chave de 56 bits durante a execução (8 bits de paridade são retirados de uma chave completa de 64 bits). O DEA é um algoritmo *Feistel* de 16 *rounds* e foi projetado originalmente para implementação em hardware. O DEA também pode ser usado para encriptação de um único usuário, como por exemplo armazenar arquivos em um disco rígido na forma encriptada. Em um ambiente multiusuário, a distribuição das chaves de forma segura pode ser difícil e a criptografia de chave pública fornece uma solução ideal para este tipo de problema.

O NIST tem recertificado o DES (FIPS 46-1, 46-2, 46-3) a cada cinco anos. O FIPS 46-3 reafirmou o uso do DES a partir de Outubro de 1999, mas o DES simples somente é permitido para sistemas legado. O FIPS 46-3 inclui uma definição de Triplo-DES (TDEA, correspondendo a X9.52). O TDEA é o algoritmo simétrico escolhido e aprovado pela FIPS. O DES e o Triplo-DES estão sendo substituídos pelo novo Padrão de Encriptação Avançado (AES) [23].

Triplo-DES

Durante algum tempo foi prática comum proteger a informação com Triplo-DES ao invés do DES. Isto significa que os dados de entrada, na prática, são encriptados três vezes. Há uma variedade de modos de realizar esta operação. O padrão ANSI X9.52 define uma encriptação Triplo-DES com chaves k_1, k_2 e k_3 como $C = E_{k3} (D_{k2} (E_{k1} (M)))$. Onde E_k e D_k denotam respectivamente encriptação e decriptação DES, respectivamente, com a chave k. Este modo de encriptação às vezes é chamado DES-EDE. Outra variante é o DES-EEE que consiste em três encriptações sucessivas. Há três opções de chaveamento definidas no ANSI X9.52 para DES-EDE:

1- As três chaves k_1, k_2 e k_3 são independentes.
2- As chaves k_1 e k_2 são independentes, mas $k_1 = k_3$.
3- As três chaves são iguais, k1 = k2 = k3.

A terceira opção faz o Triplo-DES ser compatível (*backward*) com DES.

Como todos os algoritmos de bloco, o Triplo-DES pode ser usado em uma variedade de modos de operação. O padrão ANSI X9.52 detalha sete modos, inclusive os quatro modos padrões descritos anteriormente.

O uso de dupla e tripla encriptação nem sempre fornece a segurança adicional que pode ser esperada. Por exemplo, considere o seguinte ataque *meet-in-the-middle* na encriptação dupla. Temos um algoritmo de bloco com tamanho de chave n e $E_k(P)$ denota a encriptação da mensagem P que usa a chave k. A encriptação dupla com duas chaves diferentes dá um tamanho total da chave de 2n. Porém, supõe que somos capazes de armazenar $E_k(P)$ para todas as chaves k e temos o texto puro P, e suponha mais adiante que temos um texto cifrado C tal que $C = E_{k2} (E_{k1} (P))$ para as chaves

secretas k_1 e k_2. Para cada chave L existe exatamente uma chave k tal que $D_L(C) = E_k(P)$. Há exatamente 2^n chaves possíveis que levam ao par (P;C) e essas chaves podem ser encontradas em aproximadamente $O(2^n)$ passos. Com a capacidade de armazenar só $2^p < 2^n$ chaves, podemos modificar este algoritmo e achar todas as possíveis chaves em $O(2^{2n-p})$ passos.

Outro exemplo de uma falsa segurança foi mostrada em [24], onde uma tripla encriptação EDE com três chaves diferentes é analisada. Sejam $K = (k_a; k_b; k_c)$ e $K0 = (k_{a \oplus} \Delta; k_b; k_c)$ duas chaves secretas onde Δ é uma conhecida constante e $\oplus$ denota a operação XOR. Suponha que temos o texto cifrado C e a correspondente decriptação P e P' de C com as chaves K e K' respectivamente. Desde $P' = D_{ka \oplus \Delta} (E_{ka} (P))$, podemos determinar k_a (ou todos os possíveis candidatos para k_a) em $O(2^n)$ passos onde n é o tamanho da chave. Usando um ataque semelhante ao descrito acima, podemos determinar o resto da chave (quer dizer, k_b e k_c) em outros $O(2^n)$ passos.

C.2.2.3 - International Data Encryption Algorithm - IDEA

O algoritmo IDEA (*International Data Encryption Algorithm*) [25] é a segunda versão de um algoritmo de bloco projetado e apresentado por Lai e Massey. É um algoritmo de bloco iterativo de 64 bits com uma chave de 128 bits. O processo de encriptação requer oito *rounds* complexos. Embora a cifra não tenha uma estrutura Feistel, a decriptação é realizada da mesma maneira como a encriptação uma vez que a subchave de decriptação é calculada a partir da subchave de encriptação. A estrutura do algoritmo foi projetada para ser implementada facilmente em software e hardware. A segurança do IDEA confia no uso de três tipos incompatíveis de operações aritméticas em palavras de 16 bits. Porém algumas das operações aritméticas usadas no IDEA não são rápidas em software. Como resultado a velocidade do IDEA em software é semelhante a do DES.

Um dos princípios usados durante o projeto do IDEA era facilitar a análise de sua força contra a criptanálise diferencial e o IDEA é considerado imune a criptoanálise diferencial. Além disso, não há nenhum ataque criptoanálitico linear para o IDEA e não há nenhuma fraqueza algébrica conhecida no IDEA. O resultado de criptoanálise mais significante é devido a Daemen, que descobriu uma classe

grande de 2^{51} chaves fracas no qual o uso de tal chave durante a encriptação poderia ser detectado facilmente e a chave recuperada. Porém, como há 2^{128} possíveis chaves, este resultado não tem nenhum impacto na segurança prática do algoritmo para encriptação quando as chaves de encriptação são escolhidas de modo aleatório. Geralmente é considerado que o IDEA é um algoritmo muito seguro. Tanto o seu projeto de desenvolvimento como a base teórica foram abertamente e amplamente discutidos.

C.2.2.4 - Rivest Cipher #5 e #6 - RC5 e RC6

O RC5 [26] é um algoritmo de bloco rápido projetado por Ronald Rivest da empresa RSA Data Security (agora RSA Security) em 1994. É um algoritmo parametrizado com um tamanho de bloco variável, um tamanho chave variável e um número variável de *rounds*. As escolhas permitidas para o tamanho de bloco são 32 bits (para experimentação e avaliação somente), 64 bits (como um substituto ao DES) e 128 bits. O número de *rounds* pode variar de 0 a 255 enquanto a chave pode variar de 0 a 2040 bits de tamanho. Tal variabilidade embutida fornece uma flexibilidade em todos os níveis de segurança e eficiência.

Há três rotinas básicas no algoritmo RC5: a expansão da chave, encriptação e decriptação. Na rotina de expansão, a chave secreta é expandida para preencher uma tabela de chave cujo tamanho depende do número de *rounds*. A tabela chave é então usada na encriptação e decriptação. A rotina de encriptação consiste em três operações primitivas: adição de inteiro, operação lógica XOR bit a bit e rotação variável. A simplicidade excepcional do RC5 torna fácil a implementação e a análise.

O uso constante de rotações dependentes dos dados e a mistura de operações diferentes fornecem a segurança do RC5. Em particular, o uso das rotações ajuda a frustar a criptoanálise diferencial e linear. Nos cinco anos desde que o RC5 foi proposto, houve numerosos estudos da segurança do RC5 e cada estudo forneceu um entendimento maior da forma como a estrutura e os componentes do RC5 contribuem para a sua segurança.

O RC6 é um algoritmo de bloco baseado no RC5 e projetado por Rivest, Sidney e Yin para a empresa RSA Security. Como o RC5, o RC6 é um algoritmo

parametrizado onde o tamanho de bloco, o tamanho chave e o número de *rounds* são variáveis. Novamente, o limite superior do tamanho da chave é 2040 bits. A meta principal para os inventores foi satisfazer as exigências do AES. Realmente, o RC6 esteve entre os cinco finalistas.

Há duas características novas principais no RC6 comparado ao RC5: a inclusão da operação de multiplicação de inteiro e o uso de quatro registradores b/4 bits ao invés de dois registradores b/2 bits como no RC5 (b é o tamanho de bloco). A multiplicação de inteiro é usada para aumentar a difusão alcançada por *round* de forma que menos *rounds* sejam necessários e a velocidade da encriptação possa ser aumentada. A razão de usar quatro registradores em vez de dois é técnica e não teórica, o tamanho de bloco *default* do AES é 128 bits e embora o RC5 realize operações de 64 bits quando usando este tamanho de bloco, as operações de 32 bits são preferíveis dada a arquitetura pretendida para o AES.

C.2.2.6 - Ataques Conhecidos aos Algoritmos Simétricos de Blocos

Há vários ataques que são específicos aos algoritmos simétricos de bloco e nesta seção serão explicados três tipos de ataques: a criptanálise diferencial, a criptanálise linear e a exploração de chaves fracas.

A criptoanálise diferencial é um tipo de ataque que pode ser montado sobre os algoritmos de blocos iterativos. Esta técnica foi introduzida primeiro por Murphy em um ataque ao FEAL-4, mas ela foi aperfeiçoada por Biham e Shamir que a usou para atacar o DES. A criptoanálise diferencial é basicamente um ataque dirigido ao texto puro. Baseia-se na análise da evolução das diferenças entre dois textos puros relacionados e como eles são encriptados pela mesma chave. Com uma análise cuidadosa dos dados disponíveis, podem ser nomeadas probabilidades a cada uma das possíveis chaves e eventualmente a chave mais provável é identificada como a correta.

A criptoanálise diferencial foi usada contra uma grande variedade de algoritmos e obteve variados graus de sucesso. Sua efetividade no ataque ao DES foi limitada por um projeto muito cuidadoso das *S-boxes* durante o projeto do DES. Estudos para proteção dos algoritmos contra a criptanálise diferencial foram conduzidos por Nyberg e Knudsen como também por Lai, Massey e Murphy. A criptanálise diferencial também foi útil no ataque a outras primitivas de criptografia como as funções de hash.

Matsui e Yamagishi foram os primeiros a inventar a criptanálise linear em um ataque ao FEAL. Ele foi estendido por Matsui para atacar o DES. A criptanálise linear é um ataque dirigido a um texto puro conhecido que usa uma aproximação linear para descrever o comportamento do algoritmo de bloco. Dado pares suficientes de texto puro e de textos encriptados correspondentes, alguns bits da informação sobre a chave podem ser obtidos e aumentando a quantidade de dados normalmente aumenta a probabilidade de sucesso do ataque.

Houve uma variedade de aperfeiçoamentos ao ataque básico. Langford e Hellman introduziram um ataque chamado de criptanálise diferencial-linear que combina elementos da criptanálise diferencial com os de criptanálise linear. Também, Kaliski e Robshaw mostraram que um ataque criptanálitico linear usando aproximações múltiplas poderia permitir uma redução na quantidade de dados

requerida para um ataque com sucesso. Outras questões como a proteção dos algoritmos contra criptanálise linear foram analisadas por Nyberg, Knudsen e O'Conner.

As chaves fracas são aquelas que possuem um certo valor para o qual o algoritmo de bloco em questão exibirá certas regularidades na encriptação ou, em outros casos, um nível pobre de encriptação. Por exemplo, no DES há quatro chaves para as quais a encriptação é exatamente igual a decriptação. Isto significa que se fosse encriptar duas vezes com estas chaves fracas, o texto puro original seria recuperado. Para o IDEA, há uma classe de chaves para qual a criptanálise é facilitada grandemente e a chave pode ser recuperada. Porém, em ambos estes casos, o número de chaves fracas é uma fração tão pequena de todo o universo de chaves possíveis que a chance de escolher ao acaso estas é excepcionalmente desprezível. Em tais casos, eles representam nenhuma significante ameaça a segurança do algoritmo bloco quando usado para a encriptação.

C.2.3 - Message Authentication Code - MAC

Um código de autenticação de mensagem (*Message Authentication Code* - MAC) é uma *tag* de autenticação, conhecido também como *checksum*, derivado da aplicação de algum esquema de autenticação, junto com uma chave secreta, em uma mensagem. Diferentemente das assinaturas digitais, os MACs são computados e verificados com a mesma chave, de maneira que somente pode ser verificado pelo receptor correto.

Um tipo de MAC muito utilizado é o baseado nas funções de *hash*, frequentemente conhecido como HMAC. A especificação encontra-se na RFC 2104. Utiliza uma ou mais chaves em conjunto com uma função de *hash* (como MD5 e SHA) para produzir um *checksum* que é adicionado a mensagem.

C.3 - Algoritmos Assimétricos

Na criptografia tradicional, o remetente e receptor de uma mensagem sabem e usam a mesma chave secreta. O remetente usa a chave secreta para codificar a mensagem e o receptor usa a mesma chave secreta para decifrar a mensagem. Este método é conhecido como algoritmo (criptografia) de chave simétrica ou de chave secreta. O desafio principal é conseguir que o remetente e receptor combinem a chave secreta sem ninguém mais descobrir. Se eles estão em localizações físicas distintas, eles têm que confiar no mensageiro, no sistema telefônico ou algum outro meio de transmissão para impedir a revelação da chave secreta.

Qualquer um que escutar ou interceptar a chave em trânsito pode depois ler, modificar e forjar todas as mensagens encriptadas ou autenticadas usando a chave. O processo de geração, transmissão e armazenamento das chaves é conhecido como Gerenciamento de chave. Todos os criptosistemas têm que lidar com as questões relativas a administração das chaves e todas as chaves em um criptosistema de chave secreta têm que permanecer secretas. A criptografia de chave secreta tem freqüentemente dificuldade de fornecer um administração das chave de forma segura, especialmente em sistemas abertos com um número grande de usuários.

Para resolver o problema de administração das chaves, Whitfield Diffie e Martin Hellman [27] introduziram o conceito de criptografia de chave pública em 1976. Os algoritmos assimétricos, também conhecidos como algoritmos de chave pública, implementam a criptografia de chave pública.

Estes algoritmos têm dois usos primários: na encriptação e na autenticação (assinatura digital). Nestes sistemas, cada pessoa adquire um par de chaves, uma chamada de chave pública e o outra chamada de chave privada. A chave pública é publicada enquanto a chave privada é mantida em segredo. A necessidade do remetente e do receptor de compartilhar a informação secreta é eliminada. Todas as comunicações envolvem chaves públicas somente e nenhuma chave privada é transmitida ou compartilhada. Neste sistema, não é mais necessário confiar na segurança dos meios de comunicações. A única exigência é que as chaves públicas sejam associadas com os respectivos usuários delas de uma maneira confiável. Qualquer um pode enviar uma mensagem confidencial usando simplesmente uma

informação pública, mas a mensagem só pode ser decifrada com a chave privada que está na posse exclusiva do receptor pretendido.

Em um criptosistema de chave pública, a chave privada é sempre unida matematicamente à chave pública. Sendo assim, sempre é possível atacar um sistema de chave pública derivando a chave privada a partir da chave pública. Tipicamente, a defesa contra este tipo de ataque é tornar o problema de derivar a chave privada a partir da chave pública tão difícil quanto possível. Por exemplo, alguns criptosistemas de chave pública são projetados de tal forma que a derivação da chave privada a partir da chave pública exija ao atacante que fatore um número muito grande, e é impraticável computacionalmente executar a derivação. Esta é a idéia embutida no algoritmo de chave pública RSA.

Os algoritmos de chave pública tem o processamento muito mais lento quando comparados aos simétricos. Este é um grande problema para os sistemas em tempo real, como as aplicações multimídia que estão sendo disponibilizadas de forma crescente na Internet, que necessitam utilizar os algoritmos assimétricos.

O algoritmo de chave pública mais popular e de reconhecimento internacionalmente como um dos mais seguros atualmente é o RSA e será explicado detalhadamente na seção seguinte. Uma boa história da criptografia de chave pública foi dada por Diffie [27].

Encriptação

Quando a Alice desejar enviar uma mensagem secreta a Bob, ela procura a chave pública de Bob em um diretório, usa a mesma para codificar a mensagem e envia. Bob usa então a sua exclusiva chave privada para decifrar a mensagem e possibilitar a leitura. Ninguém que esteja escutando a mensagem poderá decifrá-la. Qualquer um pode enviar uma mensagem encriptada ao Bob, mas somente o Bob pode ler (porque somente o Bob sabe a chave privada de Bob).

C.3.1 - Rivest-Shamir-Adleman - RSA

O criptosistema RSA é um algoritmo de chave pública que oferece serviço de encriptação e de assinatura digital (autenticação). Ronald Rivest, Adi Shamir e Leonard Adleman desenvolveram o sistema RSA em 1977 [28]. O acrônimo RSA foi obtido da primeira letra de cada um dos nomes (último) de seus inventores.

O algoritmo RSA funciona como mostrado a seguir: escolhe-se dois números grandes grandes, p e q. Computa-se o produto deles n = pq, n é chamado de módulo. Escolha um número “e” menor que n e primo relativo a (p-1)(q-1), o qual significa que “e” e (p-1)(q-1) não tem nenhum fator comum exceto 1. Ache outro número d tal que (ed-1) seja divisível por (p-1)(q-1). Os valores e e d são chamados de expoente público e privado, respectivamente. A chave pública é o par (n, e) e a chave privada é o par (n, d). Os fatores p e q podem ser destruídos ou podem ser mantidos com a chave privada.

Atualmente é difícil de obter a chave privada d a partir da chave pública (n, e). Porém se a pessoa pudesse fatorar n em p e q, então poderia obter a chave privada d. Assim a segurança do sistema de RSA está baseada na suposição que fatorar é difícil. A descoberta de um método fácil de fatorar irá quebrar o RSA. A seguir é explicado como o sistema RSA pode ser usado para encriptação.

Encriptação

Suponha que Alice quer enviar para uma mensagem m para Bob. Alice cria o texto encriptado c através da exponenciação: $c = m^e \bmod n$, onde e e n é a chave pública de Bob. Ela envia c a Bob. Para decriptar, Bob também realiza uma exponenciação: $m = c^d \bmod n$. A relação entre “e” e d assegura que Bob recuperou m corretamente. Como somente Bob conhece d, somente ele pode decriptar esta mensagem.

C.4 - Assinaturas Digitais

Autenticação é qualquer processo pelo qual alguém prova e verifica certa informação. Às vezes, a pessoa pode querer verificar a origem de um documento, a identidade do remetente, a hora e data que um documento foi enviado e/ou assinado, a identidade de um computador ou usuário e assim por diante. Uma assinatura digital é um meio criptográfico pelo qual muitas destas opções podem ser verificadas. A assinatura digital de um documento é um pedaço de informação baseada no documento e na chave privada do signatário. Isto é criado tipicamente pelo uso de uma função de hash e uma função de assinatura privada (encriptando com a chave privada do signatário), mas há outros métodos.

Diariamente, as pessoas assinam seus nomes em cartas, recibos de cartão de crédito e outros documentos e assim demonstram que eles estão de acordo com o conteúdo. Quer dizer, eles autenticam que eles são de fato o remetente ou originador do objeto em questão. Isto permite que outros verifiquem que uma mensagem particular realmente foi originada pelo signatário. Porém, isto não é perfeitamente seguro, pois as pessoas podem furtar a assinatura de um documento e colocar em outro e assim criar documentos fraudulentos. As assinaturas escritas também são vulneráveis a falsificação porque é possível reproduzir uma assinatura em outros documentos bem como alterar documentos depois que eles foram assinados.

Ambas as assinaturas digitais e as assinaturas escritas a mão confiam no fato que é muito difícil achar duas pessoas com a mesma assinatura. As pessoas usam a criptografia de chave pública para computar assinaturas digitais associando algo único com cada pessoa. Quando a criptografia de chave pública é usada para encriptar uma mensagem, o remetente codifica a mensagem com a chave pública do recipiente pretendido. Quando a criptografia de chave pública é usada para calcular uma assinatura digital, o remetente encripta a impressão digital (realmente digital) do documento com a sua própria chave privada. Qualquer um com acesso a chave pública do signatário pode verificar a assinatura.

Suponha que a Alice queira enviar um documento assinado ou uma mensagem a Bob. O primeiro passo geralmente é aplicar uma função de hash a mensagem, criando o que é chamado de sumário da mensagem (*message digest*). O sumário de mensagem

é normalmente bem menor que a mensagem original. De fato, o trabalho da função de hash é levar uma mensagem de tamanho arbitrário e encolher até um tamanho fixo. Para criar uma assinatura digital, normalmente se assina (encripta) o sumário da mensagem ao invés da própria mensagem. Isto economiza uma quantia considerável de tempo, entretanto cria uma leve insegurança mostrada posteriormente.

Alice envia ao Bob o sumário encriptado (assinatura) da mensagem e a mensagem (que pode ou não estar encriptada). Para que Bob autentique a assinatura ele tem que aplicar a mesma função de *hash* utilizada pela Alice à mensagem que ela enviou e tem que decriptar o sumário da mensagem usando a chave pública da Alice e finalmente tem que comparar os dois resultados. Caso os dois valores sejam iguais, então a assinatura da Alice foi autenticada com sucesso. E caso sejam diferentes, há algumas explicações possíveis: ou alguém está tentando personificar a Alice ou a mensagem foi alterada posteriormente ao instante que a Alice efetuou a assinatura ou um erro aconteceu durante a transmissão.

Há um problema potencial com este tipo de assinatura digital. Alice não somente assinou a mensagem que ela pretendia mas também assinou todas as outras mensagens que tem o mesmo hash (sumário de mensagem). Quando duas mensagens tem o mesmo sumário de mensagem é chamado de colisão. A propriedade da função de hash ser livre de colisão é uma exigência de segurança necessária para a maioria dos esquemas de assinatura digitais. Uma função de hash é considerada segura se é consumido muito tempo para descobrir (se possível) a mensagem original dado o seu sumário. Porém, há um ataque conhecido como ataque de aniversário que confia no fato que é mais fácil de achar duas mensagens que tem o mesmo hash que encontrar uma mensagem que tem um valor particular de sumário. Seu nome surge do fato que para um grupo de 23 ou mais pessoas a probabilidade que duas ou mais pessoas compartilhem a mesma data de aniversário é maior que 50%.

Além disso, alguém poderia fingir ser Alice e assinar documentos com um par de chaves que reivindica ser dela. Para evitar cenários como este, há documentos digitais chamados certificados digitais que associam uma pessoa com uma chave pública específica.

Podem ser usados *digital timestamps* em conjunto com as assinaturas digitais para vincular um documento a uma determinada hora de recebimento. Não é suficiente verificar a data da mensagem pois nos computadores a data pode ser

manipulada facilmente. O melhor é que o *timestamping* seja realizado por alguém em que todos confiam, como uma autoridade certificadora.

C.4.1 - Rivest-Shamir-Adleman - RSA

O algoritmo de chave pública RSA pode ser usados para autenticar ou identificar uma pessoa ou entidade. Essa aplicação é possibilitada pois cada entidade tem uma chave privada associada que teoricamente ninguém mais tem acesso. Isto permite uma identificação única e positiva.

Suponha que Alice deseje enviar uma mensagem assinada a Bob. Ela aplica uma função de hash a mensagem para criar um sumário da mensagem, o qual serve como uma impressão digital da mensagem. Ela encripta então o sumário de mensagem com a chave privada dela, criando assim a assinatura digital do documento, e ela envia a Bob juntamente com a própria mensagem. Ao receber a mensagem e assinatura, decripta a assinatura com a chave pública de Alice para recuperar o sumário de mensagem. Ele então calcula o sumário da mensagem com a mesma função de hash que Alice usou e compara o resultado para o sumário de mensagem decriptado da assinatura. Se eles forem precisamente iguais, a assinatura foi verificada com sucesso e ele pode ficar confiante que a mensagem realmente adveio de Alice. Se eles não forem iguais, então a mensagem ou foi originada em outro lugar ou foi alterada depois que foi assinada e ele rejeita a mensagem.

Qualquer pessoa que ler a mensagem pode verificar a assinatura. Isto não satisfaz situações onde a Alice deseja manter segredo do documento. Neste caso ela pode desejar assinar o documento, e então encriptar usando a chave pública de Bob. Bob precisará decriptar usando a chave privada dele e então verificar a assinatura na mensagem recuperada que usa a chave pública de Alice. Alternadamente, se for necessário entidades intermediárias para validar a integridade da mensagem mas que não tenham a permissão de decriptar seu conteúdo, um sumário da mensagem pode ser computado da mensagem encriptada ao invés da mensagem em texto puro.

Na prática, o expoente público no algoritmo RSA normalmente é muito menor que o expoente privado. Isto significa que a verificação de uma assinatura é mais rápida que o processo de assinar propriamente. Isto é desejável porque uma

mensagem será assinada somente uma vez por um indivíduo mas a assinatura pode ser verificada muitas vezes.

Devia ser impraticável para qualquer um encontrar uma mensagem a qual gere um hash de valor específico ou achar duas mensagens que tenham o mesmo hash. Se fosse possível, um intruso poderia colocar uma falsa mensagem com a assinatura da Alice. Funções de hash como MD5 e SHA foram especificamente projetadas para inviabilizar tais ataques.

Um ou mais certificados podem acompanhar uma assinatura digital. Um certificado é um documento assinado que vincula a chave pública a identidade da pessoa. Seu propósito é prevenir alguém de personificar outra pessoa. Se um certificado está presente, o receptor pode conferir se a chave pública pertence a parte intencionada, assumindo que a chave pública do certificador é confiável.

C.4.2 - Digital Signature Algorithm - DSA

O *National Institute of Standards and Technology* (NIST) publicou o *Digital Signature Algorithm* (DSA) no *Digital Signature Standard* (DSS), o qual é uma parte do projeto Capstone do governo norte-americano. O DSS foi selecionado pelo NIST, em cooperação com o NSA, para ser o padrão de autenticação digital do governo norte-americano. Este padrão foi emitido em Maio de 1994.

O DSA está baseado no problema do logaritmo discreto e está relacionado a esquemas de assinatura que foram propostos por Schnorr e ElGamal. Embora o sistema RSA possa ser utilizado para encriptação e assinaturas digitais, o DSA somente pode ser usado para fornecer as assinaturas digitais. Para uma descrição detalhada de DSA, veja [29].

No DSA, o procedimento de assinar é mais rápido que a verificação de assinatura, ao contrário do algoritmo RSA onde a verificação de assinatura é muito mais rápida que a assinatura (se os expoentes público e privado, respectivamente, são escolhidos para obter esta propriedade e é o caso habitual). Poderia ser mencionado a vantagem do procedimento de assinatura ser mais rápido, mas como em muitas aplicações cada porção da informação digital é assinada uma vez, mas verificada freqüentemente, pode ser bem mais vantajoso ter a verificação mais rápida. Os *tradeoffs* e assuntos envolvidos foram explorados por Wiener. Tem sido realizados

trabalhos por muitos autores inclusive Naccache et al. para o desenvolvimento de técnicas que melhorem a eficiência do DSA, tanto para a assinatura como na verificação.

Embora vários aspectos do DSA tenham sido criticados desde seu anúncio, ele está sendo incorporado em vários sistemas e especificações. A crítica inicial focou em alguns pontos principais: (i) a ausência da flexibilidade tal como o criptosistema RSA, (ii) a verificação de assinaturas com DSA era muito lenta, (iii) a existência de um segundo mecanismo de autenticação causou problemas para os vendedores de hardware e software que já haviam padronizado o uso do algoritmo RSA, (iv) o processo pelo qual o NIST escolheu DSA foi muito reservado e arbitrário. Outras críticas relacionadas à segurança do esquema foram enviadas ao NIST o qual modificou a proposta original.

O DSS foi proposto originalmente pelo NIST com um tamanho de chave fixo de 512 bits. Depois de várias críticas de que não era suficientemente seguro, especialmente para a segurança de longo prazo, o NIST revisou o DSS para permitir tamanhos de chaves de até 1024 bits. Na verdade, agora são permitidos tamanhos de chaves até maiores na norma ANSI X9.30. O algoritmo DSA , até o presente momento, é considerado seguro com chaves de 1024 bits.

O DSA faz uso de computação de logaritmos discretos em certos subgrupos no campo finito GF(p) para algum primo p. O problema foi proposto inicialmente para uso criptográfico em 1989 por Schnorr. Nenhum ataque eficiente ainda foi informado nesta forma do problema de logaritmo discreto. Alguns investigadores advertiram sobre a existência de primos *trapdoor* no DSA que poderia habilitar uma chave ser quebrada facilmente. Estes primos *trapdoor* são relativamente raros e facilmente evitados se forem seguidos procedimentos apropriados na geração da chave.

C.5 - Algoritmos para Troca de Chave

Um protocolo para acordo de chave (*key agreement protocol*), também chamado de protocolo para troca de chave, compõe-se de uma série de passos usados quando duas ou mais partes precisam definir uma chave para utilizar nos algoritmos de chave secreta (simétricos normalmente). Estes protocolos permitem as pessoas compartilharem chaves livremente e com segurança sobre qualquer meio inseguro sem a necessidade de um prévio estabelecimento de um segredo compartilhado.

Suponha que Alice e Bob queiram usar um algoritmo de chave secreta para comunicar com segurança. Eles têm que primeiramente decidir qual chave compartilhada utilizar. Ao invés de Bob chamar Alice no telefone e discutir qual chave será utilizada, o qual os deixaria vulneráveis para um atacante, eles deveriam usar um protocolo para acordo de chave. Usando um protocolo para acordo de chave, Alice e Bob podem trocar com segurança a chave em um ambiente inseguro.

Um exemplo de tal protocolo é o conhecido como Diffie-Hellman. Em muitos casos, a criptografia de chave pública é usada em um protocolo para acordo de chave. Outro exemplo de protocolo é o conhecido como Envelope Digital para acordo da chave.

C.5.1 - Diffie-Hellman

O protocolo Diffie-Hellman (também conhecido protocolo para acordo de chave exponencial) foi desenvolvido por Diffie e Hellman [27] em 1976 e publicou no famoso artigo "Novas direções na Criptografia". O protocolo permite a dois usuários trocarem uma chave secreta sobre um meio inseguro sem o uso de qualquer segredo anterior.

O protocolo tem dois parâmetros de sistema p e g. Eles são públicos e podem ser utilizados por todos os usuários no sistema. O parâmetro p é um número primo e o parâmetro g (normalmente chamado de gerador) é um inteiro menor que p, com a propriedade seguinte: para todo número n entre 1 e p-1 inclusive, há uma potência k de g tal que $n = g^k \bmod p$.

Suponha que Alice e Bob queiram definir uma chave secreta compartilhada usando o protocolo Diffie-Hellman para acordo da chave. Eles procedem como mostrado a seguir: Alice gera um valor privado aleatório a e Bob gera um valor privado aleatório b. Ambos a e b são retirados do conjunto de inteiros {1, ... , p-2}. A seguir, eles derivam os seus valores públicos usando os parâmetros p e g e os seus valores privados. O valor público de Alice é g^a mod p e o valor público de Bob é g^b mod p. A seguir, eles compartilham entre si os seus valores públicos. Finalmente, Alice computa $g^{ab} = (g^b)^a$ mod p, e Bob computa $g^{ba} = (g^a)^b$ mod p. Desde $g^{ab} = g^{ba} =$ k, Alice e Bob têm agora uma chave secreta k compartilhada.

O protocolo depende do problema de logaritmo discreto para sua segurança. Ele assume que é impraticável computacionalmente calcular a chave secreta compartilhada $k = g^{ab}$ mod p dados os dois valores públicos g^a mod p e g^b mod p quando o primo p é suficientemente grande. Maurer mostrou que quebrar o protocolo Diffie-Hellman é equivalente a computação de logaritmos discretos sob certas hipóteses.

O protocolo de troca de chave Diffie-Hellman é vulnerável a um ataque de *man-in-the-middle*. Neste ataque, a oponente Carol por exemplo, intercepta o valor público de Alice e envia o seu próprio valor público para Bob. Quando Bob transmitir o valor público dele, a Carol substitui este valor com o valor dela própria e envia isto à Alice. Carol e Alice concordam assim com uma chave compartilhada e Carol e Bob concordam com outra chave compartilhada. Depois desta troca, Carol simplesmente decripta qualquer mensagem enviada por Alice ou Bob e assim pode ler e alterar a mensagem antes de reencriptar com a chave apropriada e os transmitir à outra parte. Esta vulnerabilidade é possível porque o protocolo Diffie-Hellman não autentica os participantes da comunicação.

As possíveis soluções para este problemas incluem o uso de assinaturas digitais e outras variantes do protocolo. O protocolo Diffie-Hellman Autenticado ou o protocolo Estação-para-Estação (STS) foi desenvolvido por Diffie, Van Oorschot e Wiener em 1992 para impedir o ataque *man in the middle* no protocolo Diffie-Hellman. A imunidade é alcançada permitindo as duas partes autenticar um ao outro pelo uso de assinaturas digitais e de certificados de chave pública.

A idéia básica aproximada é a seguinte: antes da execução do protocolo, cada uma das partes, Alice e Bob obtém um par de chave pública/privada e um certificado para a chave pública. Durante o protocolo, Alice computa uma assinatura em certas mensagens e encobre o valor público g^a mod p e o Bob procede de modo semelhante. Embora Carol ainda possa interceptar as mensagens entre Alice e Bob, ela não pode forjar assinaturas sem a chave privada de Alice e a chave privada de Bob. Conseqüentemente, este protocolo aperfeiçoado evita o ataque *man-in-the-middle*.

Nos anos recentes, o protocolo Diffie-Hellman original foi compreendido como um exemplo de técnica de criptografia muito mais genérica, o elemento comum é a derivação de um valor secreto compartilhado (isto é, a chave) a partir da chave pública de uma parte com a chave privada da outra parte. As chaves públicas podem ser certificadas, de forma que as partes comunicantes possam ser autenticadas. O *draft* ANSI X9.42 ilustra algumas técnicas variantes do protocolo Diffie-Hellman original e um recente artigo de Blake-Wilson, Johnson e Menezes fornece algumas provas de segurança pertinentes.

Apêndice D - Syslog Sign

O protocolo Syslog-Sign adiciona ao protocolo Syslog as seguintes características: autenticação de origem, verificação da integridade da mensagem, resistência ao ataque *reprodução*, sequenciamento das mensagens e a possibilidade de detectar a falta de algumas mensagens. No entanto, não adiciona a privacidade na comunicação através da encriptação dos dados.

Este conjunto de características de segurança acarreta um mínimo de impacto as implementações atuais do Syslog. Nenhuma mudança foi realizada ao formato do pacote do syslog tradicional. A mensagem Syslog-Sign contém um bloco de assinatura dentro da campo MSG da mensagem do syslog.

O mecanismo utilizado para aumentar a confiabilidade deste protocolo (pois o Syslog utiliza o UDP como camada de transporte) é a retransmissão do certificado público (da mensagem de syslog-sign) algumas vezes. O número de retransmissões é configurável.

Apêndice E – SNMP versão 3

O protocolo SNMP foi inicialmente especificado no final dos anos 80 e rapidamente tornou-se um modo padrão para gerenciamento de rede IPs entre diversos fabricantes. Contudo, era limitado para acomodar uma série de funcionalidades necessárias para o gerenciamento das redes. Três melhorias solidificaram o SNMP como uma ferramenta poderosa para gerenciamento da rede: (i) a primeira foi a especificação RMON, a qual é baseada no SNMP, foi liberada em 1991. RMON foi revisado em 1995 e uma nova versão do RMON, conhecido como RMON2, foi publicado em 1997. RMON define uma MIB para gerenciamento remoto de LANs; (ii) o segundo melhoramento foi o lançamento do SNMPv2c em 1993 e sua revisão em 1995. SNMPv2c fornece mais funcionalidade e eficiência que a versão original do SNMP; (iii) finalmente, o SNMPv3 foi publicado em 1998. E definiu uma estrutura global para as atuais e futuras versões de SNMP **e adiciona características de segurança ao SNMP**.

Como citado na RFC 3411, "*Architecture for SNMP Management Frameworks*", um dos objetivos da nova versão era fornecer segurança a operação de SET, a qual era considerado a maior deficiência do SNMPv1 e SNMPv2c. O título de um artigo IEEE reforça ainda mais esta afirmação, *"SNMPv3: A Security Enhancement to SNMP"*, de Bill Stallings.

A RFC 2574, "*User-based Security Model (USM) for version 3 of the Simple Network Management Protocol (SNMPv3)*", descreve o uso do HMAC-MD5-96 e HMAC-SHA-96 como protocolos para autenticação e CBC-DES como protocolo para encriptação (privacidade).

Todas as três versões de SNMP são normalmente implementadas usando o protocolo de transporte UDP. Um trabalho publicado demonstra que o uso de SNMPv1/TCP/TLS tem desempenho superior ao SNMPv3/UDP com as mesmas opções de segurança implementadas. E neste trabalho é feita a seguinte assertiva: "... como o SNMP é implementado sobre o UDP, o uso do TLS não é uma opção e o IPSec poderia ser usado na camada 3. Contudo, se o SNMP fosse implementado sobre o TCP para melhorar o desempenho conforme descrito acima, o TLS seria uma opção natural ...".

Apêndice F – DNSSEC

O DNSSEC (*DNS Security Extensions*) é um mecanismo proposto para tornar o protocolo DNS seguro. É composto de uma série de extensões ao DNS, a qual fornece autenticação e integridade fim a fim e foi projetado para proteger a Internet de certos tipos de ataques, como *DNS spoofing*.

Todas as respostas as requisições no DNSSEC são digitalmente assinadas. E através da verificação da assinatura, o *resolver* pode checar se a informação é idêntica a mantida pelo servidor autorizado.

De acordo com Mockapetris, DNSSEC estabeleceria um selo a prova de falsificações para as aplicações como Web e E-mail. Hoje em dia, os *hackers* e *spammers* personificam (forjar credencial) fontes de informação da Internet de forma relativamente fácil. O DNSSEC pode aumentar a probabilidade de que um email é na verdade do domínio a qual indica e que a informação obtida de um servidor Web é realmente advinda do site Web desejado e não de um impostor. Hoje é possível

redirecionar as requisições HTTP ou FTP para domínios forjados através dos servidores maliciosos de DNS.

O protocolo DNS utiliza o protocolo de transporte UDP para as tradicionais requisições devido ao baixo *overhead* e melhor desempenho [6]. E utiliza o protocolo TCP para a funcionalidade de transferência das zonas.

Apêndice G – NTP versão 2

O protocolo NTP, *Network Time Protocol*, é utilizado para sincronizar os relógios dos computadores em alguma referência de tempo. Este protocolo usa a camada de transporte UDP embora possa ser facilmente adaptável a outros protocolos. É uma evolução do *Time Protocol* e da mensagem *ICMP Timestamp* mas foi especialmente projetado para manter a acurácia e robustez mesmo quando usado no ambiente da Internet o qual envolve múltiplos caminhos de roteamento, atrasos variáveis *(jitter)* e redes não confiáveis.

A primeira implementação do NTP tinha uma acurácia de centenas de milisegundos e a primeira especificação apareceu na RFC 778 com o nome de *Internet Clock Service*. O NTP foi publicado pela primeira vez na RFC 958 e descrevia principalmente o formato dos pacotes e alguns cálculos básicos envolvidos. A primeira especificação completa (com algoritmos) para o NTP versão 1 apareceu na RFC 1059.

O NTP versão 2 introduziu segurança básica com a autenticação através do uso de chaves simétricas usando DES-CBC e foi descrita na RFC 1119.

Combinando as boas idéias do DTSS (*Digital Time Synchronization Service*), um outro protocolo para sincronização de relógio, com o NTPv2 foi especificado então NTP versão 3, RFC 1305.

A próxima versão de NTP (versão 4) fornecerá novas características como configuração automática (*manycast mode*), confiabilidade, redução de tráfego na Internet e autenticação (usando criptografia de chave pública).

O uso do mecanismo de autenticação do NTP versão 3 é planejado apenas enquanto padrões de segurança e implementações na camada de rede ou transporte não estejam disponíveis para referido uso. O mecanismo opera na

camada de aplicação e foi projetado para proteger contra modificações não autorizadas na mensagem original. Utiliza um *crypto-checksum* computado pelo transmissor e verificado pelo receptor através de um conjunto de chaves previamente distribuídas e indexadas por uma chave identificadora incluída na mensagem. Contudo, não há previsto no NTP técnicas para distribuir ou manter chaves ou certificados.

Apêndice H – NFS versão 4

O protocolo NFS, *Network File System*, fornece um acesso remoto transparente a sistemas de arquivos compartilhados através de uma rede de comunicação. Este protocolo foi projetado para ser utilizado em diferentes máquinas, sistemas operacionais, arquitetura de redes e protocolos de transporte. Esta portabilidade é alcançada pelo uso das primitivas RPC, *Remote Procedure Call*, contruídas sobre uma XDR, *eXternal Data Representation*.

Através do protocolo NFS, os dados de arquivos importantes podem ser transmitidos ou recebidos de um servidor. E assim as questões como autenticação, privacidade e integridade dos dados deveriam ser endereçadas pelas implementações deste protocolo.

Do mesma forma que a versão anterior, o NFS versão 3 transfere ao protocolo RPC a função de autenticação e assume que a privacidade e integridade dos dados são providenciadas pela camada de transporte conforme disponibilizado por cada implementação do protocolo.

O protocolo NFS versão 4 finalmente acrescenta mecanismo de segurança sofisticados e torna obrigatório a implementação em todos os clientes. O NFSv4 é uma aplicação RPC, *Remote Procedure Call*, que utiliza RPC versão 2 e a correspondente representação XDR, *eXternal Data Representation*, conforme definido nas RFC 1831 e 1832.

A arquitetura RPCSEC_GSS definida na RFC 2203 tem que ser usada para fornecer uma segurança forte ao protocolo NFSv4. Ela é usada para extender a segurança básica oferecida pelo RPC. Com o uso do RPCSEC_GSS, vários mecanismos podem ser usados para disponibilizar autenticação, integridade e

privacidade ao protocolo NFS versão 4. O Kerberos V5 será usado como descrito na RFC 1964 para oferecer uma infra-estrutura de segurança. O mecanismo LIPKEY GSS-API descrito na RFC 2847 será usado para possibilitar o uso de senhas pelos usuários e chaves públicas de servidores (utiliza SPKM3). Com o uso do RPCSEC_GSS, outros mecanismos podem também ser especificados e usados para adicionar segurança ao NFSv4.

O NFSv4 requer suporte do RPC sobre protocolos de transporte com controle de congestionamento como o TCP e o SCTP. Embora muitas implementações clientes continuem a suportar RPC via datagramas UDP, este suporte será provavelmente retirado.

Apêndice I – RIP versão 2

O protocolo RIP, *Routing Internet Protocol*, é utilizado para troca de informações de roteamento entre *hosts* e *gateways*. Está baseado no algoritmo *Bellman-Ford* (vetor de distância). O formato do pacote e o protocolo foram baseados no programa *routed*, o qual estava incluído na distribuição Berkeley do UNIX. Utiliza o protocolo de transporte UDP.

O protocolo RIP versão 2 adicionou um mecanismo de autenticação em relação ao RIP versão 1. E a autenticação através de senha trafegada na rede sem encriptação foi definida na RFC 1387.

Posteriormente, na RFC 2082 foi definido o uso do algoritmo de autenticação confiável *Keyed MD5*, que foi originalmente proposto no SNMP versão 2, e além disso, adicionou-se também um número de sequência. A chave de autenticação nunca deve ser transmitida pela rede sem encriptação mas nenhum método de distribuição das chaves foi descrito. Com este método, as mensagens ficam protegidas contra adulterações ou falsificação (*Keyed-MD5*) e dificulta o ataque por *reprodução* (número de sequência). **Este mecanismo não acrescenta confidencialidade a comunicação.** Nesta RFC é mencionada a possibilidade do uso do IPSec para adicionar privacidade a comunicação. **E caso sejam utilizadas as trocas de tabelas RIP via *unicast* também pode ser utilizado o TLS/UDP.**

Apêndice J – PGPFone

PGPfone (*Pretty Good Privacy Phone*) é um pacote de software comercial que transforma o computador em um telefone seguro. **Utiliza técnicas de compressão de voz e protocolos de criptografia fortes para habilitar uma conversação em tempo real de telefone segura**. PGPfone recebe a voz atráves de um microfone e então continuamente digitaliza, comprime e encripta e a envia por um modem para a outra pessoa na extremidade, a qual também está executando o PGPfone. Toda a criptografia e os protocolos de compressão de voz são negociados de forma dinâmica e transparente, fornecendo uma natural interface de usuário semelhante ao uso de um telefone convencional. São usados protocolos de chave pública para negociar a chave simétrica sem a necessidade de canais seguros para tal. E todos os pacotes do PGPfone através da Internet utilizam o protocolo de transporte UDP na porta 4747.

O PGPfone faz uso de *acknowledgement* para o processo de *handshake* (*call setup*). Cada parte envia quatro pacotes e quatro *acks* e cada pacote é enviado somente após o pacote precedente for recebido pela outra parte. Utiliza também um número de sequência para verificação de erros de transmissão e para ajudar a sincronização do algoritmo simétrico de bloco, o qual utiliza o modo de operação contador.

O modo contador oferece as seguintes vantagens: ele não encripta ou decripta o dado original verdadeiramente, a encriptação e decriptação via contador pode ser pré-computada para cada pacote requerendo somente uma operação XOR logo que os dados se tornarem disponíveis. E isto permite reduzir a latência através da pré-computação do *keystream* encriptado durante o intervalo entre pacotes e antes que os dados de voz tornem-se disponível para encriptação ou decriptação. Similar ao modo CFB, o modo contador pode encriptar um tamanho arbitrário de bytes de dados sem necessitar de *padding*.

Para a troca e acordo das chaves é usado o algoritmo *Diffie-Hellman*. Um dos seguintes algoritmos de encriptação simétrico podem ser utilizados: CAST5 (128 bits), Triplo DES (168 bits) e BlowFish (192 bits). Para assegurar a integridade do processo de handshake e evitar dificultar o ataque *man-in-the-middle* é usado o algoritmo de hash SHA.

Para autenticação das partes é usada uma solução biométrica. O processo de autenticação consiste em ouvir a pessoa, com a qual se deseja conversar, ler algumas palavras solicitadas dentro de uma lista de 256, especialmente criada pelo protocolo para identificar verdadeiramente o indivíduo.

A atual versão do PGPfone não utiliza o protocolo RTP, mas informa que as futuras versões do protocolo terão formato similar a deste protocolo. É mencionado também que a versão atual não utiliza as características de controle de temporização suportadas pelo RTP que melhorariam o tratamento dos atrasos imprevisíveis dos pacotes na Internet.

É mais um aplicativo UDP que criou uma solução proprietária para adicionar segurança ao seu funcionamento.

Apêndice K - Connectionless LDAP

A especificação CLDAP, *Connection-less Lightweight Directory Access Protocol*, RFC 1798, foi publicada em 1995 como padrão proposto mas foi modificado para *Historic status* conforme mencionado na RFC 3352. Nesta RFC são explicados os motivos pelo qual o CLDAP foi descontinuado.

Este protocolo foi projetado para aplicações que requeriam a procura de uma pequena quantidade de informação armazenada em um diretório**. O protocolo evitava o *overhead* do estabelecimento (e fechamento) de uma conexão e as operações de *bind* e *unbind* necessárias nos protocolos de acesso a diretórios orientados a conexão.**

O CLDAP foi projetado para complementar a versão 2 do LDAP, RFC1777, *Lightweight Directory Access Protocol*. Sete anos após a sua publicação, o CLDAP não se tornou muito utilizado na Internet. Há algumas prováveis razões para isso:

- Limitada funcionalidade:
  - Somente leitura;
  - Somente anônimo;
  - Tamanho pequeno dos resultados;

- **Segurança deficiente:**
    - **Sem proteção a integridade;**
    - **Sem proteção a confidencialidade;**
- Suporte a internacionalização inadequado;
- Extensibilidade insuficiente;

A comunidade reconheceu no meio dos anos 90 que CLDAP precisava ser atualizado. Em resposta, o IETF criou um grupo de trabalho chamado *LDAP Extensions Working Group* (LDAPext WG) em 1997 comprometido com esta atualização. Mas este grupo está terminando sem produzir nenhuma atualização ao CLDAP. **Deve ser observado que a comunidade ainda tem interesse no desenvolvimento de um protocolo de acesso a diretório *connection-less*.** Caso haja interesse em acesso *connection-less* a serviços de diretórios baseados em X.500 é recomendado verificar o trabalho em andamento de Johansson e Hedberg.

A RFC 2830 descreve uma extensão ao LDAP versão 3 e tem como objetivo adicionar segurança ao protocolo através do TLS (*Transport Layer Security*). O objetivo da utilização do TLS com o LDAP é assegurar privacidade e integridade a comunicação e opcionalmente fornecer autenticação.