Universiteit Antwerpen
Departement Wiskunde-Informatica
2003-2004

MAN-IN-THE-MIDDLE AANVAL OP HET SSL PROTOCOL

Kristof Boeynaems

Proefschrift ingediend tot het behalen van de graad
LICENTIAAT IN DE WETENSCHAPPEN

Promotor: Prof. Dr. Serge Demeyer
Begeleider: Andy Zaidman

"Live as if you were to die tomorrow
Learn as if you were to live forever"
– Mahatma Gandhi

Woord van dank

Het schrijven van een thesis is een niet te onderschatten werk, met ontelbare invloeden van vele personen. Hoe graag ik ook al deze mensen expliciet zou bedanken; ik waag mij er niet aan, uit angst belangrijke personen te vergeten en tekort te doen. Daarom zal ik mij beperken tot de mensen die onlosmakelijk met mij of deze thesis verbonden zijn, en zo een rechtstreekse en duidelijk aanduidbare hulp zijn geweest bij het maken van deze studie.

Als eerste wil ik mijn promotor Prof. Dr. Serge Demeyer bedanken, voor het mogelijk maken van deze thesis, en de nuttige raadgevingen, klein in aantal, maar van grote waarde.

Daarnaast ben ik ook veel dank verschuldigd aan mijn begeleider, Andy Zaidman, die meer was dan een begeleider in naam, en zijn begeleidende rol op een geweldige manier in praktijk heeft waargemaakt, door altijd open te staan voor vragen, quasi onmiddellijk te antwoorden en daarnaast nog vele nuttige tips te geven.

Tenslotte wil ik mijn beide ouders bedanken, Bie en Luk, omdat elk groot werk van een inwonende zoon onvermijdelijk invloed heeft op henzelf. Mijn moeder verdient een bijzondere vermelding omdat zij erin slaagde talrijke spellingsfouten te vinden in een voor haar ongetwijfeld erg saaie en lange tekst.

Ik herhaal nog eens dat deze lijst niet beperkend is, en bedank hierbij alle mensen die niet in dit korte overzicht vermeld worden, maar toch hun steentje hebben bijgedragen bij het maken van deze thesis tot wat ze nu is.

Moesten er ondanks de onmisbare hulp van al deze mensen onverhoopt toch nog onnauwkeurigheden of zelfs regelrechte fouten in deze thesis staan, zijn deze volledig mijn verantwoordelijkheid.

Inhoudsopgave

Woord van dank iii

Inhoudsopgave iv

Abstract xii

1 Inleiding 1

1.1 Inleiding . . . 1

1.2 Het Internet Threat Model . . . 1

1.3 Thesis . . . 2

1.4 Overzicht . . . 3

1.5 Beperking . . . 3

2 Basiscomponenten van een veilige connectie 5

2.1 Inleiding . . . 5

2.2 Een netwerkmodel . . . 5

2.2.1 ISO/OSI model . . . 6

2.2.2 TCP/IP model . . . 6

2.2.3 Een hybride model . . . 7

2.3 Eigenschappen van een beveiligde communicatie . . . 7

2.4 Cryptografie: geheimschrift voor volwassenen . . . 8

2.4.1 Inleiding . . . 8

2.5 Symmetrische encryptie . . . 9

2.5.1 Stream ciphers . . . 11

2.5.2 Block ciphers 13
2.5.3 Het sleutelmanagementprobleem 16
2.6 Asymmetrische encryptie 17
2.6.1 RSA 18
2.6.2 Diffie-Hellman 19
2.7 Symmetrisch versus Asymmetrisch 19
2.8 Message Digest 20
2.9 Digitale Handtekening 21
2.10 Message Authentication Code 23
2.11 Certificaten 24
2.12 Veilige Random Getallen 25
2.12.1 Pseudo-Random Number Generators (PRNG) 26
2.12.2 Hardware Random Number Generators (HRNG) 27
2.13 Cryptografische technieken en veilige communicatie 27

3 Het SSL protocol **28**
3.1 Inleiding 28
3.2 Ontstaan van SSL 28
3.2.1 Doel 28
3.2.2 Geschiedenis 30
3.3 Het SSL Protocol 31
3.4 SSL Handshake Protocol 32
3.4.1 Inleiding 32
3.4.2 Een eerste handshake 32
3.4.3 Een dynamische handshake 34
3.4.4 Integriteitscontrole 34
3.4.5 Meerdere geheime sleutels 36
3.4.6 De eigenlijke SSL handshake 37
3.4.7 Sessies 39
3.5 SSL Record Protocol 40

3.5.1 Gegevenstransfer 40
3.5.2 Alerts 43
3.6 Voorwaarden voor een veilige SSL connectie 43

4 Details van het SSL protocol 45

4.1 Inleiding 45
4.2 Implementatie details 45
4.2.1 Negotiatie 45
4.2.2 Sleutelafleiding 48
4.2.3 Finished berichten 51
4.2.4 Record Protocol 53
4.2.5 Beëindigen van een SSL connectie 55
4.3 Uitbreiding: Ephemeral RSA 56
4.3.1 Exportregulering 56
4.3.2 Ephemeral RSA 58
4.3.3 Server Gated Cryptography 61

5 Het SSL Protocol aangevallen 62

5.1 Inleiding 62
5.2 Analyse van het SSL protocol 63
5.3 Rechtstreekse aanval op SSL 65
5.3.1 Inleiding 65
5.3.2 Verkeersanalyse (‘traffic analysis’) 65
5.3.3 Replay attack 66
5.3.4 Versleutelsetverlaging (‘cipher suite rollback attack’) 66
5.3.5 Versieverlaging (‘version rollback attack’) 67
5.3.6 Principe van Horton 68
5.3.7 Overzicht van rechtstreekse aanvallen op SSL 69
5.4 Onrechtstreekse aanval op SSL 70
5.4.1 Inleiding 70

5.5 RSA implementatie aangevallen . . . 71
5.5.1 Inleiding . . . 71
5.5.2 Timing aanval . . . 71
5.5.3 Bleichenbacher aanval . . . 73
5.6 CBC implementatie aangevallen . . . 76
5.6.1 Inleiding . . . 76
5.6.2 CBC padding aanval: algemeen . . . 76
5.6.3 CBC padding aanval: een specifieke aanval . . . 77
5.7 Certificaat implementatie aangevallen . . . 78
5.7.1 Inleiding . . . 78
5.7.2 Authenticiteit in SSL . . . 79
5.7.3 Certificaatvervalsing . . . 80
5.7.4 Certificaatinjectering . . . 80
5.7.5 Phishing Scam . . . 81
5.7.6 Schaakgrootmeester-aanval . . . 82
5.8 Overzicht van onrechtstreekse aanvallen op SSL . . . 84
5.9 Frontale aanval op de onderliggende cryptografie van SSL . . . 84
5.9.1 Inleiding . . . 84
5.9.2 Digitale handtekeningen en asymmetrische encryptie aangevallen . . 88
5.9.3 Symmetrische encryptie aangevallen . . . 90
5.9.4 Message digest aangevallen . . . 92
5.9.5 Overzicht van frontale aanvallen op SSL . . . 93
5.10 SSL Exploits . . . 93
5.11 Voorwaarden voor een veilige SSL connectie herbekeken . . . 94

6 Man-in-the-middle aanval **97**

6.1 Inleiding . . . 97
6.2 Man-in-the-middle aanval . . . 98
6.2.1 Definitie . . . 98
6.2.2 Toepasbaarheid . . . 98

6.3 ARP vergiftiging (‘ARP poisoning’) 99
6.3.1 ARP 99
6.3.2 ARP Vergiftiging 100
6.3.3 Tegenmaatregelen 101
6.3.4 Tools 102
6.4 DNS bedrog (‘DNS spoofing’) 102
6.4.1 DNS 102
6.4.2 DNS bedrog 102
6.4.3 Tegenmaatregelen 103
6.4.4 Tools 103
6.5 Andere MITM aanvallen 103
6.6 Toepasbaarheid: overzicht 104
6.7 Na een MITM aanval 104
6.8 Een MITM aanval in praktijk 105
6.8.1 Inleiding 105
6.8.2 Het netwerk 105
6.8.3 ARP Vergiftiging in praktijk 106
6.9 Conclusie 108

7 Versleutelsetverlagingsaanval 109

7.1 Inleiding 109
7.2 Overzicht 109
7.3 ClientHello vervalsen 111
7.4 Finished vervalsen 112
7.4.1 Symmetrische encryptiesleutels 113
7.4.2 MAC over Finished 114
7.4.3 MAC over handshake 115
7.4.4 Conclusie 115
7.5 Voorwaarden voor een succesvolle versleutelsetverlaging 116
7.6 De aanval in praktijk: voorbereiding 116

7.6.1 Het netwerk 116
7.6.2 Voorwaarden aangepakt 116
7.7 Implementatie van de versleutelsetverlagingsaanval 118
7.7.1 Gebruikte softwarebibliotheken 119
7.7.2 Pakketten vangen 120
7.7.3 Pakketten voorverwerken 122
7.7.4 Pakketten besnuffelen 124
7.7.5 Pakketten wijzigen 125
7.7.6 Pakketten versturen 127
7.7.7 Decryptie van de `pre_master_secret` 128
7.7.8 Resultaten 129
7.8 Conclusie 129

8 Toepasbaarheid van de aanval in de echte wereld 130
8.1 Inleiding 130
8.2 SSL servers onderzocht 130
8.2.1 Inleiding 130
8.2.2 Extern onderzoek 131
8.2.3 Eigen onderzoek 133
8.2.4 Conclusie 135
8.3 SSL clients onderzocht 135
8.3.1 Inleiding 135
8.3.2 Meestgebruikte browsers 135
8.3.3 Kwetsbaarheid 136
8.3.4 Conclusie 137
8.4 Conclusie 137

9 Een versleutelsetverlagingsaanval op SSL voorkomen 138
9.1 Inleiding 138
9.2 Beschermen tegen MITM aanvallen 139

9.2.1 MITM aanval voorkomen 139
9.2.2 MITM aanval detecteren 140
9.2.3 Conclusie 141
9.3 Zwakke asymmetrische encryptie voorkomen 141
9.3.1 Versleutelsetuitsluiting 142
9.3.2 Versleutelsetbeperking 143
9.3.3 Frequentie van ephemeral sleutel-generatie 143
9.3.4 Beschermen van de Ephemeral RSA handshake 144
9.3.5 Ephemeral RSA verwijderen 145
9.3.6 Uitbreiding van de bescherming tegen versieverlaging 145
9.3.7 Conclusie 145

10 Relativering en besluit **147**

10.1 Inleiding 147
10.2 Relativering aan de hand van het *goed genoeg* principe 147
10.2.1 Goed genoeg principe (‘Good enough principle’) 147
10.2.2 Relativering 148
10.3 Relativering aan de hand van het *zwakste schakel* principe 149
10.3.1 Zwakste schakel principe (‘Weakest link principle’) 149
10.3.2 Relativering 149
10.4 Besluit 151

Nawoord **152**

Bibliografie **154**

Bijlages **157**

A SSL Details **157**

A.1 SSL/TLS versleutelsets 157
A.2 SSL/TLS alerts 158

B Ethereal dumps **159**

B.1 Inleiding 159

B.2 ARP vergiftiging 160

B.3 Versleutelsetverlagingsaanval 161

B.3.1 Overzicht 161

B.3.2 Manipulatie van eerste keuze 163

B.3.3 NULL overschrijving 164

B.4 Versleutelsetverlagingsaanval bij een beschermde Ephemeral RSA handshake 165

Lijst van tabellen **167**

Lijst van figuren **167**

Aantekeningen **169**

Abstract

Het **Secure Socket Layer (SSL) Protocol** is een veelgebruikte techniek om netwerkcommunicatie, en meer in het bijzonder communicatie over het internet, te beveiligen.

Deze thesis gaat na hoe goed SSL in dit opzet slaagt, door bekende zwakheden van het SSL protocol te onderzoeken, met een bijzondere interesse voor *man-in-the-middle (MITM)* aanvallen. Eén MITM aanval in het bijzonder, de *versleutelsetverlagingsaanval*, wordt diepgaand besproken en praktisch gedemonstreerd.

Daarnaast wordt de toepasbaarheid van de versleutelsetverlagingsaanval onderzocht op basis van een eigen onderzoek naar de kwetsbaarheid van reële SSL implementaties, zowel servers als clients, op het internet.

Hoofdstuk 1

Inleiding

"I quote others only in order the better to express myself."
– Michel De Montaigne

1.1 Inleiding

Een 100% veilig netwerk is een utopie, en niet realiseerbaar, zoals we intuïtief wel begrijpen. Elk systeem, hoe goed beveiligd ook, heeft zijn zwakheden, en kan gekraakt worden door een gemotiveerde aanvaller met de juiste (en voldoende) middelen. Het is de taak van de netwerkbeveiliger om de kosten van een mogelijk succesvolle aanval zo hoog te maken, dat de kost van de inbraak de waarde van de verkregen informatie niet waard is. Tegelijkertijd mag ook de kost van de beveiliging de waarde van de informatie in kwestie niet overstijgen.

Deze kosten- en baten-afweging wordt gemaakt door een *bedreigingsmodel* ("threat model") op te stellen. Door middel van dit model probeert de beveiligingsspecialist de middelen van een mogelijke aanvaller in te schatten, en een beeld te schetsen van de mogelijke aanvallen die een aanvaller kan (en wil) lanceren. De beveiligingsspecialist definieert welke aanvallen een echte bedreiging zijn, en welke aanvallen hij buiten beschouwing laat, wegens onrealistisch. Vervolgens probeert hij de realistische bedreigingen zo goed mogelijk aan te pakken met de middelen die hij ter beschikking heeft.

1.2 Het Internet Threat Model

Netwerkbeveiligingsspecialisten die zich bezighouden met beveiligde communicatie over het internet gaan uit van het volgende model, dat bekend staat als het **Internet Threat Model** [60, p.1-2]:

1. De *eindsystemen* waartussen de communicatie zich afspeelt, zijn *veilig*.

   De communicatie beveiligen tussen twee eindsystemen, waarvan één (of zelfs beide) systeem gecontroleerd wordt door de aanvaller, is extreem moeilijk, als het al niet onmogelijk is.
   Aan deze veronderstelling koppelen we drie nuanceringen.

(a) De compromittering van één enkel eindsysteem mag de veiligheid van andere communicaties dan die met dit eindsysteem niet beïnvloeden. Dit noemt men de eis van **geen** "*enig punt van falen*"("single point of failure").

(b) Een aanvaller kan over een *gelegitimeerd eindsysteem* beschikken, we eisen alleen dat gebruikers uit mogen gaan van het feit dat hun eigen systeem niet gecompromitteerd is.

2. Het *kanaal* waarover de communicatie verloopt, wordt als *volstrekt onveilig* beschouwd.

   Een aanvaller kan pakketten in het netwerk injecteren, pakketten van het netwerk halen en bekijken, en pakketten veranderen. Gebruikers moeten ervan uitgaan dat elk pakket dat ze via het netwerk aankrijgen een pakket kan zijn dat mogelijk afkomstig is van, of veranderd door, een aanvaller. Daarenboven moeten zij ervan uitgaan dat elk pakket dat zij zelf versturen, door een aanvaller bekeken en/of gewijzigd kan worden.

Een gevolg van de laatste veronderstelling is het feit dat een aanvaller dus eenvoudigweg een bestaande connectie kan beëindigen, door alle pakketten op die connectie te verwijderen. Dit is een vorm van een **Denial-of-Service**-aanval (**DOS**). Een andere DOS-aanval doet juist het omgekeerde: er worden zoveel connecties naar één bepaald systeem gelegd dat dit systeem, in een poging elke connectie te beantwoorden, volledig overbelast geraakt en zo onbereikbaar wordt. DOS-aanvallen zijn echter zo moeilijk te voorkomen dat ze meestal niet als een 'realistische' bedreiging in het bedreigingsmodel worden opgenomen.

Ook maken we een onderscheid tussen twee grote categorieën van bedreigingen. Een aanval waarin de aanvaller niets méér doet dan pakketten van het kanaal bekijken, noemen we een *passieve* aanval. Daartegenover staat de *actieve* aanval, waarin de aanvaller pakketten wijzigt, verwijdert of toevoegt.

Het komt er dus op aan om de capaciteiten van de aanvaller goed in te schatten, aangezien het hele bedreigingsmodel hierop gebaseerd is. Een significante onderschatting van de bedreigingen maakt het systeem kwetsbaar voor een geslaagde aanval, maar ook een overschatting kan het model totaal nutteloos maken. Een belangrijke overschatting kan immers tot een onaanvaardbaar hoge beveiligingskost leiden, of zelfs een onbruikbaar systeem, omwille van de te hoge beveiliging.

1.3 Thesis

Zodra een beveiligingsspecialist een bedreigingsmodel heeft opgesteld, kan hij maatregelen beginnen te ontwikkelen om het netwerk tegen de als realistisch aangenomen bedreigingen te beschermen.

Hierbij is het onontbeerlijk dat hij zo nauwkeurig mogelijk de kracht van potentiële tegenmaatregelen kan inschatten.

Deze thesis probeert een beveiligingsspecialist daarbij te helpen door één, erg brede, tegenmaatregel te onderzoeken, het **SSL Protocol**.

Wij onderzoeken de vraag:

Hoe goed beschermt het *SSL Protocol* netwerkcommunicatie tegen bedreigingen uit het *Internet Threat Model*?

1.4 Overzicht

Om onze thesisvraag te beantwoorden, beginnen we in Hoofdstuk 2 met een aantal inleidende begrippen voor te stellen, met als doel de lezer voldoende vertrouwd te maken met het domein van de netwerkbeveiliging om de rest van deze studie te begrijpen.

Vervolgens geven we in Hoofdstuk 3 een overzicht van het SSL Protocol, bespreken we uitgebreid de werking van dit protocol, en gaan we in Hoofdstuk 4 in op een aantal implementatiedetails; dit alles om de lezer een diepgaand begrip van het SSL protocol bij te brengen.

Na deze inleidende hoofdstukken komen we in Hoofdstuk 5 tot de kern van deze thesis, waar we een uitgebreide bespreking geven van alle mogelijke zwakheden die in het verleden in het SSL Protocol ontdekt zijn, samen met de aanvallen die deze zwakheden uitbuiten, en tegenmaatregelen om deze zwakheden uit het protocol te remediëren.

Uit deze bespreking leiden we af dat één aanval in het bijzonder, de *versleutelsetverlagingsaanval*, meer onderzoek verdient, omdat het SSL Protocol hier onvoldoende tegen beschermd lijkt.

De versleutelsetverlagingsaanval maakt gebruik van een *man-in-the-middle* aanval, een categorie aanvallen die we in Hoofdstuk 6 uitgebreid bespreken, en waarvan we een praktische demonstratie geven.

Hierna gaan we met Hoofdstuk 7 dieper in op de versleutelsetverlagingsaanval zelf, die we theoretisch uitwerken en praktisch demonstreren.

In Hoofdstuk 8 gaan we na hoe toepasbaar de door ons geïmplementeerde versleutelsetverlagingsaanval in 'echte' omstandigheden is, in eerste instantie op basis van een extern onderzoek, maar vooral aan de hand van een uitgebreider eigen onderzoek.

Na zo besloten te hebben dat een versleutelsetverlagingsaanval praktisch haalbaar is, en zelfs relatief toepasbaar, stellen we in Hoofdstuk 9 in detail een aantal tegenmaatregelen voor om deze aanval erg te bemoeilijken, of zelfs totaal te verhinderen. Bovendien implementeren we zelf een sluitende tegenmaatregel, en demonstreren deze praktisch.

We eindigen in Hoofdstuk 10 met een relativering van de bereikte resultaten, en een kort besluit.

1.5 Beperking

In deze studie beperken we ons tot SSL met **RSA** als sleuteluitwisselingsalgoritme.

Naast RSA biedt SSL sinds versie 3.0 ook ondersteuning voor **Diffie-Hellman (DH)**, maar om de studie enigszins te beperken beschouwen we dit minder bekende mechanisme niet.

Dit wil niet zeggen dat we dit algoritme compleet negeren, daar waar onvermijdelijk brengen we DH kort ter sprake. Wel betekent dit dat wij geen actief onderzoek hebben gedaan naar mogelijke veiligheidsproblemen in SSL tengevolge van een gebruik van DH.

Alle resultaten van deze studie moeten steeds beschouwd worden vanuit deze beperking.

Hoofdstuk 2

Basiscomponenten van een veilige connectie

"Even Bach comes down to the basic suck, blow, suck, suck, blow."
– Mouth organist Larry Adler

2.1 Inleiding

Het is het doel van dit hoofdstuk om de lezer kennis te laten maken met de belangrijkste concepten uit het domein van de netwerkbeveiliging, en hem zo de noodzakelijke bagage mee te geven om de rest van deze studie te begrijpen.

Er wordt begonnen met een korte bespreking van een belangrijk netwerkmodel. Hierna wordt een overzicht gegeven van de mogelijke bedreigingen waaraan een netwerk (en dus de communicatie op dit netwerk) blootgesteld is, waarna enkele eigenschappen worden voorgesteld die men van een veilig netwerk kan verlangen. Vervolgens bespreken we kort het domein van de cryptografie, stellen we enkele cryptografische algoritmen voor en laten we zien hoe we deze algoritmen kunnen toepassen om de eerder besproken eigenschappen te bekomen.

Dit hoofdstuk is grotendeels gebaseerd op [60], [94] en [72].

2.2 Een netwerkmodel

Communicatie over een netwerk is een complexe aangelegenheid. Conceptueel wordt de verantwoordelijkheid voor het versturen van data gespreid over verschillende *lagen*, waarbij elke laag zich baseert op de onderliggende laag, en steeds een extra abstractieniveau inbouwt. Binnen elke laag wordt gecommuniceerd aan de hand van bepaalde regels en conventies ('*protocols*'), eigen aan deze laag. Een laag is verbonden met de laag er vlak boven door middel van een *interface*.

2.2.1 ISO/OSI model

Dit lagenmodel wordt gestandaardiseerd in het **International Standard Organization's Open System Interconnect (ISO/OSI)** model, waarin zeven lagen gedefinieerd worden:

- **Fysische laag**
  Dit is de onderste laag en beschrijft de fysische eigenschappen van het medium waarover gecommuniceerd wordt. Het bepaalt tevens de elektrische eigenschappen en interpretatie van de uitgewisselde signalen.
- **Datalink laag**
  Beschrijft de logische organisatie van de databits die over een bepaald medium verstuurd worden. Het meest gebruikte protocol op deze laag is **Ethernet**.
- **Netwerklaag**
  Beschrijft hoe een opeenvolging van gegevensuitwisselingen over verschillende datalinks de gegevens kunnen afleveren tussen om het even welke twee nodes in een netwerk. Het standaardprotocol op deze laag is het **Internet Protocol (IP)**.
- **Transportlaag**
  Beschrijft de kwaliteit en de eigenschappen van de gegevenstransfer. Hier wordt meestal het **Transmission Control Protocol (TCP)** of **User Datagram Protocol (UDP)** gebruikt.
- **Sessielaag**
  Beschrijft de organisatie van datasequenties die langer zijn dan de pakketten die behandeld worden door de onderliggende lagen.
- **Presentatielaag**
  Beschrijft de syntax van de data die getransfereerd wordt.
- **Applicatielaag**
  Deze laag omvat de applicatie protocollen zoals het **Hypertext Transfer Protocol (HTTP)** en het **File Transfer Protocol (FTP)**.

Dit lagenmodel blijkt echter vooral theoretisch, en weinig aangepast aan de praktijk. De belangrijkste kritiek op dit model is het feit dat de datalink en netwerklaag erg vol zitten, terwijl de presentatie- en sessielaag zo goed als leeg zijn [85, p. 40-43].

2.2.2 TCP/IP model

Bij het ISO/OSI model werd eerst het theoretische model ontwikkeld, waarna later de bijhorende protocols volgden. Het TCP/IP model ging echter de andere richting: eerst waren er de protocols, waarna aan de hand van deze protocols een theoretisch model werd opgesteld. Dit model is gebaseerd op de praktische implementatie van ARPANET, de voorloper van het internet, en bestaat uit slechts vier lagen:

- **Host-to-network laag**
  Deze laag is de onderste, en omvat de fysische en datalink laag van het ISO/OSI model.
- **Internetlaag**
  Deze laag is vergelijkbaar met de netwerklaag in het vorige model, waarop IP loopt.
- **Transportlaag**
  Deze laag komt overeen met de transportlaag van het ISO/OSI model, en bevat TCP en UDP.
- **Applicatielaag**
  Alle hogere protocols opereren binnen deze laag. Deze laag is gelijk aan de applicatielaag van het ISO/OSI model.

We zien dat de presentatie- en sessielaag niet in dit model aanwezig zijn, een goed punt, aangezien deze in praktijk amper gebruikt blijken te worden. Daarentegen zijn de fysische en de datalink laag gemengd, een duidelijk nadeel omdat zo'n onderscheid in praktijk wel erg nuttig is. Verder heeft dit model nog andere belangrijke nadelen, die vooral een gevolg zijn van het feit dat de *ad hoc* wijze waarop dit model werd opgesteld [85, p. 43-44].

2.2.3 Een hybride model

Beide modellen hebben dus hun problemen, daarom stelt Andrew Tanenbaum [85, p. 44] een hybride model voor, grotendeels gebaseerd op het ISO/OSI model, maar met slechts vijf lagen:

- **Fysische laag**
- **Datalink laag**
- **Netwerklaag**
- **Transportlaag**
- **Applicatielaag**

Het is dit model dat we in deze studie zullen hanteren.

2.3 Eigenschappen van een beveiligde communicatie

Communicatiebeveiliging is geen monolithische eigenschap, maar bestaat zelf uit een aantal verschillende, in min of meerdere mate gerelateerde, eigenschappen. Afhankelijk van de mogelijke bedreigingen, zal een beveiligingsspecialist voor een bepaald communicatiekanaal één of meerdere van deze eigenschappen trachten te bekomen, om het slagen van een mogelijke aanval te voorkomen.

Eric Rescorla [60, p.3-4] onderscheidt de drie volgende eigenschappen:

1. **Vertrouwelijkheid**
   De informatie die over het communicatiekanaal wordt uitgewisseld blijft geheim.

2. **Integriteit**
   Meer bepaald integriteit van de berichten in de communicatie. Berichten worden onderweg niet onopgemerkt gewijzigd[1].

3. **Authenticiteit**
   Er wordt gecommuniceerd met diegene waarmee men denkt te communiceren.

Andere auteurs maken een andere opdeling, of voegen hier nog eigenschappen[2] aan toe, maar deze partitionering is nauwkeurig genoeg voor onze bespreking.

Om deze eigenschappen te verduidelijken, volgt hier een herkenbare situatie:

> Twee lagere school-kinderen willen tijdens de les met elkaar communiceren met behulp van geschreven berichtjes. Zij zitten echter beide aan een ander uiteinde van het klaslokaal, en moeten dus rekenen op hun klasgenoten om de berichten bij elkaar te laten aankomen. Toch willen zij niet dat een van de andere kinderen hun berichten leest. De kinderen willen dus *vertrouwelijkheid*, ondanks het *onveilige communicatiekanaal*. Zij bereiken die vertrouwelijkheid door hun op papier geschreven boodschappen dicht te plakken met een plakbandje. Tegelijkertijd garandeert dat plakbandje de *integriteit* van hun boodschap: als iemand onderweg hun boodschap heeft proberen te openen, zal de ontvanger dit merken aan een gescheurd plakbandje. Verder kennen beiden elkaars handschrift zo goed, dat zij zo de *authenticiteit* van de berichten kunnen controleren.

Ook volwassenen hebben in de loop der tijd een aantal (meer geavanceerde) technieken ontwikkeld om deze eigenschappen te garanderen. Zo ontstond de cryptografie[3], "*de wetenschap van het geheimschrift*".

2.4 Cryptografie: geheimschrift voor volwassenen

2.4.1 Inleiding

Cryptologie[4] is het studiegebied dat de steganografie, cryptografie en cryptanalyse omvat.

Steganografie[5], probeert het *bestaan* van het bericht *zelf* te verbergen, in tegenstelling tot de cryptografie, die de *informatie in* het bericht zal verbergen.

[1] Deze term wordt hier heel ruim genomen. Ook een bericht volledig verwijderen, valt hier onder 'wijzigen'

[2] een eigenschap die hier niet behandeld wordt, hoewel dikwijls erg belangrijk (vooral juridisch), is *onweerlegbaarheid* of *non-repudiatie* ('*non-repudation*'). Deze eigenschap houdt o.a. in dat een partij niet kan ontkennen een bericht verstuurd te hebben

[3] Grieks voor "geheim schrijven", "geheimschrift"

[4] Van het Griekse *kryptós lógos*, "geheim woord"

[5] Grieks voor "beschermd schrijven"

Cryptanalyse is de tegenhanger van cryptografie, een cryptanalist probeert informatie, beveiligd met behulp van cryptografische technieken, te weten te komen.

Cryptografie heeft de bedoeling een bericht onleesbaar (voor onbevoegde partijen) te maken, hoewel het bericht zelf volledig blootgesteld wordt aan de buitenwereld.

De bekendste cryptografische techniek, en de basis van alle anderen, is ongetwijfeld die van *encryptie* en *decryptie*.

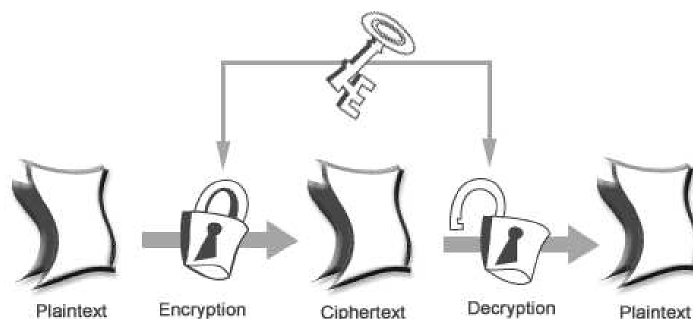
Encryptie is het omzetten van gewone tekst ('*plaintext*') naar geheimschrift ('*ciphertext*'), en decryptie is de omgekeerde operatie, het terug vertalen van geheimschrift naar de oorspronkelijke tekst. Dit omzetten gebeurt aan de hand van een *geheime sleutel*. De wijze waarop dit gebeurt wordt bepaald door een *cryptografisch algoritme* ('*cipher*'), een wiskundige functie die plaintext in ciphertext zal omzetten aan de hand van de sleutel en vice versa. Encrypteren noemt men ook wel *versleutelen*.

In de volgende secties wordt dieper ingegaan op de cryptografische algoritmen achter encryptie/decryptie, en hierop gebaseerde cryptografische technieken.

2.5 Symmetrische encryptie

Het eenvoudigste geval van encryptie is *symmetrische encryptie*. Algemeen gezien behelst de symmetrische encryptie alle encryptiemethodes waarbij de decryptiesleutel afgeleid kan worden uit de encryptiesleutel, en vice versa. In praktijk betekent dit meestal dat eenvoudigweg eenzelfde geheime sleutel voor zowel encryptie als decryptie wordt gebruikt. Om deze reden wordt symmetrische encryptie ook wel **geheime sleutel-cryptografie** ('*secret key cryptography*') genoemd.

Het schema in Figuur 2.1 illustreert dit mechanisme.



Figuur 2.1. Symmetrische encryptie

Laten we deze nieuwe kennis toepassen op het voorbeeld van de twee kinderen:

De twee vriendjes ondervinden al snel dat de andere klasgenoten weinig respect hebben voor het plakbandje op hun berichten en de berichten simpelweg openen en lezen.
Merk op dat zij in dit geval wel merken wanneer hun bericht gelezen is (aan het kapotte plakbandje), en dat dus aan de integriteitseigenschap voldaan blijft.

Maar aangezien de vertrouwelijkheidseigenschap uit hun communicatie is verdwenen, zoeken zij een betere methode om hun privacy terug te winnen.

Eén van de twee ontwerpt een geheimschrifttabel zoals in Tabel 2.1.

Letter	Symbool		Letter	Symbool		Letter	Symbool
A	$\mp$		J	$\bigotimes$		R	$\wr$
B	$\bigcap$		K	$\bigodot$		T	$\bigtriangledown$
C	$\uplus$		L	$\ll$		U	$\oplus$
D	$\sqcup$		M	$\in$		V	$\ominus$
E	$\bigvee$		N	$\times$		W	$\oslash$
F	$\backslash$		O	$\cup$		X	$\bigcirc$
G	$\bigtriangleup$		P	$\biguplus$		Y	$\subset$
H	$\triangleright$		Q	$\bigsqcup$		Z	$\vdash$
I	$\bigoplus$		R	$\wedge$			

Tabel 2.1. Een kinderlijk geheimschrift

De kinderen sturen hun berichtjes nu niet meer in plaintext naar elkaar, maar versleutelen het bericht door elke letter te vervangen door zijn overeenkomstige symbool, zoals voorgesteld in de geheimschrifttabel.

De sleutel bestaat in dit geval dus uit een tabel waarin voor elke letter een speciaal symbool is verzonnen[6]. Als het kind op deze manier een bericht encrypteert, zullen enkel de kinderen die ook over deze sleutel beschikken het bericht kunnen ontcijferen (decrypteren). Natuurlijk beschikken enkel de beide vriendjes over deze sleutel, en wordt dezelfde sleutel (tabel) gebruikt bij het encrypteren en decrypteren. Zij passen dus een eenvoudige vorm van geheime sleutel-cryptografie toe.

We merken hierbij nog op dat de sleutel niet altijd uit zo'n volledige geheimschrifttabel moet bestaan. Een sleutel kan ook uit een stukje kennis bestaan, waarmee beide partijen dan vervolgens de volledige tabel afleiden.

Een voorbeeld van deze laatste methode is de encryptie die Julius Caesar gebruikte voor zijn oorlogscorrespondentie. De sleutel van elke *Caesar Cipher* bestond uit het aantal plaatsen dat een letter in het alfabet verschoven moest worden om de overeenkomende symboolletter te verkrijgen.

[6] Dit 'verzinnen' kan gezien worden als het genereren van *random data*

2.5.1 Stream ciphers

Symmetrische encryptie-algoritmen kunnen ingedeeld worden in twee grote groepen, **stream ciphers** en **block ciphers**. We beginnen met een bespreking van de eerste groep.

Werking

Stream cipher algoritmen werken byte-georiënteerd. De encryptiesleutel wordt intern eerst omgezet in een *keystream*, een eindeloze sequentie data, die in het ideale geval puur random[7] is, maar in praktijk altijd een zich herhalend patroon zal vertonen. Hoe beter de keystream het pure random-ideaal benadert, hoe veiliger de stream cipher.

Om vervolgens een bericht (plaintext) te encrypteren, wordt één byte van dit bericht met één byte van de keystream gecombineerd, om zo één byte ciphertext te bekomen. De combinatie van keystream byte en plaintext byte gebeurt meestal met behulp van de exclusieve-of (XOR)[8] operatie omdat die een aantal nuttige eigenschappen bezit:

- **De output is uniek**
  Voor elke mogelijke sleutel vormt de XOR-operatie een *one-to-one* functie. Als sleutel en ciphertext bekend zijn, kan de plaintext op ondubbelzinnige wijze gevonden worden. Met andere woorden, twee verschillende plaintexts zullen nooit tot eenzelfde ciphertext XOR'n.
- **XOR is de inverse functie van zichzelf**
  Voor elke sleutel K geldt (met ⊕ de XOR-operatie): `(B ⊕ K) ⊕ K = B`.

De combinatie van een keystream byte met een plaintext byte ziet er dan als volgt uit:

```
C[i] = KS[i] ⊕ B[i]

(C[i] is de i-de byte ciphertext, KS[i] de i-de byte uit de keystream
en B[i] staat voor de i-de byte van het bericht.  ⊕ stelt de XOR-ope-
ratie voor)
```

Om de ciphertext terug om te zetten in het oorspronkelijke bericht, gebruiken we volgende operatie:

```
B[i] = KS[i] ⊕ C[i]
```

[7] In dit geval is de encryptie onbreekbaar, perfect veilig, zelfs ten opzichte van een aanvaller met een oneindige hoeveelheid rekenkracht. Men noemt dit een *one-time pad* of *Vernam cipher*.

[8] De XOR functie geeft 1 terug als exact 1 van beide inputs 1 is. In alle andere gevallen geeft deze functie 0.

Opmerkingen

Deze methode heeft echter, juist omwille van de symmetrie in encryptie en decryptie, een aantal eigenschappen met nadelige effecten vanuit beveiligingsoogpunt.
We beginnen met twee eigenschappen die de *vertrouwelijkheid* bedreigen:

- Gegeven twee berichten B_1 en B_2, beiden geëncrypteerd met dezelfde sleutel (en bijgevolg keystream) tot de respectievelijke ciphertexts C_1 en C_2. Een aanvaller die bericht B_1 (plaintext) te weten komt, kan via een eenvoudige operatie de keystream KS berekenen, en daarmee vervolgens B_2 ontcijferen, omwille van de volgende eigenschap:

  Gegeven B_1, C_1 en C_2:

  $$KS = B_1 \oplus C_1$$
  $$B_2 = KS \oplus C_2$$

  De situatie dat een aanvaller over een plaintext bericht B_1 beschikt, is niet zo uitzonderlijk als ze op het eerste gezicht misschien lijkt. Veel communicatieverkeer bevat, omwille van de gestandaardiseerde formaten (met bijhorende headers) waarin het verstuurd wordt, grote delen voorspelbare en repetitieve data die een aanvaller toelaten grote delen van een bepaald bericht te voorspellen, en zo dus deze eigenschap uit te buiten. Bovendien laat deze eigenschap de aanvaller ook toe om zelf plaintexts op een geldige manier te versleutelen.

- Maar zelfs zonder plaintext kan de aanvaller het effect van de keystream volledig elimineren, om een XOR van de twee plaintexts over te houden, door de volgende operatie uit te voeren:

  Gegeven twee ciphertexts C_1 en C_2:

  $$B_1 \oplus B_2 = C_1 \oplus C_2$$

  Aan de hand van deze XOR van twee plaintexts[9], kan een aanvaller de inhoud van deze plaintexts met grote nauwkeurigheid "raden"[14].

Het is dus duidelijk dat, indien we een stream cipher willen gebruiken voor meerdere berichten, we ofwel verschillende sleutels, ofwel verschillende secties van de keystream moeten gebruiken om de vertrouwelijkheid te garanderen.

Maar zelfs indien we ons hieraan houden, kunnen er problemen optreden. De volgende twee eigenschappen laten een aanvaller immers toe om de inhoud van het bericht te wijzigen, zelfs indien voor elk bericht een andere sleutel (keystream) gebruikt wordt. Deze eigenschappen vormen dus geen gevaar voor de vertrouwelijkheid, maar bedreigen wel de *integriteit* van de communicatie.

- Stel dat de aanvaller de plaintext B_1 en de ciphertext C_1 heeft bemachtigd. Met die gegevens is het voor hem een koud kunstje om de inhoud van het plaintext bericht B_1 te wijzigen door een nieuwe ciphertext C_2 voor een nieuw plaintext bericht B_2 te genereren, zelfs zonder dat hij de keystream kent, zoals de volgende eigenschap stelt:

[9] Met data met bekende eigenschappen, zoals tekst of kredietkaartnummers.

```
Gegeven B1, C1 en C2:
    C2 = C1 ⊕ (B1 ⊕ B2)
    (want C1 = KS ⊕ B1)
```

Dit is in feite een gevolg van de eerder besproken eerste eigenschap die de vertrouwelijkheid bedreigt.

- Er bestaat een *one-to-one mapping* tussen elke ciphertext en plaintext bit. Hierdoor kan de aanvaller (zelfs zonder de plaintext te kennen) *voorspelbare*[10] aanpassingen maken aan de plaintext, door bits te inverteren in de ciphertext. Zo'n aanpassing aan een ciphertext bit heeft dan een gelijkaardige invertering van de bijhorende plaintext bit tot gevolg.

Een stream cipher moet daarom *altijd* in combinatie met een extra techniek gebruikt worden om de integriteit te waarborgen. Een voorbeeld van zo'n techniek is een **Message Authentication Code (MAC)**, die verderop besproken wordt.

Algoritmen

De enige stream cipher die wijdverspreid gebruikt wordt, is **RC4**. RC4[11] werd in 1987 ontworpen door Ron Rivest voor RSA Data Security, Inc. RC4 is heel erg snel en heeft een variabele sleutellengte; de sleutel kan variëren tussen 8 en 2048 bits. Aangezien de sleutel geëxpandeerd wordt in een interne statentabel met een constante lengte, is het algoritme altijd even snel, onafhankelijk van de sleutellengte.

2.5.2 Block ciphers

We bespreken nu de andere soort symmetrische cryptografische algoritmen, block ciphers.

Werking

Block ciphers werken blok-georiënteerd. De te encrypteren data wordt opgesplitst in blokken (typisch 8 of 16 bytes groot), die dan elk afzonderlijk versleuteld worden. Die versleuteling kan men zien als een soort uitbreiding van de geheimschrifttabel uit het voorbeeld met de twee kinderen (Tabel 2.1). Men maakt nu een tabel met voor elk mogelijk plaintext blok een speciaal geheimschriftsymbool (in tegenstelling tot voor elke mogelijke letter). Daarenboven heeft men per blok niet één mogelijke kolom met een symbool, maar een hele verzameling van mogelijke symbolen, in verschillende kolommen. Welke kolom (en welk symbool) gekozen wordt, wordt bepaald door de sleutel. Natuurlijk zou zo'n tabel onhandelbaar groot zijn, daarom maakt men in praktijk gebruik van een speciale functie die zo'n tabel simuleert.

[10] We drukken hier op het feit dat de aanpassingen voorspelbaar zijn. Het is immers *altijd* mogelijk om ciphertexts aan te passen.

[11] Officieel staat "RC" voor "Rivest Cipher" hoewel wordt aangenomen dat het oorspronkelijk "Ron's Code" betekende.

Men gebruikt meer bepaald twee functies, een functie E voor encryptie, en een functie D voor decryptie. De functies zien er dan als volgt uit:

```
Met S de sleutel, B het plaintext bericht en C de ciphertext:

    C = E(S, B)
    B = D(S, C)
```

Op deze manier encrypteren en decrypteren we dus één afzonderlijk blok.

Electronic Codebook mode

Als we dit willen toepassen op berichten langer dan één blok, is de meest voor de hand liggende methode de volgende:

```
Met B[i] en C[i] het i-de blok van respectievelijk de plaintext en
ciphertext:

    C[i] = E(S, B[i])
    B[i] = D(S, C[i])
```

Deze methode noemt men **Electronic Codebook (ECB)** mode, we splitsen het bericht op in blokken, en passen de E en D functies toe op elke blok afzonderlijk. Deze methode heeft echter een eigenschap die erg nadelig is vanuit beveiligingsoogpunt:

> Stel dat twee blokken B[i] en B[j] aan elkaar gelijk zijn. Dan zullen ook de geëncrypteerde blokken C[i] en C[j] identiek zijn. Als we dus een patroon hebben in de plaintext B dat dikwijls voorkomt, zal dat te zien zijn in de ciphertext C, en daaruit kan een mogelijke aanvaller iets leren over de plaintext.

Cipher Block Chaining

Cipher Block Chaining (CBC) mode biedt een oplossing voor dit probleem, door de encryptie van een plaintext blok B[i] te laten afhangen van de ciphertext van het vorige blok C[i-1]:

```
C[i] = E(S, (B[i] ⊕ C[i-1]))
B[i] = D(S, C[i]) ⊕ C[i-1]
```

De nadelige eigenschap van ECB mode wordt hier in de meeste gevallen vermeden, aangezien twee identieke plaintext blokken waarschijnlijk niet dezelfde ciphertext zullen opleveren omdat hun vorige ciphertext verschillend is:

> Stel dat B[j] $=$ B[k] en B[j-1] $\neq$ B[k-1].
> Dan C[j-1] $\neq$ C[k-1] en bijgevolg C[j] $\neq$ C[k].

Het probleem wordt echter niet volkomen vermeden, aangezien het nog altijd mogelijk is dat twee data blokken B[i] en B[j] in eenzelfde ciphertext C vertaald worden (C[i] = C[j]). Als vervolgens B[i+1] = B[j+1] dan geldt ook C[i+1] = C[j+1]. In dit geval weet de aanvaller dus dat B[i+1] gelijk is aan B[j+1]. Men noemt dit *CBC rollover.* Gemiddeld blijken bij een cipher met blokgrootte X, twee ciphertexts te 'botsen' om de $2^{\frac{X}{2}}$ blokken.

CBC mode heeft wel een extra complicatie tot gevolg. De berekening van het eerste ciphertext blok kan niet op deze manier uitgevoerd worden, aangezien er nog geen voorgaand ciphertext blok bestaat. Om dit op te lossen zal een random blok gegenereerd worden, de *initialisatie vector (IV).* De IV wordt gebruikt als het *nulde* ciphertext blok, en wordt dus geXOR-ed met het eerste plaintext blok, om zo het eerste ciphertext blok te verkrijgen. Deze IV's moeten niet geheim zijn, maar wel *vers*, dat wil zeggen, een nieuwe IV moet gegenereerd worden voor elk bericht. Dit om te vermijden dat een aanvaller kan ontdekken dat de eerste ciphertext blokken van twee verschillende berichten gelijk zijn.

Naast CBC zijn er nog andere modes waarin block ciphers kunnen opereren, zoals **Output Feedback (OFB)** en **Cipher Feedback (CFB)**, maar CBC is de meest gebruikte, en de enige block cipher die gebruikt wordt in SSL.

Opmerkingen

Een belangrijk verschil tussen block en stream ciphers is het feit dat block ciphers de data gewoonlijk lichtjes expanderen. Deze expansie is het gevolg van het feit dat de meeste berichten niet precies een veelvoud zijn van de gebruikte blokgrootte, en dat er dus in de meeste gevallen een beetje data (onder de vorm van bytes) moet worden toegevoegd, om toch een veelvoud van de blokgrootte te bekomen. Dit proces wordt *padding* genoemd. Het laatste stukje van het bericht wordt dus *gepad* tot het de lengte van een geheel blok heeft. Dit zorgt voor een bijkomende moeilijkheid: na het decrypteren moet exact alle padding verwijderd worden, maar niets meer en niets minder. De manier waarop dit meestal opgelost wordt, is door het bericht te padden met een bytewaarde die gelijk is aan het aantal *padding bytes* dat toegevoegd werd. Dit impliceert dat er altijd padding aanwezig moet zijn, we hebben immers altijd minstens één extra padding byte nodig om het aantal padding bytes in te schrijven. Dus indien het bericht juist een veelvoud is van de bloklengte, wordt er een volledig extra padding blok toegevoegd (de grootst mogelijke expansie omwille van padding, dus het slechtst mogelijke geval).

Eerder zagen we dat stream ciphers een aantal onaantrekkelijke eigenschappen bezitten op beveiligingsgebied. Block ciphers zijn echter robuuster en blijken geen eigenschappen te vertonen die de vertrouwelijkheid schaden. Hierdoor is het perfect veilig om dezelfde sleutel voor verschillende berichten te gebruiken, zolang maar telkens een verse IV aangemaakt wordt. Omwille van CBC rollover is het ook belangrijk om niet meer dan $X*2^{\frac{X}{2}}$ bytes te encrypteren met eenzelfde sleutel (met lengte X).

Block ciphers blijken nog wel kwetsbaar voor integriteitsaanvallen, zij het in veel mindere mate dan stream ciphers.
Een aanvaller kan in het algemeen geen voorspelbare veranderingen teweegbrengen in de plaintext door de ciphertext te veranderen. Elke verandering zal zowel het veranderde blok, als het blok dat daarop volgt, beschadigen, omwille van de afhankelijkheid tussen twee opeenvolgende blokken bij decryptie.

Een uitzondering hierop is het eerste blok, waarvan de integriteit niet gegarandeerd is. Aangezien de IV meestal eenvoudigweg wordt meegestuurd met het versleutelde bericht, kan een aanvaller voorspelbare veranderingen induceren in het eerste blok door dit IV te veranderen.

Dus hoewel block ciphers duidelijk robuuster zijn wat betreft integriteitsaanvallen, is het toch belangrijk om ook bij deze methode een MAC te gebruiken om de integriteit te waarborgen.

Algoritmen

De belangrijkste block ciphers zijn de volgende:

- **DES**
  DES staat voor *Data Encryption Standard.* Het is een 64-bit block cipher met een 56-bit sleutel. De echte lengte van een DES sleutel is 64 bits lang, met de laagste-orde bit in elke byte gebruikt als een *parity check*, om transmissiefouten te detecteren. In praktijk is deze parity check echter overbodig, en wordt hij meestal genegeerd.
- **3DES**
  Aangezien DES heel goed bestand bleek te zijn tegen analytische aanvallen, maar een te korte sleutel bleek te hebben, lag het voor de hand hierop een nieuw, sterker algoritme te baseren met een dubbele sleutellengte, door de data eenvoudigweg tweemaal door DES te jagen. Dit proces noemt men *superencryptie*, en resulteerde in 2DES. Al snel werd echter duidelijk dat 2DES niet veel veiliger was dan DES (omwille van de zogenaamde *meet-in-the-middle attack*, niet te verwarren met de verderop besproken *man-in-the-middle attack*). Drie keer DES toepassen (3DES) bleek echter wel een belangrijke verbetering te bieden, met een effectieve sleutelsterkte van 112 bits (de sterkte die je van 2DES zou verwachten).
- **RC2**
  RC2 (van dezelfde ontwerper als de stream cipher RC4, Ron Rivest) is een 64-bit block cipher met een sleutel van variabele lengte. Zoals bij 3DES is zijn effectieve sleutelsterkte echter kleiner dan zijn sleutellengte; een 64-bit sleutel bijvoorbeeld blijkt een effectieve sleutelsterkte van 40 bit te hebben (met andere woorden, zo'n sleutel kan gebroken worden in 2^{40} operaties).
- **AES**
  AES, de *Advanced Encryption Standard* wordt beschouwd als de opvolger van DES. De oorspronkelijke naam luidt *Rijndael*, en dit algoritme werd ontworpen door twee Vlaamse onderzoekers, Joan Daemen and Vincent Rijmen. Deze block cipher heeft een variabele blok- en sleutellengte, die beide variëren over 128, 192 en 256 bit.

2.5.3 Het sleutelmanagementprobleem

Een belangrijk probleem van elk symmetrisch encryptie-algoritme is het feit dat beide partijen op de een of andere manier een gemeenschappelijke geheime sleutel moeten overeenkomen, alvorens van een veilige communicatie sprake kan zijn. En hoe krijgen we die sleutel bij de andere partij, zonder een veilig communicatiekanaal?

Dit probleem wordt de *catch-22 paradox*[12] van de cryptografie genoemd en is bekender als het **sleutelmanagementprobleem** (‘*Key Management Problem*’ of ‘*Key Distribution Problem*’). Asymmetrische encryptie probeert hier een oplossing voor te bieden, zoals besproken in de volgende sectie.

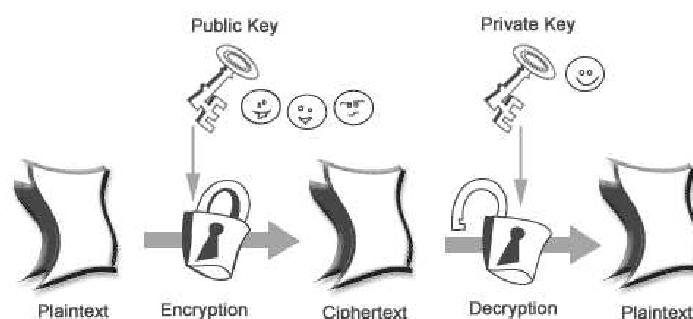
2.6 Asymmetrische encryptie

Het basisidee van *asymmetrische encryptie* is het gebruik van verschillende sleutels voor encryptie en decryptie.
Hiervoor genereert men twee verschillende sleutels, waartussen de volgende twee unieke verbanden bestaan:

- elke plaintext geëncrypteerd met de ene sleutel, kan gedecrypteerd worden met de andere sleutel, en vice versa.
- de ene sleutel uit de andere afleiden is computationeel onmogelijk.

We noemen de ene sleutel[13] de *publieke sleutel* (‘*public key*’) en deze wordt ter beschikking gesteld van het grote publiek. De andere sleutel is de bijhorende *private sleutel* (‘*private key*’) en blijft geheim.
Asymmetrische encryptie wordt ook wel **publieke sleutel-cryptografie** (‘*public key cryptography*’) genoemd.

In Figuur 2.2 wordt dit mechanisme geïllustreerd.



Figuur 2.2. Asymmetrische encryptie

Het is duidelijk dat via dit mechanisme om het even wie naar een willekeurig persoon een geëncrypteerd bericht kan sturen, zelfs zonder die persoon ooit ontmoet te hebben, en bovenal zonder de vereiste van een gedeelde geheime sleutel.

[12] Deze paradox duidt op een naar zichzelf refererende cirkelredenering of *lose-lose* situatie, analoog aan de militaire *catch-22* regel die in 1961 door Joseph Heller geïntroduceerd werd in zijn boek “Catch-22”.

[13] deze opdeling is niet willekeurig, maar hangt af van de implementatie

Asymmetrische encryptie biedt dus duidelijk een oplossing voor het eerder besproken sleutelmanagementprobleem. Toch is het geen complete oplossing, aangezien publieke sleutel-cryptografie relatief traag blijkt te zijn. In sectie 2.7 wordt hierop teruggekomen.

2.6.1 RSA

Het belangrijkste en meest bekende asymmetrische encryptie-algoritme is ongetwijfeld **RSA**. Ontworpen in 1977 en genoemd naar zijn ontwerpers Ron **R**ivest, Adi **S**hamir en Len **A**delman, is dit het algoritme waaraan de meeste mensen denken bij de term "publieke sleutel-cryptografie".

Een *RSA sleutelpaar* ('*key pair*') bestaat uit een publieke en private sleutel:

- De publieke sleutel is een paar *(e,n)*, met n de *modulus* en e de *publieke exponent*, waarbij de modulus het product is van twee erg grote priemgetallen p en q.

  Het is van het grootste belang dat p en q geheim blijven, aangezien de sterkte van RSA juist gebaseerd is op de moeilijkheid om een heel groot getal (hier de modulus) op te splitsen in zijn priemgetallen ("*factorizatie*")[14]. Als we spreken over de sleutellengte van een RSA sleutel, hebben we het over de lengte van zijn modulus.

  De publieke exponent wordt meestal gekozen uit een beperkte verzameling kleine priemgetallen[15] (3, 17 of 65537). Een kleine exponent maakt publieke sleutel operaties aanzienlijk sneller.

- De bijhorende private sleutel wordt dan gevormd door het koppel *(d,n)*, met n weer dezelfde modulus. d wordt berekend aan de hand van de priemgetallen p,q en de publieke exponent e (en zal zo van ruwweg dezelfde grootte als n zijn):

  $$d = e^{-1} \bmod ((p-1)(q-1))$$

Encryptie met de publieke sleutel gebeurt dan als volgt:

$$C = B^{e} \bmod n$$

Decryptie vereist de overeenkomstige private sleutel:

$$B = C^{d} \bmod n$$

Volledigheidshalve moeten we hier opmerken dat RSA dus uitgaat van een getal als input, waardoor we ons bericht op de een of andere manier moeten omzetten naar een getal. Procedures als deze staan ook wel bekend als *padding schema's* ('*padding schemes*'). De standaard procedure hiervoor is gespecificeerd in de **Public-Key Cryptography Standard *(PKCS)* #1**[64].

[14] Om helemaal correct te zijn moeten we hier opmerken dat dit zogenaamde 'RSA-probleem' en 'factorizatieprobleem' twee problemen zijn waarvan nog niet bewezen is dat ze computationeel equivalent zijn. Wel is bewezen dat het RSA-probleem reduceert tot het factorizatieprobleem, het RSA-probleem is dus in elk geval niet moeilijker dan het factorizatieprobleem, en naar algemeen wordt aangenomen ook niet makkelijker [48, p. 99].

[15] e moet relatief priem zijn t.o.v. e en *(p-1)(q-1)*, d.w.z. ggd(e,((p-1)(q-1))) = 1

2.6.2 Diffie-Hellman

Een tweede asymmetrisch algoritme is **Diffie-Hellman (DH)**. Dit is het oudste publieke sleutel algoritme en werd gepubliceerd in 1976 [18].

We vermelden dit algoritme uitsluitend voor de volledigheid en beperken ons in de rest van deze studie tot SSL met RSA als asymmetrisch encryptie-algoritme.

2.7 Symmetrisch versus Asymmetrisch

Nu we een overzicht gegeven hebben van zowel symmetrische als asymmetrische encryptie, is het nuttig beide methoden eens te vergelijken.

Performantie

Symmetrische encryptie is heel snel, maar zit met een *sleutelmanagementprobleem*. Asymmetrische encryptie daarentegen biedt hier een oplossing voor, maar is relatief *traag*.

Deze twee feiten hebben ertoe geleid dat men beide methoden op een complementaire manier ging gebruiken, om de voordelen van beide zoveel mogelijk te benutten, en tegelijk de nadelen te beperken:

> Eerst wordt een geheime sleutel (die relatief klein is) uitgewisseld met behulp van een trage asymmetrische encryptie, om daarna de eigenlijke communicatie (die heel veel data kan omvatten) te beveiligen met een snelle symmetrische encryptie, die gebruikt maakt van de via asymmetrische encryptie overeengekomen geheime sleutel. Deze geheime sleutel wordt de *datasleutel* (’*data key*’, ook wel sessiesleutel) genoemd, en de asymmetrische sleutel die gebruikt wordt om de datasleutel te beschermen noemt men een sleutelencryptie-sleutel (’*key-encryption key*’).

Met behulp van deze *hybride* methode, lossen we het sleutelmanagementprobleem op zonder de performantie al te zeer naar beneden te halen.

Sleutellengte

Sleutellengtes van symmetrische algoritmen zijn absoluut niet te vergelijken met die van asymmetrische. Sleutellengtes voor asymmetrische algoritmen liggen in de regel tussen 512 en 4096 bits, terwijl deze voor het symmetrische geval veel kleiner zijn: tussen de 40 en 256 bit.

De sterkte van beide soorten encryptie zijn bij de genoemde sleutellengtes vergelijkbaar, maar relatief gezien zijn de asymmetrische sleutels dus veel zwakker. De reden hiervoor wordt gevormd door de vele voorwaarden waaraan voldaan moet zijn om een geldig asymmetrisch sleutelpaar te vormen, niet om het even welke random waarde voldoet. Daardoor is de *sleutelruimte* (‘*key space*’) die doorzocht moet worden om een asymmetrische sleutel

van een bepaalde lengte te breken veel kleiner dan die bij een symmetrische sleutel van vergelijkbare lengte.

In Tabel 2.2 wordt weergegeven bij welke sleutellengtes de sterkte van symmetrische en asymmetrische sleutels vergelijkbaar is [53].

Symmetrische sleutellengte (bits)	Asymmetrische sleutellengte (bits)
70	947
80	1228
90	1553
100	1926
150	4575
200	8719
250	14596

Tabel 2.2. Een vergelijking van de sterkte van asymmetrische en symmetrische sleutels op basis van sleutellengte

Merk op dat het weinig zin heeft om een 250-bit symmetrische sleutel uit te wisselen met behulp van een 1024-bit asymmetrische sleutel. Het hybride geheel zal dan immers niet sterker zijn dan een 70- of 80-bit symmetrische encryptie. Indien men gebruik wilt maken van deze hybride methode om het sleutelmanagementprobleem op te lossen, moet men er altijd voor waken dat de sleutelsterktes van het gebruikte asymmetrische en symmetrische algoritme vergelijkbaar zijn, om nodeloos performantieverlies of een onterecht gevoel van hoge veiligheid te voorkomen.

2.8 Message Digest

Werking

Een **message digest** is een functie die een string van variabele lengte (het bericht) omzet in een string van vaste lengte (de message digest) die *karakteristiek* is voor de inputstring. Een message digest wordt ook wel eens een *one-way hash function* genoemd.

Message digests hebben twee belangrijke eigenschappen:

1. **Onomkeerbaarheid**
   Het moet onmogelijk zijn om een bericht te recupereren aan de hand van zijn message digest.

   Intuïtief begrijpen we dat we te weinig informatie hebben om de originele boodschap vanuit de message digest te berekenen: het aantal mogelijke berichten (van variabele lengte) is veel groter dan het aantal mogelijke message digests (van vaste lengte).

   De eis van onomkeerbaarheid is echter sterker dan dit: men spreekt pas van een *veilige* message digest, als het moeilijk is om *om het even welk* bericht te vinden

dat gekarakteriseerd wordt door die bepaalde message digest. Met moeilijk bedoelen we dat de grootte van de berichtenruimte die moet afgezocht worden proportioneel moet zijn ten opzichte van de grootte van de digest. Dat wil zeggen, als de digest 128 bit lang is, is de berichtenruimte 2^{128} berichten groot.

2. **Botsingsweerstand ("collision-resistance")**
   Het moet moeilijk zijn om twee berichten B_1 en B_2 te produceren met eenzelfde message digest.

   Uit onderzoek blijkt dat deze weerstand slechts proportioneel is ten opzichte van de helft van de digestgrootte, omwille van de zogenaamde *Birthday Paradox*[28]. Een 128 bit digest heeft dus slechts een weerstand van 64 bit, dat wil zeggen, er zijn gemiddeld 2^{64} operaties nodig om een botsing te genereren.

De belangrijkste toepassing van message digests is het gebruik in *digitale handtekeningen* en *Message Authentication Codes* (MAC), die beide verderop besproken worden.

Algoritmen

De twee belangrijkste en meest gebruikte message digest algoritmen zijn **Message Digest 5 (MD5)** [61] en **Secure Hash Algorithm 1 (SHA-1)** [2]. Beiden zijn afgeleid van een vroeger algoritme, **MD4**. Daarnaast bestaat er een nog ouder algoritme, **MD2**.

MD5, MD4 en MD2 hebben een 128 bit output, een SHA-1 digest is 160 bits lang.

2.9 Digitale Handtekening

Eerder zagen we al dat we vertrouwelijkheid konden garanderen door encryptie te gebruiken. We hebben echter nog niets voorgesteld om aan de integriteitseis te voldoen, of om de afzender te authenticeren. Zowel de **digitale handtekening**, die hier besproken wordt, als een *message authentication code (MAC)*, een techniek besproken in de volgende paragraaf, kunnen hier een oplossing bieden[16].

Werking

Een digitale handtekening maakt altijd gebruik van publieke sleutel-cryptografie, maar kan op twee verschillende manieren gegenereerd worden:

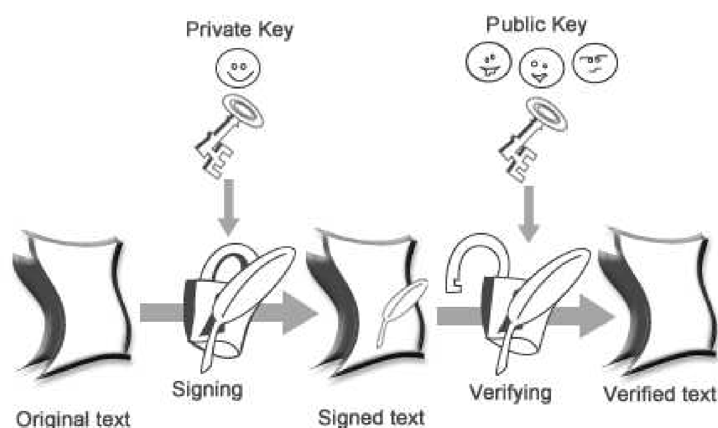
- Het asymmetrische encryptie-proces wordt omgekeerd[17] gebruikt (Figuur 2.3). Met andere woorden, de *private* sleutel wordt gebruikt voor *encryptie*, en de *publieke* sleutel om te *decrypteren*.

  Zo'n versleuteld bericht kan iedereen dan ontcijferen en lezen (de publieke sleutel is immers vrij beschikbaar), maar slechts de bezitter van de private sleutel kan zo'n

[16]Deze methodes garanderen bovendien de juridisch interessante eigenschap die bekend staat als *onweerlegbaarheid* of *non-repudiatie*.

[17]Deze omkering bestaat wiskundig gezien uit het vervangen van de publieke exponent (e) door de private exponent (d) in de encryptie en decryptie formules

ciphertext genereren. Een ontvanger kan zo dus nagaan of de verwachte afzender wel degelijk de echte afzender is (authenticatie).



Figuur 2.3. Een eenvoudige digitale handtekening

- De afzender genereert eerst een message digest van het bericht, om dit dan te encrypteren met zijn private sleutel (Figuur 2.4). De ontvanger ontcijfert dan dit message digest, en vergelijkt dit met het message digest dat hij zelf genereert over het ontvangen bericht. Indien beide digests gelijk zijn, is de afzender geauthenticeerd, en tegelijkertijd de integriteit van het bericht verzekerd.

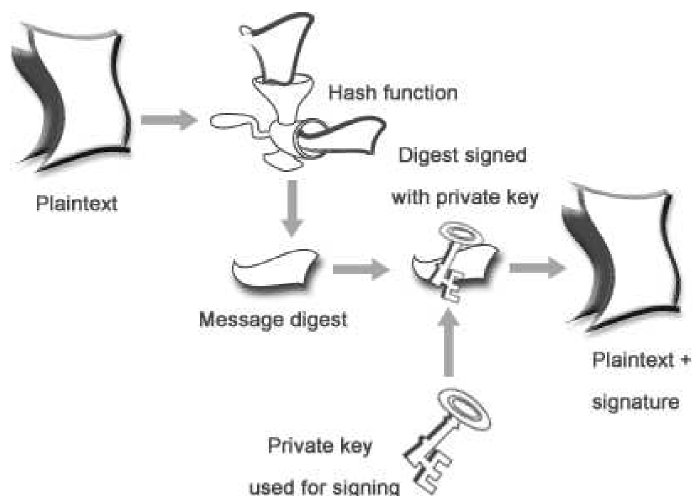
Aangezien asymmetrische encryptie en decryptie trage operaties zijn, waarbij de snelheid afhankelijk is van de lengte van de data die verwerkt moet worden, wordt meestal de tweede methode gebruikt. Deze is immers aanzienlijk sneller, omdat een message digest altijd relatief kort is (128 of 160 bit), onafhankelijk van de lengte van het bericht zelf.

Wanneer in deze thesis het begrip *digitale handtekening* gebruikt wordt, doelen we op de laatste methode, tenzij anders aangegeven.

Algoritmen

Aangezien digitale handtekeningen op publieke sleutel-cryptografie steunen, is **RSA** ook hier de meest gebruikte methode.

Daarnaast bestaat er een ander algoritme, specifiek ontwikkeld voor digitale handtekeningen, **Digital Signature Standard (DSS)**. Aangezien DSS binnen het SSL protocol altijd gebruikt wordt in combinatie met Diffie-Hellman, een asymmetrische encryptiemethode die niet behandeld wordt, valt een gedetailleerd overzicht buiten het bereik van deze studie. Het is voldoende te weten dat DSS enkel gebruikt kan worden voor digitale handtekeningen en op een iets andere manier werkt dan de methode die besproken werd.



Figuur 2.4. Digitale handtekening m.b.v. message digest

2.10 Message Authentication Code

Deze techniek biedt een gelijkaardige oplossing aan als de digitale handtekening, en zal ook zowel integriteit als authenticatie garanderen[18].

Werking

De werking lijkt erg op die van digitale handtekeningen, met dit verschil dat een digitale handtekening gebruik maakt van een private sleutel, waar een MAC een geheime sleutel gebruikt. Bovendien wordt bij een MAC niet eerst de digest gegenereerd, waarna deze geëncrypteerd wordt, maar wordt de sleutel mee als input aan het digestalgoritme gegeven, wat een gelijkaardig effect heeft.

Voor de rest is het proces identiek aan dat gebruikt bij een digitale handtekening. Men kan dus stellen dat een digitale handtekening tegenover een MAC staat als asymmetrische encryptie tegenover symmetrische encryptie. Een digitale handtekening behoort tot het domein van de *publieke sleutel-cryptografie*, waar een MAC tot de *geheime sleutel-cryptografie* behoort. Omwille van het vermijden van een asymmetrische encryptiestap is een MAC ook performanter dan een digitale handtekening.

Algoritmen

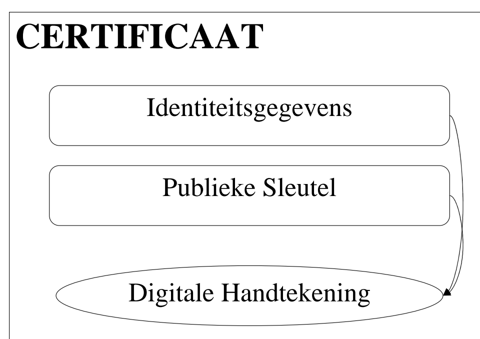
De meest gebruikte constructie voor het genereren van MAC's is **HMAC**, die beschrijft hoe men een MAC, met bewijsbare veiligheidseigenschappen zoals integriteit en authenticatie, kan genereren met behulp van om het even welk digest algoritme dat voldoet aan een aantal voorwaarden, beschreven in [43].

[18]indien men enkel geïnteresseerd is in de integriteitseigenschap, noemt men deze techniek ook wel een MIC (Message Integrity Check)

2.11 Certificaten

Eerder werd gesteld dat publieke sleutel-cryptografie het antwoord was op het sleutelmanagementprobleem. Er is toen echter voorbijgegaan aan een nieuw probleem, namelijk het verspreiden van publieke sleutels.
Hoe weet men welke publieke sleutel van wie is? Hoe kan dit geverifieerd worden?

Certificaten pogen hier een antwoord op te geven. Een certificaat bestaat uit een publieke sleutel, met daarnaast allerlei informatie die de eigenaar identificeert (naam, adres, ...). Over het geheel van publieke sleutel en identificatie-informatie wordt dan een *digitale handtekening* berekend, die wordt toegevoegd aan het certificaat (Figuur 2.5).



Figuur 2.5. Een certificaat

Nu rest de vraag met welke private sleutel die digitale handtekening berekend zal worden. Alle eigenaars hun certificaten zelf laten handtekenen is geen oplossing, want dan kan eender wie zichzelf (en zijn publieke sleutel) voordoen als om het even wie.

Dit probleem wordt opgelost door de hulp van een zogenaamde *trusted third party* in te roepen. Deze derde partij, die in de context van certificaten een **Certificate Authority (CA)** genoemd wordt, is een organisatie die alle gebruikers van het systeem vertrouwen. In praktijk wordt deze rol op zich genomen door commerciële bedrijven zoals Verisign.

Het zijn deze CA's die met behulp van hun private sleutel een digitale handtekening over het certificaat berekenen, het certificaat op deze manier bekrachtigen, en de certificaten uitgeven. Ook de identiteitsgegevens van de uitgevende CA worden aan het certificaat toegevoegd, samen met nog andere informatie zoals geldigheidsduur.

Eender wie kan dan vervolgens de geldigheid van een certificaat testen door de digitale handtekening (van een bepaalde CA) te controleren met behulp van de publieke sleutel van deze CA.

Ook de publieke sleutels van de CA's moeten op een veilige manier verspreid en geverifieerd kunnen worden, dit wordt opgelost door wereldwijd slechts een handvol *root CA*'s aan te stellen, waarvan de publieke sleutels in de eindapplicaties zelf worden ingebakken. Deze root CA's kunnen dan speciale certificaten uitreiken om andere partijen als CA te autoriseren, zogenaamde *intermediate CA*'s. Op deze manier wordt een ketting van certificaten opgebouwd, van root CA over één of meerdere intermediate CA's, naar het certificaat van

een normale gebruiker, waarbij het certificaat van elke intermediate CA kan geverifieerd worden met de publieke sleutel van de bovenliggende CA. Op deze manier kan om het even wie certificaten van om het even wie verifiëren, door de achterliggende ketting na te gaan.

Deze complete infrastructuur, die instaat voor de verspreiding van publieke sleutels, wordt een **Public Key Infrastructure (PKI)** genoemd.

Zo'n PKI lost echter niet alleen dit probleem van sleutelverspreiding op. Het kan ook onrechtstreeks gebruikt worden om de eigenaar van een certificaat te authenticeren, omdat alle data die versleuteld wordt met de publieke sleutel uit dat certificaat, enkel kan gedecrypteerd worden door de bezitter van de bijhorende private sleutel. Een gebruiker die data versleutelt met behulp van de publieke sleutel uit een certificaat, weet dus zeker (indien hij de moeite doet om de digitale handtekening van de bevoegde CA te controleren) dat deze data enkel gelezen kan worden door de persoon waarvan de identiteitsgegevens in het certificaat zijn opgenomen. Hij communiceert in dit geval dus met wie hij denkt te communiceren, waarmee aan de authenticatie-eigenschap voldaan is.

We merken wel op dat zo'n PKI niet perfect werkt, maar met een belangrijk probleem kampt, namelijk *certificaat revocatie*, waarbij de ongeldigheid van een certificaat (bijvoorbeeld omwille van compromittering), bekend gemaakt moet worden in heel de gebruikersgemeenschap. Meer informatie over dit probleem, en PKI's in het algemeen, kan gevonden worden in [94].

We besluiten met kort samen te vatten wanneer we een certificaat van een bepaalde server X *geldig* noemen:

- identiteitsgegevens en publieke sleutel in het certificaat komen overeen met die van server X
- vervaldatum van het certificaat is nog niet overschreden
- certificaat is gecertificeerd (gehandtekend) door een bevoegde CA (bevoegd in de zin van: zelf gecertificeerd door een hogerliggende bevoegde CA, en zo verder tot we bij een root CA komen)
- certificaat is niet gerevoceerd

Algoritmen

De belangrijkste standaard voor certificaten is **X.509** [34], aangepast voor het internet in RFC 2459 [32].

2.12 Veilige Random Getallen

Vele cryptografische technieken steunen in grote mate op de mogelijkheid om sterke (veilige) *random getallen* te genereren. Random getallen zijn getallen in een volstrekt willekeurige volgorde; getallen waartussen geen enkel verband bestaat, waarin geen enkel patroon is terug te vinden.

We willen dat zo een stroom van getallen de volgende eigenschappen bezit, om veilig random te zijn:

- **Onpartijdigheid ('unbiased')**
  Op bitniveau moet het aantal 0- en 1-bits in de stroom ongeveer gelijk zijn.
- **Decorrelatie**
  Er mag geen enkel verband bestaan tussen de getallen in de stroom. Uit een deel van de stroom mag geen enkel ander stuk van de stroom afgeleid kunnen worden.

De normale random generator van een programmeertaal voldoet niet aan deze eigenschappen, en volstaat dus niet. Om sterke random getallen te produceren, bestaan er in het algemeen twee manieren, die verderop uitgebreider besproken worden: **Pseudo-Random Number Generators (PRNG)** en **Hardware Random Number Generators**[19].

2.12.1 Pseudo-Random Number Generators (PRNG)

Het idee achter een PRNG is eenvoudig: genereer met behulp van een functie een stroom onvoorspelbare bytes. Een stroom die in theorie niet te onderscheiden is van echte random data, zoals bijvoorbeeld gegenereerd door het opgooien van een munt.

In praktijk is dit echter nooit het geval, aangezien elke PRNG na verloop van tijd terug dezelfde bytestroom zal herhalen. De output van een PRNG is immers een functie van de interne staat van de PRNG. Zo'n interne staat kan natuurlijk nooit groter zijn dan het geheugen van de computer waarop de PRNG loopt, dus ooit moet deze stroom zich herhalen.

Maar aangezien 256 bits geheugen voor de interne staat ruimschoots genoeg blijkt te zijn om een voldoende groot aantal random bytes te genereren zonder enige herhaling, is deze beperking niet van belang.

PRNG's zijn typisch gebaseerd op digest of encryptie-algoritmen. Voor een goed voorbeeld verwijzen we naar [36]. Alle PRNG's hebben echter één ding gemeen: ze maken gebruik van een *seed*. Een seed bestaat uit een zeker hoeveelheid geheime random data, die gebruikt wordt om de interne staat van de PRNG te initialiseren. De seed is geheim, omdat deze zo de interne staat, en dus alle gegenereerde data, volledig voorspelt.

Het grootste probleem bij het gebruik van een PRNG bestaat er uit om deze op een goede manier te *seeden*. Gewoonlijk wordt dit opgelost door de PRNG te seeden met een grote hoeveelheid random data van lage kwaliteit. Hieruit *destilleert* de PRNG dan sterke random data.

Zulke seeding gebeurt dan met behulp van bijvoorbeeld netwerkactiviteit of beeldschermdata. Om echt sterke random data te verkrijgen, bijvoorbeeld voor het genereren van een private sleutel, wordt dikwijls expliciet om user input (via muis of toetsenbord[20]) gevraagd.

[19] Deze laatste worden ook wel eens TRNG's genoemd, "Truly Random Number Generator"
[20] Meer bepaald de tijd tussen muisbewegingen en toetsaanslagen wordt gebruikt.

2.12.2 Hardware Random Number Generators (HRNG)

Een tweede manier baseert zich op hardware. Bepaalde fysieke apparaten blijken zich op een random manier te gedragen op kwantummechanisch niveau, wat gebruikt kan worden om random data te produceren. Een voorbeeld is het fenomeen van radioactief verval, waarbij een constante stroom van signalen geproduceerd wordt op willekeurige (random) momenten[21].

Een random stroom die via hardware gegenereerd wordt, blijkt meestal niet te voldoen aan de onpartijdigheidseigenschap. Zo'n stroom kan echter wel *unbiased* gemaakt worden op een eenvoudige manier, zoals John von Neumann aantoonde [88].

In praktijk wordt meestal een combinatie tussen een HRNG en PRNG gebruikt, door de PRNG te seeden met random data van de HRNG. Dit lost het probleem van partijdigheid op en heeft het bijkomende voordeel dat men niet op hardware apparaten moet wachten voor het genereren van random data.

2.13 Cryptografische technieken en veilige communicatie

In paragraaf 2.3 werd uitgelegd dat communicatiebeveiliging geen monolithisch begrip is, maar een aantal eigenschappen omvat. Er werden toen drie eigenschappen voorgesteld: *vertrouwelijkheid*, *integriteit* en *authenticiteit*.

Hierna volgt een samenvatting van welke cryptografische technieken welke eigenschappen kunnen garanderen:

1. **Vertrouwelijkheid**
   Vertrouwelijkheid kan gegarandeerd worden door gebruik van encryptie/decryptie, zij het *symmetrische* of *asymmetrische*.

2. **Integriteit**
   Om de integriteit te waarborgen, worden *message digests* gebruikt in combinatie met een sleutel, onder de vorm van een *digitale handtekening* of een *message authentication code (MAC)*.

3. **Authenticiteit**
   Ook deze eigenschap kan verkregen worden door een *MAC* te gebruiken. Meestal wordt hier echter aan voldaan met een *digitale handtekening*, al dan niet in combinatie met een *certificaat*.

Zulke cryptografische technieken, die kunnen gebruikt worden als bouwstenen in meer geavanceerde systemen (om bijvoorbeeld communicatie te beveiligen), noemt men ook wel eens *cryptografische primitieven*.

[21] Hoewel het gemiddelde van deze signalen wel voorspelbaar is.

Hoofdstuk 3

Het SSL protocol

"Gentlemen do not read other's mail"

– Henry L. Stimson
Staatssecretaris van de Verenigde Staten in 1929

3.1 Inleiding

Nu we de werking van een aantal cryptografische primitieven begrijpen, zullen deze gebruikt worden om een ingewikkelder cryptografisch systeem te ontwikkelen, namelijk het **Secure Socket Layer (SSL) protocol**.

Het doel van dit *cryptosysteem* bestaat uit het beveiligen van de communicatie tussen twee partijen over een onveilig kanaal. Met veiligheid bedoelen we hier de drie eigenschappen zoals besproken in Hoofdstuk 2: vertrouwelijkheid, integriteit en authenticatie.

Dit hoofdstuk wordt gestart met een kort overzicht van de ontstaansgeschiedenis van het SSL protocol. Vervolgens wordt een uitvoerige bespreking gegeven van de werking van dit protocol, evenwel zonder in te gaan op al te technische details. Als laatste vatten we kort samen door welke elementen de veiligheid van een SSL connectie gewaarborgd wordt.

3.2 Ontstaan van SSL

3.2.1 Doel

Het primaire doel van SSL is het beveiligen van de communicatie tussen twee partijen, meer bepaald communicatie over het internet.

Daarenboven werd SSL ontworpen met het oog op het paradigmatische gebruikscenario van een online-betaling met kredietkaart, een scenario dat de volgende eisen met zich meebrengt:

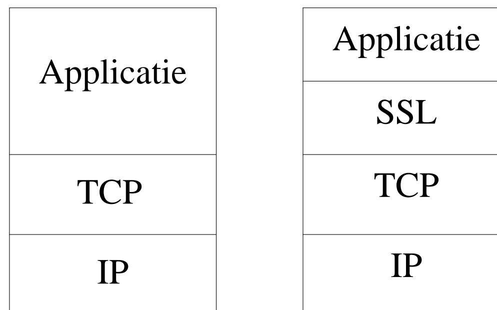
- **Vertrouwelijkheid**
  De gegevens (in het bijzonder het kredietkaartnummer) moeten geheim blijven.

- **Server authenticatie**
  De koper wil zeker weten dat zijn kredietkaartgegevens bij de juiste winkel (*server*) terechtkomen. Andersom is het minder belangrijk voor de server om te weten wie de koper (*client*) is. Om deze reden is *client authenticatie* optioneel in SSL.

- **Spontaniteit**
  Kopers (clients) en winkels (servers) hoeven elkaar niet op voorhand te kennen. Toch moet een transactie met onbekenden vlot verlopen, dit noemt men de eis van spontaniteit.

Daarnaast wilden de ontwerpers van SSL een universele oplossing creëren, die gebruikt kan worden om ook andere internet protocols[1] dan het meestgebruikte **Hypertext Transfer Protocol (HTTP)** te beveiligen. Omdat de meeste internet protocols gebaseerd zijn op **TCP** binnen de transportlaag, werd beslist om SSL als een "veilige TCP laag" te implementeren. Dat wil zeggen dat SSL connecties en hun eindpunten (*sockets*) zich op dezelfde manier gedragen als TCP connecties en TCP sockets, althans vanuit het oogpunt van een programmeur.

Dit laatste noemt men ook wel eens de eis van **transparantie**. SSL vormt als het ware een **transparante beveiligingslaag**, die tussen de TCP-laag (transportlaag) en de applicatielaag wordt geschoven (Figuur 3.1).



Figuur 3.1. Protocol stack, zonder en met SSL (fysische en datalink laag zijn niet weergegeven)

Verderop komen we terug op het doel van veilige communicatie over het internet en deze bijkomende eisen, maar eerst geven we een stukje geschiedenis van SSL.

[1]Met de term internet protocol doelen we hier op applicatieprotocols van het internet die in de applicatielaag draaien, niet op het Internet Protocol (IP), het internet protocol voor de netwerklaag

3.2.2 Geschiedenis

SSLv2

Een eerste versie van SSL, SSLv1, werd ontworpen in 1994 door Netscape. Deze versie werd nooit gebruikt maar werd nog datzelfde jaar opgevolgd door SSLv2 [31], die wel een wereldwijde verspreiding kende, niet in het minst doordat deze versie werd geïmplementeerd in de internet browser Netscape Navigator.

SSLv3

SSLv2 bleek echter een aantal serieuze beveiligingslekken te bevatten en Netscape besliste om van nul te herbeginnen. Er werd onder andere een compleet nieuwe specificatietaal ontworpen. Bovendien werden een aantal *features* toegevoegd die nog niet in SSLv2 aanwezig waren, met als belangrijkste de ondersteuning voor extra cryptografische algoritmen. Het resultaat was SSLv3.

TLS

In Mei 1996 richtte de **Internet Engineering Task Force (IETF)** de **Transport Layer Security (TLS)** werkgroep op, in een poging het SSL protocol te standaardiseren. Sinds het ontstaan van SSL hadden immers ook andere bedrijven dan Netscape gepoogd om een soort SSL protocol te implementeren. Van deze bedrijven stond Microsoft op kop, dat zijn **Private Communications Technology (PCT)** publiceerde in oktober 1995 [3].

In feite ging deze standaardisatie om een poging om de Microsoft aanpak (PCT) te harmoniseren met het protocol van Netscape (SSLv3).

Uiteindelijk werd TLS grotendeels gebaseerd op SSLv3, intern draagt het zelfs het versie nummer SSLv3.1. De verschillen tussen SSLv3 en TLS zijn van weinig belang in deze bespreking, en bestaan hoofdzakelijk uit nog meer mogelijkheden op gebied van cryptografische algoritmen, en een andere aanpak voor bepaalde cryptografische berekeningen.

Aangezien TLS, omwille van een aantal problemen, pas gepubliceerd werd in 1999, in RFC 2246 [16], is het tot nu toe nog niet echt doorgebroken[2], en blijft SSLv3 de meest gebruikte versie. In deze thesis zullen wij ons dan ook in de eerste plaats richten op SSLv3, hoewel de verschillen met TLS besproken zullen worden waar relevant.

Met de term SSL bedoelen we dus altijd SSLv3, tenzij anders aangegeven.

[2] Een uitgebreide studie die dit aantoont zal in Sectie 8.2 besproken worden

3.3 Het SSL Protocol

In Sectie 3.2.1 werd het doel van SSL voorgesteld, samen met een aantal bijkomende eisen. In deze sectie zullen we bespreken hoe het SSL Protocol hieraan zal voldoen.

Doel

Het doel van SSL bestaat uit het opzetten van een veilige communicatie over een onveilig kanaal. In Sectie 2.3 zagen we aan welke eigenschappen een communicatie moet voldoen, om als veilig beschouwd te worden. Vervolgens werd in Sectie 2.13 uitgelegd welke cryptografische algoritmen gebruikt kunnen worden om aan deze eigenschappen te voldoen.

Die kennis passen we nu toe om te bepalen welke cryptografische algoritmen in het SSL Protocol aangewend zullen worden:

1. **Vertrouwelijkheid**
   De data die beide partijen met elkaar uitwisselen moet versleuteld worden. Aangezien deze uitwisseling liefst zo vlot mogelijk gebeurt, kiezen we hier voor *symmetrische encryptie*. Uit die keuze volgt echter een sleutelmanagementprobleem, maar zoals we zagen kan dit opgelost worden door *asymmetrische encryptie* (met een certificaatmechanisme) te gebruiken om de sleutel uit te wisselen. We zullen dus een *hybride* encryptie methode gebruiken.

2. **Integriteit**
   De data mag onderweg niet ongemerkt gewijzigd worden. Hier wordt de efficiëntere *MAC* boven een digitale handtekening verkozen.

3. **(Server) Authenticatie**
   Minstens één van beide partijen (de client) wil zeker zijn dat zij communiceert met diegene waar zij *denkt* mee te communiceren. Dit wordt opgelost met het certificaat van de asymmetrische encryptie uit puntje 1.

Samengevat zal het SSL Protocol steunen op asymmetrische encryptie om een geheime sleutel tussen beide partijen uit te wisselen, om daarna de eigenlijke data te beveiligen met symmetrische encryptie op basis van die eerder uitgewisselde sleutel. De integriteit van de data zal gewaarborgd worden door een MAC (ook op basis van die uitgewisselde sleutel) en de authenticatie zal verzorgd worden door het certificaatmechanisme (of PKI) waarmee de publieke sleutel voor de asymmetrische encryptie verspreid wordt.

Eisen

Merk op dat door aan de voorwaarden voor een veilige connectie te voldoen, ook de bijkomende eisen tengevolge van het eerder geformuleerde gebruikersparadigma (sectie 3.2.1) vervuld zijn:

- **Vertrouwelijkheid**
  We voldoen expliciet aan deze eis door de vertrouwelijkheidseigenschap te garanderen.
- **Server Authenticatie**
  Ook aan deze eis wordt expliciet tegemoet gekomen door de authenticiteitseigenschap te garanderen.
- **Spontaniteit**
  Hieraan wordt impliciet voldaan door publieke sleutel-cryptografie te gebruiken. Deze eis kan eigenlijk gezien worden als een variant van het sleutelmanagementprobleem.
- **Transparantie**
  Het volledige protocol zal als een transparante laag op het TCP protocol functioneren.

We besluiten met op te merken dat er een duidelijke scheiding bestaat tussen enerzijds het *opzetten* van de beveiligde connectie met *authenticatie en sleuteluitwisseling* en anderzijds het *gebruiken* van deze connectie waarbij de *vertrouwelijkheid en integriteit van de datatransfer beschermd wordt.* Het SSL protocol is dan ook opgesplitst in twee grote delen, die ruwweg overeenkomen met deze twee connectiefases: het **SSL Handshake protocol** (authenticatie en sleuteluitwisseling) en het **SSL Record protocol** (beschermen datatransfer).

Deze twee subprotocols worden nu apart besproken.

3.4 SSL Handshake Protocol

3.4.1 Inleiding

In Sectie 3.2.2 zagen we dat de ontwikkeling van het SSL protocol een evolutie was, een opeenvolging van verbeteringen om eerdere ontwerpfouten op te lossen. Meer bepaald de handshake fase heeft doorheen deze ontwikkeling grote veranderingen ondergaan. We zullen de werking van deze fase dan ook verklaren door een gelijkaardige evolutie te doorlopen, om zo een aantal finesses van de handshake te verduidelijken.

We maken hiervoor gebruik van een *imaginaire beveiligingsspecialist met de naam Joe*[3], die in de volgende secties zijn initiële handshake stap voor stap zal verbeteren.

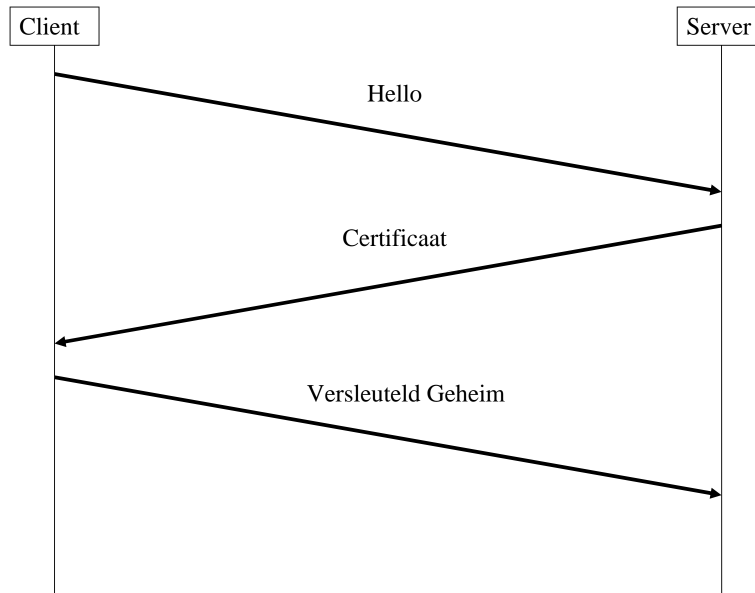
3.4.2 Een eerste handshake

Zoals we eerder zagen heeft een SSL handshake tot taak een veilige connectie over een bestaand, maar onveilig, kanaal op te zetten. Dit bleek neer te komen op het uitwisselen van een geheime sleutel in vertrouwen (vertrouwelijkheid), en met de juiste partij

[3] Volledigheidshalve moeten we hier opmerken dat een *team* van beveiligingsspecialisten dichter aanleunt bij de werkelijkheid, maar om de fantasie niet al te zeer op de proef te stellen, werd besloten het aantal imaginaire personen te beperken tot één.

(authenticatie).

Een eerste, naïeve poging van Joe om zo'n *handshake* te ontwerpen ziet er als volgt uit (Figuur 3.2):



Figuur 3.2. Een eerste handshake

- **Hello**
  Een systeem stuurt een bericht naar een ander systeem om te laten weten dat hij een veilige connectie wilt leggen. Het systeem dat het initiatief neemt, noemt men de *client*, het andere systeem noemt men de *server*.

- **Certificaat**
  De server beantwoordt deze begroeting met een gelijkaardig bericht, en stuurt vervolgens zijn certificaat terug.

- **Versleuteld Geheim**
  De client controleert de geldigheid van het certificaat (authenticatie). Als alles klopt, genereert hij een geheim (met een veilige random generator) en stuurt hij dit geëncrypteerd (met behulp van de publieke sleutel uit het certificaat) naar de server.

De sleuteluitwisseling is met behulp van deze eenvoudige handshake dus geslaagd, en de server is geauthenticeerd. Deze eenzijdige authenticatie noemt men *one-way authentication*[4], en volstaat in vele gevallen (zie ook Sectie 3.2.1).

SSL biedt standaard enkel server authenticatie aan. Optioneel kan ook de client geauthenticeerd worden, maar dit mechanisme is enkel nuttig in gespecialiseerde toepassingen

[4]Tegenover *mutual authentication* of wederzijdse authenticatie.

waar ook de identiteit van de client geverifieerd moet worden en zal in deze thesis niet besproken worden.

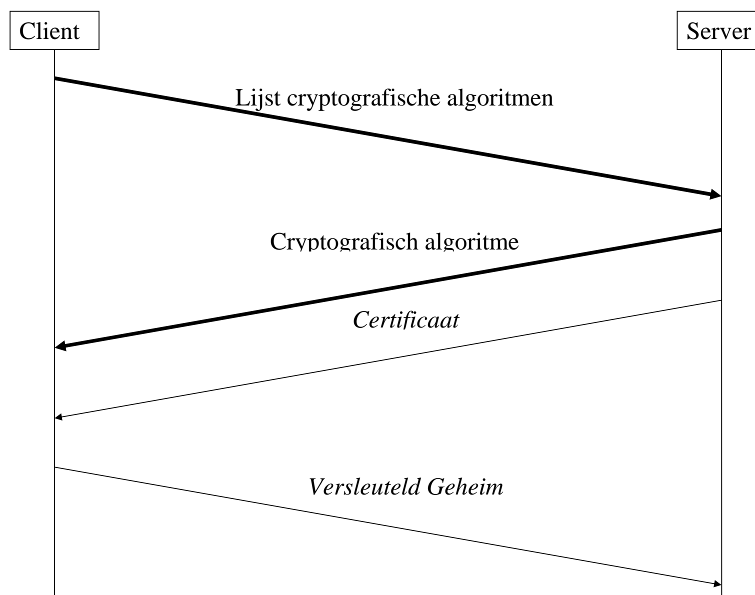
3.4.3 Een dynamische handshake

Dat eerste handshake ontwerp ziet er goed uit, maar gaat toch nog aan een aantal zaken voorbij. Zo wordt bijvoorbeeld nog nergens aangegeven welk asymmetrisch encryptie-algoritme gebruikt wordt.

In een tweede ontwerp lost Joe dit op, door in de begroeting van de client een aantal mogelijkheden voor cryptografische algoritmen op te nemen. Vervolgens kiest de server daaruit een mogelijkheid, en maakt zijn keuze kenbaar in zijn antwoord op de begroeting.

Dit proces wordt ook wel eens **negotiatie** genoemd. Client en server onderhandelen als het ware over welke algoritmen ze zullen gebruiken om de connectie te beveiligen. Joe heeft nu dus een *dynamische* handshake ontworpen.

Dit alles leidt tot het schema in Figuur 3.3.



Figuur 3.3. Een dynamische handshake

3.4.4 Integriteitscontrole

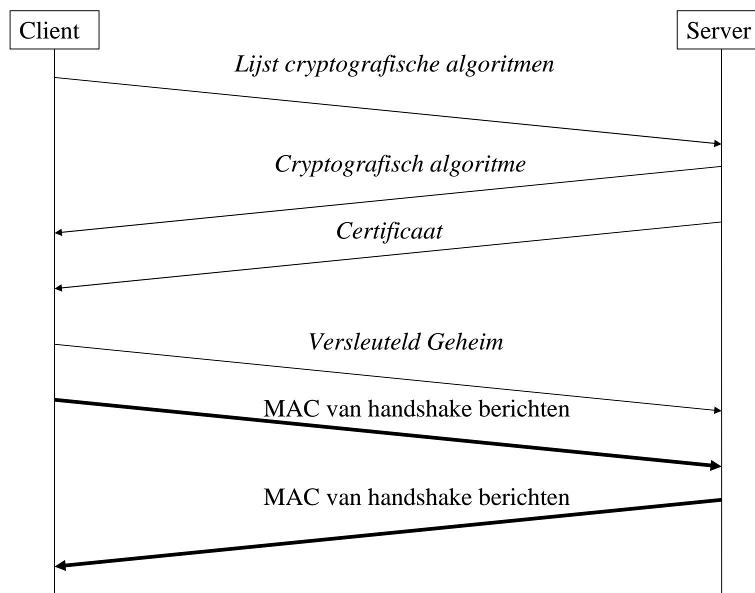
Client en server zijn nu in staat met elkaar af te spreken welke algoritmen ze zullen gebruiken om een veilige connectie op te zetten, en wisselen een geheim uit dat gebruikt zal worden in de symmetrische encryptie van de data-overdracht.

De *integriteit* van de handshake is echter nog op geen enkele manier gegarandeerd. Op het eerste zicht lijkt dit ook niet nodig, enkel de data zelf moet gegarandeerd ongewijzigd tussen beide partijen kunnen reizen. SSLv2 implementeert dan ook geen integriteitscontrole voor de handshake.

Maar als Joe de bedoeling van de handshake nader bekijkt, beseft hij dat de integriteit van de handshake wel degelijk van invloed is op de veiligheid van de data. De handshake is immers verantwoordelijk voor het opzetten van een veilige connectie. Indien de integriteit van de handshake niet gewaarborgd wordt, is het mogelijk dat een aanvaller met de handshake geknoeid heeft, en zo bijvoorbeeld de negotiatie van cryptografische algoritmen heeft gemanipuleerd.

Het gemis van zo'n integriteitscontrole in SSLv2 bleek achteraf dan ook één van de grootste zwakheden in deze versie.

Het is dus nodig ook de integriteit van de handshake te bewaken. Joe komt op het idee om op het einde van de handshake een bericht met een MAC over alle voorgaande (inkomende en uitgaande) handshake berichten te sturen (Figuur 3.4):



Figuur 3.4. Handshake met integriteitscontrole

De berekening van deze berichten gebeurt op de volgende wijze:

- De client berekent een MAC over alle voorgaande (inkomende en uitgaande) handshake berichten. Deze MAC steekt hij in een nieuw bericht, waarna hij dit bericht encrypteert met het overeengekomen symmetrische encryptie-algoritme. Vervolgens wordt dit versleutelde bericht naar de server gestuurd.
- De server ontvangt dit bericht, decrypteert het, extraheert de MAC en genereert

zelf een MAC over de voorgaande (inkomende en uitgaande) handshake berichten, waarna hij beide vergelijkt. Indien ze gelijk zijn, weet de server dat de integriteit van de handshake gewaarborgd is.

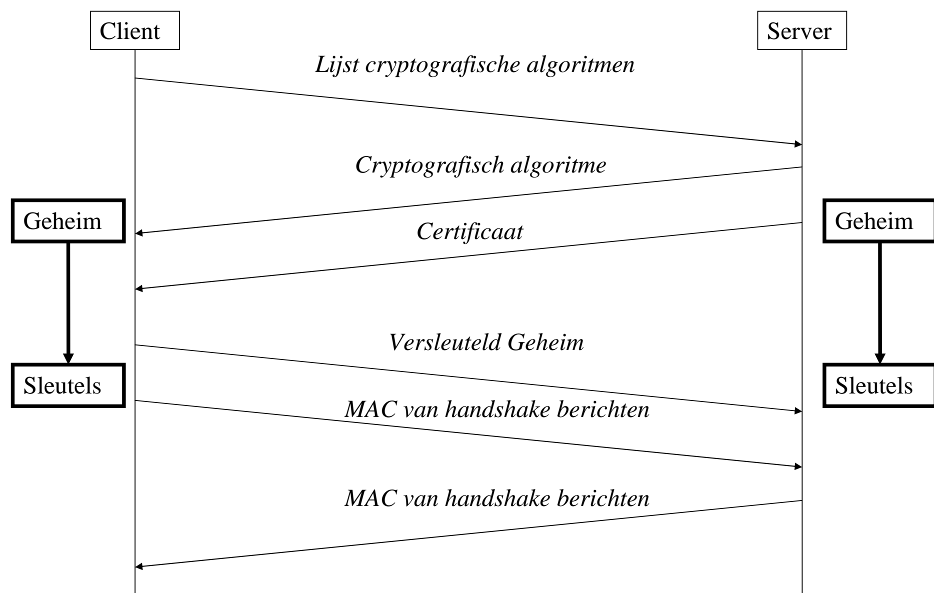
- Vervolgens genereert de server een nieuwe MAC over de voorgaande handshake, en stuurt deze geëncrypteerd naar de client.
- De client voert dezelfde controle uit als de server, en komt op deze manier op zijn beurt te weten of aan de integriteitseis voldaan is.

Deze aanpak zal inderdaad de integriteit van de handshake garanderen.

3.4.5 Meerdere geheime sleutels

Op dit moment gebruikt de handshake van Joe een aantal verschillende cryptografische algoritmen die een sleutel nodig hebben (zoals symmetrische en asymmetrische encryptie, en MAC's). De sleutel voor asymmetrische encryptie wordt gevormd door de publieke sleutel uit het certificaat. Vervolgens wordt met behulp van deze asymmetrische encryptiemethode een geheime sleutel uitgewisseld, die dan gebruikt wordt in de symmetrische encryptie en voor de MAC's.

Joe besluit echter dat het veiliger is om meerdere geheime sleutels te gebruiken, één per cryptografisch algoritme. Daarom ontwerpt hij een *sleutelafleidingsfunctie* (zie Figuur 3.5). Het uitgewisselde geheim zal nu niet meer rechtstreeks als geheime sleutel gebruikt worden, maar gebruikt worden om meerdere geheime sleutels af te leiden met deze functie.



Figuur 3.5. Handshake met sleutelafleiding

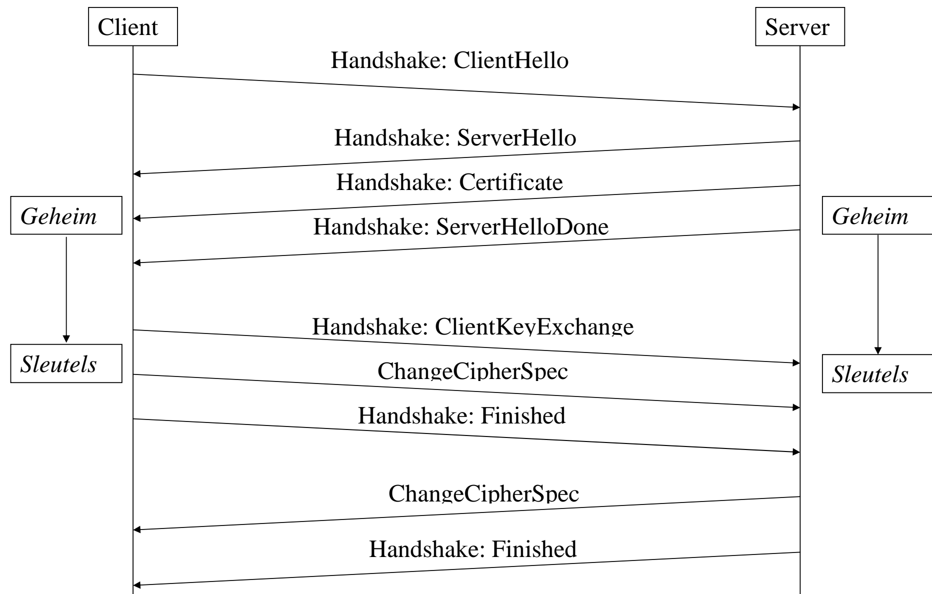
3.4.6 De eigenlijke SSL handshake

Nu onze imaginaire beveiligingsspecialist Joe in verschillende stappen eindelijk tot een bevredigende handshake is gekomen, vergelijken we zijn ontwerp met het eigenlijke SSL Handshake Protocol.

Het SSL Handshake Protocol

In Figuur 3.6 wordt een echte SSL handshake weergegeven. Op enkele details na, zijn dit de berichten uit het ontwerp van Joe (Figuur 3.5):

- **ClientHello**
  In deze 'begroeting' laat de client aan de server weten dat hij een veilige connectie wil opzetten. De client stuurt in dit bericht onder andere een lijst van cryptografische algoritmen mee die hij ondersteunt.
- **ServerHello**
  In dit bericht beantwoordt de server de begroeting van de client, en laat hij onder andere weten welke set cryptografische algoritmen hij gekozen heeft.
- **Certificate**
  De server stuurt zijn certificaat door.
- **ServerHelloDone**
  Dit is een leeg bericht dat aangeeft dat de server alle berichten heeft gestuurd die hij in deze fase wil sturen. Dit is nodig omdat er een aantal optionele berichten (bijvoorbeeld een bericht voor client authenticatie) zijn die de server na zijn certificaatbericht kan verzenden.
- **ClientKeyExchange**
  In dit bericht stuurt de client het geheim door dat hij gegenereerd heeft, geëncrypteerd met de publieke sleutel uit het certificaat van de server. Uit dit geheim, de *pre-master secret*, zal een *master secret* worden berekend, waaruit vervolgens alle geheime sleutels voor de cryptografische operaties afgeleid worden.
- **ChangeCipherSpec**
  Dit bericht geeft aan dat de client (of server) alle volgende berichten in geëncrypteerde vorm (met het overeengekomen symmetrische encryptie-algoritme) zal versturen. Dit bericht behoort **NIET** tot de handshake. Dat wil zeggen dat deze berichten NIET gebruikt worden bij het berekenen van de MAC's over de handshake berichten. Het idee was dat dit de performantie zou verhogen, maar in praktijk blijkt dit een onnodige complicatie die zelfs een veiligheidsrisico met zich meebrengt. Later wordt hier uitgebreider op teruggekomen.
- **Finished**
  Dit bericht bevat een MAC over alle voorgaande handshake berichten. Het is bovendien geëncrypteerd (want gestuurd na het *ChangeCipherSpec* bericht).



Figuur 3.6. De SSL Handshake

Structuur van de Handshake berichten

Alle handshake berichten worden verstuurd in een speciale handshake structuur, die bestaat uit een korte header gevolgd door het eigenlijke handshake bericht.

De header bestaat uit twee velden:

- **handshaketype**
  Geeft het soort handshake bericht aan. De belangrijkste mogelijkheden zijn als volgt:
  - client_hello
  - server_hello
  - certificate
  - server_hello_done
  - client_key_exchange
  - finished
- **lengte**
  De lengte van het handshake bericht (zonder header).

3.4.7 Sessies

Nu we de eigenlijke SSL Handshake hebben voorgesteld, is het duidelijk dat dit een eerder gecompliceerde operatie is, die uit heel wat berichten bestaat. Bovendien is de handshake een relatief kostelijke operatie, aangezien het een asymmetrische encryptie-operatie bevat, namelijk het encrypteren van de *pre-master secret* met de publieke sleutel uit het certificaat.

Sessies versus connecties

SSL biedt echter een mechanisme aan om bij opeenvolgende connecties, tussen eenzelfde server en client, een deel van de handshake over te slaan. Hiervoor voert men het begrip *sessie* in.

Een **SSL connectie** stelt één bepaald beveiligd communicatiekanaal voor, met zijn specifieke sleutels, cryptografische algoritmen, ...
Een SSL connectie loopt altijd bovenop precies één TCP connectie.

Een **SSL sessie** daarentegen is een virtuele constructie die de overeengekomen versleutelset en *master secret* bevat. Elke keer dat een volledige handshake doorlopen wordt, wordt een nieuwe sessie aangemaakt.

Een sessie kan dus uit meerdere connecties bestaan, die alle dezelfde *master secret* en versleutelset delen, maar waarmee telkens andere sleutels worden afgeleid (zoals we later in meer detail zullen zien) om de connectie te beveiligen.

Sessie hernemen

Sessies maken het zo mogelijk om een deel van de handshake over te slaan, en onmiddellijk over te gaan op het afleiden van de uiteindelijke sleutels om de connectie te beveiligen. Het deel dat overgeslagen wordt is het negotiëren van de versleutelset, het encrypteren en doorsturen van de *pre-master secret*, en het afleiden van *master secret*. Met andere woorden, het meest kostelijke deel wordt overgeslagen.

Dit werkt als volgt:

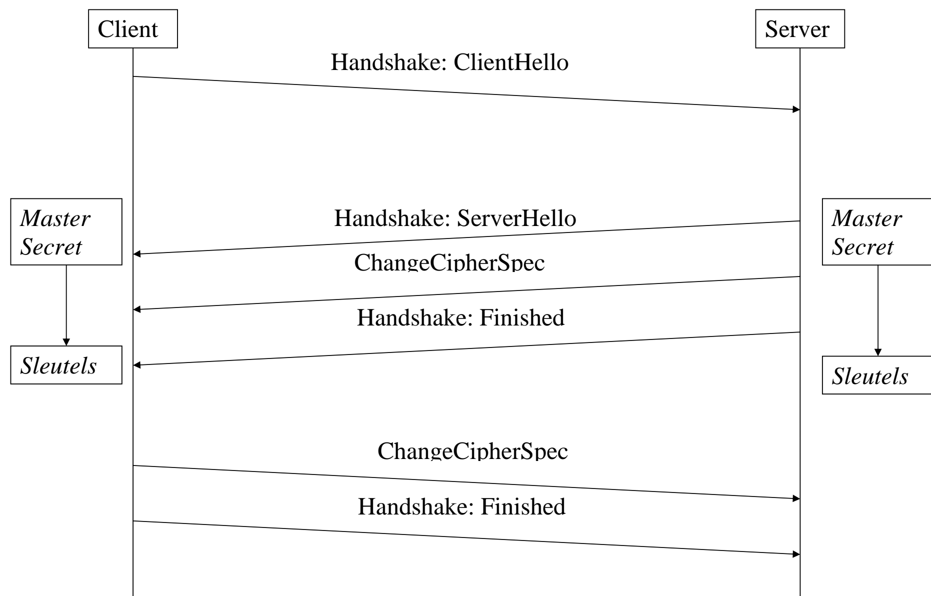
De eerste maal dat client en server met elkaar communiceren, doorlopen ze een gewone handshake. Indien de server bereid is om sessies te hernemen, stuurt hij in zijn `ServerHello` een `session_id` mee, en bewaart hij zelf de staat (met andere woorden, de overeengekomen versleutelset en de gegenereerde *master secret*) van deze sessie in zijn *sessie cache*.

De volgende maal dat de client met de server wil communiceren, stuurt hij dit `session_id` mee in zijn `ClientHello`. Indien de server dit `session_id` nog ondersteunt (sessies vervallen na verloop van tijd[5]), zal hij antwoorden met een `ServerHello` dat diezelfde `session_id` bevat. Vervolgens wordt de rest van de handshake overgeslagen, en wordt

[5] In de SSL specificatie wordt een maximum van 24 uur aangeraden

onmiddellijk overgegaan op het genereren van de rest van het sleutelmateriaal. Als de server dit `session_id` niet ondersteunt, wordt gewoon een volledige handshake doorlopen.

Dit proces is geïllustreerd in Figuur 3.7.



Figuur 3.7. De SSL Handshake bij sessies

We besluiten met op te merken dat sessies *niet* hernomen kunnen worden, indien de laatste connectie binnen deze sessie met een foutmelding beëindigd werd. Dit verhoogt de veiligheid, maar is ook en vooral omwille van performantieredenen interessant: de kans dat een sessie die eenmaal mislukt is opnieuw mislukt, is groot.

3.5 SSL Record Protocol

Nu we de handshake fase volledig besproken hebben, wordt de implementatie van het SSL Record protocol voorgesteld.

3.5.1 Gegevenstransfer

Eerder zagen we al dat het doel van dit subprotocol eruit bestaat om de veilige connectie, opgezet tijdens de handshake fase, effectief te *gebruiken* om onze gegevens op een veilige manier door te sturen. Dat wil zeggen, de integriteit en de vertrouwelijkheid van de gegevens blijft bewaard.

Deze twee eigenschappen worden als volgt gegarandeerd:

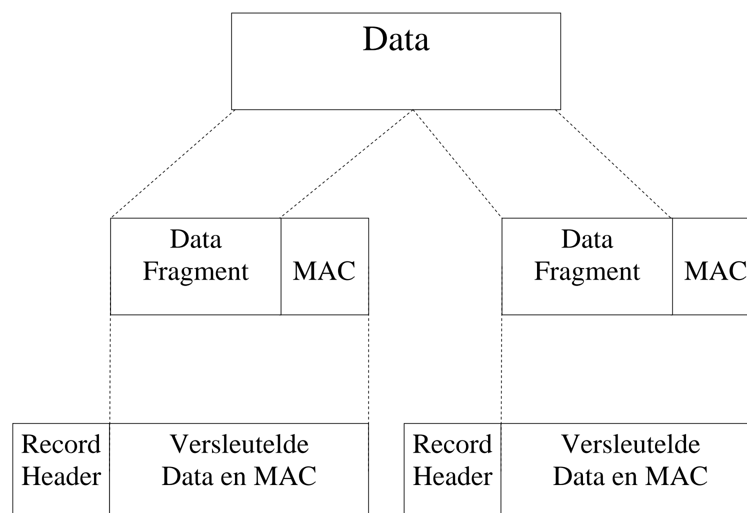
- **Integriteit**
  Hiervoor wordt een MAC berekend, die geverifieerd wordt bij ontvangst.
- **Vertrouwelijkheid**
  Het geheel van data en MAC zal geëncrypteerd worden volgens het symmetrische encryptie-algoritme dat overeengekomen werd tijdens de handshake fase.

Het versleutelde geheel uit de laatste stap wordt dan gebruikt als *payload*, waaraan vervolgens een *header* verbonden wordt. Dit geheel vormt een record.

Deze werkwijze, waarbij eerst een MAC wordt berekend over het bericht, waarna vervolgens het bericht en de MAC tezamen geëncrypteerd worden, noemt men *authenticate-then-encrypt (AtE)*. Hugo Krawczyk bewijst in 2001 dat deze manier bij sommige combinaties van MAC- en encryptie-algoritmen onveilig is, en raadt aan om *encrypt-then-authenticate(EtA)*[6] te gebruiken, welke wel altijd veilig is. Gelukkig toont hij daarnaast aan dat de in SSL gebruikte combinaties *wel* veilig zijn [42].

Het Record protocol zal de door te sturen gegevens fragmenteren, en elke fragment wordt afzonderlijk beschermd en in een record gegoten volgens de eerder beschreven procedure, waarna dit record verzonden wordt. Bij ontvangst wordt dan de inhoud van elk record afzonderlijk weer gedecrypteerd en geverifieerd. Deze onafhankelijkheid tussen de records heeft als voordeel dat de data verstuurd kan worden zodra ze beschikbaar is en verwerkt worden zodra ze aankomt.

De volledige procedure is grafisch voorgesteld in Figuur 3.8.



Figuur 3.8. Het SSL Record Protocol: datafragmentatie en -bescherming

[6] **IP Secure (IP Sec)**, een protocol dat netwerkcommunicatie beveiligt op IP niveau, gebruikt deze methode.

Record Header

De record header verschaft de ontvanger alle informatie die noodzakelijk is voor de verwerking van de bijhorende payload.

Een record header bestaat uit drie velden:

- **Inhoudstype**
  Identificeert het type van het bericht.
- **Lengte**
  Het aantal bytes waaruit de payload bestaat.
- **Versie**
  Dit geeft aan volgens welke versie het record is opgemaakt.

Inhoudstypes

Het feit dat er een onderscheid gemaakt wordt tussen verschillende recordtypes is op het eerste zicht wat vreemd. Is de inhoud van alle records niet hetzelfde? Namelijk gegevens, data die over de beveiligde connectie worden verstuurd?

Op implementatieniveau rijst echter de vraag op welke manier bijvoorbeeld de handshake berichten verstuurd zullen worden. Dit wordt opgelost door het Record Protocol te gebruiken om *alle* SSL data te versturen, ten koste van een lichte vervaging van de scheiding tussen het Record en Handshake Protocol die tot nu toe gesuggereerd werd.

SSL data wordt onderverdeeld in vier types, tegelijk ook de vier recordtypes die het Record Protocol zal onderscheiden:

- **application_data**
  Alle gegevens die verstuurd worden over de beveiligde connectie behoren tot het type *application_data*. Dit is dus de data die men zelfs bij een strikte scheidingspolitiek in het Record Protocol zou verwachten.
- **handshake**
  Dit zijn de handshake berichten van het SSL Handshake Protocol.
- **alert**
  Dit type wordt gebruikt voor foutenrapportering en het afsluiten van een connectie.
- **change_cipher_spec**
  Het speciale ChangeCipherSpec bericht wordt als een apart type beschouwd, en ook door het Record Protocol verstuurd (zie ook 3.4.6).

Het `application_data`, `handshake` en `change_cipher_spec` type hebben we al besproken, maar het type `alert` verdient wat meer uitleg en wordt besproken in de volgende sectie.

3.5.2 Alerts

Normale Alerts

Alerts, berichten van het recordtype `alert`, worden in de eerste plaats gebruikt om de andere partij te waarschuwen dat er een fout is opgetreden.

Er bestaan twee soorten alerts:

- **fatale alert ('fatal alert')**
  Dit is de ergst mogelijke gradatie. Bij het ontvangen van zo'n alert **moet** de ontvanger de connectie onmiddellijk beëindigen.
- **waarschuwing ('warning alert')**
  Een alert van dit type mag genegeerd worden door de ontvanger. Een implementatie mag er echter ook voor kiezen zo'n waarschuwing als *fataal* te behandelen, en de connectie te beëindigen.

In praktijk beschouwen de meeste implementaties elke alert als *fataal*, en wordt de connectie onmiddellijk beëindigd. Bijlage A.2 geeft een overzicht van alle mogelijke alert berichten.

Sluiting Alert

Naast de normale alerts, gebruikt voor foutenrapportering, bestaat er één speciale alert, die gebruikt wordt om het einde van een connectie aan te geven, de waarschuwing `close_notify`. Hierop wordt later dieper ingegaan.

3.6 Voorwaarden voor een veilige SSL connectie

Op dit moment zou de lezer een duidelijk beeld moeten hebben van *wat* SSL doet, en *hoe* het dit doet.

We herhalen nog eens dat het doel van SSL bestaat uit het opzetten en gebruiken van een veilige connectie over een per definitie[7] onveilig communicatiekanaal.

In de vorige secties werd uitgebreid uitgelegd hoe SSL dit doel waarmaakt.

Alles samengevat kunnen we stellen dat de goede werking (en dus de veiligheid) van SSL afhangt van de volgende, dikwijls van elkaar afhankelijke, elementen:

1. **Pre-master secret**
   De *pre-master secret* is de basis van alle gebruikte geheime sleutels. Bovendien wordt hij uitgewisseld over een onbeveiligd kanaal.

[7] volgens het Internet Threat Model

2. **Private sleutel van de server**
   Aangezien de publieke sleutel van de server gebruikt wordt om de *pre-master secret* geëncrypteerd naar de server ter sturen, is het van vitaal belang dat de bijhorende private sleutel uitsluitend bekend is aan de server. Een compromittering van deze private sleutel betekent dus automatisch een compromittering van de *pre-master secret*.

3. **Sterke random generator**
   Zowel client als server moeten een sterke random generator gebruiken, aangezien vele gebruikte cryptografische algoritmen hierop steunen. Zo is de *pre-master secret* uit puntje 1 een random waarde, door de client gegenereerd, en ook de private sleutel uit puntje 2 steunt op een randomgenerator (van de server).

4. **Sterke versleutelset**
   De veiligheid van de connectie steunt volledig op de sterkte van de gebruikte cryptografische algoritmen. De sterkte van deze algoritmen hangt voor een belangrijk deel af van de versleutelset die overeengekomen werd tijdens de handshake, aangezien deze bepaalt welke algoritmen (met welke sleutellengtes) gebruikt zullen worden.

5. **Geldig certificaat**
   Een geldig certificaat is van vitaal belang, omdat het beslist met welke publieke sleutel de *pre-master secret* zal geëncrypteerd worden, en dus welke bijhorende private sleutel (puntje 2) dit versleutelde geheel zal kunnen decrypteren.

Hoofdstuk 4

Details van het SSL protocol

"Beware of the man who won't be bothered with details."
– William Feather

4.1 Inleiding

In het vorige hoofdstuk werd een duidelijk beeld geschetst van de werking van het SSL protocol. Daarbij werd in de eerste plaats gepoogd om de lezer het theoretische concept achter het SSL protocol bij te brengen, waarbij al te technische details vermeden werden.

In het eerste deel van dit hoofdstuk zal de lezer ook kennis maken met de meer technische kant van het SSL protocol, en worden een aantal fundamentele delen van het protocol uitvoerig, tot aan de implementatiedetails toe, besproken.

Vervolgens wordt in het tweede deel van dit hoofdstuk een uitbreiding van het SSL protocol voorgesteld, die van fundamenteel belang zal blijken in de volgende hoofdstukken.

4.2 Implementatie details

4.2.1 Negotiatie

Eerder werd al gesteld dat de handshake van SSL een dynamische negotiatie is. Het primaire doel van de negotiatie is het overeenkomen van de cryptografische algoritmen die gebruikt zullen worden om de toekomstige communicatie te beveiligen. In praktijk wordt er echter over niet minder dan *drie* parameters onderhandeld:

- **Versie**
  Zoals we eerder in Sectie 3.2.2 zagen, bestaan er verschillende SSL versies. Deze SSL versies zijn *backward compatible*, dat wil zeggen dat elke hogere versie in staat is te

communiceren met een lagere versie. Client en server moeten overeenkomen welke versie ze zullen gebruiken.

- **Cryptografische algoritmen**
  De verschillende cryptografische technieken die gebruikt worden om de connectie te beveiligen, vereisen elk een ander algoritme. Client en server komen overeen welke algoritmen ze hiervoor zullen gebruiken.

- **Compressie-algoritmen**
  Client en server kunnen overeenkomen om alle data te compresseren alvorens deze te versturen, om zo het gebruikte kanaal minder te belasten, en de communicatie vlotter te doen verlopen. In dat geval moeten ze bepalen welk compressie-algoritme ze zullen gebruiken.

  Omwille van patenten op compressie-algoritmen wordt deze optie echter zelden gebruikt. We nemen aan dat als compressiemethode altijd het `NULL`-algoritme genegotieerd zal worden, met andere woorden, er wordt geen compressie gebruikt.

De negotiatie gebeurt met behulp van de *ClientHello* en *ServerHello* berichten van de SSL Handshake (zie Figuur 3.6).

Laten we de negotiatie van versie en cryptografische algoritmen wat diepgaander bekijken.

Versienegotiatie

Het SSL protocol bestaat in praktijk uit drie mogelijke versies: *SSLv2*, *SSLv3.0* en *TLS* (in oplopende volgorde).

Tijdens de versienegotiatie proberen client en server de hoogst mogelijke versie vast te stellen die beiden ondersteunen. De client stuurt de server daarom het hoogste versienummer dat hijzelf ondersteunt, waarna de server ofwel met deze versie akkoord gaat (als hij deze ook ondersteunt), ofwel een lagere versie terugstuurt (de hoogste die hij wel ondersteunt):

> Met $versie_C$ de versie van de client en $versie_S$ de serverversie:
>
> $\texttt{versie}_C$ `= maximum(client ondersteunde versies)`
> $\texttt{versie}_S$ `= minimum(maximum(server ondersteunde versies), ` $\texttt{versie}_C$`)`

Een SSL server verwacht dat een client die een hoge versie rapporteert, ook alle lagere versies aanvaardt, hoewel dit niet verplicht is. Het is voor een server of client perfect mogelijk om de *backward compatibility*, de ondersteuning voor eerdere versies, gedeeltelijk of volledig uit te schakelen. Er is echter geen manier om aan te geven dat bepaalde lagere versies niet ondersteund worden, aangezien er slechts plaats voorzien is voor één versienummer in het negotiatiebericht (de `ClientHello`).

Indien een client een bepaalde lagere versie niet ondersteunt, maar de server probeert deze toch te negotiëren, dan zal de connectie onmiddellijk worden beëindigd door de client. Andersom zal een server die een versienummer van de client krijgt dat lager is dan de versies die hij ondersteunt, ook de connectie beëindigen.

Hierbij moeten we nog opmerken dat de client een `ClientHello` zal sturen in het formaat van de *laagste* versie die hij ondersteunt. Indien dat SSLv2 is, zal hij dus een SSLv2 `ClientHello` sturen, hoewel deze echter wel het *hoogste* versienummer zal bevatten dat hij wil aanvaarden.

We geven in dit kader nog een interessant citaat uit de SSLv3 specificatie, die verscheen in 1996 [24]:

> "*De mogelijkheid om Versie 2.0 ClientHello berichten te sturen, zal met alle mogelijke haast afgebouwd worden. Implementators moeten elke mogelijke inspanning doen om zo snel mogelijk de nieuwe versie [Versie 3.0] te ondersteunen.*"

De meeste SSL clients en servers ondersteunen echter nog steeds, 8 jaar na het verschijnen van deze specificatie, SSLv2 (met bijhorende Versie 2.0 `ClientHello` berichten), zoals we later in een onderzoek zullen aantonen.

Negotiatie van de cryptografische algoritmen

Client en server moeten ook overeenkomen welke cryptografische algoritmen ze zullen gebruiken. Aangezien SSL verschillende cryptografische primitieven gebruikt die elk een eigen algoritme vereisen, moeten ze niet één algoritme negotiëren, maar een *set* van algoritmen. Zo'n set van algoritmen zullen we een *versleutelset* ('*cipher suite*') noemen.

Een versleutelset bestaat uit één cryptografisch algoritme voor elk van de gebruikte cryptografische operaties, zoals weergegeven in Tabel 4.1.

Doel	Operatie	Algoritme
Authenticatie	Digitale handtekening (certificaat)	RSA
Sleuteluitwisseling	Asymmetrische encryptie	RSA
Data-encryptie	Symmetrische encryptie	RC4/RC2/DES/3DES/AES
Integriteit	Digest (in MAC)	MD5/SHA-1

Tabel 4.1. Cryptografische operaties in SSL en hun algoritmen (enkel de meestgebruikte algoritmen worden weergegeven)

De mogelijke versleutelsets hangen af van de versie die server en client ondersteunen. Daarnaast kan elke partij ook nog bepaalde versleutelsets expliciet uitschakelen, om te vermijden dat deze genegotiëerd worden. De ondersteunde versleutelsets worden dan door de client in volgorde van zijn voorkeur (eerste keuze eerst), in het *ClientHello* bericht naar de server gestuurd.

Welke versleutelset uiteindelijk gekozen moet worden uit de lijst van mogelijkheden die de client de server toestuurt, is niet gespecificeerd in de SSL specificatie.

In OpenSSL gebeurt het zo dat de server de gesorteerde versleutelsetlijst van voor naar achter afloopt, en de eerste die hij ook ondersteunt, eruit pikt. De server kan ook zo

worden geconfigureerd dat zijn eigen voorkeuren de prioriteit krijgen[1]; hij loopt dan een eigen gesorteerde lijst af, en kiest de eerste versleutelset uit zijn lijst die ook op de lijst van de client voorkomt.

Een lijst van mogelijke versleutelsets wordt gegeven in Bijlage A.1.

In de versleutelset zit tevens de sleutellengte van het gebruikte symmetrische algoritme vervat. De sleutellengte van de asymmetrische encryptie wordt echter niet gespecificeerd, een server is volledig vrij om de sleutellengte van zijn RSA sleutelpaar zelf te bepalen. In praktijk is zo'n sleutelpaar gewoonlijk 1024 bit lang.

Het is belangrijk te beseffen dat de veiligheid van de connectie die opgezet zal worden, compleet afhankelijk is van de overeengekomen versleutelset en de lengte van de private serversleutel. De versleutelset bepaalt immers met welke cryptografische algoritmen de verschillende veiligheidseigenschappen gegarandeerd zullen worden, en de serversleutellengte beïnvloedt de veiligheid van de sleuteluitwisseling.

4.2.2 Sleutelafleiding

In de vorige sectie bespraken we hoe client en server met elkaar overeenkomen welke cryptografische algoritmen ze zullen gebruiken om een veilige connectie op te zetten. Deze algoritmen hebben echter alle één of andere sleutel nodig om te kunnen functioneren.

De digitale handtekening en asymmetrische encryptie zijn beide vormen van publieke sleutel-cryptografie, en hebben dus een *publieke sleutel* nodig, waar MAC en symmetrische encryptie onder geheime sleutel-cryptografie vallen, en een *geheime sleutel* vereisen.

De verspreiding van publieke sleutels vormt geen probleem, en gebeurt met behulp van een certificaatmechanisme, zoals we in Hoofdstuk 2 zagen. Ook het sleutelmanagement-probleem van geheime sleutel-cryptografie werd opgelost, door tijdens de handshake een sleuteluitwisseling met behulp van publieke sleutel-cryptografie te implementeren.

Sleuteluitwisseling is hier echter een beetje ongelukkig verwoord, aangezien geen sleutel wordt uitgewisseld, maar een *geheim*, dat de basis zal vormen van de echte sleutels. Dit laat toe om meerdere sleutels te gebruiken, één voor elke cryptografische operatie, waardoor de veiligheid sterk verhoogd wordt.

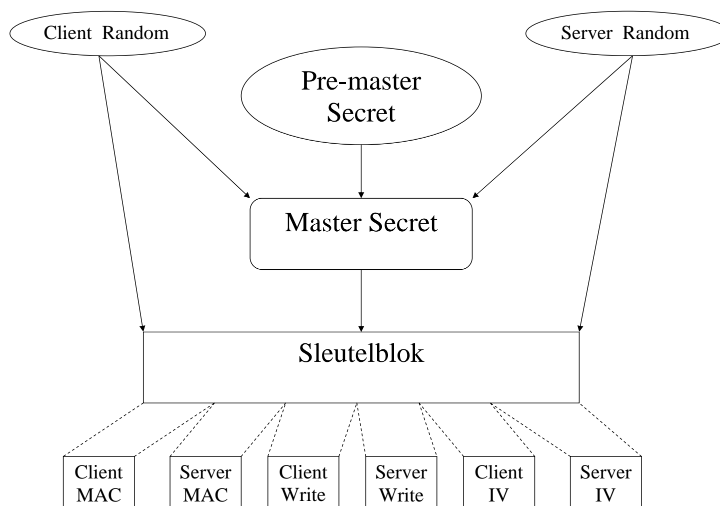
Met een cryptografische operatie wordt zowel het algoritme, als de *richting* van de operatie bedoeld. Dat wil zeggen dat er voor elk algoritme twee sleutels worden gegenereerd, één voor elke richting. Er zal dus een symmetrische sleutel zijn voor serverencryptie en clientdecryptie (de ene richting) en een andere voor clientencryptie en serverdecryptie (andere richting), hoewel telkens hetzelfde algoritme wordt gebruikt.

In SSL terminologie wordt het uitgewisselde, random gegenereerde geheim de *pre-master secret* genoemd. Door middel van een sleutelafleidingsalgoritme zal uit deze `pre_master_secret` een *master secret* worden afgeleid. Met behulp van de `master_secret` wordt dan een *sleutelblok* gegenereerd, dat in stukken wordt gehakt om de uiteindelijke sleutels voor de verschillende geheime sleutel-algoritmen te verkrijgen.

[1] Door de SSL_OP_CIPHER_SERVER_PREFERENCE optie te zetten met behulp van de SSL_CTX_set_options() functie

De sleutels die afgeleid dienen te worden zijn de sleutels voor de MAC (integriteit) en symmetrische encryptie (data-encryptie). Bovendien komt hier nog een initialisatievector (IV) bij, indien er tijdens de negotiatie van de versleutelset voor een block cipher gekozen werd. Dit zijn dus drie sleutels voor elke richting, een totaal van zes sleutels.

De sleutelafleidingsprocedure wordt grafisch voorgesteld in Figuur 4.1.



Figuur 4.1. SSL sleutelafleiding

In de figuur is duidelijk te zien dat niet enkel de `pre_master_secret` aan de basis van de eigenlijke sleutels ligt, maar dat er twee extra waarden gebruikt worden tijdens de sleutelafleiding:

- **client_random**
  Een random waarde gegenereerd door de client, en in de `ClientHello` meegestuurd naar de server.
- **server_random**
  Random waarde gegenereerd door de server, en in de `ServerHello` naar de client gestuurd.

Merk op dat deze randomwaarden onversleuteld over het communicatiekanaal verstuurd worden, aangezien `ClientHello` en `ServerHello` niet geëncrypteerd worden. De waarden worden wel beschermd tegen manipulatie door de integriteitscontrole over de hele handshake.

Het gebruik van deze randomwaarden vormt een garantie dat zelfs indien eenzelfde `pre_master_secret` meermaals gebruikt wordt, de gegenereerde sleutels toch telkens anders zijn. Zo wordt een zogenaamde *replay attack* voorkomen. In zo'n aanval onderschept en bewaart de aanvaller bepaalde berichten (in het bijzonder versleutelde berichten) en hergebruikt hij deze later (zelfs zonder dat hij de inhoud ervan kent). Een aanvaller kan bijvoorbeeld alle berichten van een client domweg hergebruiken om zo een voorgaande connectie met de server te herhalen.

Hoewel de aanvaller in een replay attack niets aan de hergebruikte berichten kan veranderen (wegens geëncrypteerd), is deze aanval te vermijden, aangezien op deze manier bijvoorbeeld een online bestelling één of meerdere keren herhaald zou kunnen worden.

Zulke aanvallen worden vermeden door het gebruik van de randomwaarden.

Samengevat bestaat de sleutelafleiding dus uit het volgende:
beide partijen genereren *onafhankelijk* van elkaar *dezelfde* zes sleutels aan de hand van twee (niet-geheime) random waarden en een geheime random waarde, de `pre_master_secret`.

SSLv3 sleutelafleiding

We bespreken nu hoe de sleutelafleiding in praktijk geïmplementeerd wordt. De sleutelafleidingen in SSLv3 en TLS zijn vergelijkbaar, maar verschillen in een aantal details. In deze sectie geven we een volledige bespreking van de SSLv3 sleutelafleiding, waarna in de volgende sectie kort de details besproken worden waarin TLS hiervan verschilt.

SSLv3 leidt de `master_secret` als volgt uit de `pre_master_secret` af (*'+' duidt op concatenatie*):

```
master_secret =
MD5(pre_master_secret + SHA-1("A" + pre_master_secret +
client_random + server_random))
+ MD5(pre_master_secret + SHA-1("BB" + pre_master_secret +
client_random + server_random))
+ MD5(pre_master_secret + SHA-1("CCC" + pre_master_secret +
client_random + server_random))
```

Het gebruik van de constanten “A”, “BB”, “CCC”[2] zorgt ervoor dat elk onderdeel van de berekening een ander resultaat geeft.

Het sleutelblok wordt dan als volgt berekend aan de hand van de `master_secret`:

```
sleutelblok =
MD5(master_secret + SHA-1("A" + master_secret +
server_random + client_random))
+ MD5(master_secret + SHA-1("BB" + master_secret +
server_random + client_random))
+ MD5(master_secret + SHA-1("CCC" + master_secret +
server_random + client_random))
+ [...]
```

De “[...]” geeft aan dat de concatenatie van MD5 digests doorgaat tot het sleutelblok voldoende lang is (één MD5 digest is immers slechts 16 bytes lang).

[2] De vorm van deze constanten is op het eerste zicht wat vreemd, maar is gericht op een efficiënte implementatie.

Let op het feit dat de volgorde van `client_random` en `server_random` in beide berekeningen omgekeerd is. Ook wordt er gebruik gemaakt van twee verschillende digest-algoritmen[3] om de beveiliging te verhogen. Een aanvaller moet nu immers beide algoritmen (MD5 en SHA-1) breken om de sleutelafleiding te compromitteren.

TLS sleutelafleiding

De TLS sleutelafleiding is geen combinatie van digest algoritmen, maar gebruikt een speciale functie, de zogenaamde **pseudo-random function (PRF)**. Deze functie is gebaseerd op het MAC-algoritme HMAC. De details van deze functie kunnen nagelezen worden in [60, p.84-85], maar zijn niet noodzakelijk voor het begrijpen van deze studie.

De PRF verwacht drie inputs, en genereert daarmee een pseudo-random string van arbitraire lengte:

- **secret**
  Een (meestal random) geheim.
- **label**
  Een vaste ASCII string. Deze parameter laat toe verschillende sleutels te genereren met eenzelfde geheim door telkens een ander label te gebruiken.
- **salt**[4]
  Een randomwaarde, maar niet geheim.

De berekeningen van de `master_secret` en het sleutelblok zien er dan als volgt uit:

```
master_secret = PRF(pre_master_secret, "master secret",
client_random + server_random)

sleutelblok = PRF(master_secret, "key expansion",
server_random + client_random)
```

Ook hier is de volgorde van de randomwaarden omgewisseld in beide berekeningen.

4.2.3 Finished berichten

De `Finished` berichten staan in voor de integriteit van de handshake door een MAC over de handshake berichten te berekenen, zoals we eerder in Sectie 3.4.4 zagen. Het zijn tevens de eerste berichten die geëncrypteerd worden met de overeengekomen cryptografische parameters.

Ook hier verschillen de berekeningen tussen SSLv3 en TLS.

[3]Deze hebben niets te maken met het digest algoritme dat genegotieerd wordt tijdens de handshake om de integriteit van de gegevenstransfer te bewaken.

[4]Eric Rescorla noemt dit een *seed*, maar aangezien wij een seed als *geheime* random data hebben gedefinieerd, en deze data niet geheim is, gebruiken we de term *salt*

SSLv3 Finished

De SSLv3 `Finished` berichten zijn gebaseerd op een gewijzigde versie van het MAC-algoritme HMAC (zie ook Sectie 2.10). Meer in het bijzonder werd het aangepast aan het feit dat in SSL de geheime sleutel pas beschikbaar is halfweg de handshake. Het MAC-algoritme in SSL aanvaardt dus eerst de handshake berichten, en pas als laatste de sleutel, waar het originele HMAC algoritme altijd de sleutel als eerste input vereist.

Een SSLv3 `Finished` bericht omvat twee MAC's, één gebaseerd op het MD5 algoritme en één met behulp van SHA-1. Beide gebruiken de `master_secret` als geheime sleutel:

De berekening van deze MAC's gebeurt als volgt (merk op dat de geheime sleutel (`master_secret`) wel degelijk als laatste ingevoerd wordt, aangezien eerst de geneste (binnenste) digest berekening gebeurt):

```
md5_MAC = MD5(master_secret + pad2 +
MD5(handshake_berichten + Sender + master_secret + pad1))
sha_MAC = SHA-1(master_secret + pad2 +
SHA-1(handshake_berichten + Sender + master_secret + pad1))
```

met de volgende parameters:

- *Sender*
  afhankelijk van de afzender:
  - Client
    `Sender = 0x434C4E54` ("CLNT" in hexadecimale vorm)
  - Server
    `Sender = 0x53535653` ("SRVR" in hexadecimale vorm)
- *handshake_berichten*
  de concatenatie van alle handshake berichten (zonder headers en MAC)
- *pad1* en *pad2*
  afhankelijk van het gebruikte digest algoritme:
  - MD5
    `pad1 = "0x36" 48 keer herhaald`
    `pad2 = "0x5C" 48 keer herhaald`
  - SHA-1
    `pad1 = "0x36" 40 keer herhaald`
    `pad2 = "0x5C" 40 keer herhaald`

De *Sender* constante heeft als doel om een reflectie van het `Finished` bericht terug naar zijn afzender te voorkomen (een soort *replay attack*). De `Finished` berichten die de server genereert zijn nu immers verschillend van de `Finished` berichten die de client verstuurt.

Merk op dat ook hier weer beide digest algoritmen gebruikt worden, wat de veiligheid vergroot.

TLS Finished

Omwille van de wijziging die SSLv3 doorvoert in zijn MAC-algoritme (geheime sleutel als laatste invoer in plaats van eerste) zou dit nieuwe algoritme bepaalde veiligheidseigenschappen van de originele HMAC ontberen [90]. Dus, hoewel er geen zwakheid van dit algoritme bekend is, werd beslist om in TLS de berekeningen op een andere manier uit te voeren.

De TLS `Finished` berichten bestaan slechts uit één component, *verify_data* genoemd. De berekening hiervan gebeurt met behulp van de al eerder besproken, op de originele HMAC gebaseerde, PRF functie:

```
verify_data = PRF(master_secret, finished_label,
MD5(handshake_berichten) + SHA-1(handshake_berichten))
```

Ook hier worden dus beide digest algoritmen gebruikt, om de veiligheid nog te vergroten.

4.2.4 Record Protocol

De primaire taak van het Record Protocol is het beveiligen van de data met de cryptografische parameters die overeengekomen zijn tijdens de handshake. We bekijken wat nader hoe dit in zijn werk gaat.

We zagen eerder al dat de data gefragmenteerd wordt in onafhankelijke records, waarover dan een integriteitscontrole wordt berekend (met een MAC), om het geheel uiteindelijk te encrypteren.

We merken nog op dat indien compressie gebruikt zou worden, de data gecomprimeerd zou worden alvorens de integriteitscontrole te berekenen. We gaan er echter vanuit dat compressie niet gebruikt wordt, zoals eerder gesteld.

Integriteitscontrole

Het berekenen van de MAC over het record is weer afhankelijk van de gebruikte versie.

De MAC over de SSLv3-records is gebaseerd op een vroege draft van HMAC, waar TLS de standaard HMAC-implementatie gebruikt. De berekeningen zien er als volgt uit:

```
SSLv3 MAC = digest(MAC_write_secret + pad2 +
digest(MAC_write_secret + pad1 + seq_num + lengte + inhoud))
TLS MAC = HMAC_digest(MAC_write_secret, seq_num +
inhoudstype + versie + lengte + inhoud)
```

met de parameters als volgt:

- *pad1* en *pad2*
  de padding strings zoals gedefinieerd in Sectie 4.2.3.
- *digest*
  het in de handshake overeengekomen digest algoritme (MD5 of SHA-1).
- *MAC_write_secret*
  de sleutel gegenereerd tijdens de sleutelafleiding (zie Sectie 4.2.2)
- *seq_num*
  een volgnummer dat client en server onafhankelijk van elkaar berekenen.

Dat volgnummer is wat extra uitleg waard:

Volgnummers starten op nul na het versturen of ontvangen van een `ChangeCipherSpec` bericht, en worden met één verhoogd bij elk bericht dat verstuurd wordt. Een clientbericht verhoogt het volgnummer van de client, terwijl een serverbericht het volgnummer van de server verhoogt. Het doel van deze volgnummers is het verhinderen van *replay attacks*, maar ook van *reordering attacks*[5]. Indien het volgnummer niet gelijk is aan het verwachte volgnummer, zal de integriteitscontrole falen.

Merk op dat het volgnummer hier dus gebruikt kan worden om een verkeerde volgorde te *detecteren* (het wordt gebruikt in de berekening van de MAC), maar niet om deze te verbeteren, aangezien het volgnummer op geen enkele manier af te lezen is uit het record. Dit heeft het belangrijke gevolg dat SSL altijd over een *reliable transport protocol* zoals TCP moet lopen, dat garandeert dat de berichten in de juiste volgorde ter bestemming worden afgeleverd. Een voorbeeld van een *niet-reliable* protocol, waarover SSL *niet* werkt, is UDP, het User Datagram Protocol.

Encryptie

Nadat de MAC berekend is, wordt het geheel versleuteld met het symmetrische encryptie-algoritme dat tijdens de handshake overeengekomen werd. De methode is afhankelijk van het type algoritme, stream of block cipher.

Stream ciphers vereisen geen padding, en kunnen dus zonder verdere verwerking het geheel encrypteren.

Block ciphers maken wel gebruik van padding, zoals uitgelegd in Sectie 2.5.2. Alle huidige SSL block ciphers gebruiken Cipher Block Chaining (CBC) mode. De *chaining* wordt aangehouden tussen de opeenvolgende records: het laatste ciphertext blok van record X wordt gebruikt als IV voor record X+1.

[5]Strikt gezien is een reordering attack een soort replay attack, maar als extra illustratie is het nuttig deze aanval hier te expliciteren.

NULL cipher

Eerder zagen we al dat de scheiding tussen het Handshake protocol en het Record protocol enigszins vervaagt op implementatieniveau, aangezien het Record protocol ook gebruikt wordt om handshake berichten te versturen. Afgezien van deze conceptuele vervaging zorgt dit echter voor een bijkomend probleem. De handshake gebeurt immers quasi volledig (met uitzondering van de `Finished` berichten) in plaintext, waardoor een extra mechanisme in het Record Protocol zou moeten worden ingebouwd om encryptie en MAC tijdens de handshake (voor de handshake berichten) uit te schakelen.

Om dit op te lossen werd een elegante oplossing bedacht. Alvorens beide partijen een versleutelset overeengekomen zijn, dus bij het versturen van de eerste handshake berichten, wordt de NULL-versleutelset[6] gebruikt als initiële versleutelset. Dat wil zeggen, alle gebruikte cryptografische algoritmen (voor authenticatie, sleuteluitwisseling, integriteit en data-encryptie) zijn NULL, onbestaande. Deze *context* wordt aangehouden tot beide partijen overschakelen op de overeengekomen versleutelset (aangegeven met het `ChangeCipherSpec` bericht).

Omwille van veiligheidsredenen is het verboden om deze NULL-versleutelset te negotiëren tijdens de handshake.

4.2.5 Beëindigen van een SSL connectie

Tot nu toe hebben we altijd gesproken over het *opzetten* en *gebruiken* van een veilige connectie. We hebben echter altijd nagelaten om dieper in te gaan op het *sluiten* of beëindigen van zo'n veilige connectie.

Op het eerste zicht lijkt deze sluiting geen enkel probleem op te leveren. Zoals we zagen in 3.2.1 loopt het SSL Protocol boven op het TCP protocol, en dit protocol biedt al een mechanisme aan om een connectie te beëindigen, met behulp van zogenaamde *TCP FIN* berichten.

Het steunen op dit TCP sluitingsmechanisme heeft echter een belangrijk nadeel vanuit beveiligingsoogpunt. Het maakt de connectie kwetsbaar voor een *truncatie-aanval* ('*truncation attack*'). In een truncatie-aanval stuurt de aanvaller een vervalst `TCP FIN` bericht in naam van één van beide partijen, vóór deze partij al haar data verzonden heeft. Zo'n bericht is makkelijk te vervalsen, aangezien er geen cryptografische operaties mee gemoeid zijn, en heeft het gevolg dat de bovenliggende SSL connectie voortijdig afgebroken wordt, zonder dat de andere partij merkt dat er iets verdachts gebeurd is.

Eerder zagen we tijdens de bespreking van het Record Protocol een speciaal alert bericht dat gebruikt wordt om een connectie te beëindigen, de `close_notify` waarschuwing. Aangezien dit bericht gestuurd wordt na het `ChangeCipherSpec` bericht, is dit bericht beveiligd met dezelfde technieken als de datatransfer, en kan een aanvaller zo'n bericht niet vervalsen. De integriteit van onze sluiting wordt op deze manier dus gewaarborgd.

Merk op dat dit bericht een truncation attack niet *voorkomt*, zo'n aanval is nog altijd mogelijk door een `TCP FIN` bericht te simuleren, en eigen aan het gebruik van het TCP protocol. Het bericht biedt echter wel de garantie dat zo'n aanval *gedetecteerd* zal worden.

[6] `TLS_NULL_WITH_NULL_NULL`

Indien een SSL connectie op ongeldige wijze (zonder `close_notify`) wordt afgesloten, wordt ook de bijhorende sessie ongeldig, en kan deze niet hernomen worden.

4.3 Uitbreiding: Ephemeral RSA

In deze sectie beginnen we met een uitgebreide bespreking over exportreguleringen die tot voor kort bestonden in de Verenigde Staten. Vervolgens bespreken we twee aanpassingen aan het SSL protocol, tengevolge van deze reguleringen, waarvan *Ephemeral RSA* de belangrijkste is voor deze studie.

4.3.1 Exportregulering

Deze sectie werd gebaseerd op [60, p.24-25].

Inleiding

In de Verenigde Staten bestond tot voor kort zoiets als *exportregulering*. Deze exportregulering was een initiatief van het **National Security Agency (NSA)**. Deze organisatie is verantwoordelijk voor de beveiliging van het communicatieverkeer van de regering en het afluisteren van het communicatieverkeer van individuen en groeperingen, die mogelijk een bedreiging vormen voor de nationale veiligheid.

Lang geleden

Tot september 1998 was het idee achter de export regulering eenvoudig. Het was legaal om encryptietechnologie te exporteren voor authenticatiegebruik, maar gebruik om vertrouwelijkheid te garanderen werd in grote mate gerestricteerd. Een product dat ook maar de minste vertrouwelijkheid aanbood, moest goedgekeurd worden voor het geëxporteerd mocht worden. De regels voor goedkeuring zijn nooit gepubliceerd, maar in praktijk begreep men dat 40 bit voor symmetrische encryptie en 512 bit voor asymmetrische encryptie het maximum toelaatbare waren.

Een product dat aan deze eisen voldeed, viel onder zogenaamde *commodities jurisdiction*, wat betekende dat een algemene exportvergunning voor dit product kon aangevraagd worden bij het **Departement of Commerce**. Onder deze vergunning mocht het product dan wereldwijd verkocht worden of ter beschikking worden gesteld voor download. Maar zelfs met deze classificatie mocht het product niet geëxporteerd worden naar zeven verboden landen: Cuba, Iran, Irak, Libië, Noord-Korea, Sudan en Syrië. Daarnaast beperkten ook andere landen zoals Frankrijk cryptografie in belangrijke mate.

Een ander opmerkelijk gegeven is het feit dat de NSA een speciaal goedkeuringsproces had voor de symmetrische sleutelalgoritmen RC2 en RC4. Indien deze algoritmen gebruikt werden (in 40 bit), in combinatie met een sleuteluitwisselingsalgoritme van ten hoogste 512 bit, was een goedkeuring zo goed als zeker, en sneller dan de standaardprocedure. Men

veronderstelde dat de NSA deze twee algoritmen uitgebreid geanalyseerd had, en misschien zelfs manieren had ontdekt om ze snel te kraken, al dan niet met speciale hardware. In elk geval bevoordeelde de NSA deze algoritmen sterk en bemoeilijkte ze de export van andere. RC2 en RC4 zijn dan ook terug te vinden in zowat alle cryptografische producten.

Er was één belangrijke uitzondering op de algemene regel. Indien kon worden aangetoond dat het te exporteren product enkel bestemd was voor financiële transacties waren sterkere cryptografische algoritmen *wel* toegelaten.

Tot voor kort

In september 1998 werden de encryptieregels versoepeld om de export van 56-bit encryptie en 1024-bit sleuteluitwisseling[7] toe te laten. Producten moesten nog altijd een goedkeuringsproces ondergaan, maar het gebruik van semi-sterke algoritmen werd nu toch mogelijk. Echt sterke algoritmen waren echter nog steeds verboden.

Tegenwoordig

In januari 2000 werden de regels dan uiteindelijk in die mate versoepeld, dat het mogelijk werd om sterke cryptografie te exporteren. Publiekelijk beschikbare broncode ('*open source software*') mocht zonder problemen op het internet ter beschikking worden gesteld. Commerciële producten moesten nog steeds een éénmalige technische keuring ondergaan waarvan het nut niet echt duidelijk was, maar in praktijk werden een heel aantal van deze producten met sterke cryptografie goedgekeurd voor export. Van toen af was het dus legaal om sterke cryptografie vanuit de Verenigde Staten te exporteren, behalve naar de eerder vermelde zeven verboden landen.

Export Regulering en SSL

Hoewel deze export reguleringen ondertussen verleden tijd zijn, hebben zij toch hun stempel gedrukt op vele producten. Zo zijn er ook in SSL nog altijd sporen terug te vinden van deze restricties.

Ten eerste ondersteunt SSL zwakke algoritmen in bepaalde versleutelsets. Deze versleutelsets zullen we de *export versleutelsets* noemen, en zijn te herkennen aan het keyword *EXPORT* in hun naam (voor een lijst van versleutelsets, zie Bijlage A.1).

Ten tweede bevat SSL een aantal aanpassingen die het mogelijk maken dat een niet-gerestricteerde server (met sterke cryptografie) en een gerestricteerde export client (met uitsluitend ondersteuning voor export versleutelsets en dus zwakke cryptografie) toch met elkaar kunnen communiceren:

- **Ephemeral RSA**

[7] We gebruiken hier de termen sleuteluitwisseling en asymmetrische encryptie door elkaar, aangezien dit in SSL (en in de cryptografische wereld in het algemeen) op hetzelfde neerkomt

In deze techniek past de server zich aan aan de client. De server zal dus overschakelen op *zwakkere* cryptografie, door een export versleutelset te aanvaarden, om met de export client te communiceren.

- **Server Gated Cryptography**
  Bij deze methode past de export client zich aan aan de server, door over te schakelen op *sterkere* cryptografie. Deze aanpassing komt tegemoet aan het feit dat sterke cryptografie *wel* was toegelaten bij financiële transacties, zelfs voor export clients.

Aangezien deze studie zwakheden van SSL probeert uit te buiten, zal vooral Ephemeral RSA uitgebreid besproken worden.

4.3.2 Ephemeral RSA

Het doel van Ephemeral RSA is het verzwakken van de cryptografie van een niet-gerestricteerde server, om de communicatie met een export-gerestricteerde client mogelijk te maken.

Een niet-gerestricteerde server is een server met sterke cryptografische algoritmen en bijgevolg sterke sleuteluitwisseling. Dat wil zeggen dat deze server publieke sleutel-cryptografie toepast met een sleutellengte van minstens 1024 bit. Met andere woorden, de server bezit een RSA sleutelpaar, bestaande uit een publieke en private sleutel, met een sleutellengte van minstens 1024 bit.

De export client daarentegen is wel gerestricteerd, en kan dus geen encryptie uitvoeren met publieke sleutels langer dan 512 bit.

Ephemeral RSA zal in dat geval de cryptografie van de server verzwakken door een nieuw, tijdelijk[8] RSA sleutelpaar te genereren, met een lengte van 512 bit. Deze nieuwe publieke sleutel wordt dan naar de client verstuurd, die deze sleutel gebruikt om de `pre_master_secret` te versleutelen en naar de server te sturen.

Om te vermijden dat een aanvaller deze nieuwe publieke sleutel manipuleert, wordt deze door de server "gehandtekend" met zijn oorspronkelijke, sterke private sleutel van minstens 1024 bit. De client kan bij ontvangst deze digitale handtekening dan verifiëren met de overeenkomstige publieke sleutel, om na te gaan of de sleutel echt afkomstig is van de server (authenticatie). Merk op dat *authenticatie* met sterke sleutels wel mogelijk is door de export client, maar *encryptie* niet, zoals gesteld in de export regulering (4.3.1).

Om Ephemeral RSA te bekomen zijn enkele aanpassingen aan het SSL protocol nodig:

- De SSL Handshake wordt uitgebreid met een extra bericht om deze nieuwe sleutel te versturen.
- Tijdens de sleutelafleiding worden zwakke sleutels afgeleid voor gebruik in de verschillende cryptografische algoritmen.
- Op de één of andere manier moet zo'n tijdelijke, korte RSA sleutel gegenereerd worden.

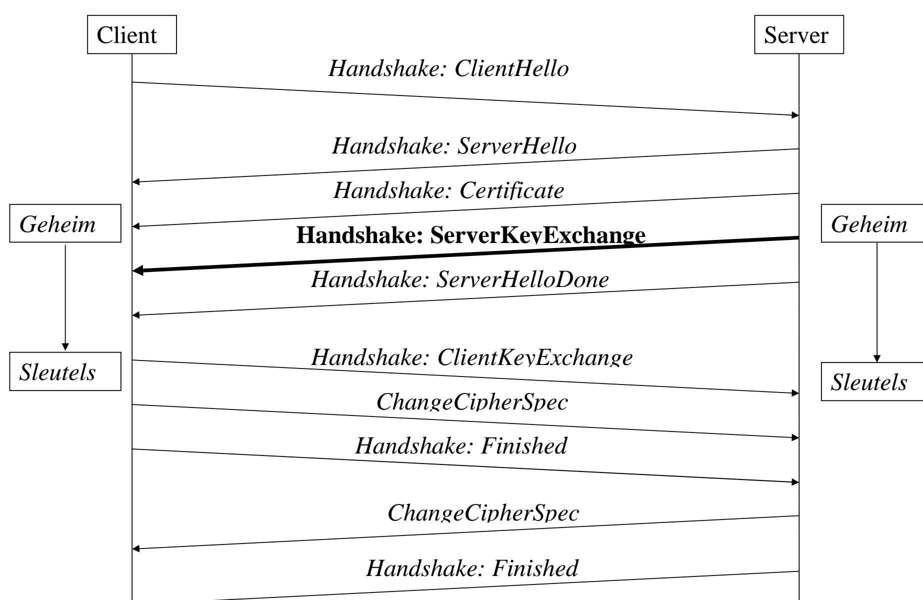
[8] ephemeral = kortstondig, tijdelijk

We bespreken deze aanpassingen één voor één.

Uitbreiding handshake

Bij gebruik van Ephemeral RSA wordt de handshake met één bericht uitgebreid om de nieuwe sleutel (met digitale handtekening) te versturen: het *ServerKeyExchange* bericht.

De handshake ziet er in dat geval uit als in Figuur 4.2.



Figuur 4.2. SSL Handshake bij Ephemeral RSA

Het `ServerKeyExchange` bericht zelf bevat in het geval van Ephemeral RSA[9] twee structuren:

- de publieke sleutel van het nieuwe Ephemeral RSA sleutelpaar
- een digitale handtekening over deze sleutel

De digitale handtekening wordt berekend aan de hand van twee message digests, waarbij de randomwaarden een replay attack voorkomen:

```
md5_digest = MD5(client_random + server_random + publieke sleutel)
sha_digest = SHA-1(client_random + server_random + publieke sleutel)
```

[9]Volledigheidshalve moeten we opmerken dat dit mechanisme ook met Diffie-Hellman (DH) bestaat, Ephemeral DH, dat we echter niet bespreken, hoewel het zekere voordelen heeft op Ephemeral RSA, zoals verderop duidelijk zal worden.

Aangepaste sleutelafleiding

In geval van Ephemeral RSA worden de gegenereerde sleutels tijdens de sleutelafleiding verder afgezwakt om te voldoen aan de exportrestricties. Bovendien verbiedt de exportregulering het gebruik van geheime IV's (of op geheime data gebaseerde IV's). Daarom worden de export IV's uitsluitend afgeleid op basis van de random waarden.

De berekening van de sleutels ziet er dan als volgt uit:

- **SSLv3**
  ```
  final_client_write_key = MD5(client_write_key +
  client_random + server_random)
  ```
  ```
  final_server_write_key = MD5(server_write_key +
  server_random + client_random)
  ```
  ```
  client_write_IV = MD5(client_random + server_random)
  server_write_IV = MD5(server_random + client_random)
  ```
- **TLS**
  ```
  final_client_write_key = PRF(client_write_key,
  "client write key", client_random + server_random)
  ```
  ```
  final_server_write_key = PRF(server_write_key,
  "server write key", client_random + server_random)
  ```
  ```
  iv_block = PRF(0, "IV block", client_random + server_random)
  ```

Het iv_block dat TLS genereert wordt opgesplitst in twee delen. Het eerste deel vormt de client IV, het tweede deel de IV van de server.

Genereren van de tijdelijke RSA sleutel

Het generen van de nieuwe, tijdelijke RSA sleutel is niet de verantwoordelijkheid van het SSL protocol zelf, en wordt meestal overgelaten aan de server implementatie.

Aangezien het genereren van RSA sleutels een computationeel dure operatie is, genereren de meeste server implementaties, waaronder die van OpenSSL, deze sleutel zodra hij nodig is, namelijk bij de eerste connectie met een export client. Diezelfde sleutel blijft dan in gebruik voor alle volgende export connecties, tot de server opnieuw gestart wordt. Zo'n export sleutel wordt dus typisch slechts éénmaal gegenereerd per *server run*, wat de performantie duidelijk ten goede komt. Vanuit veiligheidsoogpunt is deze aanpak echter iets minder positief, aangezien vele servers weken of zelfs maandenlang aan één stuk blijven draaien, en in die tijd voor alle export clients dezelfde tijdelijke RSA export sleutel gebruiken.

De SSL specificatie raadt dan ook aan [24]:

> "*Voor typische elektronische betaalapplicaties is het aangeraden om de sleutel dagelijks of elke 500 transacties te wijzigen, en zelfs meer, indien mogelijk.*"

In praktijk wordt dit echter niet gevolgd door de meeste implementaties, en moet "ephemeral" of "tijdelijk" met een korreltje zout genomen worden.

Indien echter wel voor elke sessie een nieuwe ephemeral sleutel wordt gegenereerd, voldoet de SSL sessie aan *perfecte, voorwaartse vertrouwelijkheid* ('*perfect forward secrecy*'). Dat wil zeggen dat het voor een aanvaller onmogelijk is om een *ephemeral* SSL sessie uit het verleden te compromitteren, zelfs indien hij over de originele, sterke private sleutel van de server beschikt. Dit is een gevolg van het feit dat deze private sleutel enkel gebruikt wordt om de *ephemeral* sleutel te handtekenen, en dat de *ephemeral* sleutel wordt 'weggegooid' nadat de sessie is afgelopen.

4.3.3 Server Gated Cryptography

In deze techniek, ook bekend als *Step-Up*, zal de export client zich aanpassen aan de sterke cryptografie van de server. Merk op dat dit volgens de exportreguleringen uitsluitend toegelaten was bij financiële transacties.

Het probleem hier is het feit dat een export client enkel exportversleutelsets mag ondersteunen, en een lijst met uitsluitend exportversleutelsets naar de server stuurt tijdens de negotiatiefase (in zijn `ClientHello`). De server is vervolgens verplicht een versleutelset uit die beperkte lijst te kiezen, zoals we eerder zagen.

Dit wordt opgelost als volgt:
de server bezit een speciaal certificaat, dat aangeeft dat het ook export clients met sterke cryptografie mag bedienen. De client zal dan vervolgens reageren op dit speciale certificaat door een nieuwe handshake te starten, waarbij hij ditmaal wel sterke versleutelsets aanbiedt.

Elke export client bevat in praktijk dus alle benodigde functionaliteit om sterke cryptografie te gebruiken. Er werden dan ook al snel *patches* ontwikkeld om dit feit te benutten, en zo van een export client een volwaardige client te maken[10].

[10] Bijvoorbeeld *Fortify*, een patch van 1998 die een exporteerbare Netscape browser in een volwaardige SSL client verandert.

Hoofdstuk 5

Het SSL Protocol aangevallen

"History has taught us:

never underestimate the amount of money, time, and effort

someone will expend to thwart a security system.

It's always better to assume the worst.

Assume your adversaries are better than they are.

Assume science and technology will soon be able to do things they cannot yet.

Give yourself a margin for error.

Give yourself more security than you need today.

When the unexpected happens, you'll be glad you did."
– Bruce Schneier

5.1 Inleiding

In het eerste hoofdstuk werden de basisvereisten voor een veilige connectie besproken, tezamen met een aantal cryptografische begrippen. Vervolgens zagen we in Hoofdstuk 3 en 4 de details van een ingewikkelder cryptosysteem, het SSL protocol, dat deze cryptografische begrippen combineert tot een systeem dat beveiliging op *kanaalniveau* biedt. Dat wil zeggen dat men met behulp van dit cryptografisch systeem een veilige connectie tussen twee partijen kan opzetten en gebruiken, over een onveilig kanaal. Alle data die beide partijen over deze connectie versturen, is gegarandeerd geheim (vertrouwelijkheid) en ongeoorloofde wijzigingen worden gedetecteerd (integriteit). Bovendien wordt de server altijd geauthenticeerd en is de authenticatie van de client optioneel (authenticatie). Daarnaast bezit het SSL protocol ook een mechanisme om speciale gebeurtenissen, zoals fouten en sluiting van de connectie, over diezelfde beveiligde connectie te melden.

In dit hoofdstuk wordt een eerste keer het eigenlijke doel van deze studie aangeraakt, een

onderzoek naar de sterkte van het SSL protocol.

We zullen starten met een kort overzicht van de belangrijkste analyses van het SSL protocol. Sinds het ontstaan van SSL is het protocol talrijke malen diepgaand geanalyseerd, waardoor een aantal zwakheden in het SSL ontwerp naar boven zijn gebracht.

Vervolgens zullen een aantal aanvallen op het SSL protocol voorgesteld worden, met een bespreking van hun praktische haalbaarheid.

Tenslotte wordt een kort overzicht gegeven van de besproken aanvallen, waarna geconcludeerd zal worden dat twee specifieke aanvallen een reëel gevaar lijken te vormen voor het SSL protocol.

5.2 Analyse van het SSL protocol

In deze sectie overlopen we de belangrijkste analyses van het SSL protocol.

Analysis of the SSL 3.0 protocol

David Wagner en Bruce Schneier, twee bekende beveiligingsspecialisten, publiceerden in 1996 een diepgaande technische analyse van het SSLv3 protocol [90] aan de hand van de SSLv3 draft [24]. Zijn concentreren zich vooral op de cryptografische veiligheid, en besluiten dat SSLv3 een aantal zwakheden bevat, maar dat mits een goede implementatie, dit protocol een waardevolle stap voorwaarts is in veilige communicatie over het internet.

A Formal Analysis of the Secure Sockets Layer Protocol

Sven Dietrich, een student aan de Adelphi University of New York, analyseert in zijn doctoraat [17] SSLv3 op formele wijze, door gebruik te maken van NCP (Non-monotonic Cryptographic Protocols) logica. Zijn methode is echter erg gelimiteerd, en laat niet toe om het eerder ingewikkelde gedrag van SSLv3 in alle mogelijke omstandigheden te modelleren. Dietrich controleert SSLv3 enkel op weerstand tegen passieve aanvallen, en besluit dat SSLv3 tegen dit soort aanvallen volkomen immuun is.

Finite-state analysis of SSL 3.0 and related protocols

John Mitchell, Vitali Shmatikov en Ulrich Stern benaderen in hun analyse [49] SSLv3 ook op een formele manier, zij het met een andere methode, door de *finite-state-analysis tool* ***Murphy*** te gebruiken. Zij werken met opeenvolgende versies van een initieel zwak protocol, dat voortdurend verbeterd wordt door de zwakheden die hun formele analyse blootlegt te remediëren.

In tegenstelling tot de analyse van Dietrich, analyseren zij ook de mogelijkheid van actieve aanvallen, en vinden op formele wijze een aantal aanvallen die Wagner en Schneier al eerder

hadden beschreven. Met deze methode werden echter geen nieuwe zwakheden ontdekt.

Ook deze analyse modelleert SSLv3 niet in al zijn complexiteit. Zo gaan de onderzoekers uit van de aanname dat alle gebruikte cryptografische functies per definitie onbreekbaar, dus veilig, zijn. Het spreekt voor zich dat deze assumptie van *perfecte cryptografie* voorbijgaat aan een hele reeks van mogelijke cryptanalytische aanvallen, waaronder een aantal praktisch haalbare. We komen hier later op terug.

Inductive Analysis of the Internet Protocol TLS

Lawrence Paulson beschouwt in zijn analyse van het TLS protocol [56] enkel de handshake fase. Hij gebruikt een inductieve tool, *Isabelle*, voor zijn formele analyse, en vindt geen enkel veiligheidsprobleem. Wel merkt hij op dat de integriteitscontrole over de handshake veel redundantie bevat, en stelt hij voor enkel de kritische componenten te beveiligen. Om die kritische componenten te vinden, stelt hij voor een formele methode zoals de zijne te gebruiken.

Analyzing Internet Security Protocols

Alec Yasinsac en Justin Childs concentreren zich in hun paper [11] ook uitsluitend op de handshake fase van TLS. Ook zij gebruiken een vereenvoudigde voorstelling van TLS voor hun formele analyse, en veronderstellen onder andere perfecte cryptografie. Toch lijken zij alle mogelijke paden, alle mogelijke protocolinteracties, op een uitgebreidere manier te behandelen dan de vorige formele methodes. Zij maken daarvoor gebruik van het *Cryptographic Protocol Analysis Language Evaluation System (CPAL-ES)*, met enige extensies. Op deze manier slagen ze erin om een aanval te ontdekken die Wagner en Schneier bij hun analyse van het SSLv3 protocol al opmerkten, maar die in het nieuwere TLS protocol niet op bevredigende wijze opgelost blijkt te zijn[1].

Ad hoc versus reductionistisch

Er zijn twee manieren om te bewijzen dat een systeem veilig is:

Een *ad hoc* aanpak, waarbij het cryptografische systeem als het ware vanuit de losse pols ontworpen wordt, en waar de ingebruiksneming verschillende zwakheden aan het licht brengt, die vervolgens op een *ad hoc* manier opgelost worden. Deze aanpak kan echter nooit garanderen dat een systeem volledig veilig is, enkel dat er geen zwakheden bekend zijn.

Daarnaast bestaat er de *reductionistische* aanpak, waarbij op een formele manier wordt aangetoond dat een cryptografisch systeem veilig is.

In hoofdstuk 3 zagen we dat SSL duidelijk op een *ad hoc* wijze ontwikkeld is (denk aan het handshake ontwerp van de imaginaire beveiligingsspecialist Joe).

[1] De `ClientKeyExchange` is onvoldoende beschermd, waardoor een aanvaller ervoor kan zorgen dat een DH sleutel verkeerdelijk als een RSA sleutel wordt geïnterpreteerd, of omgekeerd.

Besluit

We besluiten met de observatie dat de gebruikte formele analysemethoden nuttig zijn, maar ontoereikend om een ingewikkeld protocol als SSL volledig te testen, omwille van het grote aantal mogelijke protocolinteracties, waardoor zo'n protocol moeilijk te modelleren is.

Bovendien is geen enkele formele analyse er tot nu toe in geslaagd een andere aanval te ontdekken dan al vermeld in de informele analyse van Wagner en Schneier. Integendeel, de informele analyse blijkt een veel breder scala aan aanvallen te ontdekken dan de formele analyses. Het is dus duidelijk dat informele analyses van gereputeerde beveiligingsexperts momenteel nog onmisbaar zijn.

Toch is een formele, reductionistische analyse de enige methode om aan te tonen dat een cryptografisch systeem *bewijsbaar veilig* is. Zolang geen toereikende formele methode voor SSL ontwikkeld is, blijft het een *ad hoc* systeem, met mogelijk veiligheidslekken.

5.3 Rechtstreekse aanval op SSL

5.3.1 Inleiding

In deze sectie bespreken we de aanvallen uit de analyse van Wagner en Schneier [90]. Aangezien al deze aanvallen een zwakheid in het ontwerp van het protocol zelf proberen uit te buiten, noemen we deze aanvallen rechtstreekse of *main-channel attacks*, in tegenstelling tot de onrechtstreekse aanvallen of *side-channel attacks* die we in de volgende sectie zullen bespreken.

5.3.2 Verkeersanalyse ('traffic analysis')

Verkeersanalyse is een aanval waarbij de aanvaller probeert om vertrouwelijke informatie over een beveiligde connectie in te winnen door de ongeëncrypteerde velden en attributen van een pakket te bestuderen. Zo kan een verkeersanalist door de *IP source* en *destination* adressen te bestuderen en het volume van de netwerktrafiek te meten, te weten komen welke partijen met elkaar communiceren, hoeveel data zij uitwisselen, ... Op deze manier is het mogelijk om vertrouwelijke informatie over professionele of persoonlijke relaties in te winnen.

Bovendien merkt Benneth Yee op dat men door de lengte van SSL ciphertexts te onderzoeken informatie te weten kan komen over de gemaakte URL requests [90]. In het bijzonder kan een aanvaller te weten komen welke internetpagina een client van welke server opvraagt.

De oorzaak van deze kwetsbaarheid ligt in het feit dat de lengte van de plaintext in SSL gelijk is aan de lengte van de ciphertext, althans bij het gebruik van stream ciphers. Block ciphers maken gebruik van padding, en vertonen dit lek niet. Wagner en Schneier stellen dan ook voor om tenminste een optionele ondersteuning met random padding voor alle ciphers in te voeren [90].

Merk op dat deze aanval een *passieve aanval* is.

5.3.3 Replay attack

Eerder zagen we al dat SSL een aantal mechanismen inbouwt om replay attacks te stoppen. Het belangrijkste van deze mechanismen is het volgnummer dat gebruikt wordt in de berekening van de MAC's over de SSL Data Records. Dit mechanisme voorkomt replay attacks in zijn breedste betekenis, dus niet alleen het eenvoudigweg herhalen van bepaalde berichten, maar ook het herordenen, het uitstellen of het verwijderen van bepaalde berichten.

Merk op dat dit mechanisme enkel geldt voor geëncrypteerde data, de handshake berichten die onversleuteld over het kanaal verstuurd worden zijn niet beveiligd met deze methode. Dit is ook niet nodig, aangezien handshake berichten intrinsiek al zo'n sterke volgorde vereisen, dat het herordenen, uitstellen of verwijderen van bepaalde berichten de handshake zo sterk verstoort, dat de connectie onmiddellijk afgebroken zal worden. Daarnaast wordt het herhalen van handshake berichten in opeenvolgende sessies nutteloos gemaakt door het gebruik van de twee randomwaarden in de `Hello`-berichten, die bij elke sessie vernieuwd worden.

5.3.4 Versleutelsetverlaging ('cipher suite rollback attack')

Aanval

Een belangrijk lek in het SSLv2 protocol bestaat uit het feit dat de integriteit van de handshake niet bewaakt wordt. Hierdoor kan een aanvaller de handshake ongemerkt manipuleren, en de lijst van aangeboden versleutelsets in de `ClientHello` ongemerkt aanpassen. Zo kan een aanvaller alle sterke versleutelsets uit de lijst verwijderen, en de lijst tot een aantal (of zelfs één) zwakke versleutelsets beperken. Indien de server dan zo'n zwakke versleutelset ondersteunt, zal hij die accepteren, en zal de connectie verder beveiligd worden met de zwakke cryptografische algoritmen uit die versleutelset. Deze aanval noemt men *versleutelsetverlaging*.

In praktijk zijn de zwakste versleutelsets diegene die voldoen aan de oude export regulering, de export versleutelsets. Om deze reden wordt deze aanval ook wel eens *downgrade-to-export attack* genoemd.

Toepasbaarheid

Vanaf SSLv3 heeft SSL wel een integriteitscontrole over de handshake, waardoor deze aanval een stuk moeilijker wordt. Deze integriteitscontrole hangt echter samen met de versleutelset die overeengekomen wordt, en een zwakke versleutelset betekent automatisch een zwakkere integriteitscontrole.

In praktijk blijkt SSL daarom niet sterker dan de *zwakste gemeenschappelijke versleutelset*

tussen beide partijen. Dit is duidelijk niet de bedoeling van de dynamische versleutelset-negotiatie in SSL, en een belangrijke zwakheid, waarop we in het volgende hoofdstuk uitgebreid terugkomen.

5.3.5 Versieverlaging ('version rollback attack')

Aanval

We zagen eerder al dat SSL *backward compatible* is met eerdere versies. Dit wil zeggen dat SSLv2 connecties geaccepteerd worden, zelfs door SSLv3 of TLS versies. Bovendien zagen we dat SSLv2 een aantal belangrijke zwakheden bezit, die juist de reden vormen voor het feit dat SSLv3 ontworpen werd.

Om deze reden is er een groot risico voor *versieverlaging*, een aanval waarin de aanvaller de `ClientHello` wijzigt om hem te laten lijken op een SSLv2 `ClientHello`, waardoor de volledige SSL connectie volgens het SSLv2 protocol verloopt, zelfs indien beide partijen hogere versies ondersteunen.

Merk op dat een aanvaller de `ClientHello` op deze manier *ongemerkt* kan wijzigen, aangezien SSLv2 *geen* integriteitscontrole uitvoert op de handshake.

Tegenmaatregel

Paul Kocher, één van de SSL designers, bedacht echter een opmerkelijke strategie om dit soort aanval tegen te gaan. Clients die SSLv3 ondersteunen, gebruiken een vast patroon[2] (waar dit patroon normaal random is) voor de padding bytes in de PKCS formattering bij de RSA-encryptie van de `pre_master_secret`[3] [24]. Aangezien deze padding bytes (samen met de `pre_master_secret` zelf) geëncrypteerd worden met de publieke RSA sleutel (in het `ClientKeyExchange` bericht[4]), kan een aanvaller dit patroon niet wijzigen.

Deze oplossing zorgt ervoor dat twee SSLv3 compatibele partijen nooit een SSLv2 connectie zullen toelaten, aangezien de RSA-versleutelde sleuteluitwisseling tussen deze partijen altijd dit speciale patroon zal vertonen, waarop de server een `alert` zal genereren en de connectie afsluiten. Daarnaast kan een SSLv2 client zonder problemen met een SSLv3 server (of vice versa) een SSLv2 connectie opzetten.

Een belangrijke assumptie waarop deze oplossing steunt, is de veronderstelling dat SSLv2 enkel RSA-sleuteluitwisseling ondersteunt. Niet-RSA-sleuteluitwisselingen bevatten dit speciaal padding patroon niet, en kunnen een versieverlaging dus niet detecteren.

SSLv3 implementaties mogen SSLv2 `ClientHello`'s die een niet-RSA sleuteluitwisselingsalgoritme proberen te negotiëren dus in geen geval aanvaarden. Deze eis wordt niet

[2] De 8 minst significante random bytes worden vervangen door "0x03"

[3] In SSLv2 is dit de *master key*

[4] In SSLv2 heet dit bericht eigenlijk `Client-Master-Key`, met een publiek geëncrypteerd veld genaamd `ENCRYPTED-KEY-DATA`, maar de functie is vergelijkbaar met die van het bekende SSLv3 `ClientKeyExchange` bericht

expliciet in de originele SSLv3 specificatie vermeld, niettemin zijn alle belangrijke SSL implementaties aan deze eis aangepast.

We merken nog op dat er in de oorspronkelijke SSLv3 specificatie al een beveiliging zit tegen dit soort versieverlagingsaanvallen. Die specificatie stelt immers dat de client voor de eerste 2 bytes van de `pre_master_secret` het versienummer moet gebruiken dat hij eerder aanbood in zijn `ClientHello`. In praktijk blijkt echter dat vele SSL implementaties in plaats van het *aangeboden* versienummer hier het *genegotieerde* versienummer gebruiken, waardoor deze controle tot interoperabiliteitsproblemen leidt, en in praktijk niet wordt uitgevoerd [60, p.79].

5.3.6 Principe van Horton

Het *principe van Horton* is niet zelf een aanval, maar een principe dat, mits consequent gehanteerd, bepaalde aanvallen voorkomt.

Dit principe stelt het volgende [90]:

> *"Authenticeer wat bedoeld is, niet wat gezegd is."*

Met andere woorden, authenticeer niet alleen de data zelf, maar ook de context waarin de data geïnterpreteerd wordt.

Binnen SSL wordt deze authenticatie van de data (en context) bewerkstelligd met behulp van MAC's, die tegelijkertijd ook de integriteit garanderen.

SSLv2 gaat echter in tegen dit principe: het gebruikt wel de padding data van een bericht om de MAC over dit bericht te berekenen, maar niet de lengte van de padding. Een aanvaller kan dus de integriteit van zo'n bericht compromitteren door de lengte van de padding ongemerkt te manipuleren.

In SSLv3 wordt dit lek opgelost, en een informele hortoniaanse analyse van Wagner en Schneier [90] toont aan dat de ontwerpers van SSLv3 dit principe gerespecteerd lijken te hebben in de beveiliging van het SSL Record Protocol.

Bij de beveiliging van het SSLv3 Handshake Protocol blijken de ontwerpers echter even dit principe uit het oog te hebben verloren. Aangezien de SSL handshake duidelijk een context vormt voor de beveiliging van de data, is het vanuit dit principe een *must* om ook de handshake met een MAC te beschermen. We zagen al eerder dat dit volkomen nagelaten werd in SSLv2, de handshake is daar compleet onbeschermd. SSLv3 biedt wel bescherming voor zijn handshake, onder de vorm van een MAC, maar het `ChangeCipherSpec` bericht wordt niet in de berekening van deze MAC betrokken. Dit blijkt een zwakheid met zich mee te brengen die wordt uitgebuit in de zogenaamde *ChangeCipherSpec aanval*, die hierna wordt besproken.

ChangeCipherSpec aanval

Het `ChangeCipherSpec` bericht is het teken dat een partij overschakelt naar een connectie beveiligd met de overeengekomen versleutelset. Zoals we eerder al opmerkten, wordt het `ChangeCipherSpec` bericht niet opgenomen in de integriteitscontrole over de handshake (die gevormd wordt door de `Finished` berichten). Volgens de SSLv3 specificatie komt dit de performantie ten goede [24]. Recente implementaties blijken echter geen enkele performantiewinst uit dit *feature* te halen [60, p.80].

Deze vreemde aanpak is echter niet enkel overbodig, maar blijkt bovendien tot een aantal beveiligingsproblemen te leiden.

Een aanvaller kan dit bericht immers verwijderen uit de connectie. Dit heeft het gevolg dat de partijen overschakelen op de nieuwe versleutelset, zonder dat de andere partij daarop voorbereid is. Dit kan twee gevolgen hebben.

1. Indien een normale versleutelset genegotieerd werd, en de data geëncrypteerd wordt verstuurd, zullen beide partijen de versleutelde data die zij toegestuurd krijgen niet kunnen interpreteren, deze data als foutief beschouwen, en de connectie sluiten.
2. Stel, server en client negotiëren (al dan niet gedwongen) een versleutelset die uitsluitend authenticatie en integriteit garandeert, maar geen encryptie (vertrouwelijkheid) ondersteunt[5]. Onder deze versleutelset zullen server en client enkel een MAC aan hun berichten toevoegen, maar het geheel niet encrypteren. Indien een aanvaller de `ChangeCipherSpec` berichten uit zo'n connectie wegneemt, zullen beide partijen de data wel *versturen* onder de *nieuwe* versleutelset, maar data die ze *ontvangen* nog altijd behandelen onder de *initiële* `NULL`-versleutelset. Zonder extra ingreep van de aanvaller zal ook hier de ontvangen data als foutief beschouwd worden, en de connectie afgebroken worden. Een aanvaller kan echter de integriteitscontrole (MAC) van alle verstuurde "beveiligde" pakketten strippen, en er op deze manier voor zorgen dat beide partijen de pakketten die ze ontvangen toch kunnen interpreteren. In dit geval wordt alle data dus onversleuteld verstuurd, en kan de aanvaller bovendien ongemerkt de inhoud van een willekeurig pakket aanpassen, zonder dat één van beide partijen iets ongewoons opmerkt.

Een eenvoudige oplossing om beide gevolgen te voorkomen is het eisen van een `ChangeCipherSpec` bericht vóór wordt overgegaan op het zenden van `Finished` berichten, of erger nog, `application_data`. De TLS specificatie vermeldt dit gedrag dan ook als een *must* [16].

Bovendien is het laatste gevolg zelfs zonder deze eis erg onwaarschijnlijk, aangezien versleutelsets die enkel authenticatie zonder encryptie toepassen door bijna geen enkele server of client implementatie ondersteund (laat staan standaard geactiveerd) worden, omwille van het ontbreken van de vertrouwelijkheidseigenschap.

5.3.7 Overzicht van rechtstreekse aanvallen op SSL

We besluiten met een overzicht van alle besproken aanvallen, waarbij voor elke aanval wordt aangegeven in hoeverre de veiligheid van een SSL connectie gecompromitteerd wordt,

[5] Versleutelset TLS_RSA_WITH_NULL_MD5 of TLS_RSA_WITH_NULL_SHA

en aan welke eisen voldaan moet zijn om de aanval te lanceren (Tabel 5.1).

Merk op dat we de replay attack hier niet vermelden, aangezien die binnen SSL onmogelijk is.

Aanval	**Gevaar** (Wat wordt gecompromitteerd?)	**Voorwaarden**
Verkeersanalyse	randinformatie over connectie	geen (stream cipher-gebruik verschaft extra info)
Versleutelsetverlaging	volledige connectie	zwakste (a)symmetrische algoritme door beide partijen ondersteund te breken
Versieverlaging	volledige connectie	asymmetrisch algoritme te breken
ChangeCipherSpec aanval type 1	einde van connectie	geen
ChangeCipherSpec aanval type 2	volledige connectie	versleutelset zonder encryptie gebruikt

Tabel 5.1. Rechtstreekse aanvallen op het SSL Protocol

5.4 Onrechtstreekse aanval op SSL

5.4.1 Inleiding

Een onrechtstreekse aanval of *side-channel attack* is een aanval die het SSL protocol niet op een directe manier aanvalt, maar die een aanval uitvoert op een cryptografische primitieve die in SSL gebruikt wordt, en waarover SSL op de een of andere manier extra informatie lekt, waardoor deze cryptografische primitieve kwetsbaar wordt voor een aanval. Zo'n cryptografische primitieve vertoont dus een zwakheid doordat het op een naïve manier geïmplementeerd wordt in een groter, ingewikkelder cryptosysteem. Dit soort aanval noemt men daarom ook wel eens een *implementatie-aanval.*

Hierbij merken we op dat wij enkel side-channel attacks op software niveau beschouwen. Daarnaast bestaan er ook aanvallen die we *hardware side-channel attacks* zullen noemen, en die steunen op informatie die lekt uit de hardware implementatie die gebruikt wordt, via bijvoorbeeld elektromagnetische straling, stroomverbruik of akoestische signalen. Dit soort informatie is echter niet in te winnen over het netwerk zelf, daarom beschouwen we deze categorie van aanvallen *niet* binnen onze studie, die zich richt op netwerkbeveiliging binnen het kader van het Internet Threat Model.

Tot nu toe zijn er drie categorieën van side-channel attacks op SSL bekend, ingedeeld naar de cryptografische primitieve die de side-channel attack mogelijk maakt:

- **RSA**
  Hier wordt het asymmetrische encryptie-mechanisme binnen SSL aangevallen, dat typisch uit RSA bestaat.
- **CBC**
  Een aanval op de symmetrische encryptie-algoritmen van SSL die functioneren in CBC mode.
- **Certificaat**
  Deze aanval zal een SSL connectie trachten te compromitteren door het certificaat-mechanisme op de één of andere manier te bedriegen.

In de volgende secties worden de bekende aanvallen binnen deze categoriën één voor één toegelicht.

5.5 RSA implementatie aangevallen

5.5.1 Inleiding

Zoals we zagen in Sectie 2.6.1, is RSA het meestgebruikte cryptografische systeem van de publieke sleutel-cryptografie. Het wordt gebruikt voor encryptie en digitale handtekeningen.

Het is dus niet verwonderlijk dat de *cryptanalyse* zo'n grote belangstelling heeft voor dit systeem. Een overzicht van de belangrijkste aanvallen sinds het ontstaan van dit systeem kan gevonden worden in [5]. Geen van deze aanvallen werkt met een verwoestend resultaat. Alle zijn zij slechts mogelijk als gevolg van een verkeerd gebruik van RSA, of omwille van een slechte implementatie.

We bespreken in de volgende secties enkele aanvallen die het gevolg zijn van een ondoordachte implementatie van RSA in het SSL protocol, zogenaamde *RSA side-channel attacks*.

5.5.2 Timing aanval

Timing aanvallen tegen RSA werden in 1996 geïntroduceerd door Paul Kocher. Deze cryptanalytische aanvalstechniek steunt op het feit dat de tijd die een cryptografische operatie inneemt, afhangt van de sleutellengte en de gebruikte data (en de soort cryptografische operatie zelf). Kocher beschrijft een aantal van zulke aanvallen op asymmetrische encryptiesystemen zoals RSA, DH en DSS [39]. Er werd echter aangenomen dat zulke timing aanvallen enkel mogelijk zijn indien de aanvaller de cryptografische module in zijn bezit heeft. Dit soort aanvallen werden dus niet als een realistische bedreiging voor SSL gezien,

aangezien onder het Internet Threat Model de eindsystemen als volstrekt veilig beschouwd worden.

David Brumley en Dan Boneh bewijzen in 2003 in hun paper [7] echter dat zo'n aanval ook mogelijk is van op afstand, en demonstreren een RSA timing aanval op OpenSSL. Zij tonen aan dat het voor een aanvaller in het locale netwerk mogelijk is om de RSA private sleutel van een OpenSSL server te extraheren.

Aanval

De aanval bestaat uit het compromitteren van de private sleutel van een RSA sleutelpaar, door het factorizeren van de modulus `n = pq`, met `q < p`. De aanvaller probeert q te schatten met een waarde g, waarbij hij het feit gebruikt dat in het geval `g < q` de decryptie van g beduidend sneller blijkt dan in het geval `g > q`. Deze informatie kan gebruikt worden in een timing attack op SSL, als volgt:

Tijdens de handshake wisselen client en server in het `ClientKeyExchange` bericht een geëncrypteerde `pre_master_secret` uit, die aan de serverzijde gedecrypteerd wordt met de private RSA sleutel[6]. In plaats van een versleutelde `pre_master_secret` zal de aanvaller echter de benadering g versturen. De server zal vervolgens deze g proberen te decrypteren, als was het een versleutelde `pre_master_secret`. De server zal ontdekken dat g een ongeldige `pre_master_secret` voorstelt, en een foutmelding versturen. De aanvaller zal het tijdsverschil tussen het versturen van de `ClientKeyExchange` en het ontvangen van die foutmelding meten, en daaruit zal hij informatie afleiden om een betere benadering g te construeren. Dit proces herhaalt hij tot een g gevonden wordt die gelijk is aan q. Met die q kan dan de modulus n berekend worden, waaruit vervolgens de private sleutel afgeleid kan worden.

Toepasbaarheid

Deze aanval wordt praktisch gedemonstreerd door Brumley en Boneh, die aantonen dat een 1024-bit RSA sleutel in ongeveer een miljoen berichten, of 2 uur tijd op een normaal desktop systeem[7], gebroken kan worden. Alle OpenSSL versies tot en met versie 0.9.7a blijken kwetsbaar voor deze aanval.

Tegenmaatregelen

Er zijn drie mogelijke tegenmaatregelen tegen dit soort aanvallen:

- **RSA blinding**
  Gebruik een random waarde om de timing onvoorspelbaar te maken.

[6] We stellen dat RSA overeengekomen werd als sleuteluitwisselingsalgoritme.

[7] In 2003, een huidig 'normaal desktop systeem' kan dit natuurlijk sneller.

De server berekent eerst `X = R`^{e}`C mod n`, met R een random waarde, (e,n) de publieke RSA sleutel en C de ciphertext. Vervolgens decrypteert hij X op de normale manier met zijn private sleutel, waarna hij deze uitkomst deelt door R, met andere woorden, `B =` $\frac{X^e}{R}$ `mod n`. B is dan de plaintext van C en de tijd die decryptie-operatie innam, is afhankelijk van de randomwaarde R en bijgevolg onvoorspelbaar.

- **Ciphertext onafhankelijke decryptie**
  Maak RSA-decryptie onafhankelijk van de input ciphertext door een optimalisatie in de berekeningen weg te halen. Deze aanpak wordt voorgesteld door Werner Schindler [71].

- **Kwantificatie**
  Quantizeer alle RSA berekeningen, met andere woorden, zorg ervoor dat de tijd die een RSA berekening inneemt altijd een veelvoud van een bepaald, pre-gedefinieerd tijdsinterval is.

In praktijk wordt RSA blinding gebruikt, deze techniek wordt standaard toegepast in OpenSSL sinds versie 0.9.7b. Versies vóór deze implementatie zijn kwetsbaar voor de beschreven timing aanval.

Merk op dat al deze technieken voor performantieverlies zorgen. Het gebruik van RSA blinding zorgt voor een performantieverlies van 2 tot 10%, afhankelijk van de implementatie [65].

5.5.3 Bleichenbacher aanval

In 1998 publiceerde Daniel Bleichenbacher een aanval die bekend staat als de *million message attack* of *bleichenbacher aanval* [4]. Deze aanval is geen rechtstreeks aanval op RSA zelf, maar een aanval op het *padding schema* PKCS#1 (versie1.5) [64] waarvan het gebruik maakt. We zouden dit dus als een side-channel attack op RSA kunnen zien, die op zijn beurt een side-channel attack op SSL toelaat.

De aanval zelf bestaat uit het versturen van een massa ("*million*") berichten naar de server, die zorgvuldig opgesteld zijn. Men noemt dit een *chosen-ciphertext attack*, de aanvaller bepaalt zelf de ciphertexts die de server zal decrypteren. Uit de reactie van de server kan de aanvaller vervolgens informatie afleiden, waarmee hij de `pre_master_secret` kan recupereren. De server functioneert hier dus als *orakel*, een medium waarlangs de aanvaller onrechtstreeks het antwoord op zijn vraag ("Wat is de `pre_master_secret`?") te weten komt.

Aanval

De basis van deze aanval is het orakel. Een bleichenbacher aanval gaat ervan uit dat de aanvaller toegang heeft tot een orakel dat hem voor elke gekozen ciphertext C, geëncrypteerd met een RSA publieke sleutel *(e, n)* en met bijhorende private exponent d, kan vertellen of `C`^{d} `mod n` van het PKCS#1 formaat is, of niet. Door telkens het antwoord van het orakel te gebruiken om een nieuwe ciphertext C te genereren, en deze nieuwe ciphertext

door het orakel te laten beoordelen, wint de aanvaller meer en meer informatie in over C^d (de gedecrypteerde C). Zo'n aanval waarin de aanvaller de opeenvolgende ciphertexts zelf kiest, op basis van informatie uit de vorige aanvallen, noemt men een *adaptive chosen-ciphertext* aanval.

Een SSL server blijkt een perfect orakel voor deze aanval, aangezien het SSL protocol een alert genereert indien een RSA-geëncrypteerd blok niet voldoet aan de PKCS#1 formattering.

Een aanvaller die de connectie tussen server en client wil compromitteren, zal de gebruikte `pre_master_secret` proberen te weten te komen. Dit doet hij door een nieuwe ciphertext C_{new} te produceren, op de volgende wijze:

> Gegeven C_{PMS} de geëncrypteerde pre_master_secret, S een random integer, e de RSA publieke exponent en n de RSA modulus:
>
> $C_{new} = S^e C_{PMS} \bmod n$

Aan de hand van de antwoorden van het orakel genereert hij telkens nieuwe ciphertexts C_{new} (door de integers S op een doordachte manier te variëren), en op deze manier kan een aanvaller meer en meer informatie over de decryptie van C_{PMS} te weten komen, tot op het punt waarop de volledige `pre_master_secret` bekend is.

Een complete, wiskundige uitleg kan gevonden worden in de paper [4] zelf, maar in praktijk blijkt men ongeveer 2^{20} berichten[8] nodig te hebben bij een 1024-bit RSA sleutel.

Merk op dat de aanvaller voor elke nieuwe orakeltest een nieuwe connectie moet starten, aangezien de server de connectie na elke vraag beëindigt. Bovendien kan de server bij deze nieuwe connectie geen gebruik maken van sessieherneming, omdat de volledige handshake doorlopen moet worden[9]. Ofwel gebeurt het beëindigen van de connectie onmiddellijk na de foutmelding dat het bericht niet PKCS#1 conform is, ofwel bij het genereren van de `Finished` berichten. Het opstellen van deze laatste veronderstelt immers kennis van de `pre_master_secret`, kennis die de aanvaller niet heeft, maar juist te weten probeert te komen.

Aangezien de connectie telkens pas wordt afgebroken *nadat* het orakel (de server) een antwoord heeft verschaft, is dit echter geen enkel probleem. Het vereist enkel dat er voor elke nieuwe test een nieuwe connectie (en nieuwe sessie) gemaakt wordt. Orakels die zichzelf onmiddellijk vernietigen na het geven van één antwoord, worden *bomb oracles* genoemd [87].

Toepasbaarheid

De aanval steunt geheel op de goede werking van het orakel. De aanvaller moet het onderscheid kunnen maken tussen enerzijds een PKCS#1 geformatteerd blok, en anderzijds een blok dat daar niet aan voldoet.

[8] Dit aantal is ongeveer een miljoen, vandaar *million message attack*.

[9] Daarenboven wordt sessieherneming niet toegelaten bij een sessie die eerder beëindigd werd met een foutmelding

De eerste SSL implementaties stuurden een specifieke alert in verband met PKCS#1 formattering, en waren dus een makkelijk slachtoffer.

De `pre_master_secret` kan echter, naast zijn PKCS#1 formattering, op nog 2 andere elementen geverifieerd worden:

- Lengte van 48 bytes
- Eerste twee bytes vormen het versienummer

Een implementatie die de `pre_master_secret` ook controleert op deze bijkomende eisen, en één algemene foutmelding gebruikt om het niet voldoen *aan eender welke* eis te melden, zorgt voor extra "ruis" op het orakel. Deze ruis verslechtert de werking van deze aanval in zo'n grote mate, dat deze aanval onuitvoerbaar wordt in praktijk.

Extensies

Een oplossing die werd voorgesteld om deze aanval te voorkomen, is het gebruik van een ander padding schema, zoals PKCS#1 versie 2.x [66].

Een onvoorzichtige implementatie van deze nieuwe versie blijkt de dingen echter nog erger te maken, zoals James Manger aantoont. In zijn paper [46] demonstreert hij een aangepaste versie van de bleichenbacher aanval, die slechts een duizendtal berichten nodig heeft om eenzelfde effect te bereiken bij een 1024-bit RSA sleutel, gebruikmakende van deze nieuwe formattering.

Als reactie op de *manger aanval* werd nog een ander padding schema voorgesteld, RSA-KEM [75]. Vlastimil Klima en Tomas Rosa tonen echter in 2002 aan dat dit de zaken niet verbetert, zelfs integendeel [38]. Zij demonstreren een variant van de bleichenbacher aanval die toelaat om de RSA private sleutel van de server te weten te komen. Bovendien tonen zij in diezelfde paper aan dat het orakel dat Manger (versie 2.x) en Bleichenbacher (versie 1.5) gebruiken, ook kan gebruikt worden om digitale handtekeningen te vervalsen, indien hetzelfde RSA sleutelpaar gebruikt wordt voor zowel encryptie als digitale handtekeningen. In het SSL protocol is dit een reëel gevaar bij het gebruik van Ephemeral RSA, waar de private sleutel gebruikt wordt om een digitale handtekening te generen over het bericht dat de nieuwe, tijdelijke RSA sleutel bevat.

Klima en Rosa presenteerden wat later nog een andere extensie van de bleichenbacher aanval [37], één die zich baseert op een slecht geïmplementeerde tegenmaatregel tegen de originele bleichenbacher aanval. Die implementaties genereren een speciale foutmelding indien de versie in de `pre_master_secret` foutief is, hoewel de eerder voorgestelde tegenmaatregel duidelijk stelt dat een *uniforme* foutmelding voor *elk* van de `pre_master_secret` gerelateerde verificaties nodig is.

Tegenmaatregelen

De eenvoudigste tegenmaatregel is het gebruik van een algemene foutmelding, zoals juist besproken. Dit is dan ook de manier waarop recente implementaties tegen dit soort aanvallen beveiligd zijn.

Een andere oplossing, het gebruik van een ander padding schema, bleek geen oplossing te bieden aangezien dit bleek te leiden tot nieuwe, aangepaste extensies van de bleichenbacher aanval. Het SSL protocol gebruikt dan ook nog altijd het 'oude' PKCS#1 versie 1.5 padding schema.

5.6 CBC implementatie aangevallen

5.6.1 Inleiding

Naast aanvallen op de RSA-implementatie in SSL, bestaan er ook *side-channel attacks* die zich baseren op Cipher Block Chaining (CBC). Dit soort aanvallen zal gebruik maken van een gebrekkige implementatie van CBC mode in SSL, om een aanval op CBC block ciphers te lanceren. We stellen zo twee aanvallen voor, waarvan de tweede eigenlijk een bijzonder geval is van de eerste.

5.6.2 CBC padding aanval: algemeen

Serge Vaudenay stelt in 2002 zo'n aanval voor, die hij omschrijft als "*een bleichenbacher aanval binnen de symmetrische sleutel wereld*"[87]. Daar waar de bleichenbacher aanval en afgeleiden werken op publieke sleutel-algoritmen, werkt Vaudenay's aanval op geheime sleutel-algoritmen.

Aanval

De aanval van Vaudenay maakt gebruik van een orakel dat informatie over padding verificatie lekt, waardoor een willekeurige ciphertext gedecrypteerd kan worden. De aanval wordt in detail besproken in [87], maar verloopt in grote lijnen als volgt:

> Een aanvaller construeert een bericht, waarvan het laatste deel bestaat uit een ciphertext die hij gedecrypteerd wil zien. Het orakel zal dit bericht decrypteren, en padding in het laatste deel van het gedecrypteerde bericht verwachten. Het orakel zal deze padding vervolgens controleren en de aanvaller op de één of andere manier laten weten of de padding al dan niet geldig is. Indien de padding geldig is, kan de aanvaller hieruit informatie over de laatste byte van de ciphertext afleiden. Indien niet, dan probeert de aanvaller opnieuw met een nieuw, aangepast bericht. Door dit proces te herhalen, kan de aanvaller de plaintext van de ciphertext byte voor byte te weten komen.

> Vaudenay toont aan dat voor een bloklengte van 8 bytes, met voor elke byte 256 mogelijkheden, deze aanval een ciphertext bestaande uit `N` blokken kan decrypteren in gemiddeld `1024N` orakelvragen.

Op het eerste zicht is SSL een aantrekkelijk doelwit voor deze aanval, aangezien een SSL server of client perfect aan de voorwaarden voor zo'n orakel voldoet, zoals uit het volgende duidelijk wordt:

> SSL zal de boodschap eerst decrypteren, en daarna de padding controleren. Indien de padding niet in orde is, wordt een foutmelding (fatal alert: `decryption_failed`) gestuurd en de connectie beëindigd. Indien de padding wel in orde is, wordt overgegaan tot verificatie van de MAC (die altijd zal mislukken, aangezien de aanvaller een gemanipuleerd bericht stuurt), waarna eveneens een foutmelding (fatal alert: `bad_record_mac`) volgt. Een SSL server of client lijkt dus een perfect orakel.

Toepasbaarheid

Zo'n SSL orakel heeft echter een lastige eigenschap: het is een *bomb oracle*. Het kan per symmetrische sleutel slechts éénmaal gebruikt worden, aangezien de SSL server de connectie onherroepelijk afsluit na elke fatale foutmelding[10]. Bij een bleichenbacher aanval, die ook gebruik maakt van zo'n *bomb oracle*, is dit geen probleem, aangezien bij elke connectie steeds hetzelfde asymmetrische sleutelpaar wordt gebruikt. Symmetrische sleutels worden echter bij elke connectie opnieuw gegenereerd.

Zo'n eenmalig orakelantwoord blijkt amper voldoende om de laatste byte van het geëncrypteerde bericht te ontcijferen. Bovendien zijn deze foutmeldingen zelf ook geëncrypteerd, en is het daarom moeilijk zo'n foutmelding te onderscheiden. Daarnaast wordt in de recentste SSL implementaties eenzelfde foutmelding gegenereerd voor de verschillende types fouten (padding fout of MAC fout)[11], wat de aanval nog eens extra bemoeilijkt. Dit alles deed Vaudenay besluiten dat zijn aanval niet realiseerbaar was op het SSL protocol.

5.6.3 CBC padding aanval: een specifieke aanval

In een volgende paper stelt Vaudenay, samen met Brice Canvel, Alain Hiltgen en Martin Vuagnoux een variant op deze aanval voor, die wel praktisch uitvoerbaar is op het SSL protocol [9]. Zij demonstreren hoe het paswoord van een IMAP account op een mailserver onderschept kan worden, waarbij de connectie tussen de mailserver en de mailclient over een TLS tunnel verloopt.

[10] Bovendien is omwille van de foutmelding sessieherneming onmogelijk

[11] Volledigheidshalve moeten we opmerken dat SSLv3 nooit zo'n onderscheid gemaakt heeft, maar dat zo'n onderscheid wel in de latere TLS versie werd ingevoerd. Dit onderscheid wordt, als gevolg van het onderzoek van Vaudenay, echter niet meer gemaakt in recente TLS implementaties.

Aanval

Het probleem dat zich stelt doordat TLS geen onderscheid maakt tussen de verschillend fouttypes wordt opgelost door een soort timing aanval te gebruiken. Aangezien bij slechte padding onmiddellijk een foutmelding wordt gestuurd, en de MAC niet meer geverifieerd wordt, zal de foutmelding in dit geval sneller gestuurd worden dan in het geval dat de padding juist is, en de MAC wel geverifieerd wordt. Dit soort timing aanval is niet te verwarren met de eerder geziene timing aanvallen waarbij de tijd van de cryptografische operaties zelf gemeten wordt, niet de tijd om de foutmeldingen te genereren.

Het feit dat een connectie (en bijhorende sessie) telkens onmiddellijk beëindigd wordt, vormt in dit geval geen probleem aangezien hetzelfde paswoord telkens opnieuw, in elke nieuwe connectie, weer verstuurd wordt, op een voorspelbare plaats.

Toepasbaarheid

De aanval is erg gerestricteerd, en werkt slechts onder de volgende drie voorwaarden:

1. Een *kritisch* stukje *informatie* (bijvoorbeeld een paswoord) is herhaaldelijk geëncrypteerd op een *voorspelbare plaats.*
2. Er wordt een block cipher in *CBC mode* gebruikt.
3. De aanvaller kan de *tijdsverschillen* tussen de twee types foutmeldingen *meten*, om de *foutmeldingen* van elkaar te *onderscheiden*

Vooral de eerste voorwaarde restricteert het gebruik van deze aanval in grote mate, en maakt de aanval zo goed als onbruikbaar in andere gevallen dan dit heel specifieke mailserver-voorbeeld. Ook de tweede voorwaarde is redelijk sterk, de meeste SSL connecties maken gebruik van RC4, een stream cipher, waarbij dus geen CBC gebruikt wordt.

Tegenmaatregelen

Om deze aanval te verhinderen, is het voldoende om te zorgen dat aan minstens één van de voorgaande voorwaarden onmogelijk voldaan kan worden. Zo verifieert OpenSSL sinds versie 0.9.7a altijd de MAC, zelfs indien de padding-controle mislukt is. Dit voorkomt duidelijk de aanval.

5.7 Certificaat implementatie aangevallen

5.7.1 Inleiding

De vorige implementatie-aanvallen trachten de vertrouwelijkheid van een SSL connectie op de één of andere manier te compromitteren. Certificaat *side-channel attacks* zullen

echter ingrijpen op de authenticiteitseigenschap van SSL connecties. Dit betekent dat een aanvaller zich ten opzichte van één van beide (of beide) partijen zal voordoen als de andere partij. Dit soort aanvallen, waarbij de aanvaller zich voordoet als iemand anders, noemt men *spoofing attacks*.

Aangezien bij SSL connecties in de regel enkel de server geauthenticeerd wordt, zal dit betekenen dat een aanvaller zich naar de client toe voordoet als server. In de volgende secties wordt besproken op welke manieren zo'n *spoofing attack* mogelijk is, waarna we een concreet voorbeeld van zo'n aanval geven.

Een aanvaller die zelf één van de beide partijen in het protocol vormt, en waarbij er dus geen derde partij aanwezig is, noemt men een *bedrieger* ('*cheater*'). *Passieve bedriegers* volgen mooi het protocol maar proberen meer informatie te weten te komen dan de bedoeling is van het protocol. *Actieve bedriegers* manipuleren actief het protocol [72, p.27].

5.7.2 Authenticiteit in SSL

Eerder zagen we dat de authenticiteit van een server in SSL gegarandeerd wordt met behulp van een certificaatmechanisme (zie ook 2.11). De client krijgt van de server een certificaat toegestuurd, waarvan de client de geldigheid controleert. Indien het certificaat geldig wordt bevonden, wordt de publieke sleutel uit het certificaat gebruikt om de `pre_master_secret` door te sturen, waarna beide partijen deze gebruiken om een veilige connectie op te zetten.

Het schoentje blijkt te wringen in die *"client controleert geldigheid"*-fase.

In Sectie 2.11 zagen we drie controles die moesten uitgevoerd worden om te beslissen of een certificaat al dan niet geldig is:

- identiteitsgegevens en publieke sleutel in het certificaat komen overeen met die van server X
- vervaldatum van het certificaat is nog niet overschreden
- certificaat is gecertificeerd (gehandtekend) door een bevoegde CA
- certificaat is niet gerevoceerd

In praktijk worden deze controles uitgevoerd door de SSL implementatie van de client, maar het is altijd de verantwoordelijkheid van de menselijke gebruiker om over het al dan niet aangaan van de connectie te beslissen. Er wordt echter aangenomen[12] dat quasi niemand kiest voor het ongemak dat het weigeren van zo'n certificaat met zich meebrengt (namelijk het niet kunnen communiceren met de server waarmee gecommuniceerd wil worden), en de meeste gebruikers vertrouwen erop dat SSL desondanks toch zijn werk doet: de connectie beveiligen.

Hierbij merken we nog op dat het op het eerste zicht een beetje vreemd lijkt dat we aanvallen die op deze fouten inspelen toch implementatie-aanvallen op SSL noemen. Het gaat hier immers niet zozeer om implementatiefouten van het SSL Protocol zelf, maar

[12]Hoewel geen enkele studie bekend is waarin dit zwart op wit wordt aangetoond.

om algemene implementatieproblemen van het certificaatmechanisme met behulp van een *Public Key Infrastructure (PKI)*. Geen enkele implementatie die gebruik maakt van een PKI wil de gebruiker de mogelijkheid om een (algemeen gezien) ongeldig certificaat toch te aanvaarden, ontzeggen, aangezien een gebruiker specifieke redenen kan hebben om dit certificaat toch te vertrouwen. Deze implementatiehouding ten opzichte van certificaten blijkt echter ongewenste gevolgen te hebben voor SSL, daarom noemen we ook de aanvallen die dit gedrag uitbuiten *side-channel attacks* of implementatie-aanvallen op SSL.

5.7.3 Certificaatvervalsing

De makkelijkste manier om het veronderstelde onrealistisch hoge vertrouwen van de gebruiker in SSL connecties uit te buiten, is het gebruik van zelf-gecertificeerde certificaten. Dit zijn certificaten die niet zijn gecertificeerd door een onafhankelijke CA, maar simpelweg door de eigenaar zelf zijn "gehandtekend". Elke SSL client-implementatie waarschuwt de client bij zo'n zelf-gecertificeerd certificaat, maar er wordt aangenomen dat vele gebruikers desondanks toch doorgaan met het leggen van een connectie.

5.7.4 Certificaatinjectering

Een meer gesofisticeerde aanval bestaat eruit om de lijst van betrouwbare *root CA*'s, die in elke SSL client ingebakken zit, te wijzigen. Dit kan met behulp van kwaadaardige software zoals *virussen* of eenvoudigweg door een persoon die voldoende rechten en toegang heeft tot het systeem. Deze aanval is zeer gevaarlijk, want quasi ondetecteerbaar, zelfs door een technisch onderlegd gebruiker.

Een demonstratie van zo'n aanval wordt beschreven in [47]. Vanuit het Internet Threat Model beschouwen we deze aanval echter niet als een realistisch gevaar, aangezien we daarin stellen dat de eindsystemen volstrekt veilig zijn.

In 2002 ontdekte Benham echter een zwakheid, die een aanval vanop afstand toelaat die erg vergelijkbaar is met deze certificaatinjectering:

Bepaalde SSL client-implementaties laten een willekeurige bezitter van een geldig certificaat toe om op zijn beurt "geldige"certificaten uit te geven voor andere, willekeurige servers. Dit privilege is normaal beperkt tot bezitters van een speciaal certificaat dat hen aanduidt als *intermediate CA*, maar aangezien de bewuste implementaties deze voorwaarde niet checken, beschouwen zij *elke* bezitter van een geldig certificaat als een *intermediate CA*, die geldige certificaten kan uitgeven.

De implementaties in kwestie zijn de Microsoft internet browsers Internet Explorer versie 5.0, 5.5 en 6.0. Er zijn geen andere implementaties met deze fout bekend [80].

Benham publiceerde ook een tool, *SSLsniff*, die deze aanval demonstreert [83].

Microsoft bracht echter al snel een patch uit, waardoor deze aanval tegenwoordig onmogelijk is [79].

5.7.5 Phishing Scam

Een voorbeeld van een *server spoofing attack* is de beruchte *phishing scam*, een type aanval dat recentelijk erg veel aandacht in de pers krijgt [73], [57], [40], [15].

Phishing is (voorlopig nog) geen bestaand Engels woord, maar is afgeleid van *fishing*, vissen, aangezien een aanvaller als het ware achter vertrouwelijke informatie (zoals kredietkaartnummers) zit te '*vissen*'.

Algemeen

Phishing is het opzetten van een webpagina die bijna niet te onderscheiden is van de originele webpagina van bijvoorbeeld een bank of internetwinkel, waarna een gebruiker naar deze valse pagina geleid wordt (bijvoorbeeld met behulp van een link in een vervalste e-mail), om op die manier persoonlijke of financiële gegevens en paswoorden te weten te komen. Dit kan bijvoorbeeld als volgt gebeuren:

> De aanvaller kopieert eenvoudigweg de webcode (html, javascript, ...) van een groot bedrijf dat hij wil nabootsen, en maakt hieraan enkele aanpassingen zodat de informatie die de gebruiker op deze pagina intikt rechtstreeks naar de aanvaller gemaild wordt, of ergens wordt opgeslagen waar de aanvaller het achteraf kan opvragen. Vervolgens stuurt hij de gebruiker een e-mail uit naam van het bedrijf dat hij probeert te impersoneren, met een link naar de vervalste pagina. Een onoplettende gebruiker zal nietsvermoedend deze link volgen, een pagina zien die er erg vertrouwd uitziet, en zijn persoonlijke informatie hierlangs ingeven.

Merk op dat deze aanval enkel werkt bij niet-technisch onderlegde en/of onoplettende gebruikers. Een oplettende gebruiker met enige vertrouwdheid met het internet, zal over het algemeen onmiddellijk bemerken dat de link die hij doorgestuurd krijgt, vervalst is, door in de adresbalk van zijn browser de eigenlijke link te inspecteren[13].

Recentelijk is er in Microsoft's Internet Explorer versies 5.0, 5.5 en 6.0 echter een fout ontdekt die een aanvaller toelaat om zelfs het adres in de adresbalk te vervalsen [81]. Ook hier zijn ondertussen patches voor uitgebracht [82], maar dit voorbeeld toont aan dat zelfs een gespecialiseerd gebruiker in de doeken gedaan kan worden door een phishing scam.

Phishing en SSL

Een phishing scam uitvoeren op een webpagina die met SSL beveiligd is (een HTTPS webpagina), is echter iets gecompliceerder. SSL heeft namelijk een mechanisme om de authenticiteit van de server te controleren, onder de vorm van certificaten, waardoor het in principe onmogelijk is voor een HTTPS server om zich als iemand anders voor te doen.

[13] Hoewel technieken als *typosquatting*, waarbij de aanvaller domeinen registreert die erg op het origineel gelijken (b.v. 'g00gle' voor 'google') dit erg kunnen bemoeilijken

Om te slagen moet een phishing aanval over SSL dus een certificaat vervalsen. Een goede vervalsing is onmogelijk, maar zoals we eerder zagen is het de volledige verantwoordelijkheid van de client om de geldigheid van een certificaat na te gaan. En daarin blijkt een probleem te schuilen. Want hoewel elke browser de client zal waarschuwen indien een certificaat ongeldig is, zal geen enkele browser een SSL connectie waarbij een ongeldig certificaat wordt voorgelegd categoriek verbieden. Hoogstens wordt de gebruiker gewaarschuwd met een cryptische boodschap. Zoals eerder nemen we echter aan dat vele gebruikers dit soort waarschuwen negeren (moedwillig, of omdat ze de foutmelding niet begrijpen) en desondanks vertrouwen op het bekende "slotje"dat zij in de statusbalk van hun browser zien verschijnen bij elke SSL connectie (maar dat op zich weinig zegt over de *kwaliteit* van de SSL connectie).

Geavanceerde phishing technieken

Eileen Zishuang Ye, Yougu Yuan en Sean Smith tonen in hun paper [93] aan dat het daarnaast mogelijk is om meer geavanceerde technieken te gebruiken om webservers te *spoofen*.

Zij demonstreren dat het met behulp van *JavaScript* mogelijk is om het complete uitzicht (statusbalk, adresbalk, knoppen, waarschuwingen, ...) van een client browser te manipuleren. Alle gesimuleerde delen zien eruit, en gedragen zich naar de gebruiker toe, als ware het de originele delen.

Bovendien kan zo ook het bekende SSL slotje gesimuleerd worden, samen met de waarschuwingen die een browser geeft in verband met de geldigheid van certificaten.

De vervalsing kan zo geslaagd gemaakt worden, dat zelfs een erg onderlegde gebruiker hier makkelijk door beetgenomen wordt.

Deze techniek laat daarom erg geavanceerde *phishing scams* toe.

5.7.6 Schaakgrootmeester-aanval

Het *schaakgrootmeester-probleem* ('*Chess Grandmaster problem*') vormt de basis voor een andere certificaat-side-channel aanval, die we de *schaakgrootmeester-aanval* zullen noemen.

Schaakgrootmeester-probleem

Het schaakgrootmeester-probleem stelt dat iemand zonder enige schaakkennis toch een schaakgrootmeester kan verslaan, als geïllustreerd in het volgende voorbeeld [72, p.109]:

> Mallory, die amper weet hoe een schaakbord eruit ziet, laat staan de regels van het spel kent, daagt Gary Kasparov en Anatoly Karpov[14] uit tot een partijtje schaak, op hetzelfde tijdstip en dezelfde plaats, maar in twee verschillende

[14] Dé grootmeesters van het schaken.

kamers. Mallory speelt met wit tegen Kasparov en met zwart tegen Karpov. Geen van beide grootmeesters weet van de andere af.

Karpov, die met wit speelt, opent het spel. Mallory onthoudt deze zet, gaat naar de kamer waar Kasparov speelt, en opent daar met diezelfde zet. Kasparov beantwoordt vervolgens deze zet, waarop Mallory terug naar Karpov loopt, en daar Kasparovs zet herhaalt. Dit gaat zo heel het spel door.

Beide grootmeesters zullen onder de indruk zijn van het spel van Mallory, hoewel Mallory eigenlijk niet meer dan een boodschapper is, en de grootmeesters in feite tegen elkaar spelen.

SSL en het schaakgrootmeester-probleem

Het schaakgrootmeester-probleem kan ook toegepast worden binnen SSL, waar het een schaakgrootmeester-aanval toelaat.

In zo'n aanval zal een aanvaller Mallory een *spoofing attack* uitvoeren op een client Alice, en zich voordoen als de server Bob, met behulp van een certificaatvervalsing. Indien deze vervalsing slaagt (deze kans is groot, zoals we eerder bespraken), zullen Alice en Mallory een SSL-beveiligde connectie opbouwen, waarbij Alice gelooft dat ze met Bob communiceert.

Daarnaast voert Mallory ook een *spoofing attack* uit op de echte server, Bob, waarbij hij zich voordoet als Alice. Dit is typisch makkelijker, want kan zonder certificaatvervalsing, aangezien client authenticatie zelden gebruikt wordt in SSL. Ook Mallory en Bob bouwen zo een veilige SSL-connectie op.

Als Alice nu begint te communiceren met Mallory, waarvan ze denkt dat hij Bob is, zal Mallory deze berichten doorsturen naar de echte Bob. Bob zal vervolgens antwoorden, en Mallory zal dit antwoord weer doorsturen naar Alice.

Alice communiceert nu dus met Bob, over twee verschillende, veilige SSL-connecties, met Mallory als een soort *proxy server*[15] tussenin. Alle communicatie passeert op onversleutelde wijze langs Mallory (die de data uit de ene SSL connectie decrypteert, om ze vervolgens weer te encrypteren voor de andere SSL connectie), waardoor deze alle data kan inkijken, en eventueel zelfs wijzigen.

Merk op dat deze aanval gecombineerd kan worden met de eerder beschreven *phishing scam*, om tot een nog krachtigere aanval te komen.

Tools

Verschillende tools laten toe om dit soort aanval[16] op SSL uit te voeren, we vermelden *Dsniff* [78], het eerder besproken *SSLsniff* [83] en *WinSSLMiM* [92].

[15] Een *proxy server* is een server die tussen een client applicatie en de echte server zit, en zo bijvoorbeeld de performantie kan verhogen door het antwoord op bepaalde serverrequests te *cachen.*

[16] Al deze tools noemen deze aanval een *man-in-the-middle attack*, maar wij hanteren een andere definitie van dit begrip, zoals verderop zal blijken.

5.8 Overzicht van onrechtstreekse aanvallen op SSL

Net als bij de rechtstreekse aanvallen, besluiten we ook hier met een overzicht van alle besproken aanvallen, waarbij voor elke aanval wordt aangegeven in hoeverre de veiligheid van een SSL connectie gecompromitteerd wordt, en aan welke eisen voldaan moet zijn om de aanval te lanceren (Tabel 5.2).

Merk op dat we certificaatinjectering hier niet vermelden, aangezien zo'n aanval binnen ons Internet Threat Model niet beschouwd wordt.

Aanval	**Gevaar** (Wat wordt gecompromitteerd?)	**Voorwaarden**
Timing aanval	private sleutel	timing van cryptografische operaties mogelijk
Bleichenbacher aanval	pre_master_secret	onderscheid tussen juiste en foute PKCS#1 formattering mogelijk
CBC padding aanval	stukjes ciphertext	- CBC mode gebruikt - onderscheid tussen padding en MAC verificatie mogelijk
Certificaatvervalsing (b.v. phishing scam)	volledige connectie	ongeldigheid certificaat genegeerd door gebruiker

Tabel 5.2. Onrechtstreekse aanvallen op het SSL Protocol

5.9 Frontale aanval op de onderliggende cryptografie van SSL

5.9.1 Inleiding

Tot nu toe hebben we uitsluitend aanvallen besproken die alle één of andere zwakheid van het SSL Protocol uitbuiten. Deze zwakheden zijn ofwel een gevolg van een gebrekkig ontwerp, waardoor een rechtstreekse aanval (*main-channel attack*) kan uitgevoerd worden, ofwel zijn zij het gevolg van een ondoordachte implementatie van de cryptografische primitieven in het SSL Protocol, met een mogelijke onrechtstreekse aanval (*side-channel attack*) als gevaar.

Nu zullen we echter een soort aanval bespreken die geen gebruikmaakt van één of andere zwakheid die ongewild in het SSL Protocol is gekropen. De *frontale aanval* of *brute-force attack* zal SSL aanvallen op zijn sterkste punt, door de onderliggende cryptografie rechtstreeks proberen te breken.

In plaats van via subtiele omwegen een zwakheid te zoeken en uit te buiten, zoals een inbreker die nauwkeurig een huis onderzoekt in de hoop ergens een open raam te vinden,

wordt het evenbeeld van een ramkraak uitgevoerd, waarbij met puur geweld de versterkte etalage wordt ingereden.

Bruteforce aanval

De meest voor de hand liggende frontale aanval, is het aflopen van alle mogelijke sleutels (of digests) tot de juiste sleutel gevonden is. Dit noemt men *bruteforce guessing*.

Merk op dat een aanvaller een juiste plaintext moet kunnen herkennen. Binnen SSL is dit echter geen probleem, aangezien SSL Records een herkenbare formaat hebben.

Het aantal mogelijkheden dat doorzocht moet worden bij *bruteforce guessing* hangt exponentieel samen met de gebruikte sleutellengte (of digestlengte): 2^X mogelijkheden bij een sleutellengte X.

Om deze reden wordt *bruteforcing* onder deze vorm enkel toegepast bij symmetrische encryptie en message digests, maar nooit bij asymmetrische encryptie met zijn grote sleutellengtes. Bij asymmetrische encryptie kan men uitgaan van bepaalde eigenschappen waaraan de sleutels moeten voldoen (om te vermijden dat de private sleutel uit de publieke kan worden afgeleid), waardoor men hierbij het proces enigszins kan optimaliseren aan de hand van allerlei wiskundige technieken, die de zoekruimte verkleinen.

Zelfs deze geoptimaliseerde aanvallen zullen echter nog steeds “frontaal”zijn. Zij maken geen gebruik van implementatiefouten, maar van eigenschappen die inherent zijn aan het gebruikte cryptografische algoritme. We zullen al dit soort aanvallen dan ook als frontale aanvallen of *bruteforce attacks* beschouwen, temeer omdat zij allen in de eerste plaats gebaseerd zijn op pure computerkracht, in plaats van subtiele achterdeurtjes.

Met computerkracht wordt in de eerste plaats de *processorsnelheid* van een computer bedoeld, het aantal berekeningen per seconde. Ook de *geheugenvereisten* zijn echter belangrijk, aangezien in andere technieken dan pure *bruteforce guessing* soms met gigantische tussenresultaten wordt gewerkt.

Moore’s Law

Als we over processorsnelheid spreken, en de evolutie van processorsnelheid doorheen de tijd, komen we automatisch bij **Moore’s Law** terecht. Gordon Moore, die mee aan de basis lag van hardware fabrikant **Intel**, publiceerde in 1965 een artikel waarin hij stelde dat het aantal componenten van halfgeleiderchips elke 12 maanden ongeveer verdubbelt [50]. Later herformuleerde hij dit, en schatte hij de tijd voor een verdubbeling in op ongeveer 2 jaar. Tegenwoordig staat de **Wet van Moore** bekend als volgt:

> *“De processorsnelheid van computers verdubbelt elke 18 maanden.”*

Hoewel deze trend zich niet tot in het oneindige kan blijven doorzetten, lijkt ze de volgende 10 jaar toch nog te zullen aanhouden [33].

Merk op dat deze trend als gevolg heeft dat het voor lange berekeningen nuttiger is om een tijd te wachten ('*slacking*') met de aankoop van hardware en het starten van de berekening, in plaats van onmiddellijk met bestaande hardware te beginnen rekenen [30].

MIPS Years

Tot slot definiëren we een maatstaf waaraan de kost van zulke frontale aanvallen afgemeten wordt. Hiervoor voerde men het begrip *MIPS years (MY)* in. Eén MY komt overeen met één jaar rekenen aan een miljoen instructies per seconde ('*Million Instructions Per Second*'). Met andere woorden:

1 MY = 1 * 10^6 instructies/sec * 3600 sec/uur * 24 uren/dag * 365 dagen/jaar
* 1 jaar = 3,1 * 10^{13} instructies

Volgens een recente (januari 2004) draft van Orman [53] kan een typische PC van enkele jaren geleden 500 MY[17] genereren, en kost zo'n PC ongeveer 100 dollar. Een sleutel breken van 5000 MY op een jaar tijd, kost dan 1000 dollar (10 PC's). Wil men dit op een dag doen, dan zal men hier 365 000 dollar veil voor moeten hebben.

Hierbij wordt er uitgegaan van de veronderstelling dat 10 parallelle computers van 500 MY even krachtig zijn als een computer van 5000 MY, met andere woorden, dat de computers perfect in parallel werken. In praktijk is dit niet haalbaar, waardoor we de mogelijkheden van een aanvaller lichtjes overschatten.

Het gebruik van uitsluitend MY's is bovendien een sterke simplificatie van het eigenlijke berekeningsproces, waarbij enkel de tijdskost in rekening wordt gebracht, en geheugenvereisten niet in beschouwing worden genomen. Om deze redenen zullen we MY's altijd gebruiken in combinatie met een schatting van de geheugenvereisten.

Volgens een bulletin van RSA Data Security, Inc. [77] gaat een verhoging in MY's (tijd) gepaard met de vierkantswortel van die verhoging voor de geheugenvereisten (ruimte). Als het aantal MY's dus verviervoudigt, verdubbelen de geheugenvereisten. We zullen de geheugenvereisten telkens op deze manier berekenen.

Deze gecombineerde schatting van het aantal MY's en de geheugenvereisten zal dan een goed idee geven van de kost die een aanvaller zich zal moeten getroosten om een bepaald systeem te breken, en vormt dus een valabel instrument voor de beveiligingsspecialist die wilt afwegen tot welke kost zijn informatie beschermd moet worden.

Botnets

Merk op dat we hierbij telkens uitgaan van de veronderstelling dat een aanvaller (of aanvallers) alle hardware, nodig om de frontale aanval uit te voeren, zelf financiert.

[17] Het aantal MY's is ruwweg gelijk aan het aantal MHz, dus een pc die 500 MY kan genereren heeft een processorsnelheid van 500 MHz [25].

Dit hoeft echter niet zo te zijn, in theorie is het ook mogelijk om gebruik te maken van zogenaamde *botnets*. Dit zijn netwerken bestaande uit PC's van onwetende gebruikers, verspreid over het hele internet, die gecompromitteerd zijn door een aanvaller (bijvoorbeeld met behulp van een speciaal virus), en die vanop afstand gestuurd kunnen worden door die aanvaller (of eender welke persoon die weet hoe hij deze compromittering moet uitbuiten).

In praktijk worden deze botnets vooral gebruikt voor *spamming* (het sturen van massa's ongevraagde e-mails) en *DOS* aanvallen, maar in theorie is het ook mogelijk deze PC's in te zetten voor meer geavanceerde aanvallen, zoals een frontale aanval op een bepaalde cryptografische sleutel. Praktisch gezien is zo'n geavanceerde frontale aanval echter veel moeilijker realiseerbaar, aangezien zo'n botnet dan op erg gecoördineerde wijze moet kunnen samenwerken, de resultaten van de individuele PC's in één centraal punt moeten samengebracht worden, enzovoort.

Omdat zo'n frontale aanval door middel van botnets nog nooit[18] in praktijk is gebracht en we zelfs niet weten of zo'n aanval zelfs maar mogelijk is in praktijk, beschouwen we dit soort aanvallen niet.

Indien zo'n aanval wel mogelijk zou blijken, verandert dit enkel iets aan de financiële kostprijs van een frontale aanval, die in elk geval daalt doorheen de jaren, omwille van de Wet van Moore.

De tijdsvereisten in de vorm van MY's en de geheugenvereisten daarentegen zijn wel onveranderlijk, en kunnen daarom altijd gebruikt worden als basis voor een nieuwe kosten/baten berekening, aangepast aan nieuwe processor- en geheugenprijzen.

Cryptografische algoritmen in SSL

Laten we nu eens bekijken hoe de onderliggende cryptografie van SSL bestand is tegen frontale aanvallen. In Sectie 4.2.1 werd in een tabel weergegeven welke cryptografische elementen van belang zijn in SSL, samen met hun belangrijkste algoritmes. We herhalen deze tabel hier nog een keer (Tabel 5.3).

Doel	Operatie	Algoritme
Authenticatie	Digitale handtekening (certificaat)	RSA
Sleuteluitwisseling	Asymmetrische encryptie	RSA
Data-encryptie	Symmetrische encryptie	RC4/RC2/DES/3DES/AES
Integriteit	Digest (in MAC)	MD5/SHA-1

Tabel 5.3. Cryptografische operaties in SSL en hun algoritmen herbekeken

Voor elk van deze algoritmen zullen we in meer detail bespreken in welke mate ze bestand zijn tegen frontale aanvallen, en tot welke kost een aanvaller bereid moet zijn om in zijn aanval te slagen.

[18] Althans voorzover wij weten.

5.9.2 Digitale handtekeningen en asymmetrische encryptie aangevallen

Als we binnen SSL spreken over digitale handtekeningen en asymmetrische encryptie, spreken we over RSA, aangezien we Diffie-Hellman niet beschouwen.

Zoals al eerder aangehaald is puur *bruteforce guessing* van de private sleutel bij dit soort algoritmen computationeel onmogelijk, omwille van de grote sleutellengtes. *Bruteforcing* van een RSA sleutel komt in praktijk neer op *factorizatie* van de modulus.

Factorizatie is het vinden van de kleinste delers (de factors) van een natuurlijk getal. Zoals besproken in Sectie 2.6.1, wordt een RSA modulus opgebouwd aan de hand van twee grote priemgetallen p en q, die met elkaar vermenigvuldigd worden. Met deze p en q wordt naast de modulus ook de private exponent d berekend. Een aanvaller die uit de modulus n (die publiekelijk beschikbaar is in de publieke sleutel) q en p kan berekenen (door middel van factorizatie), kan dus de bijhorende private sleutel berekenen.

Gelukkig is factorizatie een probleem dat erg goed bestand is tegen pure rekenkracht, aangezien er tot nu toe geen algoritme is gevonden dat dit probleem in polynome tijd kan oplossen[19]. Dat wil zeggen dat het factorizeren van kleine getallen geen enkel probleem is, maar het factorizeren van grote getallen wordt al snel computationeel onmogelijk.

Maar wat is een groot getal? Als we over SSL spreken, spreken we over een RSA sleutellengte (en dus modulus) van minstens 512 bit, die gebruikt wordt bij Ephemeral RSA.

Hoe sterk is een RSA 512 bit sleutel?

RSA 512 bit

Op 22 augustus 1999 slaagde een internationaal team van onderzoekers in het factorizeren van een RSA 512-bit sleutel (ook wel RSA-155 genoemd, omdat de modulus uit 155 decimale cijfers bestaat). Zij maakten daarbij gebruik van de *General Number Field Sieve*(GNFS) methode, en hadden hier 7,4 maanden voor nodig [67].

We vermelden kort de twee stappen waarin het GNFS proces uiteenvalt:

1. In een eerste fase worden verschillende vergelijkingen opgesteld in een proces dat bekend staat als *sieving*. Deze is de meest intensieve stap, maar kan gemakkelijk gedistribueerd worden over een groot aantal eenvoudige computers die een parallel netwerk vormen. De onderzoekers gebruikten hiervoor een mengeling van PC's met processorsnelheden tussen 175 en 500 MHz en 64 MB geheugen.
2. In de tweede fase worden dan alle vergelijkingen uit de vorige fase samengebracht, en wordt dit stelsel opgelost. Dit proces is erg geheugenintensief, en vereist een computer met erg veel geheugen. De onderzoekers gebruikten hiervoor een Cray C916 supercomputer met 3,2 GB geheugen.

De details van deze fases staan uitgebreid beschreven in [10], en vallen buiten het bestek van deze thesis.

[19] Zulke problemen worden *NP problemen* genoemd in complexiteitstheorie.

We citeren uit deze paper echter nog een interessante opmerking van de auteurs:

> *"Gebaseerd op onze ervaring met het factorizeren van grote getallen schatten wij dat binnen de drie jaar de algoritmen en computertechnologie die gebruikt werden voor het factorizeren van RSA-155, wijdverspreid zullen zijn, tenminste binnen de wetenschappelijke wereld, zodat tegen dan 512-bit RSA sleutels zeker niet meer veilig zullen zijn. Dat maakt deze sleutels onbruikbaar voor authenticatie of bescherming van data die langer dan een paar dagen beveiligd moet blijven."*

Merk op dat deze paper van 2000 stamt, en dat met "*drie jaar later*" dus ongeveer 2003 bedoeld wordt. In het bijzonder merken zij op dat SSL omwille van de exportregulering gebruik maakt van 512-bit RSA (bij Ephemeral RSA) en raden zij expliciet aan dat SSL, om veilig internetverkeer te waarborgen, moet overschakelen naar tenminste 768 bit, of zelfs 1024 bit. Door middel van een extrapolatie stellen zij dat 786-bit RSA sleutels tegen 2010 gefactorizeerd zullen zijn, en 1024-bit sleutels tegen het jaar 2018. Een andere bron, Silverman, stelde rond dezelfde tijd echter dat 1024-bit RSA sleutels niet gefactorizeerd zullen worden vóór 2037 [76].

Laten we de kost voor het factorizeren van een 512-bit RSA sleutel zelf eens berekenen. We nemen zoals Orman aan dat een PC van 500 MY 100 dollar kost, en dat geheugen ons 0,5 dollar per MB armer maakt [53]. Bovendien weten we uit de voorgaande studie dat dit factorizeren zo'n 8400 MY kost.

Om een RSA 512-bit sleutel op een jaar te breken, zijn dus 17 PC's van elk 500 MY (en 64 MB geheugen) nodig. Dat geeft een kost van slechts 1700 dollar. Daarbij komt nog de kost voor een "supercomputer" met ongeveer 3,2 GB geheugen, ruwweg 1600 dollar. De totale kost van 3300 dollar ligt duidelijk binnen het bereik van een individu.

Indien we zo'n sleutel in minder dan een jaar willen breken, bijvoorbeeld een dag, is dan 620 500 (= 365*1700) dollar nodig voor 6205 van zulke PC's, voor de eerste stap van het GNSF proces. Om de tweede stap in deze veel kortere tijdspanne op te lossen blijft een "supercomputer" met 3,2 GB geheugen nodig[20]. Deze kost zo'n 1600 dollar. De totaalprijs voor het kraken van een 512 bit RSA sleutel binnen een dag, komt dan neer op 622 100 dollar, een bedrag dat duidelijk in het bereik ligt van middelgrote tot grote organisaties. Bovendien kunnen we uit Moore's Law afleiden dat deze kost elk jaar lager wordt, waardoor de kost elk jaar verkleint, en dus het risico elk jaar vergroot.

Het lijdt dus geen twijfel dat Ephemeral RSA een zwakheid van SSL is, waar zo snel mogelijk van afgestapt moet worden. Bovendien verzwakt deze exportuitbreiding het complete SSL protocol, aangezien we eerder zagen dat via een versleutelsetverlagingsaanval elke connectie kan teruggebracht worden tot de zwakste gemeenschappelijke versleutelset.

We merken nog op dat in augustus 2003 een ontwerp voor een speciale computer werd voorgesteld, TWIRL [74]. Deze machine zou in staat zijn de eerste stap van het GNFS proces voor een 512-bit RSA sleutel in slechts 10 minuten te berekenen. Bovendien zou zo'n machine slechts 10 000 dollar kosten. Al snel kwam er echter kritiek [44] op deze paper, en werd de onderzoekers verweten te rooskleurig geweest te zijn in hun berekeningen. Tot

[20] Merk op dat de geheugenvereisten hier niet toenemen, aangezien deze afhankelijk zijn van de sleutelgrootte, niet van het aantal PC's dat in de eerste stap wordt ingezet.

zo'n machine echt gebouwd wordt en zijn beloften bewijst waar te maken, houden we dus onze meer conservatieve berekeningen aan.

Opmerking

We merken nog op dat het dan wel computationeel onmogelijk mag zijn om de private sleutel aan de hand van een *bruteforce guessing* aanval uit de publieke sleutel af te leiden, toch kan een frontale aanval waarbij een aanvaller een publieke sleutel-geëncrypteerde plaintext te weten tracht te komen soms verrassend succesvol zijn. Publieke sleutel-cryptografie blijkt immers gevoelig te zijn voor *chosen-plaintext* aanvallen, waarbij de aanvaller alle mogelijke plaintexts eenvoudigweg encrypteert met de vrij beschikbare publieke sleutel, en de resulterende ciphertexts vergelijkt met de te kraken ciphertext.

Hoe makkelijk zo'n aanval is, hangt volledig af van de voorspelbaarheid van de versleutelde plaintext. Indien er slechts een beperkt aantal mogelijke plaintexts zijn, is deze aanval erg effectief. Indien de plaintext echter uit een random gegenereerde sessiesleutel bestaat, zoals in SSL het geval is, zal de hele zoekruimte van mogelijke sessiesleutels doorlopen moeten worden, zoals bij een frontale aanval op een symmetrische encryptie met die sessiesleutel. Een aanvaller zal bovendien al deze sleutels telkens moeten encrypteren met behulp van een relatief dure publieke sleutel-operatie.

Binnen SSL is zo'n *chosen-plaintext* aanval op de asymmetrische geëncrypteerde sessiesleutel dus een stuk duurder dan een frontale aanval op de symmetrische encryptie zelf (aan de hand van die sessiesleutel), we zullen dit soort *chosen-plaintext* aanval dan ook niet verder onderzoeken.

In de volgende sectie gaan we na hoe moeilijk het *bruteforcen* van een symmetrische encryptie is.

5.9.3 Symmetrische encryptie aangevallen

Inleiding

Bij het *bruteforcen* van symmetrische encryptie wordt typisch de hele sleutelruimte afgelopen, die 2^X sleutels groot is, bij een sleutellengte van X. De tijd die hiervoor nodig is, is gemiddeld[21] 2^{X-1}.We merken hierbij op dat geheugenvereisten bij dit soort *bruteforcing* geen probleem vormen, volgens een bulletin van RSA Data Security, Inc. [77] volstaan enkele kilobytes.

We geven in de eerste plaats een indicatie van de tijd en moeite die nodig is om een symmetrische sleutel van lengte X in het algemeen te *bruteforcen*, zonder onderscheid te maken tussen de verschillende algoritmes. Vervolgens bekijken we elk algoritme apart, en bespreken we de specifieke eigenschappen van dit algoritme.

[21] In het slechtste geval (voor de aanvaller) is dit 2^X - 1, maar voor de gebruiker van de encryptie is het logischer om uit te gaan van het gemiddelde geval, waarbij de aanvaller de juiste sleutel vindt na het doorzoeken van half de zoekruimte.

Bruteforcing symmetrische sleutels

Binnen SSL zijn symmetrische sleutels gewoonlijk 128 bit, en in uiterste gevallen tot 256 bit lang. In export versleutelsets kan deze lengte echter verminderen tot slechts 40 bit.

Een MY berekening wijst uit dat een 128 bit sleutel het ongeveer 60 jaar zou uithouden, tegenover hardware ter waarde van een triljoen dollar [53]. Daarmee bedoelt men dat over 60 jaar hardware beschikbaar is, ter waarde van 1 triljoen, die zo'n sleutel in een jaar kan breken, waarbij een evolutie van de hardware volgens de Wet van Moore wordt verondersteld[22]. Zelfs voor de meest veeleisende veiligheidsapplicaties is dit voldoende [53].

Een 256 bit sleutel houdt het, volgens dezelfde berekeningen, tenminste 400 jaar uit [53].

Voor 40 bit vonden we geen referentie, hiervoor gebruiken we een eigen berekening:

$$\text{zoekruimte} = 2^{40} \text{ sleutels}$$
$$\text{kost} = \frac{zoekruimte}{aantalInstructiesInMY} = \frac{2^{40-1}}{3{,}1*10^{13}} = 0,018 \text{ MY}$$

Eerder zagen we dat een gemiddelde PC gemakkelijk 500 MY aankan (en 100 dollar kost), dus zo'n PC kan een 40-bit sleutel dan in een kleine 19 ($= (0,018/500) * 365 * 24 * 60$) minuten breken.

Met 4 PC's, ter waarde van 400 dollar, zou dit slechts 5 minuten kosten, met 20 PC's (2000 dollar) ongeveer 1 minuut, met 120 PC's (12 000 dollar) een kleine 10 seconden.

Een organisatie die een miljoen dollar voor deze operatie veil heeft, kan 10 000 van zulke pc's inzetten, waardoor het 5 000 000 MY kan genereren, waarmee zo'n sleutel in ongeveer 0,1 seconden ge*bruteforced* kan worden.

RC4

De RC4 implementatie binnen SSL blijkt geen zwakheden te vertonen waardoor andere dan *bruteforce guessing* aanvallen mogelijk worden.

Wel ontdekten Scott Fluhrer, Itsik Mantin, and Adi Shamir in 2001 een *ciphertext-only* aanval op RC4 [23], maar deze bleek niet van toepassing te zijn binnen SSL omdat SSL de encryptiesleutel en IV *pre-processed* met behulp van de hashing functies MD5 en SHA-1 [84].

De sterkte van RC4 is dus compleet afhankelijk van zijn sleutellengte. Binnen SSL is dat normaal 128 bit. Bij de export versleutelsets is dit 40 bit.

[22]Dit is een voorbeeld van *slacking*

RC2

Op RC2 zijn geen speciaal performante aanvallen bekend, dus ook hier is de sterkte afhankelijk van de sleutellengte. Binnen SSL is dit altijd 128 bit, tenzij voor export versleutelsets, waar het 64 bit is. Merk op dat bij RC2 een sleutellengte van 64 bit overeenkomt met een effectieve sleutellengte van 40 bit. Een sleutellengte van 128 bit is echter ook effectief 128 bit [60, p.31].

DES en 3DES

Hoewel ook DES geen specifieke zwakheden bezit, bouwde de **Electronic Frontier Foundation (EEF)** in 1998 voor 130 000 dollar een machine die elke DES sleutel binnen 10 dagen kan kraken [26]. Bovendien gaven de ontwerpers toe dat hun machine niet goed geoptimaliseerd was, en men schatte toen dat met 10 keer meer geld een machine kan gebouwd worden die ongeveer 50 keer sneller is [53].

Het feit dat DES zo makkelijk te breken is, ligt uitsluitend in het feit dat zijn sleutellengte zo kort is (56 bit). Daarom werd 3DES ontwikkeld, dat een effectieve sleutellengte van 112 bit bezit (zie ook 2.5.2).

Een MY berekening toont dat men zelfs met hardware ter waarde van een triljoen 50 jaar zou nodig hebben om een 3DES sleutel (112 bit) te kraken.

AES

Geen goede aanvallen buiten *bruteforce guessing* zijn bekend op AES. Het breken van dit algoritme is dus volkomen afhankelijk van zijn sleutellengte, 128 of 256 bit.

5.9.4 Message digest aangevallen

SSL maakt intensief gebruik van de digest algoritmen MD5 en SHA-1.

Omdat de message digests in SSL altijd nog extra beschermd zijn met behulp van encryptie, lijkt er op het eerste zicht binnen SSL geen markt te zijn voor lekken in digestalgoritmes, tenzij deze lekken een side-channel attack op de omhullende encryptie zouden toelaten.

We zullen echter later zien dat zo'n lek wel degelijk zijn nut kan hebben, bij bijvoorbeeld een versleutelsetverlagingsaanval. Om deze reden bespreken we toch kort hoe moeilijk het is zo'n digest algoritmes te compromitteren.

We kunnen zo'n digest algoritme op twee manieren compromitteren:

- **Omkering**
  We vinden bij een digest de[23] input. De onomkeerbaarheidseigenschap wordt dus

[23] Aangezien we nooit zeker kunnen weten of dit de oorspronkelijke input is (omkering is niet uniek), is *een* input eigenlijk correcter.

aangevallen.

- **Botsing**
  We vinden een botsing tussen twee willekeurige[24] inputs, met andere woorden, we vinden twee inputs die dezelfde digest geven. Hier wordt dus de botsweerstand op de proef gesteld.

Aangezien er geen goede aanvallen op MD5 of SHA-1 bekend zijn [63], vallen we ook hier terug op frontale aanvallen.

Een frontale aanval op de omkering van digest algoritmen komt neer op het afgaan van alle mogelijke inputs, waarvan telkens de bijhorende digest berekend wordt, die vergeleken wordt met de oorspronkelijke, om te keren digest. Dit is ondoenbaar indien we de lengte van de oorspronkelijke input niet kennen, aangezien elke inputlengte eenzelfde digestlengte geeft.

Indien de inputlengte echter gekend is, is het omkeren van zo'n digest vergelijkbaar met het bruteforcen van een symmetrische encryptiesleutel van gelijke lengte, met dit verschil dat een aanvaller rekening moet houden met het feit dat de omkering niet uniek is, dit wil zeggen, er zijn meerdere inputs die tot eenzelfde digest herleiden. Meestal is de juiste input echter wel herkenbaar, aangezien die moet bestaan uit begrijpelijke tekst, of enige andere bekende formattering bezit. We merken hierbij op dat enkel de lengte van de geheime input telt, indien daarnaast nog extra input gebruikt wordt, die niet geheim is, verhoogt de weerstand tegen omkering niet.

Botsingen kunnen met behulp van een frontale aanval gevonden worden voor een kost die gelijk is aan het bruteforcen van een symmetrische sleutel met een sleutellengte die gelijk is aan de helft van de digestlengte, omwille van de *Birthday Paradox*, zoals we eerder zagen. Met andere woorden, de botsingsweerstand van een digest met lengte X is gelijk aan $\frac{X}{2}$, bruteforcen kost dan gemiddeld $2^{\frac{X}{2}-1}$ operaties.

5.9.5 Overzicht van frontale aanvallen op SSL

Tabel 5.4 werd opgesteld aan de hand van [77], waar, via een extrapolatie van factorizaties uit het verleden, schattingen worden gegeven voor de toename van de kost en de geheugenvereisten, bij een vergroting van de sleutellengte, en het gebruik van GNFS.

Tabel 5.5 geeft dan een overzicht van de kost om symmetrische sleutels te bruteforcen. Dit houdt eenvoudigweg in dat de volledige zoekruimte doorlopen wordt. Als tijdskost wordt de formule 2^{X-1} gebruikt. Geheugenvereisten worden hier niet vermeld, aangezien deze triviaal (= enkele kilobytes) zijn volgens [77].

5.10 SSL Exploits

Naast de besproken rechtstreekse en onrechtstreekse aanvallen op SSL, en de frontale aanvallen op de onderliggende cryptografie van SSL, is er in praktijk nog een vierde soort

[24]Indien deze niet willekeurig zijn, en één van beide inputs bekend is, is dit in feite een variante van de omkering

Sleutel (bit)	**Kost** (in MY)	**Geheugen** (GB)
512	8400	3,2
768	$5{,}04 * 10^{7}$	246
1024	$5{,}88 * 10^{10}$	8480
2048	$7{,}56 * 10^{19}$	$2{,}88 * 10^{8}$

Tabel 5.4. Kost voor het bruteforcen van asymmetrische RSA sleutels (met GNFS methode)

Sleutel (bit)	**Zoekruimte** (aantal sleutels)	**Kost** (in MY)
40	$1{,}1 * 10^{12}$	0,018
56	$7{,}2 * 10^{16}$	1162
128	$3{,}4 * 10^{38}$	$5{,}49 * 10^{24}$
256	$1{,}16 * 10^{77}$	$1{,}87 * 10^{63}$

Tabel 5.5. Kost voor het bruteforcen van symmetrische sleutels (m.b.v. bruteforce guessing)

aanvallen: SSL *exploits*.

Een *exploit* is een stukje software dat beveiligingslekken uitbuit ('exploiteert'), die een gevolg zijn van programmeerfouten die bijvoorbeeld *buffer overflows* doen ontstaan, of *format string attacks* [41, p. 55-82] mogelijk maken.

In een onze definitie van het Internet Threat Model gaan we er echter vanuit dat de eindsystemen veilig zijn. We interpreteren deze definitie ruim, en stellen dat dit ook betekent dat de software op deze systemen vrij is van programmeerfouten.

We beschouwen dit soort aanvallen dus niet, hoewel er een heel aantal beveiligingslekken (en overeenkomstige exploits) voor bijvoorbeeld OpenSSL bekend zijn [55]. In de regel worden deze beveiligingslekken trouwens erg snel gedicht, door middel van zogenaamde *patches*[25].

5.11 Voorwaarden voor een veilige SSL connectie herbekeken

In Sectie 3.6 werd een overzicht gegeven van de elementen die cruciaal zijn in het garanderen van een veilige SSL connectie. Laten we deze voorwaarden nu eens terug overlopen, en bekijken hoe deze elementen bedreigd worden door de voorgaande aanvallen.

[25]Hoewel we wel moeten opmerken dat het *toepassen* van deze patches dikwijls erg achterop hinkt.

Pre-master secret

Deze voorwaarde wordt bedreigd door de *bleichenbacher aanval* en afgeleiden. Geen enkele van deze aanvallen zijn echter mogelijk in recente implementaties.

Private sleutel van de server

Het doel van de *timing aanval* bestaat uit het compromitteren van deze voorwaarde. Ook een bepaalde afgeleide van de bleichenbacher aanval (met name diegene die gebruikt maakt van RSA-KEM) vormt een bedreiging voor de private sleutel. Recente implementaties van SSL zijn volkomen beschermd tegen al deze aanvallen.

Tot slot wordt de private sleutel bedreigd door een frontale aanval. Zo'n *bruteforce* aanval is computationeel onmogelijk voor de normale sleutellengtes, maar een export RSA sleutel van 512 bit blijkt te kraken, al is de kost relatief hoog.

Sterke random generator

Netscape's implementatie van SSLv2 vertoonde in 1996 op dit gebied een zwakheid [27]. Recente implementaties hebben echter allemaal cryptografisch sterke PRNG's.

Sterke versleutelset

In de eerste plaats zal de *versleutelsetverlagingsaanval* hierop proberen in te spelen. Deze aanval blijkt af te hangen van de kracht van de cryptografische algoritmen, het al dan niet computationeel mogelijk zijn van een *bruteforce* aanval, in de ondersteunde versleutelsets. Aangezien we zagen dat de sleutellengtes in bepaalde versleutelsets (in het bijzonder de export versleutelsets) te *bruteforcen* zijn in redelijk korte tijd (maar voor een betrekkelijk hoge kost), lijkt deze aanval wel degelijk praktisch haalbaar.

Daarnaast zullen ook *versieverlaging* en de *changecipherspec-aanval* ingrijpen op de versleutelset, zij het op een minder rechtstreekse manier. Recente implementaties zijn echter immuun voor deze laatste twee aanvallen.

Geldig certificaat

De onrechtstreekse certificaat-aanvallen bedreigen deze voorwaarde. Aangezien kan worden aangenomen dat gebruikers over het algemeen redelijk laks zijn in het controleren van deze voorwaarde, is een aanval op deze voorwaarde reëel. Meer bepaald de *phishing scam* en *schaakgrootmeester-aanval* blijken erg haalbaar in praktijk.

Conclusie

We concluderen dat SSL een robuust protocol blijkt te zijn, waarvan de voorwaarden goed beschermd worden, althans in recente implementaties.

De recentste SSL implementaties blijken volkomen immuun voor de meeste side-channel attacks, enkel certificaatvervalsing (zoals bij een phishing scam) lijkt een realistisch gevaar te vormen, indien we uitgaan van de veronderstelling dat vele gebruikers de certificaatwaarschuwingen van hun SSL implementaties domweg negeren. Het zou interessant zijn om de juistheid van deze veronderstelling na te gaan, aangezien ons geen enkele onderzoek bekend is waarin dit zwart op wit wordt aangetoond. In deze studie beperken we ons echter tot puur computerwetenschappelijk onderzoek, en zullen we deze assumptie eenvoudigweg aannemen. We beschouwen certificaatvervalsing dus als een belangrijk gevaar, dat zeker niet verwaarloosd mag worden.

Daarnaast lijkt van alle besproken main-channel attacks enkel de versleutelsetverlagingsaanval (in combinatie met de cryptografische zwakheid van de export versleutelsets) mogelijk een gevaar te vormen, hoewel dit nooit overtuigend is aangetoond. In de rest van deze studie zullen wij ons dan ook concentreren op deze aanval, en nagaan of deze echt een gevaar vormt voor het SSL protocol, en zo ja, in welke mate.

Hoofdstuk 6

Man-in-the-middle aanval

"Internet is not a game,

we need qualified admin and programmer to make secure protocols and programs.

It isn't our fault if the net is so insecure...

but we can do something to harden it.

Ettercap was developed with this idea in mind:

if even a skript kiddies can do an hijack of the local connections,

probably someone will find a solution for that!"

– De Ettercap manpage
(Letterlijk overgenomen, met bijhorende taalfouten)

6.1 Inleiding

Als we tot hiertoe over een aanval op het SSL protocol spraken, hadden we het meestal over drie partijen: een *client*, een *server* en een *aanvaller*.

Bovendien gingen we er telkens vanuit dat deze aanvaller in de mogelijkheid is om een passieve aanval uit te voeren, waarbij hij pakketten die tussen client en server vloeien kan inkijken, of zelfs een actieve aanval, zodat hij deze pakketten ook kan wijzigen, of pakketten aan de connectie kan toevoegen of verwijderen. Dat impliceert dat de aanvaller er op één of andere manier in slaagt om zich *tussen* client en server te plaatsen.

Tot nu toe is echter altijd nagelaten om dieper in te gaan op de vraag *hoe* de aanvaller hierin slaagt. Dit maken we in dit hoofdstuk goed.

We beginnen met een definitie van een zogenaamde *man-in-the-middle (MITM) aanval*, waarna we de toepasbaarheid van zulke aanvallen bespreken.

Vervolgens gaan we dieper in op enkele technieken die gebruikt kunnen worden om zo'n

MITM aanval te lanceren. Daarna stellen we kort de acties voor die een aanvaller kan ondernemen na het uitvoeren van zo'n MITM aanval, met andere woorden, wat is het gevaar van een MITM aanval?

We besluiten met een praktische implementatie van een MITM aanval.

6.2 Man-in-the-middle aanval

6.2.1 Definitie

Elke aanval waarbij een aanvaller *actief* actie onderneemt om ervoor te zorgen dat de communicatie tussen client en server via hemzelf passeert, noemen we een *man-in-the-middle (MITM) aanval.*

Zo'n MITM aanval geeft de aanvaller vervolgens de mogelijkheid om een passieve of actieve aanval op de communicatie tussen client en server uit te voeren.

6.2.2 Toepasbaarheid

Een MITM aanval kan toegepast worden op drie niveaus, ingedeeld vanuit het oogpunt van de aanvaller [54]:

- **Lokaal ('Local')**
  Client, server en aanvaller bevinden zich allen op hetzelfde lokale netwerk.
- **Lokaal naar niet-lokaal ('Local to remote')**
  Aanvaller en één van beide communicerende partijen (b.v. client) bevinden zich op eenzelfde lokaal netwerk, de andere communicerende partij (b.v. server) bevindt zich buiten dit lokale netwerk, en is bereikbaar langs een netwerkpunt op dat lokale netwerk dat zich gedraagt als een toegangspoort tot externe netwerken ('*gateway*').
- **Niet-lokaal ('Remote')**
  De aanvaller bevindt zich noch op het lokale netwerk van de client, noch op het lokale netwerk van de server.

Hierbij moeten we opmerken dat zelfs een niet-lokale aanval nog altijd vereist dat een aanvaller zich op betrekkelijk korte afstand van de beide slachtoffers bevindt. Hiermee bedoelen we dat er zich niet teveel verschillende netwerken mogen bevinden tussen de aanvaller en minstens één van de slachtoffers. Voor een aanvaller in bijvoorbeeld Australië is het onmogelijk om een MITM aanval op het verkeer tussen twee Amerikanen uit te voeren, omdat het aantal verschillende netwerken en bijgevolg *routers*[1] tussen aanvaller en slachtoffer (die allen op de één of andere manier bedrogen moeten worden) eenvoudigweg te groot is.

[1] De netwerkapparaten die twee netwerken op de netwerklaag met elkaar verbinden, en op een selectieve manier, op basis van IP adressen, pakketten tussen deze netwerken doorsturen.

In praktijk is het voor een aanvaller echter zelden moeilijk om zich op een redelijke afstand van minstens één van beide slachtoffers te plaatsen, waardoor een MITM aanval een reëel gevaar vormt.

6.3 ARP vergiftiging ('ARP poisoning')

ARP vergiftiging is een MITM aanval op *datalink niveau*, en kan gebruikt worden voor een *lokale* of *lokaal naar niet-lokale* MITM aanval.

We leggen eerst uit wat ARP is, waarna besproken wordt op welke manier dit mechanisme gebruikt kan worden om een MITM aanval uit te voeren.

6.3.1 ARP

Communicatie over het internet gebeurt over de *netwerklaag*, via het Internet Protocol (IP). Binnen een **Local Area Network (LAN)** wordt echter gewerkt op de *datalink laag*, meestal **Ethernet**. Voor meer informatie over dit lagenmodel verwijzen we naar Sectie 2.2.

Ethernet gebruikt geen IP adressen, maar *hardware adressen* om pakketten op hun bestemming te krijgen, om netwerkapparaten verbonden met het netwerk ('*hosts*') te identificeren. Dit hardware adres wordt ook wel eens een **Media Access Control (MAC) adres** genoemd, maar om verwarring met Message Authentication Codes (MAC) te vermijden, zullen we altijd de term hardware adres gebruiken.

Er moet echter een manier zijn om IP adressen naar hardware adressen te vertalen om IP pakketten binnen het LAN op hun bestemming te krijgen. Dit vertalingsproces noemt men *adres resolutie*, en het mechanisme dat hiervoor verantwoordelijk is, wordt het **Address Resolution Protocol (ARP)** genoemd.

ARP wordt uitgebreid gespecificeerd in RFC 826 [58] en werkt als volgt:

Alle hosts binnen het Ethernet netwerk houden een tabel bij met *mappings* tussen IP adressen en hun overeenkomstige hardware adressen. Deze tabel wordt een *ARP cache* genoemd, en is *dynamisch*. Dat wil zeggen dat de mappings in de cache na een vastgesteld tijdsinterval (typisch 2 minuten) vervallen en verdwijnen, en dat er regelmatig mappings worden toegevoegd of veranderd.

Het toevoegen of veranderen van mappings in de ARP cache gebeurt met behulp van ARP berichten. Er zijn twee soorten ARP berichten:

- **ARP Request**
  Dit bericht wordt verstuurd door een host die het hardware adres behorende bij een bepaald IP adres te weten wil komen.
- **ARP Reply**
  Dit bericht wordt gestuurd als antwoord op een ARP Request, en bevat het gevraagde hardware adres.

Een client die wil communiceren met een server op hetzelfde LAN handelt dan als volgt, waarbij we ervan uitgaan dat de client het IP adres van de server kent:

- de client *broadcast* een `ARP Request` (met daarin het IP adres van de server als de bestemming), om het hardware adres van de server te weten te komen.
- de server (die zich herkent als de bestemming aan de hand van het IP adres in de `ARP Request`) antwoordt met een `ARP Reply`, waarin hij zijn hardware adres naar de client stuurt. De server extraheert tegelijkertijd het hardware en IP adres van de client uit de `ARP Request`, en slaagt deze mapping op in zijn ARP cache.
- de client stuurt de pakketten die bestemd zijn voor de server naar het hardware adres uit de `ARP Reply`.

Bovendien blijkt ARP een interessante eigenschap te vertonen, die men *statenloosheid* (‘*statelessness*’) noemt. Dat wil zeggen dat een host een `ARP Reply` kan sturen zonder een eerdere `ARP Request` ontvangen te hebben; zo’n `ARP Reply` noemt men dan een ongevraagde (‘*gratuitous*’) `ARP reply`. Andersom zal een host die een `ARP Reply` ontvangt, de informatie uit dit bericht gebruiken om een bestaande mapping in zijn ARP cache *aan te passen*, zelfs indien hij geen `ARP Request` heeft gestuurd, of de `ARP Reply` niet voor hem bestemd is.

Daarenboven zal, volgens de originele specificatie [58], een host de informatie uit zo’n ongevraagde `ARP Reply` zelfs gebruiken om een mapping *toe te voegen*, indien deze als bestemming het IP adres van deze host bevat.

Eigen experimenten tonen echter aan dat de specificatie hier niet altijd gevolgd wordt, toch niet door de besturingsystemen die wij getest hebben (Debian 3.0, Mandrake 9.0 en Windows XP). Deze systemen blijken *geen* mapping *toe te voegen* indien zij een ongevraagde `ARP Reply` ontvangen die expliciet voor hen bestemd is. Een bestaande mapping *updaten*, gebeurt wel. Daarnaast zou ook Solaris dit gedrag vertonen [8].

6.3.2 ARP Vergiftiging

Omwille van deze statenloosheid leent ARP zich uitermate goed voor een MITM aanval. Een aanvaller kan immers een vervalst `ARP Reply` bericht (al dan niet ongevraagd) naar de client sturen, waarin wordt verkondigd dat het IP adres van de server naar het hardware adres van de aanvaller vertaald moet worden. Op deze manier zorgt de aanvaller ervoor dat de client een verkeerde mapping naar zijn ARP cache schrijft, en dat de client vervolgens alle pakketten, bestemd voor de server, naar de aanvaller zal sturen.

Dit proces noemt men *ARP vergiftiging*, de ARP cache van de client wordt letterlijk vergiftigd.

Indien de aanvaller zowel client als server op deze manier *vergiftigt*, en de pakketten na ontvangst telkens verder stuurt (‘*forward*’) naar de juiste partij (al dan niet na een wijziging), krijgen we een MITM aanval. Alle verkeer tussen client en server wordt dan langs de aanvaller geleid.

Hierbij moeten we wel opmerken dat deze aanval volgens onze experimenten enkel mogelijk is als er al een mapping voor het bewuste IP adres in de ARP cache aanwezig is, aangezien een `ARP Reply` zonder voorgaande `ARP Request` enkel mappings in de ARP cache blijkt te kunnen *updaten*, niet toevoegen, althans in de door ons geteste systemen.

Hoewel deze besturingssystemen enkel `ARP Reply`'s aanvaarden als updates voor mappings die al in de ARP cache aanwezig zijn, kunnen zij toch ook vergiftigd worden, al kost dit iets meer moeite.

Dit laatste kan bijvoorbeeld met behulp van het **Internet Control Message Protocol (ICMP)** [59], een deel van het IP protocol dat gebruikt wordt voor foutenrapportering. Door een vervalst *ICMP Echo Request* pakket (uit naam van client of server) naar de te vergiftigen host (client of server) te sturen, die dan als reactie een *ICMP Echo Reply* zal terugsturen, maar hiervoor eerst een `ARP Request` zal uitvoeren, kan een aanvaller zo het juiste hardware adres te weten te komen.

Het versturen van een *ICMP Echo Request* (een '*ping*'), waarop een *ICMP Echo Reply* verwacht wordt, noemt men ook wel eens *pingen*.

6.3.3 Tegenmaatregelen

Gelukkig bestaan er een aantal tegenmaatregelen die dit soort MITM aanval kunnen voorkomen [8]:

- **Detectie**
  Een *Intrusion Detection System (IDS)*, persoonlijke firewall of speciale tools zoals *ARPwatch* [1] laten toe om verdacht ARP verkeer te detecteren, en de gebruiker te waarschuwen. In vele gevallen negeert de gebruiker zulke waarschuwingen echter, zoals we ook bij ongeldige certificaten al zagen.
- **Static ARP**
  ARP kan ook gebruikt worden op een statische manier, in plaats van dynamisch. Hierbij worden alle mappings expliciet door een administrator in de verschillende hosts geconfigureerd. Dit is echter slecht schaalbaar en moeilijk te onderhouden. Bovendien beschrijft Sean Whalen dat sommige besturingssystemen (zoals sommige Windows versies) zo'n statische ARP mappings overschrijven bij het ontvangen van een dynamisch ARP bericht, waardoor dit in bepaalde gevallen zelfs geen oplossing vormt [91]. Eigen tests tonen echter aan dat static ARP in Debian 3.0, Mandrake 9.0 en Windows XP een goed werkende tegenmaatregel vormt.
- **Secure ARP**
  Secure ARP gebruikt publieke sleutel-cryptografie om ARP berichten te authenticeren (en de integriteit te garanderen) door middel van een digitale handtekening. Dit voorkomt vervalste ARP berichten en ARP vergiftiging, ten koste van een kleine performantiekost.

In de regel zijn LAN's, en de hosts op deze LAN's echter volkomen onbeschermd tegen ARP vergiftiging aanvallen, en zal deze MITM aanval slagen.

6.3.4 Tools

Verschillende vrij verkrijgbare tools laten toe om op eenvoudige wijze ARP vergiftiging uit te voeren. De belangrijkste zijn *Ettercap* [21] en *Dsniff* [78].

6.4 DNS bedrog ('DNS spoofing')

DNS bedrog is een MITM aanval op *netwerk niveau*, en kan gebruikt worden voor een *lokale*, *lokaal naar niet-lokale* of *niet-lokale* MITM aanval.

We starten met het uitleggen van het DNS mechanisme, waarna we een MITM aanval voorstellen die hiervan gebruikt maakt.

6.4.1 DNS

Het **Domain Name System (DNS)** is een gedistribueerde database die mappings tussen *hostnames* en IP adressen bevat, om zo menselijke gebruikers toe te laten om op basis van een makkelijk te onthouden naam een host te identificeren, zonder het IP adres te kennen, dat uit een moeilijker te memoriseren cijfercombinatie bestaat.

Dit werkt als volgt:

Een gebruiker wilt via een browser een connectie maken naar een internetserver, waarvan hij enkel de hostname kent. De gebruiker zal deze hostname dan ingeven in de adresbalk van zijn browser (bijvoorbeeld "www.google.com"), waarna de browser het IP adres dat bij deze hostname behoort, opzoekt door een DNS server te raadplegen. Die DNS server raadpleegt eventueel andere DNS servers, tot de juiste mapping gevonden is, waarna hij de browser het juiste IP adres verschaft. De browser kan dan vervolgens een connectie met de server leggen op basis van dat IP adres, en de gevraagde webpagina ophalen.

6.4.2 DNS bedrog

De aanvaller kan via dit systeem een MITM aanval lanceren, door op zo'n DNS vraag van de client te antwoorden vóór de DNS server dit doet. In dit vervalste DNS antwoord zal dan een mapping van de hostname van de server naar het IP adres van de aanvaller voorgesteld worden. Hierdoor zal de client denken dat het IP adres in het DNS antwoord het IP van de server is, en al zijn pakketten naar dit adres sturen, waardoor ze bij de aanvaller terecht komen.

Indien de aanvaller zowel client als server op deze manier *DNS bedriegt*, is een MITM aanval het resultaat.

Bovendien kan de aanvaller in een meer geavanceerde DNS bedrog-aanval de *DNS server* zelf bedriegen door:

1. het antwoord van de DNS servers, die deze server op zijn beurt raadpleegt, te vervalsen

2. een *dynamische update* naar deze DNS server te sturen

Als client en server eenzelfde DNS server raadplegen, is het bedriegen van deze DNS server voldoende om een MITM aanval te lanceren, in het andere geval moet de aanvaller beide DNS servers manipuleren. Een combinatie van een *geavanceerde* DNS bedrog-aanval en een *gewone* is natuurlijk ook mogelijk.

6.4.3 Tegenmaatregelen

We stellen nu een aantal tegenmaatregelen voor die dit soort MITM aanval kunnen voorkomen [8]:

- **Detectie**
  Een IDS of persoonlijke firewall laat toe om, net als bij ARP vergiftiging, verdacht DNS verkeer te detecteren, en de gebruiker te waarschuwen. Ook hier negeert de gebruiker echter in veel gevallen de waarschuwingen.
- **Static DNS**
  Veelgebruikte hostnames kunnen in een hostfile gezet worden, waardoor geen DNS vragen meer nodig zijn.
- **DNS Security Extensions (DNSSEC)**
  Dit mechanisme is vergelijkbaar met Secure ARP. De DNS antwoorden worden digitaal gehandtekend, waardoor de authenticiteit en de integriteit gewaarborgd zijn, ten koste van een performantieverlaging.

In de regel zijn LAN's, en de hosts op deze LAN's echter volkomen onbeschermd tegen DNS aanvallen, en zal deze MITM aanval slagen.

6.4.4 Tools

Verschillende vrij verkrijgbare tools laten toe om op eenvoudige wijze DNS bedrog uit te voeren. Hieronder ook de eerder besproken *Ettercap* [21] en *Dsniff* [78] pakketten. Daarnaast is er ook nog Zodiac, dat zich uitsluitend op zwakheden in het DNS protocol concentreert.

6.5 Andere MITM aanvallen

Naast de twee eerder besproken methoden vermelden we nog kort enkele andere technieken die een MITM aanval mogelijk maken [54]:

- **Switch port stealing**
  Switches leren hardware adressen aan een bepaalde poort te verbinden door het netwerkverkeer dat door die poort gaat te observeren. Bij *switch port stealing* zal de aanvaller de switch port van de host die hij wilt aanvallen "stelen".

- **ICMP redirection**
  Met behulp van vervalste `ICMP Redirect` berichten kan een aanvaller bepaald verkeer naar zichzelf dirigeren.

- **ICMP Router Discovery Protocol (IRDP) spoofing**
  Een aanvaller kan `IRDP Router Advertisements` vervalsen, om zich ten opzichte van het slachtoffer voor te doen als de *default gateway*, de gateway waarnaar een host alle netwerkpakketten zal sturen waarmee hij zich geen raad weet.

- **Route mangling**
  De aanvaller kan zich door middel van vervalste pakketten voordoen als een router met de "goedkoopste" route naar het slachtoffer.

6.6 Toepasbaarheid: overzicht

In Tabel 6.1 geven we een overzicht van de toepasbaarheid van alle besproken MITM aanvallen [54].

Aanval	Lokaal	Lokaal naar niet-lokaal	Niet-lokaal
ARP vergiftiging	X	X	
DNS bedrog	X	X	X
Switch port stealing	X		
ICMP redirection		X	
IRDP spoofing		X	
Route mangling		X	X

Tabel 6.1. Toepasbaarheid van MITM aanvallen

6.7 Na een MITM aanval

Eens een aanvaller zich met behulp van één van de vorige technieken succesvol tussen client en server heeft geplaatst, kan hij op een aantal manieren op de connectie ingrijpen. De gevaren die zo'n MITM aanval inhoudt, worden hierna op een rijtje gezet.

- **Snuffelen of sniffing**
  Dit is de meest voor de hand liggende aanval. Aangezien alle pakketten tussen client en server langs de aanvaller worden omgeleid, is het geen enkel probleem om de inhoud van die pakketten te lezen. Deze passieve aanval noemt men *sniffing*.

- **Overnemen of hijacking**
  Een aanvaller kan een bestaande connectie (op TCP niveau) overnemen door naar één van beide partijen een `TCP FIN` bericht te sturen, en met de andere partij de communicatie verder te zetten.

- **Injecteren van pakketten**
  Een aanvaller kan pakketten in de connectie injecteren.
- **Wijzigen van pakketten**
  We gebruiken hier een ruime definitie van “wijzigen”, en bedoelen dat een aanvaller de inhoud van bepaalde pakketten kan aanpassen, of zelfs hele pakketten uit de connectie kan laten verdwijnen.

6.8 Een MITM aanval in praktijk

6.8.1 Inleiding

Laten we nu de theorie eens aan de praktijk toetsen, en een MITM aanval praktisch implementeren.

In onze praktische implementatie hebben we gekozen om ARP vergiftiging uit te voeren, waarbij we uitgaan van het eenvoudigste geval, waarbij de aanvaller zich op het zelfde lokale netwerk (LAN) als zowel client en server bevindt (een lokale MITM aanval).

Deze keuze voor ARP vergiftiging is volkomen arbitrair. We hadden evengoed DNS bedrog of nog een andere methode kunnen gebruiken.

Maar aangezien DNS bedrog in verband met SSL al eerder praktisch werd gedemonstreerd door Vuagnoux, die deze techniek gebruikte om de specifieke *CBC side-channel attack* uit de vorige sectie te implementeren [89], besloten we ARP vergiftiging toe te passen.

6.8.2 Het netwerk

Om zo’n praktische MITM aanval te demonstreren, hebben we een netwerk nodig met tenminste drie eindsystemen: een client, een server en een aanvaller.

We besloten hier geen fysiek netwerk voor te gebruiken (omwille van financiële en praktische redenen), maar een *virtueel* netwerk.

Een virtueel netwerk is een simulatie van een fysiek netwerk op één enkel fysiek systeem, waarvan het gedrag niet te onderscheiden is van dat van een gelijkaardig fysiek netwerk.

Ons virtueel netwerk bestaat uit drie virtuele eindsystemen, met de volgende details:

- **Server**
  - besturingssysteem: Debian 3.0
  - geheugen: 32 MB RAM
  - netwerk: IP = 192.168.0.104, netmask = 255.255.255.0,
    hardware adres = 00:0c:29:95:58:d1
- **Client**

  - besturingssysteem: Mandrake 9.0
  - geheugen: 64 MB RAM
  - netwerk: IP = 192.168.0.105, netmask = 255.255.255.0, hardware adres = 00:0c:29:ae:6a:25

- **Aanvaller**
  - besturingssysteem: Mandrake 9.0
  - geheugen: 128 MB RAM
  - netwerk: IP = 192.168.0.106, netmask = 255.255.255.0, hardware adres = 00:0c:29:81:fe:06

Dit gehele virtuele netwerk draait op een eindsysteem met 512 MB RAM geheugen en een Pentium-4 1,9 GHz processor.

Ook de keuze van deze details is weer volkomen arbitrair, en van weinig invloed op de praktische demonstratie van de aanval. Merk wel op dat de drie virtuele eindsystemen tot eenzelfde lokaal netwerk behoren, dit laat toe om de ARP vergiftiging aanval lokaal uit te voeren, het meest eenvoudige en doorzichtige geval.

6.8.3 ARP Vergiftiging in praktijk

In deze sectie stellen we een praktische MITM aanval voor waarbij we ARP vergiftiging uitvoeren binnen het virtuele lokale netwerk dat we in de vorige sectie samenstelden, en waarop client, server en aanvaller zich allen bevinden.

Arpspoof

We gebruiken *arpspoof* uit het *Dsniff* pakket [78].

De syntax van dit commando is als volgt:

```
arpspoof -t IP_target IP_host_to_spoof
```

met IP_target het IP adres van het slachtoffer van onze vergiftiging en IP_host_to_spoof het IP adres dat de aanvaller aan zijn eigen hardware adres wil koppelen.

In ons eigen netwerk voert de aanvaller dus de volgende twee commando's uit, om een MITM aanval uit te voeren aan de hand van ARP vergiftiging:

```
arpspoof -t 192.168.0.104 192.168.0.105
arpspoof -t 192.168.0.105 192.168.0.104
```

met 192.168.0.104 en 192.168.0.105 respectievelijk het IP adres van de server en de client

Na het uitvoeren van deze commando's zal een constante stroom van `ARP Reply` pakketten naar de client en server gestuurd worden, waardoor de client (server) denkt dat het IP adres van de server (client) bij het hardware adres van de aanvaller hoort. Alle pakketten van client naar server, en van server naar client, zullen dus naar de aanvaller gestuurd worden. De MITM aanval is geslaagd.

In Bijlage B.2 wordt een gedetailleerd verloop van zo'n MITM aanval door middel van ARP vergiftiging gegeven.

Beperkingen van arpspoof

Arpspoof werkt, maar we ondervonden toch een aantal beperkingen bij de implementatie van onze MITM aanval.

Een eerste beperking is het feit dat `arpspoof` *ARP vergiftiging* toepast in zijn pure vorm, zoals eerder beschreven. We bedoelen hiermee dat het enkel vervalste `ARP Reply`'s zal versturen, en verwacht dat het slachtoffer de informatie uit deze berichten zal gebruiken om zijn ARP cache aan te passen. We zagen echter eerder al dat dit enkel werkt als het slachtoffer al een eerdere mapping voor dit IP adres in zijn ARP cache heeft, zodat hij die kan *updaten*. Toevoegen gebeurt niet, althans niet bij de door ons geteste en gebruikte besturingssystemen.

Eerder zagen we hiervoor een oplossing: gebruik vervalste `ICMP Echo` pakketten.

Wij besloten echter om enigszins *vals* te spelen; we zorgden ervoor dat al een mapping in de caches van de client en server aanwezig was, alvorens onze MITM aanval uit te voeren, door één `ICMP Echo Request` (*ping*) van client naar server sturen, met het volgende commando:

```
ping -c 1 192.168.0.104
```

We gebruikten dus geen vervalste ICMP berichten, zoals een aanvaller zou doen, maar *echte*; dit komt in praktijk op hetzelfde neer.

Arpspoof heeft nog een ander belangrijke beperking. Deze tool maakt gebruik van *IP adressen* om de ARP vergiftiging aanval uit te voeren. Dit betekent dat het besturingssysteem van de aanvaller zelf eerst adres resolutie zal moeten toepassen, met behulp van ARP, om het hardware adres behorende bij dat IP adres te weten te komen. De aanvaller zal dus eerst zelf een onvervalst `ARP Request` uitsturen, naar het slachtoffer, waarop het slachtoffer een `ARP Reply` zal sturen, *én* de mapping tussen het hardware en IP adres van de aanvaller zal opslagen in zijn ARP cache.

Op deze manier laat de aanvaller dus sporen achter bij zijn slachtoffers. Bovendien zullen er, zodra de eigenlijke ARP vergiftiging begint, meerdere IP mappings bestaan voor eenzelfde hardware adres. De ARP implementatie zal hiertegen niet protesteren, maar een oplettende gebruiker kan zo deze de ARP vergiftiging van `arpspoof` detecteren. Eventueel kan zelfs een tool gebruikt worden die deze detectie automatiseert, zoals *ARPwatch* [1].

Dit belangrijke nadeel zou te voorkomen zijn door de slachtoffers niet te specificeren op basis van IP adres, maar op basis van hun hardware adres. Een aanvaller moet dan wel kans zien dit hardware adres te weten te komen, maar dat kan op meerdere manieren:

- **Snuffelen**
  De aanvaller kan al het verkeer op het netwerk besnuffelen door zijn netwerkkaart in zogenaamde *promiscuous* of *overspelige* mode te zetten. Op die manier kan hij het hardware adres van het slachtoffer ontdekken. Dit is eenvoudig binnen netwerken waarin de hosts verbonden zijn met behulp van *hubs*, die al het verkeer van alle hosts simpelweg over het hele (lokale) netwerk *broadcasten*. Indien in plaats van hubs, *switches* gebruikt worden, is dit moeilijker, aangezien binnen een *switched netwerk* het verkeer enkel en alleen naar de juiste bestemming wordt geleid. Er bestaan echter methodes om switches in *fail-open* te zetten[2], dat wil zeggen dat ze zich gedragen als hubs, waardoor snuffelen toch mogelijk is.

- **Andere**
  Een aanvaller kan enige tijd vóór zijn aanval een verbinding met het slachtoffer leggen, eventueel zelfs vanop een andere host, om zo het hardware adres rechtstreeks te weten te komen, en zich niet al te verdacht te maken.

  Daarnaast kunnen in specifieke gevallen ongetwijfeld nog andere methoden worden toegepast, wanneer er bijvoorbeeld een logisch verband is tussen de hardware adressen van alle hosts binnen een netwerk.

Zo'n tool die werkt op basis van hardware adressen in plaats van IP adressen zou gemakkelijk te construeren zijn, het is zelfs mogelijk om eenvoudigweg de `arpspoof` broncode aan te passen.

We besloten echter om `arpspoof` in zijn originele vorm te gebruiken, aangezien dit voor ons doel, het demonstreren van een MITM aanval, duidelijk volstaat.

6.9 Conclusie

We besluiten dat een MITM aanval een erg realistisch gevaar is.

Er is immers een brede waaier van mogelijke technieken beschikbaar die allen op een ander netwerkmechanisme inwerken om zo'n aanval tot een goed einde te brengen. Hierdoor is er een grote kans dat het netwerk op tenminste één front onvoldoende beveiligd is, en dat een aanvaller een succesvolle MITM aanval kan uitvoeren.

Daarenboven zijn er tal van tools vrij beschikbaar, waardoor een praktische implementatie van zo'n aanval erg eenvoudig is.

Een MITM aanval heeft eigenlijk maar één belangrijke beperking: de afstand tussen aanvaller en minstens één van de slachtoffers moet relatief klein zijn, zelfs bij niet-lokale aanvallen. Dit vormt in praktijk echter zelden een probleem.

[2] Het Dsniff pakket heeft hier zelfs een speciale tool voor, *macof*

Hoofdstuk 7

Versleutelsetverlagingsaanval

"In theory, there is no difference between theory and practice;

In practice, there is."

– Chuck Reid

7.1 Inleiding

In Hoofdstuk 5 werd al een korte bespreking van een *versleutelsetverlagingsaanval* gegeven. Daarbij werd gesteld dat het SSL protocol niet sterker is dan de *zwakste gemeenschappelijke versleutelset* van de twee communicerende partijen. Dit kan duidelijk niet de bedoeling geweest zijn van de SSL protocol ontwerpers, maar in welke mate vormt dit onvoorziene gedrag echt een gevaar?

Op deze vraag zullen we nu een antwoord geven. We zullen beginnen met een gedetailleerde bespreking van de voorwaarden om zo'n versleutelsetverlagingsaanval te lanceren. Vervolgens zullen we vanuit die bespreking een praktische aanpak destilleren om zo'n aanval te implementeren, waarna we deze implementatie ook echt demonstreren.

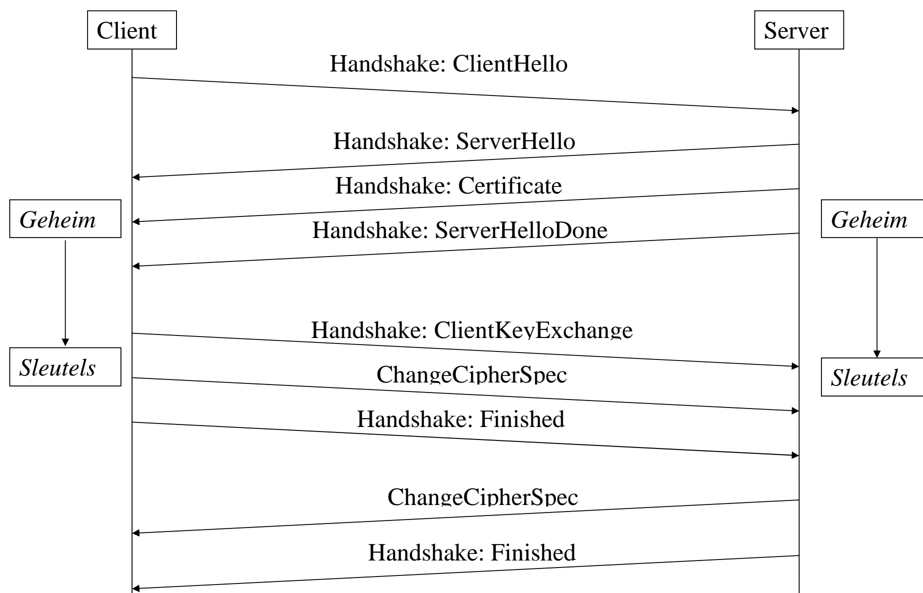
We besluiten met een conclusie die antwoordt op de vraag: "Vanuit onze ervaring met het implementeren van een versleutelsetverlagingsaanval, hoe haalbaar is dit voor een aanvaller?"

7.2 Overzicht

In een versleutelsetverlagingsaanval zal de aanvaller proberen om client en server een bepaalde versleutelset op te dringen. Hiervoor zal hij de SSL handshake moeten manipuleren.

SSL handshake

We geven nog eens een gedetailleerd overzicht van een normale SSL Handshake in Figuur 7.1.



Figuur 7.1. De SSL Handshake herbekeken

Een uitgebreide bespreking van de werking van de handshake en de gebruikte berichten zullen we niet meer herhalen, hiervoor verwijzen we naar Hoofdstuk 3. We behandelen hier enkel de berichten die van toepassing zijn op de versleutelsetverlagingsaanval.

De aanval

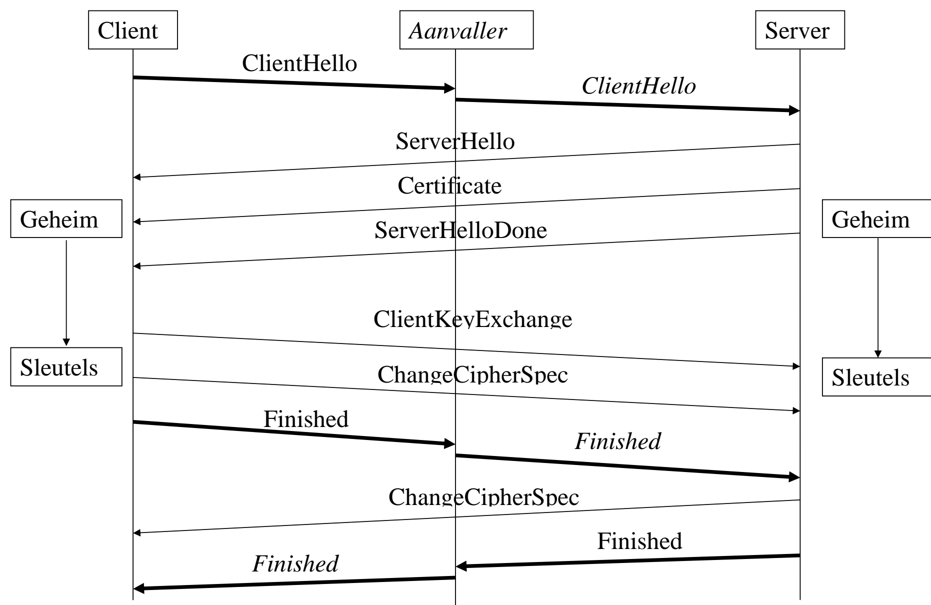
Een aanvaller die een versleutelsetverlaging wil uitvoeren, zal drie berichten moeten aanpassen:

- **ClientHello**
  In het `ClientHello` bericht wordt een lijst van versleutelsets meegestuurd die de client bereid is te ondersteunen. Een aanvaller die zijn eigen versleutelset wil opleggen, zal deze lijst vervangen door één of meerdere zelfgekozen versleutelsets.
- **Finished (tweemaal)**
  Aangezien de aanvaller de `ClientHello` zal aanpassen, wordt aan de integriteit van de handshake geraakt, en zal de aanvaller ook de integriteitscontrole moeten vervalsen. Dit komt neer op het vervalsen van de twee `Finished` berichten.

Een aanvaller zal dus de handshake tussen client en server moeten manipuleren, en deze berichten vervangen door zijn eigen, vervalste versies. Een *man-in-the-middle (MITM) aanval* is hiervoor het perfecte instrument, zoals we in Hoofdstuk 6 zagen.

Versleutelsetverlaging: SSL handshake aangevallen

Een aanval op de SSL handshake waarbij de aanvaller zich eerst, door een MITM aanval uit te voeren, tussen client en server plaatst, om vervolgens de versleutelsetverlagingsaanval uit te voeren, ziet er uit als in Figuur 7.2.



Figuur 7.2. Versleutelsetverlagingsaanval op de SSL Handshake

Op de figuur is duidelijk te zien dat de `ClientHello` en de `Finished` berichten gewijzigd worden door de aanvaller. Daarnaast blijven alle andere berichten onaangeroerd.

In de volgende secties wordt achtereenvolgens het vervalsen van de `ClientHello` en de `Finished` berichten besproken.

7.3 ClientHello vervalsen

De `ClientHello` vervalsen is geen enkel probleem. Dit bericht wordt in plaintext verstuurd, zonder enige rechtstreekse beveiliging. Een aanvaller kan eenvoudigweg de lijst versleutelsets in dit bericht vervangen door zijn eigen lijst.

De `ClientHello` is wel onrechtstreeks beveiligd door middel van de MAC's over de volledige handshake, in de `Finished` berichten. Het vervalsen van de `Finished` berichten (en de MAC's) wordt aangepakt in de volgende sectie.

7.4 Finished vervalsen

De `Finished` berichten blijken wat moeilijker. Deze zijn immers wel beveiligd, want gestuurd over een veilige connectie. Dat wil zeggen dat deze berichten (symmetrisch) geëncrypteerd zijn (vertrouwelijkheid), en dat hun integriteit gewaarborgd is met behulp van een MAC. Daarnaast bestaat de inhoud van de `Finished` berichten ook nog eens uit twee MAC's, beide over alle voorgaande handshake berichten.

Samengevat staan we voor drie problemen bij het vervalsen van een `Finished` bericht:

- **Symmetrische encryptie**
  Het vervalste `Finished` bericht van de aanvaller moet geëncrypteerd worden onder de juiste symmetrische sleutel. Een aanvaller moet hiervoor maximum vijf dingen te weten komen:
  - overeengekomen symmetrische encryptie-algoritme
  - `client_write_key`
  - `server_write_key`
  - `client_IV` (bij block ciphers)
  - `server_IV` (bij block ciphers)
- **MAC over Finished zelf**
  Voor het vervalsen van de MAC's van beide `Finished` berichten moet de aanvaller de volgende zaken op één of andere manier verkrijgen:
  - overeengekomen digest algoritme
  - `client_MAC_key`
  - `server_MAC_key`
- **MAC's over handshake**
  Om deze vervalsing uit te voeren moeten we volgende elementen te weten komen:
  - `master_secret`
  - voorgaande handshake berichten
  - volgnummer van de berichten

We merken bovendien op dat de twee laatste problemen afhankelijk zijn van het eerste. Zowel de MAC over een `Finished` bericht, als de MAC's over de handshake die in de **Finished** berichten zitten, zijn geëncrypteerd met behulp van symmetrische encryptie, dus om deze MAC's te vervalsen moet de symmetrische encryptie gebroken zijn.

Het ontdekken van het overeengekomen symmetrische encryptie- en digestalgoritme vormt geen enkel probleem. Een aanvaller kan (omwille van de MITM aanval) de connectie

immers “besnuffelen”, en uit de inhoud van de ServerHello afleiden welke versleutelset (en daaruit volgend, welke cryptografische algoritmen) beide partijen overeen zijn gekomen. Ook de uitgewisselde handshake berichten kunnen op deze “snuffelwijze” gelezen worden door een aanvaller. Door zelf eigen tellers bij te houden kan de aanvaller bovendien ook de volgnummers berekenen.

De andere elementen vormen echter een groter probleem, aangezien zij allen strikt geheim zijn. Zij worden nu besproken.

7.4.1 Symmetrische encryptiesleutels

Bruteforcen

Een eerste manier om de symmetrische encryptiesleutels te verkrijgen bestaat uit het *bruteforcen* van de originele Finished berichten. Dit moet dan wel op heel korte tijd (quasi onmiddellijk) kunnen gebeuren, aangezien de Finished berichten onmiddellijk vervalst moeten worden, anders zal de connectie afgebroken worden omwille van een *time-out* of een ongeduldige gebruiker. In OpenSSL is de standaard time-out waarde 300 seconden, dus we veronderstellen dat het geduld van de gebruiker de bepalende factor is (want korter dan 300 seconden).

In Sectie 5.9.3 zagen we dat dit *real-time bruteforcen* mogelijk is indien de gebruikte sleutel kort genoeg is, en de investeringsbereidheid van de aanvaller hoog genoeg.

We zagen daar dat een symmetrische 40 bit sleutel in 0,1 seconden gebroken kan worden, voor een kost van 1 miljoen dollar. Dit is snel genoeg om een time-out te voorkomen, en geen enkele gebruiker zal zo’n kleine vertraging opmerken. Maar zelfs voor 12 000 dollar is het mogelijk zo’n sleutel in minder dan 10 seconden te breken, en we veronderstellen dat ook zo’n time-out nog goed verdragen wordt door de meeste gebruikers.

De eis dat de sleutel kort genoeg moet zijn, is daarbij geen probleem[1], aangezien de aanvaller de gebruikte versleutelset bepaalt, en op deze manier ook het gebruikte symmetrische algoritme en de sleutellengte.

Afleiden uit pre_master_secret

Daarnaast kunnen deze sleutels ook verkregen worden door de pre_master_secret te compromitteren, waarna hieruit de sleutels kunnen afgeleid worden met behulp van de sleutelafleidingsfunctie. Hiervoor zijn de client_random en server_random waarden nodig, maar ook deze plaintext waarden kunnen op een eenvoudige manier, met “gesnuffel” uit de ClientHello en ServerHello gehaald worden.

Merk op dat de sleutelafleidingsfunctie verschilt tussen SSLv3 en TLS. Door de ServerHello te “besnuffelen” kunnen we echter op triviale wijze te weten komen welke functie gebruikt wordt. Bovendien kunnen we de gebruikte versie (SSLv3 of TLS) zelf bepalen door de versienegotiatie in de ClientHello te manipuleren.

[1] Indien de client versleutelsets met zulke korte sleutels ondersteunt.

De pre_master_secret wordt (asymmetrisch) geëncrypteerd onder een publieke sleutel, in het ClientKeyExchange bericht. In Hoofdstuk 5 zagen we een aantal aanvallen die deze encryptie zouden kunnen compromitteren:

- **Onrechtstreekse aanval**
  We zagen een aantal onrechtstreekse SSL aanvallen (bleichenbacher en afgeleiden) die zich specifiek richten op het decrypteren van zulke asymmetrisch geëncrypteerde ciphertexts. In praktijk blijkt echter dat elke implementatie hiervoor ondertussen immuun gemaakt is, en is zo'n aanval niet mogelijk, zoals we in Hoofdstuk 5 aantoonden.

  Daarnaast zagen we ook enkele *side-channel attacks* (met als belangrijkste de timing aanval) die de private sleutel trachten te compromitteren. Maar ook deze zijn in praktijk onmogelijk omwille van de tegenmaatregelen in recente implementaties.

- **Frontale aanval**
  Het is ook mogelijk om een frontale aanval op de asymmetrische encryptie uit te voeren, en op die manier de private sleutel te weten te komen. Zo'n bruteforce aanval, uitgevoerd op een RSA 512 bit sleutel (de zwakst mogelijke asymmetrische sleutellengte in SSL), bleek te kunnen op één dag, indien de aanvaller hier 622 500 dollar voor over heeft.

  Een andere frontale aanval bestaat uit *bruteforce guessing* van de pre_master_secret zelf. Aangezien de pre_master_secret bestaat uit 48 bytes, waarvan 46 geheim en random, komt dit overeen met het *bruteforcen* van een symmetrische sleutel van 368 bits lang. Dit is computationeel onmogelijk, zoals we eerder zagen.

We besluiten dus dat het compromitteren van de pre_master_secret in praktijk neerkomt op het *bruteforcen* van de private sleutel, gebruikt om de ClientKeyExchange te decrypteren.

We merken hierbij op dat een aanvaller bij een aanval op zo'n private sleutel de tijd wel aan zijn kant heeft (in tegenstelling tot bij de symmetrische sleutels), aangezien een server dezelfde private sleutel gedurende lange tijd (typisch jaren) gebruikt, zelfs in geval van Ephemeral RSA, waar dit typisch weken of maanden is.

7.4.2 MAC over Finished

Omwille van de eerder vermelde afhankelijkheid tussen de MAC's en de symmetrische encryptie, veronderstellen we dat de symmetrische encryptie op de een of andere manier gebroken is.

Bruteforcen

Om de MAC sleutels te compromitteren, kunnen we de MAC's, in feite message digests, proberen om te keren door middel van een frontale aanval. Zoals we eerder zagen komt dit overeen met het bruteforcen van een symmetrische sleutel met een lengte gelijk aan de MAC sleutel. In het zwakste geval, bij de export versleutelsets, is zo'n sleutel 16 byte, of 128 bit lang [24]. Dit is computationeel onmogelijk te bruteforcen.

Afleiden uit `pre_master_secret`

Door de `pre_master_secret` te compromitteren, zoals uitgelegd in de vorige sectie, kan echter ook deze geheime informatie op eenvoudige wijze afgeleid worden door de aanvaller met behulp van de sleutelafleidingsfunctie.

7.4.3 MAC over handshake

Ook voor deze MAC's veronderstellen we weer dat een aanvaller er ten eerste al in geslaagd is om de symmetrische encryptie te breken.

Bruteforcen

Ook hier kan aanvaller een poging wagen om de MAC's in de `Finished` berichten om te keren door middel van *bruteforcing.* Deze MAC's zijn gebaseerd op de `master_secret`, een geheim van 48 bytes. Het bruteforcen van deze MAC's komt dus neer op het bruteforcen van een 384 bit symmetrische sleutel, en is duidelijk computationeel onmogelijk.

Afleiden uit `pre_master_secret`

Ook hier is het echter mogelijk de benodigde geheime informatie (onder de vorm van de `master_secret`) af te leiden uit de `pre_master_secret`, zoals we in Hoofdstuk 4 zagen.

7.4.4 Conclusie

We besluiten dat er eigenlijk maar één haalbare strategie is om de `Finished` berichten te vervalsen, en dat is door de private sleutel te *bruteforcen.* Hierdoor kunnen we de `pre_master_secret` te weten komen, waarmee we vervolgens, via de sleutelafleidingsfunctie, al het geheime sleutelmateriaal kunnen afleiden.

Met andere woorden, we hebben de veiligheid van SSL van "veiligheid zwakste gemeenschappelijke versleutelset" afgezwakt tot "*veiligheid zwakste gemeenschappelijke asymmetrische encryptie*".

Dit is belangrijk aangezien we zowel de *sleutellengte* als het *algoritme* van de asymmetrische encryptie kunnen beïnvloeden, door middel van een handshake manipulatie die ingrijpt op de versleutelsetnegotiatie.

Strikt gezien is de sleutellengte van de asymmetrische encryptie *niet* beïnvloedbaar, want volledig de verantwoordelijkheid van de server, zoals we eerder zagen. Een aanvaller die de handshake echter zodanig kan manipuleren dat een server overgaat op Ephemeral RSA, dwingt de server zo om een vastgestelde (zwakkere) sleutellengte (512 bit) te gebruiken. Het is zo'n beïnvloeding van de sleutellengte waar we op doelen.

7.5 Voorwaarden voor een succesvolle versleutelsetverlaging

We besluiten dat een versleutelsetverlagingsaanval wel degelijk mogelijk is, al is deze gebonden aan de volgende voorwaarden:

- **MITM aanval mogelijk**
  De aanvaller is in staat zich tussen client en server te plaatsen.
- **Zwakke asymmetrische encryptie**
  De aanvaller heeft de mogelijkheid om de *zwakste gemeenschappelijke asymmetrische encryptie* te kraken.

In de volgende sectie zullen deze voorwaarden verder uitgewerkt worden, om vervolgens een praktische implementatie van de versleutelsetverlagingsaanval te demonstreren.

7.6 De aanval in praktijk: voorbereiding

In de vorige sectie gaven we een uitgebreide analyse van een versleutelsetverlagingsaanval, waaruit we concludeerden dat deze praktisch mogelijk moet zijn, onder twee voorwaarden.

Laten we nu de theorie aan de praktijk toetsen, en deze aanval praktisch implementeren. Als eerste definiëren we hiervoor een testomgeving. Vervolgens zullen we bespreken hoe we aan de twee eerder vermelde voorwaarden zullen voldoen.

7.6.1 Het netwerk

Als testnetwerk zullen we hetzelfde virtuele netwerk gebruiken als datgene gebruikt om de ARP vergiftiging aanval te demonstreren. Binnen dat netwerk zullen we gebruik maken van een OpenSSL server (`openssl s_server`) en OpenSSL client (`openssl s_client`) om de SSL connectie te leggen.

7.6.2 Voorwaarden aangepakt

MITM aanval

Een praktische MITM aanval door middel van ARP vergiftiging werd in Sectie 6.8 behandeld. We zullen deze aanval dan ook gebruiken in onze versleutelsetverlagingsaanval.

Elke MITM aanvalstechniek is echter mogelijk, de eigenlijke versleutelsetverlagingsaanval is onafhankelijk van de gebruikte methode.

Zwakke asymmetrische encryptie

Met de voorwaarde van "zwakke asymmetrische encryptie" bedoelen we dat het mogelijk moet zijn voor de aanvaller om de `pre_master_secret` uit het `ClientKeyExchange` bericht te extraheren. We toonden eerder aan dat dit enkel mogelijk is met behulp van een *bruteforce* aanval.

In onze praktische aanval zullen we uitgaan van de stelling dat zowel client als server een export versleutelset ondersteunen, meer bepaald de `TLS_RSA_EXPORT_WITH_RC4_40_-MD5` versleutelset die het nummer 0x0003 draagt. Een aanvaller die dan beide partijen kan overtuigen (door middel van een versleutelsetverlagingsaanval) om deze export versleutelset te gebruiken, verplicht server en client zo om Ephemeral RSA te gebruiken, waardoor de `pre_master_secret` slechts beveiligd zal worden met een tijdelijke 512-bit RSA sleutel. Dit is het ideale geval voor een aanvaller, aangezien dit de zwakst mogelijke asymmetrische encryptie is in het SSL protocol.

Merk op dat *elke* export versleutelset zou volstaan, aangezien deze allen een asymmetrische encryptie gebruiken van vergelijkbare (zwakke) kracht[2]. We kozen echter specifiek voor `TLS_RSA_EXPORT_WITH_RC4_40_MD5` aangezien deze het meest gebruikt wordt, zoals we later zullen aantonen.

De zwakke sleutel van de gebruikte asymmetrische encryptie kan de aanvaller vervolgens *bruteforcen*. Zoals we eerder berekenden is het, zelfs voor een erg welgestelde aanvaller, quasi onmogelijk om zo'n sleutel *onmiddellijk* te breken, dat wil zeggen, snel genoeg om nog diezelfde connectie tot een goed einde te brengen (door de `Finished` berichten te vervalsen).

We zagen echter in de bespreking van Ephemeral RSA dat de tijdelijkheid van zo'n "tijdelijke" sleutel erg relatief is, dus een aanvaller kan rustig de tijd nemen om de sleutel te *bruteforcen*.

Hoeveel tijd zo'n aanvaller hiervoor kan nemen, hangt af van twee factoren:

- **Tijdelijkheid van de sleutel**
  Eerder zagen we dat de meeste server implementaties dezelfde tijdelijke sleutel gedurende weken tot maanden gebruiken, aangezien deze pas wordt veranderd bij een herstart.
- **Middelen van de aanvaller**
  Een aanvaller met voldoende middelen kan zo'n sleutel binnen een dag kraken, zoals we eerder zagen. Een aanvaller met minder middelen zal hier een week, een maand, of nog langer voor nodig hebben.

Het is duidelijk dat een aanval enkel mogelijk is indien de tijd die de aanvaller nodig heeft voor het kraken van de sleutel *korter* is dan de tijdelijkheid van deze sleutel.

Dit veronderstelt wel dat een aanvaller voldoende materiaal heeft verzameld om de sleutel te kunnen breken. Dat vormt geen enkel probleem, aangezien het `ClientKeyExchange`

[2]Met uitzondering van de EXPORT1024 versleutelsets, die een ephemeral sleutel van 1024 bit hanteren. Aangezien deze versleutelsets echter niet voldoen aan de originele exportreguleringen, beschouwen we deze niet als export versleutelsets.

bericht op zich hiervoor voldoende informatie bevat. Een aanvaller kan op drie manieren zo'n `ClientKeyExchange`, geëncrypteerd met een tijdelijke RSA sleutel, te weten komen:

- **Zelf een export connectie leggen**
  Dit verraadt echter de echte identiteit van de aanvaller, en is niet aan te raden.
- **Een export connectie besnuffelen**
  Een aanvaller kan MITM aanvallen op de server uitvoeren, tot hij er op deze manier in slaagt een export connectie langs zichzelf te herleiden, waaruit hij dan de nodige informatie simpelweg kan aflezen. Dit vraagt een betrekkelijk grote inspanning van de aanvaller, afhankelijk van het aantal export connecties waar de server mee te maken krijgt. Hoe groter dit aantal, hoe groter de kans dat een willekeurige connectie een export connectie is, en hoe makkelijker voor de aanvaller.
- **Een poging tot versleutelsetverlaging op een willekeurige connectie**
  Een aanvaller kan een versleutelsetverlaging uitvoeren op een willekeurige connectie. Aangezien hij nog niet beschikt over de private sleutel, zal hij er niet in slagen om deze versleutelsetverlaging tot een goed einde te brengen, maar de connectie zal pas afgebroken worden *nadat* de aanvaller de `ClientKeyExchange` heeft onderschept. Deze methode is de meest optimale voor de aanvaller.

7.7 Implementatie van de versleutelsetverlagingsaanval

Nu we een duidelijk beeld hebben van de theoretische voorwaarden van een versleutelsetverlagingsaanval, en deze zo goed mogelijk hebben proberen te vertalen naar een meer specifieke, praktische aanval, zullen we met deze informatie zo'n praktische aanval implementeren.

De aanval zal bestaan uit het manipuleren van de SSL handshake, waarbij de `ClientHello` en de `Finished` berichten uit de connectie worden gewijzigd, en de inhoud van de andere berichten gebruikt wordt om deze aanpassingen uit te voeren.

Op netwerkniveau zal deze handshake manipulatie bestaan uit het manipuleren van netwerkpakketten, op de volgende manier:

- **Vangen**
  We zullen alle pakketten die omwille van de MITM aanval naar ons toegestuurd worden, en uitsluitend deze pakketten, vangen voor verdere verwerking.
- **Voorverwerken**
  Pakketten die van het netwerk gehaald worden in de vorige stap moeten enigszins voorverwerkt worden alvorens hun inhoud geïnterpreteerd kan worden, zoals we verder zullen zien.
- **Besnuffelen**
  We zullen de inhoud van bepaalde pakketten inkijken om informatie af te leiden waarmee we dan vervolgens de eigenlijke aanval zullen implementeren.

- **Wijzigen**
  Bepaalde pakketten zullen gewijzigd worden.
- **Versturen**
  De pakketten die we onderschept hebben met behulp van de MITM aanval zullen we, al dan niet na wijziging, naar hun oorspronkelijke bestemming sturen.

Alvorens we de implementatie van deze stappen uitgebreid bespreken, stellen we enkele softwarebibliotheken voor die bij de implementatie van deze stappen kunnen helpen.

7.7.1 Gebruikte softwarebibliotheken

Voor het ontwerpen van netwerkbeveiligingstools zijn er verschillende softwarebibliotheken beschikbaar, die de implementatie van zo'n tool sterk vereenvoudigen.

Onze tool zal gebruikmaken van de volgende drie *open-source software* bibliotheken:

- **Libpcap-0.7.2**
  Deze bibliotheek bezit functies voor het vangen van pakketten ("*packet capturing*"). Daarenboven kan gespecificeerd worden welke pakketten gevangen moeten worden, door een *filter* in te stellen ("*packet filtering*").
- **Libnet-1.1.x**
  Aan de hand van deze bibliotheek kan een aanvaller netwerkpakketten construeren ("*packet construction*") en deze in het netwerk injecteren ("*packet injection*").
- **OpenSSL-0.9.7c**
  Deze bibliotheek omvat eigenlijk twee bibliotheken. De eerste vormt een toolkit voor het implementeren van het SSL protocol. De tweede bibliotheek biedt allerlei cryptografische functies aan. We zullen bijna uitsluitend deze tweede bibliotheek gebruiken.

Het lijkt wat raar dat we, voor het implementeren van een SSL aanvalstool, geen gebruik maken van de SSL toolkit die de OpenSSL bibliotheek aanbiedt, daarom verduidelijken we nu deze keuze.

Het doel van onze tool is het vervalsen van SSL berichten. Met andere woorden, we willen, los van andere berichten, nieuwe berichten creëren, aan de hand van volledig op zichzelf staande parameters. We willen op een *dynamische* manier *eender welk* SSL bericht kunnen genereren, aan de hand van *willekeurige* informatie.

Natuurlijk is dit niet de functionaliteit waarvoor de SSL toolkit in de OpenSSL bibliotheek ontworpen is, deze toolkit verwacht juist dat een gebruiker de verschillende SSL berichten afhankelijk van elkaar en op een sequentiële manier wil genereren, om tot een normale SSL connectie te komen. Daarom is deze toolkit grotendeels gebaseerd op één enkel object, het *SSL Context* object. Alle publieke SSL functies die deze toolkit ter beschikking stelt, werken op basis van deze generische structuur die alle informatie (staat van de connectie, cryptografische informatie, ...) bevat. Hierbij is het gebruik van dit context object een sterke weerspiegeling van het eigenlijke, sequentiële verloop van een SSL connectie. Om

deze reden maakt het gebruik van dit object het erg moeilijk om op zichzelf staande SSL berichten te produceren, op basis van specifieke informatie.

We beslissen dus om onze eigen C functies te implementeren voor het genereren van SSL berichten, waarbij we ons wel baseren op de overeenkomstige SSL toolkit functies en zoveel mogelijk gebruikmaken van de functies die de cryptografische bibliotheek van OpenSSL ter beschikking stelt. Een vergroot inzicht in de eigenlijke werking van SSL en zijn cryptografische functies is hierbij mooi meegenomen.

Meer informatie over het gebruik van `Libpcap` [45], `Libnet` [69] en OpenSSL [52] kan gevonden worden in [70].

We gaan nu dieper in op de implementatie van de verschillende netwerkpakketmanipulaties die nodig zijn om onze aanval te doen slagen.

7.7.2 Pakketten vangen

Voor het vangen van pakketten zullen we gebruik maken van de `Libpcap` bibliotheek.

Hierbij is het belangrijk om enkel die pakketten te vangen die belangrijk zijn in onze aanval. Dit bevordert de efficiëntie van de aanval en vermijdt een overbelasting van onze "pakketvanger" waardoor bepaalde belangrijke pakketten verloren zouden kunnen gaan.

De filter

Het is dus belangrijk om zo duidelijk mogelijk te definiëren welke pakketten gevangen moeten worden. Dit doet men met behulp van een filter. In `Libpcap` wordt zo'n filter in *tcpdump syntax* [86] aangegeven.

Aangezien de aanvaller enkel geïnteresseerd is in pakketten van de SSL connectie tussen server en client, zal de filter er als volgt uit zien:

> `ip and port 443 and ether dst` *`hardware_adres_aanvaller`* `and not dst` *`IP_aanvaller`* `and not ip broadcast`
>
> met IP_aanvaller het IP adres van de aanvaller, en hardware_adres_aanvaller het hardware adres van de aanvaller

Met andere woorden, de aanvaller zal enkel IP pakketten (`'ip'`) accepteren die verstuurd worden over een SSL connectie (`'port 443'`) en die omwille van de ARP vergiftiging naar zijn hardware adres gestuurd worden (`'ether dst` *`hardware_adres_aanvaller`*`'`) maar eigenlijk niet voor hem bestemd zijn (`'not dst` *`IP_aanvaller`* `and not ip broadcast'`).

Hierbij merken we op dat SSL pakketten onderscheiden worden aan de hand van hun poortnummer, dit is standaard 443 voor SSL connecties. Natuurlijk zijn ook andere poorten mogelijk, in deze gevallen moet de filter aangepast worden.

Een alternatief filtervoorstel

Op het eerste zicht lijkt het mogelijk om een meer generisch toepasbare filter (zonder te steunen op een variabel poortnummer) te maken aan de hand van zogenaamde *byte offset filters*. Dit zijn *tcpdump syntax* filters waarbij gefilterd kan worden, op om het even welke waarde, op om het even welke plaats, in bijvoorbeeld een TCP pakket. We zouden zo'n een filter kunnen maken die SSL pakketten herkent, op de volgende manier:

```
tcp[20] = 0x14 or tcp[20] = 0x15 or tcp[20] = 0x16 or tcp[20] = 0x17
or tcp[20] = 0x80
```

Hierbij is `tcp[20]` de 21e byte (0 is de eerste byte) in het TCP pakket, dat is de eerste byte van de *payload*, de eigenlijke inhoud van het TCP pakket die na de *TCP header* volgt. We vergelijken de inhoud van deze byte met de waarde die het SSLv3 Record Protocol op die plaats kan aannemen naargelang het inhoudstype (`0x80 = SSLv2 pakket, 0x14 = change_cipher_spec, 0x15 = alert, 0x16 = handshake, 0x23 = application_data`), of de waarde die een SSLv2 `ClientHello` in die byte zou aannemen (`0x80`). Dit laatste is nodig aangezien elke SSL versie die *backward compatible* is met SSLv2 een SSLv2 `ClientHello` verstuurd.

Deze filter zal duidelijk alle SSL pakketten vangen, en uitsluitend deze.

Hierbij laten we na een onderscheid te maken tussen TCP pakketten mét opties, en zonder opties. Een TCP pakket met opties zal op de gecontroleerde plaatsen opties bevatten, waardoor de filter deze pakketten niet zal doorlaten, hoewel ze mogelijk wel SSL informatie bevatten. Dit is echter op te lossen met een meer generische filter die ook *alle* pakketten met TCP optie-informatie doorlaat, door het volgende toe te voegen:

```
or tcp[12] & 0xf0 > 5
```

Deze filter gaat na of de *high-order nibble* (de vier meest significante bits van een byte) van de 13e TCP byte groter dan 5 is, met behulp van een techniek die men *bit masking* noemt (`& 0xf0`). Deze *nibble* geeft het aantal 32-bit *words* aan in de TCP header, dus 5 betekent een normale TCP header van 20 bytes, een waarde groter dan 5 duidt op een TCP header met extra opties.

Deze gehele filter zal dan alle TCP pakketten met SSL inhoud doorlaten, met daarnaast ook nog alle TCP pakketten met TCP opties, die we dan extra kunnen controleren bij de verdere verwerking zelf.

Toch is dit niet voldoende voor onze aanval, aangezien we niet uitsluitend de SSL pakketten willen vangen, maar ook de pure TCP pakketten die geen SSL informatie bevatten maar wel gebruikt worden om de TCP connectie waarop de SSL connectie steunt, op te zetten en te onderhouden. Deze pakketten moeten immers ook doorgestuurd worden naar hun oorspronkelijke bestemming. Deze pakketten zijn echter enkel te onderscheiden van andere TCP pakketten aan de hand van hun poortnummer.

Het blijkt dus dat we uiteindelijk toch aan de hand van het SSL poortnummer moeten filteren, daarom hebben we voor de eerste, eenvoudige filter gekozen, zonder *byte offset* notaties.

Opmerking

Merk op dat door deze filter te gebruiken enkel pakketten behorende tot een SSL connectie verwerkt en doorgestuurd worden. Indien server en client daarnaast nog op een andere manier met elkaar communiceren, die ook beïnvloed wordt door de MITM aanval waardoor ook dit verkeer wordt omgeleid langs de aanvaller, zal deze communicatie falen, aangezien de pakketten van deze communicatie niet doorgestuurd worden. Dit zou eenvoudig op te lossen zijn door de poortspecificatie uit de filter te laten vallen, waardoor *alle* omgeleide IP pakketten door de aanvaller verwerkt en doorgestuurd worden.

We kiezen echter voor de iets minder generische filter mét poortnummer, aangezien dat garandeert dat we onder alle omstandigheden uitsluitend SSL-gerelateerde pakketten vangen, wat precies is wat we willen en verwachten. Bovendien zagen we eerder al dat het *best practice* is om uitsluitend pakketten te vangen die nodig zijn.

7.7.3 Pakketten voorverwerken

Alvorens we de inhoud van de netwerkpakketten kunnen interpreteren, is er een voorverwerking nodig, om van een bytesequentie naar een netwerkpakket te gaan.

Van netwerk naar host

De pakketten die we met behulp van onze filter hebben opgevangen, worden door `Libpcap` in *bytevorm* doorgegeven. Hierbij is het belangrijk om te weten of de byte volgorde *big endian* of *little endian* is. In *big endian* volgorde wordt eerst de meest significante byte doorgegeven, in *little endian* is dit juist andersom[3].

Bytes op het netwerk zijn altijd *big endian*, om deze reden wordt *big endian order* ook wel eens *network byte order* genoemd.

Verschillende processor architecturen gebruiken echter verschillende formaten, zo gebruikt Intel een *little endian*-bytevolgorde. Om deze reden is het belangrijk om de bytestroom altijd van *netwerkbytevolgorde* ('*network byte order*') naar hostbytevolgorde ('*host byte order*') om te zetten, alvorens verder te verwerken. Dit proces wordt vergemakkelijkt door een aantal macro's[4] die in elke standaard C implementatie aanwezig zijn, en die deze omzetting transparant voor hun rekening nemen.

Nadat op deze manier de bytevolgorde van de bytestroom op de juiste manier geïnterpreteerd werd, zal de inhoud van deze aaneenschakeling van bytes moeten geïnterpreteerd worden. Met andere woorden, deze bytestroom moet omgezet worden in herkenbare netwerkpakketten, waarna de inhoud verder verwerkt kan worden. Deze omzetting van bytestroom naar pakket wordt nu besproken.

[3]De begrippen big-endian en little-endian zijn gebaseerd op Jonathan Swift's *Gulliver's Travels* waarin de Big Endians een politieke fractie waren die hun eieren langs de dikke kant opensneden ("de primitieve manier") en zo rebelleerden tegen de Lilliputiaanse koning die zijn onderdanen (de Little Endians) verplichtte om de eieren te breken aan de smalle kant.

[4]htonl(), htons(), ntohl() en ntohs()

Bytestroom naar pakket

Een pakket bestaat uit verschillende velden. In programmeertalen wordt zo'n object het best voorgesteld met behulp van een klasse, of een structuur. In de programmeertaal ***C*** wordt hiervoor het ***`struct`***-mechanisme gebruikt.

Een belangrijke eigenschap van C-structs is het feit dat de elementen (velden) van de structuur op byteniveau in de volgorde worden opgeslagen waarin ze gedeclareerd worden. Een compiler mag deze velden niet herordenen. Wel mag een compiler, om de efficiëntie te verhogen, *padding bytes* toevoegen tussen de elementen [19]. Deze eigenschap zal erg nuttig blijken, zoals we dadelijk zullen zien.

`Libnet` biedt standaard C-structs aan voor TCP en IP pakketten, maar voor de SSL pakketten moeten we deze zelf construeren op basis van de SSL specificatie ([24]).

Eens die structuren geconstrueerd, zijn er twee manieren om de bytestroom in deze structuren te gieten:

- Lees de bytestroom byte voor byte, en wijs telkens de inhoud van de byte toe aan het overeenkomstige struct-veldje.
- Leg de struct als het ware *over* de bytestroom, waarbij elke byte precies op de juiste plaats "valt". Deze methode maakt gebruik van de eerder vernoemde eigenschap van C-structs, en gebeurt met behulp van *pointers*.

De eerste methode is robuuster, laat meer controle toe, en is om deze redenen de aangeraden manier vanuit software-engineering oogpunt [19].

De netwerkbeveiligingswereld heeft zo echter zijn eigen voorkeuren, en in praktijk blijkt daarin vooral de tweede methode gebruikt te worden [70].

We proberen zoveel mogelijk de netwerkbeveiligingsconventies te volgen, dus we passen de tweede methode toe.

We illustreren deze werkwijze met een voorbeeld, waarin we de bytestroom van een `ChangeCipherSpec` bericht in een TCP pakket inlezen in een SSLv3 Record Header structuur, om het inhoudstype te ontdekken:

> Onze zelf gedefinieerde C-struct voor een SSLv3 Record Header ziet eruit als volgt:

```
struct sslv3_record_hdr {

    u_int8_t ssl_t;         /* type */

    u_int8_t ssl_v[2];      /* version */

    u_int8_t ssl_len[2];          /* total length */

};
```

De bytestroom van een TCP pakket met een `ChangeCipherSpec` inhoud is van de volgende vorm, waarbij de eerste regel de TCP header voorstelt, en het **vetgedrukte** deel de SSLv3 Record Header:

01 bb 04 7b df 7f 63 3b cd e2 00 37 50 18 1f fe e0 3b 00 00
14 03 00 00 01 01

Indien deze bytestroom door de `Libpcap` bibliotheek doorgegeven wordt met behulp van de `char pointer` *pakket*, kan de inhoud van deze bytestroom als volgt in een `sslv3_record_hdr` struct geplaatst worden:

```
char* pakket;

... //lees ChangeCipherSpec bytestroom in in pakket

struct libnet_tcp_hdr *tcph = (struct libnet_tcp_hdr *)(pakket);

struct sslv3_record_hdr *sslv3_rec_h =

(struct sslv3_record_hdr*)(pakket + TCP_H);
```

`TCP_H` is hier de lengte van de TCP header en `libnet_tcp_hdr` is een C-struct uit de `Libnet` bibliotheek.

De informatie uit de bytestroom zit nu in de record klasse. Het inhoudstype van het SSL Record kan dan op de volgende manier gebruikt worden:

```
sslv3_rec_h->ssl_t
```

Belangrijk hier is wel dat de struct *precies* op de bytestroom past, anders is de informatie in de struct betekenisloos, omdat verkeerde bytes in verkeerde velden terechtkomen.

Het is dus belangrijk om de struct zo te construeren dat er geen padding bytes tussen de verschillende veldjes worden ingevoegd door de compiler. Jakob Engblom geeft hiervoor in zijn paper [19] een aantal tips.

7.7.4 Pakketten besnuffelen

Na het omzetten van de bytestroom in leesbare netwerkpakketten, gebruiken we de informatie uit deze pakketten om de eigenlijke versleutelsetverlaging uit te voeren. Om deze informatie te weten te komen, zullen we de pakketten *besnuffelen*, hun inhoud doorzoeken.

Eerder zagen we dat we voor het compromitteren van de `Finished` berichten de `pre_-master_secret` moeten bemachtigen, om dan met behulp van de random waarden uit de `Hello` berichten alle geheime sleutels af te leiden. Deze informatie kunnen we uit de volgende berichten extraheren:

- `ClientHello`
  Hieruit lezen we de `client_random` waarde af.

- `ServerHello`
  Hierin vinden we de `server_random` waarde.
- `ClientKeyExchange`
  Dit bericht bevat de `pre_master_secret` onder geëncrypteerde vorm.

Alle andere SSL handshake pakketten worden gebruikt om de interne staat van onze aanvalstool aan te passen.

7.7.5 Pakketten wijzigen

Als we door middel van snuffelen voldoende informatie hebben verzameld, kunnen we de eigenlijke versleutelsetverlagingsaanval uitvoeren door drie pakketten te wijzigen: de `ClientHello` en de beide `Finished` berichten.

We bespreken de implementatie van deze wijzigingen voor elk bericht in meer detail.

ClientHello

De aanpassing van de `ClientHello` kan op twee eenvoudige manieren, waarbij we telkens proberen te bereiken dat client en server de zwakke export versleutelset `TLS_RSA_EXPORT_WITH_RC4_40_MD5` negotiëren:

- **Manipulatie van de eerste keuze**
  Vervang de eerste versleutelset in de lijst door de zwakke export versleutelset.

  Deze methode werkt enkel indien de server de voorkeur van de client respecteert, en zijn eerste keuze aanvaardt. In de meeste gevallen (bijvoorbeeld bij een OpenSSL server met standaard instellingen) slaagt deze aanpak.
- **NULL overschrijving**
  Vervang alle aanwezige versleutelsets door de `NULL` versleutelset, behalve de zwakke export versleutelset.

  Dit zal ervoor zorgen dat de server de zwakke export versleutelset aanvaardt (indien hij deze ondersteunt), aangezien de `NULL`-versleutelset een ongeldige optie is die geen enkele server zal aanvaarden. Deze methode is minder elegant dan de vorige, maar werkt gegarandeerd.

Naast deze twee eenvoudige aanpassingen is het ook mogelijk de `ClientHello` wat grondiger te manipuleren, en de hele versleutelsetlijst te vervangen door een lijst die slechts één versleutelset bevat, de door de aanvaller gewilde zwakke export versleutelset.

Dit heeft echter tot gevolg dat de lengte van het `ClientHello` bericht verandert, wat tot problemen leidt op het TCP niveau, zoals we nu zullen bespreken.

Zoals we eerder zagen loopt SSL bovenop het TCP protocol. Dit protocol heeft een aantal ingebouwde mechanismen die er een *betrouwbaar* protocol van maken, dat wil zeggen dat

het garandeert dat alle verstuurde pakketten in de juiste volgorde op hun bestemming arriveren. Het gebruikt hiervoor een systeem van *TCP acknowledgements*, berichten die door de ontvanger terug naar de verstuurder gestuurd worden om te melden dat de pakketten goed aangekomen zijn. Dit mechanisme steunt hiervoor op de lengte van de TCP pakketten. Indien we de lengte van de `ClientHello` veranderen, wijzigen we ook de lengte van het onderliggende TCP pakket, waardoor het TCP acknowledgement-mechanisme van de kaart geraakt, en de connectie gesloten wordt.

Het is mogelijk om al deze TCP acknowledgments te vervalsen, maar dit zou betekenen dat *elk* TCP pakket gewijzigd moet worden, een erg zware ingreep op de connectie, en een inspanning die volgens ons niet in verhouding staat tot het verkregen resultaat.

In onze implementatie hebben we dus voor de elegante, maar minder zekere, oplossing van het vervangen van de eerste keuze gekozen. Daarnaast hebben we ook een gegarandeerd werkende oplossing geïmplementeerd onder de vorm van de *NULL overschrijving*.

Hierbij moeten we nog opmerken dat deze 'elegante' oplossing van het vervangen van de eerste keuze nog verbeterd zou kunnen worden door niet te *vervangen*, maar te *verwisselen*. Hiermee bedoelen we dat de zwakke versleutelset[5] verwisseld wordt met de versleutelset op de eerste plaats. De oorspronkelijke eerste keuze komt dus op de oorspronkelijke plaats van de zwakke export versleutelset terecht, die zelf op de eerste plaats komt te staan.

Deze drie methoden (`NULL` overschrijving, manipulatie van eerste keuze door middel van vervanging en door middel van verwisseling) zijn allemaal even krachtig, alle hebben hetzelfde resultaat tot gevolg. Wel zijn de minder 'elegante' methoden makkelijker detecteerbaar en verdachter. Zo is de `NULL` overschrijving duidelijk weinig subtiel, maar ook bij manipulatie van de eerste keuze door vervanging gebeurt er duidelijk iets vreemd. De zwakke export versleutelset komt in dat geval immers *tweemaal* voor in de lijst versleutelsets.

OpenSSL blijkt met dit vreemde gedrag echter geen enkel probleem te hebben, dus we hebben nagelaten de meeste 'elegante' oplossing door middel van verwisseling te implementeren.

Wel is het belangrijk om ook deze variatie te vermelden, om niet de indruk te wekken dat een versleutelsetverlagingsaanval eenvoudig te voorkomen is door duplicaten[6] in de versleutelsetlijst te detecteren en verbieden. Desalniettemin denken we dat een extra controle in alle SSL implementaties, speciaal gericht op het detecteren van zulke duplicaten, nuttig is, aangezien duplicaten in een legale SSL connectie in principe nooit voorkomen, en dus op verdacht gedrag wijzen.

Finished

De `Finished` berichten zijn heel wat moeilijker te vervalsen, zoals we eerder zagen. Hiervoor moeten we de `pre_master_secret` te weten komen, wat het bruteforcen van een 512 bit RSA sleutel inhoudt. Hoe we dit praktisch aanpakken, daar komen we wat verder op

[5] Die altijd in de lijst aanwezig zal zijn, aangezien de client deze moet ondersteunen om de aanval te doen slagen. Dit is de voorwaarde van 'zwakke asymmetrische encryptie', die we vervuld achten.

[6] Zijnde een hele reeks NULL versleutelsets of een dubbele export versleutelset.

terug, we bespreken eerst hoe we deze `pre_master_secret` zullen gebruiken om nieuwe `Finished` berichten te genereren.

Dit gaat in verschillende stappen:

1. **Sleutelafleiding**
   Gebruik de eerder geëxtraheerde `pre_master_secret`, `client_random` en `server_random` waarden om het (export) sleutelmateriaal van de SSL connectie te berekenen. Hierbij bestaat er een verschil tussen de SSLv3 en TLS sleutelafleiding.
2. **Bereken nieuwe `Finished` berichten**
   Herbereken de MAC's in de `Finished` berichten, die veranderd zijn omwille van de nieuwe `ClientHello`.
3. **Beveilig nieuwe `Finished` berichten**
   Bereken een nieuwe MAC over de vernieuwde `Finished` berichten (gebruik een juist volgnummer) en encrypteer het geheel.

Voor elk van deze stappen implementeren we een eigen C functie, gebaseerd op de overeenkomstige functies in het OpenSSL pakket. De reden waarom we niet eenvoudigweg deze functies van het OpenSSL pakket rechtstreeks aanspreken, werd eerder besproken.

De nieuwe `Finished` berichten worden dan vervolgens naar de juiste partij doorgestuurd.

7.7.6 Pakketten versturen

Nadat we een pakket gevangen, besnuffeld en mogelijk gewijzigd hebben, moet dit pakket uiteindelijk op zijn juiste bestemming worden gebracht, waarvan het was afgeweken omwille van de MITM aanval.

Hiervoor verkozen we `Libnet` te gebruiken op *geavanceerd IP niveau*[7], dat wil zeggen dat we volledige IP pakketten aan de `Libnet` bibliotheek doorgeven, waarna deze zelf de IP checksum berekent en een datalink laag header (in ons geval een Ethernet header) aan het geheel toevoegt, om dit Ethernet pakket vervolgens op het netwerk te zetten.

Aangezien de pakketten die we ontvangen van `Libpcap` deze volledige IP vorm hebben, kunnen we deze vervolgens integraal doorsturen, zonder verdere verwerking, althans als het pakket niet gewijzigd moet worden. Het overgrote deel van alle pakketten zit in dit geval en kan op deze manier dus eenvoudigweg, zonder enige verdere ingreep, rechtstreeks terug op het netwerk gezet worden, door `Libpcap` en `Libnet` naadloos te laten samenwerken.

Voor pakketten die wel gewijzigd worden, moet, naast de wijziging zelf, echter ook de *TCP checksum* herberekend worden, alvorens deze te versturen. Het is mogelijk om deze berekening door `Libnet` te laten uitvoeren, maar daarvoor moeten we `Libnet` meer verantwoordelijkheid over het pakket geven en deze aanpak is weinig compatibel met de geavanceerde mode van `Libnet` waarvoor we eerder kozen.

Dus, hoewel het mogelijk is om de TCP checksum berekening aan `Libnet` over te laten,

[7] aangeduid met de initialisatieparameter `LIBNET_RAW4_ADV`

kozen we ervoor deze zelf te implementeren. Dit laat ons dan toe om `Libnet` zonder problemen in geavanceerde modus te gebruiken, de meest efficiënte aanpak voor het grootste deel van de behandelde pakketten, zoals we eerder zagen.

7.7.7 Decryptie van de `pre_master_secret`

Het moeilijkste deel van de aanval is het verkrijgen van de `pre_master_secret` uit de `ClientKeyExchange`. Eerder zagen we dat we hiervoor de publieke sleutel encryptie moeten *bruteforcen*. Omdat we de versleutelset van de connectie hebben verlaagd tot een zwakke export versleutelset, komt dit neer op het *bruteforcen* van een 512-bit RSA sleutel.

In het vorige hoofdstuk toonden we aan dat dit mogelijk is, indien de aanvaller over voldoende middelen beschikt. We rekenden toen uit dat het breken van zo'n sleutel in één dag geen enkel probleem is voor een aanvaller die bereid is 622 500 dollar te spenderen.

Helaas kunnen we bij onze praktische aanval niet uitgaan van een gelijkaardig budget, daarom zullen we aannemen dat dit *bruteforcen* mogelijk is, maar voor onze praktische implementatie een *work-around* gebruiken, waarbij de aanvaller die 512-bit RSA sleutel op een andere manier dan *bruteforcen* in zijn bezit krijgt.

Die work-around vraagt actie ten opzichte van alle betrokken partijen, als volgt:

- **Server**
  De OpenSSL server implementatie wordt aangepast om de tijdelijke 512-bit RSA sleutel, die deze implementatie genereert bij de eerste connectie die gebruikmaakt van Ephemeral RSA, uit te schrijven naar een *PEM bestand*, het standaardformaat van OpenSSL om certificaten, private of publieke sleutels te bewaren [52]. Vervolgens zal de server dit bestand via een HTTPS server ter beschikking stellen op het netwerk.
- **Client**
  De server genereert pas een tijdelijke RSA sleutel wanneer hij nodig is, dat is bij de eerste connectie die Ephemeral RSA gebruikt. Daarom zal de client in onze implementatie een eerste keer expliciet verbinding leggen met behulp van een zwakke versleutelset (zonder dat de aanvaller hier invloed op uitoefent), waarop de server een tijdelijke sleutel zal genereren.
- **Aanvaller**
  De aanvaller zal, nadat de server gestart is en de client het aanmaken van een tijdelijke sleutel heeft gestimuleerd, het bestand dat de tijdelijke sleutel bevat van de server downloaden.

Dit alles zorgt ervoor dat de tijdelijke 512-bit RSA sleutel bij de aanvaller terechtkomt, waarna hij deze kan gebruiken om de `pre_master_secret` te decrypteren, en de rest van de aanval uit te voeren.

Een aanvaller die wel de financiële middelen heeft om de sleutel te *bruteforcen* kan eenvoudigweg de *gebruteforcede* sleutel naar een bestand schrijven, en dit bestand op commandline aan onze tool meegeven.

7.7.8 Resultaten

Een gedetailleerd verloop van de geïmplementeerde versleutelsetverlagingsaanval is te vinden in Bijlage B.3).

7.8 Conclusie

We besluiten dat de versleutelsetverlagingsaanval relatief makkelijk praktisch te implementeren is, indien de aanvaller aan twee voorwaarden kan voldoen:

- **MITM aanval mogelijk**
- **Zwakke gemeenschappelijke asymmetrische encryptie**

De eerste voorwaarde blijkt niet zo sterk. Een aanvaller mag zich niet te ver van client of server bevinden, en het netwerk waarop hij zijn MITM aanval uitvoert mag niet te sterk beveiligd zijn. In praktijk vormt deze voorwaarde zelden een probleem.

De enige echte hindernis bij een versleutelsetverlagingsaanval is het kraken van de zwakste gemeenschappelijke asymmetrische encryptie. Bij normaal SSL gebruik is dit altijd minstens een 1024-bit RSA sleutel, tot nog toe computationeel onmogelijk om te kraken. SSL biedt echter nog steeds export versleutelsets aan, waarbij het, via Ephemeral RSA, terugvalt op een private sleutel van slechts 512-bit. Zo'n sleutels blijken wel te kraken, binnen redelijke tijd, voor een relatief aanvaardbare kost, zoals we eerder berekenden.

Dus, indien de zwakste gemeenschappelijke asymmetrische encryptie gevormd wordt door 512-bit RSA, blijkt een versleutelsetverlagingsaanval wel degelijk uitvoerbaar, zoals wij in dit hoofdstuk aantoonden met een theoretische uiteenzetting gevolgd door een praktische demonstratie.

In het volgende hoofdstuk gaan we na hoeveel servers en clients tegenwoordig, vier jaar na het afschaffen van de export reguleringen, nog zo'n zwakke export versleutelset ondersteunen, en dus kwetsbaar zijn voor de versleutelsetverlagingsaanval.

Hoofdstuk 8

Toepasbaarheid van de aanval in de echte wereld

"There are three kinds of lies: lies, damn lies, and statistics."

– Benjamin Disraeli

8.1 Inleiding

In dit hoofdstuk gaan we de praktische implicaties van de versleutelsetverlagingsaanval na, door te onderzoeken hoe toepasbaar deze aanval is in de "echte wereld".

De versleutelsetverlagingsaanval blijkt mogelijk indien zowel server als client een zwakke export versleutelset ondersteunen, waardoor een aanvaller in staat is de connectie tot Ephemeral RSA te dwingen (met behulp van een MITM aanval), waarbij een asymmetrische encryptie gebruikt wordt met een RSA sleutel van slechts 512 bit.

We onderzoeken eerst in welke mate reële servers zo'n zwakke export versleutelsets ondersteunen, waarna we nagaan hoe kwetsbaar verschillende client implementaties zijn voor deze aanval.

8.2 SSL servers onderzocht

8.2.1 Inleiding

In deze sectie bekijken we welke servers kwetsbaar zijn voor de versleutelsetverlagingsaanval die besproken werd in Hoofdstuk 7. Er wordt dus nagegaan hoeveel servers in de "echte wereld" de zwakke export versleutelset `TLS_RSA_EXPORT_WITH_RC4_40_MD5` ondersteunen. We zullen ook aantonen dat deze specifieke versleutelset de meest gebruikte export versleutelset is, en onze eerdere keuze zo verklaren.

8.2.2 Extern onderzoek

SecuritySpace is een service die aangeboden wordt door een Canadees consultant bureau dat onder andere gespecialiseerd is internetbeveiliging, en dat op deze manier regelmatig verschillende internet-gerelateerde onderzoeken publiceert.

Zo publiceert het maandelijks de resultaten van een uitgebreid onderzoek in verband met SSL servers op het internet, waarin onder andere wordt nagegaan welke versleutelsets deze servers ondersteunen.

We geven hier de resultaten van het onderzoek dat op 1 april 2004 gepubliceerd werd, en dat gebaseerd is op statistieken die verzameld werden doorheen de maand maart van datzelfde jaar. Recentere resultaten kunnen elke maand gevonden worden op de SecuritySpace website[1].

Ondersteunde versies

Tabel 8.1 geeft een overzicht van de ondersteunde protocolversies van de onderzochte servers.

Versie	Aantal	Percentage
SSLv3	164987	97,95%
TLSv1	160775	95,44%
SSLv2	160231	95,12%

Tabel 8.1. SSL versies, SecuritySpace Secure Server Survey, April 1st, 2004

Uit het onderzoek blijkt dat bijna alle geteste servers zowel SSLv2, SSLv3 als TLSv1 ondersteunen. Het is opmerkelijk dat nog steeds zoveel servers SSLv2 connecties accepteren, jaren nadat aangetoond is dat deze versie belangrijke zwakheden bevat. Hoewel het grootste deel van deze SSLv2 compatibele servers duidelijk servers zijn die in de eerste plaats een hogere versie aanbieden, waardoor de verantwoordelijkheid van het al dan niet aangaan van een SSLv2 connectie volledig bij de client ligt, nemen we aan dat er ook een klein percentage servers is dat *uitsluitend* SSLv2 connecties aanbiedt. Dit laatste is eigenlijk onaanvaardbaar, aangezien een client die met zo'n server een verbinding moet leggen, gedwongen wordt om een zwak protocol te gebruiken. SecuritySpace geeft echter geen data vrij over het precieze aantal *SSLv2-only* servers, dus dit blijft gokwerk.

Daarnaast merken we op dat SSLv3 nog steeds de belangrijkste en meest voorkomende versie is, hoewel het verschil tussen de verschillende versies opvallend klein is. Dit laatste is een gevolg van het feit dat de meeste server implementaties alle drie de versies standaard ondersteunen.

[1] www.securityspace.com

Ondersteunde versleutelsets

Naast data over de ondersteunde protocolversies, biedt SecuritySpace ook data over de ondersteunde versleutelsets aan (Tabel 8.2).

Versleutelset	Aantal	Percentage
RC4-MD5	158103	98,67%
EXP-RC4-MD5	157976	98,59%
DES-CBC3-MD5	157573	98,34%
EXP-RC2-CBC-MD5	156239	97,51%
DES-CBC-MD5	156167	97,46%
RC2-CBC-MD5	155589	97,10%
RC4-64-MD5	83625	52,19%

Tabel 8.2. SSL versleutelsets, SecuritySpace Secure Server Survey, April 1st, 2004

De “EXP” geeft aan dat het om een export versleutelset gaat, en EXP-RC4-MD5 is onze `TLS_RSA_EXPORT_WITH_RC4_40_MD5` versleutelset. We zien onmiddellijk dat dit inderdaad de meest voorkomende versleutelset is.

Belangrijk is wel te beseffen dat dit enkel de resultaten zijn van SSLv2 compatibele servers, aangezien SecuritySpace zijn versleutelsetdata enkel op dit soort servers baseert. We kunnen echter nog altijd een schatting maken van het *minimum* aantal servers die de bewuste exportversleutelset ondersteunen, op de volgende wijze:

Stel:
$P(A)$ = kans dat server SSLv2 ondersteunt
$P(B)$ = kans dat server EXP-RC4-MD5 versleutelset ondersteunt
$P(A \cap B)$ = kans dat server SSLv2 én EXP-RC4-MD5 versleutelset ondersteunt
$P(B|A)$ = kans dat server EXP-RC4-MD5 versleutelset ondersteunt
als we weten dat de server SSLv2 ondersteunt

Gegeven:
$P(A) = 95{,}12\%$
$P(B|A) = 98{,}59\%$

Dan:
$P(A \cap B) = P(A) * P(B|A) = 93{,}78\%$

We kunnen dus berekenen hoeveel van de servers SSLv2 compatibel zijn én EXP-RC4-MD5 versleutelset ondersteunen, maar er is geen manier om uit de data af te leiden hoe groot het percentage van *alle* servers is dat deze export versleutelset ondersteunt (=$P(B)$). Dit percentage zal groter zijn dan $P(A \cap B)$, aangezien er ook servers zijn die niet compatibel zijn met SSLv2, maar wel deze export versleutelset ondersteunen. $P(A \cap B)$ is dus een minimumwaarde, in werkelijkheid zal het percentage kwetsbare servers (servers die deze export versleutelset ondersteunen) hoger liggen, met een maximumpercentage dat 4,88% (= 1 - $P(\neg A)$) hoger ligt.

We vinden dus dat het percentage van het aantal servers dat de export versleutelset EXP-RC4-MD5 ondersteunt, gelegen is in het interval [93,78% ; 98,66%], althans volgens SecuritySpace.

8.2.3 Eigen onderzoek

De resultaten van het SecuritySpace onderzoek lijken tot een erg goede schatting te leiden van het aantal servers in de echte wereld dat onze versleutelset ondersteunt. We besluiten echter een eigen beperkt onderzoek uit te voeren, specifiek gericht op onze noden, om de resultaten van dit onderzoek te verifiëren.

We schrijven hiervoor een kleine tool, die zich als client gedraagt, en test of een bepaalde server onze export versleutelset ondersteunt, door te trachten om een SSL connectie met die server te leggen met behulp van de EXP-RC4-MD5 versleutelset. We zorgen ervoor dat onze client compatibel is met alle drie de protocolversies, zodat dit testen onafhankelijk gebeurt van de protocolversie die de server ondersteunt, in tegenstelling tot het onderzoek van SecuritySpace.

Deze tool wordt in de **JAVA** programmeertaal geschreven, aangezien die door middel van zijn `javax.net.ssl` bibliotheek [35] toelaat om zo'n SSL client erg makkelijk en snel op te zetten.

De tool op zich is echter niet genoeg om ons onderzoek uit te voeren, we hebben ook een lijst van bekende, veelgebruikte SSL servers nodig als input voor onze testtool.

Om deze lijst te genereren maken we gebruik van de experimentele **Google Web API** [29], waarmee we een JAVA tool construeren die zoveel mogelijk SSL servers verzamelt.

SSL servers op het internet zijn typisch HTTPS (HTTP over SSL) servers, die het prefix *https* hanteren, in plaats van het gewone *http*. Door te zoeken op dit prefix, met behulp van de zoekstring *"allinurl: https"*, zijn we in staat een hele hoop SSL servers te verzamelen.

De Google Web API zit nog in een experimentele fase, waardoor het een aantal nadelen heeft die het genereren van zo'n lijst van SSL servers bemoeilijkt:

- Het maximum aantal resultaten per query is 10
- Er zijn maximum 1000 queries per dag toegelaten
- De maximum diepte is 1000 resultaten
- Het maximum aantal woorden binnen één query is 10, het maximum aantal bytes is 2048.

Hoewel de Google Web API slechts 10 resultaten per query kan teruggeven, heeft het een mechanisme om aan te geven *welke* 10 resultaten. Dit gebeurt met behulp van een *startwaarde*, een gebruiker die bijvoorbeeld resultaten 800-809 wilt bekomen, geeft de startwaarde 800 op. Daarbij geldt wel dat de gebruiker altijd beperkt is tot een diepte van 1000 resultaten.

De tijdslimiet is niet zo'n probleem, we kunnen gerust verschillende dagen of weken aan dit onderzoek spenderen.

Het feit dat de diepte beperkt is, vormt echter wel een probleem. Dat wil immers zeggen dat we, voor onze query *"allinurl: https"*, slechts de eerste 1000 resultaten kunnen ophalen, en dus maximum een lijst van 1000 verschillende servers kunnen samenstellen. Dit is erg weinig, zelfs als we even vergeten dat die eerste 1000 resultaten erg veel duplicaten bevatten.

Bovendien moeten we erop letten niet teveel woorden te gebruiken, waardoor het niet mogelijk is om eenvoudigweg alle eerder gevonden resultaten uit te sluiten (met behulp van '*-resultaat*').

We kunnen het diepteprobleem echter ook oplossen door onze query slechts een beetje uit te breiden, door deze telkens te restricteren tot slechts één land. De Google Web API heeft hiervoor een speciaal mechanisme, het *restrict* mechanisme, waarmee 241 verschillende landen onderscheiden kunnen worden [29].

Deze truc laat ons toe om 241 verschillende queries samen te stellen, waaruit we telkens 1000 HTTPS servers kunnen halen. In theorie laat deze methode dus toe om een lijst van 241 000 SSL servers te bekomen, maar in praktijk blijkt deze methode heel veel duplicaten op te leveren, waardoor we uiteindelijk slechts 13 821 unieke servers overhouden.

Deze lijst wordt vervolgens getest met onze testtool, die de servers indeelt in drie categorieën:

- **Veilig**
  Deze servers ondersteunen de `TLS_RSA_EXPORT_WITH_RC4_40_MD5` versleutelset NIET, en zijn dus veilig voor de versleutelsetverlagingsaanval.
- **Kwetsbaar**
  Deze servers ondersteunen de bewuste versleutelset wel, en zijn dus kwetsbaar voor de versleutelsetverlagingsaanval zoals besproken in Hoofdstuk 7.
- **Onbekend**
  Met deze servers kon geen verbinding gelegd worden.

De servers die in de klasse *onbekend* vallen, worden enkele dagen later nogmaals getest, waardoor een groot deel van deze servers alsnog kan ingedeeld worden in één van de twee andere klassen. Servers die hierna nog *onbekend* zijn, worden enkele dagen later nog een derde en laatste maal getest. Servers die na drie maal nog steeds in de categorie *onbekend* vallen, beschouwen we als ongeldig.

Op deze manier houden we uiteindelijk 13 117 geldige servers over, ingedeeld zoals in Tabel 8.3.

Dit gehele onderzoek, uitgevoerd gedurende maart 2004, vond dat 96,15% van alle HTTPS servers de bewuste export versleutelset ondersteunt, en dus kwetsbaar is.

Dit komt erg goed overeen met het interval dat we berekenden uit de SecuritySpace resultaten ([93,78% ; 98,66%]); ons resultaat blijkt immers midden in dit interval te vallen.

	Aantal	Percentage
Kwetsbaar	12612	96,15%
Veilig	505	3,85%

Tabel 8.3. Eigen onderzoek naar SSL servers, Maart 2004

8.2.4 Conclusie

We besluiten aan de hand van twee onafhankelijke onderzoeken, waarvan één door onszelf werd uitgevoerd en één door een gereputeerd consultant bureau, dat ongeveer 96,15% van alle SSL servers op het internet kwetsbaar is voor een versleutelsetverlagingsaanval zoals beschreven in Hoofdstuk 7.

8.3 SSL clients onderzocht

8.3.1 Inleiding

Om na te gaan in hoeverre clients kwetsbaar zijn voor de eerder beschreven versleutelset-verlagingsaanval, gaan we na welke bekende client implementaties de bewuste versleutelset ondersteunen.

Ook hier baseren we ons weer op HTTPS connecties, de meest gebruikte soort SSL connecties, om ons onderzoek uit te voeren. Als we spreken over clients in verband met HTTPS connecties, spreken we over *internet browsers* met SSL ondersteuning.

We zullen eerst nagaan welke de meestgebruikte browsers zijn, waarna we voor een aantal browsers nagaan of ze kwetsbaar zijn voor de versleutelsetverlagingsaanval.

8.3.2 Meestgebruikte browsers

Om te bepalen welke de meest gebruikte browsers zijn, baseren we ons op de browser statistieken van W3Schools [6], naar eigen zeggen de "grootste website in verband met webontwikkeling op het net".

Hun meest recente resultaten zien eruit als in Tabel 8.4.

Hierbij moeten we wel opmerken dat zulke statistieken steunen op *browserdetectie*, een proces dat weinig accuraat is, en waarin dikwijls verschillende browsers met elkaar verward worden. Daarenboven laten vele browsers de gebruiker toe om zelf de identificatie van de browser te bepalen, waardoor een `Opera` gebruiker zich kan voordoen als bijvoorbeeld een Internet Explorer gebruiker. Voor meer informatie in verband met browserdetectie, zie [13].

Wat we wel met zekerheid kunnen afleiden is het feit dat Internet Explorer veruit de meest gebruikte internet browser is.

Browser	Percentage
Internet Explorer 6	72,2%
Internet Explorer 5	10,6%
Mozilla	10,0%
Opera 7	2,1%
Netscape Navigator 7	1,4%
Netscape Navigator 4	0,4%
Netscape Navigator 3	0,4%

Tabel 8.4. Meestgebruikte browsers, W3Schools Browser Statistics, April 2004

8.3.3 Kwetsbaarheid

We gaan de kwetsbaarheid van vijf bekende en minder bekende browsers na:

- Internet Explorer
- Netscape Navigator
- Opera
- Konquerer
- Mozilla

Om na te gaan of deze browsers standaard de beruchte export versleutelset `TLS_RSA_EXPORT_WITH_RC4_40_MD5` ondersteunen, starten we een OpenSSL HTTPS server, die uitsluitend deze versleutelset ondersteunt:

```
openssl s_server -accept 443 -key key.pem -cert cert.pem -www -cipher
EXP-RC4-MD5
```

waarbij de *-www* parameter ervoor zorgt dat elke client die een HTTPS verbinding maakt met deze server een rapport te zien krijgt over de parameters van de gebruikte SSL connectie, zoals overeengekomen versleutelset, time-out waarden, ...

Vervolgens maken we met elke browser een verbinding naar "https://localhost". Indien de browser de export versleutelset ondersteunt, zal hij een webpagina tonen met een kort rapport over de gelegde SSL connectie. In het andere geval zal de browser een foutmelding genereren. De resultaten worden getoond in Tabel 8.5.

Uit onze tests blijkt dat *enkel* Konquerer *standaard* zo geconfigureerd is dat de beruchte export versleutelset niet aanvaard wordt. Alle andere browsers zijn kwetsbaar voor de beschreven versleutelsetverlagingsaanval, hoewel elke browser behalve Internet Explorer de mogelijkheid heeft om gedetailleerd aan te geven welke versleutelsets ondersteund moeten worden en welke niet. Internet Explorer biedt slechts de mogelijkheid om in te stellen welke protocolversies aanvaard moeten worden[2].

[2]Ongetwijfeld kunnen ook de afzonderlijke versleutelsets ingesteld worden door het Windows Registry te manipuleren, maar daarover vonden we geen informatie.

Browser	Kwetsbaar?
Internet Explorer 6.0	**JA**
Netscape Navigator 7.1	**JA**
Opera 7.23	**JA**
Konquerer 3.0	neen
Mozilla 1.1	**JA**

Tabel 8.5. Welke browsers ondersteunen de EXP-RC4-MD5 export versleutelset?

8.3.4 Conclusie

Zo goed als *elke* browser ondersteunt *standaard* een export versleutelset die toelaat een versleutelsetverlagingsaanval uit te voeren. En hoewel deze ondersteuning in de meeste browsers eenvoudig is uit te schakelen, veronderstellen we dat het overgrote deel van de gebruikers dit niet doet.

Uitgaande van het feit dat de veilige browser Konquerer slechts voor een heel klein deel van alle HTTPS (SSL) connecties gebruikt wordt, en van de veronderstelling dat slechts een marginaal deel van alle niet-Konquerer-gebruikers zijn browser goed geconfigureerd heeft, schatten we dat 90 tot 99% van alle clients kwetsbaar is.

8.4 Conclusie

We besluiten dat 94 tot 99% van alle servers en 90 tot 99% van alle clients kwetsbaar is voor de eerder beschreven versleutelsetverlagingsaanval.

Met andere woorden, een aanvaller die de versleutelsetverlagingsaanval beheerst, kan 85 (=90*94%) tot 98% (=99*99%) van alle SSL connecties ter wereld compromitteren.

Hoofdstuk 9

Een versleutelsetverlagingsaanval op SSL voorkomen

"Security is a never-ending battle.

Ah, the Yin-Yang of it all!

For every action, a counter; for every good, a balancing evil.

Accepting this reality and judging solutions as achieving an ever changing,

dynamic balance is essential to avoiding the pitfalls of absolutism.

...who wants to be in a rut anyway?"

– C. W. Flink

9.1 Inleiding

In dit hoofdstuk zullen we bespreken welke maatregelen een gebruiker kan treffen om een versleutelsetverlagingsaanval te voorkomen.

In Hoofdstuk 7 zagen we dat het uitvoeren van de versleutelsetverlaging afhangt van twee precondities:

- **MITM aanval mogelijk**
- **Zwakke gemeenschappelijke asymmetrische encryptie**

Indien één van deze twee voorwaarden niet vervuld kan worden, zal de aanval mislukken.

We zullen beginnen met een bespreking van de manieren waarop een netwerkbeveiliger zijn netwerk kan beschermen tegen MITM aanvallen in het algemeen.

Vervolgens gaan we na hoe client en server een vervulling van de tweede voorwaarde zo onwaarschijnlijk mogelijk kunnen maken, door enkele tegenmaatregelen voor te stellen die specifiek gericht zijn op het voorkomen van een versleutelsetverlagingsaanval.

9.2 Beschermen tegen MITM aanvallen

Een netwerk kan op twee manieren beschermd worden tegen MITM aanvallen:

- **Preventie**
  We kunnen actief maatregelen nemen om MITM aanvallen te *voorkomen*.
- **Detectie**
  We kunnen trachten MITM aanvallen te *detecteren*, om alsnog de schade te beperken.

Preventie verdient de voorkeur, maar in sommige gevallen wegen de kosten van preventiemaatregelen niet op tegen de veiligheid die deze bieden, in dat geval kan gekozen worden voor detectie. Beide kunnen natuurlijk ook in combinatie gebruikt worden, voor een nog beter effect.

We bespreken nu beide opties.

9.2.1 MITM aanval voorkomen

In Hoofdstuk 6 stelden we een aantal mogelijkheden voor om een MITM aanval uit te voeren. Nu geven we voor elk van deze aanvallen één goede tegenmaatregel [8]:

- **Secure ARP**
  ARP berichten worden geauthenticeerd, waardoor *ARP vergiftiging* verhinderd wordt.
- **DNS Security Extensions (DNSSEC)**
  Dit mechanisme voorkomt *DNS vergiftiging*, door DNS verkeer te authenticeren.
- **Poortbeveiliging op de switch**
  Dit voorkomt *switch port stealing*.
- **Negeer ICMP Redirects**
  Maak geen gebruik van het *ICMP redirect*-mechanisme, dit voorkomt MITM aanvallen die steunen op vervalste `ICMP Redirect` berichten.
- **Verbied ICMP Router Discovery Protocol (IRDP)**
  Schakel, indien mogelijk, IRDP uit, waardoor IRDP gerelateerde MITM aanvallen worden uitgesloten.
- **Beperk dynamische routing**
  Dit beperkt *route mangling* aanvallen.

Elk van deze maatregelen garandeert veiligheid, maar ten koste van een prijs. Zo steunen Secure ARP en DNSSEC op extra cryptografische berekeningen, met een performantiekost tot gevolg. De drie laatste maatregelen beperken dan weer sterk de dynamiek van een netwerk. Het is aan de beveiligingsspecialist om geval per geval de kosten en baten van deze maatregelen af te wegen.

Hierbij is de aanvaller sterk in het voordeel: de beveiliger moet *elke* zwakheid uit zijn netwerk weren, een aanvaller moet er slechts *één* vinden om een aanval uit te voeren.

9.2.2 MITM aanval detecteren

Indien een netwerkbeveiligingsspecialist berekent dat de kosten van preventie te hoog zijn, kan hij beslissen om uitsluitend detectiemaatregelen te treffen. Deze kunnen we indelen in twee klassen:

- **Onmiddellijke detectie**
  Idealiter wordt een MITM aanval onmiddellijk gedetecteerd, waardoor deze alsnog verhinderd kan worden.
- ***Post mortem* detectie**
  Of detectie nadat "het kalf al verdronken is". De gedetecteerde aanval kan niet meer verhinderd worden, want al geschied, maar *post mortem* detectie kan de beveiligingsspecialist wel attent maken op een zwakheid die verholpen moet worden, waardoor herhaling voorkomen wordt.

Onmiddellijke detectie gebeurt typisch door het *real-time monitoren* van netwerkverkeer. Dit kan gebeuren met behulp van geautomatiseerde systemen zoals *Intrusion Detection Systems (IDS)*, maar kan ook uitgevoerd worden door een menselijke netwerkbeheerder. Daarnaast is het ook mogelijk dat de detectie gebeurt door een oplettende gebruiker, die ongewoon gedrag van zijn applicaties opmerkt.

Post mortem detectie daarentegen gebeurt aan de hand van loganalyses. Ook deze analyses kunnen uitgevoerd worden door een netwerkbeheerder of een geautomatiseerde analysetool. Meestal zal een combinatie gebruikt worden; een analysetool zal de logs voorverwerken, waarna een netwerkbeheerder het gegenereerde rapport bestudeert.

Het spreekt vanzelf dat deze methode volledig steunt op de generatie van kwaliteitsvolle logbestanden, hoewel we in deze studie niet ingaan op de vraag "Wat is een logbestand van goede kwaliteit?". We beperken ons tot de (vage) eis dat zo'n bestand in de eerste plaats *volledig* moet zijn.

Ook laten we na om dieper in te gaan op de wijze waarop zo'n detectie kan gebeuren. Wel geven we hiertoe een aanzet, door te stellen dat een speciale aandacht voor gedupliceerde pakketten op het netwerk hierbij erg zinvol lijkt. In een MITM aanval worden immers typisch identieke pakketten herhaald op het netwerk, aangezien een aanvaller vele pakketten eenvoudigweg ongewijzigd zal *forwarden.*

9.2.3 Conclusie

We besluiten dat het niet eenvoudig is om MITM aanvallen te voorkomen, aangezien deze op zoveel verschillende manieren uitgevoerd kunnen worden. Ideaal zijn een stel sterke preventiemaatregelen, aangevuld met een grondig detectiemechanisme.

Preventieve maatregelen zijn maatregelen die, eenmaal ingesteld, weinig onderhoud behoeven, en dus makkelijk uitbesteed kunnen worden aan externe, gespecialiseerde beveiligingsfirma's.

Detectie is echter een continu proces, dat de voortdurende oplettendheid van vooral netwerkbeheerders, maar ook gebruikers vereist. Gelukkig zijn er geautomatiseerde tools zoals IDS'n en loganalyse tools die dit erg kunnen vergemakkelijken.

9.3 Zwakke asymmetrische encryptie voorkomen

Een versleutelsetverlagingsaanval kan enkel slagen indien de aanvaller in staat is om de zwakste gemeenschappelijke asymmetrische encryptie te kraken.

Eerder stelden we dat de zwakheid van een asymmetrische encryptie afgemeten kan worden aan twee factoren die voor een aanvaller belangrijk zijn bij het breken van een encryptie:

- **Tijdelijkheid van de sleutel**
  Om de hoeveel tijd wordt een nieuwe sleutel gegenereerd en gebruikt?
- **Middelen van de aanvaller**
  Over hoeveel middelen beschikt een aanvaller?

Te stellen dat uitsluitend deze twee factoren bepalend zijn om te beslissen of een asymmetrische encryptie al dan niet zwak is, is echter een oversimplificatie.

Strikt gezien geven deze factoren enkel en alleen een idee van de *technische haalbaarheid* van de aanval. De technische haalbaarheid geeft een goede indicatie van de kost van de aanval, maar zegt op zich niets over de *waarschijnlijkheid* van de aanval.

De waarschijnlijkheid van de aanval is een afweging tussen de *haalbaarheid* (kosten) en de *aantrekkelijkheid* (baten) van de aanval.

De aantrekkelijkheid van de aanval hangt samen met de waarde van de informatie die de aanvaller op deze manier verkrijgt. Die waarde hangt af van twee factoren:

- **Kwantiteit**
  Hoeveel connecties worden met behulp van diezelfde zwakke sleutel beveiligd?
- **Kwaliteit**
  Hoe waardevol is de informatie van elke connectie afzonderlijk die met deze zwakke sleutel beveiligd is?

Een zwakke sleutel die honderden connecties beschermt, al is de waarde van de verstuurde informatie beperkt, is aantrekkelijk. Aan de andere kant kan een zwakke sleutel, die slechts een handvol connecties beschermd, ook erg aantrekkelijk zijn, indien over elke connectie telkens erg gevoelige informatie verstuurd wordt.

Een betere manier om de zwakheid van een asymmetrische encryptie te meten, bestaat dus uit het bepalen van de waarschijnlijkheid van een aanval. Merk op dat het begrip zwakheid dus sterk afhankelijk is van de situatie; een asymmetrische encryptie kan in de ene situatie onaanvaardbaar zijn, terwijl diezelfde encryptie in een andere situatie als veilig beschouwd wordt.

Zoals in Hoofdstuk 4 in detail uitgelegd, wordt het algoritme (en de sleutellengte) van de gebruikte asymmetrische encryptie overeengekomen tijdens de handshake fase, wanneer server en client negotiëren over de te gebruiken versleutelset. Dit feit, gekoppeld aan onze nieuwe definitie van *zwakheid* aan de hand van het begrip *waarschijnlijkheid*, laat toe om de zwakheid van de gemeenschappelijke asymmetrische encryptie binnen SSL connecties op drie manieren te beïnvloeden:

- **Versleutelsetuitsluiting**
  Ga na welke versleutelsets te zwak zijn voor het doel, en verbied deze.
- **Versleutelsetbeperking**
  Ga na welke versleutelsets voldoende sterk zijn, en gebruik uitsluitend deze versleutelsets.
- **Frequentie van ephemeral sleutel-generatie**
  Hiermee kan zowel de tijdelijkheid van de *ephemeral* sleutel (genereer een nieuwe sleutel nadat er een vastgestelde hoeveelheid tijd verstreken is), als de kwantiteit (genereer om de zoveel connecties een nieuwe sleutel) beïnvloed worden.

We bespreken elk van deze technieken meer in detail, waarna we twee oplossingen voorstellen die steunen op een protocolwijziging.

9.3.1 Versleutelsetuitsluiting

Elk van de partijen kan een eenvoudige maatregel treffen om een versleutelsetverlagingsaanval te voorkomen:

Aanvaard geen versleutelsets met zwakke asymmetrische encryptie-algoritmen.

Eerder zagen we al dat de zwakheid van een versleutelset afhangt van de situatie waarin deze gebruikt wordt.

Een goede richtlijn is echter om te stellen dat alle *export versleutelsets* in principe zwak zijn, aangezien deze een asymmetrische encryptie met een sleutellengte van slechts 512 bit gebruiken, een sleutellengte die algemeen als zwak en onaanvaardbaar beschouwd wordt (zie ook Sectie 5.9.2).

Het uitsluiten van bepaalde zwakke versleutelsets kan op twee niveaus gebeuren:

- **Gebruikersniveau ("best practice")**
  Indien geweten is welke versleutelsets zwak zijn, kunnen deze op een eenvoudige manier door zowel beheerders van server implementaties als eindgebruikers van client implementaties verboden worden, daar elke implementatie (zoals we in Hoofdstuk 8 zagen) toelaat om versleutelset per versleutelset aan te geven of deze al dan niet ondersteund moet worden.

- **Protocolniveau**
  Een meer ingrijpende aanpassing is het simpelweg verbannen van alle zwakke (export) versleutelsets uit het SSL protocol. Dit werd al eerder gedaan met de "*anonymous Diffie-Hellman*" versleutelsets, die in de TLS specificatie expliciet als *deprecated* (afgeraden) aangeduid worden[1] [16].

Het uitsluiten van de export versleutelsets heeft geen enkel noemenswaardig nadeel, aangezien kan aangenomen worden dat *export-only* clients of servers zo goed als verdwenen zijn [60, p.173]. Een client, laat staan een server, heeft dus geen enkele reden om deze versleutelsets nog te ondersteunen.

9.3.2 Versleutelsetbeperking

Naast een versleutelsetuitsluiting, kan een versleutelsetbeperking toegepast worden. In dit geval beperkt client of server zich tot het ondersteunen van een beperkte verzameling sterke versleutelsets.

Deze methode lijkt erg op een versleutelsetverbod, maar is duidelijk veiliger: bij een sleutelverbod worden *enkel* versleutelsets uitgesloten waarvan bekend is dat ze *zwak* zijn, bij sleutelbeperking worden *alle* versleutelsets uitgesloten, behalve diegene waarvan bekend is dat ze *sterk* zijn. Dit is een subtiel maar belangrijk verschil.

Bovendien hoeft deze maatregel niet te betekenen dat het aantal partijen waarmee server of client kunnen communiceren erg beperkt wordt; de meeste implementaties ondersteunen een erg breed scala aan versleutelsets, waaronder velen sterke algoritmen bevatten.

Voor erg gevoelige communicatie die gedurende lange tijd veilig moet blijven, en/of erg paranoïde partijen, is het aangeraden om zowel client als server zo in te stellen dat zij uitsluitend één erg sterke versleutelset ondersteunen, zoals `TLS_RSA_WITH_AES_256_CBC_SHA` met een computationeel onmogelijk te breken 8192-bit RSA serversleutel[2].

9.3.3 Frequentie van ephemeral sleutel-generatie

Indien een server toch export versleutelsets wenst te ondersteunen, schieten zowel versleutelsetuitsluiting als versleutelsetbeperking te kort. Toch kan er ook in dat geval een maatregel getroffen worden om het gevaar van compromittering te beperken.

[1] Deze versleutelsets worden afgeraden omdat zij geen enkele vorm van authenticatie gebruiken, waardoor zij erg kwetsbaar zijn voor *man-in-the-middle attacks*

[2] Merk op dat de sterkte van symmetrische en asymmetrische sleutel hier eigenlijk niet vergelijkbaar is, de asymmetrische is veel zwakker, waardoor de symmetrische lengte nodeloos overdreven is, zoals we in Sectie 2.7 zagen.

Deze maatregel bestaat uit het aanpassen van de frequentie van de *ephemeral* sleutel-generatie. Door deze frequentie te verhogen, wordt de veiligheid verhoogd, maar gaat de performantie naar beneden (het genereren van zo'n sleutel is computationeel erg duur), en vice versa.

Ook hier is het weer aan de netwerkbeheerder om situatie per situatie af te wegen waaruit de beste verhouding beveiliging/performantie bestaat.

In zijn meest veilige vorm bestaat deze maatregel uit het opnieuw genereren van de *ephemeral* sleutel voor *elke* nieuwe connectie die gebruik maakt van Ephemeral RSA[3]. Dit geeft een belangrijk performantieverlies, maar voorkomt de versleutelsetverlagingsaanval quasi volledig, aangezien we zagen dat het zo goed als onmogelijk is om een 512-bit RSA sleutel onmiddellijk (binnen enkele seconden) te breken. De kost voor één enkele aanval wordt dus sterk verhoogd. Bovendien moet een aanvaller voor elke nieuwe connectie een nieuwe sleutel breken; één *ephemeral* sleutel beschermt slechts één connectie. De versleutelset-verlagingsaanval wordt zo wel erg onaantrekkelijk. Ook is de SSL connectie nu *perfect voorwaarts vertrouwelijk*.

Een betere performantie, met een verminderde veiligheid, kan bereikt worden door de ephemeral sleutel-generatie niet voor elke connectie opnieuw te herhalen, maar slechts elke tien connecties, of elke uur, of elke dag, al naargelang de vereisten.

Merk op dat deze maatregel enkel door beheerders van servers genomen kan worden, een client kan hier geen enkele invloed op uitoefenen.

9.3.4 Beschermen van de Ephemeral RSA handshake

De eigenlijke oorzaak van de zwakheid voor de besproken versleutelsetverlagingsaanval is het feit dat SSL zijn handshake onvoldoende beschermt. Meer specifiek is de bescherming van de normale handshake wel voldoende (althans in SSLv3 en hoger), maar een uitbreiding van de handshake, als gevolg van Ephemeral RSA, laat een aanvaller toe om de handshake te manipuleren in een versleutelsetverlagingsaanval.

Deze aanval kan voorkomen worden door de Ephemeral RSA handshake extra te beschermen. Eric Rescorla stelt zo'n aanpassing aan het protocol voor, die een wijziging van het `ServerKeyExchange` bericht behelst[60, p.172]. Dit bericht bevat de *ephemeral* sleutel, digitaal gehandtekend met de normale, sterke private serversleutel. Rescorla's aanpassing bestaat erin ook de `ClientHello`, die de aanzet geeft tot Ephemeral RSA, in dit bericht op te nemen, onder de digitale handtekening.

Dit voorkomt duidelijk de aanval, aangezien de client bij het ontvangen van deze `ServerKeyExchange` onmiddellijk zal inzien dat de handshake gemanipuleerd is, tenzij een aanvaller in staat is ook dit bericht te vervalsen. Dit laatste is erg onwaarschijnlijk, aangezien het `ServerKeyExchange` bericht beveiligd is met de volwaardige private serversleutel. Een aanvaller die deze sleutel toch kan kraken, heeft er geen enkele baat bij een versleutelset-verlagingsaanval uit te voeren, aangezien hij met behulp van deze sleutel de `pre_master_secret` op triviale wijze uit de `ClientKeyExchange` kan extraheren.

We implementeerden deze bescherming in `OpenSSL v0.9.7c`, en concludeerden dat onze

[3]Deze maatregel wordt expliciet vermeld in een aanvulling van de TLS specificatie [12]

aanval hierdoor voorkomen wordt. Een gedetailleerde verloop van de versleutelsetverlagingsaanval op een client en server die op deze wijze beschermd zijn, kan gevonden worden in Bijlage B.4.

9.3.5 Ephemeral RSA verwijderen

Een andere oplossing om versleutelsetverlagingsaanvallen te voorkomen, bestaat uit het verwijderen van de complete Ephemeral RSA uitbreiding uit SSL. Om een reden die we niet konden achterhalen wordt dit echter in geen enkel van de geraadpleegde werken als optie vermeld. Mogelijk wilt men zo de eigenschap van *voorwaartse vertrouwelijkheid* mét RSA behouden, hoewel *Ephemeral Diffie-Hellman* dezelfde eigenschap biedt, zonder bijkomende beveiligingsproblemen [12].

9.3.6 Uitbreiding van de bescherming tegen versieverlaging

Tot slot stellen we nog een techniek voor die we onderzocht hebben, maar die ***niet*** blijkt te werken.

Deze techniek is gebaseerd op de *padding check* in RSA, bedacht door Paul Kocher, die een versieverlagingsaanval voorkomt. Zoals we eerder zagen gebruiken clients die SSLv3 ondersteunen een vast patroon (waar dit patroon normaal random is) voor de *padding bytes* in de PKCS formattering bij de RSA-encryptie van de `pre_master_secret` [24]. Aangezien deze padding bytes (samen met de `pre_master_secret` zelf) geëncrypteerd worden met de publieke RSA sleutel (in het `ClientKeyExchange` bericht, kan een aanvaller dit patroon niet wijzigen.

Is het mogelijk om de versleutelsetverlagingsaanval te voorkomen, door in het `ServerKeyExchange` bericht een gelijkaardige bescherming in te bouwen?

Met andere woorden, is het mogelijk om de padding bytes, die gebruikt worden bij het genereren van de digitale RSA handtekening over de *ephemeral* sleutel, op zo'n manier te karakteriseren dat een client bij ontvangst van de `ServerKeyExchange` een versleutelsetverlagingsaanval zal detecteren?

Het antwoord blijkt "*neen*" te zijn.
De PKCS#1 specificatie stelt immers dat de padding bytes, gebruikt bij een RSA private sleutel-operatie, *niet* random mogen zijn, zoals bij een RSA publieke sleutel-encryptie, maar van de specifieke vorm "`FF`" moeten zijn [64].

Deze techniek kan dus niet gebruikt worden om een versleutelsetverlagingsaanval te voorkomen.

9.3.7 Conclusie

De veiligste oplossing bestaat uit een protocolwijziging, zij het door een verandering aan de Ephemeral RSA handshake, de verwijdering van Ephemeral RSA uit het protocol of het verbannen van bepaalde zwakke versleutelsets.

De eerste twee protocolaanpassingen zijn erg verregaand, en vereisen een belangrijke aanpassing in alle bestaande implementaties. De laatste is echter makkelijker door te voeren, en het valt te verwachten dat hoe langer hoe meer implementaties de ondersteuning van export versleutelsets zullen verwijderen, of althans standaard deactiveren, al dan niet onder invloed van een duidelijke hint in een nieuwe, aanvullende SSL specificatie.

Een oplettende gebruiker of beheerder hoeft echter niet zolang te wachten, door middel van versleutelsetbeperking of versleutelsetuitsluiting is het mogelijk om elke bestaande implementatie zodanig te configureren dat versleutelsetverlagingsaanvallen zo goed als uitgesloten worden.

De beheerder van een server implementatie kan daarenboven nog de frequentie van de ephemeral sleutel-generatie aanpassen, waardoor hij de veiligheid van zijn server implementatie, zelfs bij gebruik van Ephemeral RSA en export versleutelsets, toch nog in belangrijke mate kan beïnvloeden.

Hoofdstuk 10

Relativering en besluit

"Don't outrun the bear, just outrun the people you're with"
– *Anonymous*

10.1 Inleiding

In dit afsluitende hoofdstuk relativeren we de impact van de versleutelsetverlagingsaanval. We baseren ons daarvoor op twee bekende beveiligingsprincipes: het *"goed genoeg"-principe*, en het *principe van de zwakste schakel.*

We bespreken eerst achtereenvolgens twee principes, waarbij we deze telkens toepassen op de resultaten van onze studie.

10.2 Relativering aan de hand van het *goed genoeg* principe

10.2.1 Goed genoeg principe ('Good enough principle')

Het "goed genoeg"-principe is eigenlijk een andere formulering van het kosten/baten principe. De kost van beveiliging moet in verhouding staan tot het baten, waar de kost vooral uit performantieverlies bestaat, en het baten uit beveiligingswinst.

Ravi Sandhu formuleert dit principe als volgt [68]:

> *Alles moet veilig genoeg gemaakt worden, maar niet veiliger.*[1]

Dit impliceert dat er niet zoiets is als dé perfecte beveiliging, maar dat per situatie een kosten/baten berekening moet gemaakt worden, per situatie moet nagegaan worden welke maatregelen "goed genoeg" zijn.

[1] Naar Albert Einsteins "*Everything should be made as simple as possible, but not simpler.*"

Sandhu herformuleert daarnaast de essentie van het "goed genoeg"-principe in drie deelprincipes [68]:

De drie principes van "Goed Genoeg"-informatiebeveiliging

1. *Goed genoeg is goed genoeg*
2. *Good genoeg is altijd beter dan perfect*
3. *De echte moeilijkheid bestaat uit het bepalen van wat goed genoeg is*

Belangrijk in de beschouwing van dit "goed genoeg"-principe is het besef dat het traditionele *spion-versus-spion* ('*spy versus spy*') paradigma hoe langer hoe minder van toepassing is [72, p.213].

Dit paradigma stelt dat de te beveiligen informatie ultragevoelig en *top secret* is, en ten koste van alles beveiligd moet worden, aangezien het belang van ontelbare mensenlevens en hele naties ermee gemoeid is.

In praktijk bestaat echter slechts een erg klein deel van alle informatie uit dit soort gevoelige, typisch militaire, informatie. Het overgrote deel van de te beveiligen informatie bestaat uit persoonlijke geheimen, allerlei soorten financiële transacties, ... Informatie die van erg grote waarde kan zijn voor een individu persoonlijk, maar waarvan het universele belang miniem is.

Dat wil niet zeggen dat de beveiliging van dit soort informatie verwaarloosd mag worden, integendeel. Wel wil dit zeggen dat deze informatie niet noodzakelijk moet beveiligd worden met de laatste militaire toptechnologie. Zulke informatie moet in de eerste plaats beschermd worden tegenover andere individuen, niet tegenover de computerkracht van een hele natie, in de meeste gevallen zelfs niet tegen de computerkracht van een groot bedrijf.

Het "goed genoeg"-principe stelt niet dat het verboden is voor zo'n individu om zijn informatie te beschermen als ware het militaire topgeheimen. Wel stelt het dat dit soort bescherming voor de meeste mensen[2] *overkill* is, de kost niet waard, onnodig.

10.2.2 Relativering

Laten we dit principe eens toepassen op onze studie.

Vanuit het "goed genoeg"-principe is de verleiding groot om het SSL protocol af te doen als goed genoeg, of zelfs té goed. Zo bekeken, lijkt deze studie erg zinloos, waarom willen we iets versterken dat al te sterk is?

In dit kader willen we zelf nog een vierde deelprincipe aan het "goed genoeg"-principe toevoegen:

4. *Om te bepalen welke veiligheidsmaatregel goed genoeg is, moet men eerst bepalen hoe goed deze maatregel zelf is*

[2] Terroristen en topspionnen daargelaten

Deze assumptie zit impliciet vervat in het eigenlijke “goed genoeg”-principe, maar het is nuttig ze eens te expliciteren.

Als we deze studie nu een tweede maal beschouwen, waarbij we deze geëxpliciteerde veronderstelling in gedachten houden, blijkt deze studie wel degelijk nuttig, zelfs en vooral vanuit het “goed genoeg”-principe.

Deze studie gaat immers na of het SSL Protocol wel degelijk de sterkte bezit die het belooft. En hoewel deze studie geenszins de ambitie heeft om een complete, diepgaande veiligheidsanalyse van het SSL Protocol in al zijn facetten te vormen, draagt ze haar steentje bij op weg naar dit grotere doel, een doel dat op zijn beurt slechts een kleine stap voorwaarts betekent naar de ideale situatie van perfecte informatiebeveiliging.

10.3 Relativering aan de hand van het *zwakste schakel* principe

10.3.1 Zwakste schakel principe (‘Weakest link principle’)

Het zwakste schakel principe gaat als volgt:

> *Geen enkel beveiligingssysteem is sterker dan zijn zwakste schakel.*

Denk aan een beveiligingssysteem als een ketting, waarin één zwakke schakel genoeg is om de volledige ketting (beveiliging) te breken. Een beveiliger moet er dus voor waken dat alle schakels in zijn beveiligingsketting van *vergelijkbare* sterkte zijn.

Hoewel dit principe misschien wel het oudste aller beveiligingsprincipes is, en tevens het meest bekende, wordt dit principe zelden gerespecteerd [22].

10.3.2 Relativering

We bekijken nu de resultaten van onze studie, met dit principe in het achterhoofd.

Het doel van deze studie is het nagaan van de sterkte van het SSL protocol. Ons belangrijkste resultaat is het bewijs dat de versleutelsetverlagingsaanval een reëel gevaar is, en bovendien relatief eenvoudig praktisch te implementeren.

We zouden de lezer echter een onterecht negatief beeld van het SSL protocol suggereren, moesten we deze resultaten niet proberen te plaatsen in het omvattende kader van de netwerkbeveiliging, of zelfs de informatiebeveiliging in het algemeen.

We relativeren daarom onze resultaten aan de hand van een fictieve illustratie:

> Alice, hoofd van de O&O afdeling van een grote multinational *Kakokalo*, verzendt een hoogst geheime aanpassing aangaande het nog geheimere *Kakokalo recept* over een met SSL beveiligde connectie naar Bob, hoofd van de productieafdeling. Mallory, bedrijfsspion van Kakokalo’s grootste concurrent, *Bibse*,

en als parttime-flessenafwasser geïnfiltreerd op Kakokalo's recyclage-afdeling, heeft opdracht deze aanpassing te weten te komen. Hij krijgt een budget van 1 miljoen euro, met de garantie dat hij bij het slagen in zijn opdracht het overschot van dit budget persoonlijk mag houden.

Gegeven deze situatie, op welke manieren kan Mallory zijn opdracht volbrengen?

De SSL connectie compromitteren, is één manier, en een manier waarvoor hij waarschijnlijk een erg groot deel van zijn budget moeten aanspreken (onder andere om een *cracker* met de nodige kennis in te huren, en het PC-park te financieren dat nodig is om de asymmetrische encryptie binnen redelijke tijd te kraken), waardoor zijn persoonlijke overschot van het budget erg klein wordt.

Mallory kan daarentegen ook een professionele inbreker inhuren, die de PC van Alice of Bob steelt, of inbreekt in het O&O lab, om de aanpassingen aan het recept, en zelfs het recept zelf, te weten te komen. Deze optie is waarschijnlijk een stuk goedkoper dan het compromitteren van SSL. Indien Mallory niet te beroerd is om persoonlijk in te breken, kan hij zelfs het hele miljoen zelf opstrijken.

Onze bedrijfsspion kan daarnaast trachten zijn charmes uit te spelen, en Alice mee uit eten nemen, in de hoop het geheim rechtstreeks van haar los te krijgen[3]. Indien het hoofd O&O een eenvoudige flessenafwasser te min vindt om mee uit te gaan, heeft hij ook nog altijd de mogelijkheid om een professionele verleider in te huren, haar rechtstreeks te bedreigen, of om haar te chanteren met een compromitterende foto. Dit alles kan hij ook op Bob trachten toe te passen, of op andere Kakokalo medewerkers die toegang hebben tot het geheim.

We zouden zo bijna eindeloos kunnen doorgaan met manieren te verzinnen waarop Mallory het geheim te weten kan komen, maar het lijkt duidelijk dat een aanval op SSL niet de goedkoopste, noch de makkelijkste manier is.

Al is dit voorbeeld lichtjes overdreven, en neigt het naar een *spion-versus-spion* situatie, toch geeft het een goed beeld van de plaats van het SSL protocol in de algemene beveiligingsketting:

Het SSL protocol is **niet** de zwakste schakel in het beveiligen van informatie, zelfs verre van.

Onmiddellijk rijst de vraag waaruit de zwakste schakel dan wel bestaat.

Hierop is geen eenduidig antwoord te geven, dit verschilt van situatie tot situatie, en is dikwijls erg moeilijk te bepalen.

Hoewel...

Het is niet onredelijk te veronderstellen dat de feitelijke "zwakste schakel" in informatiebeveiliging in het overgrote deel van de gevallen bestaat uit de menselijke gebruiker. Dit werd al gesuggereerd bij onze bespreking van certificaatvervalsing-aanvallen, en wordt nog

[3] Dit is een voorbeeld van *social engineering*, een techniek waarbij een aanvaller op de een of andere slimme manier het goed vertrouwen van mensen uitbuit.

eens benadrukt in onze *KakoKalo* illustratie; de oplossingen waarbij Mallory zich richt op de menselijke gebruikers (Alice, Bob, andere medewerkers) lijken de goedkoopste, en tevens de meest haalbare. Een ander, informeel onderzoek dat dit vermoeden ondersteunt [62], stelt dat 70% van toevallige voorbijgangers computerpaswoorden vrijgeven in ruil voor een reep chocolade.

Om deze reden drukken we op het belang van een goede en continue training van elke gebruiker die toegang heeft tot belangrijke informatie, gaande van nachtwakers over netwerkbeheerders tot kaderleden. Netwerkbeveiliging, en informatiebeveiliging in het algemeen, is een erg dynamisch gebied, met snelle ontwikkelingen op de meest onverwachte momenten, vanuit de meest onwaarschijnlijke hoeken.

10.4 Besluit

Tot slot is het nuttig om na te gaan in hoeverre we onze oorspronkelijke vraag beantwoord hebben:

> **Hoe goed beschermt het *SSL Protocol* netwerkcommunicatie tegen bedreigingen uit het *Internet Threat Model*?**

Ons antwoord hierop is genuanceerd.

Op basis van de resultaten uit deze studie, en ons onderzoek naar de versleutelsetverlagingsaanval in het bijzonder, concluderen we dat de standaardconfiguratie van nagenoeg alle SSL implementaties, zij het voor server of client, onvoldoende beveiliging biedt om de communicatie tussen individuen gedurende langer dan enkele maanden voor compromittering te vrijwaren. Gaat het over erg gevoelige communicatie, die daarenboven tegen grote bedrijven of zelfs regeringen beschermd moet worden, dan kan de standaardconfiguratie van een SSL implementatie de veiligheid zelfs niet langer dan enkele uren garanderen.

Hierbij drukken we op het feit dat deze schattingen *enkel en alleen* afhangen van de kost om een 512-bit RSA sleutel te compromitteren, de *zwakste gemeenschappelijke asymmetrische encryptie* die de huidige SSL implementaties standaard bereid zijn te aanvaarden. Alle cryptanalytische vorderingen met betrekking tot het compromitteren van een 512-bit RSA sleutel hebben dus *rechtstreeks* gevolg op de veiligheid die de courante SSL implementaties standaard bieden.

Tevens wijzen we op het feit dat het negeren van de waarschuwingen van SSL implementaties met betrekking tot certificaten, het SSL protocol totaal compromitteert.

Indien een client of server (beter nog, de ontwikkelaars van SSL implementaties) echter de tijd en moeite neemt om zijn SSL implementatie nauwgezet te configureren aan de hand van onze richtlijnen in Hoofdstuk 9, en waarschuwingen in verband met certificaatvervalsingen niet domweg negeert, biedt SSL een *erg goede* bescherming tegen bedreigingen uit het Internet Threat Model.

Nawoord

Ons voornaamste doel in deze studie bestond uit het gestructureerd en coherent presenteren van een zo volledig en correct mogelijk antwoord op de initiële these.

Om deze reden is het misschien niet altijd even duidelijk in hoeverre de gepresenteerde informatie afkomstig is uit synthese en interpretatie van geraadpleegde literatuur enerzijds, en ondervinding en ervaring uit eigen praktisch onderzoek anderzijds.

Daarom geven we een kort overzicht van de stellingen die we *persoonlijk*, met behulp van *eigen* onderzoek, geverifieerd hebben.

In volgorde van belangrijkheid, met het belangrijkste onderzoek eerst:

1. Een praktische implementatie van de *versleutelsetverlagingsaanval*, na deze volledig en tot in detail op een theoretische manier te hebben uitgewerkt (Hoofdstuk 7).

   Dit is naar wij weten nooit eerder gedemonstreerd, noch deze praktische implementatie, noch deze uitgebreide theoretische uitwerking.

   Wel zijn we de aanzet tot deze uitwerking verschuldigd aan Rescorla, die de versleutelsetverlagingsaanval summier beschrijft [60, p. 171-172].

2. Uitgebreide theoretische bespreking en praktische uitvoering van mogelijke *tegenmaatregelen tegen de versleutelsetverlagingsaanval*.

   In het bijzonder de praktische verificatie van versleutelsetuitsluiting en versleutelsetbeperking, en eigen implementatie en verificatie in OpenSSL van de door Rescorla [60, p. 172] voorgestelde Ephemeral handshake beveiliging (Sectie 9.3).

3. Onderzoek naar de *ondersteuning* voor de `TLS_RSA_EXPORT_WITH_RC4_40_MD5` versleutelset van *SSL servers in de 'echte wereld'*, met behulp van de Google Web API (Sectie 8.2.3).

   Het idee om een zoekmachine te gebruiken om 'echte wereld' SSL servers te verzamelen komt van Eric Murray [51]. Wij pasten dit idee echter toe op Google, met behulp van de experimentele Google Web API, en werkten dit idee volledig uit in Java.

4. Praktisch geëxperimenteer met het *arpspoof* programma, waarmee een MITM aanval door middel van een *ARP vergiftiging gedemonstreerd* werd, en een aantal beperkingen van deze tool bovenkwamen (Sectie 6.8.3).

5. Onderzoek naar de *ondersteuning* voor de `TLS_RSA_EXPORT_WITH_RC4_40_MD5` versleutelset van *SSL client implementaties* (Sectie 8.3.3).

6. Verificatie van de *certificaatwaarschuwingen* in verband met certificaatvervalsing in *SSL client implementaties.* (Internet Explorer 6.0, Netscape 7.1, Opera 7.23, Konquerer 3.0 en Mozilla 1.1) (Sectie 5.7.3).

7. Falsificatie (voor de besturingssystemen Debian 3.0, Mandrake 9.0 en Windows XP) van de stelling dat *ongevraagde* `ARP Reply`'s in staat zijn een *mapping toe te voegen* aan de ARP cache van de *bestemmeling* (Sectie 6.3.2).

8. Nuancering van de stelling dat *statische* ARP mappings in Windowssystemen *overschreven* worden door *dynamische* `ARP Reply`'s (dit geldt alvast niet voor Windows XP, Sectie 6.3.3).

Naast dit praktische onderzoek is natuurlijk ook de synthese van verschillende theoretische bronnen volledig onze eigen verdienste.

In het bijzonder vermelden we hier het erg uitgebreide overzicht van historische zwakheden in het SSL protocol, de hieruitvolgende aanvallen en de bijhorende tegenmaatregelen, ingedeeld naar eigen inzicht.

Bibliografie

[1] ARPWatch. http://www.securityfocus.com/tools/142.

[2] B. Aupetit. *A Primer on Spectral Theory.* Springer-Verlag, New York, 1991.

[3] J. Benaloh, B. Lampson, D. Simon, T. Spies, and B. Yee. Private Communication Technology Protocol, September 1995. The second PCT draft.

[4] D. Bleichenbacher. Chosen Ciphertext Attacks Against Protocols Based on the RSA Encryption Standard PKCS #1. *Lecture Notes in Computer Science*, 1462:1–12, 1998.

[5] D. Boneh. Twenty years of attacks on the RSA cryptosystem. *Notices of the American Mathematical Society (AMS)*, 46(2):203–213, 1999.

[6] Browser Statistics. http://www.w3schools.com/browsers/browsers_stats.asp.

[7] D. Brumley and D. Boneh. Remote Timing Attacks are Practical, 2003. 12th Usenix Security Symposium.

[8] D. Bruschi, A. Ornaghi, and E. Rosti. S-ARP: a Secure Address Resolution Protocol, 2003.

[9] B. Canvel, A. Hiltgen, S. Vaudenay, and M. Vuagnoux. Password Interception in a SSL/TLS Channel. *Advances in Cryptology - CRYPT'03*, LNCS 2729:583–599, 2003.

[10] S. Cavallar, B. Dodson, A. Lenstra, W. Lioen, P. Montgomery, B. Murphy, H. Riele, K. Aardal, J. Gilchrist, G. Guillerm, P. Leyland, J. Marchan, F. Morain, A. Muffett, C. Putnam, and P. Zimmermann. Factorization of a 512-bit RSA Modulus. Eurocrypt 2000.

[11] J. Childs. Evaluating the TLS Family of Protocols with Weakest Precondition Reasoning, 2000.

[12] P. Chown. Advanced Encryption Standard (AES) Ciphersuites for Transport Layer Security (TLS), Juni 2002. RFC 3268.

[13] B. Clary. Browser Detection and Cross Browser Support, 2002. http://devedge.netscape.com/viewsource/2002/browser-detection/, geraadpleegd in April 2004.

[14] E. Dawson and L. Nielsen. Automated Cryptoanalysis of XOR Plaintext Strings, April 1996.

[15] K. De Groote. 11 miljoen Amerikanen trappen in 'phishing'-val, 7 mei 2004. http://www.vnunet.be/datanews/detalle.asp?ids=/News/Top_Stories/Personal_Computing/20040507004.

[16] T. Dierks and C. Allen. The TLS Protocol Version 1.0, Januari 1999. RFC 2246.

[17] S. Dietrich. *A Formal Analysis of the Secure Sockets Layer Protocol.* PhD thesis, Adelphi University, Department of Mathematics and Computer Science, Garden City, New York, 1997.

[18] W. Diffie and M. Hellman. New Directions in Cryptography. *IEEE Transactions on Information Theory*, 22, 6:655–664, 1976.

[19] J. Engblom. Structs on a diet: getting rid of that padding, 2002. Whitepaper.

[20] Ethereal, 2004. http://www.ethereal.com/.

[21] Ettercap. http://ettercap.sourceforge.net/.

[22] C. W. Flink. Weakest Link in Information System Security, 2002.

[23] S. Fluhrer, I. Mantin, and A. Shamir. Weaknesses in the Key Scheduling Algorithm of RC4, 2001. In Eighth Annual Workshop on Selected Areas in Cryptography.

[24] A. Freier, P. Karlton, and P. Kocher. The SSL Protocol Version 3.0, November 1996. Draft302.

[25] S. Gilheany. Evolution of Intel Microprocessors: 1971 to 2007, 2004. http://www.cs.rpi.edu/~chrisc/COURSES/CSCI-4250/SPRING-2004/slides/cpu.pdf.

[26] J. Gilmore. *Cracking DES: Secrets of Encryption Research, Wiretap Politics & Chip Design.* O'Reilly & Associates, 1998.

[27] I. Goldberg and D. Wagner. Randomness and the Netscape Browser, 1996. Dr. Dobb's Journal.
[28] A. Goodman and J. Ratti. *Finite Mathematics With Applications*. MacMillan, 1971.
[29] Google Web APIs (beta). http://www.google.com/apis/.
[30] C. Gottbrath, J. Bailin, C. Meakin, T. Thompson, and J. J. Charfman. The Effects of Moore's Law and Slacking on Large Computations, 1999.
[31] K. Hickman. The SSL Protocol, 1995.
[32] R. Housley, W. Ford, W. Polk, and D. Solo. Internet X.509 Public Key Infrastructure Certificate and CRL Profile, 1999. RFC 2459.
[33] Intel. Moore's Law. http://www.intel.com/research/silicon/mooreslaw.htm.
[34] ITU. The Directory–Authentication Framework, 1988. ITU Recommendation X.509.
[35] Package javax.net.ssl. http://java.sun.com/j2se/1.4.2/docs/api/javax/net/ssl/package-summary.html.
[36] J. Kelsey, B. Schneier, and N. Ferguson. *Notes on the Design and Analysis of the Yarrow Cryptographic Pseudorandom Number Generator*. Springer-Verlag, Berlin, 1999.
[37] V. Klima, O. Pokorny, and T. Rosa. Attacking RSA-based Sessions in SSL/TLS.
[38] V. Klima and T. Rosa. Further Results and Considerations on Side Channel Attacks on RSA, 2002.
[39] P. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. *Lecture Notes in Computer Science*, 1109:104–113, 1996.
[40] M. Kotadia. 'Phishing' scams luring more users, 19 april 2004. http://news.com.com/2100-7355_3-5194807.html.
[41] J. Koziol, D. Litchfield, D. Aitel, C. Anley, S. Eren, N. Mehta, and R. Hassell. *The Shellcoder's Handbook: Discovering and Exploiting Security Holes*. Wiley, 2004. Chapter 4: Introduction to Format String Bugs.
[42] H. Krawczyk. The Order of Encryption and Authentication for Protecting Communications (Or: how secure is SSL?), 2001.
[43] H. Krawczyk, M. Bellare, and R. Canetti. HMAC: Keyed-Hashing for Message Authentication, 1997. RFC 2104.
[44] A. Lenstra, E. Tromer, A. Shamir, W. Kortsmit, B. Dodson, J. Hughes, and P. Leyland. Factoring estimates for a 1024-bit RSA modulus. Asiacrypt 2003.
[45] Libpcap. http://www.tcpdump.org/.
[46] J. Manger. A Chosen Ciphertext Attack on RSA Optimal Asymmetric Encryption Padding (OAEP) as standardized in PKCS#1 v2.0. *Advances in Cryptology CRYPTO'01, Lecture Notes in Computer Science*, 2139:230–238, 2001.
[47] S. McLeod and M. Cohen. SSL Vulnerabilities, 2002. Computer Network Vulnerability Team, Australia Department of Defense.
[48] A. Menezes, P. van Oorschot, and S. Vanstone. *Handbook of Applied Cryptography*. CRC-Press, 2001. 5th Printing.
[49] J. C. Mitchell. Finite-State Analysis of Security Protocols. In *Computer Aided Verification*, pages 71–76, 1998.
[50] G. Moore. Cramming more components onto integrated circuits, 1965.
[51] E. Murray. SSL Server Security Survey, 2000. http://www.meer.net/~ericm/papers/-ssl_servers.html.
[52] OpenSSL. http://www.openssl.org/.
[53] H. Orman. Determining Strengths for Public Keys used for Exchanging Symmetric Keys, 2004. Draft.
[54] A. Ornaghi and M. Valleri. Man in the Middle Attacks, 2003. Blackhat Conference 2003.
[55] Open Source Vulnerability Database, 2004. http://osvdb.org.
[56] L. C. Paulson. Inductive Analysis of the Internet Protocol TLS. *ACM Transactions on Information and System Security*, 2(3):332–351, 1999.
[57] Phishing Scams Incorporate SSL Certificates, 2004. http://slashdot.org/articles/04/03/10/0156200.shtml.
[58] D. Plummer. Ethernet Address Resolution Protocol: Or converting network protocol addresses to 48.bit Ethernet address for transmission on Ethernet hardware, 1982. RFC 826.
[59] J. Postel. Internet Control Message Protocol, 1981. RFC 792.

[60] E. Rescorla. *SSL and TLS: Designing and Building Secure Systems*. Addison-Wesley, 2000.
[61] R. Rivest. The MD5 message-digest algorithm, 1992. RFC 1321.
[62] G. Rivlin. Pssst, Computer Users . . . Want Some Candy?, 2004.
[63] M. Robshaw. On Recent Results for MD2, MD4 and MD5, 1996.
[64] PKCS #1: RSA Encryption Standard, 1993. Version 1.5.
[65] RSA Bulletin, Number 2, January 23, 1996, 1996. http://islab.oregonstate.edu/koc/ece575/rsalabs/bulletn2.pdf.
[66] RSA cryptography standard: PKCS#1 v2.0, 1998.
[67] Factorization of RSA-155, 1999. http://www.rsasecurity.com/rsalabs/challenges/ factoring/rsa155.html.
[68] R. Sandhu. How to Fix the Information Security Crisis: Towards a Business-Driven Discipline of Good Enough Security, 2002.
[69] M. Schiffman. Libnet. http://www.packetfactory.net/Projects/Libnet/.
[70] M. Schiffman. *Building Open Source Network Security Tools*. Wiley, 2003.
[71] W. Schindler. A Timing Attack against RSA with the Chinese Remainder Theorem. *CHES 2000*, pages 109–124, 2000.
[72] B. Schneier. *Applied Cryptography*. Wiley, 1996.
[73] A. Shain. Phishing to steal your information, 25 juli 2003.
[74] A. Shamir and E. Tromer. Factoring Large Numbers with the TWIRL Device, 2003.
[75] V. Shoup. A Proposal for an ISO Standard for Public Key Encryption (version 2.0), 2001.
[76] R. Silverman. Exposing the mythical MIPS year. *Computer*, 32(8):22–26, 1999.
[77] R. Silverman. A Cost-Based Security Analysis of Symmetric and Asymmetric Key Lengths, 2000. RSA Security Bulletin, number 13.
[78] D. Song. Dsniff, 2000. http://monkey.org/ dugsong/dsniff/.
[79] Microsoft Security Bulletin MS02-050: Certificate Validation Flaw Could Enable Identity Spoofing (Q329115), 2002. http://www.microsoft.com/technet/security/bulletin/MS02-050.mspx.
[80] SSL Spoofing vulnerabilities in SSL Client Applications, 2002. http://www.thoughtcrime.org/ie-ssl-chain.txt.
[81] Internet Explorer URL Spoofing Vulnerability, 2003. http://secunia.com/advisories/10395/.
[82] Microsoft Security Bulletin MS04-004: Cumulative Security Update for Internet Explorer (832894), 2004. http://www.microsoft.com/technet/security/bulletin/MS04-004.mspx.
[83] SSLSniff, 2002. http://www.thoughtcrime.org/ie.html.
[84] A. Stubblefield, J. Ioannidis, and A. Rubin. Using the Fluhrer, Mantin, and Shamir Attack to Break WEP, 2001.
[85] A. Tanenbaum. *Computer Networks*. Prentice Hall, 1996. 3th Edition.
[86] Tcpdump syntax. http://www.tcpdump.org/tcpdump_man.html.
[87] S. Vaudenay. Security Flaws Induced by CBC Padding - Applications to SSL, IPSEC, WTLS, ... *Advances in Cryptology - EUROCRYPT'02*, LNCS 2332:534–545, 2002.
[88] J. von Neumann. Various Techniques used in Connection with Random Digits. *Applied Mathematics Series*, 12:36–38, 1951.
[89] M. Vuagnoux. The CBC SSL/TLS Timing Attack, 2003.
[90] M. Wagner and B. Schneier. Analysis of the SSL Protocol. *The Second USENIX Workshop on Electronic Commerce Proceedings*, pages 29–40, 1996.
[91] S. Whalen. An Introduction to ARP spoofing, 2001. http://packetstormsecurity.nl/papers/protocols/intro to arp spoofing.pdf.
[92] WinSSLMiM. http://www.securiteinfo.com/outils/WinSSLMiM.shtml.
[93] E. Z. Ye, Y. Yuan, and S. Smith. Web Spoofing Revisited: SSL and Beyond, 2002. Technical Report TR2002-417.
[94] A. Zaidman. Internetbeveiliging zonder privacycompromissen? Master's thesis, Universiteit Antwerpen, 2002.

Bijlage A

SSL Details

A.1 SSL/TLS versleutelsets

Versleutelset	Authen-ticatie	Sleutel-uitwisseling	Encryptie	Digest	Nummer
TLS_RSA_WITH_NULL_MD5	RSA	RSA	NULL	MD5	0x0001
TLS_RSA_WITH_NULL_SHA	RSA	RSA	NULL	SHA	0x0002
TLS_RSA_EXPORT_WITH_RC4_40_MD5	RSA	RSA_EXPORT	RC4_40	MD5	0x0003
TLS_RSA_WITH_RC4_128_MD5	RSA	RSA	RC4_128	MD5	0x0004
TLS_RSA_WITH_RC4_128_SHA	RSA	RSA	RC4_128	SHA	0x0005
TLS_RSA_EXPORT_WITH_RC2_CBC_40_MD5	RSA	RSA_EXPORT	RC2_CBC_40	MD5	0x0006
TLS_RSA_WITH_IDEA_CBC_SHA	RSA	RSA	IDEA_CBC	SHA	0x0007
TLS_RSA_EXPORT_WITH_DES40_CBC_SHA	RSA	RSA_EXPORT	DES40_CBC	SHA	0x0008
TLS_RSA_WITH_DES_CBC_SHA	RSA	RSA	DES_CBC	SHA	0x0009
TLS_RSA_WITH_3DES_EDE_CBC_SHA	RSA	RSA	3DES_EDE_CBC	SHA	0x000A

Tabel A.1. De belangrijkste SSL/TLS versleutelsets met RSA sleuteluitwisseling

A.2 SSL/TLS alerts

Alert	Level	SSL Versie	Nummer
close_notify	warning/fatal	SSLv3/TLS	0
unexpected_message	fatal	SSLv3/TLS	10
bad_record_mac	fatal	SSLv3/TLS	20
decryption_failed	fatal	TLS	21
record_overflow	fatal	TLS	22
decompression_failure	fatal	SSLv3/TLS	30
handshake_failure	fatal	SSLv3/TLS	40
no_certificate	warning/fatal	SSLv3	41
bad_certificate	warning/fatal	SSLv3/TLS	42
unsupported_certificate	warning/fatal	SSLv3/TLS	43
certificate_revoked	warning/fatal	SSLv3/TLS	44
certificate_expired	warning/fatal	SSLv3/TLS	45
certificate_unknown	warning/fatal	SSLv3/TLS	46
illegal_parameter	fatal	SSLv3/TLS	47
unknown_ca	fatal	TLS	48
access_denied	fatal	TLS	49
decode_error	fatal	TLS	50
decrypt_error	warning/fatal	TLS	51
export_restriction	fatal	TLS	60
protocol_version	fatal	TLS	70
insufficient_security	fatal	TLS	71
internal_error	fatal	TLS	80
user_cancelled	fatal	TLS	90
no_renegotiation	warning	TLS	100

Tabel A.2. SSL/TLS Alerts

Bijlage B

Ethereal dumps

B.1 Inleiding

Ethereal is een vrij verkrijgbaar *networksniffer* die naar eigen zeggen "*gebruikt wordt door netwerkexperten over de hele wereld bij het debuggen en analyseren van software en protocolontwikkeling, en voor educatie*" [20].

Het is ook voor deze doeleinden dat wij[1] het doorheen deze studie gebruikten, als hulpmiddel bij onderzoek, en om SSL connecties te analyseren, maar ook om onze aanvalstool te debuggen.

Specifiek in deze bijlage gebruiken we Ethereal echter om bepaalde netwerkgebeurtenissen gedetailleerd weer te geven en te illustreren. We doen dit door Ethereal op het netwerk te laten "snuffelen", terwijl we een bepaald experiment uitvoeren. Op die manier "vangt" Ethereal elk netwerkpakket dat een gevolg is van dit experiment, waarop we deze pakketten achteraf in detail kunnen bestuderen. Daarnaast kan Ethereal de gevangen pakketten ook uitschrijven naar een bestand, dat is dan wat wij een *Ethereal dump* noemen, en wat we in deze bijlage tonen.

Eerst geven we zo'n Ethereal dump van een bepaalde MITM aanval (ARP vergiftiging), waarna een dump van de eigenlijke versleutelsetverlagingsaanval wordt gegeven.

Als laatste wordt een dump gegeven van een poging tot versleutelsetverlaging, die mislukt omwille van de geïmplementeerde veiligheidsmaatregelen.

Telkens wordt, waar nodig, meer uitleg gegeven bij de dumps. Deze uitleg staat altijd *onder* de dump.

[1]Hoewel wij niet durven stellen netwerkexpert te zijn!

Het netwerk dat gebruikt wordt is een virtueel netwerk, dat in Sectie 6.8 meer uitgebreid besproken wordt. In de volgende tabel herhalen we nog eens de belangrijkste eigenschappen van de hosts op dit netwerk:

Host	**IP**	**Hardware adres**
Server	192.168.0.104	00:0c:29:ae:6a:25
Client	192.168.0.105	00:0c:29:95:58:d1
Aanvaller	192.168.0.106	00:0c:29:81:fe:06

Daarnaast gebruiken we altijd '`openssl s_server`' als SSL server implementatie en '`openssl s_client`' als SSL client implementatie.

B.2 ARP vergiftiging

We bespreken hier een Ethereal dump van een MITM aanval onder de vorm van ARP vergiftiging.

```
No.Time       Source             Destination        Protocol    Info
 1 0.000000   00:0c:29:95:58:d1  ff:ff:ff:ff:ff:ff   ARP    Who has 192.168.0.104?  Tell 192.168.0.10
 2 0.004513   00:0c:29:ae:6a:25  00:0c:29:95:58:d1   ARP    192.168.0.104 is at 00:0c:29:ae:6a:25
 3 0.004517   192.168.0.105      192.168.0.104       ICMP   Echo (ping) request
 4 0.029150   192.168.0.104      192.168.0.105       ICMP   Echo (ping) reply
 5 4.560555   00:0c:29:ae:6a:25  00:0c:29:95:58:d1   ARP    Who has 192.168.0.105?  Tell 192.168.0.104
 6 4.560559   00:0c:29:95:58:d1  00:0c:29:ae:6a:25   ARP    192.168.0.105 is at 00:0c:29:95:58:d1
```

In deze dump is het resultaat van het ping commando te zien, uitgevoerd van client naar server:

ping -c 1 192.168.0.104

Dit ping commando zorgt ervoor dat er een mapping van de client in de ARP cache van de server komt, en vice versa.

Merk op dat dit ping commando normaal door de aanvaller vervalst wordt, dit maakt echter geen verschil in praktijk.

Vervolgens zal de aanvaller starten met de ARP vergiftiging:

```
arpspoof -t 192.168.0.104 192.168.0.105
arpspoof -t 192.168.0.105 192.168.0.104
```

Arpspoof krijgt de IP adressen van de slachtoffers mee, en zal dus eerst de bijhorende hardware adressen te weten moeten komen.

```
 7 16.281062   00:0c:29:81:fe:06   ff:ff:ff:ff:ff:ff   ARP     Who has 192.168.0.105?  Tell 192.168.0.106
 8 16.282475   00:0c:29:81:fe:06   ff:ff:ff:ff:ff:ff   ARP     Who has 192.168.0.104?  Tell 192.168.0.106
 9 16.284841   00:0c:29:95:58:d1   00:0c:29:81:fe:06   ARP     192.168.0.105 is at 00:0c:29:95:58:d1
10 16.285226   192.168.0.106       192.168.0.105       BOOTP   [Malformed Packet]
11 16.289537   192.168.0.105       192.168.0.106       ICMP    Destination unreachable
12 16.305418   00:0c:29:ae:6a:25   00:0c:29:81:fe:06   ARP     192.168.0.104 is at 00:0c:29:ae:6a:25
13 16.305474   192.168.0.106       192.168.0.104       BOOTP   [Malformed Packet]
14 16.308100   192.168.0.104       192.168.0.106       ICMP    Destination unreachable
```

In deze dump zien we hoe dat *resolven* van de hardware adressen gebeurt.

Arpspoof doet dit door een speciaal `UDP` pakket te sturen, dat Ethereal interpreteert als een fout `BOOTP` pakket. Dit dwingt de Linux kernel om een `ARP Request` uit te sturen om het bijhorende hardware adres te weten te komen.

Eenvoudigweg een eigen gemaakte `ARP Request` uitsturen helpt niet, aangezien de Linux kernel de daarop teruggestuurde `ARP Reply` zal negeren, wegens niet gevraagd (en omdat er nog geen mapping aanwezig is om aan te passen).

```
15 17.303794   00:0c:29:81:fe:06   00:0c:29:95:58:d1   ARP   192.168.0.104 is at 00:0c:29:81:fe:06
16 17.305122   00:0c:29:81:fe:06   00:0c:29:ae:6a:25   ARP   192.168.0.105 is at 00:0c:29:81:fe:06
17 19.310730   00:0c:29:81:fe:06   00:0c:29:ae:6a:25   ARP   192.168.0.105 is at 00:0c:29:81:fe:06
18 19.311799   00:0c:29:81:fe:06   00:0c:29:95:58:d1   ARP   192.168.0.104 is at 00:0c:29:81:fe:06
19 21.165406   00:0c:29:ae:6a:25   00:0c:29:81:fe:06   ARP   Who has 192.168.0.106?  Tell 192.168.0.104
20 21.165540   00:0c:29:81:fe:06   00:0c:29:ae:6a:25   ARP   192.168.0.106 is at 00:0c:29:81:fe:06
21 21.198798   00:0c:29:95:58:d1   00:0c:29:81:fe:06   ARP   Who has 192.168.0.106?  Tell 192.168.0.105
22 21.198831   00:0c:29:81:fe:06   00:0c:29:95:58:d1   ARP   192.168.0.106 is at 00:0c:29:81:fe:06
23 21.320505   00:0c:29:81:fe:06   00:0c:29:95:58:d1   ARP   192.168.0.104 is at 00:0c:29:81:fe:06
24 21.321537   00:0c:29:81:fe:06   00:0c:29:ae:6a:25   ARP   192.168.0.105 is at 00:0c:29:81:fe:06
25 23.331510   00:0c:29:81:fe:06   00:0c:29:ae:6a:25   ARP   192.168.0.105 is at 00:0c:29:81:fe:06
26 23.333149   00:0c:29:81:fe:06   00:0c:29:95:58:d1   ARP   192.168.0.104 is at 00:0c:29:81:fe:06
```

In deze dump zien we dan tenslotte de echte ARP vergiftiging gebeuren, er worden voortdurend vervalste `ARP Reply`'s uitgestuurd.

B.3 Versleutelsetverlagingsaanval

In deze sectie geven we een gedetailleerd verloop van een SSL connectie waarop een versleutelsetverlagingsaanval wordt uitgevoerd.

De MITM aanval die de aanvaller in staat stelt om de connectie te manipuleren, is een ARP vergiftigingsaanval, zoals in de vorige sectie besproken, en wordt hier niet meer uitgebreid getoond.

B.3.1 Overzicht

Een overzicht van een complete aanval ziet er dan als volgt uit:

```
No.Time       Source              Destination        Protocol    Info
 1 0.000000   00:0c:29:81:fe:06   00:0c:29:ae:6a:25   ARP   192.168.0.105 is at 00:0c:29:81:fe:06
 2 0.010025   00:0c:29:81:fe:06   00:0c:29:95:58:d1   ARP   192.168.0.104 is at 00:0c:29:81:fe:06
 3 2.010096   00:0c:29:81:fe:06   00:0c:29:ae:6a:25   ARP   192.168.0.105 is at 00:0c:29:81:fe:06
 4 2.020826   00:0c:29:81:fe:06   00:0c:29:95:58:d1   ARP   192.168.0.104 is at 00:0c:29:81:fe:06
 5 4.021354   00:0c:29:81:fe:06   00:0c:29:ae:6a:25   ARP   192.168.0.105 is at 00:0c:29:81:fe:06
 6 4.030880   00:0c:29:81:fe:06   00:0c:29:95:58:d1   ARP   192.168.0.104 is at 00:0c:29:81:fe:06
 7 6.031442   00:0c:29:81:fe:06   00:0c:29:ae:6a:25   ARP   192.168.0.105 is at 00:0c:29:81:fe:06
 8 6.041263   00:0c:29:81:fe:06   00:0c:29:95:58:d1   ARP   192.168.0.104 is at 00:0c:29:81:fe:06
```

Hier zien we vervalste `ARP Reply`'s tengevolge van de ARP vergiftiging die bezig is.

```
 9 6.808110   192.168.0.105   192.168.0.104   TCP   1034 > 443 [SYN] Seq=909981042 Ack=0 Win=5840 Len=0
10 6.808530   192.168.0.105   192.168.0.104   TCP   1034 > 443 [SYN] Seq=909981042 Ack=0 Win=5840 Len=0
```

```
11 6.810158   192.168.0.104   192.168.0.105   TCP   443 > 1034 [SYN, ACK] Seq=2737772029 Ack=909981043 Win=5792 Len=0
12 6.810521   192.168.0.104   192.168.0.105   TCP   443 > 1034 [SYN, ACK] Seq=2737772029 Ack=909981043 Win=5792 Len=0
13 6.812327   192.168.0.105   192.168.0.104   TCP   1034 > 443 [ACK] Seq=909981043 Ack=2737772030 Win=5840 Len=0
14 6.799793   192.168.0.105   192.168.0.104   TCP   1034 > 443 [ACK] Seq=909981043 Ack=2737772030 Win=5840 Len=0
```

In deze dump opent de client een TCP connectie naar de server, met behulp van een *three-way handshake*. Merk op dat deze pakketten (en zoals we verder zullen zien *elk* pakket dat tussen client en server vloeit) dubbel worden getoond. Dit is geen fout van Ethereal, maar een gevolg van de MITM aanval, waarin elk pakket eerst naar de aanvaller wordt gestuurd, waarna de aanvaller dit pakket *forward*.

We bekijken de twee eerste pakketten in meer detail:

```
Frame 9 (74 bytes on wire, 74 bytes captured)
Ethernet II, Src:00:0c:29:95:58:d1, Dst:  00:0c:29:81:fe:06
Internet Protocol, Src Addr:  192.168.0.105 (192.168.0.105), Dst Addr:  192.168.0.104
(192.168.0.104)
Transmission Control Protocol, Src Port:  1033 (1033), Dst Port:  443 (443), Seq:
909981042, Ack:  0, Len:  0
Frame 10 (74 bytes on wire, 74 bytes captured)
Ethernet II, Src:00:0c:29:81:fe:06, Dst:  00:0c:29:ae:6a:25
Internet Protocol, Src Addr:  192.168.0.105 (192.168.0.105), Dst Addr:  192.168.0.104
(192.168.0.104)
Transmission Control Protocol, Src Port:  1033 (1033), Dst Port:  443 (443), Seq:
909981042, Ack:  0, Len:  0
```

Op IP niveau is er geen enkel verschil tussen de pakketten, maar als we deze pakketten in meer detail bekijken op het datalink niveau, zien we duidelijk dat de hardware adressen (in *cursief*) verschillen, en dat de pakketten inderdaad eerst naar de aanvaller gaan, alvorens doorgestuurd te worden naar hun echte bestemming.

```
15 6.823874   192.168.0.105   192.168.0.104   SSLv2   Client Hello
16 6.825096   192.168.0.105   192.168.0.104   SSLv2   Client Hello
17 6.827358   192.168.0.104   192.168.0.105   TCP     443 > 1034 [ACK] Seq=2737772030 Ack=909981173 Win=6432 Len=0
18 6.827809   192.168.0.104   192.168.0.105   TCP     443 > 1034 [ACK] Seq=2737772030 Ack=909981173 Win=6432 Len=0
19 6.844400   192.168.0.104   192.168.0.105   SSLv3   Server Hello, Certificate, Server Key Exchange, Server Hello Done
20 6.845551   192.168.0.104   192.168.0.105   SSLv3   Server Hello, Certificate, Server Key Exchange, Server Hello Done
21 6.857369   192.168.0.105   192.168.0.104   TCP     1034 > 443 [ACK] Seq=909981173 Ack=2737772965 Win=7480 Len=0
22 6.857399   192.168.0.105   192.168.0.104   SSLv3   Client Key Exchange, Change Cipher Spec, Encrypted Handshake Message
23 6.853133   192.168.0.105   192.168.0.104   TCP     1034 > 443 [ACK] Seq=909981173 Ack=2737772965 Win=7480 Len=0
24 6.873779   192.168.0.105   192.168.0.104   SSLv3   Client Key Exchange, Change Cipher Spec, Encrypted Handshake Message
25 6.876297   192.168.0.104   192.168.0.105   SSLv3   Change Cipher Spec, Encrypted Handshake Message
26 6.890789   192.168.0.104   192.168.0.105   SSLv3   Change Cipher Spec, Encrypted Handshake Message
27 6.913171   192.168.0.105   192.168.0.104   TCP     1034 > 443 [ACK] Seq=909981313 Ack=2737773032 Win=7480 Len=0
28 6.913695   192.168.0.105   192.168.0.104   TCP     1034 > 443 [ACK] Seq=909981313 Ack=2737773032 Win=7480 Len=0
```

Hier zien we de eigenlijke SSL handshake, uitgevoerd tussen client en server en gemanipuleerd door de aanvaller. De manipulatie zelf is in dit overzicht niet te zien (buiten het feit dat alle SSL berichten gedupliceerd zijn, want eerst langs aanvaller passeren), maar wordt in de volgende secties in meer detail getoond.

Merk op dat na het `ChangeCipherSpec` bericht overgeschakeld wordt op de genegotieerde versleutelset, en de rest van de handshake (het `Finished` bericht) geëncrypteerd wordt.

Let ook op de `TCP ACK` berichten, de TCP acknowledgements.

```
29 8.041006   00:0c:29:81:fe:06   00:0c:29:ae:6a:25   ARP    192.168.0.105 is at 00:0c:29:81:fe:06
30 8.051919   00:0c:29:81:fe:06   00:0c:29:95:58:d1   ARP    192.168.0.104 is at 00:0c:29:81:fe:06
31 8.218655   192.168.0.105       192.168.0.104       SSLv3  Application Data
32 8.219133   192.168.0.105       192.168.0.104       SSLv3  Application Data
33 8.264082   192.168.0.104       192.168.0.105       TCP    443 > 1034 [ACK] Seq=2737773032 Ack=909981336 Win=6432 Len=0
34 8.264535   192.168.0.104       192.168.0.105       TCP    443 > 1034 [ACK] Seq=2737773032 Ack=909981336 Win=6432 Len=0
```

Hier zien we de eigenlijke SSL connectie, waarover client en server op een beveiligde manier data uitwisselen (de `Application Data` berichten).

De ARP pakketten geven aan dat de ARP vergiftiging nog steeds aan de gang is.

```
35 9.613962   192.168.0.105   192.168.0.104   SSLv3   Encrypted Alert
36 9.614196   192.168.0.105   192.168.0.104   SSLv3   Encrypted Alert
37 9.615170   192.168.0.105   192.168.0.104   TCP     1034 > 443 [FIN, ACK] Seq=909981359 Ack=2737773032 Win=7480 Len=0
38 9.615678   192.168.0.105   192.168.0.104   TCP     1034 > 443 [FIN, ACK] Seq=909981359 Ack=2737773032 Win=7480 Len=0
39 9.638725   192.168.0.104   192.168.0.105   TCP     443 > 1034 [ACK] Seq=2737773032 Ack=909981359 Win=6432 Len=0
40 9.638729   192.168.0.104   192.168.0.105   TCP     443 > 1034 [FIN, ACK] Seq=2737773032 Ack=909981360 Win=6432 Len=0
41 9.626099   192.168.0.104   192.168.0.105   TCP     443 > 1034 [ACK] Seq=2737773032 Ack=909981359 Win=6432 Len=0
42 9.626283   192.168.0.104   192.168.0.105   TCP     443 > 1034 [FIN, ACK] Seq=2737773032 Ack=909981360 Win=6432 Len=0
43 9.627416   192.168.0.105   192.168.0.104   TCP     1034 > 443 [ACK] Seq=909981360 Ack=2737773033 Win=7480 Len=0
44 9.627857   192.168.0.105   192.168.0.104   TCP     1034 > 443 [ACK] Seq=909981360 Ack=2737773033 Win=7480 Len=0
```

In dit overzicht vinden we de `TCP FIN` berichten, die aanduiden dat de TCP connectie op een normale manier afgesloten wordt.

Merk op dat deze voorafgegaan worden door een speciaal SSL bericht, een alert. Dit is de `close_notify`, die aangeeft dat de client de connectie beëindigt. Dit bericht is geëncrypteerd, want gestuurd over de beveiligde connectie, en voorkomt truncatie-aanvallen.

B.3.2 Manipulatie van eerste keuze

In deze sectie laten we in meer detail de `ClientHello` zien vóór en na een "manipulatie van de eerst keuze".

Originele ClientHello

```
Secure Socket Layer
    SSLv2 Record Layer: Client Hello
        Length: 128
        Handshake Message Type: Client Hello (1)
        Version: SSL 3.0 (0x0300)
        Cipher Spec Length: 87
        Session ID Length: 0
        Challenge Length: 32
        Cipher Specs (29 specs)
            Cipher Spec: TLS_DHE_RSA_WITH_3DES_EDE_CBC_SHA (0x000016)
            Cipher Spec: TLS_DHE_DSS_WITH_3DES_EDE_CBC_SHA (0x000013)
            ...
            Cipher Spec: TLS_RSA_EXPORT_WITH_RC2_CBC_40_MD5 (0x000006)
            Cipher Spec: TLS_RSA_EXPORT_WITH_RC4_40_MD5 (0x000003)
            Cipher Spec: SSL2_RC2_CBC_128_CBC_WITH_MD5 (0x040080)
            Cipher Spec: SSL2_RC4_128_EXPORT40_WITH_MD5 (0x020080)
        Challenge
```

Aangepaste ClientHello

```
Secure Socket Layer
    SSLv2 Record Layer: Client Hello
        Length: 128
        Handshake Message Type: Client Hello (1)
        Version: SSL 3.0 (0x0300)
        Cipher Spec Length: 87
        Session ID Length: 0
        Challenge Length: 32
        Cipher Specs (29 specs)
            Cipher Spec: TLS_RSA_EXPORT_WITH_RC4_40_MD5 (0x000003)
            Cipher Spec: TLS_DHE_DSS_WITH_3DES_EDE_CBC_SHA (0x000013)
            ...
            Cipher Spec: TLS_RSA_EXPORT_WITH_RC2_CBC_40_MD5 (0x000006)
            Cipher Spec: TLS_RSA_EXPORT_WITH_RC4_40_MD5 (0x000003)
            Cipher Spec: SSL2_RC2_CBC_128_CBC_WITH_MD5 (0x040080)
            Cipher Spec: SSL2_RC4_128_EXPORT40_WITH_MD5 (0x020080)
        Challenge
```

Merk op dat de aanvaller eenvoudigweg de eerste versleutelset heeft overschreven, waardoor de `TLS_RSA_EXPORT_WITH_RC4_40_MD5` versleutelset tweemaal voorkomt in de lijst.

Zoals eerder besproken zou een nog elegantere aanval de eerste versleutelset niet overschrijven, met een duplicaat als gevolg, maar de eerste en de export versleutelset van plaats wisselen.

B.3.3 NULL overschrijving

Naast een "manipulatie van de eerst keuze", kunnen we ook kiezen voor "`NULL` overschrijving".

Originele ClientHello

```
Secure Socket Layer
    SSLv2 Record Layer: Client Hello
        Length: 128
        Handshake Message Type: Client Hello (1)
        Version: SSL 3.0 (0x0300)
        Cipher Spec Length: 87
        Session ID Length: 0
        Challenge Length: 32
        Cipher Specs (29 specs)
            Cipher Spec: TLS_DHE_RSA_WITH_3DES_EDE_CBC_SHA (0x000016)
            Cipher Spec: TLS_DHE_DSS_WITH_3DES_EDE_CBC_SHA (0x000013)
            ...
            Cipher Spec: TLS_RSA_EXPORT_WITH_RC2_CBC_40_MD5 (0x000006)
            Cipher Spec: TLS_RSA_EXPORT_WITH_RC4_40_MD5 (0x000003)
            Cipher Spec: SSL2_RC2_CBC_128_CBC_WITH_MD5 (0x040080)
            Cipher Spec: SSL2_RC4_128_EXPORT40_WITH_MD5 (0x020080)
        Challenge
```

Aangepaste ClientHello

```
Secure Socket Layer
    SSLv2 Record Layer: Client Hello
        Length: 128
        Handshake Message Type: Client Hello (1)
        Version: SSL 3.0 (0x0300)
```

```
Cipher Spec Length: 87
Session ID Length: 0
Challenge Length: 32
Cipher Specs (29 specs)
    Cipher Spec: TLS_RSA_EXPORT_WITH_RC4_40_MD5 (0x000003)
    Cipher Spec: TLS_NULL_WITH_NULL_NULL (0x000000)
    ...
    Cipher Spec: TLS_NULL_WITH_NULL_NULL (0x000000)
    Cipher Spec: TLS_NULL_WITH_NULL_NULL (0x000000)
    Cipher Spec: TLS_NULL_WITH_NULL_NULL (0x000000)
    Cipher Spec: TLS_NULL_WITH_NULL_NULL (0x000000)
Challenge
```

We zien duidelijk dat de eerste versleutelset met de exportversleutelset wordt overschreven, en alle andere versleutelsets met de ongeldige `NULL` versleutelset.

B.4 Versleutelsetverlagingsaanval bij een beschermde Ephemeral RSA handshake

In deze sectie stellen we een poging tot een versleutelsetverlagingsaanval voor, die faalt omwille van de bescherming die zowel server als client hebben ingebouwd in het `ServerKeyExchange` bericht.

```
No.Time       Source              Destination        Protocol    Info
 1 0.000000   00:0c:29:81:fe:06   00:0c:29:ae:6a:25   ARP     192.168.0.105 is at 00:0c:29:81:fe:06
 2 0.036542   00:0c:29:81:fe:06   00:0c:29:95:58:d1   ARP     192.168.0.104 is at 00:0c:29:81:fe:06
 3 2.006384   00:0c:29:81:fe:06   00:0c:29:ae:6a:25   ARP     192.168.0.105 is at 00:0c:29:81:fe:06
 4 2.046209   00:0c:29:81:fe:06   00:0c:29:95:58:d1   ARP     192.168.0.104 is at 00:0c:29:81:fe:06
```

De aanvaller start zijn aanval met een ARP vergiftiging.

```
 5 2.468777   192.168.0.105   192.168.0.104   TCP     1106 > 443 [SYN] Seq=3926828298 Ack=0 Win=5840 Len=0
 6 2.438276   192.168.0.105   192.168.0.104   TCP     1106 > 443 [SYN] Seq=3926828298 Ack=0 Win=5840 Len=0
 7 2.441042   192.168.0.104   192.168.0.105   TCP     443 > 1106 [SYN, ACK] Seq=1228485403 Ack=3926828299 Win=5792 Len=0
 8 2.441265   192.168.0.104   192.168.0.105   TCP     443 > 1106 [SYN, ACK] Seq=1228485403 Ack=3926828299 Win=5792 Len=0
 9 2.445486   192.168.0.105   192.168.0.104   TCP     1106 > 443 [ACK] Seq=3926828299 Ack=1228485404 Win=5840 Len=0
10 2.445496   192.168.0.105   192.168.0.104   SSLv2   Client Hello
11 2.447859   192.168.0.105   192.168.0.104   TCP     1106 > 443 [ACK] Seq=3926828299 Ack=1228485404 Win=5840 Len=0
12 2.449750   192.168.0.105   192.168.0.104   SSLv2   Client Hello
13 2.462821   192.168.0.104   192.168.0.105   TCP     443 > 1106 [ACK] Seq=1228485404 Ack=3926828447 Win=5792 Len=0
14 2.462961   192.168.0.104   192.168.0.105   TCP     443 > 1106 [ACK] Seq=1228485404 Ack=3926828447 Win=5792 Len=0
15 2.554973   192.168.0.104   192.168.0.105   SSLv3   Server Hello, Certificate, Server Key Exchange, Server Hello Done
16 2.529482   192.168.0.104   192.168.0.105   SSLv3   Server Hello, Certificate, Server Key Exchange, Server Hello Done
17 2.543620   192.168.0.105   192.168.0.104   TCP     1106 > 443 [ACK] Seq=3926828447 Ack=1228486339 Win=7480 Len=0
18 2.543761   192.168.0.105   192.168.0.104   TCP     1106 > 443 [ACK] Seq=3926828447 Ack=1228486339 Win=7480
```

In deze dump is te zien dat alles tot nu toe als gepland verloopt voor de aanvaller. Hij heeft de `ClientHello` vervalst, en wacht nu vol ongeduld op het `ClientKeyExchange` bericht, waarin de (zwak) geëncrypteerde `pre_master_secret` zit.

Wat hij echter niet weet, is dat zowel client als server een beschermde `ServerKeyExchange` gebruiken, de server heeft in zijn digitale handtekening in dit bericht ook de `ClientHello` verwerkt die hij eerder ontving.

Een aanvaller kan dit echter onmogelijk zien aan de `ServerKeyExchange`, aangezien deze digitale handtekening aan de hand van een message digest gebeurt, en dus van constante lengte is, `ClientHello` mee verwerkt of niet.

```
19 2.590554   192.168.0.105   192.168.0.104   SSLv3   Alert (Level: Fatal, Description: Handshake Failure)
20 2.590656   192.168.0.105   192.168.0.104   TCP     1106 > 443 [RST, ACK] Seq=3926828454 Ack=1228486339 Win=7480 Len=0
21 2.563523   192.168.0.105   192.168.0.104   SSLv3   Alert (Level: Fatal, Description: Handshake Failure)
22 2.684620   192.168.0.104   192.168.0.105   TCP     443 > 1106 [ACK] Seq=1228486339 Ack=3926828454 Win=5792 Len=0
23 2.576214   192.168.0.105   192.168.0.104   TCP     1106 > 443 [RST, ACK] Seq=3926828454 Ack=1228486339 Win=7480 Len=0
24 2.576680   192.168.0.104   192.168.0.105   TCP     443 > 1106 [ACK] Seq=1228486339 Ack=3926828454 Win=5792 Len=0
25 2.578541   192.168.0.105   192.168.0.104   TCP     1106 > 443 [RST] Seq=3926828454 Ack=0 Win=0 Len=0
26 2.578980   192.168.0.105   192.168.0.104   TCP     1106 > 443 [RST] Seq=3926828454 Ack=0 Win=0
```

> De aanvaller slaakt een verraste kreet. De aanval mislukt?! De client stuurt de server immers een fatale alert (Handshake failed), en beëindigt de onderliggende TCP connectie onmiddellijk met een `TCP RST`.

De extra bescherming in het `ServerKeyExchange` bericht heeft de versleutelsetverlagingsaanval succesvol voorkomen.

Lijst van tabellen

2.1 Een kinderlijk geheimschrift 10

2.2 Symmetrisch versus asymmetrisch 20

4.1 Cryptografische operaties in SSL en hun algoritmen 47

5.1 Rechtstreekse aanvallen op het SSL Protocol 70

5.2 Onrechtstreekse aanvallen op het SSL Protocol 84

5.3 Cryptografische operaties in SSL en hun algoritmen herbekeken 87

5.4 Kost voor het bruteforcen van asymmetrische RSA sleutels 94

5.5 Kost voor het bruteforcen van symmetrische sleutels 94

6.1 Toepasbaarheid van MITM aanvallen 104

8.1 SecuritySpace onderzoek naar SSL versies 131

8.2 SecuritySpace onderzoek naar SSL versleutelsets 132

8.3 Eigen onderzoek naar SSL servers 135

8.4 Meestgebruikte browsers 136

8.5 Welke browsers ondersteunen de EXP-RC4-MD5 export versleutelset? . . . 137

A.1 De belangrijkste SSL/TLS versleutelsets met RSA sleuteluitwisseling 157

A.2 SSL/TLS Alerts 158

Lijst van figuren

2.1 Symmetrische encryptie 9

2.2 Asymmetrische encryptie 17

2.3 Een eenvoudige digitale handtekening 22

2.4 Digitale handtekening m.b.v. message digest 23

2.5 Een certificaat 24

3.1 Protocol stack, zonder en met SSL 29

3.2 Een eerste handshake 33

3.3 Een dynamische handshake 34

3.4 Handshake met integriteitscontrole 35

3.5 Handshake met sleutelafleiding 36

3.6 De SSL Handshake 38

3.7 De SSL Handshake bij sessies 40

3.8 Het SSL Record Protocol: datafragmentatie en -bescherming 41

4.1 SSL sleutelafleiding 49

4.2 SSL Handshake bij Ephemeral RSA 59

7.1 De SSL Handshake herbekeken 110

7.2 Versleutelsetverlagingsaanval op de SSL Handshake 111

Aantekeningen