dSik Noter

Michael Lind Mortensen, illio, DAT4

23. juni 2009

Indhold

1 Cryptography, Confidentiality **4**
- 1.1 Disposition . . . 4
- 1.2 Details . . . 4
  - 1.2.1 Sikkerhedsmål . . . 4
  - 1.2.2 Kryptologi . . . 5
  - 1.2.3 Stream ciphers . . . 8
  - 1.2.4 Block ciphers . . . 8
  - 1.2.5 Modes of Operation . . . 9
  - 1.2.6 RSA . . . 12
  - 1.2.7 Angreb på RSA . . . 14

2 Cryptography, Authentication **16**
- 2.1 Disposition . . . 16
- 2.2 Details . . . 16
  - 2.2.1 Sikkerhedsmål . . . 16
  - 2.2.2 Kryptologi . . . 17
  - 2.2.3 MAC . . . 20
  - 2.2.4 Hashing . . . 22
  - 2.2.5 RSA . . . 23
  - 2.2.6 Angreb på RSA . . . 24

3 Key Management and Infrastructures **26**
- 3.1 Disposition . . . 26
- 3.2 Details . . . 26
  - 3.2.1 Sikkerhedsmål . . . 26
  - 3.2.2 Key Management . . . 27
  - 3.2.3 KDC . . . 29
  - 3.2.4 PKI (CA) . . . 29
  - 3.2.5 Passwords . . . 33
  - 3.2.6 Angreb på passwords . . . 34
  - 3.2.7 Biometrics . . . 36

4 Network Security **38**
- 4.1 Disposition . . . 38
- 4.2 Details . . . 38
  - 4.2.1 Sikkerhedsmål . . . 38
  - 4.2.2 Netværkssikkerhed . . . 39
  - 4.2.3 Authenticated Key Exchange . . . 40
  - 4.2.4 SSL/TLS . . . 42
  - 4.2.5 IPSec (VPN) . . . 44
  - 4.2.6 Firewalls . . . 45
  - 4.2.7 Intrusion Detection Systems . . . 49

5 System Security and Models for Security Policies **51**
5.1 Disposition 51
5.2 Details 51
5.2.1 Sikkerhedsmål 52
5.2.2 Systemsikkerhed 53
5.2.3 Firewalls 53
5.2.4 Intrusion Detection Systems 57
5.2.5 Malware 58
5.2.6 Access Control 60
5.2.7 Sikkerhedspolitiker 62

6 Threats and Pitfalls **65**
6.1 Disposition 65
6.2 Details 65
6.2.1 Sikkerhedsmål 65
6.2.2 Klacificering af Angreb 66
6.2.3 Buffer Overflows 69
6.2.4 Cross-site Scripting 72
6.2.5 SQL Injection 73
6.2.6 Covert Channels 74
6.2.7 Cold-boot 75

1 Cryptography, Confidentiality

1.1 Disposition

1. **Sikkerhedsmål**
   - CIA/CAA
   - Def. sikkert system
2. **Kryptologi**
   - Kryptografiske mål (confidentiality/authentication)
   - Secret-key/Public-key Confidentiality
3. **Stream ciphers**
4. **Block ciphers**
5. **Modes of Operation**
   - ECB
   - CBC
6. **RSA**
   - Elliptical Curve
7. **Angreb på RSA**
   - Exhaustive key search
   - Prime factorization
   - Discrete Logarithms
   - Side-channel attack

1.2 Details

Her gives detaljer for hvert punkt i dispositionen og muligvis mere til.

1.2.1 Sikkerhedsmål

Når vi snakker om IT-sikkerhed er det typisk i relation til at ville forbedre på et systems sikkerhed i en eller anden forstand. Hvad det præcist er vi vil forbedre på sikkerheden kommer dog meget an på hvad vi gerne vil opnå.

CIA/CAA
Til det formål opstiller man typisk en række sikkerhedsmål der samlet repræsenterer alle aspekter IT-sikkerhed dækker. I løbet af kurset har vi arbejdet med tre opdelinger:

Confidentiality: Confidentiality er at personer kun skal have adgang til den information de er berettiget til og intet mere. Dette gælder uanset om informationen sendes, opbevares eller behandles.

Authenticity: Authenticity er at information er authentisk, altså ikke manipuleret eller på anden måde ændret af en uautoriseret person. Dette gælder også mht. til authenticity af afsender, modtager mv. Således en bruger ikke kan nares til at tro han har modtaget kommunikation fra en han i virkeligheden ikke har.

Availability: Availability er at vi ønsker vores systemer er tilgængelige når de skal bruges, således legale brugere kan få adgang til deres data og de services et system tilbyder.

Normalvis formulere litteraturen dog ovenstående tre mål som Confidentiality, Integrity og Availability, således det staver CIA, hvilket man åbenbart synes er popsmart! Principperne er dog mere eller mindre de samme.

Def. sikkert system
For lige at få det på plads, må vi hellere lige definere hvad et sikkert system er. Det er typisk svært eller umuligt at bevise et system er sikkert, i det at vi så skal til at beskrive systemet ud fra en matematisk model og bevise et teorem om at systmet er sikkert under de og de omstændigheder. Typisk kan det ikke lade sig gøre og hvis det kan, er det typisk fordi vi har antaget så mange forkerte ting om vore angribers muligheder, at vi alligevel ikke ender op med et sikkert system, men blot falsk tryghed.
Så der er i virkeligheden ikke rigtig nogen definition på et "sikkert system", så i stedet nøjes vi med en definition på et "sikret system". Dette gør vi ved at definere en overordnet *Sikkerhedspolitik* på baggrund af en *Trusselsmodel*, og implementerer herefter denne sikkerhedspolitik vha. nogle *Sikkerhedsmekanismer*. På den måde får vi, at et sikret system kan beskrives som:

Sikret system = Sikkerhedspolitik + Trusselsmodel + Sikkerhedsmekanismer

1.2.2 Kryptologi

Indenfor dette emne skal vi så kigge på den sikkerhedsmekanisme der er kryptologi. Kryptologi kan deles op i to hoveddiscipliner:

Kryptografi: Videnskaben om at lave koder og ciffere.

Kryptoanalyse: Videnskaben om at bryde koder og ciffere.

Jeg vil komme ind på få kryptoanalytiske ting, men generelt er det kryptografi vi kigger på.

Kryptografiske mål

Generelt kan alt kryptografi deles op i to grupper af de problemer kryptografimekanismen løser. Dette er: Confidentiality og Authenticity.

	Confidentiality	Authenticity
Secret-Key	AES, DES, RC4, IDEA	CBC-MAC, HMAC
Public-Key	RSA, El-Gamal, Ell. curves	RSA, El-Gamal, Ell. Curves
Hash Functions		RIPEMD-160, SHA-1, MD5

Vi fokuserer på Confidentiality i dette emne og fokuserer desuden udelukkende på beregneligt sikre systemer, og altså ikke ubetinget sikre løsninger som one-time-pad o.lign.. De er alligevel ikke relevante i praksis.

Secret-key/Public-key Confidentiality

Kerckhoffs' princip, og til dels Shannon's maxim, fortæller os at vi må antage vore fjender altid kender til vore krypteringsalgoritmer i lige så intime detaljer som os selv, så et cryptosystem må aldrig afhænge af algoritmen i sig selv. I stedet skal vi definere nøgler, som så skal distribueres imellem de betroede parter og bruges i kombination med krypteringsalgoritmen.
Der er generelt to måder at bruge nøgler i krypteringsalgoritmer: Secret-key og Public-key systemer (til tider også kaldt henholdsvis Symmetrisk kryptering og Asymmetrisk kryptering).

Det betyder, at hvis vi har et Secret-key cryptosystem, så krypterer man overordnet set sin meddelse m med nøgle k således:

$$c = E_k(m)$$

Og den modsatte part dekrypterer cifferteksten c, ligeledes med nøgle k, således:

$$m = D_k(c) = D_k(E_k(m))$$

Altså bruger begge parter samme nøgle (hence, betegnelsen Symmetrisk kryptering). Forklaringen her ville dog betyde samme meddelse krypteret igen ville resultere i samme ciffertekst, hvilket ikke er hensigtsmæssigt. Derfor bruger de fleste algoritmer desuden en *nonce* (number-used-once) udover nøglen. En nonce skal således sikre at der tilføjes noget variabilitet til hver ciffertekst og begge parter skal så have mekanismer der kan vælge nonces så de er tilfældige.

Har vi derimod et Public-key cryptosystem, så sørger man for at hver bruger har 2 nøgler, en privat nøgle (sk) og en offentlige nøgle (pk). Den offentlige nøgle gøres herefter offentlig for enhver (typisk ved at uploade den til en af de mange offentlige nøgleservere) og enhver kan herefter kryptere beskeder til en således:

$$c = E_{pk}(m)$$

Men det er kun en selv (aka. ejeren af den private nøgle) der kan dekryptere beskeden igen:

$$m = D_{sk}(c) = D_{sk}(E_{pk}(m))$$

Fælles for begge typer cryptosystemer er at en angriber ikke nødvendigvis behøver bryde krypteringsalgoritmen, men i stedet blot kan lave såkaldt *Exhaustive key search*, hvor han/hun bliver ved med at gætte på nøglen indtil der kommer noget meningsfuldt ud af dekrypteringen. Dette sørger man dog for er usandsynligt i praksis, ved at gøre nøglerne så store, at exhaustive key search tager meget meget lang tid (typisk adskillige år). For Public-key kryptering findes der dog algoritmer der kan udregne den private nøgle sk ud fra den offentlige nøgle pk langt hurtigere end den tid det tager at lave exhaustive key search, og derfor er nøglestørrelserne for Public-key kryptering også langt større end for Secret-key (Typisk siger man minimum 1024bit. GPG anbefaler pt. 2048bit).
Men modsat Secret-key systemer, så behøver man ikke, i Public-key, udveksle nøgler før man kommunikerer sikkert og man risikerer derfor ikke at en angriber opsnapper nøglen i den initielle udveksling. Så hvorfor bruger man ikke Public-key kryptering altid? Fordi det er alt for beregningstungt og derfor langsomt. I stedet bruger man ofte udelukkende Secret-key kryptering, en kombination af de to eller Public-key til meget små ting (f.eks. kryptering af e-mails).
Af praktisk erfaring kan jeg fortælle at det tager godt 9 timer at kryptere 70 GB data med en 2048bit RSA krypteringsnøgle - og sandsynligvis skræmmende langt længere tid at dekryptere igen når jeg engang har tålmodighed

til det.

Det bør dog også nævnes Public-key systemer har nogle fantastiske egenskaber man kan bruge til authentication af forskellig art, men det er udenfor dette emne. Lad os nu tage et kig på nogle Secret-key systemer:

1.2.3 Stream ciphers

Den første type Secret-key cipher jeg vil nævne, vil jeg ikke bruge forfærdelig meget tid på, da den ikke bliver brugt så meget længere og desuden kan efterlignes med block ciphers, som vi kigger på om lidt. Den benævnes dog for komplethedens skyld. Det drejer sig om såkaldte Stream ciphers.

En stream ciffer er kort fortalt en ciffer hvor vi udvider nøglen til en meget længere tilfældigt udseende streng, som vi så herefter bruger til at kryptere som et one-time pad ($c = k \oplus m$). Det kunne f.eks. være et system hvor vi har en 128bit nøgle, som vi så udvider til et tilfældigt udseende 1000bit langt output. Det betyder dog ikke vi får ækvivalenten af en 1000bit nøgle eller at de 1000bit reelt er tilfældige, da der stadig kun er 2^128 mulige outputs. Pointen er dog vi antager angriberen, grundet manglende computerkraft, ikke vil kunne se forskellen og derfor ikke kunne bryde krypteringen. Vha. en stream cipher kan man så kryptere on-the-fly så at sige, således man kan kryptere en bit af gangen og sende den afsted - derved er stream ciphers en oplagt løsning til kryptering af konstant kommunikation som lyd, video, wifi, bluetooth mm.
Et eksempel på en stream ciffer er RC4, som noterne siger bruges i mange web browsere. Udover dette er RC4 også meget kendt for at være blevet brugt i WEP, men brugt på en sådan måde, at det ledte til et dybt usikkert cryptosystem der nu kan brydes på få minutter. Udnyttelsen i WEP skyldes bl.a. at WEP brugte en *Initialization Vector* (IV) på kun 24bit, hvilket gjorde at der var en 50% sandsynlighed for key colition efter 5000 pakker - så vha. et passivt aflytningsangreb kunne RC4 nøglen findes meget nemt, som Scott Fluhrer, Itsik Mantin og Adi Shamir viste i 2001.

1.2.4 Block ciphers

Block ciphers er Secret-key ciphers der krypterer en plaintext i blokke af data, således at den krypterer en blok af gangen. En block cipher består typisk af 3 ting:

1. En krypteringsalgoritme som DES (56bit), 3DES (112bit), AES (128/192/256bit) eller IDEA (128bit) fra PGP

2. Et Mode of operation, som jeg kommer ind på hvad er om lidt
3. En *Initialization Vector* (IV), som varierer fra krypteringsoperation til krypteringsoperation. (OBS: Det er ikke alle Modes of operation der bruger en IV, men de fleste gør)

Hver af krypteringsalgoritmerne har således en nøglelængde (AES har både en 128bit, 192bit og en 256bit variant), men udover dette har de også en blockstørrelse, som er størrelsen af blocks de processerer pr. iteration. Blockstørrelsen er 64bit for DES, 3DES og IDEA, og 128bit for alle varianter af AES.
Lad os nu kigge på de Modes of operation vi har til rådighed.

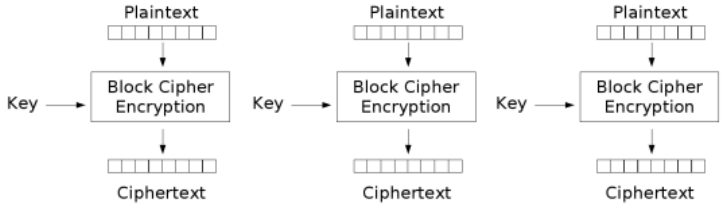
1.2.5 Modes of Operation

Enhver block cipher skal som sagt have en *Mode of operation*. Vi har set flere i løbet af kurset, som f.eks. *Cipher Feedback (CFB)* der var speciel i den forstand den kunne bruges til at skabe block ciphers der opfører sig som stream ciphers. Men en vi specielt har kigget meget på er *Control Block Chaining (CBC)*. Inden jeg gennemgår den vil jeg dog lige give noget motivation ved at vise hvordan man naivt bruger block ciphers og hvorfor CBC er væsentligt bedre end denne løsning - så vi starter lige med at kigge på en mode of operation der hedder *Electronic Code Book (ECB)*.

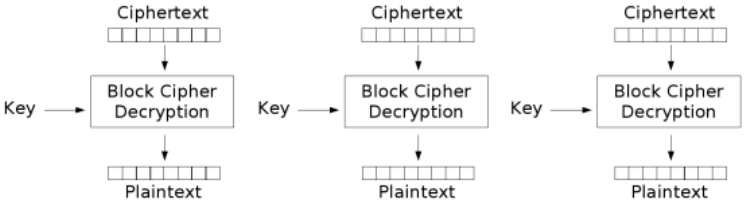
ECB
Electronic Code Book (ECB) er det mest simple Mode of Operation og går blot ud på, at man deler klarteksten op i den pågældende algoritmes blockstørrelser (f.eks. 64bit for DES), således man får separate meddelelser $M_0 \dots M_{t-1}$ hvor t er antallet af blocks. Herefter tager man blot hver block og kører den igennem algoritmen med nøgle k, altså for hvert $i = 0 \dots t-1$:

$$C_i = E_k(M_i)$$

Til sidst har man således en samlet ciphertext C. Decryption er samme princip, blot omvendt. Processerne vises også her *(tak til Wikipedia for alle billeder i dette afsnit)*:

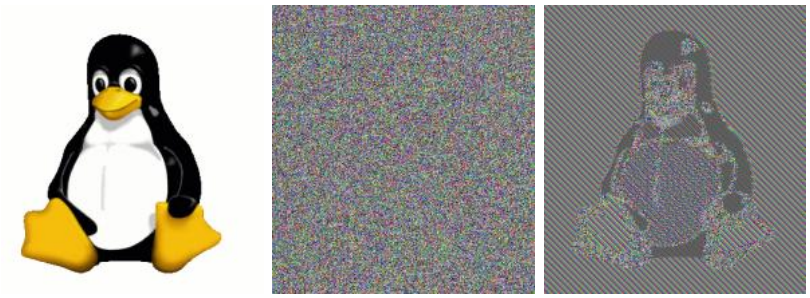


Electronic Codebook (ECB) mode encryption



Electronic Codebook (ECB) mode decryption

Vi har dog nogle store problemer her, da en given klartekst altid resulterer i samme ciffertekst for en nøgle k. Dette betyder at en angriber eksempelvis vil kunne lave såkaldte "replay attacks" hvor han kan gætte sig frem til hvad en ciffertekst kommunikation har af virkning og så gentage den for at få samme virkning igen. Denne slags angreb er blevet brugt til alt fra at overføre penge flere gange, til at snyde i online rollespil for at få flere experience points.
Udover replay attacks gør den manglende randomness også, at visse datamønstre bliver meget tydelige. Dette ses også tydeligt på nedenstående billeder, hvor det første billede er originalen, det næste billede er billedet krypteret med en ordentlige Mode of operation, og den sidste er samme billede krypteret med en ECB Mode of operation.



Det er altså på ingen måde en god ide at bruge ECB i praksis, og som vi kan se ovenfor, så kan valget af Mode of Operation virkelig være essentielt for sikkerheden af ens block cipher - blot at bruge 256bit AES løser ikke alle

ens problemer.

CBC

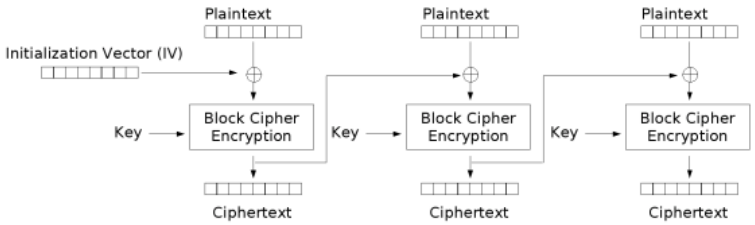
En langt bedre Mode of operation er *Control Block Chaining* (CBC), som virker ved at vi igen deler klarteksten op i den pågældende algoritmes blockstørrelser (f.eks. 128bit for AES), således man får separate meddelelser $M_1 \dots M_t$ hvor t er antallet af blocks i klarteksten og $t+1$ bliver antallet af blocks i cipherteksten. Grunden til den ekstra block er at CBC bruger en *Initialization Vector* (IV) som den placerer i ciphertekstens position C_0, således det er den første block. Denne IV er ikke hemmelig, men skal vælges tilfældigt for at sikre kryptering to gange af samme besked, med samme nøgle, stadig resulterer i forskellige ciffertekster. Herefter fungerer kryptering af hver block ud fra følgende formel med nøgle k:

$$C_i = E_k(M_i \oplus C_{i-1})$$

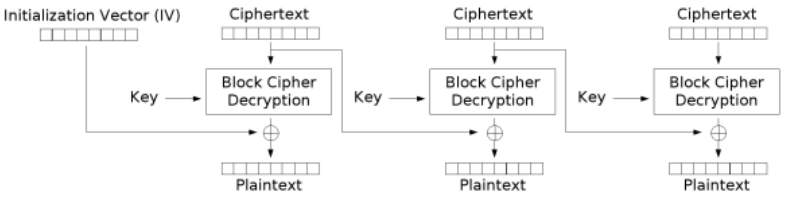
På den måde får vi at hver block afhænger af cifferteksten for den forudgående block, og den første block klartekst krypteres på baggrund af IV'en og krypteringsalgoritmen med nøglen.
Dekryptering virker således nærmest omvendt:

$$M_i = D_k(C_i) \oplus C_{i-1}$$

Hele processen ses nedenfor *(Tak til Wikipedia for billederne ..igen)*:



Cipher Block Chaining (CBC) mode encryption



Cipher Block Chaining (CBC) mode decryption

Selvom CBC mode er den mest brugte, så er den dog ikke perfekt. For at CBC skal virke, skal man sørge for at meddelsens størrelse er et multiplum af cipher block størelsen. Er det ikke det, skal man padde meddelsen på en eller anden måde. Derudover lider CBC også under det problem, at hvis en bit bliver corrupted i en plaintext block, så bliver hele blocken og alle efterfølgende blocks corrupted i ciffereksten. Er det derimod i ciffereksten vi får en corrupted bit, så vil dekryptering gøre hele den block af plaintext corrupt og vil desuden invertere den pågældende bits position i alle efterfølgende blocks.
Sidst men ikke mindst lider CBC under et performance problem, da dens sekventielle natur gør at den ikke kan paralleliseres under enkryptering. Ikke destro mindre er det en af de mere fornuftige valg til Mode of Operation.

1.2.6 RSA

Vi vil nu tage et kig på det nok mest almindelige Public-key system i verden, nemlig RSA. RSA blev skabt i 1970'erne af Ronald Rivest, Adi Shamir og Leonard Adleman, og er i dag stadig et af de mest brugte systemer i praksis. Det er en bereningsmæssig sikker algoritme, der baserer sig på, at for et givent meget stort tal n, så er det beregningsmæssigt meget svært at udregne n's faktorer. Lad os nu kigge på hvordan den præcis opnår denne sikkerhed:

Algoritmen følger 4 punkter:

1. Vælg to store primtal p og q og sæt $n = pq$, således n bliver et semiprimtal (i.e. produktet af 2 primtal).
2. Udregn nu $t = \phi(n) = (p-1)(q-1)$, som kaldes Eulers totient af n.[1]
3. Vælg en positiv integer e, som er større end 1 og mindre end t, og som desuden er indbyrdes primisk[2] med t. Dette kan også skrives som at $e \in \mathbb{Z}$, $1 < e < t$, $gcd(e,t) = 1$. En måde at gøre dette, er at vælge e til at være et primtal. Man vil typisk gerne have e lille dog, da det gør krypteringen nemmere og ingen sikkerhedsmæssig betydning har.
4. Udregn d, således at den opfylder kongruens relationen $ed \equiv 1 (\mathrm{mod}\ t)$. Hvilket vil sige at $ed - 1$ skal være deleligt af t (i.e. ingen remainder).

[1] $\phi(n)$ af et naturligt tal n er defineret til at være antallet af naturlige tal, mindre end eller lig med n, som er indbyrdes primiske med n. For eksempel er $\phi(8) = 4$, da tallene 1, 3, 5 og 7 er indbyrdes primiske med 8. *phi* udtales som "fi".

[2] To tal er indbyrdes primiske hvis deres største fælles faktor er 1. Med andre ord findes der ikke noget tal større end 1 som deler begge tallene. For eksempel er 42 og 25 indbyrdes primiske, mens 42 og 15 ikke er det, da 3 deler begge tallene.

OBS: hvor e kan vælges lille for at gøre kryptering hurtig, så kan d ikke gøre det samme, da den bliver for nem at gætte. Derfor er kryptering med RSA typisk meget hurtigere end dekryptering.

Nu har vi så 3 tal: n, d og e. Disse bruger vi så til at forme den offentlige nøgle (n, e) og den private nøgle (n, d).
Vi kan nu bruge denne offentlige nøgle til at kryptere en besked m:

$$c \equiv m^e (\text{mod } n)$$

Og det er herefter kun ejeren af den private nøgle der kan dekryptere vha. den private nøgle således:

$$m \equiv c^d \equiv m^{ed} (\text{mod } n)$$

Den sidste udregning her virker siden at Euler's totient theorem siger os:

$$m^{ed} \equiv m^{de} \equiv m^{t+1} \equiv m^1 \equiv M (\text{mod } n)$$

Vi har nu en måde at kryptere og dekryptere asymmetrisk. Vi har dog et problem hvis vi vil sende noget småt som en 128bit AES nøgle til Secret-key encryption, simpelthen fordi RSA typisk er minimum 1000bit i outputstørrelse. Vi skal derfor sørge for at have en måde at padde vores data på, således det ikke vil være tydeligt at den relle plaintext kun er 128bit. Dette udelukker automatisk noget simpelt som at padde med rene 0'er og generelt er selve paddingen egentlig også udenfor pensum, så I stedet vil jeg blot nævne, at man sammen med RSA typisk bruger *Optimal Asymmetric Encryption Padding* (OAEP), hvor man tager beskeden der skal sendes og padder den med nogle random bits vha. noget snedigt OAEP stads vi ikke kender til, hvorefter det er denne paddede besked der krypteres vha. den offentlig nøgle (n, e).

Elliptical Curve
Elliptical Curve kryptografi er det nye up-and-coming scheme man forventer vil erstatte alle andre om få år. Flere producenter er allerede begyndt at anbefale elliptical curve algortimer, heriblandt GPG der nu har DSA/ElGamal som default valg for nye nøgler.
Detaljerne for elliptical curve er kraftigt udenfor pensum, men meget kort går det ud på at man bruger finite fields og et punkt beskrevet ud fra den elliptiske kurve. Hvorefter man udnytter det såkaldte *Elliptic curve discrete logarithm problem* (ECDLP), som er et beregningsmæssigt svært problem ligesom primfaktorisering.

Et andet algoritmeeksempel end ElGamal, er Elliptical Curve Diffie Hellman, som virker næsten ligesom en anden algoritme der hedder Diffie Hellman, men blot baserer sig på elliptiske kurver. Denne baserer sig også på discrete logarithm matematik til sin sikkerhed.

1.2.7 Angreb på RSA

Vi vil lige hurtigt kigge på de måder hvorpå vi ville kunne bryde disse Public-key systemer. Den første løsning er den simple, men ufatteligt langsomme.

Exhaustive key search
Både RSA og Elliptical Curve algoritmerne kan blive forsøgt brudt ved blot at brute-force samtlige nøgler og prøve dem en efter en. Det antager dog vi kan genkende hvorvidt resultatet af en dekryptering er meningsfuldt. Er dataene af en sådan art, at dette ikke umiddelbart er til at gøre, så kan brute-forcing stadig være umuligt selv på en uendeligt kraftig maskine.
Brute-forcing er dog sjællendt muligt i praksis - selv hvis man er en hemmelig statsorganisation med tonsvis af supercomputere, så bryder man ikke lige en 4096bit RSA nøgle vha. brute-forcing lige foreløbelig.
Muligheden for exhaustive key search betyder dog, at vi konstant er nød til at bruge større og større nøglestørrelser, simpelthen fordi vi godt nok ikke finder på bedre algoritmer til at løse f.eks. primfaktorisering, men computerne bliver kraftigere og kraftigere. Hvad der var anset som en sikker nøglestørrelse for 10 år siden, er ikke sikker i dag!

Prime factorization
At brute-force en RSA-nøgle er typisk en rigtig dårlig ide. En langt bedre løsning (omend stadig dårlig), er at forsøge at faktorisere n således vi kan få fat i p og q, som vi herefter kan bruge til at udregne d (i.e. det hemmelige komponent af den private nøgle).
Der findes adskillige algoritmer til at faktorisere store tal, som f.eks. *Fermat's Difference of Squares*, *Pollard's* ρ og *Elliptic Curve Factorization Method* (ECM). Tallet n i RSA er så dog særligt svær at faktorisere, da det er et semiprimtal, som er de pt. sværeste typer tal at faktorisere.

Discrete Logarithms
Da det at brute-force f.eks. en ElGamal-nøgle typisk også er en rigtig dårlig ide, så har vi også her muligheden for i stedet at bryde matematikken krypteringen bygger på - i dette tilfælde diskrete logaritmer ($g^x = h$, hvor x er den diskrete logaritme til base g af h i en finite cyclic group G). Eksempler på sådanne algoritmer er *Baby-Step Giant-Step*, *Pollard's* λ og *Index Calculus*

Method.

Side-channel attack
Kort for bare at stjæle den private nøgle fra offeret, så er der en anden måde at bryde krypteringen på, som har vist sig at være særligt effektiv! Et side-channel attack er enhver form for udnyttelse af et cryptosystems implementation fremfor dens teknologi. Det udformer sig typisk i situationer hvor f.eks. man som angriber kan kigge på strømforbruget af en krypteringsenhed og opdage at den i visse situationer bruger lidt mere strøm end ellers. Ud fra disse observationer kan man så finde frem til, hvornår enheden læser et binært 1 og hvornår enheden læser et binært 0 og derigennem ofte direkte aflæse nøglens bits.
Det er også derfor implementationen altid bør sørge for at håndteringen af f.eks. 1 og 0 i nøglen bør tage akkurat lige så lang tid, således man ikke på tid eller strømforbrug kan se en forskel på at processere 0 og 1.

2 Cryptography, Authentication

2.1 Disposition

1. **Sikkerhedsmål**
   - CIA/CAA
   - Def. sikkert system
2. **Kryptologi**
   - Kryptografiske mål (confidentiality/authentication)
   - Secret-key/Public-key Authenticity
3. **MAC**
   - CBC-MAC
   - HMAC
4. **Hashing**
5. **RSA**
6. **Angreb på RSA**
   - Exhaustive key search
   - Prime factorization
   - Side-channel attack

2.2 Details

Her gives detaljer for hvert punkt i dispositionen og muligvis mere til.

2.2.1 Sikkerhedsmål

Når vi snakker om IT-sikkerhed er det typisk i relation til at ville forbedre på et systems sikkerhed i en eller anden forstand. Hvad det præcist er vi vil forbedre på sikkerheden kommer dog meget an på hvad vi gerne vil opnå.

CIA/CAA
Til det formål opstiller man typisk en række sikkerhedsmål der samlet repræsenterer alle aspekter IT-sikkerhed dækker. I løbet af kurset har vi arbejdet med tre opdelinger:

Confidentiality: Confidentiality er at personer kun skal have adgang til den information de er berettiget til og intet mere. Dette gælder uanset om informationen sendes, opbevares eller behandles.

Authenticity: Authenticity er at information er authentisk, altså ikke manipuleret eller på anden måde ændret af en uautoriseret person. Dette gælder også mht. til authenticity af afsender, modtager mv. Således en bruger ikke kan nares til at tro han har modtaget kommunikation fra en han i virkeligheden ikke har.

Availability: Availability er at vi ønsker vores systemer er tilgængelige når de skal bruges, således legale brugere kan få adgang til deres data og de services et system tilbyder.

Normalvis formulere litteraturen dog ovenstående tre mål som Confidentiality, Integrity og Availability, således det staver CIA, hvilket man åbenbart synes er popsmart! Principperne er dog mere eller mindre de samme.

Def. sikkert system
For lige at få det på plads, må vi hellere lige definere hvad et sikkert system er. Det er typisk svært eller umuligt at bevise et system er sikkert, i det at vi så skal til at beskrive systemet ud fra en matematisk model og bevise et teorem om at systmet er sikkert under de og de omstændigheder. Typisk kan det ikke lade sig gøre og hvis det kan, er det typisk fordi vi har antaget så mange forkerte ting om vore angribers muligheder, at vi alligevel ikke ender op med et sikkert system, men blot falsk tryghed.
Så der er i virkeligheden ikke rigtig nogen definition på et "sikkert system", så i stedet nøjes vi med en definition på et "sikret system". Dette gør vi ved at definere en overordnet *Sikkerhedspolitik* på baggrund af en *Trusselsmodel*, og implementerer herefter denne sikkerhedspolitik vha. nogle *Sikkerhedsmekanismer*. På den måde får vi, at et sikret system kan beskrives som:

$$\text{Sikret system} = \text{Sikkerhedspolitik} + \text{Trusselsmodel} + \text{Sikkerhedsmekanismer}$$

2.2.2 Kryptologi

Indenfor dette emne skal vi så kigge på den sikkerhedsmekanisme der er kryptologi. Kryptologi kan deles op i to hoveddiscipliner:

Kryptografi: Videnskaben om at lave koder og ciffere.

Kryptoanalyse: Videnskaben om at bryde koder og ciffere.

Jeg vil komme ind på få kryptoanalytiske ting, men generelt er det kryptografi vi kigger på.

Kryptografiske mål

Generelt kan alt kryptografi deles op i to grupper af de problemer kryptografimekanismen løser. Dette er: Confidentiality og Authenticity.

	Confidentiality	Authenticity
Secret-Key	AES, DES, RC4, IDEA	CBC-MAC, HMAC
Public-Key	RSA, El-Gamal, Ell. curves	RSA, El-Gamal, Ell. Curves
Hash Functions		RIPEMD-160, SHA-1, MD5

Vi fokuserer på Authenticity i dette emne og fokuserer desuden udelukkende på beregneligt sikre systemer, og altså ikke ubetinget sikre løsninger som forudlavede tabeller af tilfældige t-bit MACs til en begrænset mængde beskeder o.lign.. De er alligevel ikke relevante i praksis.

Secret-key/Public-key Authenticity
Kerckhoffs' princip, og til dels Shannon's maxim, fortæller os at vi må antage vore fjender altid kender til vore cryptoalgoritmer i lige så intime detaljer som os selv, så et cryptosystem må aldrig afhænge af algoritmen i sig selv. I stedet skal vi definere nøgler, som så skal distribueres imellem de betroede parter og bruges i kombination med algoritmer.
Der er generelt to måder at bruge nøgler i cryptoalgoritmer: Secret-key og Public-key systemer (til tider også kaldt henholdsvis Symmetrisk og Asymmetrisk).

Det betyder, at hvis vi har et Secret-key cryptosystem til authenticity, så har man to algoritmer: S, for signing og V, for verification. Herefter kan man authenticate sin meddelse m med nøgle k således:

$$c = S_k(m)$$

Og så sende m, c til modtageren, således han har ens meddelse og ens signatur. Dette kaldes en *Message Authentication Code* (MAC).
Den modsatte part der modtager m, c kan så verificere signaturen c vha. V, ligeledes med nøgle k og meddelse m, således:

$$V_k(m, S_k(m)) = accept/reject$$

Således at V giver accept hvis signaturen er korrekt og reject hvis den ikke er. På denne måde forhindrer vi andre i at modificere vore beskeder uden vi opdager det. Sagt på en anden måde garantere ovenstående altså at angriberen ikke kan finde en besked m' som offeret ikke har sendt og samtidig

finde en gyldig MAC for m', medmindre angriberen kender den hemmelige nøgle k.
Altså bruger begge parter samme nøgle (hence, betegnelsen Symmetrisk).

Har vi derimod et Public-key cryptosystem, så sørger man for at hver bruger har 2 nøgler, en privat nøgle (sk) og en offentlige nøgle (pk). Den offentlige nøgle gøres herefter offentlig for enhver (typisk ved at uploade den til en af de mange offentlige nøgleservere), men det er kun en selv der har adgang til den private nøgle, som typisk gemmes lokalt. Vi har så igen to algoritmer: S, for signing og V, for verification. Herefter kan man authenticate/signe sin meddelse m med sin private nøgle sk således:

$$c = S_{sk}(m)$$

Og så sende m, c til modtageren, således han har ens meddelse og ens signatur.
Den modsatte part der modtager m, c kan så verificere signaturen c vha. V, den offentlige nøgle pk på afsenderen og meddelse m, således:

$$V_{pk}(m, S_{sk}(m)) = accept/reject$$

Igen således at V giver accept hvis signaturen er korrekt og reject hvis den ikke er. Men udover at angriberen ikke kan finde en besked m' som offeret ikke har sendt og samtidig finde en gyldig signatur for m' uden nøglen, så skal det selvfølgelig også gælde selvom han/hun har adgang til den offentlige nøgle pk (hvilket han/hun har).

Fælles for begge typer cryptosystemer er at en angriber ikke nødvendigvis behøver bryde authenticationalgoritmen, men i stedet blot kan lave såkaldt *Exhaustive key search*, hvor angriberen obsnapper en række beskeder og dertilhørende signaturer. Når angriberen har opsnappet tilstrækkelig mange, så vil nøglen være unikt bestemt ud fra disse signaturer, så nu kan angriberen blot gætte på nøglen og prøve hver gang at lave en signatur med nøglen på hver meddelse og se om disse passer med de opsnappede. Før eller siden vil angriberen således nå frem til den rigtige nøgle.
Dette sørger man dog for er usandsynligt i praksis, ved at gøre nøglerne så store, at exhaustive key search tager meget meget lang tid (typisk adskillige år). For Public-key løsninger findes der dog algoritmer der kan udregne den private nøgle sk ud fra den offentlige nøgle pk langt hurtigere end den tid det tager at lave exhaustive key search, og derfor er nøglestørrelserne for Public-key kryptering også langt større end for Secret-key (Typisk siger man minimum 1024bit. GPG anbefaler pt. 2048bit).
Men modsat Secret-key systemer, så behøver man ikke, i Public-key, udveksle nøgler før man kommunikerer sikkert og man risikerer derfor ikke at en

angriber opsnapper nøglen i den initielle udveksling.
Man skal dog også være opmærksom på, at store nøglestørrelser ikke er nok i sig selv. De resulterende signaturer skal også være tilstrækkelige store, til at en korrekt signatur ikke bare kan gættes. For MAC's er det f.eks. minimum 64bit.

Et par sidste ting man skal være opmærksom på er, at hvis en bruger A sender en besked til B, hvor A har udregnet en Secret-key MAC på beskeden, så er B overbevist om A har sendt beskeden, men B kan ikke overbevise f.eks. C om det, da nøglen mellem A og B ikke er den samme som den mellem A og C. Derfor kunne MAC'en lige så godt være blevet udregnet af B, som forsøger at forfalske A! Samme problem har vi ikke med Public-key, da det netop her kun er A der kunne have beregnet signaturen $S_{sk}(m)$ og ingen andre. Under antagelsen af at den offentlige nøgle kan garanteres korrekt og sikker, så får man altså en global signatur for A, hvilket også er hvorfor mange Public-key systemer bruges som digitale signaturer.
Digitale signaturer bruger man bl.a. for at garantere non-repudiation - hvilket basalt set betyder at man kan hænges op på hvad man gør. Man kan f.eks. ikke gå ind og købe noget med en digital signatur og bagefter påstå man ikke har købt det, da virksomheden blot kan fremvise du har authentikeret dig selv unikt i forhold til shoppen.
Det bør desuden nævnes at en signatur egentlig ikke siger at f.eks. A har sendt en given meddelse, men blot at A engang lavede pågældende meddelse. Angriberen kan nemlig godt opsnappe beskeden og så lave et replay angreb hvor han sender beskeden flere gange, f.eks. for at overføre penge flere gange som A. Derfor bør man inkorporere en eller anden form for sekvensnummer eller timestamp i medddelsen, således at denne indgår som en del af signaturen og tjekkes af modtageren.

2.2.3 MAC

Hvis vi så tager et kig på nogle af disse Message Authentication Code (MAC) typer vi har, startende med den der følger direkte af den tilsvarende krypteringsteknologi, nemlig CBC-MAC.

CBC-MAC

Hvis man stoler på ens cryptoalgoritme til kryptering (f.eks. AES) og bruger Mode of Operation typen CBC, så kan man opbygge authentication ud fra det. For lige at forklare processen bedre, så tager jeg lige først og forklarer CBC dybdegående.
Control Block Chaining (CBC) virker ved at vi deler klarteksten op i den pågældende algoritmes blockstørrelser (f.eks. 128bit for AES), således man

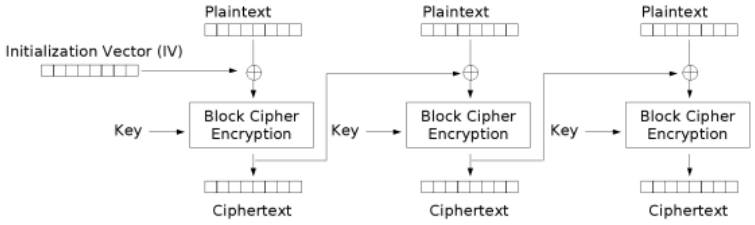
får separate meddelelser $M_1 \dots M_t$ hvor t er antallet af blocks i klarteksten og $t + 1$ bliver antallet af blocks i cipherteksten. Grunden til den ekstra block er at CBC bruger en *Initialization Vector* (IV) som den placerer i ciphertekstens position C_0, således det er den første block. Denne IV er ikke hemmelig, men skal vælges tilfældigt for at sikre kryptering to gange af samme besked, med samme nøgle, stadig resulterer i forskellige cifferteksters. Herefter fungerer kryptering af hver block ud fra følgende formel med nøgle k:

$$C_i = E_k(M_i \oplus C_{i-1})$$

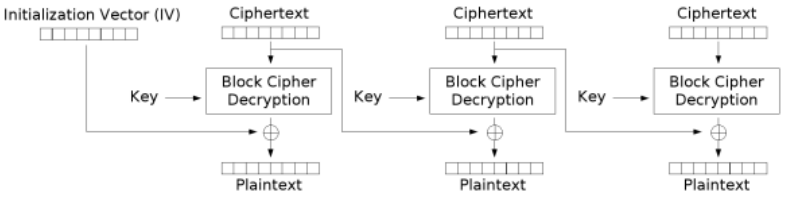
På den måde får vi at hver block afhænger af cifferteksten for den forudgående block, og den første block klartekst krypteres på baggrund af IV'en og krypteringsalgoritmen med nøglen.
Dekryptering virker således nærmest omvendt:

$$M_i = D_k(C_i) \oplus C_{i-1}$$

Hele processen ses nedenfor *(Tak til Wikipedia for billederne ..igen)*:



Cipher Block Chaining (CBC) mode encryption



Cipher Block Chaining (CBC) mode decryption

Skal man derimod bruge CBC til authentication i form af CBC-MAC, så tager man samme proces som ovenfor, men sætter i stedet *Initilization Vector* (IV) til rene 0'er og definerer så MAC'en til at være den sidste block af cifferteksten. Denne ciffertekst kan således blot blive bekræftet på den

modsatte ende ved at kryptere beskeden igen med IV'en sat til rene 0'er og se om den sidste block svarer til den fremsendte MAC.

HMAC
En alternativ løsning er den såkaldte HMAC, som bruges i rigtig mange internet applikationer. HMAC går ud på vi tager en hashing funktion som MD5, SHA e.lign. og bruger denne til at lave et hash af meddelsen og nøglen sammen, og bruger disses output som MAC'en. Udregningen er som følger:

$$HMAC_k(m) = H\left[(k \oplus opad)H((k \oplus ipad)m)\right]$$

Hvor nøglen k er blevet padded til at passe til hashingfunktionen H's bitstørrelse, og hvor *opad* (outer padding) og *ipad* (inner padding) er to 1-blocks hex padding konstanter med værdierne:

$$opad = 0x5c5c5c5c\ldots5c$$
$$ipad = 0x36363636\ldots36$$

HMAC's sikkerhed afhænger i høj grad af sikkerheden af den underliggende hashingfunktion. HMAC-MD5 er derved usikker, hvorimod HMAC-SHA256 ikke er det. Nu hvor vi har fået lidt snak om hashfunktioner i gang, så lad os kigge lidt nærmere på dem.

2.2.4 Hashing

En kryptografisk hash funktion er en funktion der har følgende egenskaber:

- Den skal kunne tage en meddelse af enhver længde som input.
- Den skal producere et output af en fastlagt længde.
- Den skal være hurtig
- Det skal være et svært beregningsmæssigt problem at finde en kollision (i.e. at to forskellige værdier x og y leder til samme hashværdi).

Hvor svært det er at finde en kollision kommer an på hashing teknologien, men det kan vises at hvis vi har en hashing funktion H med k bits output og vi forsøger at finde en kollision ved at bruge H på tilfældige beskeder, så vil vi få en kollision efter $2^{k/2}$ gentagelser af H. Ifølge teksten betyder det at hash funktioner skal have output på minimum 160bit i dag.

Men hvis vi så har en hash funktion der opfylder kravene ovenfor, så kan vi bruge denne til at tage en hash af vores meddelse og så lave vores MAC på baggrund af denne hash, fremfor på baggrund af meddelsen. På den måde bruger vi mindre plads pr. meddelse og MAC'en kan beregnes hurtigere. Det baserer sig dog på den ide, at vi ikke får en kollision, da hvis vi gjorde ville vi kunne få to MAC's der var identiske til trods for meddelserne ikke var det.

Som nogle eksempler på hash funktioner kan nævnes MD5, SHA-1, SHA-256, SHA-512, RIPEMD-160 m.fl. Af disse bruges MD5 og SHA-varianterne mest i praksis, men MD5 er blevet bevist usikker og SHA-1 er fremvist som usikker ved at vise der kunne findes kollisioner hurtigere end hvad der ville kunne ved et normalt bruteforce angreb. Dette sidste angreb er dog aldrig blevet udført i praksis. Ikke destro mindre bør man sandsynligvis bruge SHA-256 eller SHA-512, som de fleste kommercielle systemer sandsynligvis også vil skifte til inden længe.

2.2.5 RSA

Da vi har snakket om Public-key authentication metoder, men ikke givet et eksempel, vil vi nu tage et kig på det nok mest almindelige Public-key system i verden, nemlig RSA. RSA blev skabt i 1970'erne af Ronald Rivest, Adi Shamir og Leonard Adleman, og er i dag stadig et af de mest brugte systemer i praksis. Det er en bereningsmæssig sikker algoritme, der baserer sig på, at for et givent meget stort tal n, så er det beregningsmæssigt meget svært at udregne n's faktorer. Lad os nu kigge på hvordan den præcis opnår denne sikkerhed:

Algoritmen følger 4 punkter:

1. Vælg to store primtal p og q og sæt $n = pq$, således n bliver et semi-primtal (i.e. produktet af 2 primtal).
2. Udregn nu $t = \phi(n) = (p-1)(q-1)$, som kaldes Eulers totient af n.[3]
3. Vælg en positiv integer e, som er større end 1 og mindre end t, og som desuden er indbyrdes primisk[4] med t. Dette kan også skrives som at $e \in \mathbb{Z}$, $1 < e < t$, $gcd(e,t) = 1$. En måde at gøre dette, er at vælge e

[3] $\phi(n)$ af et naturligt tal n er defineret til at være antallet af naturlige tal, mindre end eller lig med n, som er indbyrdes primiske med n. For eksempel er $\phi(8) = 4$, da tallene 1, 3, 5 og 7 er indbyrdes primiske med 8. *phi* udtales som "fi".

[4] To tal er indbyrdes primiske hvis deres største fælles faktor er 1. Med andre ord findes der ikke noget tal større end 1 som deler begge tallene. For eksempel er 42 og 25 indbyrdes primiske, mens 42 og 15 ikke er det, da 3 deler begge tallene.

til at være et primtal. Man vil typisk gerne have e lille dog, da det gør krypteringen nemmere og ingen sikkerhedsmæssig betydning har.

4. Udregn d, således at den opfylder kongruens relationen $ed \equiv 1 (\text{mod } t)$. Hvilket vil sige at $ed - 1$ skal være deleligt af t (i.e. ingen remainder).

OBS: hvor e kan vælges lille for at gøre verifikation hurtig, så kan d ikke gøre det samme, da den bliver for nem at gætte. Derfor er verifikation med RSA meget hurtigere end signering *(og kryptering er meget hurtigere end dekryptering, når det er det vi vil bruge nøglerne til).*

Nu har vi så 3 tal: n, d og e. Disse bruger vi så til at forme den offentlige nøgle (n, e) og den private nøgle (n, d).
Vi kan nu bruge denne private nøgle til at signere en besked m:

$$s = m^d (\text{mod } n)$$

Og så sende m, s til modtageren, således han/hun har ens meddelse og ens signatur.
Den modsatte part der modtager m, s kan så verificere signaturen s vha. den offentlige nøgle pk på afsenderen og meddelsen m, således:

$$m = s^e (\text{mod } n) = (m^d \text{mod } n)^e \text{mod } n$$

Således at hvis verifikationen leder til m korrekt, så er signaturen gyldig.

2.2.6 Angreb på RSA

Vi vil lige hurtigt kigge på de måder hvorpå vi ville kunne bryde RSA. Den første løsning er den simple, men ufatteligt langsomme.

Exhaustive key search
RSA kan blive forsøgt brudt ved blot at brute-force samtlige nøgler og prøve dem en efter en. Det antager dog vi kan genkende hvorvidt resultatet af en dekryptering er meningsfuldt. Er dataene af en sådan art, at dette ikke umiddelbart er til at gøre, så kan brute-forcing stadig være umuligt selv på en uendeligt kraftig maskine.
Brute-forcing er dog sjællendt muligt i praksis - selv hvis man er en hemmelig statsorganisation med tonsvis af supercomputere, så bryder man ikke lige en 4096bit RSA nøgle vha. brute-forcing lige foreløbelig.
Muligheden for exhaustive key search betyder dog, at vi konstant er nød til at bruge større og større nøglestørrelser, simpelthen fordi vi godt nok

ikke finder på bedre algoritmer til at løse f.eks. primfaktorisering, men computerne bliver kraftigere og kraftigere. Hvad der var anset som en sikker nøglestørrelse for 10 år siden, er ikke sikker i dag!

Prime factorization

At brute-force en RSA-nøgle er typisk en rigtig dårlig ide. En langt bedre løsning (omend stadig dårlig), er at forsøge at faktorisere n således vi kan få fat i p og q, som vi herefter kan bruge til at udregne d (i.e. det hemmelige komponent af den private nøgle).
Der findes adskillige algoritmer til at faktorisere store tal, som f.eks. *Fermat's Difference of Squares*, *Pollard's* ρ og *Elliptic Curve Factorization Method* (ECM). Tallet n i RSA er så dog særligt svær at faktorisere, da det er et semiprimtal, som er de pt. sværeste typer tal at faktorisere.

Side-channel attack

Kort for bare at stjæle den private nøgle fra offeret, så er der en anden måde at bryde krypteringen på, som har vist sig at være særligt effektiv! Et side-channel attack er enhver form for udnyttelse af et cryptosystems implementation fremfor dens teknologi. Det udformer sig typisk i situationer hvor f.eks. man som angriber kan kigge på strømforbruget af en krypteringsenhed og opdage at den i visse situationer bruger lidt mere strøm end ellers. Ud fra disse observationer kan man så finde frem til, hvornår enheden læser et binært 1 og hvornår enheden læser et binært 0 og derigennem ofte direkte aflæse nøglens bits.
Det er også derfor implementationen altid bør sørge for at håndteringen af f.eks. 1 og 0 i nøglen bør tage akkurat lige så lang tid, således man ikke på tid eller strømforbrug kan se en forskel på at processere 0 og 1.

3 Key Management and Infrastructures

3.1 Disposition

1. **Sikkerhedsmål**
   - CIA/CAA
   - Def. sikkert system
2. **Key Management**
3. **KDC**
4. **PKI (CA)**
   - Certificate Chains
   - X.509
   - Tyveri af nøgler
5. **Passwords**
   - Passphrases
6. Angreb på passwords
   - Password Crackers
   - Social Engineering
   - Phishing/Smishing
   - Indbrud
7. **Biometrics**

3.2 Details

Her gives detaljer for hvert punkt i dispositionen og muligvis mere til.

3.2.1 Sikkerhedsmål

Når vi snakker om IT-sikkerhed er det typisk i relation til at ville forbedre på et systems sikkerhed i en eller anden forstand. Hvad det præcist er vi vil forbedre på sikkerheden kommer dog meget an på hvad vi gerne vil opnå.

CIA/CAA
Til det formål opstiller man typisk en række sikkerhedsmål der samlet repræsenterer alle aspekter IT-sikkerhed dækker. I løbet af kurset har vi arbejdet med tre opdelinger:

Confidentiality: Confidentiality er at personer kun skal have adgang til den information de er berettiget til og intet mere. Dette gælder uanset om informationen sendes, opbevares eller behandles.

Authenticity: Authenticity er at information er authentisk, altså ikke manipuleret eller på anden måde ændret af en uautoriseret person. Dette gælder også mht. til authenticity af afsender, modtager mv. Således en bruger ikke kan nares til at tro han har modtaget kommunikation fra en han i virkeligheden ikke har.

Availability: Availability er at vi ønsker vores systemer er tilgængelige når de skal bruges, således legale brugere kan få adgang til deres data og de services et system tilbyder.

Normalvis formulere litteraturen dog ovenstående tre mål som Confidentiality, Integrity og Availability, således det staver CIA, hvilket man åbenbart synes er popsmart! Principperne er dog mere eller mindre de samme.

Def. sikkert system
For lige at få det på plads, må vi hellere lige definere hvad et sikkert system er. Det er typisk svært eller umuligt at bevise et system er sikkert, i det at vi så skal til at beskrive systemet ud fra en matematisk model og bevise et teorem om at systmet er sikkert under de og de omstændigheder. Typisk kan det ikke lade sig gøre og hvis det kan, er det typisk fordi vi har antaget så mange forkerte ting om vore angribers muligheder, at vi alligevel ikke ender op med et sikkert system, men blot falsk tryghed.
Så der er i virkeligheden ikke rigtig nogen definition på et "sikkert system", så i stedet nøjes vi med en definition på et "sikret system". Dette gør vi ved at definere en overordnet *Sikkerhedspolitik* på baggrund af en *Trusselsmodel*, og implementerer herefter denne sikkerhedspolitik vha. nogle *Sikkerhedsmekanismer*. På den måde får vi, at et sikret system kan beskrives som:

$$\text{Sikret system} = \text{Sikkerhedspolitik} + \text{Trusselsmodel} + \text{Sikkerhedsmekanismer}$$

3.2.2 Key Management

I mange af de andre emner i dSik antager vi, at vi altid har adgang til de nøgler vi skal bruge og at ingen andre har samme, såfremt nøglerne ligger

lokalt hos os. Desværre er realiteten ikke så pæn og vi er nød til at bekymre os om hvordan vores nøgler transporteres, holdes sikre og holdes opdaterede. Desuden er vi nød til at anerkende to grundlæggende principper:

> **Enhver hemmelig systemparameter løber en større risiko for at blive fundet jo længere tid den holdes konstant og jo mere man bruger den.**

Og det andet princip:

> **Ethvert sikkert system der bruger kryptografi må have en eller flere nøgler der er beskyttet udelukkende vha. fysiske, ikke-kryptografiske metoder.**

Det første princip ligger kraftigt op af Kerckhoffs' princip, og til dels Shannon's maxim, der fortæller os at vi må antage vore fjender altid kender til vore krypteringsalgoritmer i lige så intime detaljer som os selv, så et cryptosystem må aldrig afhænge af algoritmen i sig selv. Men udover blot dette, så lægger princippet også op til, at vi må antage fjenden på et tidspunkt finder vore nøgler hvis vi bare bruger dem nok - derfor er vi nød til at ændre nøgler med jævne mellemrum.
Det andet princip er i en anden boldgade, men ikke destro mindre uhyre vigtigt. Vi kan forsøge at beskytte vore nøgler mod tyveri fra alle sider ved at kryptere en nøgle med en anden nøgle, hvor denne nøgle er beskyttet af en tredje nøgle, der er beskyttet af en fjedre nøgle osv. osv. Før eller siden er vi dog nød til at nå enden af denne kæde og have minimum en nøgle der ikke er beskyttet af andre nøgler - vi er altså nød til i sidste ende at have en sikkerhedsmekanisme der ikke er kryptografi.

Hvis vi så starter med problematikken omkring nøgler der bruges flere gange, så er den simplistiske Secret-key løsning for de to parter A og B følgende:

1. A og B beslutter sig for en nøgle K_{AB}, som kun skal bruges til nøgletransport.
2. Hvis A vil sende en meddelse til B, så genererer A først en tilfældig session nøgle k.
3. A sender herefter $E_{K_{AB}}(k), E_k(M)$ til B.
4. B bruger herefter nøglen K_{AB} til at dekryptere session nøglen k og bruger derefter denne til at dekryptere meddelelsen.

I dette simple eksempel bruges session nøglen k kun 1 gang, men i virkeligheden ville den sandsynligvis blive brugt en række gange før den til sidst kaseres. Typisk når en given session afsluttes!
Løsningen skalerer dog elendigt, da man i flerbruger-netværk ville have en eksplosion af nøgler, siden alle ville skulle have en nøgle til nøgletransport for alle andre brugere i systemet. Derfor skal vi bruge mere intelligente løsninger!
I løbet af kurset har vi set på to sådanne løsninger.

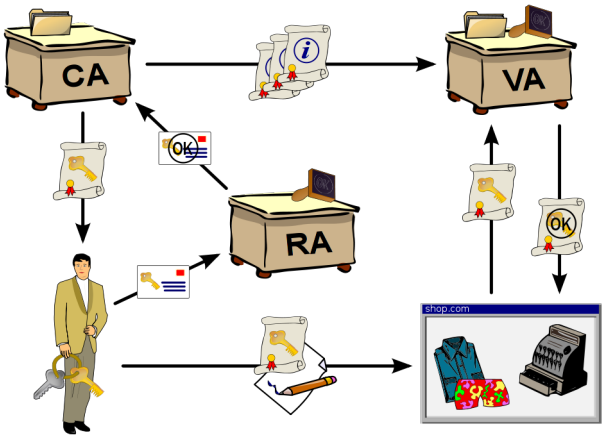
3.2.3 KDC

Den første mulige løsning er en centraliseret løsning i form af et *Key Distribution Center* (KDC). Grundlæggende fungerer løsningen ved at brugere deler en nøgle med KDC'en, således at bruger A har en nøgle K_A med KDC'en. Når så A ønsker at tale med B, så bliver KDC'en bedt om at generere en nøgle K for den pågældende session og sender $E_{K_A}(K)$ til A og $E_{K_B}(K)$ til B. På den måde kan de begge få fat i nøglen og kan begynde at kommunikere sikkert sammen. Denne løsning har også den fordel at A kan sætte sin lid til, at KDC'en sørger for at A ikke bliver narret til at tro C i virkeligheden er B, da det netop er KDC'en der styrer distribution af nøgler til hver part og ikke parterne selv.
KDC løsninger er dog ikke så ofte forekommende mere, da de lider under nogle klare problemer. Først og fremmest er det en løsning der skalerer dårligt og er et single-point-of-failure i et distribueret system. Men noget endnu vigtigere er, at brugeren skal stole fuldt ud på, at KDC'en er troværdig og ikke udnytter sin magt, da KDC'en har adgang til alle nøgler.
For at kunne understøtte distribution over et stort netværk, som f.eks. Internettet, så er det altså nødvendigt med en mere decentraliseret løsning.

3.2.4 PKI (CA)

Den anden løsning teksten taler om er *Certification Authorities* (CA) hvor den delte nøgle K_{ab} mellem A og B erstattes af offentlige og private nøglepar $[(sk_A, pk_A), (sk_B, pk_B)]$, hvor kun ejeren af nøglen kender den private nøgle sk, men alle kender den offentlige nøgle pk *(typisk ved at have dem på offentlige keyservere)*. Dette betyder at vi, modsat Secret-key løsningen, ikke behøver dele en hemmelig nøgle mellem parterne på forhånd og blot kan bruge den offentlige nøgle - vi skal dog være sikker på vi bruger den rigtige offentlige nøgle.
Det er netop til dette vi skal bruge en Certification Authority (CA). Blot lige for at sætte det ud i et større perspektiv, så er en CA blot en del af en større struktur kaldet en *Public Key Infrastructure* (PKI), som består af al hardware, software, procedurer og politiker indblandet i at skabe, styre, op-

bevare, distribuere og tilbagekalde digitale certifikater. Der er således andre komponenter i et PKI, men vi vil kun fokusere på CA'erne.



Den måde en CA fungerer på er, at den har sit eget nøglepar (sk_{CA}, pk_{CA}), hvor vi så sørger for alle brugere på en eller anden måde har en korrekt kopi af pk_{CA}. En bruger A kontakter herefter CA'en, identificerer sig selv unikt *(typisk vha. en Registration Authority (RA), der også er en del af PKI, der tjekker en brugers in-real-life identitet.)* og sender sin offentlige nøgle pk_A til CA'en. Hvis CA'en acceptere bruger A's rigtige identitet, så skaber den et såkaldt certifikat, som bl.a. indeholder en streng ID_A der identificerer A unikt, A's offentlige nøgle pk_A og CA'ens signatur af ID'et og nøglen således:

$$S_{sk_{CA}}(ID_A, pk_A)$$

På den måde kan enhver bruger med CA'ens offentlige nøgle pk_{CA} nu tjekke at de har fat i en gyldig offentlig nøgle for A og kan derfor begynde at kommunikere sikkert med A.
Godt nok skal vi her have tillid til at CA'en har tjekket en brugers identitet ordentligt, men det er en anden form for tillid, for modsat et KDC har CA'en ikke adgang til ens private nøgler, så den kan ikke signere dokumenter eller dekryptere for en.
Hvis en bruger opdager sin private nøgle er blevet stjålet, eller hvis den på anden måde er gået tabt, så kan han kontakte sin CA og få tilbagekaldt sit certifikat (kaldes *certificate revocation*). CA'en tilføjer herefter certifikatets ID til en *Certificate Revocation List* (CRL) som den offentliggører.
Udover dette kan ens certifikat også udløbe, hvilket den, for de fleste CA'er, også gør efter 1 år. Herefter skal man registrere sig pany.

Certificate Chains
Det er så alt sammen fint. Nu har vi en måde at tjekke om en public key er valid eller ej. Men hvad nu hvis to brugere har fået deres certifikater fra to forskellige CA'er? Hvordan kontakter de så den pågældende brugers CA, når de ikke har CA'ens offentlige nøgle? Svaret er konceptet om Certificate Chains.
Hvis vi har to brugere A og B med henholdsvis CA'er CA_1 og CA_2, så kan vi have en situation hvor A modtager $Cert_{CA_2}(B, pk_B)$, altså B's certifikat. Men da A ikke har CA_2's offentlige nøgle, så skal den skaffe denne på en sikker måde et eller andet sted fra. Det vi så kan gøre er, at CA'er sørger for at validere hinandens offentlige nøgler, således A kan anmode CA_1 om CA_2's certifikat og få $Cert_{CA_1}(CA_2, pk_{CA_2})$ tilbage. Nu ved A således hvad CA_2's offentlige nøgle er, og den kan gå videre og verificere $Cert_{CA_2}(B, pk_B)$.
Forklaret lidt mere generelt har vi en kæde af CA'ers certifikater fra A's egen CA, her kaldet CA_n, til certifikatet for B:

$$Cert_{CA_1}(B, pk_B), Cert_{CA_2}(CA_1, pk_{CA_1}), \ldots, Cert_{CA_n}(CA_{n-1}, pk_{CA_{n-1}})$$

Således at hvis A kan danne en sådan kæde hele vejen fra sin egen CA (CA_n) til B's CA (CA_1), så kan A også verificere B.

Der er dog nogle problemer med løsningen. Eksempelvis kan A kun stole på B's certifikat, såfremt han/hun stoler på alle CA'er i kæden. Desuden vil det stadig være nødvendigt at kende minimum en CA's offentlige nøgle helt fra starten, så denne skal kunne verificeres på anden vis. Sidstnævnte problem er dog ikke så stort et problem, da de fleste softwareløsninger med understøttelse for at tjekke CA'er har en pre-indbygget liste af kendte CA'ers selv-signerede rodcertifikater. Så til den grad at man stoler på sin software *(i.e. fordi man har fået det fra en betroet kilde)*, så kan man også stole på CA'ernes certifikater er gyldige.

X.509
Jeg har på yndigste vis undgået at nævne særlig meget om hvad certifikater egentlig indeholder og begrundelsen for dette er, at ingen kan blive helt enige om hvad de skal indeholde! Normalvis har man en standard kaldet X.509, som definerer en række minimumskrav til et certifikat, men pga. manglende fleksibilitet i de tidlige versioner af X.509 har man lavet flere tilføjelser, men ikke alle har valgt at bruge disse - resultatet er, at man nu har flere applikationer der påstår at understøtte X.509, men alligevel ikke nødvendigvis kan tale sammen.
Typisk vil et certifikat dog som minimum indeholde information som:

- X.509 versionnummeret
- Et serienummer for certifikatet
- Navnet på certifikatejeren
- Adresse på certifikatejeren
- Navnet på CA'en der har signeret certifikatet
- Måden hvorpå CA'en tjekkede ejerens rigtige identitet
- Dato for udstedelse
- Hvornår certifikatet udløber
- Ejerens rettigheder og privilegier
- Cryptoalgoritmen der skal bruges for at tjekke certifikatet
- Cryptoalgoritmen certifikatejeren bruger
- Den offentlige nøgle for certifikatejeren
- ..og CA'ens signatur.

Tyveri af nøgler
Det er en ting at vi nu har en sikker måde at finde den korrekte offentlige nøgle for en bruger, men det gavner os ikke meget hvis vore private nøgler alligevel ikke er sikre. Problemet er nemlig det jeg omtalte i starten, nemlig at vi ikke kan beskytte alt vha. kryptografi - før eller siden skal andre sikringer til. Dette problem får vi f.eks. når vi har vores private nøgle.
Hvis man er en virksomhed med mange penge og høje sikkerhedskrav, så vil man måske investere i noget sikkert hardware der kan beskytte nøgler inde i den og styre tilgangen til dem, men for de flestes vedkommende vil dette dog være en alt for dyr løsning - i stedet gemmer disse den private nøgle lokalt på disken.
Men hvordan beskytter man så sådan en nøgle fra simpelt tyveri *(f.eks. tyveri af disken, hackerindbrud, malware mm.)*? Jo, det gør man ved at styre brugen af nøglen vha. et password og kryptere nøglen med det pågældende password som nøgle. Desværre er passwords dog karaktersekvenser og ikke strenge af bits med en bestemt længde, så man bruger i stedet en hash function med minimum 128bit output og hasher passwordet med dette, hvorefter man krypterer:

$$E_{h(pw)}(sk)$$

Nu har vi krypteret nøglen, men den er dog ikke overdrevet svær at bryde ind i, da man kan brute-force sig vej til passwordet langt hurtigere end man

normalt ville kunne en 128bit nøgle. Dette skyldes at antallet af mulige passwords er langt mindre end 2^{128}, så angriberen kan blot prøve alle af disse hashet som nøgle.
For at bekæmpe dette problem bruger man det man kalder *Moderately Hard Functions*, hvilket går ud på man bevidst nedsætter hastigheden af hashfunktionen ved at ændre den således den udfører samme hashing adskillige gange *(flere tusinde gange typisk)* og derved gør beregningen meget langsommere.

$$h(pw) = \text{SHA-256}(\text{SHA-256}(\text{SHA-256}(\ldots \text{SHA-256}(pw)\ldots))$$

For den legale bruger, der kun skal udregne $h(pw)$ en gang, er dette ikke et problem. Men for angriberen der skal udregne samme hash $h(pw)$ flere tusinde gange er det et stort problem.

3.2.5 Passwords

Som jeg forklarede tidligere kan vi ikke regne med kryptografi til at løse alle vore problemer. I sidste ende er vi nød til at have andre sikkerhedsmekanismer også. Jeg vil dog ikke snakke om alle mekanismer der blev nævnt i løbet af kurset, men blot om to af dem. Og den først og mest almindelige sikkerhedsmekanisme? Passwords!

Passwords er en essentiel del af ethvert IT-system og de fleste mennesker bruger i dag passwords på adskillige forskellige lokationer - hvad end de er i form af passwords til ens computer, til Daimi, til Facebook, til ens Dankort *(Pinkode)*, Mikrobølgeovn *(at få den til at lave min minipizza korrekt kan ofte føles som at brute-force et password i hånden.. all those buttons!)* eller noget helt sjette.
Selvom mennesker har haft passwords i rigtig mange år, så er det dog stadig de færreste der bruger dem forsvarligt. Når man evaluerer et passwords sikkerhed, så bør man stille sig følgende fire spørgsmål:

- Hvordan blev passwordet valgt?
- Hvordan bliver passwordet transmitteret til systemet der skal verificere det?
- Hvordan opbevarer du passwordet?
- Hvordan opbevarer systemet passwordet?

Hvis svarene til ovenstående spørgsmål er: “Min fødselsdag”, “Papirflyver”, “Opslagstavle” og “Krypteret disk med 24 timers overvågning”, så bør du

nok overveje om du ikke kunne gøre det lidt bedre.[5]

Generelt skal man sørge for ens passwords bliver valgt så tilfældigt som muligt, så lange som muligt og opbevares så skjult som muligt. I en ideel verden er "så skjult som muligt" inde i ens eget hoved, som noterne også foreslår. Jeg vil dog argumentere for, at det er counter-productive at opbevare passwords på denne måde, da det automatisk begrænser dig til passwords du kan huske, fremfor passwords du kan lære *(i.e. jeg kan ikke sige dig hvad visse af mine passwords er, men jeg kan taste dem på et dansk QWERTY tastatur)*. Desuden kan du sagtens opbevare en papirlap med dit password indtil du helt har det i fingrene, sålænge du har papirlappen på dig *(f.eks. i din pung)*. Hvis en angriber virkelig var modig nok til at stjæle din pung, så kunne han jo lige så godt slå dig i hovedet med et bat og tvinge dig til at skrive dit password ned.. og så ville du nok virkelig begynde at fortryde du ikke bare lærte det i fingrene som han lige brækkede.
Udover selve valget af password, er der også et spørgsmål om transmission af passwordet. Hvis passwordet sendes over en ukrypteret linje på et åbent netværk, så kan enhver person med en netværkssniffer samle dit password op og bruge det som var de dig *(det er den måde POP3, IMAP, FTP og Telnet virker på - thumbs up til dem!)*. Derfor skal der først skabes en krypteret forbindelse, hvorefter du kan identificere dig vha. dit password.

Passphrases
En måde der ofte foreslås til at vælge passwords er at gøre det vha. passphrases. Passphrases er sætninger som "Min papirflyver blev hacket af 7 onde dværgnisser fra Sønderjylland" hvilket ville oversætte til passwordet: Mpbha7odfS.
Nogle studier har vist at disse passwords er lige så svære at gætte som tilfældigt valgte passwords, men nemmere at huske.

3.2.6 Angreb på passwords

Lad os kigge på hvordan man så som angriber ville udnytte de passwords folk vælger. Der er egentlig mange måder man kan gøre det på, men jeg vil blot lige kigge på 4 muligheder.

Password Crackers
Den første løsning vi kigger på er også den mest simple - dictionary attacks og brute-force cracking! Der findes mange applikationer der hjælper

[5] Okay, in all fairness, man har typisk ikke meget at sige I hvordan ens password bliver transmitteret... men stadig.

angriberen med at gætte sig vej til passwords, enten ved at bryde løs på en online server eller webapplikation, eller ved at bryde løs på en lokal gemt passwordfil *(som f.eks. kunne være stjålet fra en server et sted)*.
For rigtig mange mennesker vil deres passwords blive fundet uhyre stærkt, typisk på få sekunder. Dette er især mennesker der vælger at bruge deres hunds navn, meningsfulde ord, datoer o.lign. For dem der har været lidt mere opfindsomme og lavet tilfældigt udseende passwords, så vil der gå lidt længere tid før deres passwords bliver brudt - men medmindre deres passwords er meget lange (16+ tegn) eller indeholder meget unormale tegn *(som en password cracker ikke normalvis tjekker)*, så skal man forvente passwordet bliver brudt før eller siden.
En måde at forhindre angriberen i at gøre dette "online", er ved at sørge for han ikke kan verificere om hans valg af password er korrekt - f.eks. ved at blokere ip'en efter 3 mislykkedes loginforsøg eller ved kun at godkende 3 mislykkedes loginforsøg og derefter låse kontoen.[6]

Social Engineering
En anden løsning, som især den hypede Kevin Mitnick er glad for, er social engineering. Social engineering er en ikke teknisk løsning hvor man udgiver sig for at være en anden med højere privilegier end en selv og derigennem snyder sig vej til information. Det kunne f.eks. være hvor man udgav sig for at være tekniker fra IBM der skulle ind og løse en fejl i en hardwareenhed i virksomhedens IBM-serversetup, og derigennem fik nærmest ubetinget adgang til systemer normalt kun administratorer ville kunne få lov til at behandle.
Et andet eksempel, som noterne også nævner, er hvor man sender en e-mail hvor man udgiver sig for at være f.eks. administrationen på Daimi der anmoder om passwords til folks konti pga. nogle datatab, som administrationen forsøger at rette op på. Et sådant eksempel er dog mere teknisk i karakter og ville derfor nærmere blive betegnet som phishing jvf. bl.a. Wikipedia.
En ægte social engineer gør det over telefonen eller face-to-face ;-).

Phishing/Smishing
Phising/Smishing er to tekniske løsninger til henholdsvis e-mail og sms kommunikation, hvor man vha. falske meddelser forsøger at lokke ofre ind på sider der ligner f.eks. en banks eller hvor man forsøger at lokke information ud af dem i form af passwords, kreditkortnumre mm.
Smishing er ikke set så mange gange endnu, men kunne ske at eskplodere på et tidspunkt grundet de manglende sikkerhedsforanstaltninger ved SMS kommunikation og dens forøgede brug til bl.a. notification fra overvågningssystemer.

[6]En sådan løsning kan dog lede til et Availability issue, så det er ikke altid optimalt.

Indbrud
Sidst men ikke mindst har vi simpelt indbrud af andres servere. Nu siger jeg af andres servere og ikke af vores egen, hvilket simpelthen er fordi mange mennesker egentlig vælger ganske fine passwords, men fremfor at dele deres passwords op i sikkerhedszoner eller have et password til hver system, så har de blot 1 eller 2 passwords til alting. Så når det der forum http://example.com/PHPBB-Forum du meldte dig til bliver kompromiteret, så har angriberne pludselig dit brugernavn og i bedste fald dit password hashed med SHA-1.[7] Så kan du ellers bare vente indtil angriberne bruteforcer passwordet lokalt og efterfølgende bruger det til at logge ind på dit supersikre semi-ubrydelige system. Og hvad der er endnu bedre: Du finder sandsynligvis aldrig ud af de stjal dit password fra den anden webside, for websiden vil ikke offentliggøre det da det ødelægger deres image.

3.2.7 Biometrics

Den anden mekanisme jeg vil snakke om er Biometriske løsninger. Biometriske løsninger er en måde at implementere et "Something you are" authenticationssystem, hvor "nøglen" til systemet er en del af din krop, som f.eks. dit fingeraftryk eller dit iris. Man tager således og måler disse data på en eller anden måde og konvertere dette til digital data, som så sammenlignes med nogle gemte værdier for samme person. Grundet at mennesker ændrer sig over tid, så er systemet dog nød til kun at kræve tætte matches mellem authenticationforsøg, da præcise matches fra gang til gang er urealistiske, så et system skal være krævende nok til kun at godkende dem det skal godkende og accepterende nok til ikke at afvise reele brugere. Visse former for biometri giver dog mere konsistente resultater end andre, hvoraf fingeraftryk og iris er de mest kendte.
Fordelen ved at bruge biometri er således at du aldrig skal bekymre dig om at tage din nøgle med dig eller huske dit password. Du har således altid din authenticationnøgle med dig alle steder.

Så har jeg vist også været sød nok overfor Biometriske løsninger nu. Tid til at være lyseslukker! Det er min klare overbevisning *(og heldigvis også overbevisningen af mange fagfolk)*, at biometriske løsninger er en rigtig rigtig dårlig ide. Den første og mest normale kritik er, at du på ingen måde kan identificere dig selv anonymt ved et biometrisk system. Normalvis vil man med brugernavn/password blot kunne vælge noget der slet ikke er relateret til ens person og derfor, i praksis, være anonymiseret. Denne frihed har man ikke med biometriske løsninger, så Big-Brother komplekset kommer hurtig til at være en, muligvis kun opfattet, trussel. Men alligevel en trussel der i visse lande, kunne være meget reel.

[7] Men dog mere sandsynligt slet ikke hashed eller hashed med MD5.

En anden kritik er, at hvor andre systemer antager at intet holdes hemmeligt for evigt, *(i.e. før eller siden brydes din kryptering, eller hvis du bruger samme nøgle/password for lang tid, så finder nogen nøglen til sidst)* så er biometriske løsninger pr. definition at bruge den samme nøgle for evig og altid uden nogen mulighed for at ændre eller revoke noget som helst! Sagt på en anden måde: Stjæler nogen de digitale data for dit fingeraftryk, så er du, pardon my french, fucked! Til evig tid vil en anden kunne identificere sig som dig i biometriske løsninger og du vil aldrig kunne bruge noget som helst der kræver en biometrisk godkendelse.
Sidst men ikke mindst bygger mange af teknologierne også på tåbelige principper hvor man forsøger at være popsmart med implementationen af den biometriske authentication. Dette gælder f.eks. fingeraftrykslæsere på laptops. Aldrig har der været en mere komplet åndsvag ide! Et er, at det i forvejen er uhyre nemt at tage et fingeraftryk af en overflade, et glas, et tastatur, en bil, en cykel, en nøgle, et vindue, en dør, et stykke papir etc. etc.. Men at man så synes det er smart at proppe fingeraftrykslæseren på lige ved siden af over 30 kilder til selv samme fingeraftryk er mere end bare en smule uintelligent. Det svarer til at have et 40 karakter langt password, skrive det på 30 postits og fuldstændig plastrer ens bærbar til med dem.
Nu påstår mange producenter så, at sådanne stjålne fingeraftryk ikke kan snyde deres systemer, fordi de bruger ting som pulsmåler, fedtmåler, temperaturmålere og mange andre ting til at detektere om det er en reel finger eller om det er en forfalsket finger. Men som mange uafhængige grupper har vist, heriblandt Chaos Computer Club i Tyskland der stjal en ministers fingeraftryk og lavede et POC med at de kunne bryde ind i enhver statslig bygning, eller for den sags skyld noget så simpelt som Mythbusters der på kort tid kunne bryde en af datidens mest anerkendte fingeraftrykslåse.
Pointen er, at der er stor forskel på at få stjålet sit password og på at få stjålet sin biometriske identitet - sidstnævnte er oceaner værre og kan aldrig laves om på igen.. førstnævnte kræver blot en *passwd myuser* kommando.

4 Network Security

4.1 Disposition

1. **Sikkerhedsmål**
   - CIA/CAA
   - Def. sikkert system
2. **Netværkssikkerhed**
   - Prevent-Detect-Recover Policy
3. **Authenticated Key Exchange**
   - Needham-Schroeder
4. **SSL/TLS**
   - SSL Key Exchange
5. **IPSec VPN**
6. **Firewalls**
   - Packet Filtering Firewalls
   - Proxy Firewalls
   - Stateful Inspection Firewalls
7. **IDS**
   - Regel-baseret IDS
   - Statistik-baseret IDS

4.2 Details

Her gives detaljer for hvert punkt i dispositionen og muligvis mere til.

4.2.1 Sikkerhedsmål

Når vi snakker om IT-sikkerhed er det typisk i relation til at ville forbedre på et systems sikkerhed i en eller anden forstand. Hvad det præcist er vi vil forbedre på sikkerheden kommer dog meget an på hvad vi gerne vil opnå.

CIA/CAA
Til det formål opstiller man typisk en række sikkerhedsmål der samlet repræsenterer alle aspekter IT-sikkerhed dækker. I løbet af kurset har vi arbejdet med tre opdelinger:

Confidentiality: Confidentiality er at personer kun skal have adgang til den information de er berettiget til og intet mere. Dette gælder uanset om informationen sendes, opbevares eller behandles.

Authenticity: Authenticity er at information er authentisk, altså ikke manipuleret eller på anden måde ændret af en uautoriseret person. Dette gælder også mht. til authenticity af afsender, modtager mv. Således en bruger ikke kan nares til at tro han har modtaget kommunikation fra en han i virkeligheden ikke har.

Availability: Availability er at vi ønsker vores systemer er tilgængelige når de skal bruges, således legale brugere kan få adgang til deres data og de services et system tilbyder.

Normalvis formulere litteraturen dog ovenstående tre mål som Confidentiality, Integrity og Availability, således det staver CIA, hvilket man åbenbart synes er popsmart! Principperne er dog mere eller mindre de samme.

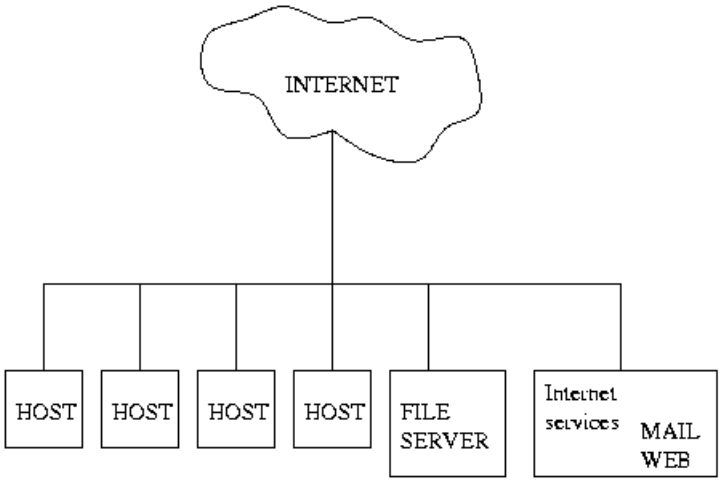
Def. sikkert system
For lige at få det på plads, må vi hellere lige definere hvad et sikkert system er. Det er typisk svært eller umuligt at bevise et system er sikkert, i det at vi så skal til at beskrive systemet ud fra en matematisk model og bevise et teorem om at systmet er sikkert under de og de omstændigheder. Typisk kan det ikke lade sig gøre og hvis det kan, er det typisk fordi vi har antaget så mange forkerte ting om vore angribers muligheder, at vi alligevel ikke ender op med et sikkert system, men blot falsk tryghed.
Så der er i virkeligheden ikke rigtig nogen definition på et "sikkert system", så i stedet nøjes vi med en definition på et "sikret system". Dette gør vi ved at definere en overordnet *Sikkerhedspolitik* på baggrund af en *Trusselsmodel*, og implementerer herefter denne sikkerhedspolitik vha. nogle *Sikkerhedsmekanismer*. På den måde får vi, at et sikret system kan beskrives som:

Sikret system = Sikkerhedspolitik + Trusselsmodel + Sikkerhedsmekanismer

4.2.2 Netværkssikkerhed

Netværkssikkerhed er mange ting og især i dette kursus er der et kraftigt overlap mellem Netværkssikkerhed og Systemsikkerhed. Jeg har derfor valgt

at fokusere på de aspekter af netværks-/systemsikkerhed der omhandler kommunikation over netværket og de sikkerhedsmekanismer der direkte arbejder med netværkstrafik.
Blot lige for at sætte et overblik på tingene, så lad os forestille os vi har et firmanetværk med en gateway ud til omverden, så kan vi løbende udvide på den model.



Prevent-Discover-Recover Policy
Til trods for sikkerhedspolitikker ikke er noget jeg vil gennemgå synderligt her i emnet, så er det dog vigtigt lige at nævne, at de fleste designere af netværk styrer netværkssikkerheden vha. en *Prevent-Discover-Recover* politik. Dette vil sige at man specifikt designer sikkerheden på netværket på en sådan måde, at man har et first-line-of-defense der forsøger at forhindre angreb, et næste lag der forsøger at opdage et angreb fandt sted og præferabelt gemmer så mange data om angriberen som muligt, og så sidst men ikke mindst et lag der sørger for, at hvis det går galt, så går virksomheden ikke nedenom og hjem.

4.2.3 Authenticated Key Exchange

Men lad os nu kigge på vores første fokus, nemlig sikker kommunikation mellem systemer. Vi ved at hvis bruger A og bruger B begge har certificerede offentlige nøgler pk_A og pk_B, så kan vi bruge disse til at skabe en forbindelse mellem de to og få dem til at generere en midlertidig session key K og så kommunikere sikkert vha. denne.
Ikke destro mindre ignorerer vi ofte hvordan A og B helt præcis gør dette, for der er en række krav for at vi kan kalde nøgleudvekslingen reelt sikker.

Overordnet skal tre ting gælde:

- Hvis A ønsker at kommunikere med B, og B ønsker at kommunikere med A, og begge konkluderer at protokollen til nøgleudveksling var succesfuld, så er de enige om session nøglen K.
- Hvis A ønsker at kommunikere med B, og konkluderer at protokollen til nøgleudveksling var succesfuld, så skal det være tilfældet at B deltog med intentionen om at kommunikere med A og at ingen andre end A og B har adgang til session nøglen K. Samme regel gælder for B i forhold til A.
- Den aftalte session nøgle K skal være "frisk" i den forstand at den ikke må have været brugt før. En eventuel angriber må altså ikke kunne tvinge A eller B til at bruge en gammel session nøgle.

Generelt kan vi dog ikke kræve begge brugere er enige om hvorvidt protokollen var succesfuld. Eksempelvis kunne en angriber forhindre en verifikation fra B til A i at dukke op - i så fald ville B tro protokollen var succesfuld, hvorimod A ville tro den fejlede. Denne situation er dog usandsynlig i praksis, da designere af implementationerne burde være kloge nok til at vide man aldrig kan forvente data dukker op på et usikkert medium som et netværk - angriber eller ingen angriber!

Needham-Schroeder
Lad os først kigge på hvordan man ikke bør gøre det, ved at kigge på Needham-Schroeder løsningen fra 1978. Løsningen antager brugerne A og B allerede kender hinandens offentlige nøgler.

1. A vælger en nonce n_A og sender $E_{pk_B}(ID_A, n_A)$ til B.
2. B dekrypterer beskeden, tjekker at ID_A er gyldig *(ud fra en CA)* og sender $E_{pk_A}(n_A, n_B)$ til A.
3. A dekrypterer beskeden og tjekker at den korrekte værdi for n_A er i resultatet og sender herefter $E_{pk_B}(n_B)$ tilbage til B.
4. B dekrypterer og tjekker at den korrekte værdi for n_B er i resultatet.
5. Herefter kan A og B bruge n_A, n_B til at skabe en session nøgle K.

Problemet ved ovenstående løsning er, at den ikke er sikker så snart vi bruger protokollen på en utilsigtet måde. Vi forestiller os vi har en ond bruger E. Nu kan E snyde løsningen således:

1. A starter en session med E, så A sender $E_{pk_E}(ID_A, n_A)$ til E.
2. E dekrypterer beskeden og starter en session med B, hvor han sender $E_{pk_B}(ID_A, n_A)$ til B og derved forsøger at nare B til at tro E er A.
3. B dekrypterer beskeden og ser at ID'et passer med A, så B sender $E_{pk_A}(n_A, n_B)$ tilbage til E.
4. E kan ikke dekryptere denne besked, men kan blot fremsende den til A, som vil dekryptere beskeden og se n_A i indholdet som den forventede.
5. A sender derfor $E_{pk_E}(n_B)$ tilbage til E, hvorefter E dekrypterer beskeden og sender $E_{pk_B}(n_B)$ til B.
6. B dekrypterer og ser så det han forventede, så han accepterer.

På den måde har E nu snydt løsningen, da B tror han snakker med A, hvor han i virkeligheden snakker med E. Vi skal derfor have en smartere løsning, og til dette formål er SSL/TLS en god kandidat.

4.2.4 SSL/TLS

Secure Socket Layer / Transaction Security Layer (SSL/TLS) bygger på lidt andre principper end Needham-Schroeder, ved bl.a. at bruge digitale signaturer, således at brugere skal signere en nonce valgt af den anden bruger. Nu nævner jeg både SSL og TLS, og det er simpelthen blot fordi SSL er den standard der er i brug flest steder nu, men TLS begynder så småt at erstattet den en hel række steder. Derfor bør begge nævnes, til trods for forskellene er minimale.
Som noterne nævner bruges SSL ofte til sikre HTTP forbindelser i form af HTTPS forbindelser, men modsat hvad noterne siger, så er SSL slet ikke begrænset til HTTP trafik. SSL bruges ligeledes til at kryptere FTP-forbindelser *(i form af FTPS)*, IMAP4 forbindelser *(i form af IMAPS)*, POP3 *(i form af POP3s)* og en række andre ting. Det er en protokol der implementeres imellem Applikations- og Transportlageret af den 4-lags DoD-modellen (også kaldet IP-modellen), således at det er den pågældende applikation der skal have understøttelse for SSL for at kommunikationen virker.
Men lad os nu kigge på hvordan SSL egentlig virker! Noget der sandsynligvis er overraskende for de fleste er, at SSL som standard kræver at begge sider af forbindelsen har offentlige certifikater med tilsvarende private krypteringsnøgler. Dette er selvfølgelig upraktisk da langt langt hovedparten af brugere netop ikke har dette og det er da også oftest "one-sided" SSL man ser i praksis - men protokollen forventer som sådan som standard et certifikat på begge sider.
Først skal vi lige se på hvordan SSL er opbygget. SSL består af adskillige protokoller, som f.eks.:

Record Protocol: Record protokollen er beregnet til den rå overførelse af data, og bruges af de andre komponenter af SSL til at styre datatransport. Desuden bruger den en såkaldt *cipher spec*, som er de algoritmer parterne er blevet enige om at bruge til forbindelsen. Denne cipher spec er således tom i starten.

Handshake Protocol: Handshake protokollen er den del af SSL der laver den reele authenticated key exchange. Dette gør den ved at bringe begge parter fra en situation hvor de intet har aftalt, til en hvor de har et fælles *cipher spec* og har udvekslet nøgler. Måden de vælger et *cipher spec* på er at klienten fremsender en liste over de kryptografiske algoritmer den understøtter i aftagende sortering efter preference. Serveren svarer herefter med en meddelse om hvilken algoritme den har valgt.[8]

Change Cipher Spec Protocol: Change Cipher Spec protokollen er blot en enkelt meddelse en part sender til den anden part for at indikere de nu skal til at bruge den aftalte *cipher spec* og de aftalte nøgler.

Alert Protocol: Sidst men ikke mindst er Alert protokollen til for at signalere fejl til den anden part.

SSL Key Exchange

Lad os nu tage et dybere blik på hvordan Handshake protokollen laver det reele key exchange. Som noterne siger er denne gennemgang meget forenklet, men beskriver alligevel de essentielle pointer. Vi har en klient C og en server S:

1. C sender en "hello" besked indeholdende en nonce n_C den har valgt.
2. S sender en nonce n_S tilbage, samt dens certifikat $Cert_S(ID_S, pk_S)$.
3. C verificerer certifikatet ved at kontakte en CA og vælger herefter en *pre master secret* (pms) tilfældigt. C sender så $E_{pk_S}(pms)$, dens certifikat $Cert_C(ID_C, pk_C)$ og dens signatur af konkateneringen af de to nonces og pms krypteret, altså $sig_C(n_C n_S E_{pk_S}(pms))$.
4. S verificerer så $Cert_C$ og sig_C vha. en CA og hvis de er okay, dekrypterer pms.

[8] Vær opmærksom på man i nogle tilfælde kan snyde her, ved at modificere klientens liste til kun at indeholde svage algoritmer der er nemme at bryde, i håb om at serveren så ville vælge en af disse fordi den tror klienten ikke kan andet. Dette virker dog ikke længere, da nyere versioner af SSL først sender listen af krypteringsalgoritmer som en af de sidste ting i handshaket.

5. S sender herefter en "finished" besked til C, indeholdende et MAC på alle beskeder sendt mellem parterne i handshaket, alt sammen krypteret med pms som nøgle.

6. C verificerer MAC'en, og hvis den er okay sender den selv en MAC tilbage af alle beskeder den har set siden starten af handshaket - igen med pms som nøgle.

7. Nu kan begge parter udregne hemmelige nøgler af de to nonces n_S, n_C og af pms.

Grunden til den sidste "finished" besked er, at parterne derved tjekker at de begge har haft samme view af begivenhederne og at ingen angriberer derved har ændret på beskeder. Denne metode kaldes *Final Authentication of Views* og kan være rigtig nyttig, hvilket også er hvorfor adskillige key exhange algoritmer bruger den. I effekt gør metoden at en angriber ikke vil kunne fungere som andet end en simpel kommunikationskanal, da ethvert forsøg på at manipulere en besked vil blive detekteret i handshaket.

Det er ikke umiddelbart nemt at bevise SSL er sikker, eller at den løser problemet som Needham-Schroeder havde, og desuden er det kraftigt udenfor pensum, så det vil jeg blot undlade at kaste mig ud i.

4.2.5 IPSec (VPN)

IPSec er en teknologi der gør noget tilsvarende SSL og bruges i en række VPN-teknologier, som f.eks. Cisco's VPN. IPSec virker dog på et lavere niveau af DoD-modellen (IP-modellen), nemlig imellem Transportlaget og IPlaget *(kommer an på definition om det er i transportlaget eller imellem de to.. men det er pretty much same same)*. Dette betyder, at modsat SSL der skal være understøttet af hver applikation der vil bruge det, så er IPSec uafhængig af applikation og alt trafik genereret på maskinen bliver derfor routed over IPSec forbindelsen, og er derved krypteret.
Den måde IPSec laver key exchange på, er vha. *Internet Key Exchange* (IKE) protokollen, som er en samling af metoder for hvilket man kan opsætte det man i IPSec terminologi kalder en *Security Association* (SA). Det fungerer ved at parterne bruger offentlige nøgler, eller en pre-shared secret[9], til at authentikere sig i forhold til hinanden, og bruger herefter en algoritme kaldet *Diffie-Hellman Key Exchange* algoritmen.
Diffie-Hellman fungerer således

1. Man vælger et tal g i intervallet $0 \ldots p-1$ hvor p er et stort primtal.

2. A vælger nu et tilfældigt tal a og sender g^a mod p til B.

[9]Daimi bruger pre-shared secret og ikke nøgler

3. B vælger ligeledes et tilfældigt tal b og sender g^b mod p til A.
4. A udregner nu $(g^b \bmod p)^a \bmod p$ og B udregner tilsvarende $(g^a \bmod p)^b \bmod p$.
5. Det viser sig så dette betyder at begge parter nu har $g^{ab} \bmod p$ og kan bruge det som en fælles nøgle.

Diffie-Hellman baserer sig på ideen om at discrete logaritmer er et svært beregneligt problem og det derfor er usandsynligt en angriber vil kunne udregne a ud fra $g^a \bmod p$. Algoritmen giver dog ingen garanti for parten du taler med er den du tror han er, men der er andre aspekter af IPSec der løser dette problem.

Som sagt er IPSec en teknologi der bruges i mange VPN-teknologier, men det er dog langt fra den eneste VPN-teknologi i verden. Andre kendte eksempler er PPTP og OpenVPN. Man kan også bruge en ssh-forbindelse til at lave en billigmands VPN, hvor man simpelthen får ssh-forbindelsen til at være en tunnel der lokalt fungerer som en SOCKS proxy, hvorefter man går ind i f.eks. Firefox og sætter den til at bruge denne SOCKS proxy til al kommunikation. Det er dog ikke et reelt VPN, da det netop fungerer på samme niveau af DoD modellen (IP-modellen) som SSL.

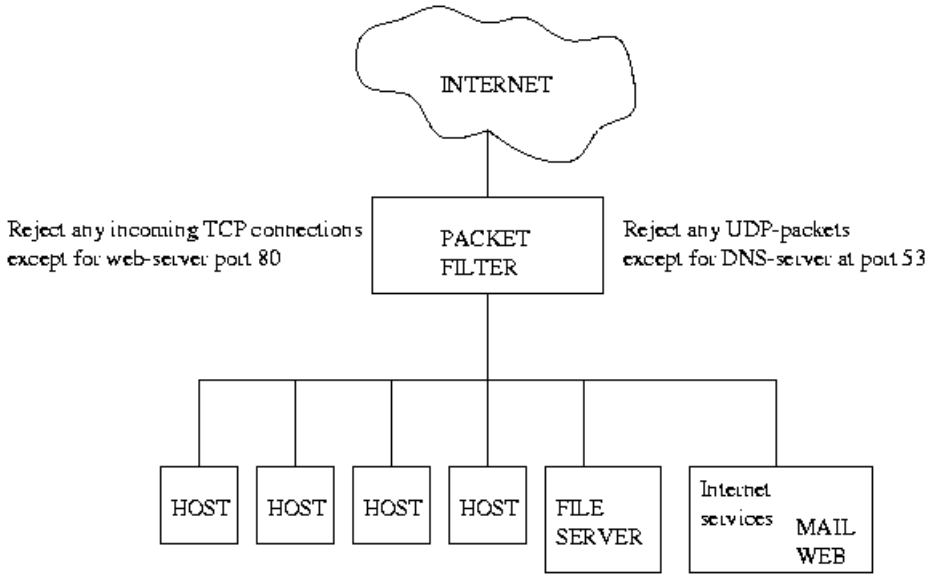
4.2.6 Firewalls

Nu har vi så kigget på nogle løsninger til at beskytte kommunikationen imellem systemer, men selv hvis alle systemer insisterede på at lave disse authenticated key exchanges, så ville de stadig komme til at lave et minimum af kommunikation med ubetroede parter - og herigennem kan disse parter forsøge at udnytte systemet og snyde uden om authentikationen. Vi skal derfor have en måde at bestemme hvad og med hvem vi vil kommunikere. Til dette formål har vi firewalls.
Der findes en række forskellige firewall typer, men fælles for dem alle er, at de på en eller anden måde sorterer i hvilke kommunikationer vi vil tillade. Ifølge mange sikkerhedseksperter er det dog generelt forkert at sige en firewall er til for at blokere trafik. Tværtimod er en firewall til for at tillade trafik, ved at vi definerer hvilke slags kommunikationer vi godt vil have og alt andet er blokeret. Dette kaldes *whitelisting*.
Men lad os nu se på nogle typer af firewalls.

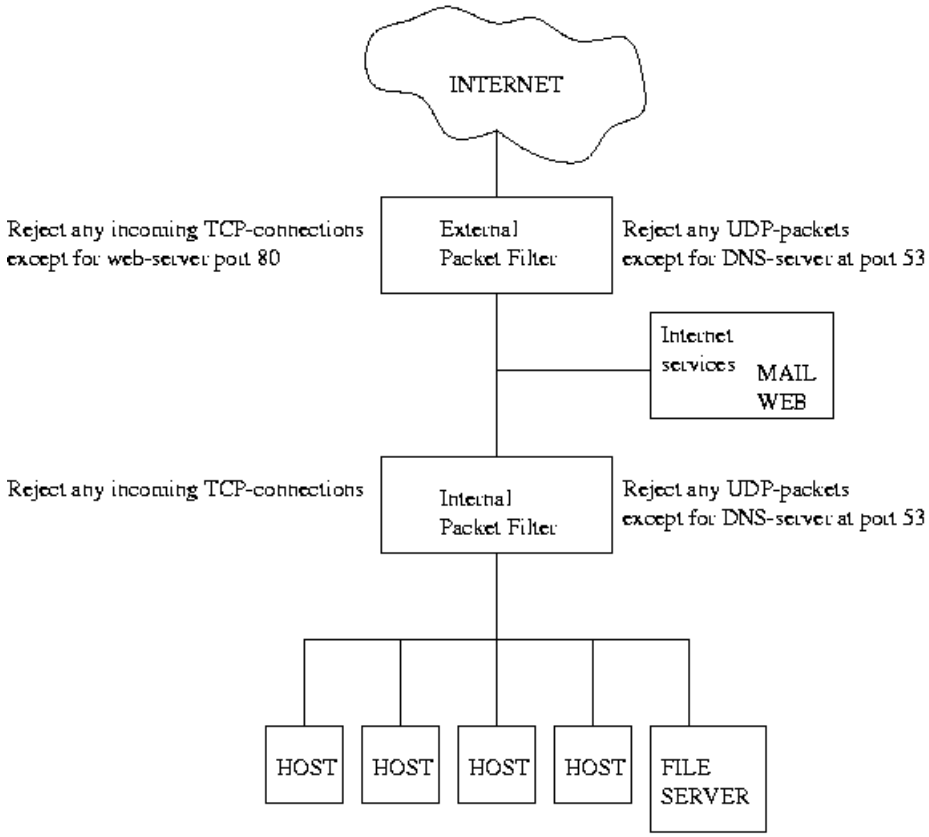
Packet Filtering Firewalls

Den første, og mest simple, type er Packet Filtering firewalls. En Packet Filtering firewall kan kigge på felter som afsender, modtager, protokoltype og SYN/ACK flags i datapakker den modtager, og derigennem f.eks. benægte

al TCP kommunikation med undtagelse af måske port 80 til ens webserver og port 25 til SMTP. Eller den kunne, som i eksemplet nedenfor nægte al trafik pånær port 80 til webserveren og port 53 til UDP.



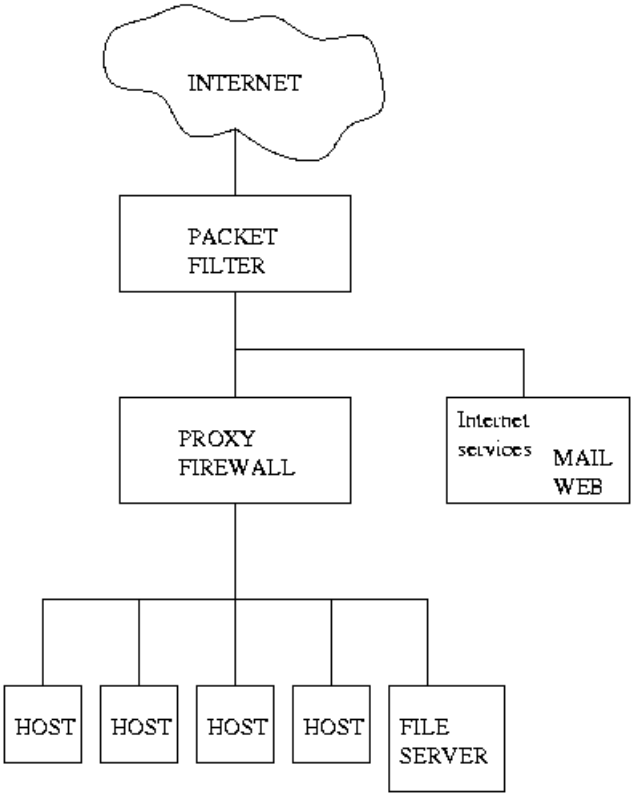
Dette ville dog betyde at hvis webserveren har et sikkershul, så ville en angriber kunne bryde ind på denne og derved lige pludselig have brudt sig om bag firewallen og kan nu kontakte enhver anden maskine direkte. Vi er derfor nød til at bruge 2 packet filtre:



Dette nye, mere usikre, subnet med webserveren kaldes for en *Demilitarized Zone* (DMZ). Man kunne også have lavet et tilsvarende setup vha. en packet filtering firewall der deler netværket op i to logiske subnets *(VLANs)* og laver et af disse om til et DMZ. Her ville de to packet filtre således være implementeret i samme enhed som software.
En packet filtrerings firewall har dog ingen forståelse for state, så vi kan ikke få den til at tillade kommunikation der oprindeligt er startet af brugere på netværket f.eks., eller på anden vis lave mere intelligent filtrering på baggrund af højniveau protokoller eller trafiktyper. Bottom line er, at packet filtre simpelthen er for simplistiske og hvis de skal være sikre betyder det som regel også de bliver umulige at leve med.

Proxy Firewalls
En mere intelligent form for packet filtrering firewall er en såkaldt Proxy Firewall. Sådan en firewall gør som standard at al kommunikation indefra og udefra er blokeret. I stedet skal hvert stykke software på interne maskiner sættes til at bruge Proxy Firewallen som en HTTP Proxy, således al deres trafik køres igennem firewallen, som så sørger for at sende data videre til de reele modtagere *(f.eks. Google eller YouTube)*.

Dette betyder at netværket udefra blot ser ud som en enkelt maskine! Angribere kan simpelthen ikke kontakte andet end proxy firewallen.



Proxy firewalls anses generelt for at være meget sikre, men de er ikke just særlig fleksible. Hvis vi f.eks. har noget software der kræver adgang til noget på internettet, men som ikke har mulighed for at bruge en proxy, så er vi nød til at lave gøjlede løsninger med undtagelser i proxy firewallen for den enkelte applikation eller ved at nare applikationen til at tro den kommunikere med servicen, hvor den i virkeligheden kommunikere med firewallen. Sidstnævnte kan gøres ved at ændre i */etc/hosts* i Linux/BSD og sætte webservicens IP til at være firewallens i stedet - dette vil betyde softwaren tror den kommunikerer med servicen på normal vis, men i virkeligheden bliver dens trafik først sendt til proxien.
Det er dog ikke nogen særlig pæn løsning, så lad os kigge på en lidt mere intelligent firewall.

Stateful Inspection Firewalls
En stateful firewall er enhver firewall der udfører *Stateful Packet Inspection* (SPI), hvilket vil sige at den laver normal packet inspection som andre firewalls, men udover dette holder den også styr på de forbindelser der skabes

igennem den. Derigennem kan den f.eks. se om en pakke hører til en allerede eksisterende forbindelse og afvise pakker på dette grundlag. Dette ser man f.eks. i mange praktiske tilfælde, hvor man ønsker at tillade alle forbindelser udadtil, men meget få forbindelser indadtil. Her kan en Stateful Inspection Firewall sætte en regel, således at forbindelser man har skabt indefra bliver ved med at være godkendte hele vejen igennem deres løbetid, men forbindelser der forsøges skabt udefra efterfølgende, selv hvis det er fra den modsatte part fra sidste session, nægtes adgang.
Stateful Firewalls kan også fungere som det man kalder en *Masquerading Firewall*, ved at den oversætter adresser fra det lokale netværk til noget andet udenfor netværket og derigennem skjuler maskiner på det interne netværk ligesom proxy firewallen. Et eksempel på denne funktionalitet er *Network Address Translation* (NAT), som egentlig kun ved et tilfælde har vist sig nyttig i forhold til firewalls - teknologien blev oprindeligt lavet for at spare IPv4 adresser på Internettet, således kun gateways havde offentlige IP-adresser.
En sidste særlig afart af Stateful Firewalls er de firewalls der analyserer trafikken de ser og leder efter farlige trafikmønstre som forsøg på SQL injections mod en side ved webserveren e.lign. Sådanne firewalls placeres dog typisk dedikeret foran den enkelte server, f.eks. i form af Web Application Firewalls som Mod-Security.

4.2.7 Intrusion Detection Systems

I samme boldgade som disse analyserende Stateful Firewalls har vi såkaldte *Intrusion Detection Systems* (IDS), som ligeledes overvåger og analyserer f.eks. processer, brugere, trafik og en masse andre ting. De kommer i en række forskellige typer, med alt fra trafikanalyserende IDS systemer som Snort, helt til filsystems-integritet IDS'er som Tripwire. Fælles for alle er dog at de enten er *regel-baserede IDS'er* eller *statistik-baserede IDS'er*.

Regel-baseret IDS

Regel-baserede IDS'er definerer et kæmpe sæt regler for hvad normal adfærd er og reagerer hvis adfærden kraftigt afviger fra disse regler. Det kan dog være meget svært at lave disse regler således de reagerer når der skal reageres, men ellers siger intet når der intet er galt.
Eksempler på regel-baserede IDS'er er Snort, Mod-Security og pretty much alle Cisco IDS'er i Ciscos firewallprodukter.

Statistik-baseret IDS

Statistik-baserede IDS'er samler derimod først en helt masse statistik om hvordan det normale system er og hvad der er normal adfærd, og ud fra

det kan den så reagere når der opstår en adfærd der afviger fra det statisk observerede.
Ek eksempel på et statistik-baseret IDS er Tripwire, der netop først opbygger information om hvilke filer der eksisterer på den betroede del af et system og sørger for at gemme hashværdier for alle disse filer. Hvis så filerne engang ændres uforventet, så notificerer den en administrators.

Som jeg sagde i starten bruger man ofte i netværksdesign en Prevent-Detect-Recover sikkerhedspolitik, hvor firewalls er beregnet til at opfylde *Prevent* kriteriet og Intrusion Detection Systems typisk kunne opfylde *Detect* rollen. Men har man ikke noget til at opfylde *Recover* rollen, så hjælper det alligevel ikke meget, så det er vigtigt at sørge for man har procedurer på plads til at reagere på alarmer fra IDS'et - f.eks. ved at blokere den pågældende IP der giver problemer eller i ekstreme tilfælde ligefrem aflukke systemet fra internettet.
Desuden bør *Recover* delen også indeholde nogle backup-procedurer til at gendanne data der bliver destrueret ved et eventuelt dataindbrud. For mange virksomheder betyder et total tab af data det samme som konkurs!

5 System Security and Models for Security Policies

5.1 Disposition

1. **Sikkerhedsmål**
   - CIA/CAA
   - Def. sikkert system
2. **Systemsikkerhed**
   - Prevent-Detect-Recover Policy
3. **Firewalls**
   - Packet Filtering Firewalls
   - Proxy Firewalls
   - Stateful Inspection Firewalls
4. **IDS**
   - Regel-baseret IDS
   - Statistik-baseret IDS
5. **Malware**
   - Trojanske heste
   - Vira
   - Orme
6. **Access Control**
   - Access Control List
   - User Capabilities
   - Opdatering af ACM
7. **Sikkerhedspolitiker**
   - Bell-Lapadula / Biba
   - Chinese Wall
   - Prevent-Detect-Recover
   - Separation of Duty

5.2 Details

Her gives detaljer for hvert punkt i dispositionen og muligvis mere til.

5.2.1 Sikkerhedsmål

Når vi snakker om IT-sikkerhed er det typisk i relation til at ville forbedre på et systems sikkerhed i en eller anden forstand. Hvad det præcist er vi vil forbedre på sikkerheden kommer dog meget an på hvad vi gerne vil opnå.

CIA/CAA
Til det formål opstiller man typisk en række sikkerhedsmål der samlet repræsenterer alle aspekter IT-sikkerhed dækker. I løbet af kurset har vi arbejdet med tre opdelinger:

Confidentiality: Confidentiality er at personer kun skal have adgang til den information de er berettiget til og intet mere. Dette gælder uanset om informationen sendes, opbevares eller behandles.

Authenticity: Authenticity er at information er authentisk, altså ikke manipuleret eller på anden måde ændret af en uautoriseret person. Dette gælder også mht. til authenticity af afsender, modtager mv. Således en bruger ikke kan nares til at tro han har modtaget kommunikation fra en han i virkeligheden ikke har.

Availability: Availability er at vi ønsker vores systemer er tilgængelige når de skal bruges, således legale brugere kan få adgang til deres data og de services et system tilbyder.

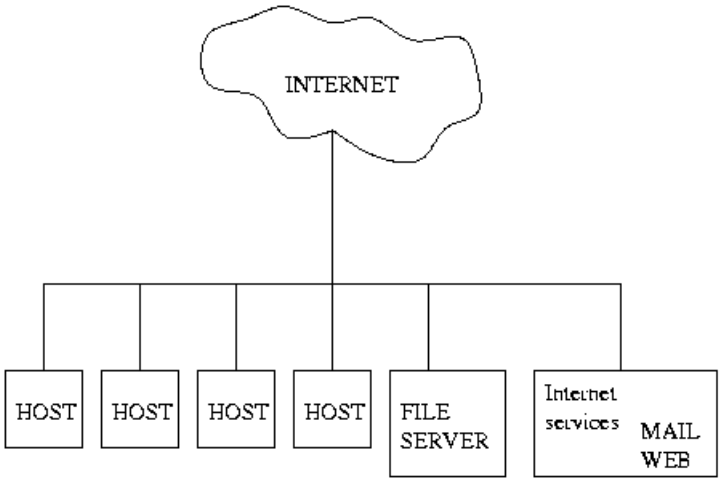
Normalvis formulere litteraturen dog ovenstående tre mål som Confidentiality, Integrity og Availability, således det staver CIA, hvilket man åbenbart synes er popsmart! Principperne er dog mere eller mindre de samme.

Def. sikkert system
For lige at få det på plads, må vi hellere lige definere hvad et sikkert system er. Det er typisk svært eller umuligt at bevise et system er sikkert, i det at vi så skal til at beskrive systemet ud fra en matematisk model og bevise et teorem om at systmet er sikkert under de og de omstændigheder. Typisk kan det ikke lade sig gøre og hvis det kan, er det typisk fordi vi har antaget så mange forkerte ting om vore angribers muligheder, at vi alligevel ikke ender op med et sikkert system, men blot falsk tryghed.
Så der er i virkeligheden ikke rigtig nogen definition på et "sikkert system", så i stedet nøjes vi med en definition på et "sikret system". Dette gør vi ved at definere en overordnet *Sikkerhedspolitik* på baggrund af en *Trusselsmodel*, og implementerer herefter denne sikkerhedspolitik vha. nogle *Sikkerhedsmekanismer*. På den måde får vi, at et sikret system kan beskrives som:

Sikret system = Sikkerhedspolitik + Trusselsmodel + Sikkerhedsmekanismer

5.2.2 Systemsikkerhed

Systemsikkerhed er mange ting og især i dette kursus er der et kraftigt overlap mellem Systemsikkerhed og Netværkssikkerhed. Jeg har derfor valgt at fokusere på de aspekter af netværks-/systemsikkerhed der i betydelig grad påvirker det enkelte system og den måde vi bruger systemerne på.
Blot lige for at sætte et overblik på tingene, så lad os forestille os vi har et firmanetværk med en gateway ud til omverden, så kan vi løbende arbejde os ind i, og udvide på, den model - startende med firewalls.



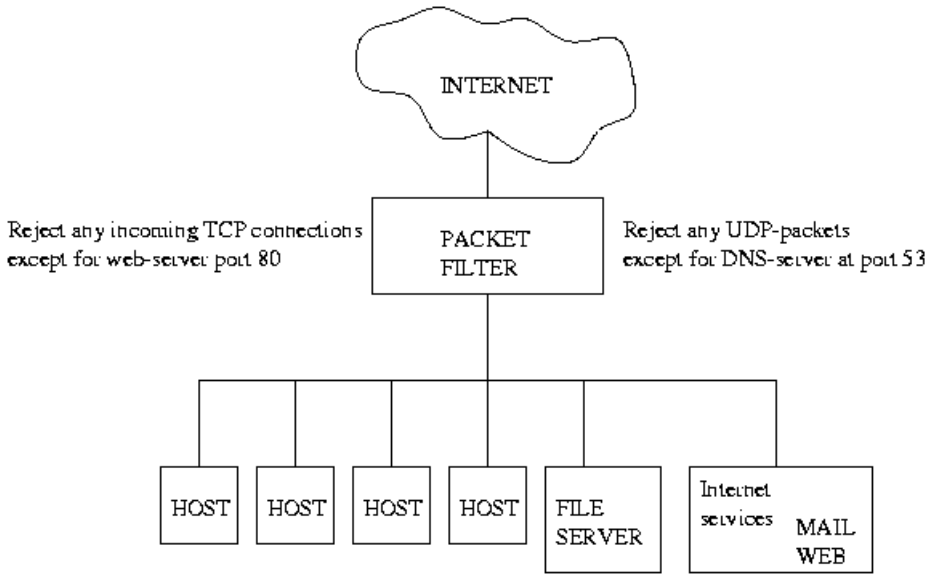
Prevent-Discover-Recover Policy
Der bør for en god ordens skyld kort nævnes at netværk som disse typisk bruger en såkaldt *Prevent-Discover-Recover* politik. Dette vil sige at man specifikt designer sikkerheden på netværket på en sådan måde, at man har et first-line-of-defense der forsøger at forhindre angreb, et næste lag der forsøger at opdage et angreb fandt sted og preferabelt gemmer så mange data om angriberen som muligt, og så sidst men ikke mindst et lag der sørger for, at hvis det går galt, så går virksomheden ikke nedenom og hjem.

5.2.3 Firewalls

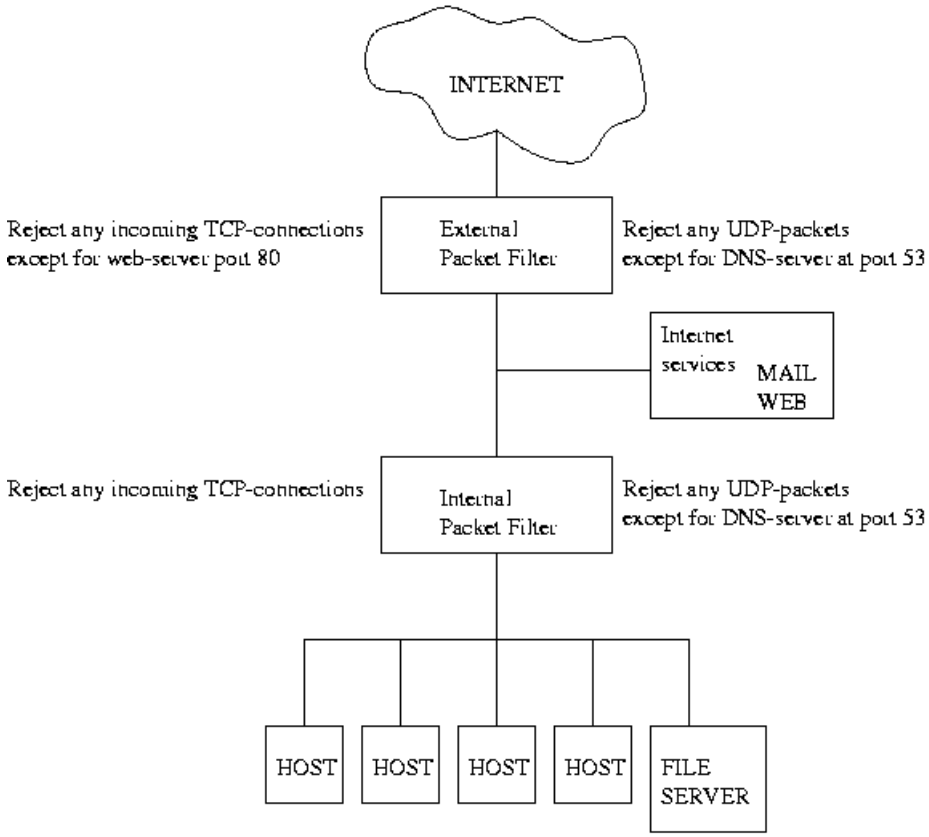
I de andre emner har vi fokuseret meget på kryptografiske løsninger af forskellig art, som f.eks. kryptografiske metoder til at beskytte kommunikationen imellem systemer, men selv hvis alle systemer insisterede på at lave

authenticated key exchanges, så ville de stadig komme til at lave et minimum af kommunikation med ubetroede parter - og herigennem kan disse parter forsøge at udnytte enkelte systemer og snyde uden om authentikationen. Vi skal derfor have en måde at bestemme hvad og med hvem vi vil kommunikere. Til dette formål har vi firewalls.
Der findes en række forskellige firewall typer, men fælles for dem alle er, at de på en eller anden måde sorterer i hvilke kommunikationer vi vil tillade. Ifølge mange sikkerhedseksperter er det dog generelt forkert at sige en firewall er til for at blokere trafik. Tværtimod er en firewall til for at tillade trafik, ved at vi definerer hvilke slags kommunikationer vi godt vil have og alt andet er blokeret. Dette kaldes *whitelisting*.
Men lad os nu se på nogle typer af firewalls.

Packet Filtering Firewalls
Den første, og mest simple, type er Packet Filtering firewalls. En Packet Filtering firewall kan kigge på felter som afsender, modtager, protokoltype og SYN/ACK flags i datapakker den modtager, og derigennem f.eks. benægte al TCP kommunikation med undtagelse af måske port 80 til ens webserver og port 25 til SMTP. Eller den kunne, som i eksemplet nedenfor nægte al trafik pånær port 80 til webserveren og port 53 til UDP.



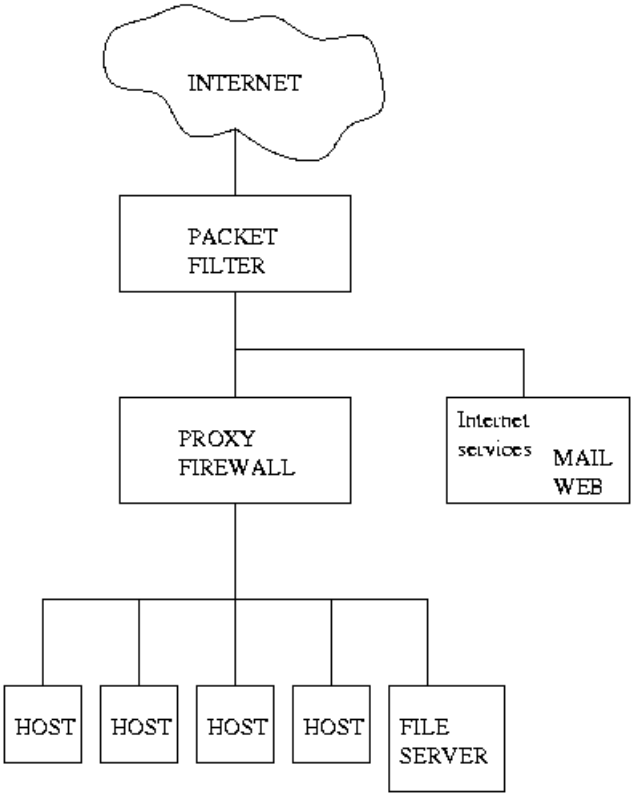
Dette ville dog betyde at hvis webserveren har et sikkershul, så ville en angriber kunne bryde ind på denne og derved lige pludselig have brudt sig om bag firewallen og kan nu kontakte enhver anden maskine direkte. Vi er derfor nød til at bruge 2 packet filtre:



Dette nye, mere usikre, subnet med webserveren kaldes for en *Demilitarized Zone* (DMZ). Man kunne også have lavet et tilsvarende setup vha. en packet filtering firewall der deler netværket op i to logiske subnets *(VLANs)* og laver et af disse om til et DMZ. Her ville de to packet filtre således være implementeret i samme enhed som software.
En packet filtrerings firewall har dog ingen forståelse for state, så vi kan ikke få den til at tillade kommunikation der oprindeligt er startet af brugere på netværket f.eks., eller på anden vis lave mere intelligent filtrering på baggrund af højniveau protokoller eller trafiktyper. Bottom line er, at packet filtre simpelthen er for simplistiske og hvis de skal være sikre betyder det som regel også de bliver umulige at leve med.

Proxy Firewalls
En mere intelligent form for packet filtrering firewall er en såkaldt Proxy Firewall. Sådan en firewall gør som standard at al kommunikation indefra og udefra er blokeret. I stedet skal hvert stykke software på interne maskiner sættes til at bruge Proxy Firewallen som en HTTP Proxy, således al deres trafik køres igennem firewallen, som så sørger for at sende data videre til de reele modtagere *(f.eks. Google eller YouTube)*.

Dette betyder at netværket udefra blot ser ud som en enkelt maskine! Angribere kan simpelthen ikke kontakte andet end proxy firewallen.



Proxy firewalls anses generelt for at være meget sikre, men de er ikke just særlig fleksible. Hvis vi f.eks. har noget software der kræver adgang til noget på internettet, men som ikke har mulighed for at bruge en proxy, så er vi nød til at lave gøjlede løsninger med undtagelser i proxy firewallen for den enkelte applikation eller ved at nare applikationen til at tro den kommunikere med servicen, hvor den i virkeligheden kommunikere med firewallen. Sidstnævnte kan gøres ved at ændre i */etc/hosts* i Linux/BSD og sætte webservicens IP til at være firewallens i stedet - dette vil betyde softwaren tror den kommunikerer med servicen på normal vis, men i virkeligheden bliver dens trafik først sendt til proxien.
Det er dog ikke nogen særlig pæn løsning, så lad os kigge på en lidt mere intelligent firewall.

Stateful Inspection Firewalls
En stateful firewall er enhver firewall der udfører *Stateful Packet Inspection* (SPI), hvilket vil sige at den laver normal packet inspection som andre firewalls, men udover dette holder den også styr på de forbindelser der skabes

igennem den. Derigennem kan den f.eks. se om en pakke hører til en allerede eksisterende forbindelse og afvise pakker på dette grundlag. Dette ser man f.eks. i mange praktiske tilfælde, hvor man ønsker at tillade alle forbindelser udadtil, men meget få forbindelser indadtil. Her kan en Stateful Inspection Firewall sætte en regel, således at forbindelser man har skabt indefra bliver ved med at være godkendte hele vejen igennem deres løbetid, men forbindelser der forsøges skabt udefra efterfølgende, selv hvis det er fra den modsatte part fra sidste session, nægtes adgang.
Stateful Firewalls kan også fungere som det man kalder en *Masquerading Firewall*, ved at den oversætter adresser fra det lokale netværk til noget andet udenfor netværket og derigennem skjuler maskiner på det interne netværk ligesom proxy firewallen. Et eksempel på denne funktionalitet er *Network Address Translation* (NAT), som egentlig kun ved et tilfælde har vist sig nyttig i forhold til firewalls - teknologien blev oprindeligt lavet for at spare IPv4 adresser på Internettet, således kun gateways havde offentlige IP-adresser.
En sidste særlig afart af Stateful Firewalls er de firewalls der analyserer trafikken de ser og leder efter farlige trafikmønstre som forsøg på SQL injections mod en side ved webserveren e.lign. Sådanne firewalls placeres dog typisk dedikeret foran den enkelte server, f.eks. i form af Web Application Firewalls som Mod-Security.

5.2.4 Intrusion Detection Systems

I samme boldgade som disse analyserende Stateful Firewalls har vi såkaldte *Intrusion Detection Systems* (IDS), som ligeledes overvåger og analyserer f.eks. processer, brugere, trafik og en masse andre ting. De kommer i en række forskellige typer, med alt fra trafikanalyserende IDS systemer som Snort, helt til filsystems-integritet IDS'er som Tripwire. Fælles for alle er dog at de enten er *regel-baserede IDS'er* eller *statistik-baserede IDS'er*.

Regel-baseret IDS

Regel-baserede IDS'er definerer et kæmpe sæt regler for hvad normal adfærd er og reagerer hvis adfærden kraftigt afviger fra disse regler. Det kan dog være meget svært at lave disse regler således de reagerer når der skal reageres, men ellers siger intet når der intet er galt.
Eksempler på regel-baserede IDS'er er Snort, Mod-Security og pretty much alle Cisco IDS'er i Ciscos firewallprodukter.

Statistik-baseret IDS

Statistik-baserede IDS'er samler derimod først en helt masse statistik om hvordan det normale system er og hvad der er normal adfærd, og ud fra

det kan den så reagere når der opstår en adfærd der afviger fra det statisk observerede.
Ek eksempel på et statistik-baseret IDS er Tripwire, der netop først opbygger information om hvilke filer der eksisterer på den betroede del af et system og sørger for at gemme hashværdier for alle disse filer. Hvis så filerne engang ændres uforventet, så notificerer den en administrators.

Som jeg sagde i starten bruger man ofte i netværksdesign en Prevent-Detect-Recover sikkerhedspolitik, hvor firewalls er beregnet til at opfylde *Prevent* kriteriet og Intrusion Detection Systems typisk kunne opfylde *Detect* rollen. Men har man ikke noget til at opfylde *Recover* rollen, så hjælper det alligevel ikke meget, så det er vigtigt at sørge for man har procedurer på plads til at reagere på alarmer fra IDS'et - f.eks. ved at blokere den pågældende IP der giver problemer eller i ekstreme tilfælde ligefrem aflukke systemet fra internettet.
Desuden bør *Recover* delen også indeholde nogle backup-procedurer til at gendanne data der bliver destrueret ved et eventuelt dataindbrud. For mange virksomheder betyder et total tab af data det samme som konkurs!

5.2.5 Malware

Et typisk mål for mange angribere er at få listet noget ondsindet software ind på vore systemer for enten at ødelægge data, stjæle data eller ligefrem lave en bagdør ind i systemet. Grundlæggende kan man sige der er 3 overordnede opdelinger af malware:

1. Trojanske heste
2. Vira
3. Orme

Trojanske heste
En trojansk hest er typisk noget software der i første omgang virker til at være nyttigt, men hvor softwaren måske rent faktisk har noget funktionalitet brugeren har gavn af, så har den på samme tid også skjulte processer der enten skaber bagdøre i systemet, stjæler og/eller sletter data.
En særlig form for trojansk hest er spyware, som er software der overvåger en bruger mht. Internet brug eller lignende information og sender dette til skaberen af programmet - typisk til videresalg.
Særligt farlig er den form for trojansk hest der bevidst forsøger at stjæle krypteringsnøgler og aflure tastetryk ved indtastning af passwords i f.eks. ens netbank.

Trojanske heste kommer typisk ind på maskinen pga. brugeren downloader noget dumt, men kan også komme ind pga. f.eks. JavaScript exploits i browsere o.lign.

Vira

Vira er en anden form for software, der er specialdesignet til at inficere andre programmer og piggy-back på disse når de bliver kørt. Herigennem spreder de sig så til andre programmer i systemet og gør som regel før eller siden noget skadeligt, som at slette data.
Vira kommer, ligesom Trojanske heste, typisk ind fordi brugeren downloader noget dumt. Men ligesom med trojanske heste, så kan det også skyldes f.eks. JavaScript exploits i browsere eller f.eks. overflow exploits.

Orme

Orme minder om vira i den forstand at de spreder sig ligesom en virus gør. Men modsat vira er de ikke afhængige af andre programmer for at overleve, men er i stedet reele programmer der aktivt forsøger at sprede sig fra maskine til maskine - typisk ved at udnytte et kendt hul i operativsystemet e.lign.
Orme får ofte meget fokus i medierne for at være dybt skadelige for IT-sikkerheden, men i virkeligheden er andre former for malware som vira og trojanske heste typisk langt værre, simpelthen fordi de ikke får samme mediecoverage og befolkningen derfor ikke er lige så påpasselige med at beskytte sig imod dem.

Anti-Malware software

Ligesom der er malware, så er der også software designet til at finde og fjerne malware. Dette gælder især den velkendte Anti-Virus løsning, som baserer sig på en stor gruppe virusanalytikere sidder og skriver virus signaturer på de vira de finder, således anti-virus softwaren herefter kan detektere disse vira vha. signaturen og fjerne dem fra systemet - typisk inden de gør nogen skade.
Visse virusdesignere har dog været lidt opfindsomme og har bl.a. forsøgt at kryptere dele af virusen for at nare anti-malware softwaren. Dog kan de jo ikke kryptere det hele, da der skal være noget plaintext kode der kan dekryptere resten igen.
De fleste er nu til dags godt klar over de skal have Anti-malware software, især inden for Windows verden. Men desværre er der opstået en stor gruppe Apple Mac OS X og *nix amatører der er blevet ganske glade for de alternative produkter, men har lyttet lidt for meget til hypen og Apple's PR

afdeling, og derfor nu er ledt til at tro Macs, Linux, BSD osv. ikke kan blive inficeret med malware. Dette er selvfølgelig så langt fra sandheden som det overhovedet kan være, og når først markedsandelene på disse systemer bliver sådan at malware på dem spreder sig mere effektivt, så vil vi sandsynligvis opleve store problemer med malware på disse platforme.[10]

5.2.6 Access Control

Efter at have snakket en del om de trusler vi møder udefra i form af hackere, malware osv. Og desuden efter at have snakket om firewalls og IDS som beskytter tilgangen til vore systemer, så er det nu på tide at få kigget på hvordan det enkelte system så tester hvad en bruger egentlig har ret til at gøre. Uden en form for access control ville normale, ondsindede eller dumme, brugere kunne skade systemet ved at udføre handlinger der enten slettede store mængder data eller på anden vis ændrede disse. Desuden kunne man også have en situation hvor en legal bruger gjorde noget illegalt, som at stjæle en anden brugers private nøgler og dokumenter. Vi er nød til at implementere access control for at forhindre sådanne ting.

En meget simpel løsning til Access Control, er en Access Control Matrix, hvor indgang $A[s, o]$ således er alle operationer en bruger s har ret til at gøre på object o. Denne løsning skalerer dog elendigt, så i praksis bruges andre løsninger.

Access Control List
Access Control Lists er basalt set det vi kender fra *nix systemer, hvor vi gemmer informationen om rettighederne ved hvert object. Derved kan man slå op på et givent object og finde ud af hvad rettighederne er herpå. I *nix verdenen organiseres dette ved at man har læse/skrive/udførsels rettighedsfelter for ejeren af objektet, ejergruppen for objektet og så for alle andre til sidst, og sætter disse værdier vha. henholdsvis *chmod* og *chown* kommandoerne. Objekter indeholder således ikke værdier for alle brugere på systemet, men blot for dens ejer, ejergruppe og så for alle andre.

User Capabilities
En anden løsning kaldet User Capabilities er hvor man gemmer rettigheder ved brugeren fremfor ved objekterne. På den måde er det nemt at udregne hvad en bruger kan, men svært at se, ud fra et givent objekt, hvem der kan bruge objektet til hvad. Denne løsning er pretty much hvad Microsoft Windows platformen gør brug af, bortset fra de laver en tilføjelse i forhold til

[10] Blot så dig der læser dette ikke tror jeg er en M$-lover eller sådan noget, så bør det nævnes jeg bruger FreeBSD, Gentoo og Debian som mine systemer.. jeg kan dog ikke fordrage Apple.. den er god nok..!

det der hedder Role-Based Access Control, hvor brugere bliver organiseret i roller som "Normale brugere", "Superbrugere", "Administratorer" mm. På den måde kan man sætte nogle standardrettigheder for disse roller, og derved sørge for at brugere altid som minimum har rettigheder som deres rolle.

Opdatering af ACM
Men at have fastlagt hvilke rettigheder en given bruger har er kun en del af problemet, for det vil ofte være påkrævet at man kan ændre disse rettigheder løbende i et system, så vi skal på en eller anden måde kunne opdatere Access Control Matrixen. Og når vi har fundet ud af hvordan, hvem skal så have ret til det?
Generelt er der to fremgangsmåder:

- Mandatory Access Control
- Discretionary Access Control

I førstnævnte har man fastlagte rettigheder og disse ikke kan ændres af brugerne/entiteterne, men i den anden har man mulighed for at ændre, i hvert fald dele af, matrixen som bruger.
Vi er selvfølgelig nød til at sørge for brugere ikke kan give dem selv højere rettigheder end de har fået fra systemets hånd *(i hvert fald ikke uden hjælp fra en administrator)*. Dette kan vi forsøge at teste og bevise, hvilket vi nogle gange kan. Vi gør det ved at problemet beskrives ud fra en given access control matrix A, en bruger s, et objekt o, en given rettighed r og et sæt kommandoer c. Så spørger vi så: Er der et begrænset sæt kommandoer c for hvilket matrixen A ændres til matrixen A', hvor operationen $r \in A'[s, o]$ men hvor denne samme operation $r \notin A[s, o]$? Hvis svaret er nej, kaldes matrix A for sikker i forhold til r. Her kan følgende så vises:

- Hvis sættet af kommandoer kun består af en enkelt operation, så er det bestemmeligt at A er sikker i forhold til r.
- Hvis sættet af kommandoer består af mere end en operation, så er det ubestemmeligt om A er sikker i forhold til r.
- Hvis antallet af brugere er endeligt, så er det altid bestemmeligt om A er sikker i forhold til r.

Det betyder altså, at vi generelt ikke kan sige en given matrix A er sikker i forhold til en given rettighed r, men at vi under særlige omstændigheder dog stadig godt kan bevise lidt.

5.2.7 Sikkerhedspolitiker

Som jeg talte om til starte med, så når vi skal beskrive sikkerhedskrav til et system, så definerer vi typisk noget vi kalder en Sikkerhedspoltik. Jeg vil lige her til sidst, hvis jeg kan nå det, tage et dybere kig på emnet. Lad os lige igen få et overblik hvad en sikkerhedspolitik er for en størrelse, sammen med dens to relaterede begreber trusselsmodellen og sikkerhedsmekanismerne.:

Sikkerhedspolitik: En sikkerhedspolitik er en kort beskrivelse af de sikkerhedsmål vi har for systemet og en højniveau strategy for at opnå disse mål.

Trusselsmodel: En trusselsmodel er en beskrivelse af de mulige angreb vi vil have vores system er beskyttet imod.

Sikkerhedsmekanismer: Sikkerhedsmekanismerne er de tekniske og administrative løsninger vi bruger til at opnå vore sikkerhedsmål.

Så ud fra dette kan vi se, at stortset alt vi har snakket om indtil nu har været enten til trusselsmodellen eller sikkerhedsmekanismer. Men for ordentligt at strukturere vores sikkerhed og have et håb om at blive overbevist om vores systems sikkerhed, er vi nød til at have en sikkerhedspolitik.

Formelt definerer vi i en sikkerhedspolitik hvilke "states" af systemet vi ønsker systemet skal kunne være i, og vores højniveau strategy beskriver så hvordan vi har tænkt os at garantere systemet kun kan gå ind i disse sikre "states". En vigtig ting sikkerhedspolitiken skal adressere er f.eks. også den såkaldte *Trusted Computing Base* (TCB), som er den del af det indre system, som man altid skal kunne regne med opfører sig ud fra dens specifikationer - også selvom den angribes af interne brugere af systemet. For at dette skal kunne lade sig gøre er sikkerhedspolitiken nød til at beskrive systemet og beskrive en højniveau strategi for hvorledes TCB'en holdes sikker *(typisk gøres det vha. hardware security, passwords, biometri e.lign.)*.
Vi vil nu kigge på forskellige modeller for sikkerhedspolitiker. Modellerne er ikke stive fastlagte opskrifter der skal følges til punkt og prikke, men beskriver i stedet nogle grundlæggende strukturer der kan tweakes og kombineres. Man kan også sagtens have en sikkerhedspolitik der slet ikke er baseret på nogen kendt model.

Bell-Lapadula / Biba

For at kunne beskrive vore to første modeller skal vi først definere hvad en *lattice* er. En lattice er et begrænset sæt S og en relation $\leq$, hvor vi for elementer $a, b, c \in S$ ved der skal gælde:

- $a \leq a$
- $a \leq b$ og $b \leq a$ betyder at $a = b$
- $a \leq b$ og $b \leq c$ betyder at $a \leq c$

Man bruger en lattice til f.eks. at beskrive brugers privilegier i forhold til hinanden, således man f.eks. kan sige at $a \leq b$ betyder at man med rettigheder b kan gøre minimum lige så meget som med rettigheder a. Der er flere aspekter til en lattice, bl.a. konceptet af en greatest lower bound og least upper bound, men det vælger jeg dog at se bort fra her for at kunne nå at snakke om nogle flere sikkerhedspolitikmodeller.

Men to af de simpleste modeller der bygger på laticer er henholdsvis *Bell-Lapadula* og *Biba*. De minder uhyre meget om hinanden, men har blot forskellige mål.
Bell-Lapadula blev oprindeligt skabt til militæret og er beregnet til at opnå confidentiality. Dette gør den ved at definere en række klasificeringsniveaur som en mængde lattices, f.eks. *public* $\leq$ *secret* $\leq$ *topsecret*. Vi definerer så to regler:

No read up: Subject s må læse fra object o kun hvis $C(s) \geq C(o)$.

No write down: Subject s må skrive til object o kun hvis $C(s) \leq C(o)$.

Således at folk med højere klacificeringsniveau ikke må skrive til objekter med lavere (for at undgå information leaks) og folk må kun læse fra objekter op til deres egen klacificeringsniveau.
Biba er som sagt samme princip som Bell-Lapadula, men med målet om integritet af information fremfor confidentiality. Derfor omdefinerer den reglerne til at være:

No read down: Subject s må læse fra object o kun hvis $C(s) \leq C(o)$.

No write up: Subject s må skrive til object o kun hvis $C(s) \geq C(o)$.

Således man her sikrer sig højere klacificeret information (i.e. mere troværdig information) ikke besudles af lavere klacificeret information.
Lattice strukturer kan være uhyre simple, som disse vi har lige har kigget på, men de kan også være meget komplekse med meget fine opdelinger. Der findes sågar lattices der deles op i grupper på forskellige måder hvor det pludselig ikke er nær så nemt at sige hvornår noget har højere klacificering.

Chinese Wall
En anden model der ikke baserer sig på lattices er Chinese Wall. I Chinese Wall har vi en situation hvor vi f.eks. har et konsulentfirma C der har en række forskellige kunder, hvor nogle af disse kunder er konkurenter. For ikke at skabe en conflict-of-interest, så skal vi sikre os en given ansat af C ikke kan afsløre information om en konkurent til et firma. Derfor definerer vi et predikat $compete(c_1, c_2)$ hvor c_1 og c_2 er klienter og predikatet kun evaluere til true hvis disse klienter er konkurenter. Hvis $compete(c_1, c_2)$ evaluere til true og en ansat s forsøger at tilgå f.eks. c_1 hvor han har tilgået c_2 før, så skal denne handling forhindres.

Separation of Duty
Separation of Duty kan udføres på to måder, men går generelt op i at dele beslutningen om en handling ud på flere personer. Den første metode kaldes *Dual Control*, hvor der skal to eller flere personer til at udføre en given handling. Noterne gav eksemplet med atommissiler her, men et mere jordnært eksempel kunne f.eks. være en bestyrelse der skal beslutte om der skal bruges penge på en given investering.
Den anden variant kaldes *Functional Separation*, og er en lignende model. Men hvor den anden krævede autorisation fra flere på en gang, så kræver Functional Separation autorisation af flere personer på separate tidspunkter, således at en given handling skal igennem flere organer for at blive endeligt godkendt. Et eksempel noterne giver er at melde sig til eksamen, som kræver både ens egen, men også forelæser, instruktor og studiekontorets deltagelse.

6 Threats and Pitfalls

6.1 Disposition

1. **Sikkerhedsmål**

   CIA/CAA

   Def. sikkert system

2. **Klacificering af angreb**

   STRIDE

   X.800

   EINOO

   TPM

3. **Buffer overflows**
4. **Cross-Site Scripting**
5. **SQL Injections**
6. **Covert Channels**
7. **Cold-boot**

6.2 Details

Her gives detaljer for hvert punkt i dispositionen og muligvis mere til.

6.2.1 Sikkerhedsmål

Når vi snakker om IT-sikkerhed er det typisk i relation til at ville forbedre på et systems sikkerhed i en eller anden forstand. Hvad det præcist er vi vil forbedre på sikkerheden kommer dog meget an på hvad vi gerne vil opnå.

CIA/CAA
Til det formål opstiller man typisk en række sikkerhedsmål der samlet repræsenterer alle aspekter IT-sikkerhed dækker. I løbet af kurset har vi arbejdet med tre opdelinger:

Confidentiality: Confidentiality er at personer kun skal have adgang til den information de er berettiget til og intet mere. Dette gælder uanset om informationen sendes, opbevares eller behandles.

Authenticity: Authenticity er at information er authentisk, altså ikke manipuleret eller på anden måde ændret af en uautoriseret person. Dette gælder også mht. til authenticity af afsender, modtager mv. Således en bruger ikke kan nares til at tro han har modtaget kommunikation fra en han i virkeligheden ikke har.

Availability: Availability er at vi ønsker vores systemer er tilgængelige når de skal bruges, således legale brugere kan få adgang til deres data og de services et system tilbyder.

Normalvis formulere litteraturen dog ovenstående tre mål som Confidentiality, Integrity og Availability, således det staver CIA, hvilket man åbenbart synes er popsmart! Principperne er dog mere eller mindre de samme.

Def. sikkert system
For lige at få det på plads, må vi hellere lige definere hvad et sikkert system er. Det er typisk svært eller umuligt at bevise et system er sikkert, i det at vi så skal til at beskrive systemet ud fra en matematisk model og bevise et teorem om at systmet er sikkert under de og de omstændigheder. Typisk kan det ikke lade sig gøre og hvis det kan, er det typisk fordi vi har antaget så mange forkerte ting om vore angribers muligheder, at vi alligevel ikke ender op med et sikkert system, men blot falsk tryghed.
Så der er i virkeligheden ikke rigtig nogen definition på et "sikkert system", så i stedet nøjes vi med en definition på et "sikret system". Dette gør vi ved at definere en overordnet *Sikkerhedspolitik* på baggrund af en *Trusselsmodel*, og implementerer herefter denne sikkerhedspolitik vha. nogle *Sikkerhedsmekanismer*. På den måde får vi, at et sikret system kan beskrives som:

Sikret system = Sikkerhedspolitik + Trusselsmodel + Sikkerhedsmekanismer

6.2.2 Klacificering af Angreb

Da vi i konstruktionen af trusselsmodeller skal kunne være forholdsvis sikre på vi har fået beskrevet alle sandsynlige angrebsvektorer, så er det ofte en hjælp at have en måde at kategorisere forskellige angreb på. Derfor er der blevet skabt en række forskellige metoder, hvor vi blev introduceret til 4 i løbet af kurset, nemlig STRIDE, X.800, EINOO og TPM. Jeg vil lige hurtigt forklare dem.

STRIDE
STRIDE er en kategorisering af et angreb ud fra en angribers mål med angrebet, altså hvad angriberen forsøger at opnå. STRIDE er således en

sammentrækning af de 6 forskellige ting en angriber ville have som mål, således:

S poofing Identify: At angriberen får mulighed for at udgive sig for at være en anden.

T ampering: At angriberen får mulighed for at manipulere data uden at dette bliver opdaget.

R epudiation: At angriberen får mulighed for at kunne nægte at have gjort noget han/hun reelt gjorde.

I nformation Disclosure: At angriberen får adgang til data han/hun ikke burde se.

D enial of Service: At angriberen får mulighed for at nægte andre adgang til et system.

E levation of privilege: At angriberen får mulighed for at have flere rettigheder i et system end han skulle have haft.

Selvom en angriber har haft et givent mål, så har sikkerhedshændelser typisk den egenskab, at falder en ting, så falder alting - så det er slet ikke unormalt at et angreb har et enkelt mål, men alligevel får langt flere konsekvenser.

X.800

En anden måde at kategorisere angreb på er ved at klacificere ud fra hvordan et angreb er blevet udført. Til dette har vi X.800, som er skabt af de samme mennesker der bragte os X.509 certifikatstandarden. Og ligesom X.509 blev den ganske enkelt skabt med hovedet under armen - X.800 er simpelthen for begrænset i praksis. Den simplificerer simpelthen en angribers handlinger så meget at udbyttet af en klacificering er ufyldestgørende. Ikke destro mindre vil jeg kort forklare den.
X.800 deler angreb op i Passive- og Aktive angreb på en given netværkskommunikation således:

Passive angreb:

Eavesdropping: Angriberen lytter med og kigger på informationen der sendes.

Traffic Analysis: Angriberen kigger kun på hvem der sender til hvem og hvor meget der bliver sendt.

Aktive angreb:

Replay: Angriberen gensender en gammel meddelse.

Modification: Angriberen ændrer data der sendes, injecter nye meddelser af sine egne eller stopper andres meddelser fra at dukke op hos modtageren.

Her ville man så bruge eksempelvis kryptering for at undgå passive angreb, da disse er særligt svære at opdage. Man bør dog være opmærksom på, at traffikanalyse stadig er mulig trods kryptering. Dette er eksempelvis tilfældet med WiFi netværk med kryptering, hvor man stadig kan se hvem der er online på netværket, deres samlede mængde afsendt data, deres MAC-adresse osv. - alt sammen til trods for man ikke engang er logget på netværket.
Aktive angreb derimod er nemme at detektere, men sværere at undgå - så fokus er her blot på, at man bør bruge authentication-løsninger til at sikre integriteten af ens kommunikation. Dette gælder f.eks. Message Authentication Codes (MACs), signaturer mm.

EINOO
En tredje analyse vi kan lave, er ud fra et ønske om at klacificere hvorfra og af hvem vi bliver angrebet. Dette kan være meget nyttigt når vi specificerer vores trusselsmodel og beslutter os for hvilke slags angreb vi vil beskytte os imod, for det er sjællendt praktisk muligt at beskytte sig imod alt.
EINOO er ligesom STRIDE, en sammentrækning af dens 5 klacificeringer, hvor de første 2 bogstaver er "Hvem" og de sidste 3 er "Hvorfra".

E xternal attackers: Angribere/Hackere der ikke er legale brugere af systemet.

I nsiders: Angribere der er brugere på vores system og derved allerede har et vis niveau af adgang.

N etwork attacks: Angreb hvor angriberen kun kan lytte til og måske modificere netværkstraffik. Det er næsten umuligt at forhindre netværksangreb fra at blive forsøgt, men de kan gøres ubrugelige vha. kryptering *(med undtagelse af traffikanalyse dog, som stadig kan virke).*

O ffline attacks: Angreb hvor angriberen får uautoriseret adgang til information der er permanent lagret i vores system, på disk eller lignende medie. Dette gælder f.eks. tyveri af passwordfiler, krypteringsnøgler

mm. Offline angreb er sværere at udføre, da de kræver du bryder systemets authentication på en eller anden måde, men når du så også først har fået adgang, så er løbet lidt kørt. Selv hvis du har krypteret en masse af dataene, så skal dine nøgler jo være et eller andet sted, og for manges vedkommende betyder det samme maskine.

O nline attacks: Angreb hvor angriberen bryder ind i systemet og observerer det imens processer der håndterer følsom data kører. Angriberen kan nu forsøge at læse hemmelige nøgler ud af RAM, modificere hvad der bliver vist til brugeren og på anden vis manipulere med systemet. Online angreb er meget sværere at udføre end offline angreb, da man skal bryde authenticationen ligesom med offline angreb, men man skal også være der på det rigtige tidspunkt for at udnytte den proces man ønsker, siden denne muligvis ikke kører konstant. Men når man først har fået adgang til et kørende system som en eller anden bruger, så er det ofte også muligt at udnytte svagheder i serveren til at eskalere ens rettigheder til det af en superbruger *(i.e. root på *nix)*.

IENOO klacificering kan være ganske nyttig når man udformer reele systemers trusselsmodeller, da man ikke altid vil kunne forhindre alle angreb, så kan IENOO hjælpe til at identificere de angrebsvektorer der er mest sandsynlige og sikkerhedspolitiken kan herefter fokusere på disse.

TPM

Den sidste måde at kategorisere et angreb på er ud fra hvilken del af vores forsvar der har fejlet. Sagt på en anden måde, hvorvidt det var vores Sikkerhedspolitik, Trusselsmodel eller Sikkerhedsmekanismer der gjorde angrebet muligt. TPM er endnu engang en sammentrækning for:

T hreat Model: Angrebet var muligt fordi vores trusselsmodel var ukomplet.

P olicy: Angrebet var muligt fordi vores sikkerhedspolitik kommunikerede noget andet end vi ønskede.

M echanism: Angrebet var muligt fordi angriberen kunne bryde uden om vore sikkerhedsmekanismer.

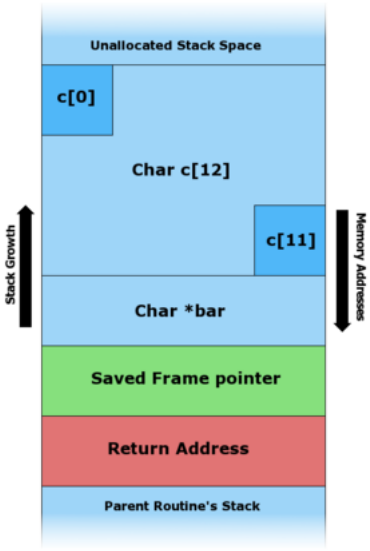
Vha. TPM kan vi så tage et arbitræt exploit på et rigtigt system og derpå se hvilken del af systemets sikkerhed der burde forbedres fremadrettet.

6.2.3 Buffer Overflows

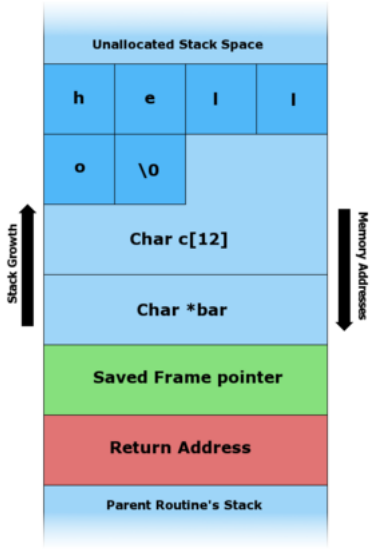
Efter denne gennemgang af forskellige angrebskategoriseringer, så vil jeg nu tage fat i nogle reele eksempler på trusler mod vore systemers sikkerhed.

Startende med de absolut mest normale: Udnyttelse af manglende input validering.!

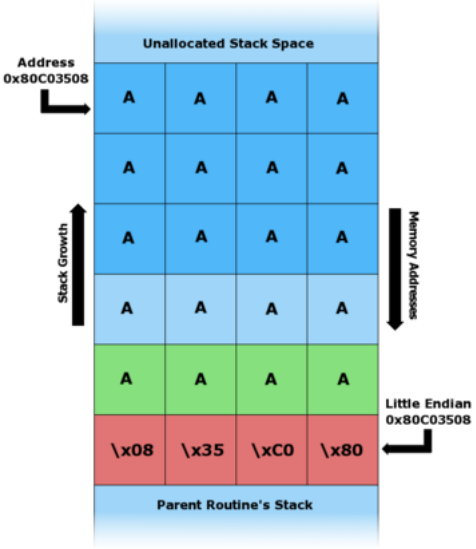
Den første type er også en af de klassisk mest skadelige: Buffer overflows *(også kaldet buffer overruns)*. En buffer overflow opstår når et program allokerer en given mængde plads i en buffer til input, men glemmer at teste det input den modtager rent faktisk kan være i denne mængde plads. Dette kan en angriber så udnytte ved at bruge snedigt udformede inputstrenge der skaber et overflow og ændrer returadressen af en funktion til adressen på noget kode angriberen har udformet. I mange tilfælde vil angriberen derved direkte kunne udføre fuldkommen arbitrær kode igennem et buffer overflow. Lidt mere detaljeret har vi en funktion som vi f.eks. kalder `foo()` der bliver udført af `main()` delen af et C-program og som eneste formål har at kopiere brugerens input til en 12 karaktersbuffer. Når `foo()` udføres har vi således situationen nedenfor *(tak til Wikipedia for illustrationer)*:



Dette er således inden der bliver kopieret noget brugerinput til stakken og char bufferen blot er allokeret sammen med en pointer til input parameter `bar`. Hvis funktionen så bruges "lovligt", som i at vi kun kopierer nogle få tegn, som f.eks. "hello", så virker den fint, som det kan ses her:



Men indsætter vi derimod en meget længere streng, så kan vi vælge vores input på en sådan måde, at vi ændrer returadressen af funktionen til noget andet end `main()` funktionen:



Derigennem har vi udført et buffer overflow.

Tidligere krævede det at skabe buffer overflows en god mængde forståelse, men med opfindelsen af exploit frameworks som Metasploit, så er de dage efterhånden talte og flere og flere ukyndige kan nu lave buffer overflow exploits relativt simpelt.
Buffer overflows er nogle af de mere seriøse sikkerhedsproblemer, da de ofte

leder til komplet rooting af et given system. Altså kan en angriber gå fra at have normal udefrakommende brugeradgang til pludselig at være superuser. Det burde derved være nemt at se, at stortset alle aspekter af STRIDE kategoriseringen er på banen her.
Meget malware udnytter også kendte buffer overflows for at komme ind på et system. Dette er især kendetegnende for orme.
Der er dog måder at beskytte sig imod visse former for buffer overflows. Den første og mest obvious er selvfølgelig at programmørerne lader være med at være fuckhoveder og skriver noget ordentligt kode ved at holde op med at bruge f.eks. `strcpy()` og i stedet bruger sikre alternativer som `_snprintf()` eller en af de vendorspecifikke som Microsofts `strsafe.h` tilføjelse.
En anden løsning, som Howard også kom ind på, er at bruge en såkaldt *Stack Canary*. En stack canary er en integer man sætter i stakken lige før stack retur pointeren, således at et buffer overflow er nød til at overskrive denne integer for at overskrive returadressen. Dette kan programmet derved detektere ved først at tjekke dens canary inden den returnerer, og hvis denne ikke er hvad den forventede så melde fejl.
En sidste løsning for at beskytte sig er ved at bruge *Non-Executable Stacks* i operativsystemet, således at der er en klar opdeling af memoryadresser og data. På den måde sikrer man sig værdier i memorydelen aldrig kan eksekveres og en angriber derfor ikke får mulighed for at manipulere returadressen. Desværre virker denne løsning dog ikke altid i praksis, da man flere gange har fundet måder at flytte returadressen ud af disse memoryafdelinger og ind i en del af stakken hvor værdier gerne må udføres. Det gør dog i det mindste udnyttelse mere besværlig, så det er ikke helt skidt.

6.2.4 Cross-site Scripting

Hvis vi nu i stedet kigger på hvordan vi kan gøre livet surt for webdesignere, så er det igen en udnyttelse af udviklernes manglende input validering i form af såkaldt Cross-site Scripting.
Cross-site Scripting sårbarheder opstår når en webside tager imod brugerinput og efterfølgende fremviser dette input igen enten for brugeren selv eller for andre brugere. Dette kan være alt fra søgefelter der viser "Du søgte efter " beskeder, til forums, til blogkommentaer, til offentlige profiler på diverse social sites.
Et typisk formål med Cross-site scripting er at få ledt brugeren eller brugerens information hen til en anden server. Det kunne f.eks. være en brugers cookie man ville have, som man trækker ud vha. JavaScript og tilføjer som en url parameter til et GET request til en server man har kontrol over. Cookies bliver nogle gange brugt dumt ved at man gemmer ID'er af forskellig art, brugernavne og passwords ubeskyttet i cookies - dette ses desværre alt for tit.

Et andet formål kunne være at udnytte muligheden for at få udført JavaScript i andres browsere til at forsøge indbrud i disses maskiner vha. et hav af JavaScript exploits til brugerens pågældende browser. Dette er en metode der tidligere har vist sig meget effektiv, og som er blevet brugt bl.a. til at inficere maskiner med forskellige former for malware.
Man ser desuden nogle gange sider hvor skaberne til dels har forstået konceptet i input validering, men alligevel har implementeret det fuldkommen forkert. Dette er typisk hvis manden er PHP-programmør og tror `mysql_real_escape_string()` løser alle hans problemer, eller hvis han blot ønsker at brugere skal kunne bruge HTML men ikke skal kunne udføre JavaScript, så han laver måske noget validering der fjerner JavaScript fra inputtet. Dette virker som sådan også, i den forstand at du så ikke kan trække brugeres cookies ud vha. JavaScript, men der er stadig intet der forhindrer dig i at lave et HTML IFrame der refererer til en side angriberen har adgang til, hvor han så kan indsætte alle JavaScript browser exploitsene som jeg omtalte før.
Mulighederne med Cross-site Scripting kan være enorme og farene mange. Og det værste er fejlen går ud over besøgende og ikke websidens ejere. Så det eneste man som bruger kan gøre, er at slå JavaScript fra som standard, men optimalt set burde websideejere blive klogere og bruge funktioner der lavede komplet sanitization af input.

6.2.5 SQL Injection

Et andet webapplikationsangreb, der ligeledes baserer sig på manglende input validering, er SQL injections. SQL injections bruges specifikt til de webapplikationer der interagerer med et eller andet form for DBMS, som f.eks. MySQL, MSSQL, Oracle, DB2 osv.
Det man gør er, at man udnytter den måde en programmør har formuleret sine queries på i et server-side scripting sprog som PHP og laver derfor et kald med en parameter der indeholder arbitræer SQL kommandoer.
Et eksempel kunne være pseudokoden nedenfor *(tak Wikipedia)*:

```
1  statement = "SELECT * FROM users WHERE name = '" + userName + "'
       AND pass = '" + userPass + "';"
2  if (mysql_query(statement) == 1) {
3      // Grant access
4  } else {
5      // Deny
6  }
```

Dette er den typiske password authentication situation, hvor en bruger logger sig selv ind på websidens medlemsafdeling vha. et brugernavn og et password. Men grundet muligheden for SQL injection, kan vi fuldkommen snyde uden om denne authentication mekanisme og logge ind som en superbruger uden at kende dennes password. Dette gør vi vha. et simpelt kald

som det nedenstående:

```
http://example.org/login?userName=superuser&userPass=a' OR '1'='1
```

Denne SQL injection blev brugt til at snyde et authentication system, men jeg kunne lige så godt have lavet en injection som denne:

```
http://example.org/login?userName=superuser&userPass=a';DROP TABLE users;
```

Hvorved jeg så ville have slettet hele tabellen af brugerdata!
At udføre SQL injections kræver dog typisk lidt gætteri først eller en dedikeret information gathering proces, da man først skal finde ud af hvilke tabeller der eksisterer og hvad de indeholder - uden denne information kan det være svært at gøre noget effektivt. Men at gætte disse værdier er typisk ikke svært, da de typisk er meget forudsigelige. En tabel med artikler hedder typisk noget med *Articles*, som f.eks. `TB_Articles` eller `company_articles`.

Der er måder at forhindre SQL Injections på dog og det første og nemmest er at validere brugernes input! Hvis man nu er for doven til dette eller noget lignende, så kan man i stedet bruge en funktion som `mysql_real_escape_string()` i PHP, som sætter et \ foran hver quote i en given streng, således at forsøg på at escape strengen som vi gjorde i første SQL injection ikke virker. Dog er `mysql_real_escape_string()` ikke perfekt - der er situationer hvor den ikke virker.
Alternativt kan man også forsøge at forhindre alle SQL injections på en gang vha. en Web Application Firewall som Mod-Security. Problemet ved denne type Stateful Inspection Firewall er dog, at vi ønsker den skal tillade alt lovlig trafik, men afvise alt ulovlig, og det kan ofte være meget svært at opstille regler der rent faktisk opfylder dette.
Ikke destro mindre er sådanne applikationsfirewalls i hvert fald stadig et plus. Hvis ikke andet kan man sætte dem til at afvise klokke klare forsøg på snyd og så være lidt mere tilgivende når den er i tvivl om et angreb virkelig var et angreb.

6.2.6 Covert Channels

Nu kommer vi så endelig til en udnyttelse der ikke baserer sig på manglende input validering, men derimod på en mangel i trusselsmodellen. Vi tager udgangspunkt i sikkerhedspolitikmodellen kaldet *Bell-LaPadula* (BP), som basalt set siger, at ingen information må skrives fra øvre klasificeringer til nedre klasificeringer *(således at vi undgår information leaks)* og at ingen i nedre klasificeringer må læse information fra øvre klasificeringer. Dette forkertes typisk til **No read up** og **No write down** reglerne.
Men hvis vi nu forestiller os systemerne der implementerer denne opdeling kun eksisterer logisk i software og at to forskellige klasificeringer derved deler

f.eks. en harddisk. Så kan det være muligt at kommunikere information om en højtklasificeret fil ned til lavere klasificeringer ved f.eks. at få læsehovedet på harddisken til at flytte sig i forhold til dataene på en forudbestemt måde mellem parterne. Således at hvis diskhovedet gik udenfor disken, så betød dette at den næste bit var 1 og hvis den gik til et bestemt sted indenfor disken, så betød det 0. Derigennem kunne en bruger på en lavere klasificering se på hvor lang tid det tager disken at reagere på et read request og derigennem udregne det højklasificerede objekts data.
Denne form for udnyttelse kaldes et Covert Channel og kan være meget svær at forhindre når separate dele af systemet ikke er decideret fysisk adskilt.

6.2.7 Cold-boot

Jeg har bevidst undladt at komme med for mange specifikke cases, da de fleste ville være nød til at blive kraftigt simplificerede hvis jeg skulle nå at snakke om dem i løbet af en eksamen. Men en case specifikt, en case der ikke nævnes i pensum, er simpelthen blot for sej til at blive ignoreret.
Sidste år (2008) fandt en række forskere hos Princeton ud af, at de vha. relativt simple metoder kunne stjæle private krypteringsnøgler til fuld harddiskkryptering ud af en computers RAM, ofte selvom maskinen var slukket. Den måde de gjorde det på var ved at de tog en tændt eller dvalende laptop med harddiskkryptering, som så blot stod låst fremme *(som om ejeren lige var gået på toilet e.lign.)*. Herefter udnyttede de noget, som relativt få mennesker ved, netop at RAM faktisk ikke slettes fuldstændig når en computer slukkes, der går altid en mængde tid før data reelt er forsvundet fra en maskines RAM, så der kan gå alt imellem relativt få sekunder og et par minutter før data reelt er forsvundet fra RAM. Dette udnyttede de så, ved at slukke laptoppen og hurtigt boote den op i et simpelt bootprogram der læste hvad der var tilbage på RAM-blokkene. Desværre forfalder bitsene dog for hurtigt til at de ville kunne hive krypteringsnøgler ud, så det de fandt ud af var, at hvis de vendte en normal støvpusterdåse på hovedet og sprayede på RAM-blokkene, så kom der en væske ud der var omkring -50 celcius. Dette gjorde de så før de slukkede laptoppen og kunne herefter tage RAM-blokkene ud af laptoppen og selv efter 10 min sætte dem i en anden laptop og læse indholdet som kørte den oprindelige laptop stadig. Herefter kunne de så direkte aflæse krypteringsnøglen til den krypterede disk ud af RAM-blokkene.

Præsentation og paper er at finde på: http://citp.princeton.edu/memory/