Introduktion til Kryptering

Forfatter: Amalie Due Jensen, Junior IT-Sikkerheds konsulent

Contents

1 Introduktion

Mange af de sårbarheder der lægges vægt på i rapporterne som Retest Security leverer handler om kryptering. Kryptering kan være forvirrende og svært at forstå, da meget litteratur omhandlende kryptering indeholder avanceret matematik. Formålet med denne artikel er at give et letforståeligt indblik i, hvad kryptering går ud på, som supplement til Retest Security's rapporter. Jeg kan ikke love, at matematikken ikke kommer til at tage fart et par gange i løbet af artiklen. Så er der også lidt ekstra til de matematikinteresserede læsere.

Kryptologi er den matematiske disciplin, der benyttes i kryptering. Wikipedias definition på kryptologi er, at det er "læren om hemmeligholdelse af information"[2]. I kryptologien støder man ofte på Alice og Bob, der fungerer som de to parter, der skal sende hemmelig information til hinanden. Det kommer vi også til her. Et af de meget omtalte og hyppige slags angreb i kryptering er det såkaldte man-in-the-middle angreb, hvor den ondskabsfulde Eave formår at placere sig skjult mellem Alice og Bob og er i stand til at aflytte de beskeder, de sender til hinanden. Navnet Eave kommer altså fra det engelske "eavesdropping", der betyder aflytning.

Lad os lige starte med at få styr på et par begreber: Den besked, der skal igennem kryptering, kaldes for plaintekst før kryptering og ciphertekst efter kryptering. Cipherteksten skal være ulæselig, ellers er der blevet brugt en dårlig kryptering. Den "modsatte" proces af kryptering kaldes for dekryptering. Det vil altså sige: Alice har en plaintekst som hun gerne vil vise Bob, uden at nogen andre får adgang til den. Hun laver derfor plainteksten om til ciphertekst ved hjælp af kryptering, sender cipherteksten til Bob, der derefter kan dekryptere cipherteksten tilbage til plainteksten.

I Retest Security's rapporter støder man ofte på begrebet "krypteringscifre", så lad os lige få styr på, hvad det egentlig betyder. En "ciffer" svarer til en "algoritme", hvilket er et begreb man bruger meget i kryptologi og i matematik generelt. Lad os tage fat i Wikipedias definition af en algoritme: "En opskrift til at løse et problem af en bestemt type, som leverer en løsning uanset den konkrete problemsituations udseende"[3]. Sagt med mine egne ord så er en algoritme altså en opskrift til en bestemt måde at løse et problem på, det kan for eksempel være en bestemt måde at kryptere hemmelig information på. En krypteringsalgoritme er altså en bestemt måde at opnå "hemmeligholdelsen af information" på. I kryptologiens historie har der været mange forskellige bud på gode krypteringsalgoritmer. Desværre viser det sig ofte, at algoritmerne alligevel ikke er stærke nok, og så bør de ikke længere bruges. Hvis en svag algoritme supporteres på en enhed, der scannes af Retest Security, bliver den inkluderet som en sårbarhed i rapporten, der leveres.

2 To kategorier i kryptering

I de to følgende afsnit vil jeg bruge Alice og Bob til at forklare de to overordnede kategorier i kryptering, der inddeles i.

2.1 Symmetrisk kryptering

Når Alice og Bob vil benytte sig af symmetrisk kryptering, så har de én nøgle hver. De to nøgler er symmetriske, hvilket betyder at begge nøgler kan bruges til både kryptering og dekryptering. Når Alice vil sende en hemmelig besked til Bob, så krypterer hun altså beskeden med sin nøgle, sender den ulæselige besked til Bob, og så kan Bob dekryptere beskeden med sin nøgle og læse beskeden fra Alice.

De mest kendte symmetriske krypteringsalgoritmer er *DES*, *3DES* og *AES*. Vores computere benytter virkelig ofte *AES* til kommunikation over internettet, så denne krypteringsalgoritme vil jeg komme ind på igen senere.

2.2 Public-key kryptologi (asymmetrisk kryptering)

Når Alice og Bob vil benytte sig af asymmetrisk kryptering, så har de to nøgler hver. De har hver især en privat nøgle, som kun ejeren selv kender, og derudover har de en offentlig nøgle hver, som er offentlig for alle. Når Alice vil sende en hemmelig besked til Bob, så krypterer hun beskeden med Bobs offentlige nøgle, sender den ulæselige besked til Bob, og så kan Bob dekryptere beskeden med sin private nøgle og læse beskeden fra Alice. På den måde er det KUN Bob, der kan dekryptere beskeden og ikke nogen andre, da det KUN er ham, der kender den private nøgle, der kan dekryptere beskeder, der er blevet krypteret med hans offentlige nøgle.

Af en eller anden grund kalder man for det meste asymmetrisk kryptering for public-key kryptologi, selv på dansk.

Den mest kendte public-key krypteringsalgoritme er *RSA*, der blev opfundet af Ron Rivest, Adi Shamir og Len Adleman i 1977. Disse mænd var også mændene bag opfindelsen af public-key kryptologi som generelt begreb.

3 Nøgleudveksling

Der er et interessant dilemma: Vores computere bruger en symmetrisk krypteringsalgoritme til at kryptere den information, vi sender over internettet. Som jeg fortalte, kræver sådan en algoritme, at de to parter, for eksempel en browser og en webserver, hver især har en nøgle, hvor de to nøgler er symmetriske. Men hvordan får de hemmeligt delt sådan et nøglepar, når de endnu ikke har været i kontakt med hinanden?

Den løsning, man i dag bruger, hedder *Diffie-Hellman nøgleudveksling*, opfundet

af Whitfield Diffie og Martin Hellman, og jeg vil nu forklare, hvordan denne nøgleudveksling fungerer:

For at kunne forklare Diffie-Hellman nøgleudveksling, er jeg nødt til først at forklare begrebet *modulo regning*, der er en matematisk operator: Hvis vi gerne vil regne 23 modulo 5, så er resultatet 3, da resten af 23 divideret med 5 er 3. Man skriver det med notationen,

$$23 \bmod 5 = 3.$$

Man finder altså ud af, hvor mange gange 5 går op i 23, hvilket er 4 gange, da $5 \cdot 4 = 20$, og dermed har vi en rest på 3, da der er 3 tilbage op til 23. Et typisk eksempel på moduloregning er uret, hvor vi helt automatisk uden at tænke over det regner modulo 12, da der er 12 inddelinger i et ur.

Næste step i forklaringen af Diffie-Hellman nøgleudveksling er at introducere begrebet *modulo eksponentiering*: Eksponentiering vil sige, at man for eksempel skriver g^5, hvilket betyder at man ganger g med sig selv 5 gange. Det vil sige, at g^5 altså er det samme som $g \cdot g \cdot g \cdot g \cdot g$. I modulo eksponentiering sammensætter vi blot eksponentiering og modulo, dvs. vi regner nu for eksempel $g^5 \bmod n$, hvilket vil sige at vi først udregner g^5 og derefter reducerer resultatet modulo n.

Når Alice og Bob skal lave en Diffie-Hellman nøgleudveksling, så bliver de enige om en værdi for g og en værdi for n, dvs. de bruger altså nogle bestemte tal i stedet for bogstaverne g og n. Lad os tage et eksempel, hvor vi vælger $g = 5$ og $n = 7$:

$5^1 \bmod 7$	5
$5^2 \bmod 7$	4
$5^3 \bmod 7$	6
$5^4 \bmod 7$	2
$5^5 \bmod 7$	3
$5^6 \bmod 7$	1

Table 1: Eksempel på modulo eksponentiering for $g = 5$ og $n = 7$.

Eksemplet i tabel 1 er et meget lille eksempel, i praksis vælges værdierne for g og n til meget meget større tal. Man skal vælge n til at være et primtal, for hvis n er et primtal, så ved vi fra matematikteorien, at vi kan vælge g på en smart måde, nemlig som en *generator*. En generator vil sige, at alle resultaterne af modulo eksponentieringerne er forskellige. I eksemplet i tabel 1 er g en generator, da alle resultaterne er forskellige (tallene fra 1 til 6 optræder præcis én gang hver). Det vil altså sige, at Alice og Bob sammen bliver enige om et stort primtal n og en tilsvarende generator g, og derefter fungerer nøgleudvekslingen på følgende måde:

1. Alice og Bob bliver sammen enige om et stort primtal n og en tilsvarende generator g.

2. Alice vælger en hemmelig værdi a, som KUN hun selv kender. Hun udregner

$$g^a \bmod n,$$

og sender resultatet til Bob.

3. Bob vælger tilsvarende en hemmelig værdi b, som KUN han selv kender. Han udregner

$$g^b \bmod n,$$

og sender resultatet til Alice.

4. Alice kender altså nu a og $g^b \bmod n$, som Bob har sendt til hende. Hun udregner nu $(g^b)^a \bmod n$.

5. Bob kender tilsvarende nu b og $g^a \bmod n$, som Alice har sendt til ham. han udregner nu $(g^a)^b \bmod n$.

6. Der gælder nu det fantastiske, at

$$(g^b)^a \bmod n = g^{ba} \bmod n = g^{ab} \bmod n = (g^a)^b \bmod n.$$

Konklusionen er altså, at Alice og Bob har regnet sig frem til præcis samme værdi, hvilket vil sige, at de nu har udvekslet en hemmelighed. Der er ingen andre end de to, der kender til den hemmelige værdi, da man er nødt til at kende værdien af enten a eller b for at kunne kende hemmeligheden. Utroligt!

Man ved i dag, at primtallet n helst skal have mindst 2048 bits (614 cifre). Man skal altså vælge n langt større end i eksemplet i tabel 1. På figur 1 kan man aflæse størrelsen af n i den sidste søjle (DHE 1024 bits). Der kommer mere om Diffie-Hellman nøgleudveksling senere, bl.a. mere om hvorfor n helst skal være mindst 2048 bits.[4]

4 Digital signatur

Det viser sig, at der er endnu en smart egenskab ved public-key kryptologi, og det er, at det også kan bruges til at konstruere digitale signaturer. Ideen med en digital signatur er, som med signaturer generelt, at man kan bekræfte brugerens identitet. I public-key kryptologi kan man altså udnytte, at det kun er brugeren selv, der kender sin private nøgle. Brugeren kan så bekræfte sin identitet ved at dekryptere en besked med sin private nøgle, og så fungerer resultatet som en signatur på beskeden. Alle andre personer kender brugerens offentlige nøgle, så alle andre kan verificere gyldigheden af brugerens signatur ved at kryptere beskeden med den offentlige nøgle og tjekke, om resultatet er beskeden. Mekanismen fungerer altså på følgende måde: Lad os kalde beskeden

m. Brugeren kan dekryptere m med sin private nøgle k_{privat}, hvilket resulterer i signaturen s:

$$s = k_{privat}(m),$$

hvorefter enhver anden kan verificere signaturen ved at kryptere med brugerens offentlige nøgle $k_{offentlig}$:

$$m = k_{offentlig}(s).$$

Hvis den korrekte m opnås, må brugerens signatur s altså være korrekt, og identiteten er bekræftet.[4]

5 TLS og HTTPS

Transport Layer Security (TLS) nævnes ofte i Retest Security's rapporter, så lad os dykke lidt ned i, hvad det er. TLS er en kryptografisk protokol, dvs. et sæt af regler, der definerer hvordan sikkerhed af følsom information over internettet opnås. TLS anvendes primært til at sikre World Wide Web-trafik mellem en webbrowser og en webserver, der er kodet med HTTP-protokollen. TLS-sikret trafik, der er kodet med HTTP-protokollen, udgør HTTPS-protokollen, som vi alle kender til. Forgængeren for TLS hedder Secure Socket Layer (SSL), men alle versioner af SSL anses i dag for værende fyldt med sårbarheder og skal derfor opgraderes til TLS.

Når en person vil besøge en HTTPS-sikret webside gennem sin browser, så bliver al kommunikationen krypteret. Der findes rigtig mange forskellige krypteringsalgoritmer, så de to parter (webbrowseren og websiden) skal blive enige om, hvordan deres kommunikation skal krypteres. Til dette formål benytter man sig af *TLS cipher suites*, der ikke nemt kan oversættes til dansk, men mit bud er umiddelbart et *TLS algoritmesæt*. Det vil altså sige, at det er et sæt af algoritmer, der sikrer kryptering af information over internettet. Når brugeren gennem webbrowseren kontakter websiden, starter webbrowseren med at fortælle websiden, hvilke algoritmesæt webbrowseren understøtter, og så er det så webserverens ansvar at vælge et algoritmesæt, der understøttes af begge parter. Det vil sige, at algoritmesættet vælges og konfigureres hos webserveren.[5]

I næste afsnit vil jeg kort gennemgå den nuværende status på, hvilke TLS versioner man skal bruge og ikke skal bruge, og i de efterfølgende afsnit vil jeg gå lidt mere ind i detaljer i de versioner, man bør bruge i dag.

5.1 TLS overblik

Jeg har valgt at inkludere følgende tabel, som jeg har fundet på `https://da.wikipedia.org/wiki/Transport_Layer_Security`, da flere af sårbarhederne i Retest Security's rapporter omhandler gamle, supporterede versioner af SSL og TLS. Denne tabel giver derfor et samlet overblik, og som det fremgår af den, bør man i dag ikke understøtte versioner tidligere end TLS 1.2.

Version	Website understøttelse	Sikkerhed	Årstal
SSL 2.0	0.6%	Usikker	Fra 1995
SSL 3.0	3.8%	Usikker	Fra 1996
TLS 1.0	48.7%	Forældet	Fra 1999
TLS 1.1	54.5%	Forældet	Fra 2006
TLS 1.2	99.3%	Afhænger af krypteringstype	Fra 2008
TLS 1.3	42.9%	Sikker	Fra 2018

Table 2: Overblik over TLS og SSL versioner fra februar 2021. Tabellen er taget fra `https://da.wikipedia.org/wiki/Transport_Layer_Security`.

5.2 TLS version 1.2

Som det fremgår af tabel 2, så understøttes TLS version 1.2 stort set overalt på internettet, men sikkerheden "afhænger af krypteringstypen". I dette afsnit vil jeg give et mere detaljeret indblik i, hvordan TLS 1.2 fungerer.

En TLSv1.2 cipher suite eller algoritmesæt består af de fire følgende dele:

1. En algoritme for nøgleudveksling.
2. En algoritme for authentification/digital signatur.
3. En primær krypteringsalgoritme (den algoritme, der faktisk krypterer kommunikationen, der sendes over internettet).
4. En algoritme for Message Authentication Code (MAC).

Figur 1 viser et eksempel på et screenshot, som man meget ofte støder på i en Retest Security rapport. Billedet viser en liste over de supporterede algoritmesæt for en ukendt enhed. Billedet er altså et screenshot fra et `sslscan`. Af billedet fremgår det, at denne enhed understøtter en masse TLSv1.2 algoritmesæt men også en del TLSv1.1. Som tidligere nævnt så skal de to parter (webbrowser og webserver), der skal kommunikere med hinanden, blive enige om en enkelt af disse algoritmesæt, som de derefter vil bruge til at opnå sikker kommunikation.

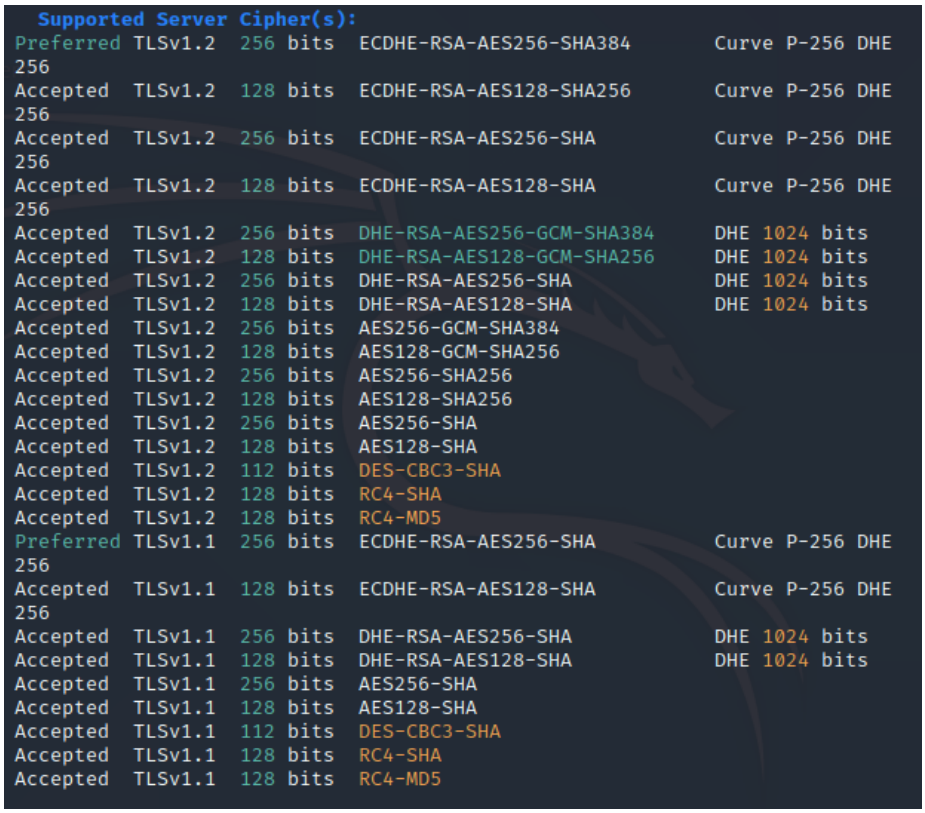

Figure 1: Liste over supporterede algoritmesæt. Screenshot fra et `sslscan` på ukendt enhed.

Jeg vil forsøge at give et lille indblik i, hvad de fire forskellige algoritmer i ét TLSv1.2 algoritmesæt hver især gør og bidrager med. For at gøre det mere overskueligt vil jeg tage udgangspunkt i det førstnævnte algoritmesæt på figur 1. Vi kan altså se, at dette algoritmesæt er fra protokollen TLSv1.2 og er navngivet `ECDHE-RSA-AES256-SHA384`:

1. **En algoritme for nøgleudveksling** er i dette tilfælde `ECDHE`, hvilket står for Elliptic Curve Diffie-Hellman nøgleudveksling. Denne algoritme er altså en variant af den Diffie-Hellman nøgleudveksling, som jeg forklarede i afsnit 3, så den grundlæggende ide er den samme. Ændringen i denne variant er, at man nu opererer over kurvepunkterne på en såkaldt elliptisk kurve i stedet for blot at operere over elementerne modulo n. En elliptisk kurve er en kurve, der er defineret ved ligningen,

   $$y^2 = x^3 + ax + b,$$

   hvor a og b er konstanter, man vælger, og x og y er variable. En elliptisk

kurve ser ud som på figur 2, der er taget fra `https://da.wikipedia.org/wiki/Elliptisk_kurve`.

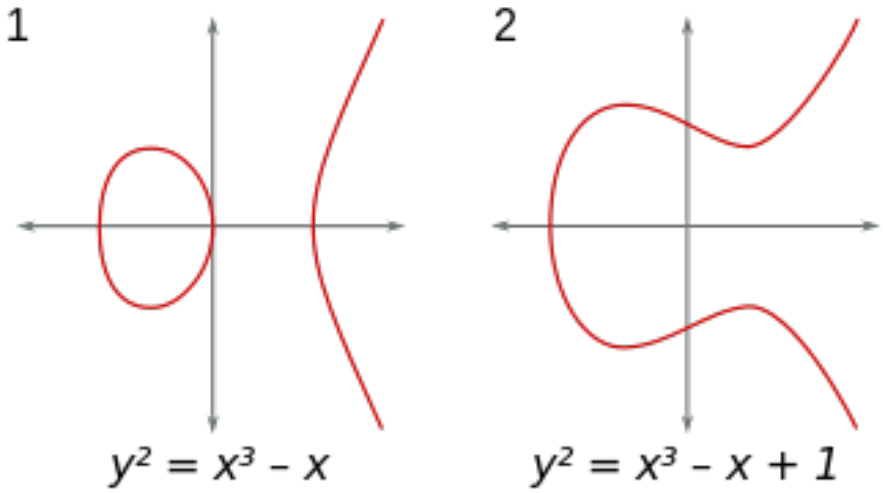

Figure 2: Elliptisk kurve

For at opsummere, så bruges der i dette algoritmesæt præcis den teknik for nøgleudveksling, der er beskrevet i afsnit 3, men over et andet sæt elementer end blot modulo n.

2. **En algoritme for authentification/digital signatur** er i dette tilfælde `RSA`. Jeg nævnte i afsnit 2.2, at RSA er en public-key krypteringsalgoritme, og den kan bruges til at lave digitale signaturer, som jeg beskrev i afsnit 4.

3. **Den primære krypteringsalgoritme** er i dette tilfælde `AES256`, hvilket man ofte ser i TLSv1.2 algoritmesæt, da AES er den mest brugte symmetriske krypteringsalgoritme. Generelt inddeler man de forskellige typer af "primære krypteringsalgoritmer" i to forskellige kategorier:

   - *Block ciphers*: Der findes ikke umiddelbart en god oversættelse til dansk, men som navnet fortæller, krypteres der i blokke. Dvs. man inddeler data i blokke med fast størrelse og krypterer én blok ad gangen. Man skal altså på forhånd beslutte sig for både en nøglestørrelse og en blokstørrelse.

   - *Stream ciphers*: Som navnet fortæller, krypteres der i denne udgave i en lang pseudo-random strøm.

4. **En algoritme for Message Authentication Code** er i dette tilfælde `SHA384`, der er en hashfunktion. Hashfunktionen eller Message Authenti-

cation Code har de to følgende funktioner:

- Autentificering af beskeden: Ligesom at man med digitale signaturer vil autentificere en persons identitet (se afsnit 4), vil man også gerne autentificere en besked, man modtager. Det vil sige, at man gerne vil kunne verificere beskedens oprindelse.
- Dataintegritet: Man vil gerne kunne verificere, at beskeden/data forblev intakt under transporteringen over internettet, dvs. man vil gerne kunne bekræfte, at det faktisk er den rigtige besked, der er nået frem til modtageren.

Der er to primære familier af disse hashfunktioner, *MD5* og *SHA*, og SHA inddeles i SHA-1 og SHA-2. MD5 og SHA-1 betragtes ikke længere som sikre, da man har bevist, at de ikke er kollisionsresistente, som er et af kriterierne for, at en hashfunktion er sikker. En hashfunktion er kollisionsresistent, hvis det er svært at finde to forskellige inputs a og b, hvor deres hashværdier er ens, dvs. $H(a) = H(b)$. Man bør altså bruge SHA-2 i dag, og SHA-2 dækker over 6 forskellige hashfunktioner med hashværdi af længde 224, 256, 384 eller 512 bits: SHA-224, SHA-256, SHA-384, SHA-512, SHA-512/224 og SHA-512/256.[5]

5.3 TLS version 1.3

I august 2018 blev den endelige TLSv1.3 udgivet. Målet med TLSv1.3 er selvfølgelig at forbedre TLSv1.2. I TLSv1.3 ser man bl.a. følgende forbedringer:

- **Øget hastighed**: Den proces hvor to parter (webbrowser og webserver) starter deres kontakt og udveksler TLS algoritmesæt kaldes for *TLS handshake*. Man har formået at gøre dette handshake kortere i TLSv1.3, hvor "kortere" betyder, at antallet af steps med ting og sager der sendes frem og tilbage er reduceret.
- **Øget hastighed**: En anden forbedring er, at man har tilføjet "zero round trip" (0-RTT), hvilket kort fortalt betyder, at når man besøger webservere, man tidligere har besøgt, så kan man hurtigere komme i gang med at sende data til serveren.
- **Øget sikkerhed**: Man har i TLSv1.3 fjernet de sårbare krypteringsalgoritmer, der er en del af TLSv1.2, dvs. blandt andet SHA-1, RC4, DES, 3DES og MD5.[6]

6 Sårbarheden "Medium stærke krypteringscifre supporteret"

Denne sårbarhed optræder i en Retest Security rapport, hvis det er opdaget, at der understøttes svage krypteringsalgoritmer, som navnet på sårbarheden også fortæller.

I sårbarhedsbeskrivelsen i Retest Security rapporterne nævnes SWEET32 angrebet i forbindelse med DES og 3DES, der er block ciphers, og dette angreb kan kort forklares på følgende måde: Som tidligere nævnt så vælger man i block ciphers både en nøglestørrelse og en blokstørrelse, og man ved, at hvis blokstørrelsen er lille, så er block cipheren sårbar over for det såkaldte *birthday attack*. I birthday attacks udnytter man, at sandsynligheden for kollisioner øges, når man laver tilfældige angrebsforsøg. Angrebet kan uddybes med følgende eksempel, der også gav angrebet sit navn: Hvis en lærer i en klasse med 30 elever spørger alle elever om deres fødselsdage, så viser det sig rent teoretisk, at sandsynligheden for at der er to fødselsdage, der falder på samme dag er helt oppe omkring 70%. Hvis læreren derimod havde valgt én bestemt dag, så er sandsynligheden for, at mindst én af elevernes fødselsdage falder på samme dag, kun omkring 7.9%.[10] I SWEET32 angrebet formåede man at benytte sig af birthday attack på DES og 3DES, selvom blokstørrelsen er 64 bits. Det vil altså sige, at man fandt kollisioner, og med kollisioner kan block cipheren brydes.[7]

7 Sårbarheden "SSL/TLS Diffie-Hellman modulus mindre end 2048 bits"

For at kunne forklare denne sårbarhed er jeg nødt til først at introducere noget mere matematik. I forbindelse med public-key krypteringsalgoritmer bruges der ofte én af de to følgende matematiske udfordringer som grundlag for algoritmen, og det er således sværhedsgraden af den matematiske udfordring der bestemmer sværhedsgraden af at bryde den pågældende krypteringsalgoritme. De to matematiske udfordringer er:

- **Faktorisering**: Givet et stort tal N, find de to primtal p og q, således at $N = p \cdot q$. Et eksempel på en public-key krypteringsalgoritme, der bygger på sværhedsgraden af faktoriseringsproblemet er RSA. Hvis faktorisering er nemt, så er det nemt at bryde RSA.
- **Diskret logaritme problem**: Givet tal a og b, find x således at $a^x = b$. Problemet har fået sit navn, fordi x kaldes den *diskrete logaritme af b i basen a*. Et eksempel på en algoritme, der bygger på sværhedsgraden af det diskrete logaritme problem er netop Diffie-Hellman nøgleudveksling. Hvis det er nemt at finde den diskrete logaritme, så er det nemt at bryde Diffie-Hellman.

Som forklaret i afsnit 3, udregner Alice $g^a \bmod n$, og Bob udregner $g^b \bmod n$, hvor a og b er hemmelige og kun kendes af Alice og Bob selv, respektivt. Hvis det er nemt at finde den diskrete logaritme, så er det nemt at finde a og b, og dermed er det også nemt at finde $g^{ab} \bmod n$, som er den hemmelige nøgle, som Alice og Bob bliver enige om at bruge. Heldigvis ER det diskrete logaritme problem svært. Man kan selvfølgelig altid bruge *brute force* teknikken til at finde en nøgle, men det kan tage meget lang tid. Det, som denne sårbarhed handler om, er, at man ofte genbruger de samme værdier for n ude på webserverne, hvilket i

høj grad hjælper hackerne. Hvis man har et bud på værdien for n, kan en hacker precompute de mulige diskrete logaritmer, dvs. man kan på forhånd udregne, hvad det er sandsynligt, at den hemmelige nøgle er. Mange webserverer bruger de samme værdier for n med omkring 512 bits eller 1024 bits. Man skal derfor nu bruge værdier med mindst 2048 bits for at beholde et godt sikkerhedsniveau.[11]

8 Sårbarheden "RC4 cifre supporteret"

Som jeg forklarede under gennemgangen af TLSv1.2 algoritmesættet, skelnes der i symmetriske krypteringsalgoritmer mellem block ciphers og stream ciphers. RC4 er en hurtig og simpel stream cipher, og da det såkaldte BEAST attack var en trussel med block ciphers i 2011, begyndte man at bruge RC4 som en ny løsning. Desværre har det vist sig, at RC4 er fuld af sårbarheder, og man bør derfor holde sig fra den i dag.

RC4 tager som input en 128-bit nøgle og bruger en *key-scheduling algoritme* og en *pseudo-random genererings algoritme* til at lave nøglen om til en lang *keystream*. Krypteringen sker derefter ved at lave XOR operation mellem keystream og beskeden, der skal krypteres. En XOR operation er en matematisk operator ligesom plus og minus, men XOR opererer på bits og bruger følgende regler:

$$1 \oplus 1 = 0$$
$$1 \oplus 0 = 1$$
$$0 \oplus 1 = 1$$
$$0 \oplus 0 = 0$$

Det største problem med RC4 er, at krypteringen ikke er random nok. Man kan finde biases i cipherteksten, hvilket kryptologisk set er en katastrofe. Det betyder nemlig, at ved at kryptere den samme plaintekst med en masse forskellige RC4 nøgler, kan man på et tidspunkt se et samlet billede og bryde krypteringen.

RC4 er også sårbar over for andre stream cipher angreb, så som det såkaldte *reused key attack*. I dette angreb genbruger man den samme RC4 nøgle og på den måde kan man bryde krypteringen og komme frem til plainteksen. Lad os kalde nøglen for K, krypteringen for $E(\cdot)$ og de to plaintekster for A og B. Så foregår krypteringen på følgende måde:

$$E(A) = A \oplus K$$
$$E(B) = B \oplus K$$

Hvis man så kender $E(A)$ og $E(B)$, så kan man på grund af egenskaber ved XOR operationen lave følgende udregning,

$$E(A) \oplus E(B) = (A \oplus K) \oplus (B \oplus K) = A \oplus B \oplus K \oplus K = A \oplus B,$$

hvorefter det er forholdsvis nemt at finde A og B fra $A \oplus B$.

References

[1] Attack of the week: RC4 is kind of broken in TLS, https://blog.cryptographyengineering.com/2013/03/12/attack-of-week-rc4-is-kind-of-broken-in/, besøgt 28/6 2021.

[2] Kryptologi, Wikipedia, https://da.wikipedia.org/wiki/Kryptologi, besøgt 25/6 2021.

[3] Algoritme, Wikipedia, https://da.wikipedia.org/wiki/Algoritme, besøgt 25/6 2021.

[4] Hvad er kryptologi for noget?, Lars Ramkilde Knudsen, udgivet 15/4 2021.

[5] SSL/TLS Cipher Suites, https://www.thesslstore.com/blog/cipher-suites-algorithms-security-settings/, besøgt 2/6 2021.

[6] An Overview of TLS 1.3 - Faster and More Secure, https://kinsta.com/blog/tls-1-3/, besøgt 2/6 2021.

[7] Sweet32: Birthday attacks on 64-bit block ciphers in TLS and OpenVPN, https://sweet32.info/, besøgt 28/6 2021.

[8] TLS, Wikipedia, https://da.wikipedia.org/wiki/Transport_Layer_Security, besøgt 28/6 2021.

[9] Elliptisk kurve, Wikipedia, https://da.wikipedia.org/wiki/Elliptisk_kurve, besøgt 25/6 2021.

[10] Birthday attack, Wikipedia, https://en.wikipedia.org/wiki/Birthday_attack.

[11] Weak Diffie-Hellman and the Logjam Attack, https://weakdh.org/.