AANVALLEN OP WES3$^+$

LEN SPEK & HIDDE WIERINGA

Inleiding

De uitdagende opdracht van het vak Algebra & Security luidde als volgt: *Vind de sleutel die is gebruikt bij het encrypten van de gegeven plain-cyphertext paren, en decrypt daarmee een gegeven cyphertext.*

Hierna zal worden uitgelegd hoe de WES3$^+$ encryptie werkt, wat de aanval inhoudt die wij hebben bedacht, hoe de performance van onze software is, en ten slotte de resultaten die wij hebben gevonden.

WES3$^+$

De encryptie van WES3$^+$ is een samengestelde encryptie. Hij bestaat uit drie rondes WES encryptie, waarbij er key-whitening wordt toegepast op een input die geëncrypt wordt. De block-size van WES3$^+$ is 1 byte, de key lengte is 80 bits, wat overeenkomt met 10 bytes. Voor de key-whitening wordt 1 byte gebruikt, voor de drie WES stappen worden elk 3 bytes gebruikt.

De WES encryptie werkt als volgt voor input byte x en sleutel $k = (k_0, k_1, k_2)$:

(1) Bereken $S(x)$, waarbij S de S-box van AES is (inverse van x in $GF(2^8)$)
(2) Roteer x één bit naar links
(3) XOR met k_i

De eerste twee stappen noemen we $W(x)$. Deze stappen worden drie maal uitgevoerd, en kunnen worden samengevat in een functie

$$y = WES_k(x) = W(W(W(x) \oplus k_0) \oplus k_1) \oplus k_2$$

met x de input en y de output van de encryptie, en k een sleutel van de vorm (k_0, k_1, k_2). De decryptie wordt genoteerd als $x = WES_k^{-1}(y)$.

Decryptie van WES is het omgekeerde van encryptie. Er wordt eerst geXORd met de sleutelbyte, dan geroteerd naar rechts, en ten slotte wordt de inverse in $GF(2^8)$ berekend. Ook deze stappen worden drie maal uitgevoerd.

Om een WES3$^+$ encryptie uit te voeren worden de volgende stappen gebruikt voor input x, sleutel k_w en WES sleutels $k = (k_0, k_1, k_2)$:

(1) XOR de input met k_w
(2) Encrypt de verkregen waarde drie maal, met eerst k_0, dan k_1 en als laatste met k_2

Deze encryptie kan worden samengevat in een functie $y = WES3_k(x)$, met x de input en y de output van de encryptie. De sleutel k is van de vorm

$$(k_w, (k_{0,0}, k_{0,1}, k_{0,2}), (k_{1,0}, k_{1,1}, k_{1,2}), (k_{2,0}, k_{2,1}, k_{2,2})),$$

Date: 17 april 2014.

met elke $k_{i,j}$ en k_w één byte lang. Hierna zullen we een andere indexering gebruiken, volgens

$$k = (k_0, k_1, k_2, k_3, k_4, k_5, k_6, k_7, k_8, k_9).$$

De decryptie wordt genoteerd als $x = WES3_k^{-1}(y)$.

Aanval

Wij hebben WES3$^+$ geanalyseerd, om een aanval te vinden die ervoor zorgt dat niet alle 2^{80} sleutels op een brute-force manier doorgerekend hoeven worden.

Meet-In-The-Middle. Via de colleges en internet kwamen zijn we op het spoor van een Meet-In-The-Middle (afgekort MITM) aanval gekomen, een aanval waarbij geheugen wordt opgeofferd om rekenkracht te besparen. De aanval werkt als volgt (met gegeven plain-cyphertext paren P en C):

(1) Kies een aantal sleutel bytes m dat zal worden opgeslagen,
(2) Encrypt m rondes, voor alle mogelijke x, het resultaat s in een tabel T de bijbehorende $(k_0, ..., k_{m-1})$ op,
(3) Decrypt $n - m$ rondes, voor een aantal C_i. Voor een tussenresultaat s:
    (a) Zoek mogelijke waarden voor $(k_0, ..., k_{m-1})$ horend bij s in T,
    (b) Check of $P_i = WES3_k^{-1}(C_i)$ voor alle i,
    (c) Zo ja, dan is een mogelijke sleutel gevonden.

De complexiteit van deze aanval is $2^m + 2^{n-m}$, waarbij 2^m bytes geheugen nodig is. Als ervoor kan worden gezorgd dat m gelijk is aan $n/2$, wordt de maximale efficiëntie bereikt, namelijk een complexiteit van $2^{n/2} + 2^{n/2} = 2^{n/2+1}$.

Verbetering. Deze aanpak heeft resultaten geleverd, maar is nog niet efficiënt genoeg om in redelijke tijd verder dan de Bronzen uitdaging te komen. We hebben verder gezocht, om de MITM aanval beter te maken.

Eén van de dingen die we hebben gevonden is het elimineren van één van de sleutel bytes. We hebben dit op zo'n manier gedaan dat we zelf de byte konden kiezen die werd geëlimineerd, waardoor onze aanval een factor $2^8 = 256$ sneller werd.

De theorie hierachter is de XOR met de sleutelbyte na elke encryptieronde. Neem aan dat we naar twee plaintexts kijken, x_1 en x_2. We encrypten deze $m-1$ rondes, zodat we $x_{1,m-1}$ krijgen. Dit encrypten we nog één ronde, zodat we

$$x_{1,m} = W(x_{1,m-1}) \oplus k_{m-1}$$

en net zo $x_{2,m-1}$ en $x_{2,m}$ krijgen.

Stel dat we $q_1 = x_{1,m} \oplus x_{2,m}$ uitrekenen. Dit kan worden samengevat als

$$\begin{aligned} q_1 &= W(x_{1,m-1}) \oplus k_{m-1} \oplus W(x_{2,m-1}) \oplus k_{m-1} \\ &= W(x_{1,m-1}) \oplus W(x_{2,m-1}) \end{aligned}$$

omdat twee dezelfde waarden met elkaar geXORd gelijk is aan 0, en iets geXORd met 0 gelijk is aan zichzelf. Hiermee is k_{m-1} uit de vergelijking verdwenen.

Hoe is dit toepasbaar? Als we bij de MITM de tabel maken, slaan we in plaats van $s_i = x_{i,m}$, de gevonden tussenwaarden, de waarden $q_i = x_{i,m} \oplus x_{i+1,m}$ op. Bij het decrypten XORren we $x_{i,m}$ en $x_{i+1,m}$ ook met elkaar, zodat q_i worden gemaakt

Brons	Plaintext	41	42	43	44	45	46	47	48	49	4a	4b	4c
	Cyphertext	fd	2f	1a	cd	06	8b	1b	bc	1e	53	50	d4
Zilver	Plaintext	41	42	43	44	45	46	47	48	49	4a	4b	4c
	Cyphertext	51	c4	8e	cb	d0	a0	e8	88	e4	9e	f5	3b
Goud	Plaintext	41	42	43	44	45	46	47	48	49	4a	4b	4c
	Cyphertext	43	10	f7	76	d6	6c	0c	29	97	37	e7	98

FIGUUR 1. De gegeven plain-cyphertext paren voor Brons, Zilver en Goud (allen in hexadecimale notatie)

Brons	0f	f6	a3	62	a5	b5	0f	ca	0f	92	27	ca	13	62	14
	bb	b5	c0	bf	81	ca	89	20	89	a4	89	89	c7	89	4b
Zilver	d0	ae	4f	7c	cf	ea	a6	2d	d8	e8	4f	50	7d	cf	ea
	d8	42	8d	42	42	d8	42	8d	24	f3	d8	f3	5c	cc	7d
Goud	d6	6c	19	3a	5d	d6	6c	19	f3	5d	d6	6c	19	f3	43	5d	3a	37
	7b	5d	3a	1d	3a	5d	3a	e7	77	5d	77	f0	f3	5d	0c	9d	4d	f7

FIGUUR 2. De cyphertext die ontcijferd moest worden (hexadecimale notatie)

die in de tabel kunnen worden opgezocht. Op dat moment zijn k_0 tot en met k_n bekend, behalve k_{m-1}. Die sleutelbyte kan worden bepaald door

$$k_{m-1} = W(W(...W(P_i \oplus k_0)...) \oplus k_{m-2}) \oplus W^{-1}(W^{-1}(...W^{-1}(C_i \oplus k_n)...) \oplus k_m).$$

Met de volledige sleutel kan worden nagekeken of alle C_i voldoen aan $C_i = WES3_k(P_i)$. Is dit het geval, dan is een sleutel gevonden.

PRAKTIJK EN RESULATEN

In figuur 1 staan de plain-cyphertext paren die zijn gegeven, in figuur 2 staan de cyphertexts die ontcijferd moeten worden.

WES3$^+$. Om een aanval te kunnen uitvoeren moet de encryptie en decryptie eerst efficiënt werken. Daarvoor hebben we als eerste AES S-box voor berekend en in een tabel gezet. Daarna wordt altijd een verschuiving uitgevoerd, dus de combinatie kan worden samengevat in één nieuwe tabel. Voor decryptie kan voor elke output de input worden berekend, en deze ook als (decryptie-)tabel op te slaan.

We hebben geprobeerd een volledige WES3$^+$ encryptie samen te voegen in één tabel, maar dit werd er niet sneller op (waarschijnlijk omdat de tabel niet meer op één *page* in de CPU cache past). Daarom wordt ook in onze code herhaaldelijk dezelfde tabel gelezen.

De lookup table T verdiend nog wat aandacht. Hij bevat per tussenwaarde s alle waarden voor k_0, k_1 en k_2 die van te voren zijn gegenereerd. Echter, per s is er niet precies één setje k_i. Eerst hebben we dit opgelost door een lange vector (`0x1000000` elementen) van `set<int>` te maken, maar dit bleek niet efficiënt. De oplossing die wij hebben gemaakt is een lange linked list, met voor elke s een beginpunt (die worden bijgehouden met aparte pointers). Voor elk begin punt wijst de staart van de link in de lijst naar de volgende node die bij die s hoort. Dit werkt perfect, omdat we weten hoeveel elementen er in de linked list moeten.

	Sleutel	Gevonden tekst	Tijd
Brons	`00 5a ee 1b ca b2 87 87 87 87`	algebra and security 191511410	3.1 (0.6) s
Zilver	`06 98 bc af 9e 5b 7b 9c 9c 9c`	Evariste Galois 1811 1832 20yo	110 (108) s
Goud	`b7 70 15 40 03 f1 7f 2b d2 d2`	EF36 EF37 EF37A 6J5 6V6 6K8 807 GT1C	36986 (36984) s

FIGUUR 3. De gevonden sleutels (hexadecimaal), ontcijferde tekst en rekentijden (in- en exclusief tabel genereren)

Tenslotte wordt deze lijst nog gesorteerd, om ervoor te zorgen dat bij het lezen uit de lijst (bij een gevonden s) niet door de hele lijst van hot naar her wordt gesprongen, wat voor inefficiënte CPU caching zorgt.

Aanval. We hebben als eerste de standaard MITM attack gebouwd en gebruikt om de Bronzen sleutel te berekenen. Dit duurde (toen) ongeveer anderhalf uur. Dat zou betekenen dat Zilver rond de 15 dagen zou duren, wat niet een realistische tijd is om op te wachten.

Onze implementatie slaat 2^{24} sleutelbytes op, en kan dus 3 plain-cyphertext paren tegelijk matchen. Dat beperkt het aantal sleutels dat wordt gevonden aanzienlijk. Voor deze implementatie is een paar honderd MB geheugen nodig.

Nadat we meer onderzoek aan de WES3$^+$ encryptie hebben gedaan en de 4e sleutelbyte hebben geëlimineerd kon een hoop tijd worden gewonnen. Na nog wat andere optimalisatie (geen gebruik van C++ STL, de sortering van de lookup table en multithreading) daalde de zoektijd voor de Bronzen sleutel tot 0.6 seconden. Dat zorgde voor de Zilveren sleutel in een paar minuten.

De Gouden sleutel kostte iets meer moeite. De verwachte zoek tijd was rond de 9 uur, en dit bleek ongeveer te kloppen. Echter werd er in de volle 9 uur twee keer niks gevonden: er zat nog een bug in de code die alleen op Goud te zien was (vanuit de lookup table werd na het eerste element een deel van de volgende sleutel niet goed bepaald). Pas na twee weken zoeken hebben we die eruit kunnen vissen, en de sleutel in een nacht kunnen achterhalen.

Onze uiteindelijke resultaten zijn te vinden in figuur 3. Een voorbeeld output is te zien in figuur 4.

CONCLUSIE

We hebben de theorie van Algebra & Security op een leuke manier kunnen toepassen, die tijdens de huiswerkopgaven niet aan bod kwam. Het is erg leuk om na een tijd zoeken een sleutel te vinden, en te zien hoe goed een kleine optimalisatie kan leiden tot gigantische versnellingen van de zoektocht. We zijn trots op de gevonden Bronzen, Zilveren en Gouden sleutels, en hopen ooit nog een mogelijkheid te vinden om in de buurt van Platina te komen.

```
[0------------------------------50-----------------------------100]
--- Generating reverse save table ---
[............................................................]
--- Sorting reverse save table ---
[............................................................]
    @ 2.454 seconds
--- Finding keys from back ---
[.....................................!......................]
--- Done generating keys from back ---
1 possible keys found
    @ 110.746 seconds

Key {0x6, {0x98, 0xbc, 0xaf}, {0x9e, 0x5b, 0x7b}, {0x9c, 0x9c, 0x9c}} (found at
68.889 seconds):
Evariste Galois 1811 1832 20yo

--- Releasing memory ---
    @ 110.749 seconds
--- Done ---
    @ 110.75 seconds
```

FIGUUR 4. Een voorbeeld output van de gemaakte software