MODERNÍ KRYPTOGRAFICKÉ METODY

Bezpečné použití RSA

V minulém dílu našeho volného seriálu jsme se seznámili s algoritmem RSA, u něhož očekáváme, že se stane jedním z nejpoužívanějších algoritmů pro elektronický podpis. Podrobně jsme se zabývali prvním z řady standardů PKCS pro jeho realizaci – PKCS#1 – a popsali jsme jeho nejpoužívanější verzi 1.5. Formátování podle ní má ale bezpečnostní slabinu. Ukážeme si možná protiopatření a seznámíme se i s připravovanou velmi bezpečnou a perspektivní aktualizací této normy.

PKCS#1 VERZE 1.5 PRO ŠIFROVÁNÍ KLÍČŮ

Nejprve si zopakujme základní informace z minulého dílu. Uvedli jsme formát a doplňování dat podle PKCS#1 ver. 1.5 v případě, že RSA je použit pro šifrování klíčů. Tak je tomu například u populárního protokolu SSL: na jeho počátku klient vygeneruje náhodný klíč sezení, pomocí RSA ho zašifruje a pošle serveru (obě strany pak příslušnou symetrickou šifrou šifrují další komunikaci). Dejme tomu, že tento klíč (D) je 128bitový (má d = 16 oktetů), a RSA nechť má modul 1024 bitů (délka modulu je k = 128 oktetů). Doplnění bloku D do plného 128oktetového bloku EB definuje PKCS#1 verze 1.5 takto: EB = 00 || 02 || PS || 00 || D. Před klíč D je tedy doplněn separátor (00), před ním figuruje řetězec PS (padding string) v podobě 109 náhodných nenulových oktetů (obecně jich musí být alespoň 8) a dále oktety 02 a 00. První nulový oktet vždy zajistí, že EB je menší než modul RSA (n). Druhý oktet indikuje blok typu 02, určený pro šifrování klíčů. Tolik pro zopakování, a nyní se podíváme, kde má tento dobře vyhlížející formát „Achillovu patu".

ÚTOK

Obdržený šifrový blok příjemce nejprve odšifruje a ověří, zda získaný EB' má formát typu 02, aby z něj mohl část D použít jako klíč pro další komunikaci. Ověří tedy, že první dva oktety EB' jsou 00 a 02, za nimiž následuje alespoň 8 nenulových náhodných oktetů (PS) a za nimi separátor 00. Je-li to splněno, blok má formát typu 02 a lze pokračovat v jeho využití, v opačném případě se vrací odesílateli chybové hlášení. Takhle to v protokolu SSL skutečně funguje – chybové hlášení se ale dá zneužít k útoku! Není to zvlášť složité:

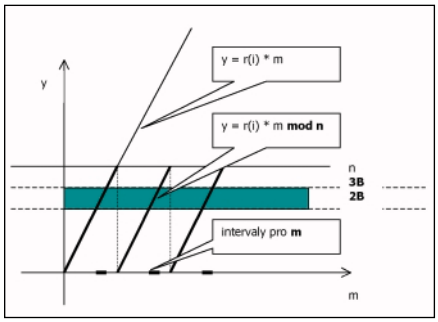
Označme **m** zprávu, kterou vysílající strana zašifruje (blok EB), a c její šifrový obraz. Útočník zachytí na komunikačním kanálu blok c a zaznamená si celou následnou šifrovou komunikaci. Po jejím skončení bude sám serveru posílat modifikované bloky c(i). Po určité době tak získá z chybových hlášení serveru dostatek informací, na jejichž základě lze určit **m**, a tedy i klíč pro původní zaznamenanou komunikaci! Není to geniální?

KRYPTOLOGICKÁ PODSTATA ÚTOKU

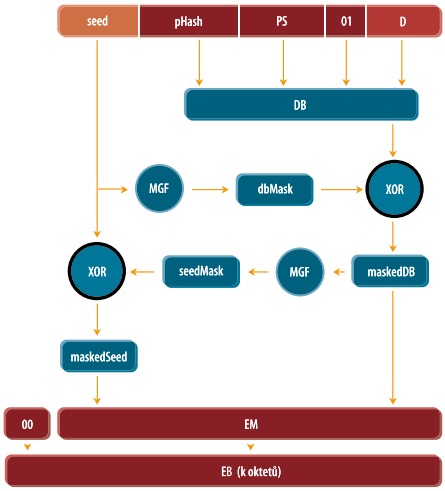
Nyní podrobněji. Útočník volenými konstantami r(i) modifikuje zachycený šifrový text c na řadu jiných šifrových textů c(i) = (c * r(i)^e) mod n, které zasílá přijímající straně (uvidíme, že jich postačí asi milion). Ta vždy daný blok odšifruje a zkontroluje, zda má formát typu 02. Pokud ne, vrátí chybové hlášení. To se bude stávat hodně často, ale útočník bude příjemce obtěžovat novými a novými c(i) tak dlouho, až se „chytí" a chybové hlášení **nepřijde**. To ale znamená, že právě teď po odšifrování zaslaného c(i) vznikla u příjemce zpráva m(i), která **má formát typu 02**. Příjemce nám tím mj. řekl, že na prvních dvou bajtech v m(i) je 00 a 02. Protože platí m(i) = c(i)^d mod n = [c * r(i)^e]^d mod n = [c^d * r(i)^{ed}] mod n = (c^d mod n)*(r(i)^{ed} mod n) = **m** * r(i) mod n, útočník nyní ví, že **m** * r(i) mod n začíná 00 02.

Teď si stačí uvědomit, že **m** je zpráva, kterou chce útočník rozluštit, **r(i) jsou jím volené konstanty** a u čísla **m***r(i) zná už jeho dva bajty! Označíme-li k-oktetové číslo 00 02 ... 00 jako B, pak z uvedeného plyne, že 2B ≤ **m** * r(i) mod n < 3B. Tato nerovnost ale vede k zúžení intervalu pro **m** (0 < **m** < n) na řadu menších intervalů, v nichž může ležet – viz obr. 1. Pokračuje-li útok dále, stejným způsobem jako c(i) lze získat další c(j), která dávají nové nerovnosti a nová zúžení intervalů pro **m**; po určitém počtu kroků lze **m** určit

U ŠIFER VŽDY ZÁLEŽÍ NEJEN NA DEFINICI ALGORITMU, ALE TAKÉ NA JEHO IMPLEMENTACI.



Obr. 1. Omezující intervaly pro m, vyplývající z nerovnosti 2B ≤ m*r(i) mod n < 3B



Obr. 2. Maskovaný formát dat pro šifrování klíčů podle PKCS#1 verze 2.0

už zcela přesně. Celá tato akce vyžaduje jen asi 2^{20} volených šifrových textů c(i). Útok byl reálně vyzkoušen s 512- a 1024bitovým modulem RSA a průměrně bylo potřeba mezi 300 000 až 2 000 000 šifrových textů c(i); blíže viz infotipy [BL].

Možná protiopatření

To podstatné, co se kontroluje po rozšifrování příchozího c(i), jsou první dva bajty. Pravděpodobnost jejich správnosti je poměrně velká – 1/65536. Proto se útočník svými „náhodně" volenými šifrovými bloky do tohoto formátu trefí dost často. Kdyby se ale po odšifrování kontrolovalo více bajtů, pravděpodobnost by rapidně klesla a počet nutných zkoušek by se zvýšil. Stačí proto například kontrolovat, zda separátor 00, který odděluje klíč (D) od náhodných oktetů PS, je přítomen, a zda je na správném místě. V protokolu SSL víme, jak dlouhý klíč očekáváme, a tudíž kde má separátor ležet. Tato kontrola zvyšuje počet nutných zkoušek šifrových textů z jednoho na cca 20 milionů.

Dále v protokolu SSL verze 3.0 je už zabudován mechanismus, kdy se přímo v rámci dat D může předávat i číslo verze protokolu SSL. To je další nadbytečná informace (má dokonce dva bajty – 03 00), kterou můžeme kontrolovat, a útočník pak musí vyzkoušet přes bilion c(i), což je už prakticky neproveditelné.

Některé verze SSL také na nejnižší místa PS místo náhodných dat vkládají osm oktetů s hodnotou 03, což je ještě větší redundance než v předchozím případě. Současně s těmito opatřeními je také vhodné zavést jen jeden typ chybových zpráv, aby z různých chybových zpráv nebylo možné odlišit různé typy událostí. Více o experimentech je v [BL] a na webu PKCS (viz infotipy). V praxi se ale ukázalo, že jen málo serverů zmíněná opatření aplikovalo.

Náprava

Z principu útoku plyne, že do vlastních dat je vhodné zavést redundanci, kterou je možné kontrolovat při příjmu. Náhodnost při doplňování je ale také výhodná, protože šifrování je pokaždé jiné. Spojením obou myšlenek vznikl bezpečný maskovaný formát, který je založen na metodě OAEP (*Optimal Asymmetric Encryption Padding*) a byl definován v **PKCS#1 verze 2.0**. (Možnost použití původního formátu byla zachována, ale jen z důvodu kompatibility se staršími aplikacemi, tentokrát pod označením EME_PKCS1-v1_5). Nyní si jej popíšeme.

Maskovací funkce

Základem maskovaného formátu je hašovací funkce, kterou je obecně možné měnit, ale ve standardu je v kombinaci s EME_OAEP doporučena jen SHA-1 (viz infotipy). Vstupem SHA-1 může být libovolný řetězec, výstupem je haš o délce hLen = 20 oktetů. Pomocí hašovací funkce se vytváří maskovací funkce MGF (Mask Generation Function), jejíž užití je vidět na obrázku 2. Ve standardu je jako MGF doporučena pouze MGF1 (s SHA-1).

MGF má dva vstupní parametry (S, L). Prvním je vstupní řetězec S libovolné délky, druhým je číslo (L), udávající požadovanou délku výstupního řetězce (O). MGF potom pomocí hašovací funkce (Hash) kryptograficky natahuje nebo zkracuje vstupní řetězec S na L-oktetový výstupní řetězec O.

Postup: Výstupní řetězec MGF bude postupně skládán z řetězců o délce hLen, které vzniknou jako Hash(S || COUNTER). Přitom COUNTER je čtyřoktetové vyjádření čítače běžícího od nuly a zvyšujícího se po jedné. Při tvorbě L-oktetového řetězce vygenerujeme nezbytný počet 20oktetových řetězců typu Hash(S || COUNTER) a složíme je za sebe. Z výsledku pak zleva vybereme L oktetů tvořících výstup O. Je-li například potřeba 107oktetový výstup O, tj. O = MGF(S, 107), vytvoří se 120oktetový řetězec Hash(S || 00 00 00 05) || Hash(S || 00 00 00 04) || ... || Hash(S || 00 00 00 00) a z něj se zleva vybere 107 oktetů. Tím je definována maskovací funkce MGF. Její použití a postup tvorby bloku EB pro šifrování klíčů algoritmem RSA vidíte ve zvláštním rámečku.

Vlastnosti maskovaného formátu

Maskovaný formát jednak obsahuje velké množství redundantních informací, jednak využívá znáhodnění. Oba principy jsou vhodně zkombinovány, takže výsledek (EB) se jeví jako náhodný, ale při odšifrování lze zpětným chodem od bodu 10 dojít zpět až k bodu 1 a přitom kontrolovat jak hodnotu pHash, tak PS, i separátor 01 včetně jeho umístění. Redundance je dostatek – navíc je skryta a „zvnějšku" RSA (ze šifrového bloku) těžko dosažitelná. Uvědomme si totiž, že podstata Bleichenbacherova útoku [BL] těžila z multiplikativní vlastnosti RSA, která je symbolicky zapsána jako RSA(a * b) = RSA(a) * RSA(b). V maskovaném formátu však hašovací funkce (viz MGF) multiplikativní a aritmetické vztahy brutálně destruuje, takže „podlézání" operace RSA typu RSA(Hash(a)) = Hash(RSA(a)) a podobné triky už nehrozí.

Bezpečný formát také pro podpis

Na formát typu 01 (pro podpis) zatím útok nalezen nebyl, ale není důvodu, proč i zde nezavést kvalitní metodu. I když verze 2.0 PKCS#1 ponechala pro podpis ještě nemaskovaný formát, ve **verzi 2.1** se už maskování zavádí. Má

Postup maskování pro šifrování klíčů

Při doplňování zdrojových dat D o velikosti d oktetů do úplného bloku EB o k oktetech postupujeme v následujících krocích (viz též obr. 2):

1. Urči konstantu pHash = Hash(prázdný řetězec); pro SHA-1 je hLen = 20 oktetů.
2. Vytvoř řetězec PS obsahující k – d – 2 – 2*hLen nulových oktetů (délka doplňujícího řetězce je určena tak, aby doplňovala povinné položky do délky k oktetů).
3. Vypočti DB = pHash || PS || 01 || D.
4. Vytvoř řetězec seed obsahující hLen **náhodných** oktetů.
5. **První maska:** dbMask = MGF(seed, k – 1 – hLen).
6. Použij první masku: maskedDB = DB ⊕ dbMask.
7. **Druhá maska:** seedMask = MGF(maskedDB, hLen).
8. Použij druhou masku: maskedSeed = seed ⊕ seedMask.
9. Vypočti EM = maskedSeed || maskedDB.
10. Výstupní blok pro zašifrování algoritmem RSA je EB = 00 || EM.

označení *EMSA-PSS* (*Encoding Method for Signatures with Appendix – Probabilistic Signature Scheme*) a podíváme se i na něj.

Metoda EMSA-PSS je opět parametrizována volbou hašovací funkce (Hash) a volbou „soli“ (salt). Sůl je nějaký zvolený (konstantní nebo náhodný) řetězec o délce hLen. Když podepisujeme zprávu M, doplníme ji solí a teprve pak pořídíme digitální otisk Hash(salt || M). V dalším procesu (viz další rámeček) se použije MGF podobně jako u maskování klíčů. MGF také používá nějakou hašovací funkci a standard doporučuje použít tutéž funkci Hash, která byla použita k hašování zprávy M. Jedinou podporovanou volbou je SHA-1. Pokud se týká soli, standard se stanovením její hodnoty nezabývá, tj. může jí být konstanta nebo náhodný řetězec, ale před vlastním podpisem zprávy musí být samozřejmě už definována. Naopak při verifikaci podpisu známa být nutně nemusí, protože vyplyne z dekódování bloku EB.

Další normy pro podpis

Formátem dat pro digitální podpis se zabývají ještě standardy mezinárodní organizace pro normalizaci ISO, dále institut IEEE (skupina P1363) a americká národní normalizační organizace ANSI. Tyto standardy jsou navzájem provázané, ale bohužel základní dva z nich (ISO 9796 Part 1 a ISO 9796 Part 2) se přepracovávají. V minulém roce byly totiž objeveny účinné útoky také na jejich datové formáty. Popis těchto útoků a stav prací na jejich úpravách jsou značně komplikované (zabývali jsme se tím v srpnovém článku, viz infotipy).

Protože letos také došlo ke zrušení amerických exportních omezení, bude nejbližší doba poznamenána přechodem na silnou kryptografii a na opravené verze standardů u různých aplikací. Jak nám sdělil koordinátor standardů PKCS Burt Kaliski, předpokládá se, že současný draft PKCS#1 verze 2.1. vstoupí v platnost jako standard až v roce 2001, protože se vyvíjí paralelně se standardy IEEE P1363a a ISO 9796-2, avšak jeho definice bude stabilní už koncem tohoto roku. Opravy standardů všech tří vydavatelů (RSA, IEEE a ISO) by tak měly být v zásadě k dispozici ještě letos, a pak by se už mělo čekat jen na formální proces schvalování.

infotipy

Citované články z Chipu naleznete také na
▸ *www.decros.cz/Security_Division/Crypto_Research/archiv.htm*
pod mnemotechnickým označením *časopis-rok-měsíc-strana(od-do).ext:*

Nástroje pro digitální podpis:
Návrat šampiona, Chip 8/00, str. 40 – 42
Pojmy a definice pro PKCS#1:
Bude nás podepisovat RSA?, Chip 9/00, str. 50 – 52
Úvod k hašovacím funkcím a popis SHA-1:
Výživná haše, Chip 3/99, str. 40 – 43
Standardy PKCS:
▸ *http://www.rsasecurity.com/rsalabs/pkcs/*
PKCS#1 ver. 2.0 jako RFC:
„PKCS#1: RSA Cryptography Specifications“, Version 2.0, October 1998, RFC 2437
Útok na formát PKCS#1 ver. 1.5:
[BL] Bleichenbacher, D.: „Chosen Ciphertext Attacks against Protocols Based on the RSA Encryption Standard PKCS#1“, Crypto '98, Springer-Verlag, 1998, str. 1 – 12
Protokol SSL:
The SSL protocol, version 3.02, November 18, 1996, internet draft

Shrnutí

Stále ještě hojně používaný formát dat podle PKCS#1 ver. 1.5 pro přenos klíčů pomocí RSA není zcela bezpečný; nápravu představuje až verze 2.0. Také u formátu pro elektronický podpis se přechází na bezpečnější maskovaný formát, který vstoupí v platnost s verzí 2.1. Ten je velmi robustní, a proto lze očekávat jeho platnost po velmi dlouhou dobu. Společně s algoritmem AES (viz článek na jiném místě) tak budeme mít silné nástroje jak pro šifrování dat, tak i pro elektronický podpis.

Vlastimil Klíma
(v.klima@decros.cz)

Postup maskování pro podpis

1. Hašuj zprávu společně se solí: H = Hash(salt || M).
2. Vytvoř řetězec PS obsahující k – 1 – 2*hLen ≥ 0 nulových oktetů (délka doplňujícího řetězce je určena tak, aby doplňovala povinné položky do délky k oktetů).
3. Vytvoř datový blok DB = salt || PS.
4. **Vytvoř masku:** dbMask = MGF(seed, k – 1 – hLen).
5. Použij první masku: maskedDB = DB ⊕ dbMask.
6. Vypočti EM = H || maskedDB.
7. Výstupní blok pro aplikaci algoritmu RSA je EB = 00 || EM.