UNIVERZA V LJUBLJANI
FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO

NAPADI NA GENERATOR NAKLJUČNIH ŠTEVIL

seminar za predmet "Kriptografija in računalniška varnost"

Avtor:
Boštjan Dolenc

Ljubljana, junij 2004

1. Uvod

Naključna števila se v kriptografiji uporabljajo, kadar potrebuje nek algoritem ali protokol naključne oz. bolj natančno neuganjlive podatke. Najbolj pogosto so to ključi šifrirnih algoritmov. Pri teh sloni celotna varnost algoritma na tajnosti ključa ter na tem, da je računsko prezahtevno preizkustiti celoten prostor ključev. Če bi napadalec lahko ključ uganil, bi tako algoritem izgubil celotno varnost. Poleg tega se naključna števila uporavljajo tudi kot nonce-i[1], npr. za "challenge" v challenge-response shemah ali pa za inicializacijske vektorje. Tudi tu velja podoben razmislek glede neuganljivosti.

Računalniki so kot deterministične naprave sami po sebi nezmožni generirati naključna števila. Rezultati, ki jih računalnik da, so namreč natančno določeni z algoritmom in vhodnimi parametri. Zaradi tega moramo vedno uporabiti nek vir prave naključnosti. Običajno gre za kakšen fizikalni proces ali pa akcije uporabnika. Žal so ti viri pogosto dokaj "revni", delujejo počasi in dajo na sekundo manj naključnih bitov, kot jih lahko kriptografski algoritem porabi. Na srečo v kriptografiji potrebujemo neuganjljiva števila in ne "čisto naključna", zato sklenemo kompromis: uporabili bomo algoritem oz. generator, ki bo "povečal" kapaciteto oz. hitrost naključnega vira tako, da bo iz vira vzel naključnost in jo "razredčil"[2] v nekaj, kar bo še (čim bolj – tudi tu bomo nekoliko popustili) neuganljivo. Ker taka števila niso več naključna, jim pravimo psevdonaključna.

Na prvi pogled se zdi, da bi se z uporabo vira "prave" naključnosti popolnoma zavarovali pred napadi na generator. Vedar bi v določenih (a sicer malo verjetnih) primerih napadalec lahko meril ali vplival na vir naključnosti. Torej si s tem ne bi mogli zagotoviti popolne varnosti in omenjeni kompromis ni tako boleč. Poleg tega pa večina v praksi udejanjenih napadov ne temelji tem, da je bil uporabljen algoritem namesto pravega vira naključnosti, ali pa na spregledanih pomanjklivostih priporočenih generatorjev. Prav nasprotno – napade omogoči pretežno uporaba kriptografsko nesprejemljivih generatorjev ter slaba ocena, koliko naključnosti sploh dobimo iz vira.

Namen članka je predstaviti vlogo generatorjev naključnih števil v kriptografiji. Pri tem se bomo osredotočili na to, kako lahko napačna izbira ali uporaba generatorja oslabi kriptosistem. Zato bo posebna pozornost posvečena kriptografksim napadom ter drugim napakam.

Članek bo najprej podal nekaj teoretskega ogrodja za ocenjevanje naključnosti, ki nam bo služila pri oceni različnih algoritmov in napadov nanje. Predstavil bo tudi klasifikacijo napadov na generatorje naključnih števil. Nato bodo predstavljeni klasični generatorji naključnosti ter nekaj kriptografsko varnih generatorjev. Glavni del članka bo posvečen napadom na generatorje naključnih števil. Poleg kriptografskih napadov bodo prestavljeni tudi napačni načini uporabe generatorjev in virov naključnosti. Kot se bo izkazalo, v praksi prevladujejo slednji.

[1] Number used Only oNCe – število, ki bo znotraj serije korakov protokola uporabljeno le enkrat. Kadar ga generira PRNG, tega seveda ne moremo garantirati, lahko pa zagotovimo visoko verjetnost, da se ne bo ponovilo.

[2] Vsaka podobnost z mešanjem vina z vodo je zgolj namerna...

2. Naključnost, entropija in viri entropije

Kot smo omenili, je bistvena lastnost naključnih števil v kriptografiji njihova neuganjljivost. To "neuganljivost" oz. nedoločenost bi radi merili oz. vsaj ocenili, za kar potrebujemo neko številsko izrazljivo mero. Zanjo se uporablja kar entropija, kot jo poznamo iz teorije informacij. Formula zanjo je:

$$H(X) = -\sum_i p_i \log_2 p_i ,$$

kjer je p_i je verjetnost, da je vir X v stanju i. Podrobneje o entropiji v [10].

Vendar pa je treba poudariti, da moramo za p_i vzeti verjetnost, kot jo bi izmeril napadalec. Slednje lahko zahteva kar zahteven premislek. Morda lahko napadalec v nekem trenutku izve trenutno stanja vira in bo tako v določenih časovnih intervalih zanj entropija enaka 0. Še huje: morda lahko nanj vpliva tako, da spremeni verjetnosti posameznih stanj. Slednje je sicer malo verjetno pri klasičnih računalniških sistemih tipa odjemalec-strežnik, kjer je strežnih varovan in so odjemalci razpršeni ter težko dostopni napadalcu (kadar pa so dostopni, pa ima napadalec na voljo bolj enostavne tarče kot vir entropije). Dosti bolj verjetno pa to postane pri manjših, avtonomnih sistemih, kot so npr. pametne kartice.

Še bolj pogosta napaka pa je, da naključnost že na začetku ocenimo narobe. Za ilustracijo vzemimo za vir naključnosti tipkovnico. Zelo zgrešeno bi bilo za naključno število vzeti kar zadnji vnešeni znak in tako oceniti entropijo s 7 (za uporabnike, ki pišejo angleščino) oz. 8 biti (za tiste uporabnike, ki uporabljajo tudi "tuje" črke). Za začetek bi morali vzeti namesto znaka kodo pritisnjene tipke. Kljub temu pa bo porazdelitev zelo neenakomerna – posamezne črke nastopajo različno pogosto že v jezikih. Še bolj pride ta neenakomernost do izraza pri tipični uporabi na računalniku, kjer prevladujejo ukazi (ukazi lupine, npr. "ls –ltr" ali pa za programe npr. "Ctrl-S"), ki so še dosti bolj predvidljivi kot besedilo naravnega jezika. Poleg tega se moramo vprašati, kaj narediti, če se vir izprazni – če uporabnik dalj časa ni ničesar vtipkal ali pa če potrebujemo npr. 128 bitov za simetrični ključ. Če v tem primeru zahtevamo od uporabnika, da nekaj natipka, bo rezultat običajno izredno slab. Pogosti bomo dobili predvidljiva zaporedja kot so "asdfas" ali "jklkj".

Kot vir entropije se poleg tipkovnice pogosto uporabljajo tudi druge lastnosti naprav v računalniku, kot so premiki miške, spodnji biti interne ure računalnika, časi dostopov do podatkov na disku, akustični šum na vhodu zvočne kartice, vsebina video pomnilnika in zakasnitev prometa v omrežju. Glede na to, da je teoretično mogoče vplivati nanje, raje izberemo fizikalne pojave, na katere je izredno težko vplivati. Praktično dostopen je npr. generator naključnih števil na procesorjih iz družine x86, ki ima za vir naključnosit majhna nihanja v temperaturi procesorja.

Oceno entropije in vire je tako najbolje prepustiti specializiranim knjižnicam ali servisom operacijskega sistema (seveda le po natančnem pregledu varnostnih vidikov). Očitno je tudi, da je bolje uporabiti več virov entropije in entropijo "zmešati skupaj". Pri tem moramo paziti na morebitno odvisnost med viri in entropijo glede na to primerno zmanjšati.

Kljub temu pa se pogosto zgodi, da je računalnik enostavno ne more zagotoviti dovolj naključnih bitov na sekundo. V tem primeru moramo te bite "razširiti" na več še vedno neuganljivih bitov s pomočjo algoritma.

3. Vrste napadov na PRNG

Preden si pogledamo posamezne generatorje psevdonaključnih števil (angleško pseudo-random number generator oz. PRNG), moramo pogledati, na kakšne načine sploh lahko napademo PRNG oz. kakšne vrste defektov ima lahko PRNG. Osnovno vodilo je, da je PRNG dober, če njegovega izhoda ne moremo razločiti od toka naključnih bitov. To pomeni, da za katerikoli par zaporedij, kjer je eno zaporedje izhod PRNG, drugo pa popolnoma naključno, ne moremo ugotoviti izhoda PRNG z verjetnostjo, različno od 0.5. V praksi pa običajno izvedemo napad tako, da napovemo del še neznanega izhoda PRNG - običajno nek del iz prihodnosti, lahko pa tudi iz preteklosti.

Sledeča klasifikacija je povzeta po [1].

3.1. Direktni kriptanalitični napad

Tak napad izvedemo, kadar lahko razločimo izhod PRNG od naključnega le s poznavanjem algoritma PRNG in nekega zaporedja izhodov algoritma, ne da bi poznali vhode (bite iz virov entropije) ali predhodna stanja PRNG (ki so sicer določena z vsemi predhodnimi vhodi). Tovrstni napad je pasiven, saj napadalec z njim ne vpliva na delovanje PRNG.

Obstoj takega napada pa napadalcem očitno ne pomaga, če ne morejo dobiti izhoda PRNG. Tak primer je denimo generiranje tajnih ključev za simetrično šifro, ki se jih izmenja po varnem kanalu.

3.2. Napad preko vhodov

Tak napad izvedemo, če lahko z manipulacijo vhodov oz. virov entropije razločimo prihodnje izhode PRNG. Tovrstni napad je aktiven.

Nekoliko podobno kot napade na simetrično šifro lahko ta napad nadalje razdelimo na napade z znanim vhodom, s ponovitvijo vhoda ter z izbranim vhodom. Napad z znani vhodom pogosto omogoči napačno ocenjena entropija vira, zaradi česar lahko del vhoda napovemo. Poleg tega lahko včasih napadalec prebere stanje vira, npr. če za vir uporabimo latenco omrežnih paketov. Napad z izbranim vhodom je praktičen predvsem pri pametnih karticah ter kadar je del računalnika ali pa virov entropije pod nadzorom napadalca. Napad s ponovljenim vhodom je slednjemu podoben, le da je za napadalca malo manj zahteven.

3.3. Razširitev kompromitiranega stanja

Tak napad izvedemo, če lahko uporabimo znanje o internem stanju PRNG za razločitev izhodov. Vendar postavimo tu omejitev: napad uspe le, če lahko razločimo (vsaj en) izhod PRNG iz preteklosti ali pa (vsaj en) izhod iz prihodnosti, vendar pa mora PRNG pred tem obdelati (vsaj en) vhod, ki ga napadalec ne pozna. Noben algoritem namreč ne more zagotoviti nenapovedljivih izhodov v časovnem intervalu med prvim napadalcu znanim internim stanjem in prvim naslednjim za napadalca neznanim vhodom. Vzrok je seveda to, da so algoritmi deterministični.

Med tovrstnimi napadi nadalje ločimo:

- *napad s sestopanjem* (kjer s poznavanjem stanja izračunamo predhodnje izhode)

- *popolno razširitev stanja* (kjer lahko s poznavanjem stanja v nekem trenutku delno ali v celoti ugotovimo vsa prihodnja in pretekla stanja)

- *iterativni napad z ugibanjem* (kjer s poznavanjem stanja v trenutku t izračunamo stanje v trenutku $t + \varepsilon$, kjer pa v intervalu $[t, t + \varepsilon]$ napadalec vhodov ne pozna, jih pa lahko ugane)

- *meet-in-the-middle napad* (kjer s poznavanjem dveh stanj v različnih trenutkih izračunamo ostala stanja na njunem časovnem intervalu).

Na prvi pogled se zdi, da je razširitev kompromitiranega stanja v praksi težko izvedljiva. Vendar pa so PRNG na začetku delovanja pogosto v uganljivem stanju zaradi premalo začetne entropije. Poleg tega pa ne moremo izključiti vdorov v računalniški sistem oz. občasne dostope napadalca do mest, kjer je shranjeno stanje PRNG.

3.4. Testni naključnosti

Kot uporabnike PRNG nas bolj kot napadi na PRNG zanima način, kako bi lahko preverili, ali je PRNG varen pred napadi. Žal podobno kot drugje v kriptografiji ne moremo (oz. še ne znamo) dokazati odpornost pred napadi. Izjema je generator BBS (glej razdelek 5.1), za katerega je dokazana varnost pred direktnim kriptanalitičnim napadom ob predpostavki, da je faktorizacija celih števil težka.

Na voljo pa imamo razne statistične teste, s katerimi lahko preverimo, v kolikšni meri je izhod generatorja naključen. Zaradi statističe narave zahtevajo testi veliko količino izhodnih podatkov in lahko le dokažejo regularnost ("nenaključnost") v izhodih, naključnosti pa ne morejo zagotovo potrditi. Tako so ti testi le potreben, ne pa zadosten pogoj za kriptografsko uporabo. Testi sicer niso strogo potreben pogoj – včasih lahko to pomankljivost popravimo tako, da izhod PRNG pošljemo skozi enosmerno funkcijo (glej [1]).

Med statistične teste spadajo preprosti testi frekvence enic in ničel v izhodu ter enakost pogostosti n-teric. Boljši testi so npr. hi-kvadrat, test Kolmogorov-Smirnov ter spektralna analiza (s katero se zazna pozneje omenjeni efekt Marsaglije). Poleg tega pa je bilo izdelanih tudi mnogo "strogih" (stringent) testov kot alternativa do tedaj razširjenim "manj strogim", prej omenjenim testom. Le-ti pri marsikaterem PRNG niso zaznali problemov. "Strogi" testi temeljijo na konkretnih primerih statističnih simulacij (npr. računanje π po metodi Monte Carlo), kjer se primerja rezultat simulacije z uporabo testiranega PRNG s predvidenim rezultatom.

Zaradi obilice testov so se izoblikovali standardni nabori oz. "baterije" testov. Najbolj uveljavljena sta DIEHARD Georga Marsaglie ter NIST Statistical Test Suite. Ti nabori združujejo več posameznih testov, zanje pa obstajajo tudi programski paketi za enostavno testiranje.

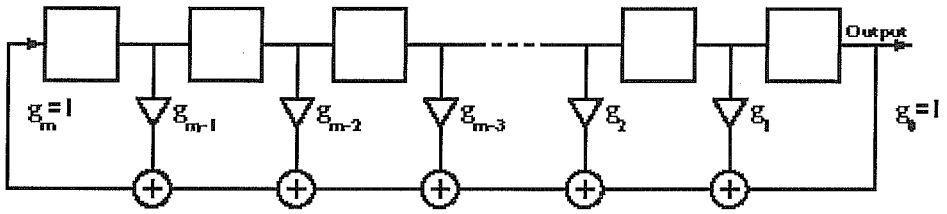
4. Klasični, ne-varni PRNG

Žal se ne moremo izogniti opisu klasičnih PRNG. Kot bo prikazano, so ti povsem neprimerni za uporabo v kriptografiji, vendar se kljub temu uporabljajo v praksi, predvsem v programski opremi. Glavni vzrok je predvsem to, da se programerji ne zavedajo njihovih slabosti. Poleg tega se ti algoritmi pojavljajo v vseh knjižnicah razširjenih programskih jezikov, ne da bi bil v njih na voljo tudi kriptografsko varen algoritem.

Glavna problema omenjenih algoritmov so naključna števila "slabe kvalitete" (razna statistična odstopanja) ter to, da se entropija vnese v PRNG le ob inicializaciji, pozneje pa ne. Prvi vodi do direktnih kriptanalitičnih napadov, drugi pa do popolne razširitve kompromitiranega stanja. Medtem ko je prvi problem moteč tudi pri "klasični" uporabi PRNG npr. za statistične simulacije, pa je drugi problem moteč le pri kriptografiji. Poudariti je treba, da so napadi prvega tipa pomankljivosti konkretnih algoritmov, ki se ji načelno da izogniti. Napade drugega tipa pa lahko preprečimo le z stalnim "primešovanjem" entropije, kar pa klasični PRNG sploh ne podpirajo.

4.1. Linearni pomični registri

Prva skupina pogosto uporabljenih, klasičnih PRNG so linearni pomični registri (Linear Feedback Shift Register oz. LFSR). Bistvo te družine PRNG lahko razberemo iz njihove splošne sheme:



Slika 1: splošna shema LFSR

LFSR lahko tudi zapišemo z rekurzivno formulo $y_n = \sum_{i=1}^{m} g_i y_{n-i}$. Ta izraz imenujemo tudi karakteristični polinom LFSR. Iz te formule sledi, da je m izhodov natančno določenih z predhodnimi m izhodi ter koeficienti g_i. Ta pogoj lahko zapišemo kot sistem m enačb, kjer so spremenlivke y_1 do y_{2m} ter g_1 do g_i. Če tedaj poznamo $2m$ izhodnih bitov, lahko izračunamo koeficiente in s tem določimo vse prihodnje izhode LFSR. Bolj napreden napad lahko izvedemo z uporabo algoritma Berlekamp-Massey (podan v sedmem poglavju [12]). Le-ta za neko zaporedje izhodov najde najkrajši LFSR, ki generira to zaporedje.

LFSR so razširjeni, kjer je potrebna hitra hardverska implementacija PRNG. Softverske implementacije so manj pogoste.

4.2. Kongruenčni generatorji

Ta družina generatorjev se najpogosteje uporablja v programskih knjižnicah zaradi razumljive sheme in enostavne implementacije. Polega tega so bili priporočeni v Knuthovi seriji knjig *The Art of Computer Programming*, ki je eno izmed najvplivnejših del na področju računalniških algoritmov.

Iz te družine so najpogostejši linearni kongruenčni generatorji (Linear Congruential Generator oz. LCG). Podamo jih z rekurzivno funkcijo $y(n+1) = A \cdot y(n) + B \bmod M$.

Očitno je, da perioda generatorja ne more biti večja od modula *M*, lahko pa je manjša. Večja pomankljvost teh generatorjev pa je, da so n-terice izhodov v n-razsežnem prostoru porazdeljene predvsem po hiperravninah, kar imenujemo tudi Marsaglijev efekt. Zaradi tega efekta so LCG neprimerni tudi pri nekriptografski uporabi npr. za Monte Carlo simulacije in računalniško grafiko.

Podobno kot pri LFSR lahko z dovolj izhodi tvorimo sistem enačb. Te enačbe lahko predstavimo kot množico vektorjev in iz njih tvorimo mrežo (lattice). Rešitev poiščemo s pomočjo algoritma LLL (imenovan po A. K. Lenstra, H. W. Lenstra, and L. Lova'sz). Algoritem je podan v [11].

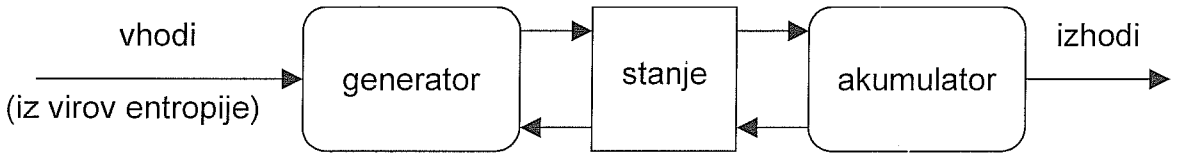
4.3. Mersenne Twister

Kot poseben primer kongruenčnih generatorjev si na hitro poglejmo lastnosti trenutno (izven kriptografije) zelo popularnega generatora "Mersenne Twister". Njegove prednosti so izredno dolga perioda, katere dolžina je $2^{19937} - 1$ ter dokazana odsotnost efekta Marsaglija do dimenzije 623. Poleg tega obstajajo zanj hitre programske implementacije, sam algoritem pa ni obremenjen s patenti. Kot tak je algoritem primeren za klasično uporabo. Kljub temo pa je za kriptografijo neuporaben.

Razlog za to je, da se interno stanje algoritma (kot pri vseh klasičnih PRNG) nastavi le ob inicializaciji PRNG. Tako je za napadalca dovolj, da prejme 19937 bitov izhoda (kolikor je veliko interno stanje), iz katerih lahko nato izračuna interno stanje. Sam algoritem je namreč le kombinacija dveh LSFR, samo računanje internega stanja iz izhodov pa je računsko nezahtevno. Shema omenjenih LFSR in postopek sta podrobneje razložena v [8].

5. Kriptografsko varni PRNG

Bistvena razlika med klasičnim PRNG in kriptografsko varnim PRNG (Cryptographicaly Secure PRNG oz. CSPRNG) je to, da slednji vsake toliko časa "primeša" nekaj naključnih bitov iz virov naključnosti v svoje stanje, medtem ko klasičen PRNG vnese naključnost v stanje le ob inicializaciji. To "primešavanje" naključnosti je namreč edina obramba pred razširitvijo kompromitiranega stanja. Brez tega bi bilo trenutno stanje PRNG vedno odvisno le od začetnega stanja, saj je PRNG deterministične algoritem.



Slika 2: shema (potencialno) kriptografsko varnega PRNG

Primešavanje entropije je potrebno (poleg branjenja pred razširitvijo kompromitiranega stanja) tudi za to, da entropija stanja generatorja raste s količino izhodnih bitov. V nasprotem primeru bi lahko napadalec po dovolj veliko izhodih izračunal interno stanje.

Seveda za (varen) CSPRNG zahtevamo, da je čim bolj odporen tudi na ostali dve vrsti napadov. Za praktično uporabo pa je zelo zaželjena tudi hitra implementacija.

5.1. Blum-Blum-Shub (BBS)

BBS je trenutno kriptografsko najbolj varen algoritem, saj zanj v nasprotju z vsemi ostalimi obstaja dokaz nenapovedljivosti izhodov iz poljubno dolgega poznanega izhoda. Dokaz sicer temelji na predpostavki težavnost faktorizacije celih števil.

Osnovna shema algoritima je $y(n+1) = y(n)^2 \bmod M$, kjer velja $M = p \cdot q, p \equiv q \equiv 3 \bmod 4$. p in q sta veliki praštevili, tako da je njun produkt prevelik za faktorizacijo. y(n) je tu interno stanje, izhod generatorja pa je lahko $\lfloor \log_2 M \rfloor$ bitov zadnjega stanja ali manj.

Algoritem je ob pogoju, da je faktorizacija M računsko neizvedljiva, dokazano varen pred direktnim kriptanalitičnim napadom. Žal pa so implementacije algoritma v primerjavi z drugimi PRNG počasne. Poleg tega algoritem ni varen pred razširitvijo kompromitiranega stanja. Še več, stanje čez N korakov bo $y(k)^{2^{(k+N) \bmod ((p-1)(q-1))}}$, če je $y(k)$ neko predhodno stanje v koraku k.

5.2. Fortuna

Fortuna je algoritem Bruca Schneierja in Neilsa Fergusona. Načrtovan je bil z namenom, da bo odporen na vse tri oblike napadov, navedenih v poglavju 3, ter da se ga bo dalo učinkovito in enostavno implementirati.

Algoritem je razdeljen na inicializacijo semena, generiranje naključnega izhoda ter na akumulacijo entropije. Poleg tega je predvidena datoteka s semenom, ki se uporablja kot poseben akumulator entropije za uporabo ob reštartu računalnika. Tedaj bo interni akumular entropije prazen, prav tako pa tudi viri entropije ne bodo vsebovali zadosti entropije in bi moral generator čakati nanje.

Inicializacija semena le nastavi 256 bitno stanje na začetno vrednost oz. doda novo seme k staremu z uporabo zgoščevalne funkcije SHA-256. Poleg tega nastavi 128 bitni števec generiranih blokov izhoda na 0. Algoritem torej predvideva največ 2^{128} generiranih blokov izhoda brez ponastavljanja semena.

Jedro Fortune je funkcija GENERATEBLOCKS, ki generira naključno vrednost. Funkcija temelji na bločni šifrirni funkciji, za katero avtorja priporočata AES. Izhod funkcije je n blokov $E(K, C)$, kjer je K seme ter C števec in E omenjena bločna šifrirna funkcija. Izhod algoritma je zahtevano število bitov funkcije GENERATEBLOCKS. Poleg tega algoritem po vsakem generiranju izhoda generira dodatna 2 bloka izhoda, s čimer nastavi seme na novo vrednost. S tem prepiše staro seme in onemogoči, da bi staro seme "izteklo" v naslednje izhode. Hitrostno je generiranje naključne vrednosti omejeno predvsem s hitrostjo bločne šifrirne funkcije – eno šifriranje za en blok izhoda. Blok je sicer lahko poljubne velikosti (pač glede na izbrano bločno šifro), običajno pa bo velik 128 bitov zaradi izbire AES-128. Pomembno je tudi, da uporabnik zahteva čim več naključnih bitov naenkrat, da zmanjša količino generiranja zaradi prehoda na novo seme.

Akumulacija entropije poskrbi za to, da lahko brez prilagoditev algoritma uporabimo več virov entropije. Algoritem zbira entropijo v 32 skladiščih. Izhodi vseh virov se porazdeljujejo zaporedno po skladiščih. Seme generatorja se osveži, ko skladišče P_0 preseže hevristično postavljeno mejo in pa periodično vsake 100 ms. Ob osvežitvi semena uporabi skladišče P_i, če je 2^i deljivo z zaporedno številko osvežitve. To povzroči, da se skladišče P_0 uporabi ob vsaki osvežitvi, P_1 v vsaki drugi, P_2 v vsaki četrti itd. Namen te izbire je preprečiti napade, ko napadalec pozna večino skladišč ali pa forsira generiranje naključnosti iz posameznih virov v upanju, da bodo večinoma uporabljeni njihovi izhodi. Taka menjava zagotavlja, da bo po osvežitvi v semenu dovolj nepoznane naključnosti že, če obstaja le eno skladišče, ki ga napadalec ne more napovedati. Frekventni viri bodo namreč najbolj zastopani v P_0, ker se bo ta najpogosteje praznil. Ostala skladišča pa bodo napadalcu bolj neznana (eksponentno glede na indeks skladišča). Omeniti je tudi treba, da algoritem pred uporabo skladišča njegovo vrednost pošlje skozi zgoščevalno funkcijo SHA-256.

Komplicirana akumulacija virov je posledica slabih izkušenj avtorjev s predhodnim algoritmom Yarrow. Pri njem je bilo potrebno entropijo virov oceniti, kar se običajno da le hevristično in zahteva podrobno poznavanje implementacije ter procesov, iz katerih vir zajema entropijo.

Avtorja ocenjujeta hitrost generatorja na 20 urinih ciklov na bajt izhoda, kadar so zahtevani izhodni bloki veliki. Glede na to, da je večina računskega časa porabljenega v bločna šifri, in da so efektivne implementacije le-te na voljo v skoraj vseh računalniških okoljih, ni težko narediti hitre implementacije Fortune. Kompleksnost algoritma pa je v primerjavi s klasičnimi PRNG ter BBS velika, kar otežuje neodvisno kriptografsko analizo ter preverjenje pravilnosti implementacije.

6. Napadi na algoritme generatorjev

6.1. ANSI X9.17

ANSI X9.17 je standard, priporočen za generiranje DES ključev in inicializacijskih vektorjev. Uporablja pa se tudi za generiranje ključev pri drugih algoritmih. Za osnovo uporablja 3DES. Algoritem je tak:

$$T_i = E_K(time)$$
$$y_i = E_K(T_i \oplus s_i)$$
$$s_{i+1} = E_K(T_i \oplus y_i)$$

Tu je *time* trenutni čas (ob izvajanju prvega koraka algoritma), funkcija E_k šifriranje po 3DES, s_i je interno stanje v koraku i (imenovano tudi seme generatorja), y_i pa je izhod algoritma v koraku i. y_i, s_i in T_i so 64 bitne spremenljivke. K je 3DES ključ, ki se nastavi na naključno vrednost ob inicializaciji generatorja in se ne sme ponovno uporabiti ob reinicializaciji generatorja. Torej gre za del tajnega stanja generatorja, na katerega vhodi ne vplivajo.

Prva, praktično manj pomembna ranljivost algoritma je, da pride do trka izhodnih vrednostih ob ponavljanju vhoda oz. zamrznjenem vhodu v 2^{32} korakih, namesto v pričakovanih (cca.) 2^{63} korakih.

Algoritem je ranljiv za razširitev kompromitiranega stanja. Najbolj problematično je to, da lahko napadalec, ki izve K, efektivno izračuna vse nadaljne izhode. Ker se K ne spremeni, je ta del stanja generatorja znan za celoten čas delovanja, tako za preteklost kot za prihodnost. Po drugi strani pa bo posamezen vhod (*time*) zaradi tega vplival le na s_i oz. na le 64 bitov internega stanja. Poleg tega velja, da je stanje generatorja funkcija K, prejšnjega izhoda s_{i-1} ter prejšnjega šifriranega vhoda T_{i-1}. Napadalec lahko tako hitro ugane s_i, saj zagotovo pozna s_{i-1}, poznavanje K predpostavimo zaradi vrste napada, T pa ima pogosto malo entropije, saj gre za čas generiranja ključa. Npr. napadalec lahko pogosto izve čas generiranja sporočila na sekundo natančno iz časovnih zapisov v spremljajočem čistopisu izhoda, kot so zaglavlja dokumentov, ali pa vzame zanj čas prejema sporočila. Tako ocenimo entropijo T_i na 10 bitov, glede na to, da je pogosta ločljivost interne ure 1 milisekunda.

Tako lahko napadalec izračuna s_i s poznavanjem K, y_i in y_{i+1} z "meet-in-the-middle" napadom. Uporabi namreč to lastnost:

$$s_{i+1} = D_K(y_{i+1}) \oplus T_{i+1}$$
$$s_{i+1} = E_K(y_i \oplus T_i)$$

Nato zgradi dva seznama vrednosti s_{i+1}, prvega po prvi in drugega po drugi enačbi. Pravi s_{i+1} je tisti, ki nastopa v obeh seznamih. Za lažje iskanje tega preseka naj bosta seznama urejena. Zaradi ocene entropije T_i bo vsak seznam dolg 2^{10} elementov, torej je celotna zahtevnost $O(2^{11})$ 3DES šifriranj/dešifriranj. To osnovno idejo se da uporabiti tudi, če zve napadalec y_i in y_{i+k}. V tem primeru je zahtevnost $O(2^{k*10+1})$.

Drugi problem je predpisana uporaba ure kot vira entropije. Pogosto se namreč zgodi, da nek algoritem potrebuje več kot 64 bitov entropije naenkrat. V tem primeru bo PRNG zaporedoma klican v zelo kratkih časovnih razmakih. Ker ANSI X9.17 zahteva en zajem entropije na 64 bitov izhoda, je zelo verjetno, da bo interval med klicoma krajši kot pa ločljivost ure. To še dodatno zmanjša entropijo vira in omogoči napad tudi v primeru, ko napadalec vidi le vsak *k*-ti izhod.

6.2. RC4

RC4 je sicer tokovna šifra, vendar je primer uporabe, za katerega je bil napad izveden, podoben uporabi PRNG. Tudi sicer so si tokovne šifre in PRNGji po sami strukturi zelo podobni. Tokovna šifra vsebuje stanje, ki se na začetku inicializira. Nato po algoritmu v vsakem koraku generira 1 bit (oz. v nekater različicah 1 bajt ali več) izhoda, ki se potem XORa z čistopisom. Tako dobimo tajnopis. Prejemnik na drugi strani spet požene tokovno šifro z istim začetnim stanjem (stanje očitno igra vlogo tajnega ključa) in izhode XORa s tajnopisom, da dobi čistopis. Očitno bi lahko kot algoritem vzeli tudi PRNG, seveda le, če bi pošiljali vanj deterministične vhode namesto naključnih.

Leta 2001 so Scott Fluhrer, Itsik Mantin, and Adi Shamir odkrili veliko družino šibkih ključev za RC4 ([9]). Za te ključe velja, da lahko s poznavanjem nekaj bitov ključa z neko verjetnostjo napovemo veliko bitov stanja in izhoda. Vzrok za to je, da je zaradi Key Scheduling Algorithm (algoritem v RC4, ki iz ključa naredi začetno stanje šifre) veliko bitov začetnega stanja odvisnih od le nekaj bitov ključa. Ključi igrajo v RC4 enako vlogo kot začetno seme v PRGN. Poleg tega Pseudo Random Generation Algorithm (algoritem RC4, ki iz stanja generira zaporedje izhodnih bitov), preslika te bite začetnega stanja na začetek izhodnega zaporedja. Tako je prvi bajt izhoda enak $S[S[1]+S[S[1]]$, kjer je S začetno stanje RC4, zapisano kot enodimenzionalno polje bajtov.

Pri praktični uporabi se RC4 uporablja tako, da je del kluča tajen in si ga pošiljatelj in prejemnik predhodno izmenjata, del (inicializacijski vektor oz. IV) pa je javen in je kot čistopis pripet kriptiranemu sporočilu. V praksi to pri npr. protokolu WEP pomeni, da je možno iz cca. 4 miljonov paketov omrežnega prometa ugotoviti tajni ključ.

Ta napad je za PRNG relevanten v primeru, ko bi radi uporabili nek PRNG kot tokovno šifro. Napad je uporaben za protokole za tokovne šifre, ki po inicializaciji ne primešavajo entropije v stanje generatorja. Protokoli običajno tega ne počnejo zaradi enostavnosti izmenjave IV. Omenjena nevarnost, da bi že iz delnega poznavanja stanja lahko napovedali del izhoda, za PRNG ni pomembna, saj privzamemo, da če napadalec pozna del stanja, da tedaj zagotovo pozna celotno stanje. Zaradi tega PRNG niso analizirani glede tovrstnega napada, saj je le-ta v klasični uporabi PRNG neuporaben. To analizo moramo zato nujno opraviti sami, če hočemo PRNG uporabiti kot gradnik za tokovno šifro. Zaradi tega uporaba PRNG kot tokovne šifre ni tako enostavna, kot je morda videti na prvi pogled.

7. Napake pri uporabi naključnih števil

Med napake prištevamo vse ostale napade, ki ne izkoriščajo pomankljivosti algoritmov, temveč nepravilno uporabo le-teh. Nepravilno uporabo bomo definirali kot uporabo, ki ni v skladu s trenutno znano prakso. Sem spadajo tudi napačne (previsoke) ocene entropije vira. Ta razmejitev sicer ni širše uveljavljena, vendar pa se nam zdi pomembno ločiti napade zaradi površnosti implementatorjev od ostalih napadov. Poleg tega je analiza teh napadov matematično manj zahtevna.

7.1. SSL v Netscape 1.1

Napad je bil objavljen leta 1995. To je bil do sedaj javno najodmevnejši napad na generator naključnih števil. Zanimanje je bilo tako veliko, ker je bila takrat Netscape 1.1 dominanten odjemalec za e-poslovanje. Varnost e-poslovanja pa je skoraj v celoti temeljila na varnosti povezav SSL. Ker je napad omogočil prisluškovanje povezavam SSL vsakomur, ki bi prestregel omrežni promet med Netscape in strežnikom, je bila to resna nevarnost takrat še novemu in neuveljavnjenemu e-poslovanju preko spleta.

Bistvo napada je v tem, da namesto ugibanja sejenega ključa ugibamo seme (začetno stanje) PRNG-ja, ki kreira ta ključ. Če seme uganemo, lahko direktno izračunamo sejni ključ, saj je preostanek vhodov v algoritem znan. Poudariti je treba, da so pri Netscape uporabili svoj algoritem za generiranje naključnih števil in ga niso javno objavili. Kljub temu so zunanji raziskovalci algoritem odkrili z dekompilacijo programa oz. vzvratnim inženiringom.

Celoten algoritem generiranja ključa je potekal tako: ponastavimo PRNG z novim semenom, iz njega vzamemo 40 do 128 bitov (odvisno od zahtevene dolžine ključa) in te bite "zmešamo" z uporabo zgostitvene funkcije MD5. Potrebno pa je še ugotoviti, kolikšna je entropija semena. Sledeča analiza entropije semena sicer velja le za najboljši premer, saj na nekaterih platformah vsi viri naključnosti niso bili dostopni. V tem primeru je bil algoritem prilagojen tako, da jih je ignoriral. Zato je imelo seme še manj entropije in je bil napad še lažji.

Seme PRNG je n-terica (*seconds*, *microseconds*, *pid*, *ppid*). Na prvi pogled je seme veliko 82 bitov - 32 bitov za *seconds* (sekunde od nekega datuma, običajno od polnoči 1 januarja 1970), $\log_2 10^6 \cong 20$ bitov za *microseconds*, 15 bitov za *pid* in 15 bitov za *ppid* (številka procesa in starševskega procesa). Ker pa se v semenu uporabi vrednost *pid* + (*ppid* << 12)[3], dobimo iz *pid* in *ppid* le 27 bitov entropije namesto 30. Skupaj imamo torej 79 bitov entropije. Že to pa je manj kot najdaljša podprta dolžina sejnega ključa, ki je 128 bitov. Kljub temu pa bi tedaj tudi ta dolžina še zagotavljala nekaj varnosti, saj je tedaj npr. iskanje 40 bitnega SSL ključa trajalo 30 ur z uporabo vseh računalnikov večjega omrežja oz. cca. 1 CPU leto na delovni postaji HP 712/80.

Na žalost ima za napadalca *seconds* 0 bitov entropije. Čas pošiljanja je za napadalca znan, saj je enak času (običajno vsaj na sekundo natančno), ko je prejel paket. Morebitne popravke zaradi narobe nastavljene ure na klientovem računalniku ali zaradi latence omrežja je enostavno implementirati in imajo konstantno računsko zahtevnost. *pid* in *ppid* lahko napadalec enostavno prebere, če ima dostop do klientovega računalnika, ali pa uporabi nekaj hevrističnih pravil, da omeji njuno entropijo. Npr. *ppid* je pogosto enak 1, če ne, pa pogosto velja *pid* = *ppid* +1. Zato lahko predpostavimo, da sta omenjeni pravili zadostni, obe pa omogočata, da iz *ppid* izračunamo

[3] a << b je binarni operator bitnega pomika števila a za b mest v levo

pid oz. obratno. Iz tega sledi, da je entropija ključa v resnici 20 bitov, če ima napadelc dostop do klientovega računalnika, oz. približno 36 bitov, če nima dostopa. V prvem primeru bi tedaj porabil približno 25 sekund, v drugem pa približno 20 dni na HP 712/80.

Algoritem je Netscape sicer hitro popravil in izdal varnostni popravek. Seveda pa so stare verzije ostale ranjlive, dokler uporabniki niso nadgradili brskalnika. V javnosti sicer ni znan noben primer uporabe omenjenega napada izven raziskovalnega okvira. Napaka je zanimiva predvsem zaradi tega, ker je tu napačna uporaba PRNG v bistvu zmanjšala dolžino ključa in s tem prizadela varnost celotenega sistema.

7.2. OpenSSL (do verzije 0.9.2a)

Napaka je bila v PRNG algoritmu v knjižnici SSLeay, na katerem bazira OpenSLL. Napaka je omogočila napad preko vhodov, kjer lahko napadalec s približno 100 posebnimi zahtevami za generiranje naključnega števila rekonstruira celotno stanje ([6]).

Lastni (custom) algoritem md_rand, ki ga uporablja OpenSSL, je razdelil interno stanje na dva dela. Prvi del je spremenljivka *md*, ki se v vsakem koraku nastavi na izhod prejšnjega koraka, zgoščen z zgoščevalno funkcijo SHA-1. Drugi del je spremenljivka *state*, ki je krožno okno preko tabele naključnih vrednosti, ki služi kot vir entropije, a se nastavi le ob inicializaciji generatorja.

Očiten problem je, da spremenljivka *md* ni tajna, ker je v celoti odvisno od (javnega) izhoda. Poleg tega je velikost spremenljivke *state* znotraj posameznega koraka odvisna od velikosti izhoda s spodnjo mejo 1 bajt. Torej lahko napadalec s tem, ko zahteva le 1 bajt izhoda, zmanjša prostor *state* na 256 kombinacij, ki pa se ga da preiskati. Ti dve pomankljivosti sta omogočili napadalcu, da je s prvo zahtevo ugotovil *md*, potem pa je naredil še 100 enobajtnih zahtev, s katerimi se je sprehodil čez celotno tabelo naključnih vrednosti.

Popravek v verziji 0.9.2a je nastavil *md* na zgostitev celotnega prejšnjega stanja (tako *md* kot *state*) in naredil velikost *state* neodvisno od zahtevane velikosti izhoda.

V praksi bi bilo to napako težko izkoristiti, saj noben program ni omogočil napadalcu direktega dostopa do algoritma md_rand niti ni bilo mogoče obstoječe protokole zmanipulitari tako, da bi uporabljali md_rand na omenjeni način.

7.3. VNC (do verzije 3.3.3)

VNC oz. Virtual Network Client je aplikacija za dostop do oddaljenega grafičnega namizja. Za avtentifikacijo uporabnika uporablja preprost challenge-response protokol in tajno geslo. Strežnik da odjemalcu naključen, 16 bajtov dolg niz. Odjemalec nato ta niz zašifrira z DES s tajnim geslom, in niz pošlje strežniku. Strežnik nato ta niz odšifrira z DES s svojim geslom. Če sta gesli na strežniku in odjemalcu enaki, dobi strežnik na koncu postopka enak naključni niz, kot ga je poslal klientu.

Nekatere implementacije (npr. tightvnc) so za generiranje gesla uporabile običajen PRNG, ki so ga reinicializirale na začetku vsakega challenge-response cikla. Žal so za seme uporabile trenutni datum s sekundno natančnostjo. Zaradi tega se je dalo zaobiti avtentifikacijski postopek z enostavnim napadom s ponovitvijo klientovega odgovora, saj so bili znotraj sekundnega intervala vsi challengi (in zato tudi vsi responsi) enaki ([3]). Napadalec je sicer za ta napad moral imeti

možnost branja prometa med odjemalcem in strežnikom, kar pa ni bilo težko glede na to, da VNC pošilja ves promet kot čistopis.

Nekatere implementacije VNC pa so imele vgrajeno omejitev ene avtentifikacije na sekundo, s čimer so se tej napaki izognile. Vendar pa je ta "popravek" omogočil primitiven "Denial of Service" napad: napadalec se enostavno poskusi avtentificirati enkrat na sekundo (oz. bolj pogosto, če ga skrbi zamik ure ali zakasnitev omrežja).

Sicer je bilo že pred odkritjem te napake za VNC priporočeno, da se uporablja le preko tajne povezave (npr. VPN ali SSH tunel). Priporočilo je pomembno ne le zaradi omenjene napake pri uporabi PRNG, ampak tudi zaradi tega, ker protokol VNC nikjer ne šifrira prometa. Promet pa vsebuje predvsem informacije o zaslonski sliki, pritiskih na tipke ter premikih in klikih miške. Ti podatki v celoti razkrijejo delo uporabnika – ne nazadnje tudi uporabniško vnešena gesla za druge sisteme.

7.4. Nevarnost uporabniško generiranih ključev in nonce-ov

Danes se v praksi uporabljajo uporabniško generirani ključi le kot gesla za avtentifikacijo. Le-ta pa so običajno lahko uganljiva in jih uporabniki slabo varujejo ([4]) Kot je nakazano v poglavju 2, pogosto vsebujejo dosti manj entropije, kot je videti na prvi pogled. Poleg omenjenih pogostih "psevdonaključnih" nizov so gesla pogosto besede iz naravnega jezika, nekoliko spremenjeno uporabniškego ime (npr. dodana številka ali črke v obratnem vrstnem redu) ali javno dostopni podatki, povezani z uporabnikom (npr. rojstni dan, imena družinskih članov).

Zaradi tega se pri uporabi gesel zaščiti sistem z strogim omejevanjem preizkusov gesel, s čimer se napadalcu prepreči presikovnje celotnega prostora gesel. Poizkuse se omeji tako, da se po določenem številu napačno vnešenih gesel prepreči nadaljni vnos gesel za ta vir (npr. uporabniški račun), ter da se oteži preiskovanje gesel tudi kadar ima napadalec dostop do avtentifikacijskega sistema. Slednje se doseže z hranjenjem zgoščenih gesel (namesto čistopisa) ter z uporabo "soli". Hranjenje zgoščenih gesel (ali pa vsaj zašifriranih, če je potreben tudi dostop do čistopisa gesla) je sicer nujno tudi, kadar so gesla generirana s PRNG.

Na uporabniško generirane nonce danes v praksi ne naletimo več. Zanimiv primer zanje pa je opisan v [5] kot eden od v praksi izpeljanih napadov na Enigmo. Generiranje tročrkovnega inicializacijskega vektorja oz. "ključa sporočil" (message key), kot so ga takrat imenovali, je bilo prepuščeno operaterjem. Le-ti so ga običajno izbrali po zelo predvidljivem vzorcu, npr. tri enake ali pa tri na tipkovnici soležne črke. Potrebno je sicer poudariti, da se je pozneje način izbire in prenosa ključa sporočila spremenil. Zanimivo pa je, da so bili ljudje nadvse neprimeren vir entropije tudi pred 70 leti, in sicer iz povsem istih razlogov kot danes.

Glede na povedano je smiselno uporabiti uporabniško generirane ključe le tam, kjer se jih da primerno zaščititi ter kjer je pretežko distribuirati in uporabljati[4] ključe, generirane s PRNG.

[4] Največja slabost naključno generiranih ključev je to, da si jih je težko zapomniti, poleg tega pa moramo zaupati generatorju.

8. Zaključek

Glavni problem pri uporabi generatorjev naključnih števil za kriptografske namene je glede na predstavljne napade predvsem napačna uporaba. Najpogostejša je previsoka ocena entropije vira. Pogosta je tudi izbira preprostih ali pa gradnja lastnih, žal kriptografsko nepreverjenih algoritmov za PRNG. Napadi na uveljavljene CSPRNG so bolj teoretične narave in jim ni sledila demonstacija uporabe na nekem v praksi razširjenem kriptosistemu. Najbližje temu je prišel napad na WEP zaradi napake v RC4, vendar pa RC4 strogo vzeto ni PRNG. Praktične posledice tega napada pa sicer lahko primerjamo z napako v SSL v Netscape 1.1. Le-ta je bila zelo odmevna, a je bil zanjo vzrok lasten, nepreverjen algoritem.

Iz prikazanih napadov in napak PRNG pa lahko ugotovimo, da je treba nujno preveriti tudi ta del kriptosistema. PRNG se namreč pogosto ne posveča dovolj pozornosti, kljub temu, da se skoraj vedno uporablja na začetku kriptografskega postopka. Napadalec pa bo vedno izbral za napad najšibkejši del kriptosistema, ki je zaradi te prezrtosti lahko ravno PRNG.

9. Viri in literatura

[1] John Kelsey, Bruce Schneier, David Wagner, Chris Hall, *Cryptanalytic Attacks on Pseudorandom Number Generators*
http://www.schneier.com/paper-prngs.pdf

[2] Ian Goldberg, David Wagner, *Randomness and the Netscape Browser*
http://www.ddj.com/documents/s=965/ddj9601h/

[3] VNC authentication weakness, Bugtraq (junij 2004)
http://www.blacksheepnetworks.com/security/security/bugtraq/1417.html

[4] Robert Moriss, Ken Thompson, *Password Security: A Case History,* Communications of the ACM, Vol.22, No.11, 1979

[5] Kris Gaj, Arkadiusz Orlowski, *Facts and myths of Enigma: breaking stereotypes*, EUROCRYPT 2003
http://ece.gmu.edu/courses/ECE543/viewgraphs_F03/EUROCRYPT_2003.pdf

[6] *Weakness of the OpenSSL PRNG in versions up to OpenSSL 0.9.6a*
http://www.openssl.org/news/secadv_prng.txt

[7] L. Blum, M. Blum, and M. Shub, *A Simple Unpredictable Pseudo-Random Number Generator*, SIAM Journal on Computing, 1986

[8] John Savard, *A Cryptographic Compendium*
http://home.ecn.ab.ca/~jsavard/crypto/jscrypt.htm

[9] Scott Fluhrer, Itsik Mantin, and Adi Shamir, *Weaknesses in the Key Scheduling Algorithm of RC4*
http://www.drizzle.com/~aboba/IEEE/rc4_ksaproc.pdf

[10] N. Pavešić, *Informacija in kodi*, založba FE in FRI, 1997

[11] A. K. Lenstra, H. W. Lenstra, Jr. and L. Lova`sz, *Factoring Polynomials with Rational Coefficients*, Math. Ann. 261, 1982

[12] E. R. Berlekamp, *Algorithmic Coding Theory*, McGraw-Hill, 1968

Fortuna