

RSA Fallen

Andreas Klein

August 19, 2008

Dies ist ein erweiterter Leserbrief zum Artikel "Computer, bitte kürzen" im Linux-Magazin 09/08.

Auf Seite 33 muss es heißen `p1:next_prime(377^220)`. Dann stimmt die Angabe, dass `p1` etwa 500 Stellen hat. Die Warnung, dass man bei RSA keine aufeinander folgende Primzahlen nehmen sollte ist noch untertrieben. Ist `p2:next_prime(p1)`, so gilt `p2:next_prime(sqrt(n))` was die Faktorisierung (d.h. das Brechen des RSA) zu einem Kinderspiel macht.

Ernster ist jedoch, ein anderer Fehler. Primzahlen der Form `next_prime(a^b)` (mit kleinen ganzen Zahlen a, b) sind für das RSA-Verfahren **nicht** geeignet.

Zum einen gibt es grob geschätzt nur jeweils 256 Möglichkeiten für a und b . Damit liegt die Entropie nur bei 16 Bit und ist damit viel zu niedrig. Was eine niedrige Entropie bei der Sicherheit anrichten kann, konnte man vor kurzem erst beim Debian-SSL Debakel erleben.

Der zweite Punkt ist noch erster und erledigt alle möglichen Primzahlen der Form $a^b + c$ mit kleinen a, b und c .

Ich werde den Angriff Schritt für Schritt entwickeln.

Eine Zahl heißt B -klatt wenn sie nur Primfaktoren $\leq B$ besitzt. Ist der Wert B nicht weiter wichtig so spricht man einfach nur von einer glatten Zahl.

Wir nehmen an, dass $n = p_1 p_2$ ist und die Primzahlen die Form $p_1 = g_1 + \epsilon_1$ und $p_2 = g_2 + \epsilon_2$, wobei g_1 und g_2 glatte Zahlen und ϵ_1 und ϵ_2 im Vergleich kleine Zahlen sind.

Dann ist

$$n = p_1 p_2 = g_1 g_2 + g_1 \epsilon_2 + g_2 \epsilon_1 + \epsilon_1 \epsilon_2 \approx g_1 g_2 .$$

Der Angriff zielt darauf ab, die glatte Zahl $g = g_1 g_2$ in der Nähe von n zu rekonstruieren. Wir schreiben $g = p_1^{e_1} \cdot \dots \cdot p_k^{e_k}$ und erhalten:

$$p_1^{e_1} \cdot \dots \cdot p_k^{e_k} \approx n$$
$$e_1 \log(p_1) + \dots + e_k \log(p_k) \approx \log(n)$$

Solche Näherungen lassen sich mit dem LLL-Algorithmus berechnen. (Der LLL-Algorithmus ist ein wichtiger Algorithmus der Computeralgebra und dient zum Berechnen von kurzen Vektoren in Gittern siehe Wikipedia http://en.wikipedia.org/wiki/Lenstra-Lenstra-Lov\'asz_lattice_reduction_algorithm oder J. von zur Gathen und J. Gerhard, Modern Computer Algebra, Cambridge

University Press.) In Axiom und Maxima ist der LLL-Algorithmus leider nicht implementiert, aber es gibt freie Implementierungen z.B. in NTL und Sage.

Wir wählen eine große Zahl N (etwas kleiner als \sqrt{n}) und bauen ein Gitter mit der Basismatrix

$$\begin{pmatrix} 1 & & 0 & \lfloor \log(p_1)N \rfloor \\ & \ddots & & \vdots \\ 0 & & 1 & \lfloor \log(p_k)N \rfloor \\ 0 & \dots & 0 & \lfloor \log(n)N \rfloor \end{pmatrix}$$

Dann suchen wir eine LLL-Basis für dieses Gitter. Diese besteht aus Vektoren mit kleinem Gewicht. Der Vektor

$$(e_1, \dots, e_k, e_1 \lfloor \log(p_1)N \rfloor + \dots + \lfloor e_k \log(p_k)N \rfloor - \lfloor \log(n)N \rfloor)$$

hat kleines Gewicht und ist mit hoher Wahrscheinlichkeit der erste Vektor der LLL-Basis. (Wir müssen nur k und N richtig wählen.)

Um die Zahl aus dem Artikel zu faktorisieren wähle ich $N = 10^{400}$ und die ersten 20 Primzahlen $p_1 = 2$ bis $p_{20} = 71$. (Ich lasse hier alle Abschätzungen weg, die zeigen warum das ganze klappt. Wer sich dafür interessiert, muss auf den Fachartikel warten den ich momentan schreibe.)

Bevor wir mit der Faktorisierung der Zahl im Artikel weiter machen können testen wir die Idee an einer Zahl mit bekannter Faktorisierung:

Wir setzen

```
p1:=next_prime(481^177);
p2:=next_prime(377^210);
n:=p1*p2;
```

Der LLL-Algorithmus findet das Basiselement.

$$(0, 0, 0, 0, 0, -387, 0, 0, 0, -210, 0, -177, 0, 0, 0, 0, 0, 0, 0, 0, 491)$$

Also ist $13^{387} \cdot 29^{210} \cdot 37^{177} \approx n$ und wir erraten korrekter Weise $g_1 = (13 \cdot 29)^{210}$ und $g_2 = (13 \cdot 37)^{177}$.

Die Zahl im Artikel ist etwas schwieriger. Mit dem LLL-Algorithmus finden wir $n \approx (7 \cdot 53)^{433}$. Also erwarten wir das die beiden gesuchten Primzahlen jeweils in der Nähe einer Potenz 371 liegen, wir kennen aber noch nicht den zugehörigen Exponenten.

Zu mindestens liegt die Vermutung nah, dass $(7 \cdot 53)^x$ für kleine x ein gemeinsamer Teiler von g_1 und g_2 ist. Dies erlaubt uns $\epsilon_1 \epsilon_2$ zu bestimmen. Bei der Zahl im Artikel gibt es wieder ein kleines Problem daher testen wir unsere Idee zu erst an der Zahl $n = (7^{210} \cdot 53^{216} + 330)(7^{217} \cdot 53^{211} + 398)$.

Wir raten, dass der ggT(g_1, g_2) zumindest $(7 \cdot 53)^5$ enthält. In diesem Fall ist $n = (g_1 + \epsilon_1)(g_2 + \epsilon_2) \pmod{(13 \cdot 37)^{177}} = \epsilon_1 \epsilon_2$. Da $\epsilon_1 \epsilon_2$ sehr klein ist wird uns das auffallen.

Wir überprüfen unsere Vermutung.

```
n mod (7*53)^4;
  131340
n mod (7*53)^5;
  131340
n mod (7*53)^6;
  131340
```

Es spricht also vieles dafür, dass $\epsilon_1\epsilon_2 = 131340$ ist.
Bei der Zahl im Artikel haben wir auf diese Weise kein Erfolg.

```
n mod (7*53)^5;
  5251693771116
n mod (7*53)^6;
  2521494664775774
n mod (7*53)^7;
  709185138652986165
```

Danach versuchte ich, ob $\epsilon_1\epsilon_2 < 0$ ist. Zunächst wieder kein Erfolg:

```
-n mod (7*53)^5;
  1776917879735
-n mod (7*53)^6;
  86120257689947
-n mod (7*53)^7;
  258239997581796326
```

Es dauerte eine Weile bis ich auf die Idee kam, wesentlich größer Produkte von $\epsilon_1\epsilon_2$ in Betracht zu ziehen:

```
-n mod (7*53)^49;
  78922592492039161826532978098784043042681682112157512
-n mod (7*53)^50;
  78922592492039161826532978098784043042681682112157512
-n mod (7*53)^51;
  78922592492039161826532978098784043042681682112157512
```

Also ist

$$\epsilon_1\epsilon_2 = -78922592492039161826532978098784043042681682112157512$$

Etwas Probieren überzeugte mich davon, dass $\text{ggT}(g_1, g_2) = (7 \cdot 53)^{212}$ ist.

```
-n mod (7*53)^212;
  78922592492039161826532978098784043042681682112157512
-n mod (7*53)^212*7;
  552458147444274132785730846691488301298771774785102584
-n mod (7*53)^212*53;
  4182897402078075576806247839235554281262129151944348136
```

Ich vermutete $g_1 = (7 \cdot 53)^{212}$ und $g_2 = (7 \cdot 53)^{221}$
Also suchen wir die Lösung von:

$$\begin{aligned} ((7 \cdot 53)^{212} + \epsilon_1)((7 \cdot 53)^{221} + \epsilon_2) &= n \\ -78922592492039161826532978098784043042681682112157512 &= \epsilon_1 \epsilon_2 \end{aligned}$$

Dieses System hat die ganzzahlige Lösung:

$$\begin{aligned} \epsilon_1 &= 8883838837776377363653637188 \\ \epsilon_2 &= -8883838837377363653636274 \end{aligned}$$

Damit ist die Faktorisierung gefunden.

Die zur Faktorisierung benötigte Rechenzeit liegt weit unterhalb einer Sekunde.

Zum Schluss möchte ich noch kurz zeigen wie man bruchbare RSA Primzahlen erzeugen kann.

Der folgende Maxima-Code erzeugt eine zufällige 500-stellige Primzahl:

```
p:random(10^500);  
unless(primep(p)) do n:random(10^500)
```

Auch zu diesem Code eine Warnung. Der Zufallsgenerator in Maxima ist für Simulationen aber **nicht** für die Verwendung in kryptographischen Funktionen gedacht. Erst wenn man die Standard `random` Funktion durch einen kryptographisch sicheren Zufallsgenerator ersetzt bekommt man ein funktionierendes RSA-Verfahren.