

Message de la discussion [Weak Keys in RC4](#)

Andrew Roos [Afficher le profil](#)

[Autres options](#) 22 sep 1995, 09:00

-----BEGIN PGP SIGNED MESSAGE-----

A CLASS OF WEAK KEYS IN THE RC4 STREAM CIPHER

PRELIMINARY DRAFT

ANDREW ROOS
VIRONIX SOFTWARE LABORATORIES

1. INTRODUCTION

This paper discusses a **class** of **weak keys** in RSA's **RC4** stream cipher. It shows that for at least 1 out of every 256 possible **keys** the initial byte of the pseudo-random stream generated by **RC4** is strongly correlated with only a few bytes of the key, which effectively reduces the work required to exhaustively search **RC4** key spaces.

2. STATE TABLE INITIALIZATION IN RC4

Although the **RC4** algorithm has not been published by RSA Data Security, source code to implement the algorithm was anonymously posted to the Cypherpunks mailing list several months ago. The success of the Cypherpunks' brute-force attack on SSL with a 40-bit key indicates that the source code published did accurately implement **RC4**.

RC4 uses a variable length key from 1 to 256 bytes to initialize a 256-byte state table which is used for the subsequent generation of pseudo-random bytes. The state table is first initialized to the sequence {0,1,2,...,255}. Then:

```
1  index1 = 0;
2  index2 = 0;
3
4  for(counter = 0; counter < 256; counter++)
5  {
6    index2 = (key_data_ptr[index1] + state[counter] + index2) % 256;
7    swap_byte(&state[counter], &state[index2]);
8    index1 = (index1 + 1) % key_data_len;
9  }
```

Note that the only line which directly affects the state table is line 7, when two bytes in the table are exchanged. The first byte is indexed by "counter", which is incremented for each iteration of the loop. The second byte is indexed by "index2" which is a function of the key. Hence each element of the state table will be swapped at least once (although possibly with itself), when it is indexed by "counter". It may also be swapped zero, one or more times when it is indexed by "index2". If we assume for the moment that "index2" is a uniformly distributed pseudo-random number, then the probability that a particular single element of the state table will be indexed by "index2" at some time during the initialization routine is:

$$P = 1 - (255/256)^{255} \\ = 0.631$$

(The exponent is 255 because we can disregard the case when "index2" and "counter" both index the same element, since this will not affect its value.)

Conversely, there is a 37% probability that a particular element will not be indexed by "index2" during initialization, so its final value in the state table will only be affected by a single swap, when it is indexed by "counter". Since key bytes are used sequentially (starting again at the beginning when the key is exhausted), this implies:

- A. Given a key length of K bytes, and $E < K$, there is a 37% probability that element E of the state table depends only on elements 0..E (inclusive) of the key.

(This is approximate since "index2" is unlikely to be uniformly distributed.)

In order to make use of this, we need to determine the most likely values for elements of the state table. Since each element is swapped at least once (when it is indexed by "counter"), it is necessary to take into account the likely effect of this swap. Swapping is a nasty non-linear process which is hard to analyze. However, when dealing with the first few elements of the state table, there is a high probability that the byte with which the element is swapped has not itself been involved in any previous exchanges, and therefore retains its initial value {0,1,2,...,255}. Similarly, when dealing with the first few elements of the state table, there is also a significant probability that none of the state elements added to index2 in line 6 of the algorithm has been swapped either.

This means that the most likely value of an element in the state table can be estimated by assuming that $state[x] == x$ in the algorithm above. In this case, the algorithm becomes:

Discussions

[+ nouveau message](#)

[À propos de ce groupe](#)

[Abonnement à ce groupe](#)

Ceci est un groupe Usenet - [En savoir plus](#)

Liens commerciaux

[SSL Library](#)

Open Source SSL/TLS

Implementation

OpenSSL compatible, fast, portable
www.yassl.com

[Protégez vos logiciels](#)

Gérez vos licences par dongles ou par clé logicielle

www.aplika.fr

[Publicité par Keyword](#)

Votre Pub Immédiatement sur Google

Vous ne Payez que les Clics Reçus !
AdWords.Google.fr

[Diffusez votre publicité sur cette page...](#)

```

1 index1 = 0;
2 index2 = 0;
3
4 for(counter = 0; counter < 256; counter++)
5 {
6     index2 = (key_data_ptr[index1] + counter + index2) % 256;
7     state[counter] = index2;
8     index1 = (index1 + 1) % key_data_len;
9 }

```

Which can be reduced to:

B. The most likely value for element E of the state table is:

$$S[E] = X(E) + E(E+1)/2$$

where X(E) is the sum of bytes 0..E (inclusive) of the key.

(when calculating the sum of key elements, the key is considered to "wrap around" on itself).

Given this analysis, we can calculate the probability for each element of the state table that it's value is the "most likely value" of B above. The easiest way to do this is to evaluate the state tables produced from a number of pseudo-randomly generated **RC4 keys**. The following table shows the results for the first 47 elements from a trial of 100 000 eighty-bit **RC4 keys**:

	Probability (%)							
0-7	37.0	36.8	36.2	35.8	34.9	34.0	33.0	32.2
8-15	30.9	29.8	28.5	27.5	26.0	24.5	22.9	21.6
16-23	20.3	18.9	17.3	16.1	14.7	13.5	12.4	11.2
24-31	10.1	9.0	8.2	7.4	6.4	5.7	5.1	4.4
32-39	3.9	3.5	3.0	2.6	2.3	2.0	1.7	1.4
40-47	1.3	1.2	1.0	0.9	0.8	0.7	0.6	0.6

The table confirms that there is a significant correlation between the first few values in the state table and the "likely value" predicted by B.

3. WEAK KEYS

The **RC4** state table is used to generate a pseudo-random stream which is XORed with the plaintext to give the ciphertext. The algorithm used to generate the stream is as follows:

x and y are initialized to 0.

To generate each byte:

```

1 x = (x + 1) % 256;
2 y = (state[x] + y) % 256;
3 swap_byte(&state[x], &state[y]);
4 xorIndex = (state[x] + state[y]) % 256;
5 GeneratedByte = state[xorIndex];

```

One way to exploit our analysis of the state table is to find circumstances under which one or more generated bytes are strongly correlated with a small subset of the key bytes.

Consider what happens when generating the first byte if state[1] == 1.

```

1 x = (0 + 1) % 256;          /* x == 1 */
2 y = (state[1] + 0) % 256;   /* y == 1 */
3 swap_byte(&state[1], &state[1]); /* no effect */
4 xorIndex = (state[1] + state[1]); /* xorIndex = 2 */
5 GeneratedByte = state[2]

```

And we know that state[2] is has a high probability of being

$$S[2] = K[0] + K[1] + K[2] + 2(2+1) / 2$$

Similarly,

$$S[1] = K[0] + K[1] + 1(1+1) / 2$$

So to make it probable that S[1] == 1, we have:

$$K[0] + K[1] == 0 \pmod{256}$$

In which case the most likely value for S[2] is:

$$S[2] = K[2] + 3$$

This allows us to identify a **class** of **weak keys**:

C. Given an **RC4** key K[0]..K[N] with K[0] + K[1] == 0 (mod 256), there is a significant probability that the first byte generated by **RC4** will be K[2] + 3 (mod 256).

Note that there are two special cases, caused by "unexpected" swapping during key generation. When K[0]==1, the "expected" output byte is k[2] + 2, and when k[0]==2, the expected value is k[2] + 1.

There are a number of similar classes of "**weak keys**" which only affect a few

keys out of every 65536. However the particular symmetry in this **class** means that it affects one key in 256, making it the most interesting instance.

Once again I took the easy way out and used simulation to determine the approximate probability that result C holds for any given key. Probabilities ranged between 12% and 16% depending on the values of K[0] and K[1], with a mean of about 13.8%. All these figures are significantly greater than the 0.39% which would be expected from an uncorrelated generator. The key length used was again 80 bits. This works the other way around as well: given the first byte B[0] generated by a **weak** key, the probability that $K[2] = B[0] - 3 \pmod{256}$ is 13.8%.

4. EXPLOITING WEAK KEYS IN RC4

Having found a **class** of **weak keys**, we need a practical way to attack **RC4** based cryptosystems using them. The most obvious way would be to search potential **weak keys** first during an exhaustive attack. However since only one in every 256 **keys** is **weak**, the effective reduction in search space is not particularly significant.

The usefulness of **weak keys** does increase if the opponent is satisfied with recovering only a percentage of the **keys** subjected to analysis. Given a known generator output which includes the first generated byte, one could assume that the key was **weak** and search only the **weak keys** which would generate the known initial byte. Since 1 in 256 **keys** is **weak**, and there is a 13.8% chance that the assumed value of K[2] will be correct, there is only a 0.054% chance of finding the key this way. However, you have reduced the search space by 16 bits due to the assumed relationship between K[0] and K[1] and the assumed value of K[2], so the work factor per key recovered is reduced by a factor of 35, which is equivalent reducing the effective key length by 5.1 bits.

However in particular circumstances, the known relationships between **weak keys** may provide a much more significant reduction in workload. The remainder of this section describes an attack which, although requiring very specific conditions, illustrates the potential threat.

As a stream cipher, a particular **RC4** key can only be used once. When multiple communications sessions are required, some mechanism must be provided for generating a new session key each time. Let us suppose that an implementation chose the simple method of incrementing the previous session key to get the new session key, and that the session key was treated as a "little endian" (least significant byte first) integer for this purpose.

We now have the interesting situation that the session **keys** will "cycle through" **weak keys** in a pattern which repeats every 2^{16} **keys**:

```
00 00 00 ... Weak
(510 non-weak keys)
FF 01 00 ... Weak
(254 non-weak keys)
FE 02 00 ... Weak
(254 non-weak keys)
FD 03 00 ... Weak
...
01 FF 00 ... Weak
(254 non-weak keys)
00 00 01 ... Weak
(510 non-weak keys)
FF 01 01 ... Weak
```

(Least significant byte on the left)

Now while an isolated **weak** key cannot be identified simply from a known generator output, this cycle of **weak keys** at known intervals can be identified using statistical techniques since each of the **weak keys** has a higher than expected probability of generating the same initial byte. This means that an opponent who knew the initial generated bytes of about 2^{16} session **keys** could identify the **weak keys**, and would also be able to locate the 510-key gap between successive cycles of **weak keys** (although not precisely). Since the 510-key gap occurs immediately following a key which begins with 00 00, the opponent not only knows that the **keys** are **weak**, but also knows the first two bytes of each key. The third byte of each key can be guessed from the first output byte generated by the key, with a 13.8% chance of a correct guess. Assuming that the "510-key gap" is narrowed down to 1 of 8 **weak keys**, the attacker can search a key space which is 24 bits less than the size of the session **keys**, with a $13.8\%/8$ chance of success, effectively reducing the key space by approximately 18 bits.

Although this particular attack depends on a very specific set of circumstances, it is likely that other **RC4** based cryptosystems in which there are linear relationships between successive session **keys** could be vulnerable to similar attacks.

5. RECOMMENDATIONS

The attacks described in this algorithm result from inadequate "mixing" of key bytes during the generation of the **RC4** state table. The following measures could be taken to strengthen cryptosystems based on the **RC4** algorithm:

- (a) After initializing the algorithm, generate and discard a number of bytes. Since the algorithm used to generate bytes also introduces additional non-linear dependencies into the state table, this would make analysis more difficult.

(b) In systems which require multiple session **keys**, ensure that session **keys** are not linearly related to each other.

(c) Avoid using the **weak keys** described.

6. CONCLUSION

This preliminary analysis of **RC4** shows that the algorithm is vulnerable to analytic attacks based on statistical analysis of its state table. It is likely that a more detailed analysis of the algorithm will reveal more effective ways to exploit the weaknesses described.

Andrew Roos <andr...@vixonix.co.za>

-----BEGIN PGP SIGNATURE-----

Version: 2.6.2i

```
iQCVAwUBMGLAk2atuqa4OR+IAQGYJQQA1W2r/giH1iPxeLRjooPEvAJJO2GHrBNy
h1fjHhPf6uBhBapEyZfN5utaUZYkkz/3tXJQC1p+17XwAJHGxb6kapHI3tAf2k5B
P7C034fo8WIOmam1GQqIG3c1MPjCvkNY02NEkYAmNtcwKMP96QgDMCbvS0kn55WE
L1GOWMVYqO4=
=iogI
-----END PGP SIGNATURE-----
```

[Répondre à l'auteur](#) [Transférer](#)

[Créer un groupe](#) - [Google Groupes](#) - [Accueil Google](#) - [Conditions d'utilisation](#) - [Règles de confidentialité](#)

©2007 Google