Data Dependent Cryptografy:

© Glenn Larsson, 2001

(Ichinin@SUESPAMMERS.org)

Abstract: I present a way of increasing the key agility in existing crypto algorithms to decrease the posibillity for analysts to make use of known plaintext-to-ciphertext pairs to decrypt message traffic.

Introduction:

Modern stream ciphers such as ”ArcFour” (*) have one Major problem; Plaintext-to-Cipertext attacks are feasible with only a very low understanding on how the algorithm works. Hardware implementations with static keys are one major threat to system security; some have no key exchange implemented at all and some have a small finite looping IV that allow a degree (~2^24) of key agility, but this is not enough.

I choose ”ARCFour” because it is easy to elaborate on and for you to relate to because it is byte based and have no bit permutations.

(* This is believed to be the RC4 algorithm, this is unverified by RSA)

I must also point out that that this is **not** a flaw in any crypto algorithms, it is a *protocol problem*.

The problem with static keys:

(First of all, I’d like to point out that Known plaintext attacks are nothing new. If you know how this works, skip this part.)

Cryptograpic systems produce similar ciphertext(s) by using fixed keys and non agile encryption protocols, i.e.

```
for (n = 0; n< sizeof(plaintext); n++)
{
      ciphertext[n] = encrypt(plaintext[n], key);
}
```

An Example:

The system we talk about right now, is perhaps a chat client, it have peer-to-peer file transfer capabilities and some kind of encryption.

The users (Alice & Bob) send files between them; MP3, ZIP, Word documents (etc) over the chat client. They also talk over the same channel, utilising the same key for each session. Utilising a sniffer and an ICMP redirect attack, (aka ”Man in the middle” attack), the Eavesdropper ”Eve” picks up 2 messages: ALPHA and BETA.

Message *ALPHA* is encrypted with an unknown key.
Message *BETA* is also encrypted with the unknown key.

Message *ALPHA* is a word document, i.e. it have this standard header in it (first 32 bytes):

```
d0 cf 11 e0 a1 b1 1a e1 00 00 00 00 00 00 00 00 -
00 00 00 00 00 00 00 00 3e 00 03 00 fe ff 09 00
```

Message *BETA* is a chat session that utilises the same key the ciphertext.

1. Eve may know that user Alice is a happy word user and prefer to write data in Word 2000. So Eve simply xor the ciphertext with the standard header from above and produce a possible Keystream(*). So, Eve have NOT recovered a possible key, but a keystream.

2. Eve now xor the possible keystream with the encrypted chat session and we prove that the security of the chat system is broken.

3. Eve can now intercept message traffic between Alice and Bob in realtime.

(*) A keystream could be described as ”the results of a key”, i.e. if we compare a crypto algorithm with with a keyed PRNG, then the random numbers that the PRNG produce is the Keystream.

Example 1: Using the regular algorithm

Key = 0x31,0x32,0x33,0x34,0x35

Plaintext1 ”beermilkshake”
Plaintext2 ”beermilkshakes” (one extra byte)

Ciphertext1 ”32,50,5D,10,41,77,60,04,E8,4B,B5,DB,BD”
Ciphertext2 ”32,50,5D,10,41,77,60,04,E8,4B,B5,DB,BD,15”

Observation:

We clearly see the resemblence between the Ciphertext pairs. Both are encrypted using the same key and both produce similar ciphertext outputs with the exception for the last byte. By recoverying the keystream for Ciphertext2 we can easily extract information from ciphertext1 by xoring the ciphertext with the keystream.

Example 2: Using the extended algorithm

Key = 0x31,0x32,0x33,0x34,0x35

Plaintext1 ” F.Zappa rule”
Plaintext2 ” F.Zappa rules” (one extra byte)

Ciphertext1 ”3C,C2,A6,63,67,76,9F,4C,27,68,F5,3E”
Ciphertext2 ”AA,D2,58,76,76,8E,D,55,4F,EC,37,6D,5D”

Observation:

Now, both ciphertext contain the similar message but the Ciphertext relationships between the Cipertext are totally Different. Recovery of any of the keystreams in one message will not allow the attacker to extract information from the other message.

Appendix A: *Pseudocode for standard ”ARCFour” Encryption algorithm*

```
// Fill S() with 0,1,2…N
// Fill K(i) with Key bytes

// Key setup
For i = 1 to 256
      j = (j + S(i) + K(i)) Modulo 256
      Swap S(I, J)
Next i

i = 0: j = 0

For x = 1 To Sizeof(Data)
      i = (i + 1) Modulo 256
      j = (j + S(i)) Modulo 256
      Swap S(I,J)
      Vector = (S(i) + (S(j)) Modulo 256) Modulo 256
      Data(x) = Data(x) xor S(Vector)
Next X
```

Appendix B: *Pseudocode for Extended ”ARCFour” Encryption algorithm:*

```
// Fill S() with 0,1,2…N
// Fill K(i) with Key bytes

// Key setup
For i = 1 to 256
      j = (j + S(i) + K(i)) Modulo 256
      Swap S(I) and S(J)
Next i

// reset I and J to 0
i = 0: j = 0

For x = 1 To Sizeof(Data) // FIRST loop
      i = (i + 1) Modulo 256
      j = (j + S(i)) Modulo 256
      Swap S(I,J)
      Vector = (S(i) + (S(j)) Modulo 256) Modulo 256
      Data(x) = Data(x) xor S(Vector)
Next X

// Note I and J stays the same, they are NOT reset to 0(!)
// just step through the Data[] array again. Nothing special.

For x = 1 To Sizeof(Data) // SECOND loop
      i = (i + 1) Modulo 256
      j = (j + S(i)) Modulo 256
      Swap S(I,J)
      Vector = (S(i) + (S(j)) Modulo 256) Modulo 256
      Data(x) = Data(x) xor S(Vector)
Next X
```

References & Further reading.

- None I know of, sorry.