RC4 Biases in TLS/SSL

Tyler Manning-Dahan

Concordia University

Abstract. These notes present the overall structure of the RC4 stream cipher for data encryption in the TLS protocol. Different biases of RC4 as well as attacks using these biases will be explored as well. These notes are not complete, nor do they represent an entire understanding of all previous research, but it is a basic introduction.

Table of Contents

Introduction

Transport Layer Security (TLS) is a cryptographic protocol that is currently used for communications security between applications over networks. It is based on Secure Sockets Layer (SSL) and is often referred to as TLS/SSL given that TLS version 1.0 was based on SSL 3.1 and since formally standardized by the Internet Engineering Task Force (IETF). For this paper it will be referred to jointly as TLS/SSL unless a specific version of the protocol is analyzed.

TLS/SSL grew out of the need for users to send sensitive information securely over the Internet. As a basic example, customers sending banking information

from an Internet browser to a server. It was designed to sit between the Transmission Control Protocol (TCP[1]) layer and any application layer (e.g. HTTP, FTP, SMTP).

TLS/SSL is comprised of several cryptographic elements but our focus will be on the versions of TLS that are dependent on a particular stream cipher known as RC4. While the IETF highly recommends never to negotiate a secure session using RC4 stream cipher, it still remains part of cipher suites in TLS 1.2 and all of its previous versions [8]. Thankfully, most clients and servers have completely dropped support for cipher suites with RC4 as an option and it will be completely removed from the protocol with TLS 1.3, making it no longer a vulnerability moving forward. I will focus on the RC4 biases that allowed several attacks to weaken the TLS/SSL protocol.

1 Background

There are several layers that makeup the TLS/SSL protocol, but I will focus on the Record protocol that is used to encapsulate the other layers and all information being transferred, referring to them as records. This protocol has two basic properties [1]:

1. The connection is private or confidential. This aspect is achieved using symmetric cryptography to encrypt the data. Symmetric algorithms such as DES, RC4, and AES are typically used here.
2. The connection is reliable, meaning it maintains data integrity. This reliability or integrity is ensured through the use of a message authenticated code (MAC) or more precisely a keyed MAC (HMAC) that is achieved by hashing a secret key using MD5, SHA-256 for example.

Stream ciphers are a natural choice for the symmetric encryption section post handshake as they require little overhead and are much faster than certain modes of operation of block ciphers. Stream ciphers are interesting because unlike their block cipher cousins, they are an endless stream or states that are defined by previous states, akin to a Markov Chain. We can therefore define a particular class of stream ciphers, known as *synchronous* stream ciphers, whose key stream is generated independently of the plain-text and later combined with the plain-text to form the cipher-text[12]. One advantage of this setup is the key stream can be computed separately or in parallel with the plain text, making it very fast. The RC4 stream cipher is an example of this. Before evaluating the RC4 stream cipher's role in TSL/SSL, we'll examine how it works as a stream cipher.

2 RC4 Stream Cipher

Initially a trade secret created by Ron Rivest at RSA Data Security Inc. [9], its algorithm was leaked on-line in the mid-90's allowing external researchers to finally test it.

[1] Often grouped as the TCP/IP layer to include the Internet Protocol

The RC4 stream is a cipher that is made up of two smaller functions. It's main algorithm is made up of a key scheduling algorithm (KSA) and a pseudo-random generator algorithm (PRGA). The KSA takes in a small, secret key and outputs a scrambled initialization array that is used as a pseudo-seed by the PRGA to make the key stream that will be used to encrypt or decrypt using an XOR function in a typical stream cipher fashion.

2.1 Key Scheduling Algorithm

First, the KSA will setup the initial state of the seed to be used. It starts with an array S of length 256 bytes with all values permuted in order to start. An index or counter, i, then increments all the way through the entire array of S, i.e. 256 times. A second index pointer j is then updated by adding its previous value to S_i and the secret key, k, at k_i. This addition is performed modulo 256 to keep the indexes within the bounds of S. The value, S_j, that the resulting pointer j is pointing to is then swapped with the value at S_i. This is then repeated 256 times. The python pseudo-code for the KSA can be found below.

```
S = range(256)
j = 0
for i in range(256):
    j = (j + S[i] + key[i mod keylength]) mod 256
    swap S[i] and S[j]
return S
```

Essentially, the KSA can be thought of as an extractor[2], though it takes in a fixed-length key (typically between 128-256 bits) and returns a scrambled 256 byte "seed". Notice in the first line of the for loop that RC4 also treats the key as an array of bytes with byte values between 0 and 255.

Examining the internals of the for loop reveals the incrementation of the j index is the source of the randomness. The term `key[i mod keylength]` pads the key in case it is shorter than the S array, which doesn't add much randomness but the `S[i]` term does. In fact it is critical for returning an appropriately random index j that will be used to choose what value S_i should swap with after each round. We can see this by running a quick test with the key initialized as *k=[1,2,3,4,5]*. Using the KSA above yields a suitably random index j for the 256 iterations. However, repeating this test with same key but omitting `S[i]` from the index update, yields the following discernible pattern for index j:

$$1, 3, 6, 10, 15, 16, 18, 21, 25, 30, 31, 33, 36, 40, 45, 46, 48.... \quad (1)$$

Now, the final swap after each iteration does alleviate the issue by scrambling the values, but the final array S would have a very noticeable pattern every second item, which is still significantly weak.

[2] A function that takes in a large data pool of weak entropy and outputs a seed of fixed length that is statistically random.

Also note that without the use of a nonce or an initialization vector, reusing the same key with would result in the same output every time, making the KSA completely deterministic. This is clear by running the KSA several times with the same key.

Furthermore, not all keys are created equal when running the KSA. This should be obvious from the algorithm that using a very short key (i.e. less than the 112 bit NIST standard) or a repetitive key (e.g. all 1's or 0's) would result in a very predictable sequence. Andrew Roos first discussed the class of "weak keys" and their affect on the final key stream [10]. Weak key attacks are another research focus, but will be left out this paper.

The output of the KSA is sometimes referred to as a "secret internal state" [3] because an adversary could not reproduce this state without the secret key and it is essentially as critical to the security of the stream as the key itself. This is because the initial state of S can completely define the PRGA in the next step. I am curious as to why we go through these unique steps of generating this seed rather than simply using a known extractor that has less vulnerabilities or simply a hash. It seems counterproductive to go through a TLS/SSL handshake to finally get fairly random, shared key, only to run it through a process that doesn't sufficiently add randomness.

One reason may be because of the number of possible states the S array could have. It is interesting to note how many possible states we could end with from the KSA. We have an array with 256 permutations or 256! where each value in the array is a byte representing 2^8 bits and two pointers to keep track of therefore, we have the following number of possible states:

$$256! * (2^8) * (2^8) \approx 2^{1700} \tag{2}$$

2.2 Pseudo-random Number Generator Algorithm

Following the KSA, the PRGA takes in the S array or "seed" and calculates its own indexes using S_i according to the initial state. Again, these indexes are used to swap S_i and S_j, thereby updating the array, S, after each iteration. Note all additions here are done modulo 256. The python pseudo-code for the PRGA is:

```
i = 0
j = 0
while True:
    i = (i + 1) mod 256
    j = (j + S[i]) mod 256
    swap S[i] and S[j]

    Z = S[(S[i] + S[j]) mod 256]
    yield Z
```

Recall that a pseudo-random generator outputs an endless stream of bits that are computationally random. Here, the output of the PRGA is the key stream, Z, which is dependent on the pointers i and j that are calculated differently

than the KSA. In the PRGA, i is simply a counter, incrementing every round by 1 where j is dependent on its previous state and S_i, which starts as the secret internal state of S. Since, the counter i is not the source of randomness, it rests upon the index j to provide this. Ideally, at each round, any value in S should have a 50% chance to be pointed to by the index j.

Note that discovering a particular state of the PRGA would allow an adversary to perfectly replicate the key-stream. This includes the original secret internal state that was generated by the KSA in the previous algorithm.

As stated earlier, the KSA is completely deterministic making the PRGA also deterministic. Some implementations of RC4 that need Chosen-Plain text Attack (CPA) security introduce an initialization vector (IV) or nonce into the algorithm. This is kind of tricky since there is no obvious place in the KSA or PRGA to add it in there, and it also adds an additional piece that must be kept confidential. I suppose one way to do this would be to concatenate the key with an IV and hash that before running it through the KSA.

Like other stream ciphers, the key stream output of RC4 is used as an encryption function by bit-wise exclusive-or with a plain-text message, similar to a one-time pad. In other words, the cipher-text, C_i is calculated at bit i by $C_i = P_i \oplus Z_i$, where P_i and Z_i are the plain text and key stream respectively at bit i. This also means that, like the one-time pad, if one bit of the plain-text or key-stream gets flipped, it will result in the same bit getting flipped in the resulting cipher-text. Decryption is done in a similar fashion, where the receiver can XOR the cipher-text with the keys stream on his end since both sides are using RC4 with the same key, the key stream output will be identical. Ideally, the randomness of this key stream should be very close to a truly random output, making the stream cipher as close to a one-time pad as possible.

2.3 Biases in RC4

The key stream produced by the PRGA should produce random sequences with uniform distribution, never favoring particular patterns. However, favoring particular outputs in the key stream has been known for a long time and are known as biases or RC4 output biases because of this uneven probability. Numbers appearing more often than they should can be referred to as "positive biases" and, by contrast, numbers appearing less often then they should are "negative biases"[11]. Furthermore, we can make a distinction between biases that appear only in the initial key stream, called *short term biases*, and biases that appear cyclically throughout the key stream, called *long term biases*[11]. The long term biases effectively occur on a recurring basis as the cycle of the key stream is eventually repeated.

This distinction is important and comes up a lot when discussing biases because if significant biases only occur in the first cycle of the key stream (e.g. short term biases) then one way to deal with it would be to simply discard the first X amount of bytes of Z until we reach a point in the PRGA that has uniform randomness. Not only would this it be incredibly wasteful and inefficient if we went through the KSA and PRGA only to throw away part of the resulting

key stream, Z, but the fact that long term biases exist tells us that no matter how much of the key stream we discard, we will always encounter some biased outputs.

It is interesting to note, that in some scenarios, such as SSH where speed and code size is not always critical, discarding a fixed length of the initial key stream is actually done in practice. In [6], the first 1536 bytes of the 128-key RC4 key stream is discarded before encryption begins. This does add more complexity because the discarded bytes could reveal the key used in the session therefore this must be discarded carefully and never shared over the network. An adversary who acquires these discarded key steam bytes could feasibly brute force it at their leisure offline, and eventually recover the entire plain text conversation since RC4 is not forward secret.

One of the most famous short-term biases was pointed out by Mantin and Shamir [7]. They discovered that for an RC4 implementation based on the 256 byte algorithm described above, the second byte of the key stream will be equal to zero for twice the expected probability, while all other values have uniform probability of appearing. In other words, $Pr[Z_2 = 0] = 2/256$ instead of the expected probability of $E[Z_i = 1, 0] = 1/256$. It is complicated to explain exactly why this happens as the proof of this theorem involves advanced mathematics that I cannot explain yet. However, I believe it is due to the factors hinted upon earlier when discussing the makeup of the KSA and the PRGA. The indices are not incremented in a perfectly random fashion resulting in certain values of the S array being favored and swapped more often than others.

In terms of long term biases, Fluhrer and McGrew discovered consecutive pairings of bytes (called a *digraph*) where (Z_k, Z_{k+1}) was found to be biased towards certain values, which are all summarized in [4]. For their research, they based their results on the counter i in the PRGA and assumed the internal state of S was uniformly random, which we know for sure not to be true in the initial key stream from the Mantin and Shamir paper.

Finding biases is an art in of itself, and doing so in a statistically coherent fashion is fairly complex. I do not fully understand the logic behind it or how they can validate one approach versus another. The cryptography field is unique in that way that tools or algorithms are used until proven guilty. Many biases have been found since and I will not list them all here, but they simply demonstrate that the previous found ones were not flukes or due to poorly chosen keys and more cracks will reveal themselves over time. All these biases are do to the inherent design of RC4.

3 RC4 Role in TLS/SSL

Following the handshake protocol, we fall back into the TLS/SSL record protocol with both parties finally having the same master secret which will be used as a session key that is only valid for the duration of their session. If both parties agreed to use RC4 then they will concatenate the plain text message to be sent with an HMAC tag, T. The HMAC is done across the concatenation of a header

number, a sequence number and the data to be sent, denoted R [11]. The tag is then concatenated with the plain text, resulting in the following $P = R||T$. From here, the plain text is encrypted by the RC4 key stream as previously described. Note that the key used to initialize the KSA is usually hashed to get it down to the desired length.

3.1 Attacks on TLS/SSL

While the biases have continued to be found over the years from as early is the mid-90's, the movement to stop using RC4 as an option in TLS/SSL cipher suites only happened recently. This was in part due to the fact that a popular block cipher that was often used in TLS/SSL was deemed insecure and as a result many servers and clients started favoring the cipher suites that used RC4 for the symmetric encryption aspect post handshake. This prompted the research communities to investigate the previous vulnerabilities found in RC4 in the context of TLS/SSL.

Now, using the fact that we know the second byte is 0 with twice the probability in [7], if an adversary were to record thousands of TLS/SSL sessions keyed with plain texts that have the first few bytes in common, you could deduce the second byte. This may not sound significant but think of it in a context where thousands of similar plain texts would be send over the wire. For example, if these messages were votes for an election and the byte that contained the vote for a particular party was contained in the second byte. Then an adversary could not only figure out which way users were voting but since the RC4 stream cipher is malleable they could potentially flip the right bits and change the end result of the election, making this massively insecure. This example requires a lot of presumptions about the setup and is improbable given today's computational power, but in 10 or 20 years this real-time attack could be feasible.

In [11] they also exploited several short term biases to After capturing 9×2^{27} encryptions of a cookie sent over HTTPS. They were able to brute-force it with high success rates in negligible time by running a piece of JavaScript code in the browser of a victim. They executed the attack in practice in about 52 hours.

Conclusion and Further Readings

In this paper, we examined the RC4 stream cipher, how it works, why it produces several biased outputs and how they may be exploited in a TLS/SSL scenario. There are many other possible attacks including attacks from weak keys, key recovery from the key stream itself, and state recovery attacks, all summarized nicely in Sen Supta et al.[5]. While RC4 is no longer used in TLS/SSL applications, it remains a very useful, simple stream cipher that is still used in other cryptographic protocols. Therefore, understanding its limitations and internal workings are important for protection against attacks that exploit these design limitations.

Bibliography

[1] Dierks, T. et Allen, C. (1999). Rfc 2246: The tls protocol. *IETF, January*.

[2] Dierks, T. et Rescorla, E. (2006). Rfc 4346: The transport layer security (tls) protocol. *URL http://www. ietf. org/rfc/rfc4346. txt*.

[3] Fluhrer, S., Mantin, I. et Shamir, A. (2001). Weaknesses in the key scheduling algorithm of rc4. *In International Workshop on Selected Areas in Cryptography*, pages 1–24. Springer.

[4] Fluhrer, S. R. et McGrew, D. A. (2000). Statistical analysis of the alleged rc4 keystream generator. *In International Workshop on Fast Software Encryption*, pages 19–30. Springer.

[5] Gupta, S. S., Maitra, S., Paul, G. et Sarkar, S. (2014). (non-) random sequences from (non-) random permutations—analysis of rc4 stream cipher. *Journal of Cryptology*, 27(1):67–108.

[6] Harris, B. (2006). Improved arcfour modes for the secure shell (ssh) transport layer protocol.

[7] Mantin, I. et Shamir, A. (2001). A practical attack on broadcast rc4. *In International Workshop on Fast Software Encryption*, pages 152–164. Springer.

[8] Popov, A. (2015). Prohibiting rc4 cipher suites. *Computer Science*, 2355: 152–164.

[9] Rivest, R. (1996). Rc4. *Applied Cryptography by B. Schneier, John Wiley and Sons, New York*.

[10] Roos, A. (1995). A class of weak keys in the rc4 stream cipher.

[11] Vanhoef, M. et Piessens, F. (2015). All your biases belong to us: Breaking rc4 in wpa-tkip and tls. *In 24th USENIX Security Symposium (USENIX Security 15)*, pages 97–112.

[12] Vanstone, S. A. et van Oorschot, P. C. (1997). *Handbook of applied cryptography*, chapitre 6, pages 191–195. CRC press.