WireGuard

with ChaCha20 Algorithm

Cryptography

Yang, Heetak (Leslie)

Introduction

In this case study, we will learn more about the operation of WireGuard, a new Virtual Private Network method. They say it's called the Next Generation Virtual Private Network, and it's a new VPN with all the old encryption algorithms and systems changed. It will also be integrated into Linux, and although it is an unfinished technology, it has a lot of potential. So this time I study new crypto algorithms with WireGuard.

Background

Security researcher and kernel developer Jason A. Donenfeld came up with the idea for WireGuard while looking for a covert traffic tunneling solution to use during penetration testing in 2017. VPN tunnels such as IPsec and Open VPN did not perform well and were difficult to properly configure and manage. So, the WireGuard project which focused on security and simplicity, began, so a completely new VPN which ignore existing design and technoloties. It was developed for the Linux kernel, but now supports all cross-platform use.

Compared to OpenVPN, which is composed of more than 100,000 lines of code and relies on another vast codebase, OpenSSL, the code of the WireGuard kernel module is only about 4,000 lines and contains encryption code. In addition, it is said to take into account the portability of transplanting to a new place. This reason, it has a smaller attack surface and does not respond to unauthorised packets compared to other VPN projects, making it much more difficult to attack.

In Linux, WireGuard works entirely in kernel space, so it performs significantly better than OpenVPN. Morever, performance and connection speeds are more than four times faster than OpenVPN on the same hardware, and even faster than IPsec-based VPNs. However, WireGuard implementations for the client softwares are written in a Go memory safe programming language. For Apple product family and Android, client programs are implemented in Swift and Java, respectively, and Rust is also being developed for more secure client programs.

Cryptography process used

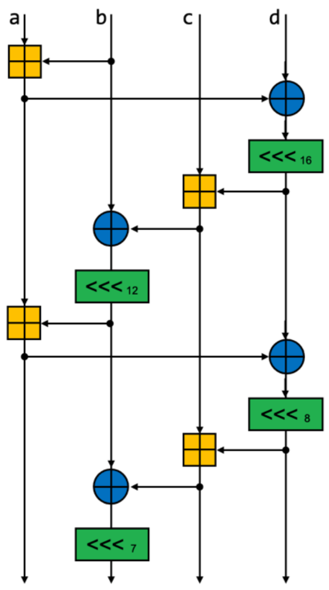
WireGuard protocol abandoned the notion of cryptographic agility, i.e. providing a choice of various encryption, key exchange, and hashing algorithms. Because in other technologies this concept has contributed to an unsafe environment. Instead, WireGuard uses modern cryptographic primitives that have been thoroughly tested and peer reviewed. The result is a strong default encryption that cannot be changed or misconfigured by the user. When a serious vulnerability is found in the encryption primitives used, a new version of the protocol emerges. There is also a mechanism to negotiate the protocol version between peers.

WireGuard uses ChaCha20 symmetric encryption along with Poly1305 for message authentication. Elliptic curve Diffie-Hellman (ECDH) Key consensus uses Curve 25519, hashing uses BLAKE2, which is faster than SHA-3. It uses a 1.5-RTT (Round Trip Time) handshake based on a noise framework that provides delivery security. And WireGuard protocol, each peer is identified to other peers through a short public key in a manner similar to OpenSSH's key-based authentication. The public key is part of a new concept the WireGuard developers refer to as cryptokey routing and is also used to set the IP address assigned to each peer in the tunnel.

Algorithms of Cryptography

The encryption algorithm uses ChaCha20 and Poly1305. These cryptographic algorithms were created by Daniel J. Bernstein. ChaCha20 is a symmetric key cipher algorithm created in 2008 based on Salsa20, which he designed in 2005. The Poly1305 was also produced by Bernstein, a cryptographic message authentication code (MAC), designed to be used to verify data integrity and message reliability.

Chacha20 is an ARX-based hash function, is keyed and runs in counter mode. It is mainly used for encryption, but the key is a pseudorandom number generator. The ciphertext is obtained by XORing the plain text into a pseudo-random stream. If you don't use the same temporary value twice with the same key, you can treat that stream as a One Time Pad (OTP).



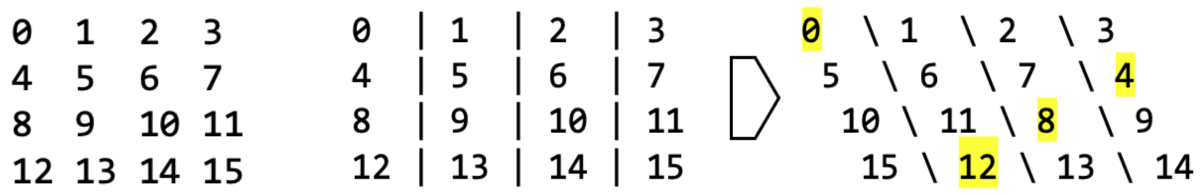
```
blk[0]: "expa"
blk[1]: "nd 3"
blk[2]: "2-by"
blk[3]: "te k"
Blk[4]: key[0]
Blk[5]: key[1]
Blk[6]: key[2]
Blk[7]: key[3]
Blk[8]: key[4]
Blk[9]: key[5]
Blk[10]: key[6]
Blk[11]: key[7]
Blk[12]: counter[0]
Blk[13]: counter[1]
Blk[14]: nonce[0]
Blk[15]: nonce[1]
```

"expa"	"nd 3"	"2–by"	"te k"
Key 0	Key 1	Key 2	Key 3
Key 4	Key 5	Key 6	Key 7
Cont. 0	Cont. 1	Non. 0	Non. 1

"expand 32-byte k"

[Picture1.]

In terms of configuration is 32bit, bit addition ⊕ (XOR: Exclusive-OR), 32bit addition mod 2^32 ⊞, and a pseudo-based rotation operation <<< (ARX: Add-Rotate-XOR). It is based on a random number function. In its internal state, it consists of 16 32-bit words arranged in a 4x4 matrix. In the initial state, in addition to the internal state, there are 128bit constants, 256bit keys, 64bit counters, 64bit bis, and rearrange some words. Indexing matrix elements from 0 to 15. It works on 32bit unsigned integers denoted by a to d. This is the explanation for the ChaCha quarter round.



[Picture2.]

```
1st round              2nd round -> diagonal 1

QR(0, 4, 8, 12)        QR(0, 5, 10, 15)
QR(1, 5, 9, 13)        QR(1, 6, 11, 12)
QR(2, 6, 10, 14)       QR(2, 7, 8, 13)
QR(3, 7, 11, 15)       QR(3, 4, 9, 14)
```

[Picture3.]

To properly list everything, I'll first apply 4 quarter rounds in parallel, column by column. Then arrange 4 more diagonally, it will be printed like this. And converting this to the formula:

```
a += b;   d ^= a;   d <<<= 16;
c += d;   b ^= c;   b <<<= 12;
a += b;   d ^= a;   d <<<= 8;
c += d;   b ^= c;   b <<<= 7;
```

[Picture4.]

Test using the above sequence and formulas. Hexadecimal is briefly checked through C.

```
a = 0x10101010;
b = 0x10010010;
d = 0x10000100;
a = a + b = 0x10101010 + 0x10010010 = 0x20111020
d = d ^ a = 0x10000100 ^ 0x20111020 = 0x30111120
d = d<<<16 = 0x11200000
```

```
#include <stdio.h>

int main() {
    int a, b, c, d;
    a = 0x10101010;
    b = 0x10010010;
    d = 0x10000100;

    a += b; d ^= a; d <<= 16;
    printf("%#x", d);
    return 0;
}
```

[Picture5-6.]

In Picture5., this is the first line following the graph of Picture1. To prove this, I simply wrote a code in C and solved it. You just need to repeat it 4 times, but this can also be easily expressed like this.

```
a = 0x10101010;
b = 0x10010010;
c = 0x10001000;
d = 0x10000100;
```

```
a = 0x11200000;
b = 0x11010000;
c = 0x32102000;
d = 0x18980000;
```

```c
#include <stdio.h>

int main() {
    int a, b, c, d;
    a = 0xa1513351;
    b = 0xf51651a2;
    c = 0x535b5216;
    d = 0x1af515c7;

    a += b; d ^= a; d <<= 16;
    printf("%#x \n", d);
    c += d; b ^= c; b <<= 12;
    printf("%#x \n", b);
    a += b; d ^= a; d <<= 8;
    printf("%#x \n", d);
    c += d; b ^= c; b <<= 7;
    printf("%#x \n", b);
    return 0;
}
```

[Picture7-8.]

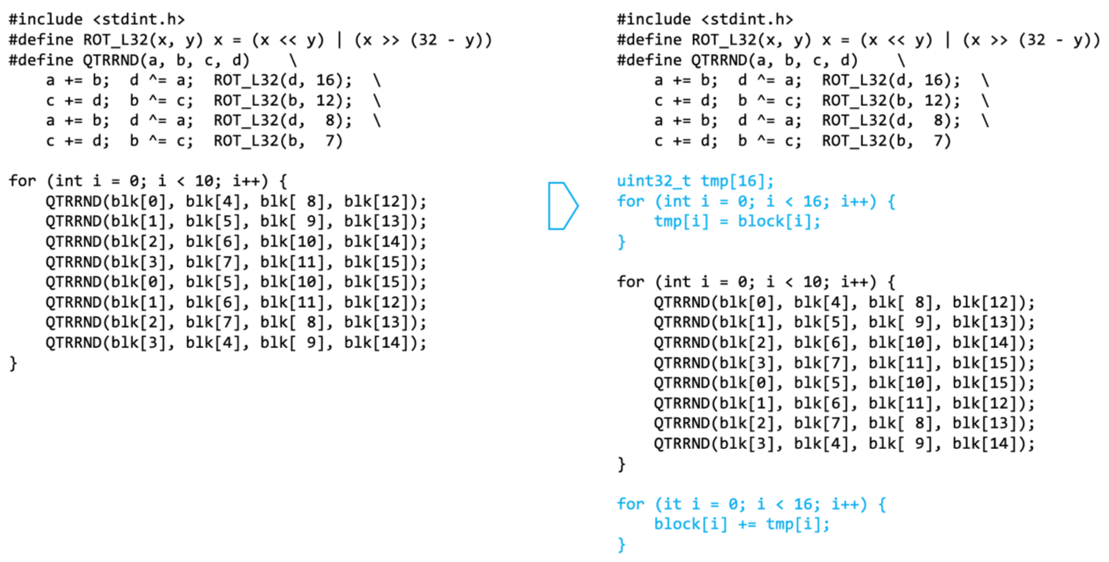
Let's check the following with another example. In the same way as above, the code of a to d is a random hexadecimal code. There is a 4x4 matrix like Picture2, and it is an example that is the first of the columns in the state in which the first round has been turned. Then, it becomes a quarter round (0, 5, 10, 15), where each position is the highlighted position in the picture below, and if the part is calculated as in Picture7-8, it changes from left to right.

```
0xa1513351 0xe8516b1c 0xd5161372 0xa5648e15
0x156e158d 0xf51651a2 0xf151687a 0x75ca8741
0xb5100a51 0x105dc156 0x535b5216 0xa854688b
0xc84923df 0xa51d8459 0x5d851b7a 0x1af515c7
```

```
0x91340000 0xe8516b1c 0xd5161372 0xa5648e15
0x156e158d 0x903b4000 0xf151687a 0x75ca8741
0xb5100a51 0x105dc156 0x96c4f300 0xa854688b
0xc84923df 0xa51d8459 0x5d851b7a 0xb7828b00
```

[Picture9.]

This method is called 1, and 2 rounds are played per each. ChaCha20's 20 is the number of rounds, so you have to play 20 rounds, so you have to play 10 rounds.



```
#include <stdint.h>
#define ROT_L32(x, y) x = (x << y) | (x >> (32 - y))
#define QTRRND(a, b, c, d)     \
    a += b;  d ^= a;  ROT_L32(d, 16);  \
    c += d;  b ^= c;  ROT_L32(b, 12);  \
    a += b;  d ^= a;  ROT_L32(d,  8);  \
    c += d;  b ^= c;  ROT_L32(b,  7)

for (int i = 0; i < 10; i++) {
    QTRRND(blk[0], blk[4], blk[ 8], blk[12]);
    QTRRND(blk[1], blk[5], blk[ 9], blk[13]);
    QTRRND(blk[2], blk[6], blk[10], blk[14]);
    QTRRND(blk[3], blk[7], blk[11], blk[15]);
    QTRRND(blk[0], blk[5], blk[10], blk[15]);
    QTRRND(blk[1], blk[6], blk[11], blk[12]);
    QTRRND(blk[2], blk[7], blk[ 8], blk[13]);
    QTRRND(blk[3], blk[4], blk[ 9], blk[14]);
}
```

```
#include <stdint.h>
#define ROT_L32(x, y) x = (x << y) | (x >> (32 - y))
#define QTRRND(a, b, c, d)     \
    a += b;  d ^= a;  ROT_L32(d, 16);  \
    c += d;  b ^= c;  ROT_L32(b, 12);  \
    a += b;  d ^= a;  ROT_L32(d,  8);  \
    c += d;  b ^= c;  ROT_L32(b,  7)

uint32_t tmp[16];
for (int i = 0; i < 16; i++) {
    tmp[i] = block[i];
}

for (int i = 0; i < 10; i++) {
    QTRRND(blk[0], blk[4], blk[ 8], blk[12]);
    QTRRND(blk[1], blk[5], blk[ 9], blk[13]);
    QTRRND(blk[2], blk[6], blk[10], blk[14]);
    QTRRND(blk[3], blk[7], blk[11], blk[15]);
    QTRRND(blk[0], blk[5], blk[10], blk[15]);
    QTRRND(blk[1], blk[6], blk[11], blk[12]);
    QTRRND(blk[2], blk[7], blk[ 8], blk[13]);
    QTRRND(blk[3], blk[4], blk[ 9], blk[14]);
}

for (it i = 0; i < 16; i++) {
    block[i] += tmp[i];
}
```

[Picture10.]

However, 20 rounds are not enough for security. If apply the reverse order of each operation in reverse order, it get the original block. Then it has the ability to predict the whole stream and recover the original plain text. So, add blue colour code above and below the original working code to help it blend properly.

Encrypted by the above process, and the entire encryption process through ChaCha20 and Poly1305 is as follows.

1. The ChaCha block function transforms the state by executing several branch rounds.
2. Run 20 rounds and add the original state to the result.
3. The state serialises the state when the ChaCha20 was done. For example:

```
0xc84923df 0x105dc156 0xf151687a 0xa5648e15   ➔   df 23 49 c8 56 c1 5d 10 7a 68 51 f1 15 8e 64 a5
```

4. Generating Poly1305 Keys Using ChaCha20: Onetime Poly1305 pseudo-random generation is allowed.
   - 256bit session integrity key is used as the ChaCha20 key.
   - Block counter is set to 0.
5. Creates authenticated encryption with associated data (AEAD) to ensure the confidentiality, integrity, and authenticity of the data.

The following are the results and details of ChaCha20 and Poly1305 made simply based on the contents written above.

```
Plain text: Hello
Pass phrase: World!
(used to generate encryption key with SHA-256)

ChaCha20:                                              Poly1305:
•  Key seed (based on SHA-256 of World!): 514b6bb7c846e  •  Plain text (Hex): 48656c6c6f
   cfb8d2d29ef0b5c79b63e6ae838f123da936fe827fda654276c  •  Pass phrase (Hex): 514b6bb7c846ecfb8d2d29e
•  Key stream (based on length of Hello): 3850787338       f0b5c79b63e6ae838f123da936fe827fda654276c
•  Text stream: 48656c6c6f                              •  Tag: 2ea65f3355559008d4268be341756c15
•  Output stream: 7035141f57
```

[Picture.11]

In the case of decryption, just follow reverse the encrypt method.

1. First, calculate the one-time Poly1305 key.
2. Create an AEAD buffer.
3. Count the Poly1305 tag and find a match.
4. Decrypt the ciphertext with the calculated tag.

Security Objectives Fulfilled

The first, noise IK of Noise Framework is used for non-repudiation, which is a protocol framework developed based on CurveCP, NaCL, KEA+, SIGMA, and etc. Developer said, "Silence is a virtue", because there is no need to assign state to the server for potentially unauthenticated messages, so the first handshake message sent requires authentication. This is to ensure that state is not saved prior to authentication and a response to unauthenticated packets is not sent. WireGuard is designed to be invisible to rogue peers and network scanners, with no stateful and no response generated for unauthorized packets.

Second is Authentication. As mentioned above, WireGuard is certifying using ChaCha20 with Poly1305. It is used to check data integrity and authenticity of messages and was created based on the use of AES as an encryption algorithm, but instead uses ChaCha20.

The last is a hashing technique used not only for confidentiality but also for integrity checking, which are used not only for confidentiality but also for integrity check, BLAKE2 is used as a hashing algorithm, and Curve25519 is an Elliptic-Curve Diffie-Hellman (ECDH), which makes the encryption key necessary for the ChaCha20 with Poly1305 secure communication. Use it as a way to exchange safely. And SipHash24 is one of the data structures that store data as [Key, Value] as a hash table, which helps you quickly search for data. Additionally, WireGuard provides confidentiality by not responding to packets from unrecognised peers: network-scan does not indicate.

Conclusion

VPN is essential to business. However, in the existing case, the encryption algorithm is obsolete, also very slow compared to security. To replace this, WireGuard is next-generation VPN, uses Poly1305 with ChaCha20 to encrypt it, which combination outperforms AES in embedded CPU architectures without cryptographic hardware acceleration. In the future, WireGuard will be the standard of VPN, and ChaCha20 will be used a lot for Internet security.

Reference

Jason A. Donenfeld. “WireGuard: Next Generation Kernel Network Tunnel” (2020), - WireGuard -

Gordon Procter. “A Security Analysis of the Composition of ChaCha20 and Poly1305”, - ChaCha20 and Poly1305 -

Yoav Nir. “ChaCha20 and Poly1305 for IETF protocols” IETF (2015), - ChaCha20 and Poly1305 -

“http://www.blake2.net” - Blake2 Hashing -

“http://www.noiseprotocol.org” - 1.5-RTT 3-handshake -