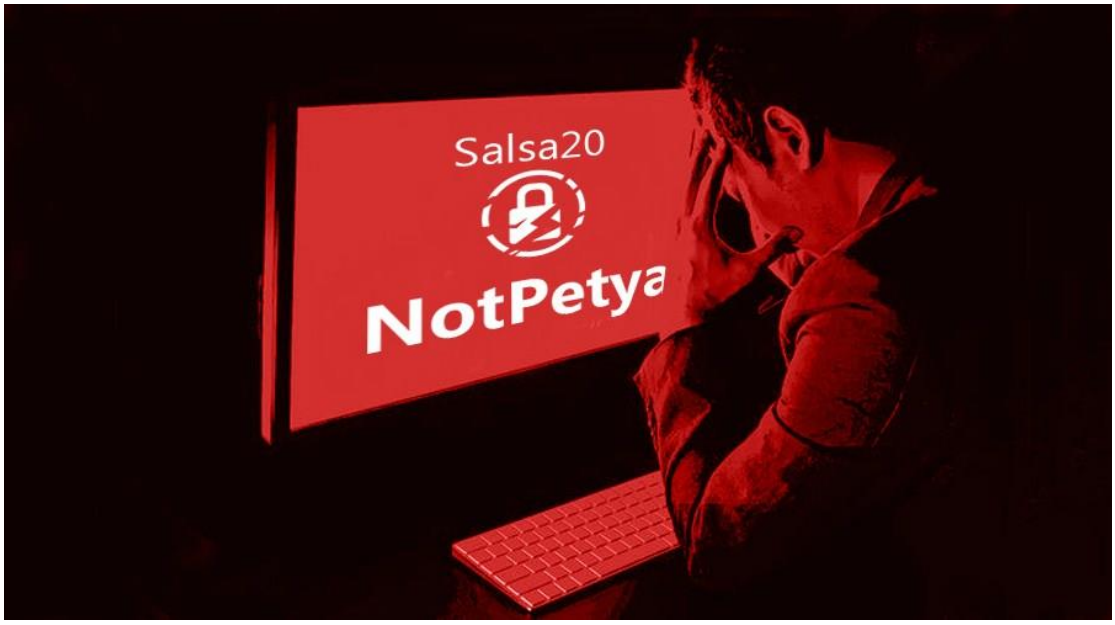

《NotPetya 勒索病毒 Salsa20 算法实现的缺陷分析》

安全报告：NotPetya Salsa20 算法实现中的缺陷
报告编号：B6-2017-072701
报告来源：360 网络安全响应中心
报告作者：Shi Lei, pwd, K-one
更新日期：2017 年 7 月 27 日

目录

0x00 背景

2017 年 6 月份，NotPetya 勒索病毒试图通过 The Shadow Brokers 泄露出的“永恒之蓝”等漏洞再次攻击全球网络系统。

目前，关于 NotPetya 的算法解密工作目前并没有明确的解密进展。

360CERT 团队在对 NotPetya 病毒所自定义实现的 Salsa20 算法分析过程中，发现存在两处变化。其中一处明显降低了标准 Salsa20 算法的加密强度，在一定条件下可以对加密数据进行解密。

0x01 Salsa20 的简单介绍

Salsa20 的原理是产生一种伪随机的字节流。 这种字节流在很长（约 2 的 70 次方）范围内和真正的随机字节流无法区分。 这样做到了和一次一密密码本（OTP，One Time Pad）加密等价的效果。

伪随机数流的产生其实就是将 64 字节（512 比特）的输入送入核心函数，然后得到 512 比特的输出的过程。 每次输入的字节包含密钥、初始向量和计数器。 这样，要产生长度是 N 字节的伪随机数流，只需要调用核心函数若干次，直到获取了足够长度（不少于 N）的输出即可。

64 字节的输入如下：

```
常量1 || 密钥前半 || 常量2 || 初始向量 || 计数器 || 常量3 || 密钥后半 || 常量4
.0-3.....4-19.....20-23.....24-31.....32-39....40-43.....44-59.....60-63.
```

本文所讨论的差异就是上面 64 字节产生的不同，核心函数都是相同的。

0x02 差异一：常量不同

原算法的常量：

```
// "expand 32-byte k"
//uint8_t o[4][4] = {
//  { 'e', 'x', 'p', 'a' },
//  { 'n', 'd', ' ', '3' },
//  { '2', '-', 'b', 'y' },
//  { 't', 'e', ' ', 'k' }
//};
```

样本的常量：

```
//"-1nvalid s3ct-id"
{ 0x2D ,0x31 ,0x6E ,0x76 },
{ 0x61 ,0x6C ,0x69 ,0x64 },
{ 0x20 ,0x73 ,0x33 ,0x63 },
{ 0x74 ,0x2D ,0x69 ,0x64 }
```

【结论】对算法的强度没有影响

0x03 差异二：小端化函数不同

原算法 s20_littleendian 函数：

```
static uint32_t s20_littleendian(uint8_t *b)
{

 return b[0] +
        ((uint_fast16_t) b[1] << 8) +
        ((uint_fast32_t) b[2] << 16) +
        ((uint_fast32_t) b[3] << 24);*/
```

样本 s20_littleendian 函数：

```
static uint32_t s20_littleendian(uint8_t *b)
{

    uint32_t result;
    __asm mov     esi, b
    __asm sub     ah, ah
    __asm mov     al, [esi + 2]
    __asm shl     ax, 10h
    __asm cwd
    __asm mov     cx, ax;  保存ax
    __asm mov     ah, [esi + 1]
    __asm sub     al, al
    __asm mov     bx, dx; 保存 dx
    __asm cwd
    __asm add     ax, cx
    __asm adc     dx, bx
    __asm mov     cx, ax
    __asm mov     ah, [esi + 3]
    __asm shl     ah, 10h
    __asm sub     al, al
    __asm mov     bx, dx
    __asm cwd
    __asm add     ax, cx
    __asm adc     dx, bx
    __asm mov     cl, [esi]
    __asm sub     ch, ch
    __asm add     ax, cx
    __asm adc     dx, 0


    __asm and     eax, 0xffff
    __asm shl     edx, 0x10
    __asm or      eax, edx
    __asm mov     result, eax


    return result;
```

【差异】样本中原本是想模拟原算法的操作，但因为要在 MBR 中运行，采用了 WORD 为单位的运算，shl ax 10h 时就会把 ax 清零。这样导致的后果就是 64 字节的输入的高位 WORD 会被填充为 0，相当于将原函数改为：

```
static uint32_t s20_littleendian(uint8_t *b)
{

    return b[0] +
        ((uint_fast16_t)b[1] << 8);
```

【效果】**原先的 64 字节输入：**

0xdeadbeef 0xdeadbeef……0xdeadbeef(一共 16 个 0xdeadbeef)

经过 s20_littleendian 函数后：

0x0000beef 0x0000beef ……0x0000beef(一共 16 个 0x0000beef)

【**影响**】进入核心函数后：

```
#define R(a,b) (((a) << (b)) | ((a) >> (32 - (b))))
void salsa20_word_specification(uint32 out[16],uint32 in[16])
{
    int i;
    uint32 x[16];
    for (i = 0;i < 16;++i) x[i] = in[i];
    for (i = 20;i > 0;i -= 2) {     // 迭代次数，注意每次 i -= 2 !
        x[ 4] ^= R(x[ 0]+x[12], 7);  x[ 8] ^= R(x[ 4]+x[ 0], 9);
        x[12] ^= R(x[ 8]+x[ 4],13);  x[ 0] ^= R(x[12]+x[ 8],18);
        x[ 9] ^= R(x[ 5]+x[ 1], 7);  x[13] ^= R(x[ 9]+x[ 5], 9);
        x[ 1] ^= R(x[13]+x[ 9],13);  x[ 5] ^= R(x[ 1]+x[13],18);
        x[14] ^= R(x[10]+x[ 6], 7);  x[ 2] ^= R(x[14]+x[10], 9);
        x[ 6] ^= R(x[ 2]+x[14],13);  x[10] ^= R(x[ 6]+x[ 2],18);
        x[ 3] ^= R(x[15]+x[11], 7);  x[ 7] ^= R(x[ 3]+x[15], 9);
        x[11] ^= R(x[ 7]+x[ 3],13);  x[15] ^= R(x[11]+x[ 7],18);
        x[ 1] ^= R(x[ 0]+x[ 3], 7);  x[ 2] ^= R(x[ 1]+x[ 0], 9);
        x[ 3] ^= R(x[ 2]+x[ 1],13);  x[ 0] ^= R(x[ 3]+x[ 2],18);
        x[ 6] ^= R(x[ 5]+x[ 4], 7);  x[ 7] ^= R(x[ 6]+x[ 5], 9);
        x[ 4] ^= R(x[ 7]+x[ 6],13);  x[ 5] ^= R(x[ 4]+x[ 7],18);
        x[11] ^= R(x[10]+x[ 9], 7);  x[ 8] ^= R(x[11]+x[10], 9);
        x[ 9] ^= R(x[ 8]+x[11],13);  x[10] ^= R(x[ 9]+x[ 8],18);
        x[12] ^= R(x[15]+x[14], 7);  x[13] ^= R(x[12]+x[15], 9);
        x[14] ^= R(x[13]+x[12],13);  x[15] ^= R(x[14]+x[13],18);
    }
    for (i = 0;i < 16;++i) out[i] = x[i] + in[i];
}
```

虽然输入的随机序列有一半为零，但是经过 1 轮异或移位的操作，随机序列已经不含零了。并且该算法要进行 10 轮这样的操作，所以得到的序列随机化程度还是很高。

进入核心函数前：

```
0x002EF624  2d 31 00 00 12 8b 00 00 67 77 00 00
0x002EF630  46 7d 00 00 86 f8 00 00 61 6c 00 00
0x002EF63C  a9 c2 00 00 aa 43 00 00 01 80 00 00
0x002EF648  00 00 00 00 20 73 00 00 48 1c 00 00
0x002EF654  90 38 00 00 4d be 00 00 df 0e 00 00
0x002EF660  74 2d 00 00 cc cc cc cc cc cc cc cc
```

经过一轮异或移位操作后：

```
0x002EF624  09 b0 78 66 fd 58 67 cb 15 a8 5f 0a
0x002EF630  fe f9 06 e0 ac 40 a7 ce 1a 60 cd b3
0x002EF63C  a3 58 e3 67 f8 1b 73 cd c3 cb bf 05
0x002EF648  89 7f f6 cd 32 a9 94 9b ab a0 8e 2a
0x002EF654  96 a1 14 e0 65 42 8b 01 6c 96 e5 fc
0x002EF660  4a c7 2f 57 cc cc cc cc cc cc cc cc
```

【结论】64 字节的输入经过小端化函数后会导致高位 2 个字节清零。这样的话，爆破该输入的规模就从 2 的 256 次方降为了 2 的 128 次方 ,约为 10 的 38 次方，所以直接爆破出密钥的可能性几乎没有。

0x04 “修改版” Salsa20 算法的缺陷攻击

NotPetya 勒索病毒的修改版 Salsa20 算法造成的差异会导致每隔 64K 块出现重复的核心函数输入项，这将极大影响这种加密算法的安全性。对此，算法攻击者只要已知连续 4MB 明文 ,就能解密全部密文。另外若已知若干离散明文块，则可解密部分密文，也可能解密全部密文（已知部分分布合适的情况）。

相关证明如下：

```
C:\>C:\NotPetya\Crack_NotPetya_Salsa20.exe
Generate random iv(64bit): 152D7B0286618FF4
Generate random password(256bit): 66D81ABD0E20C76E130678710D512CD59DB026E6AEF04C
01FA72CDA6D0DD06CF
Generate random data(10MB), CRC: 5CA265AA
Encrypting...
CRC of encrypted data: 25BD9512
Decrypting...
CRC of decrypted data: 5CA265AA

      -- by Shi, Lei@360CERT && 360GearTeam
```

0x05 与 Petya 中修改 Salsa 进行对比

Petya 中修改的 Salsa 分析链接：http://www.freebuf.com/vuls/101714.html

（1）秘钥空间不同

Petya 中为了方便用户输入，字符必须从数字和大小写字母中选取，定义了 54 种有效字符：

```
geneSet := "123456789abcdefghijkmnopqrstuvwxABCDEFGHJKLMNPQRSTUVWX"
```

来作为 8 位的原始秘钥，同时用低位为 b 与字符“z”对应 ASCII（122）之和，高位为 b*2 来扩展成 16 字节的秘钥。其实只有 8 个秘钥需要破解，所以秘钥空间为：54^8。

NotPetya 中，一共是 32 个字节，但是由于清零了一半，所以一共是 16 个字节需要破解，秘钥空间为 2^128。

（2）输出数据不同

Petya 中采用了 2 字节的 WORD 作为数据基本长度，在输出结果中字段从 2 字节扩展为 4 个字节，其高位 WORD 会被填充为 0，在接下来的异或操作中，就会暴露出明文的特征。

NotPetya 中在核心函数中保持着 4 个字节的基本长度，所以输出结果的高位不会被填充为零。可以正常加密。

综上两点区别，Petya 可以被暴力破解，而 NotPetya 很难被暴力破解，Petya 的具体破解代码：

https://github.com/leo-stone/hack-petya

这里的破解算法引用了第三方库，

"github.com/handcraftsman/GeneticGo"

"github.com/willf/bitset"

0x06 其他破解的可能性

（1）截断差分攻击

这种攻击是针对较少轮次的 Salsa20，参考资料认为能攻击到 8 轮

的 Salsa20，样本进行了 20 轮，所以这种攻击实现的可能性小。

（2）滑动攻击

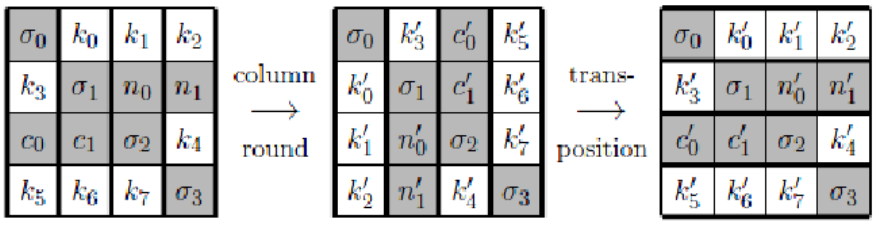

图 4.4 两个初始状态的关系

由于我们能够恢复所有的内部状态，这种攻击也能对 Salsa20 施行相关密钥恢复攻击，因为密钥是直接包含在内部状态中的。

如表 4.1 所示为滑动攻击与随机预言机的时间复杂度比较：

表 4.1 滑动攻击与随机预言机的时间复杂度比较

初始状态已知 word	Salsa20 的滑动攻击	随机猜测
一无所知	$O(2^{66})$	$O(2^{511})$
对角线	$O(2^{34})$	$O(2^{255})$
对角线，n'，c'	$O(2^{33})$	$O(2^{159})$
对角线，n，c	$O(1)$	$O(2^{127})$
对角线，n，c，n'，c'	$O(1)$	$O(2^{63})$

这使得破解 Salsa20 在理论上存在可能，参考文献里也给了具体的算法来计算。

0x07 参考文档

[1] www.freebuf.com/vuls/101714.html

[2]穆昭薇. 流密码算法 Salsa20 的安全性研究[D].西安电子科技大学,2011.