3. Oplossing

Er wordt gebruik gemaakt van HMAC-SHA-512. Een MAC functie zoals deze heeft twee parameters: een key en een message. In hesperus is de key een combinatie van verschillende systeem-specifieke variabelen zoals de computernaam en (de 4 byte CRC van) verschillende registry waarden. Al deze variabelen worden geplaatst in een blok van in totaal 31 bytes dat als geheel fungeert als de key van de MAC. Als message wordt een blok van 64 pseudo-random bytes gebruikt, dat gegenereerd wordt door een pseudo-random number generator.

De berekening van de MAC geschiedt in een lus (loop) waarbij de reeds berekende waarden telkens verder worden gemuteerd. Een beschrijving van deze en andere bijzonderheden volgt.

De key is een datastructuur van 31 bytes die er als volgt uitziet:

```
Byte 0 – 14: Computernaam (verkregen door GetComputerName, overige bytes 0x00)
Byte 15 – 18 Installatietijdstip als 32-bit UNIX timestamp
Byte 19 – 22 sub_40D607 | sub_40D434 (zie [1])
Byte 23 – 26 4 byte CRC van UNICODE-gecodeerde
      HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Cryptography\MachineGuid
Byte 27 – 30 4 byte CRC van HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows
      NT\CurrentVersion\DigitalProductId
```

[1] Deze functies genereren een waarde op basis van de waarden van SystemInfo.dwOemId (GetNativeSystemInfo) en VersionInformation.dwMajorversion / VersionInformation.dwMinorVersion (GetVersionEx)

De message is de output van een pseudo-random number generator met als initialisatiewaarde 0xFA57A170, resulterend in de volgende 64 bytes:

```
EB 15 2A 41 44 43 87 B9
CD B1 62 F9 15 54 9D 74
20 92 09 65 A6 60 17 E6
70 49 17 0A DB 89 B4 94
FF 91 05 BD A9 C4 4A E0
19 4D A6 50 87 EC 6C 60
1C 03 77 F1 17 FE AE 47
7E BD 02 25 8D 65 0D 22
```

De generatie van de werkelijke sleutel ziet er als volgt uit (pseudocode):

```
H1 = HMAC-SHA-512(key, message + 0x00 + 0x00 + 0x00 + 0x01)
H2 = H1

HERHAAL 1023 KEER:
      H1 = HMAC-SHA-512(KEY, H1)
      H2 = H2 XOR H1

SLEUTEL IS H2
```

In bijgevoegd bestand hesperus_key_generator.py staan alle functies uitgeschreven in Python code.

4. Aanpak

4.1 Lokalisatie van relevante functies

Als eerste ben ik begonnen met het maken van een disassembly en een decompilatie naar C code gebruikmakend van IDA Pro. Hoewel ik tengevolge van onder andere de ontwikkeling van mijn x86 emulator myrrh ruime ervaring heb met binaire code en assemblycode, heb ik nooit veel gebruikgemaakt van IDA Pro. Ten einde niet te veel tijd te willen verliezen met het doorgronden van de functionaliteit van deze software, ben ik na het genereren van de disassembly en decompilatie begonnen met het lezen hiervan.

Wat me al snel opviel waren bepaalde willekeurig ogende 32 bit hexadecimale waarden, met name in sub_4020CC:

```
sub_4020CC proc   near
xor   ecx, ecx
mov   [eax+48h], ecx
mov   [eax+40h], ecx
mov   [eax+44h], ecx
mov   dword ptr [eax], 0F3BCC908h
mov   dword ptr [eax+4], 6A09E667h
mov   dword ptr [eax+8], 84CAA73Bh
mov   dword ptr [eax+0Ch], 0BB67AE85h
mov   dword ptr [eax+10h], 0FE94F82Bh
mov   dword ptr [eax+14h], 3C6EF372h
mov   dword ptr [eax+18h], 5F1D36F1h
mov   dword ptr [eax+1Ch], 0A54FF53Ah
mov   dword ptr [eax+20h], 0ADE682D1h
mov   dword ptr [eax+24h], 510E527Fh
mov   dword ptr [eax+28h], 2B3E6C1Fh
mov   dword ptr [eax+2Ch], 9B05688Ch
mov   dword ptr [eax+30h], 0FB41BD6Bh
mov   dword ptr [eax+34h], 1F83D9ABh
mov   dword ptr [eax+38h], 137E2179h
mov   dword ptr [eax+3Ch], 5BE0CD19h
retn
sub_4020CC endp
```

Uit ervaring weet ik dat dergelijke waarden, zeker in een cluster zoals hierboven getoond, vaak verband houden met cryptografische functies of hash/CRC functies. Na het opzoeken van enkele van deze waarden in Google kwam ik al snel tot de ontdekking dat het hier om initialisatiewaarden van SHA-512 ging.

Omdat hesperus een groot programma is met vele functies en de aanroephierarchie hiervan lastig te doorgronden is door louter gebruik te maken van een text editor om de bronbestanden te lezen, besloot ik een Python script te schrijven dat deze taak vergemakkelijkt, zie bijgeleverd bestand path.py.

Dit script doet de door IDA Pro gegenereerde disassembly parsen en houdt bij welke functies welke andere functies aanroept. Zodoende kon ik, als ik bijvoorbeeld wilde weten vanwaaruit de hierboven genoemde functie sub_4020CC werd aangeroepen, dit script draaien met sub_4020CC als parameter en kreeg het antwoord op mijn vraag. Als het ware kon ik met mijn script op basis van de disassembly een boomstructuur van de aanroepsamenhang abstraheren.

```
$ ./path.py paths-from _hesperus_core_entry@4 | egrep "sub_4020CC"
_hesperus_core_entry@4-> sub_40142A-> sub_402354-> sub_4020CC|
_hesperus_core_entry@4-> sub_409518-> sub_408B01-> sub_40124E-> sub_40118A->
sub_4020CC|
```

sub_40142A heb ik onderzocht maar kon ik uiteindelijk afdoen als niet gerelateerd aan cryptografische functies (volgens mij is het een generator van een random number pool). Verder onderzoek naar sub_409518 volgt hieronder.

Tijdens het lezen van de broncode bespeurde ik in de datasectie van het programma een groot blok willekeurig ogende waarden die intensief worden gebruikt door een bepaalde functie.

```
db  86h ; <86>
db  56h ; V
db  55h ; U
db    9
db 0BEh ; <BE>
db  91h ; <91>
dword_414948 dd 0BCBC3275h
db 0F3h ; <F3>
db  21h ; !
db 0ECh ; <EC>
db 0ECh ; <EC>
db 0C6h ; <C6>
db  43h ; C
```

Na een waarde als 0xBCBC3275 te hebben gegoogled kwam ik uit bij broncodebestanden van de Twofish cipher. In verband met de vraagstelling was dit interessant voor me.

In mijn script path.py zit ook een optie om niet alleen alle codepaden vanuit een bepaalde functie (bijvoorbeeld _hesperus_core_entry@4) te ontdekken, maar ook om tegelijkertijd van iedere aangeroepen functie de assemblycode te dumpen.

```
$ ./path.py proc-content-from _hesperus_core_entry@4 | egrep "dword_414948" |
sort | uniq
sub_402386 - mov  edi, ds:dword_414948[edi*4]
sub_402386 - xor  edx, ds:dword_414948[edi*4]
sub_402386 - xor  esi, ds:dword_414948[edi*4]
sub_402386 - xor  esi, ds:dword_414948[edx*4]
```

Dus sub_402386 doet iets met de Twofish-gerelateerde waarde in dword_414948 (0xBCBC3275, zoals hierboven beschreven).

Als ik vervolgens het volgende commando geef:

```
$ ./path.py paths-from sub_409518 | egrep "sub_402386"
sub_409518-> sub_408BFA-> sub_409B60-> sub_406155-> sub_402386|
sub_409518-> sub_408CF6-> sub_409AE4-> sub_4099EE-> sub_409B60-> sub_406155->
      sub_402386|
sub_409518-> sub_408E46-> sub_407E97-> sub_40B6BF-> sub_409AE4-> sub_4099EE->
      sub_409B60-> sub_406155-> sub_402386|
sub_409518-> sub_408E46-> sub_4099D2-> sub_407E97-> sub_40B6BF-> sub_409AE4->
      sub_4099EE-> sub_409B60-> sub_406155-> sub_402386|
```

kan ik concluderen dat sub_409518 iets doet met zowel de SHA-512 initialisatieroutine als de Twofish-gelateerde functie.

Gebruikmakend van de hierboven beschreven methodiek om samenhang tussen functies duidelijk te maken wist ik uiteindelijk de functie sub_408B01 te identificeren als de routine waarbinnen de uiteindelijke sleutel wordt gegenereerd.

```
$ ./path.py paths-from _hesperus_core_entry@4 | grep sub_408B01
_hesperus_core_entry@4-> sub_409518-> sub_408B01-> sub_40124E-> sub_40118A->
      sub_4020CC|
_hesperus_core_entry@4-> sub_409518-> sub_408B01-> sub_40124E-> sub_40118A->
      sub_4021CC-> sub_401568|
_hesperus_core_entry@4-> sub_409518-> sub_408B01-> sub_40124E-> sub_40118A->
      sub_402147-> sub_401568|
_hesperus_core_entry@4-> sub_409518-> sub_408B01-> sub_40124E-> sub_40118A->
      sub_402147-> sub_40BB15|
_hesperus_core_entry@4-> sub_409518-> sub_408B01-> sub_40D18D|
_hesperus_core_entry@4-> sub_409518-> sub_408B01-> sub_40D249-> sub_40BAD5|
_hesperus_core_entry@4-> sub_409518-> sub_408B01-> sub_40D249-> sub_40D292->
      sub_40D1EF|
_hesperus_core_entry@4-> sub_409518-> sub_408B01-> sub_40D249-> sub_40D292->
      sub_40BAD5|
_hesperus_core_entry@4-> sub_409518-> sub_408B01-> sub_40D249-> sub_40D292->
      sub_40E044|
_hesperus_core_entry@4-> sub_409518-> sub_408B01-> sub_40D249-> sub_40D292->
      sub_40BA53|
_hesperus_core_entry@4-> sub_409518-> sub_408B01-> sub_40D249-> sub_40BF11->
      sub_40BB5F|
_hesperus_core_entry@4-> sub_409518-> sub_408B01-> sub_40D249-> sub_40BF11->
      sub_40BAD5|
_hesperus_core_entry@4-> sub_409518-> sub_408B01-> sub_40D249-> sub_40BF11->
      sub_40BA53|
_hesperus_core_entry@4-> sub_409518-> sub_408B01-> sub_40D607|
_hesperus_core_entry@4-> sub_409518-> sub_408B01-> sub_401494|
_hesperus_core_entry@4-> sub_409518-> sub_408B01-> sub_40BB29|
_hesperus_core_entry@4-> sub_409518-> sub_408B01-> sub_40D434|
```

sub_408B01 wordt maar een maal aangeroepen, vanuit sub_409518, met als parameter unk_417C94:

```
sub_408B01((int)&unk_417C94);
```

Vervolgens kijk ik vanwaaruit unk_417C94 wordt benaderd:

```
$ ./path.py proc-content-from _hesperus_core_entry@4 | grep unk_417C94 | sort |
uniq
sub_409518 - push offset unk_417C94
sub_409B60 - push offset unk_417C94
sub_409B82 - push offset unk_417C94

$ ./path.py paths-from sub_409B60
sub_409B60-> sub_406155-> sub_402386|

sub_409B60-> sub_406155-> sub_404F7C|

sub_409B60-> sub_406155-> sub_40BAD5|

sub_409B60-> sub_406155-> sub_40BA53|

$ ./path.py paths-from sub_409B82
sub_409B82-> sub_4061A6-> sub_402386|

sub_409B82-> sub_4061A6-> sub_40584F|

sub_409B82-> sub_4061A6-> sub_40BAD5|

sub_409B82-> sub_4061A6-> sub_40BA53|
```

Hier zien we sub_402386 weer. Er blijkt dat vanuit sub_409B60 unk_417C94 wordt meegegeven als parameter aan sub_406155, en vervolgens wordt doorgegeven aan de Twofish-gerelateerde functie sub_402386. Een andere parameter van sub_402386 is een heap-allocated geheugengebied dat weer wordt doorgegeven aan sub_404F7C:

```
signed int __usercall sub_406155<eax>(unsigned int a1<eax>, int a2, int a3)
{
  signed int v3; // ebx@1
  void *v4; // edi@1
  unsigned int v5; // esi@1
  int v6; // ebx@2
  unsigned int v7; // esi@2

  v5 = a1;
  v3 = 0;
  v4 = sub_40BA53(0x10A0u); // HEAP ALLOC
  if ( v4 )
  {
    v6 = a3;
    v7 = v5 >> 4;
    sub_402386(a2, (int)v4);
    for ( ; v7; --v7 )
    {
      sub_404F7C((int)v4, v6, v6);
      v6 += 16;
    }
    v3 = 1;
    sub_40BAD5(v4); // HEAP FREE
  }
  return v3;
}
```

sub_4061A6 is vergelijkbaar:

```
signed int __usercall sub_4061A6<eax>(unsigned int a1<eax>, int a2, int a3)
{
  signed int v3; // ebx@1
  void *v4; // edi@1
  unsigned int v5; // esi@1
  int v6; // ebx@2
  unsigned int v7; // esi@2

  v5 = a1;
  v3 = 0;
  v4 = sub_40BA53(0x10A0u);
  if ( v4 )
  {
    v6 = a3;
    v7 = v5 >> 4;
    sub_402386(a2, (int)v4);
    for ( ; v7; --v7 )
    {
      sub_40584F((int)v4, v6, v6);
      v6 += 16;
    }
    v3 = 1;
    sub_40BAD5(v4);
  }
  return v3;
}
```

Hieronder volgt de routine te zien die het bestand leest en vervolgens decodeert gebruikmakend van de Twofish routines.

```
signed int __cdecl sub_40B295(signed int a1, int a2, int a3, int a4)
{
  signed int v4; // ebx@1
  void *v5; // esi@3
  char v6; // al@4
  int v7; // edi@6
  LPVOID v8; // ebx@8
  unsigned int v9; // edi@9
  unsigned int v10; // eax@10
  int v11; // ecx@14
  LPVOID lpMem; // [sp+4h] [bp-8h]@6
  int v14; // [sp+8h] [bp-4h]@6

  v4 = 0;
  if ( a3 )
  {
    if ( a4 )
    {
      v5 = sub_40B0EA(a1, a2);
      if ( v5 )
      {
        v6 = 0;
        if ( a2 & 0x20000 )
          v6 = 2;
        // sub_40CC8A: ReadFile()
        v7 = sub_40CC8A((const WCHAR *)v5, (int)&lpMem, (int)&v14, v6);
        sub_40BAD5(v5);
        if ( v7 )
        {
          if ( v14 & 0xF )
```

```
          {
            sub_40BAD5(lpMem);
          }
          else
          {
            v8 = lpMem;
            if ( a2 & 0x10000 )
            {
              v9 = 0;
              while ( 1 )
              {
                v10 = v14 - v9;
                if ( v14 - v9 > 0x8000 )
                  v10 = 32768;
                // sub_409B82 > sub_409B82 > sub_4061A6 > TWOFISH
                sub_409B82((int)((char *)v8 + v9), v10);
                v9 += 32768;
                if ( v9 >= v14 )
                  break;
                Sleep(1u);
              }
            }
            v11 = v14;
            *(_DWORD *)a3 = v8;
            *(_DWORD *)a4 = v11;
            v4 = 1;
          }
        }
      }
    }
  }
  return v4;
}
```

In de vraagstelling wordt gesproken van bestanden. Ik kon de bovenstaande routines lokaliseren door af te gaan op twee strings die ik vond in de datasectie:

```
a_dat:
unicode     0, <.dat>,0
align 10h
a_bkp:
unicode     0, <.bkp>,0
align 4
```

Vervolgens:

```
$ ./path.py proc-content-from StartAddress | egrep "\.bkp|\.dat" | sort | uniq
sub_40AFBA - push offset a_dat      ; ".dat"
sub_40B05F - push offset a_bkp      ; ".bkp"

$ ./path.py proc-content-from StartAddress | egrep "ReadFile" | sort | uniq
sub_40CC8A - call ds:ReadFile
```

En ik kwam erachter dat beide functies direct of indirect worden aangeroepen vanuit sub_40B295:

```
...
...
v5 = CORE_sub_40B0EA(a1, a2);
if ( v5 )
{
      v6 = 0;
      if ( a2 & 0x20000 )
            v6 = 2;
      v7 = CORE_sub_40CC8A((const WCHAR *)v5, (int)&lpMem, (int)&v14, v6);
...
...
```

4.2 Doorgronden van de key generator

Toen ik de functie die verantwoordelijk is voor het maken van de cryptografische sleutel eenmaal gevonden, heb ik met behulp van zowel de decompilation en disassembly als OllyDBG in een Windows virtual machine elke stap nauwkeurig bestudeerd en nagemaakt als een Python script. Door de code in hesperus handmatig te reproduceren als een Python script kon ik zeker zijn dat ik niets over het hoofd zou zien, want als dat het geval was zou de uiteindelijke sleutel gemaakt door het script niet overeenkomen met die gegenereerd door de hesperus binary.

Globaal staan de operaties die moeten worden uitgevoerd om te sleutel te verkrijgen al beschreven in hoofdstuk 3 van dit document. In detail staan deze uitgeschreven in hesperus_key_generator.py.

Hieronder is de output van hesperus_key_generator.py te zien. De voor de generator relevante waarden (zoals installatietijdstip en registrywaarden) van mijn eigen Windows virtual machine zijn hardcoded in het script.

```
$ ./hesperus_key_generator.py

Hesperus key generator
----------------------

Guido Vranken <guidovranken@gmail.com>

Computername: COMPUTER_1

DigitalProductID:

    A4 00 00 00 03 00 00 00 35 35 32 37 34 2D 36 34 30 2D 38 33 36 35 33 39 31
2D 32 33 34 36 33 00 2E 00 00 00 41 32 32 2D 30
    30 30 30 31 00 00 00 00 00 00 00 76 C7 64 CC 82 46 03 68 EB 53 4A 11 A5 85
03 00 00 00 00 00 EA 02 6B 52 4F 87 00 00 00 00
    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 37 31 38
30 33 00 00 00 00 00 00 00 B8 03 00 00 B9 E9 3F
    B4 00 02 00 00 6A 3E 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 8E 47 B6 08

DigitalProductID CRC: 0x68B716AA
MachineGuid: dac8ca71-6517-40b6-92d3-e7fc27709c0b
```

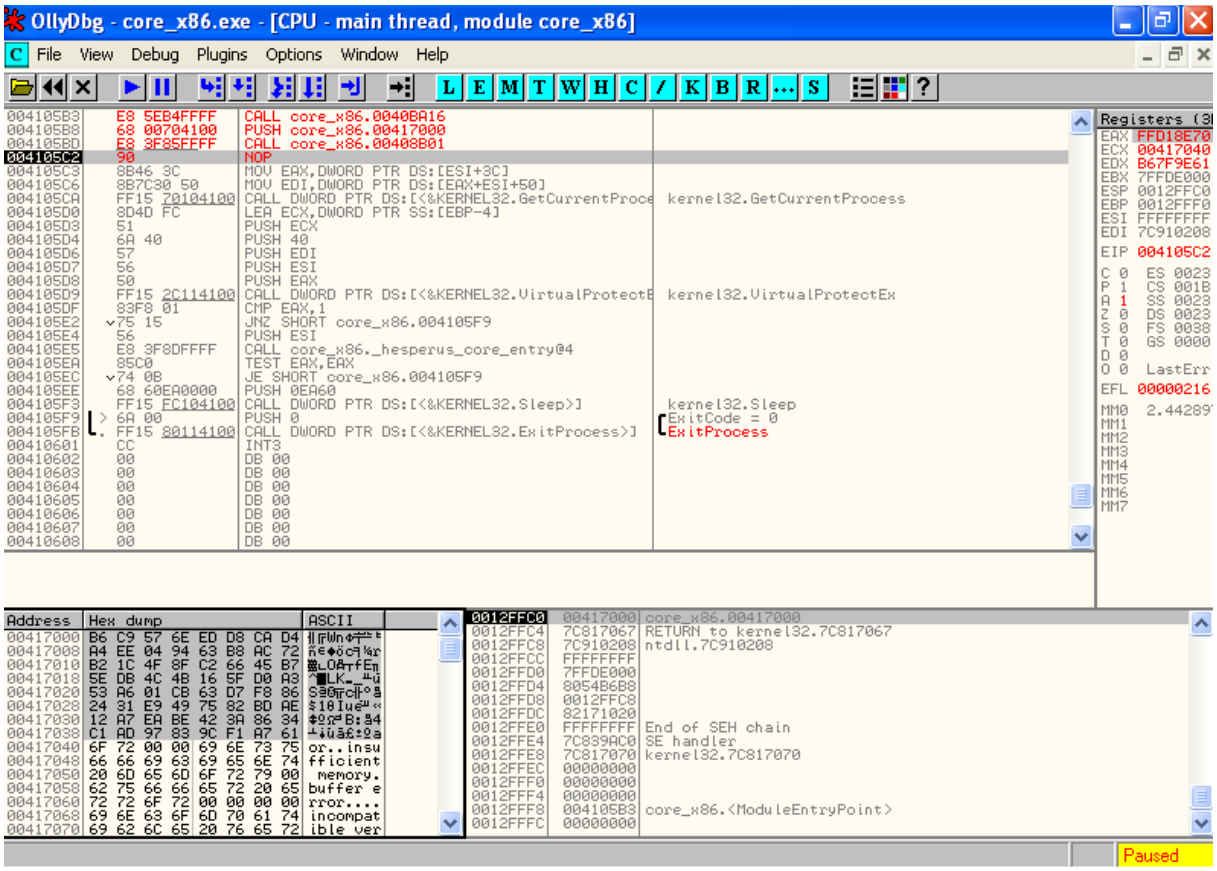
```
MachineGuid CRC: 0x76090974
dwMajorVersion: 0x00000005
dwMinorVersion: 0x00000001
dwOemId: 0x00000000
Installation time: 26-10-2013 08:54:28
Installation time UNIX timestamp: 0x526B6724

PRNG output:

    EB 15 2A 41 44 43 87 B9
    CD B1 62 F9 15 54 9D 74
    20 92 09 65 A6 60 17 E6
    70 49 17 0A DB 89 B4 94
    FF 91 05 BD A9 C4 4A E0
    19 4D A6 50 87 EC 6C 60
    1C 03 77 F1 17 FE AE 47
    7E BD 02 25 8D 65 0D 22


hesperus crypt key:

    B6 C9 57 6E ED D8 CA D4
    A4 EE 04 94 63 B8 AC 72
    B2 1C 4F 8F C2 66 45 B7
    5E DB 4C 4B 16 5F D0 A3
    53 A6 01 CB 63 D7 F8 86
    24 31 E9 49 75 82 BD AE
    12 A7 EA BE 42 3A 86 34
    C1 AD 97 83 9C F1 A7 61
```



Output van key generation functie (core_x86.0x408B01): datablok van 0x417000 – 0x41703F; hetzelfde als de output van het script.