Turla PNG Dropper is back

research.nccgroup.com/2018/11/22/turla-png-dropper-is-back/

November 22, 2018

This is a short blog post on the PNG Dropper malware that has been developed and used by the Turla Group [1]. The PNG Dropper was first discovered back in August 2017 by Carbon Black researchers. Back in 2017 it was being used to distribute Snake, but recently NCC Group researchers have uncovered samples with a new payload that we have internally named RegRunnerSvc.

It's worth noting at this point that there are other components to this infection that we have not managed to obtain. There will be a first stage dropper that will drop and install the PNG Dropper/RegRunnerSvc. Nevertheless, we think that this it is worth documenting this new use of the PNG Dropper.

PNG Dropper

The PNG Dropper component has already been well documented by the research team at Carbon Black [1], but for the purpose of clarity we will now give a quick summary of what it is and how it works.

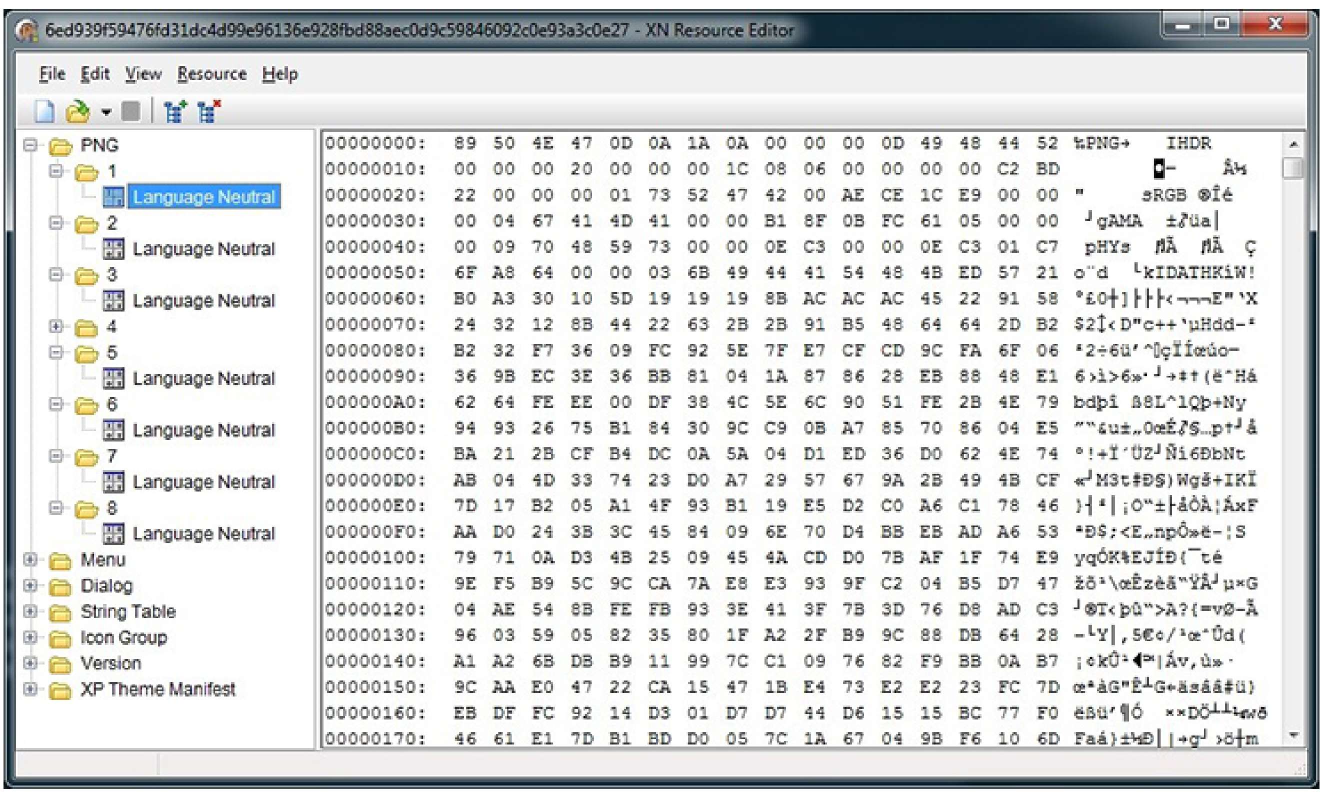

Figure 1

The purpose of the dropper is to load and run a PE file that is hidden in a number of PNG files. Figure 1 shows the resources of the dropper. Here you can see a number binary data resource entries under the name “PNG”. Each of these resources is a valid PNG file which can be viewed with any image viewer, but upon opening one you will only see a few coloured pixels (see an enlarged version in Figure 2).

Figure 2

The PNG is loaded using Microsoft’s GDI+ library. In Figure 3 we see a call to LockBits which is used to read the pixel data from the PNG file. Each byte in the pixel data represents an RGB value for a pixel. Encoded in each of the the RGB values is a byte from a PE file. It doesn’t make for a very meaningful image, but it is a novel way to hide data in seemingly innocent resources.

```
.text:000000013F700D26 8B D0                mov     edx, eax        ; dwSize
.text:000000013F700D28 49 89 17             mov     [r15], rdx
.text:000000013F700D2B 33 C9                xor     ecx, ecx        ; lpAddress
.text:000000013F700D2D 44 8D 49 40          lea     r9d, [rcx+40h]  ; flProtect
.text:000000013F700D31 41 B8 00 10 00 00    mov     r8d, 1000h      ; flAllocationType
.text:000000013F700D37 FF 15 CB 06 FF FF    call    cs:VirtualAlloc
.text:000000013F700D3D 4C 8B 6D 6F          mov     r13, [rbp+57h+arg_8]
.text:000000013F700D41 49 89 45 00          mov     [r13+0], rax
.text:000000013F700D45 48 85 C0             test    rax, rax
.text:000000013F700D48 0F 84 EC 00 00 00    jz      loc_13F700E3A
.text:000000013F700D4E 48 89 45 FF          mov     [rbp+57h+var_58], rax
.text:000000013F700D52 48 8D 45 EF          lea     rax, [rbp+57h+var_68]
.text:000000013F700D56 48 89 44 24 20       mov     [rsp+0C0h+var_A0], rax
.text:000000013F700D5B 41 B9 07 10 06 00    mov     r9d, 61007h
.text:000000013F700D61 41 B8 07 00 00 00    mov     r8d, 7
.text:000000013F700D67 48 8D 55 D7          lea     rdx, [rbp+57h+var_80]
.text:000000013F700D6B 48 8B 4F 08          mov     rcx, [rdi+8]
.text:000000013F700D6F E8 D0 9C 01 00       call    GdipBitmapLockBits
.text:000000013F700D74 8B F0                mov     esi, eax
.text:000000013F700D76 85 C0                test    eax, eax
.text:000000013F700D78 74 05                jz      short loc_13F700D7F
.text:000000013F700D7A 89 47 10             mov     [rdi+10h], eax
.text:000000013F700D7D EB 02                jmp     short loc_13F700D81
```

Figure 3

Each PNG resource is enumerated and the pixel data is extracted and then concatenated together. The result is an entire PE file contained in memory. The dropper will then manually load the PE file. The imports are processed, as are the relocations. Finally the PE file’s entry point is executed (as shown in Figure 4).

```
.text:000000013F700FE4 48 63 43 3C          movsxd  rax, dword ptr [rbx+3Ch]
.text:000000013F700FE8 B9 0B 01 00 00       mov     ecx, 10Bh
.text:000000013F700FED BA AF BE AD DE       mov     edx, 0DEADBEAFh
.text:000000013F700FF2 66 39 4C 18 18       cmp     [rax+rbx+18h], cx
.text:000000013F700FF7 8B 44 18 28          mov     eax, [rax+rbx+28h]
.text:000000013F700FFB 45 33 C9             xor     r9d, r9d
.text:000000013F700FFE 44 8B C2             mov     r8d, edx
.text:000000013F701001 48 8B CB             mov     rcx, rbx
.text:000000013F701004 48 03 C3             add     rax, rbx
.text:000000013F701007 FF D0                call    rax
.text:000000013F701009 48 8B 5C 24 60       mov     rbx, [rsp+68h+var_8]
.text:000000013F70100E 48 8B 7C 24 48       mov     rdi, [rsp+68h+var_20]
.text:000000013F701013 48 83 C4 68          add     rsp, 68h
.text:000000013F701017 C3                   retn
```

Figure 4

RegRunnerSvc

The PNG dropper will decode and run RegRunnerSvc from its PNG resources. The purpose of RegRunnerSvc is to extract an encrypted payload from the registry, load it into memory, and then run it. A first stage dropper (which we have not managed to obtain) will have already installed it as a service and performed a few additional setup operations.

Figure 5 shows the entry point for RegRunnerSvc. Here we can see the call to StartServiceCtrlDispatcher. In this case the name of the service is WerFaultSvc, obviously chosen in an attempt to seem like a legitimate part of the Windows Error Reporting service. The service also serves as a persistence mechanism for the malware.

```
push rbx                                      EntryPoint
sub rsp,40
mov rbx,rcx                                   rcx:&"WerFaultSvc"
mov ecx,8007                                  ecx:&"WerFaultSvc"
call qword ptr ds:[<&SetErrorMode>]
cmp byte ptr ds:[1400040FF],0
mov qword ptr ds:[140004138],rbx
lea r8,qword ptr ds:[140004080]               r8:"WerFaultSvc", 0000000140004080:"WerFaultSvc"
je 6e.140001BA2
mov rcx,r8                                    rcx:&"WerFaultSvc", r8:"WerFaultSvc"
mov edx,20                                    20:' '
movzx eax,byte ptr ds:[rcx-80]
add rcx,4                                     rcx:&"WerFaultSvc"
xor byte ptr ds:[rcx-4],al
movzx eax,byte ptr ds:[rcx-83]
xor byte ptr ds:[rcx-3],al
movzx eax,byte ptr ds:[rcx-82]
xor byte ptr ds:[rcx-2],al
movzx eax,byte ptr ds:[rcx-81]
xor byte ptr ds:[rcx-1],al
sub rdx,1
jne 6e.140001B73
lea rax,qword ptr ds:[140001A80]
lea rcx,qword ptr ss:[rsp+20]                 [rsp+20]:"WerFaultSvc"
mov qword ptr ss:[rsp+20],r8                  [rsp+20]:"WerFaultSvc"
mov qword ptr ss:[rsp+28],rax
xor eax,eax
mov qword ptr ss:[rsp+30],rax
mov qword ptr ss:[rsp+38],rax
call qword ptr ds:[<&StartServiceCtrlDispatcherA>]
xor eax,eax
add rsp,40
pop rbx
ret
```

Figure 5

After the service setup functions has been executed, it is time to find the data in the registry. Generally the path to the registry value would be stored as a (possibly encrypted/obfuscated) string within the binary, but interestingly this is not the case here. The registry keys and values are enumerated using the RegEnumKeyExA and RegEnumValueA functions. The enumeration starts at the root of the HKEY_LOCAL_MACHINE key and continues using a depth first search until either the data is found or the enumeration is exhausted. Another interesting implementation detail (shown in Figure 6), is that the only requirement for decryption function to be called is that the size of the value data is 0x200 (512) bytes in size. This is not as inefficient as it may first seem as the decryption function will exit relatively quickly if the first stage dropper has not performed its setup operations. Nevertheless it’s clear that for the malware authors, obfuscation is more important than efficiency.

Figure 6

The data in the registry contains the encrypted payload and the data required to decrypt it. It doesn't contain the decryption key, but it does contain data that is used to generate the key. This data, however, is itself partially encrypted using the Microsoft CNG library functions (NCrypt*). The first stage dropper will have generated a decryption key and stored it in the one of the system default key storage providers, in this case the "Microsoft Software Key Storage Provider". If the first stage dropper has not run, then the key will not be in the storage provider, and the decryption function will exit. Provided that the storage provider actually contains a key, the first 0x200 (512) bytes of the data will be decrypted. This decrypted data contains a header that contains the information needed to locate the rest of the data in the binary blob (full description of the header can be found in Table 1).

Offset	Description
0x00	Offset to secret data - used in call to the BCryptGenerateSymmetricKey() function
0x08	Size of secret data
0x10	Offset to IV
0x18	IV size
0x20	Offset to AES encrypted data

0x28	Encrypted data size

Table 1

Now the header has been decrypted, the second part of the decryption can take place. The main payload is encrypted using the AES algorithm. First a chunk of data from the registry is passed to the BCryptGenerateSymmetricKey function, which results in the AES decryption key being created. Once the key has been generated and the decryption properties have been set, the payload will be decrypted. The decrypted payload is then checked to ensure that it’s a valid PE file (it checks for the MZ & PE magic bytes, and also checks for the machine architecture entry in the PE header). If the checks pass, the file is manually loaded (imports and relocations) and the entry point is called (as shown in Figure 7).

```
memcpy(v3, v4, v5);
memcpy(v3, v8, v9);
v12 = aes_decryption(
        (__int64)v8 + *v8,        // secret key data
        v8[1],                    // secret key data size
        (__int64)v8 + v8[2],      // IV
        v8[3],                    // IV size
        (__int64)v3 + (unsigned int)v3[4],// encrypted data
        v3[5],                    // encrypted data size
        (__int64)&p_decrypted_data,
        (__int64)&decrypted_data_size);
free(Src);
Src = 0i64;
if ( !v12
  && !(unsigned int)is_pe_file(
                      (__int64)p_decrypted_data + *(unsigned int *)p_decrypted_data,
                      *((unsigned int *)p_decrypted_data + 1)) )
{
  v18 = (char *)p_decrypted_data + *((unsigned int *)p_decrypted_data + 2);
  v19 = *((_DWORD *)p_decrypted_data + 3);
  v20 = (char *)p_decrypted_data + *((unsigned int *)p_decrypted_data + 4);
  v21 = *((_DWORD *)p_decrypted_data + 5);
  v6 = load_and_exec_pe((char *)p_decrypted_data + *(unsigned int *)p_decrypted_data, 0i64, &v18);
  free(p_decrypted_data);
  p_decrypted_data = 0i64;
  if ( !v6 )
    break;
}
```

Figure 7

Summary

In this blog post we have had a quick look at a new use of the PNG Dropper by the Turla Group. The group is now using it with a new component: RegRunnerSvc, which extracts and encrypted PE file from the registry, decrypts it and runs it. It seems that the group is taking ideas from fileless malware, such as Poweliks or Kovter. The group is ensuring that it is leaving as little information as possible in the binary files, i.e. not hardcoding the name of the registry key containing the encrypted data. This means that that it is not possible to extract useful IOCs for threat hunting.

Thankfully all is not lost and we can at least detect the usage of the PNG dropper using the Yara rules below.

As part of our research we created a tool that will extract the payload from the PNG Dropper. We have decided to release this tool just in case others find it useful. It can be found here: https://github.com/nccgroup/Cyber-Defence/tree/master/Scripts/turla_image_decoder.

Yara Rules

```
rule turla_png_dropper {
    meta:
        author = "Ben Humphrey"
        description = "Detects the PNG Dropper used by the Turla group"
        sha256 =
"6ed939f59476fd31dc4d99e96136e928fbd88aec0d9c59846092c0e93a3c0e27"

    strings:
        $api0 = "GdiplusStartup"
        $api1 = "GdipAlloc"
        $api2 = "GdipCreateBitmapFromStreamICM"
        $api3 = "GdipBitmapLockBits"
        $api4 = "GdipGetImageWidth"
        $api5 = "GdipGetImageHeight"
        $api6 = "GdiplusShutdown"

        $code32 = {
            8B 46 3C                // mov     eax, [esi+3Ch]
            B9 0B 01 00 00          // mov     ecx, 10Bh
            66 39 4C 30 18          // cmp     [eax+esi+18h], cx
            8B 44 30 28             // mov     eax, [eax+esi+28h]
            6A 00                   // push    0
            B9 AF BE AD DE          // mov     ecx, 0DEADBEAFh
            51                      // push    ecx
            51                      // push    ecx
            03 C6                   // add     eax, esi
            56                      // push    esi
            FF D0                   // call eax
        }

        $code64 = {
            48 63 43 3C             // movsxd rax, dword ptr [rbx+3Ch]
            B9 0B 01 00 00          // mov ecx, 10Bh
            BA AF BE AD DE          // mov edx, 0DEADBEAFh
            66 39 4C 18 18          // cmp [rax+rbx+18h], cx
            8B 44 18 28             // mov eax, [rax+rbx+28h]
            45 33 C9                // xor r9d, r9d
            44 8B C2                // mov r8d, edx
            48 8B CB                // mov rcx, rbx
            48 03 C3                // add rax, rbx
            FF D0                   // call rax
        }

    condition:
        (uint16(0) == 0x5A4D and uint16(uint32(0x3c)) == 0x4550) and
        all of ($api*) and
        1 of ($code*)
}
```

```
rule turla_png_reg_enum_payload {
      meta:
                author = "Ben Humphrey"
                description = "Payload that has most recently been dropped by the
Turla PNG Dropper"
               shas256 =
"fea27eb2e939e930c8617dcf64366d1649988f30555f6ee9cd09fe54e4bc22b3"

      strings:
          $crypt00 = "Microsoft Software Key Storage Provider" wide
          $crypt01 = "ChainingModeCBC" wide
          $crypt02 = "AES" wide

      condition:
          (uint16(0) == 0x5A4D and uint16(uint32(0x3c)) == 0x4550) and
          pe.imports("advapi32.dll", "StartServiceCtrlDispatcherA") and
          pe.imports("advapi32.dll", "RegEnumValueA") and
          pe.imports("advapi32.dll", "RegEnumKeyExA") and
          pe.imports("ncrypt.dll", "NCryptOpenStorageProvider") and
          pe.imports("ncrypt.dll", "NCryptEnumKeys") and
          pe.imports("ncrypt.dll", "NCryptOpenKey") and
          pe.imports("ncrypt.dll", "NCryptDecrypt") and
          pe.imports("ncrypt.dll", "BCryptGenerateSymmetricKey") and
          pe.imports("ncrypt.dll", "BCryptGetProperty") and
          pe.imports("ncrypt.dll", "BCryptDecrypt") and
          pe.imports("ncrypt.dll", "BCryptEncrypt") and
          all of them
}
```

IOCs

Sample Analysed

1. 6ed939f59476fd31dc4d99e96136e928fbd88aec0d9c59846092c0e93a3c0e27 (PNG Dropper)
2. fea27eb2e939e930c8617dcf64366d1649988f30555f6ee9cd09fe54e4bc22b3 (Payload contained in the PNG dropper)

Services

1. WerFaultSvc

References

[1] https://www.carbonblack.com/2017/08/18/threat-analysis-carbon-black-threat-research-dissects-png-dropper/

Published date: 22 November 2018

Written by: Ben Humphrey

Published by Matt Lewis

Call us before you need us.

Our experts will help you.

Get in touch