Operation TA505: how we analyzed new tools from the creators of the Dridex trojan, Locky ransomware, and Neutrino botnet

By Positive Technologies

Published: 2024-08-19 · Archived: 2026-04-05 13:17:09 UTC

Distribution of TA505 attacks in 2019

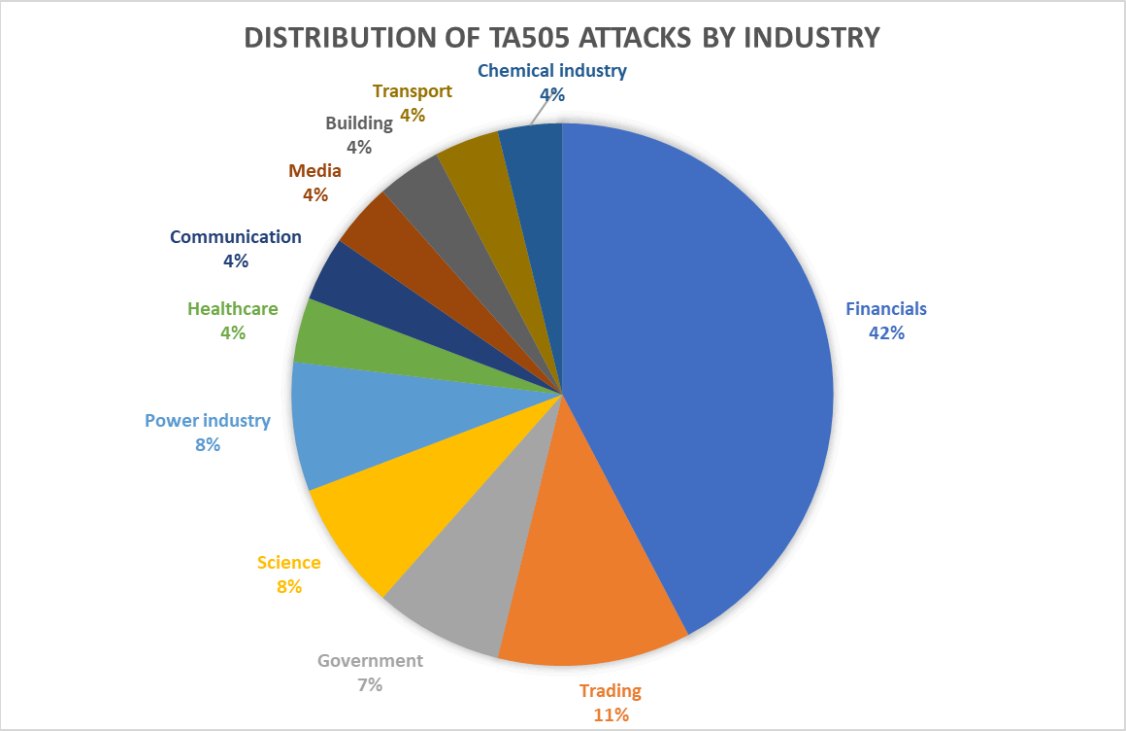
The Threat Intelligence team at the Positive Technologies Expert Security Center has been keeping a close eye on the TA505 cybercrime group for the last six months. The malefactors are drawn towards finance, with targets scattered in dozens of countries on multiple continents.

What is TA505 famous for?

The cybergang has been quite prolific since 2014: their arsenal includes the Dridex banking trojan, Neutrino botnet, as well as Locky, Jaff, GlobeImposter, and other ransomware.

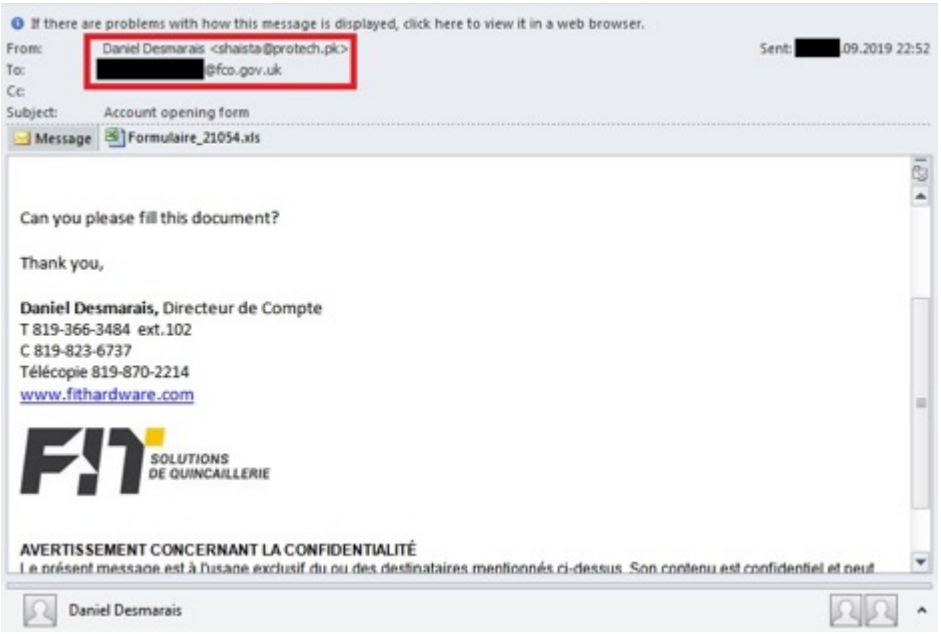
The group's attacks have been detected all around the world, from North America to Central Asia.

Despite being mainly motivated by profit, in the past six months they have also attacked research institutes, energy companies, healthcare institutions, airlines, and even government agencies.

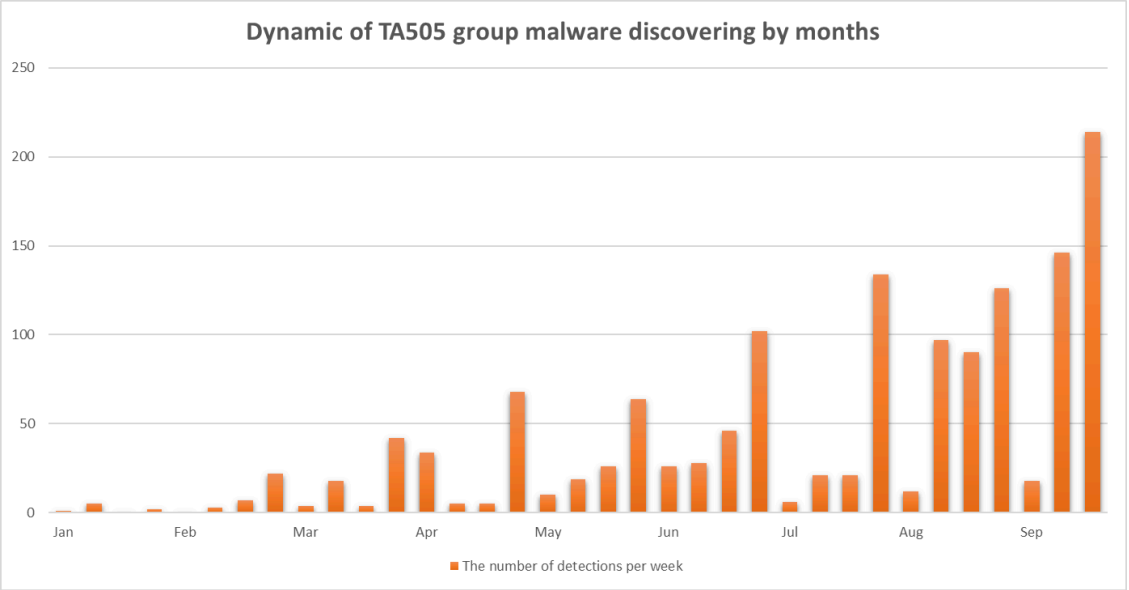


TA505 attacks by sector, 2019

Below is an example of a phishing message containing malware developed by the group. Judging by the email address, the attack targeted the British Foreign Office.



The group has been using the FlawedAmmyy remote access tool since spring 2018 and the new ServHelper backdoor since the end of 2018. TA505 is among the few groups that can boast of continuous activity over a long timeframe. Moreover, each new wave of attacks shows interesting changes in the group's tools.



TA505 detections by month, 2019

Such a high clip of attacks cannot stay invisible: our colleagues at companies including Proofpoint, Trend Micro, and Yoroi have already reported on the techniques and malicious software used by TA505. However, many intriguing issues still remain unaddressed:

- The PE packer unique to the group
- A version of the ServHelper backdoor that, instead of custom-developed functionality, relies on NetsupportManager remote control software
- Network infrastructure: registrars and hosting providers, including overlap with infrastructure of the Buhtrap group
- Other malware used by the group not covered previously

This is the first article of a series about the TA505 group.

Part 1. In the beginning was the packer

In mid-June 2019, we saw new variants of FlawedAmmy malware loaders with significant changes from previous versions. For example, the visual representation of code in hexadecimal was different. This pattern became a common theme in several samples that we analyzed.

ASCII code representation

Quick analysis showed that we were looking at an unknown packer of executable files. Later we found that this packer was used not only for the loaders in question, but for other TA505 malware, including payload. We decided to explore the unpacking logic in order to be able to automatically extract the contents.

Layer 1. Tricky XOR

The key portion of the unpacker is preceded by a large number of junk instructions. Malware developers often use this technique to evade antivirus emulators. The interesting part starts when 0xD20 of buffer memory is allocated using the WinAPI function VirtualAllocEx. Memory is allocated with PAGE_EXECUTE_READWRITE rights, which allow writing and executing code.

```
38  v13 = 1;
39  nShellcodeSize = 0xD20;
40  v9 = 56121;
41  v14 = 22953;
42  v17 = -350894538;
43  sub_401000(-350894538, -350894538, 22953);
44  sub_401000(-350894538, -350894538, 22953);
45  sub_401000(56121, aEndDataModeDat, 22953);
46  GetCommandLineA();
47  for ( i = 0; i < 3; ++i )
48  {
49    v27 = 8915;
50    v3 = 1218523844;
51  }
52  v3 = 225;
53  sub_401000(225, aRenegrightinth, 225);
54  aShellcode = (DWORD *)VirtualAllocEx((HANDLE)0xFFFFFFFF, 0, nShellcodeSize,
55  v18 = 210;
```

Start of the non-junk part of the unpacker

The data section of the file contains an array. The contents of the array are decoded and the result is written to the allocated memory. Here is the decoding process:

- Interpret 4 bytes as integer.
- Subtract the order number of the byte in the sequence.
- Perform XOR with a set constant.
- Perform a circular shift to the left by 7 positions.

- Perform XOR with set constant again.

```
71  v5 = 29651 * (343040786 - (v11 | 0x73D3));
72  aEncodedShellcode = &pEncodedShellcode;
73  v24 = 0;
74  for ( k = 0; k < nShellcodeSize >> 2; ++k )
75  {
76    nEncodedShellcode = aEncodedShellcode[k] - k;
77    v24 -= 80;
78    v24 -= 1000;
79    aShellcode[k] = nShellcodeMagic ^ __ROL4__(nShellcodeMagic ^ nEncodedShellcode, 7);
80  }
81  v30 = -123164427;
```

First-layer decoding

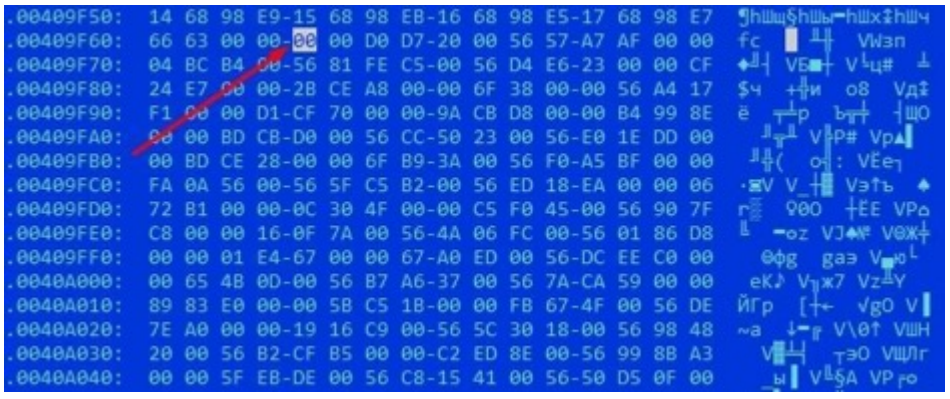
We'll call the algorithm **SUB-XOR-ROL7-XOR** when referring to it later.

Decoding is followed by sequential initialization of variables. This can be represented as declaring a C struct in the following format:

```
struct ZOZ {
    HMODULE hkernel32;
    void *aEncodedBlob;
    unsigned int nEncodedBlobSize;
    unsigned int nBlobMagic;
    unsigned int nBlobSize;
};
```

in which:

- *hkernel32* describes the library kernel32.dll.
- *aEncodedBlob* is a pointer to the encoded block of data we were talking about when noting the visual similarity of the samples.



Encoded data block

- *nEncodedBlobSize* is the 4-byte size of the encoded data block.

- *nBlobMagic* is a 4-byte constant ahead of the data block, to which we will return later on.
- *nBlobSize* is the 4-byte size of the decoded data block.

We called the struct **ZOZ** (or "505" in l33t speak).

```
for ( l = 0; l < 1; ++l )
  v29 -= v30 * v29;
pZOZStructure.hkernel32 = GetModuleHandleA(skernel32);
pZOZStructure.aEncodedBlob = (int)&pEncodedBlob;
pZOZStructure.nEncodedBlobSize = 0x18768;
pZOZStructure.nBlobMagic = nBlobMagic;
pZOZStructure.nBlobSize = nBlobSize;
for ( m = 0; m < 2; ++m )
```

Populating ZOZ

Code execution jumps to the decoded buffer (removing any doubt that the now-decoded data consists of executable code) and a pointer to the populated struct is passed in a function argument:

```
sub_401000(28882, aEvHmacOid80211, 50);
((void (__stdcall *)(ZOZ *))aShellcode)(&pZOZStructure);
GetCommandLineW();
```

Calling the decoded code, with the ZOZ struct passed as an argument

```
00000000: 55                    push    ebp
00000001: 8BEC                  mov     ebp,esp
00000003: 81ECD4000000          sub     esp,0000000D4 ;'   L'
00000009: C68570FFFFFF56        mov     b,[ebp][-000000090],056 ;'V'
00000010: C68571FFFFFF69        mov     b,[ebp][-00000008F],069 ;'i'
00000017: C68572FFFFFF72        mov     b,[ebp][-00000008E],072 ;'r'
0000001E: C68573FFFFFF74        mov     b,[ebp][-00000008D],074 ;'t'
00000025: C68574FFFFFF75        mov     b,[ebp][-00000008C],075 ;'u'
0000002C: C68575FFFFFF61        mov     b,[ebp][-00000008B],061 ;'a'
00000033: C68576FFFFFF6C        mov     b,[ebp][-00000008A],06C ;'l'
0000003A: C68577FFFFFF41        mov     b,[ebp][-000000089],041 ;'A'
00000041: C68578FFFFFF6C        mov     b,[ebp][-000000088],06C ;'l'
00000048: C68579FFFFFF6C        mov     b,[ebp][-000000087],06C ;'l'
0000004F: C6857AFFFFFF6F        mov     b,[ebp][-000000086],06F ;'o'
00000056: C6857BFFFFFF63        mov     b,[ebp][-000000085],063 ;'c'
0000005D: C6857CFFFFFF00        mov     b,[ebp][-000000084],0
00000064: C6855CFFFFFF47        mov     b,[ebp][-0000000A4],047 ;'G'
0000006B: C6855DFFFFFF65        mov     b,[ebp][-0000000A3],065 ;'e'
00000072: C6855EFFFFFF74        mov     b,[ebp][-0000000A2],074 ;'t'
00000079: C6855FFFFFFF50        mov     b,[ebp][-0000000A1],050 ;'P'
00000080: C68560FFFFFF72        mov     b,[ebp][-0000000A0],072 ;'r'
00000087: C68561FFFFFF6F        mov     b,[ebp][-00000009F],06F ;'o'
0000008E: C68562FFFFFF63        mov     b,[ebp][-00000009E],063 ;'c'
00000095: C68563FFFFFF41        mov     b,[ebp][-00000009D],041 ;'A'
0000009C: C68564FFFFFF64        mov     b,[ebp][-00000009C],064 ;'d'
000000A3: C68565FFFFFF64        mov     b,[ebp][-00000009B],064 ;'d'
000000AA: C68566FFFFFF72        mov     b,[ebp][-00000009A],072 ;'r'
000000B1: C68567FFFFFF65        mov     b,[ebp][-000000099],065 ;'e'
000000B8: C68568FFFFFF73        mov     b,[ebp][-000000098],073 ;'s'
000000BF: C68569FFFFFF73        mov     b,[ebp][-000000097],073 ;'s'
000000C6: C6856AFFFFFF00        mov     b,[ebp][-000000096],0
000000CD: C645B456              mov     b,[ebp][-04C],056 ;'V'
000000D1: C645B569              mov     b,[ebp][-048],069 ;'i'
000000D5: C645B672              mov     b,[ebp][-04A],072 ;'r'
000000D9: C645B774              mov     b,[ebp][-049],074 ;'t'
```

Decoded and disassembled code portion

Layer 2. Less is more

Once the portion of code is decoded and run, it starts gathering addresses of the WinAPI functions GetProcAddress, VirtualQuery, VirtualAlloc, VirtualProtect, VirtualFree, and LoadLibraryA. These functions are often used with shellcodes, in order to groom memory to run the payload.

When everything is ready, the encoded data block is passed and then "slimmed down." The first two of each five bytes are discarded and the remaining three are kept:

```
if ( result )
{
  i2 = 0;
  i1 = 0;
  while ( i2 < pZOZStructure->nEncodedBlobSize )
  {
    if ( !(i1 % 3) )
      i2 += 2;
    aBlob1[i1++] = pZOZStructure->aEncodedBlob[i2++];
  }
  nNewBlobSize = 3 * pZOZStructure->nEncodedBlobSize / 5u;
```

Reduction of the encoded data block

Then starts the decoding, which we have called **SUB-XOR-ROL7-XOR**. To perform XOR, the nBlobMagic value passed in **ZOZ** is used as a constant.

```
for ( j = 0; j < nNewBlobSize >> 2; ++j )
  aBlob1[j] = pZOZStructure->nBlobMagic ^ __ROL4__(pZOZStructure->nBlobMagic ^ (aBlob1[j] - j), 7);
```

Reuse of the SUB-XOR-ROL7-XOR algorithm

After that, the resulting array is passed to a function in which more complicated transformations take place. Judging by the characteristic constant values, we can easily identify a popular FSG (Fast Small Good) PE packer. Curiously enough, the original FSG packer version compresses PE by sections, whereas in our case the algorithm works with the PE as-is.

```
if ( v12 || v8 != 2 )
{
  if ( v12 )
    v8 -= 2;
  else
    v8 -= 3;
  v8 <<= 8;
  v8 += (unsigned __int8)*v4++;
  v9 = sub_A20(&v4);
  if ( v8 >= 0x7D00 )
    ++v9;
  if ( v8 >= 0x500 )
    ++v9;
  if ( v8 < 0x80 )
    v9 += 2;
  memset(v5, v5[-v8], v9);
  v5 += v9;
  v9 = 0;
  v10 = v8;
}
```

FSG packer implementation

At this stage, the memory contains the unpacked PE file ready for further analysis. The remaining part of the shellcode will overwrite the original PE in the address space with the unpacked version and will then run it correctly. Interestingly, during modification of the module entry point, there are manipulations involving PEB structures. We do not know why the attackers decided to forward the kernel32 descriptor from the first-layer logic instead of getting it with the help of the same PEB structures.

Entry point for the loaded module is overwritten in PEB

Conclusion

The payload is unpacked as follows:

- Decode shellcode with SUB-XOR-ROL7-XOR.
- Populate the ZOZ struct and call the shellcode.
- Slim payload (five to three).
- Decode payload with SUB-XOR-ROL7-XOR.
- Decompress with FSG packer.

As the malware evolved, so did its logic: the **SUB-XOR-ROL7-XOR** circular shift (in our case, by seven positions) has been changed to five and nine positions and an x64 packer version was released, among other changes. The cybergang's "calling card" packer is an excellent start to a series of upcoming tales about TA505 tools and techniques.

In future articles, we will discuss how the group's tools have changed during recent attacks and how its participants have interacted with other cybergroups. We will also explore malware samples not covered before.

Authors: Alexey Vishnyakov and Stanislav Rakovsky, Positive Technologies

IOCs

b635c11efdf4dc2119fa002f73a9df7b (packed FlawedAmmyy loader)

71b183a44f755ca170fc2e29b05b64d5 (unpacked FlawedAmmyy loader)

Source: https://www.ptsecurity.com/ww-en/analytics/pt-esc-threat-intelligence/operation-ta505/