TrueBot Analysis Part I - A short glimpse into packed TrueBot samples

By Robert Giczewski

Published: 2023-02-12 · Archived: 2026-04-05 20:26:56 UTC

In October 2022, Microsoft published a blog post about Raspberry Robin and it's role in the current cyber crime ecosystem. Microsoft reported, among other things, that they have observed Raspberry Robin delivering the well-known malware families IcedID, Bumblebee and TrueBot besides the already known delivery of FakeUpdates/SocGholish. At this time I was not really aware of TrueBot or I simply had forgotten about it.

In December 2022, Cisco Talos published a blog post in which they reported increased activity from TrueBot and mentioned that TrueBot might be related to TA505. They have observed TrueBot delivering Grace (aka FlawedGrace and GraceWire) as a follow-up payload, which is known to be exclusive tooling of TA505.

Since I have already analyzed some TA505 campaigns a few years ago and anything related to Raspberry Robin is of interest to me, TrueBot now had my attention and I finally found some time to take a closer look and here we are.

I have decided to start a small blog series that will cover the following points:

1. **Analyzing different packed samples and identifying decryption/unpacking code**
2. **How to statically unpack with Python using Malduck?**
3. **Analyzing TrueBot Capabilities**
4. **IOC/Config extraction with Python using Malduck**
5. **C2/Bot Emulation**
6. **Bonus (maybe): Infrastructure analysis**

The blog series is structured so that we gain the knowledge step by step to be able to take the next step.

In this first post, we'll look at some packed samples and gain enough knowledge to write a static unpacker in the next step.

Identifying decryption/unpacking code

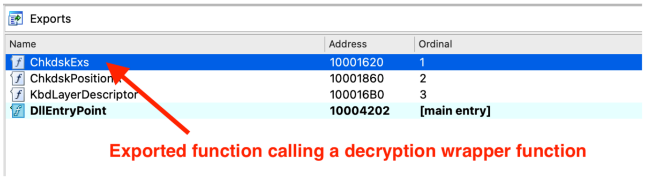
We are primarily looking at the packed samples that Talos also mentioned in their blog post including one sample that I have found on VirusTotal. All of these files are 32 Bit samples, mostly DLLs except for one sample which is a regular executable.

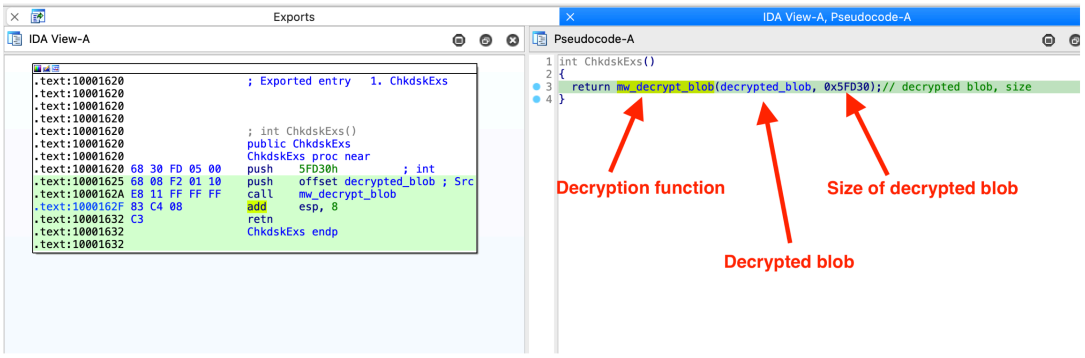
```
 092910024190a2521f21658be849c4ac9ae6fa4d5f2ecd44c9055cc353a26875
1ef8cdbd3773bd82e5be25d4ba61e5e59371c6331726842107c0f1eb7d4d1f49
2d50b03a92445ba53ae147d0b97c494858c86a56fe037c44bc0edabb902420f7
31272235fcdce1d28542c0bc30c069cdb861ff34dd645fe5143ad911fdb1e8a9
```

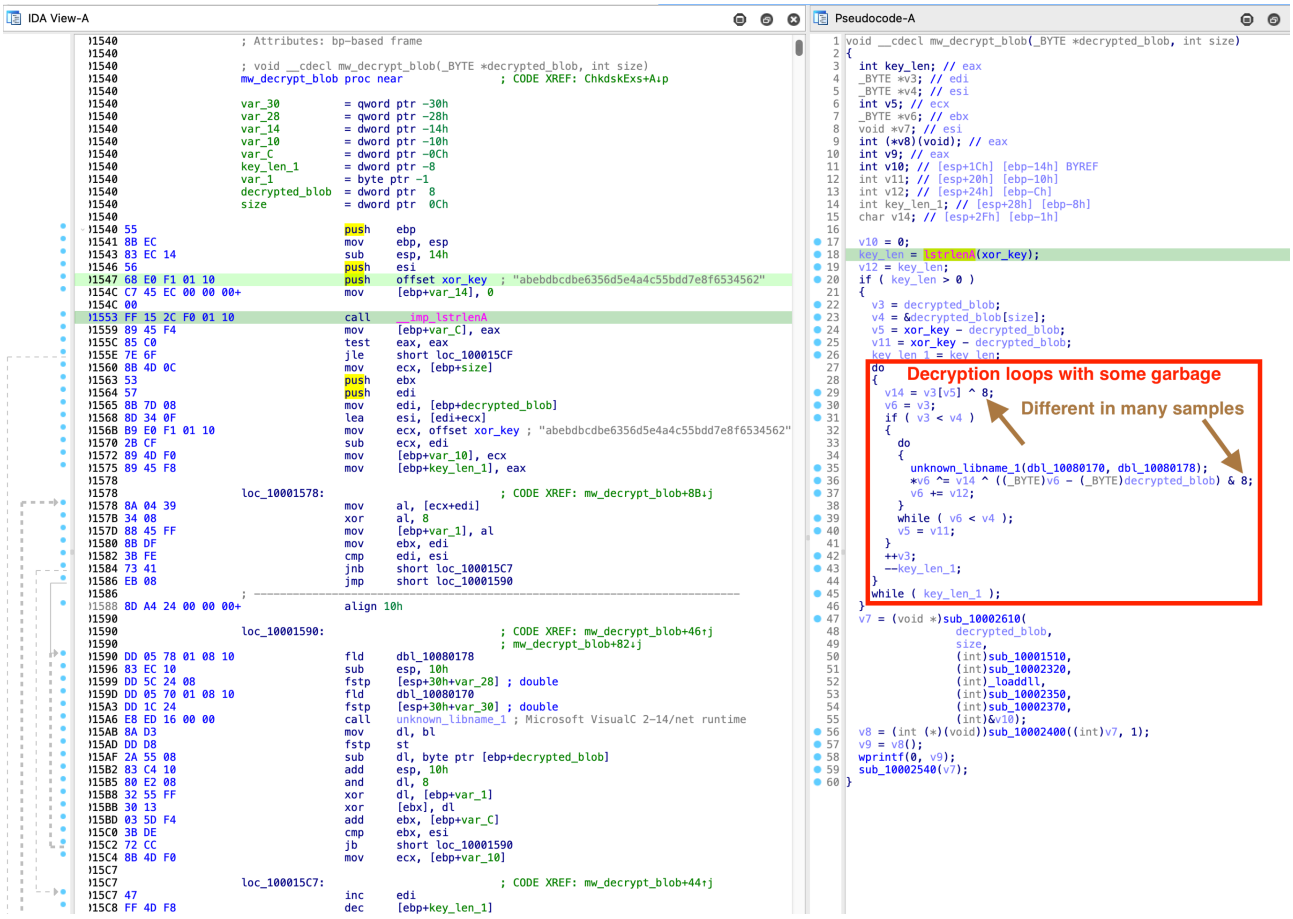
```
55d1480cd023b74f10692c689b56e7fd6cc8139fb6322762181daead55a62b9e
58b671915e239e9682d50a026e46db0d775624a61a56199f7fd576b0cef4564d
6210a9f5a5e1dc27e68ecd61c092d2667609e318a95b5dade3c28f5634a89727
68a86858b4638b43d63e8e2aaec15a9ebd8fc14d460dd74463db42e59c4c6f89
72813522a065e106ac10aa96e835c47aa9f34e981db20fa46a8f36c4543bb85d
7a64bc69b60e3cd3fd00d4424b411394465640f499e56563447fe70579ccdd00
7e39dcd15307e7de862b9b42bf556f2836bf7916faab0604a052c82c19e306ca
bf3c7f0ba324c96c9a9bff6cf21650a4b78edbc0076c68a9a125ebcba0e523c9
c3743a8c944f5c9b17528418bf49b153b978946838f56e5fca0a3f6914bee887
c3b3640ddf53b26f4ebd4eedf929540edb452c413ca54d0d21cc405c7263f490
c6c4f690f0d15b96034b4258bdfaf797432a3ec4f73fbc920384d27903143cb0
```

If you look at the binary, you will relatively quickly stumble upon a large binary blob that is referenced in only one function in the binary. The two loops in which the blob is referenced should give you a good indication that something might be decrypted here, see the screenshot below.

I have checked all available samples and the decryption algorithm is identical in each case, however, there are a few different variations, how the decryption function is called. In the most common variant there is an export, which calls a wrapper function, which in turn calls the decryption function. Sometimes there is only one wrapper function, sometimes several, and sometimes the decryption code is directly in the export of the DLL.







Regular executable where the call to decryption function is located in `WinMain`:

IDA View-A | Pseudocode-B | Pseudocode-A

```
1  int __stdcall WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int nShowCmd)
2  {
3    unsigned int v4; // eax
4    size_t v6; // [esp+8h] [ebp-1660h]
5    DWORD NumberOfBytesWritten; // [esp+Ch] [ebp-165Ch] BYREF
6    HANDLE hFindFile; // [esp+14h] [ebp-1654h]
7    HANDLE hFile; // [esp+18h] [ebp-1650h]
8    int i; // [esp+1Ch] [ebp-164Ch]
9    struct _WIN32_FIND_DATAW FindFileData; // [esp+20h] [ebp-1648h] BYREF
10   WCHAR FileName[1000]; // [esp+270h] [ebp-13F8h] BYREF
11   WCHAR Filename[514]; // [esp+A40h] [ebp-C28h] BYREF
12   WCHAR v15[260]; // [esp+E44h] [ebp-824h] BYREF
13   WCHAR pszPath[260]; // [esp+104Ch] [ebp-61Ch] BYREF
14   __int16 v17[260]; // [esp+1254h] [ebp-414h] BYREF
15   int v18[4]; // [esp+145Ch] [ebp-20Ch] BYREF
16   __int16 v19; // [esp+146Ch] [ebp-1FCh]
17   char v20[502]; // [esp+146Eh] [ebp-1FAh] BYREF
18
19   DialogBoxParamW(hInstance, (LPCWSTR)5, 0, DialogFunc, 0);
20   _InterlockedExchange(&dword_48A630, 1);
21   WaitForMultipleObjects(nCount, hObject, 1, 0xFFFFFFFF);
22   while ( nCount-- )
23     CloseHandle(hObject[nCount]);
24   memset(Filename, 0, 0x402u);
25   GetModuleFileNameW(hModule, Filename, 0x200u);
26   memset(v17, 0, sizeof(v17));
27   v4 = sub_4013E0(0);
28   srand(v4);
29   for ( i = 0; i < 6; ++i )
30     v17[i] = rand() % 21 + 97;
31   SHGetSpecialFolderPathW(0, pszPath, 35, 0);
32   wsprintfW(v15, L"%s\\%s.AF1TMP", pszPath, v17);
33   v18[0] = 3014698;
34   v18[1] = (int)&unk_460041;
35   v18[2] = 5505073;
36   v18[3] = 5242957;
37   v19 = 0;
38   memset(v20, 0, sizeof(v20));
39   memset(FileName, 0, sizeof(FileName));
40   wsprintfW(FileName, L"%s\\%s", pszPath, v18);
41   hFindFile = FindFirstFileW(FileName, &FindFileData);
42   if ( hFindFile == (HANDLE)-1 )
43   {
44     hFile = CreateFileW(v15, 0x40000000u, 2u, 0, 2u, 0x80u, 0);
45     FindClose((HANDLE)0xFFFFFFFF);
46     if ( hFile != (HANDLE)-1 )
47     {
48       NumberOfBytesWritten = 0;
49       v6 = wcslen(Filename);
50       WriteFile(hFile, Filename, 2 * v6, &NumberOfBytesWritten, 0);
51       CloseHandle(hFile);
52     }
53   }
54   return sub_401F10(&unk_424028, 417072);
55 }
```

Call to decryption function

Decryption code directly in an exported function:

```
24  int v21; // [esp+20h] [ebp-6Ch]
25  void *v22; // [esp+24h] [ebp-68h]
26  int v23; // [esp+28h] [ebp-64h]
27  int (__stdcall *v24)(int); // [esp+2Ch] [ebp-60h]
28  __int64 *v25; // [esp+30h] [ebp-5Ch]
29  int v26; // [esp+34h] [ebp-58h]
30  int v27; // [esp+38h] [ebp-54h]
31  __int128 *v28; // [esp+3Ch] [ebp-50h]
32  void *v29; // [esp+40h] [ebp-4Ch]
33  int JobObjectInformation[2]; // [esp+44h] [ebp-48h] BYREF
34  int v31; // [esp+4Ch] [ebp-40h] BYREF
35  __int128 v32[2]; // [esp+50h] [ebp-3Ch] BYREF
36  char Buffer[12]; // [esp+70h] [ebp-1Ch] BYREF
37  int v34; // [esp+88h] [ebp-4h]
38
39  v26 = 0;
40  v20 = 0i64;
41  v0 = (void *)unknown_libname_4(0x8000);
42  v1 = v0;
43  if ( v0 )
44    memset(v0, 0, 0x8000u);
45  else
46    v1 = 0;
47  v29 = v1;
48  v34 = 0;
49  v22 = v1;
50  v23 = 0x4000;
51  LODWORD(v20) = v20 & 0xFFFFFF50 | 0x2D;
52  v32[0] = xmmword_1001DE0C;
53  v25 = &v20;
54  v32[1] = xmmword_1001DE1C;
55  v24 = sub_10001660;
56  v28 = v32;
57  v21 = 0;
58  v27 = 300;
59  hObject = CreateJobObjectW(0, &Name);
60  v3 = unknown_libname_36(v2, (int)"time");
61  JobObjectInformation[0] = 5;
62  JobObjectInformation[1] = 100 * v3;
63  if ( !SetInformationJobObject(hObject, JobObjectCpuRateControlInformation, JobObjectInformation, 8u) )
64    printf_0_0("CPU limit");
65  v4 = 8201545;
66  do
67  {
68    gets_s(Buffer, 0xAu);
69    --v4;
70  }
71  while ( v4 );                         Decryption loop
72  v31 = 0;
73  v5 = lstrlenA(xor_key);
74  for ( i = 0; i < v5; ++i )
75  {
76    v7 = xor_key[i];
77    for ( j = (char *)&decrypted_blob + i; j < xor_key; j += v5 )
78      *j ^= v7 ^ ~((_BYTE)j - (unsigned __int8)&decrypted_blob) & 0xC;
79  }
80  lpMem = (LoadedModule *)sub_10002170(sub_10001260, sub_10002110, v15, v16, v17, &v31);
81  if ( *(_DWORD *)(*(_DWORD *)lpMem + 124)
82    && (v9 = (_DWORD *)(*(_DWORD *)lpMem + 1) + *(_DWORD *)(*(_DWORD *)lpMem + 120)), v9[6])
83    && (v10 = v9[5]) != 0
84    && (v11 = v9[4], v11 <= 1)
85    && (v12 = 1 - v11, 1 - v11 <= v10) )
86  {
87    v13 = lpMem;
88    v14 = ((int (__cdecl *)(int, int))(*((_DWORD *)lpMem + 1) + *(_DWORD *)(v9[7] + 4 * v12 + *((_DWORD *)lpMem + 1))))(
89            1,
90            2);
91  }
92  else
93  {
94    SetLastError(0x7Fu);
95    v13 = lpMem;
96    v14 = MEMORY[0](1, 2);
97  }
98  printf_0(0, v14);
99  LoadedModule::dtor_free(v13);
100 CloseHandle(hObject);
101 if ( v1 )
102   j_j__free(v1);
103 }
```

The decryption algorithm uses a hardcoded key and is XOR'ing through the entire binary blob, with incrementing the iterator by the length of the key. Additionally, another part of the decryption "formula" is a boolean `and` operation with a hardcoded value. By using a debugger, it's pretty easy to get to the unpacked code. However, since we want a have static unpacker, I reimplemented the function in Python.

```
def decrypt(data_blob, key, param):
    result = list(data_blob)
    i = 0
    while i < len(key):
        x = i
```

```
        key_xor = key[i] ^ param
        while x <= len(result) - 1:
            result[x] = result[x] ^ key_xor ^ ((x & 0xff) & param)
            x += len(key)
        i += 1

    return result
```

Now, all we need to decrypt is the binary blob, the decryption key and the parameter for the `and` operation. In my next blog post, I will describe how to get these values with help of Python and Malduck.
