

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

malware.love/malware_analysis/reverse_engineering/2023/02/12/analyzing-truebot-packer.html

February 12, 2023

12 Feb 2023 » [malware_analysis](#), [reverse_engineering](#)

In October 2022, [Microsoft published a blog post](#) about Raspberry Robin and its 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/SocGhosh. 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
1ef8cdbc3773bd82e5be25d4ba61e5e59371c6331726842107c0f1eb7d4d1f49
2d50b03a92445ba53ae147d0b97c494858c86a56fe037c44bc0edabb902420f7
31272235fcdce1d28542c0bc30c069cdb861ff34dd645fe5143ad911fdb1e8a9
55d1480cd023b74f10692c689b56e7fd6cc8139fb6322762181daead55a62b9e
58b671915e239e9682d50a026e46db0d775624a61a56199f7fd576b0cef4564d
6210a9f5a5e1dc27e68ecd61c092d2667609e318a95b5dade3c28f5634a89727
68a86858b4638b43d63e8e2aaec15a9ebd8fc14d460dd74463db42e59c4c6f89
72813522a065e106ac10aa96e835c47aa9f34e981db20fa46a8f36c4543bb85d
7a64bc69b60e3cd3fd00d4424b411394465640f499e56563447fe70579ccdd00
7e39dcd15307e7de862b9b42bf556f2836bf7916faab0604a052c82c19e306ca
bf3c7f0ba324c96c9a9bfff6cf21650a4b78edbc0076c68a9a125ebcba0e523c9
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.

Name	Address	Ordinal
ChkdskExs	10001620	1
ChkdskPosition	10001860	2
KbdLayerDescriptor	100016B0	3
DllEntryPoint	10004202	[main entry]

Exported function calling a decryption wrapper function

IDA View-A

```

10001620      ; Exported entry 1. ChkdskExs
10001620      .text:10001620
10001620      .text:10001620
10001620      .text:10001620
10001620      .text:10001620
10001620      .text:10001620      ; int ChkdskExs()
10001620      public ChkdskExs
10001620      ChkdskExs proc near
10001620      push    5FD30h      ; int
10001620      push    offset decrypted_blob ; Src
10001625      68 08 F2 01 10      call    mw_decrypt_blob
1000162A      E8 11 FF FF FF      add    esp, 8
10001632      C3                      retn
10001632      ChkdskExs endp
10001632
10001632

```

Pseudocode-A

```

1 int ChkdskExs()
2 {
3     return mw_decrypt_blob(decrypted_blob, 0x5FD30); // decrypted_blob, size
4 }

```

Decryption function Size of decrypted blob

Decrypted blob

IDA View-A

```

1540      ; Attributes: bp-based frame
1540      ; void __cdecl mw_decrypt_blob(_BYTE *decrypted_blob, int size)
1540      mw_decrypt_blob proc near
1540      var_30      = qword ptr -30h
1540      var_28      = qword ptr -28h
1540      var_14      = dword ptr -14h
1540      var_10      = dword ptr -10h
1540      var_C       = dword ptr -0Ch
1540      key_len_1   = byte ptr -1
1540      decrypted_blob = dword ptr 8
1540      size       = dword ptr 0Ch
1540
1540      55          push    ebp
1541      8B EC      mov    ebp, esp
1543      83 EC 14    sub    esp, 14h
1546      56          push    esi
1547      68 E0 F1 01 10      push    offset xor_key ; "abebdbcdbe6356d5e4a4c55bdd7e8f6534562"
154C      C7 45 EC 00 00 00+  mov    [ebp+var_14], 0
154C      00
1553      FF 15 2C F0 01 10      call    _imp_IsStrLenA
1559      89 45 F4      mov    [ebp+var_C], eax
155C      85 C0      test   eax, eax
155E      7E 6F      jle    short loc_100015CF
1560      8B 4D 0C      mov    ecx, [ebp+size]
1563      53          push    ebx
1564      57          push    edi
1565      8D 7D 08      mov    edi, [ebp+decrypted_blob]
1568      8D 34 0F      lea    esi, [edi+ecx]
156B      89 E0 F1 01 10      mov    ecx, offset xor_key ; "abebdbcdbe6356d5e4a4c55bdd7e8f6534562"
1570      2B CF      sub    ecx, edi
1572      89 4D F0      mov    [ebp+var_10], ecx
1575      89 45 F8      mov    [ebp+key_len_1], eax
1578
1578      loc_10001578:
1578      8A 04 39      mov    al, [ecx+edi] ; CODE XREF: mw_decrypt_blob+8B;j
157B      34 08      xor    al, 8
157D      8B 45 FF      mov    [ebp+var_1], al
1580      8B FF      mov    ebx, edi
1582      3B FE      cmp    edi, esi
1584      73 41      jnb    short loc_100015C7
1586      EB 08      jmp    short loc_10001590
1586
1588      8D A4 24 00 00 00+  align 10h
1590
1590      loc_10001590:
1590      ; CODE XREF: mw_decrypt_blob+46;j
1590      ; mw_decrypt_blob+82;j
1590      DD 05 78 01 08 10      fld    dbl_10000178
1596      83 EC 10      sub    esp, 10h
1599      DD 5C 24 08      fstp   [esp+30h+var_28]; double
159D      DD 05 70 01 08 10      fld    dbl_10000170
15A3      DD 1C 24      fstp   [esp+30h+var_30]; double
15A6      E8 ED 16 00 00      call   unknown_libname_1 ; Microsoft Visual C 2-14/net runtime
15AB      8A 03      mov    dl, bl
15AD      DD D8      fstp   st
15AF      2A 55 08      sub    dl, byte ptr [ebp+decrypted_blob]
15B2      83 C4 10      add    esp, 10h
15B5      80 E2 08      and    dl, 8
15B8      32 55 FF      xor    dl, [ebp+var_1]
15BB      30 13      xor    [ebx], dl
15BD      03 5D F4      add    ebx, [ebp+var_C]
15C0      3D DE      cmp    ebx, esi
15C2      72 CC      jb    short loc_10001590
15C4      8B 4D F0      mov    ecx, [ebp+var_10]
15C7
15C7      loc_100015C7:
15C7      47          inc    edi ; CODE XREF: mw_decrypt_blob+44;j
15C7      FF 4D F8      dec    [ebp+key_len_1]

```

Pseudocode-A

```

1 void __cdecl mw_decrypt_blob(_BYTE *decrypted_blob, int size)
2 {
3     int key_len; // eax
4     _BYTE *v3; // edi
5     _BYTE *v4; // esi
6     int v5; // ecx
7     _BYTE *v6; // ebx
8     void *v7; // esi
9     int (*v8)(void); // eax
10    int v9; // eax
11    int v10; // [esp+1Ch] [ebp-14h] BYREF
12    int v11; // [esp+20h] [ebp-10h]
13    int v12; // [esp+24h] [ebp-Ch]
14    int key_len_1; // [esp+28h] [ebp-8h]
15    char v14; // [esp+2Fh] [ebp-1h]
16
17    v10 = 0;
18    key_len = IsStrLen(xor_key);
19    v12 = key_len;
20    if ( key_len > 0 )
21    {
22        v3 = decrypted_blob;
23        v4 = &decrypted_blob[size];
24        v5 = xor_key - decrypted_blob;
25        v11 = xor_key - decrypted_blob;
26        key_len_1 = key_len;
27
28        do
29        {
30            v14 = v3[v5] ^ 8;
31            v6 = v3;
32            if ( v3 < v4 )
33            {
34                do
35                {
36                    unknown_libname_1(dbl_10000170, dbl_10000170);
37                    *v6 ^= v14 ^ ((_BYTE)v6 - (_BYTE)decrypted_blob) & 8;
38                    v6 += v12;
39                } while ( v6 < v4 );
40                v5 = v11;
41                ++v3;
42                --key_len_1;
43            } while ( key_len_1 );
44        }
45        v7 = (void *)sub_10002610(
46            decrypted_blob,
47            size,
48            (int)sub_10001510,
49            (int)sub_10002320,
50            (int)_loaddl,
51            (int)_loaddl,
52            (int)sub_10002350,
53            (int)sub_10002370,
54            (int)sub_10002370,
55            (int)&v10);
56        v8 = (int (*)(void))sub_10002400((int)v7, 1);
57        v9 = v8();
58        wprintf(0, v9);
59        sub_10002540(v7);
60    }


```

Decryption loops with some garbage

Different in many samples

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(&word_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    SHGetSpecialFolderPath(0, pszPath, 35, 0);
32    wprintfW(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    wprintfW(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 & 0xFFFFFFFF50 | 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 );
72 v31 = 0;
73 v5 = strlenA(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_IO(1, 2);
97 }
98 printf_0(0, v14);
99 LoadedModule::dtor_free(v13);
100 CloseHandle(hObject);
101 if ( v1 )
102     j__free(v1);
103 }

```

Decryption loop

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.

Related Posts

- [TrueBot Analysis Part II - Static unpacker](#) (Categories: [malware_analysis](#), [reverse_engineering](#))
- [Python stealer distribution via excel maldoc](#) (Categories: [malware_analysis](#), [reverse_engineering](#))
- [Having fun with an Ursnif VBS dropper](#) (Categories: [malware_analysis](#), [reverse_engineering](#))
- [Trickbot tricks again \[UPDATE\]](#) (Categories: [malware_analysis](#), [reverse_engineering](#))
- [Trickbot tricks again](#) (Categories: [malware_analysis](#), [reverse_engineering](#))