Technical Analysis of DanaBot Obfuscation Techniques

zscaler.com/blogs/security-research/technical-analysis-danabot-obfuscation-techniques

Key Points

- DanaBot is a malware-as-a-service platform discovered in 2018 that is designed to steal sensitive information that may be used for wire fraud, conduct cryptocurrency theft, or perform espionage related activities
- The malware is heavily obfuscated which makes it very difficult and time consuming to reverse engineer and analyze
- Zscaler ThreatLabz has reverse engineered the various obfuscation techniques used by DanaBot and developed a set of tools using IDA Python scripts to assist with binary analysis

DanaBot, first discovered in 2018, is a malware-as-a-service platform that threat actors use to steal usernames, passwords, session cookies, account numbers, and other personally identifiable information (PII). The threat actors may use this stolen information to commit banking fraud, steal cryptocurrency, or sell access to other threat actors.

While DanaBot isn't as prominent as it once was, the malware is still a formidable and active threat. Recently, version 2646 of the malware was spotted in the wild and also a researcher tweeted screenshots of Danabot's advertisement website shown in Figure 1.

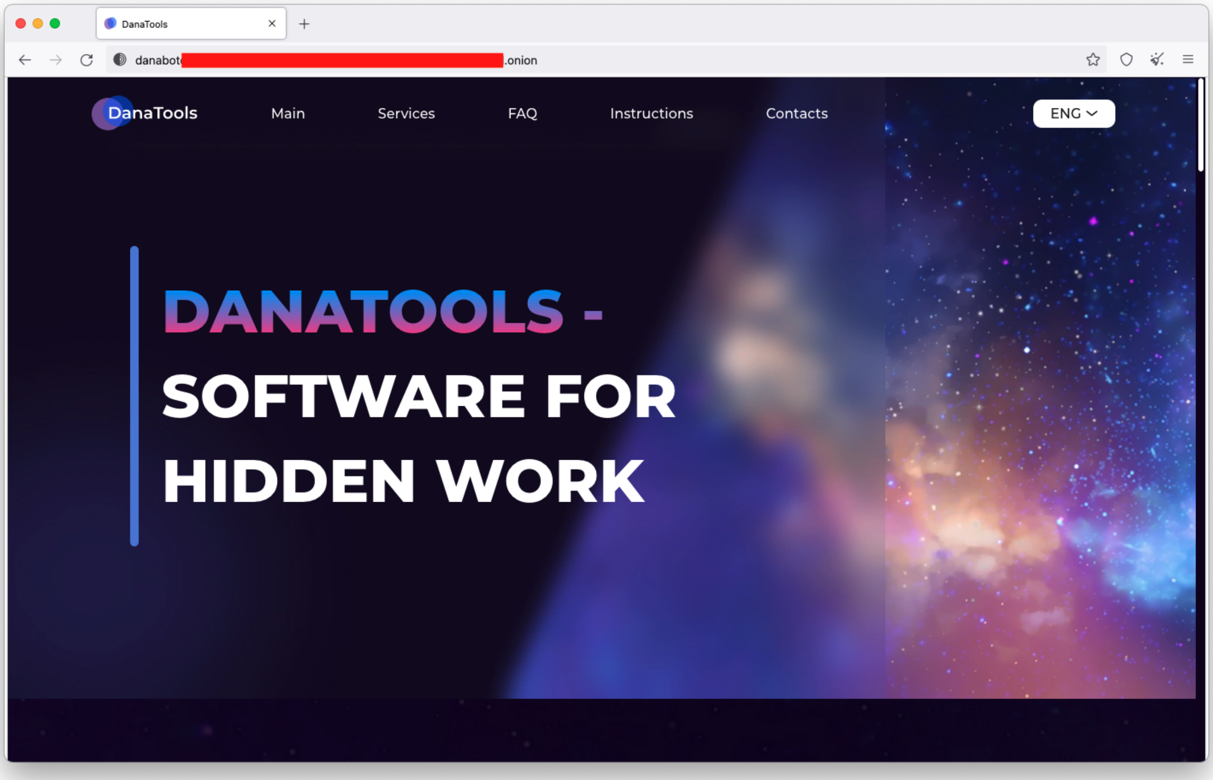


Figure 1: DanaBot's advertisement website

Unfortunately, the DanBot developers have done a very good job of obfuscating the malware code. Therefore, it is very difficult and time consuming process to to reverse engineer and analyze. This is a companion blog post to a set of IDA Python scripts that Zscaler ThreatLabz is releasing on our Github page. The goal of the scripts is to help peel away some of the layers of DanaBot's obfuscations and inspire additional research into not only the obfuscation techniques, but the malware itself.

Technical Analysis

The following sections summarize the numerous techniques that the DanaBot developers have implemented to obfuscate the malware binary code.

Junk Byte Jumps

One of the first anti-analysis techniques that DanaBot employs is a "junk byte jump" instruction. This is an anti-disassembly technique where a jump instruction will always jump over a junk byte. The junk byte is skipped during normal program execution, but causes IDA Pro to display an incorrect disassembly. An example of this technique is shown in Figure 2:

```
.text:00C73EA4 B8 00 00 00 00                 mov     eax, 0          ; jumptable 00C73E5D case 0
.text:00C73EA9 83 F8 01                       cmp     eax, 1
.text:00C73EAC 72 01                          jb      short near ptr loc_C73EAE+1
.text:00C73EAE
.text:00C73EAE                 loc_C73EAE:                            ; CODE XREF: sub_C6A01C+9E90↑j
.text:00C73EAE 0D 8B 45 E0 8B                 or      eax, 8BE0458Bh
.text:00C73EB3 10 FF                          adc     bh, bh
.text:00C73EB5 52                             push    edx
.text:00C73EB6 14 85                          adc     al, 85h
.text:00C73EB8 C0 0F 8E                       ror     byte ptr [edi], 8Eh
.text:00C73EBB DF 19                          fistp   word ptr [ecx]
```

Figure 2: An example of a junk byte jump

The *01_junk_byte_jump.py* IDA Python script searches for junk byte jump patterns and patches them with NOP instructions. This operation fixes IDA Pro's disassembly as shown in Figure 3.

```
.text:00C73EA4 90                nop                    ; jumptable 00C73E5D case 0
.text:00C73EA5 90                nop
.text:00C73EA6 90                nop
.text:00C73EA7 90                nop
.text:00C73EA8 90                nop
.text:00C73EA9 90                nop
.text:00C73EAA 90                nop
.text:00C73EAB 90                nop
.text:00C73EAC 90                nop
.text:00C73EAD 90                nop
.text:00C73EAE 90                nop
.text:00C73EAF 8B 45 E0          mov     eax, [ebp+var_20]
.text:00C73EB2 8B 10             mov     edx, [eax]
.text:00C73EB4 FF 52 14          call    dword ptr [edx+14h]
.text:00C73EB7 85 C0             test    eax, eax
.text:00C73EB9 0F 8E DF 19 00 00 jle     def_C73E5D     ; jumptable 00C73E5D default case
```

Figure 3: An example of a patched junk byte jump

Dynamic Returns

The next anti-analysis technique is a "dynamic return" operation. This technique calculates a new return address at the end of a function, causing a change in the program's control flow. In DanaBot's implementation, they are used to "extend" a function–exposing additional hidden code. An example of a dynamic return is shown in Figure 4.

```
.text:00D14456 E8 00 00 00 00    call    $+5        ; push next address (0x00D1445B) onto the stack
.text:00D1445B 58                pop     eax        ; pop that pushed address (0x00D1445B) into eax
.text:00D1445C 83 C0 09          add     eax, 9     ; add 9 to the address: 0x00D1445B + 9 = 0x00D14464
.text:00D1445F 50                push    eax        ; push this calculated address (0x00D14464) onto the stack as the new return address
.text:00D14460 C2 00 00          retn    0          ; return to the calculated address (0x00D14464)
.text:00D14460               ; ---------------------------------------------------------------------------
.text:00D14463 E6                db 0E6h            ; junk byte
.text:00D14464 8D                db  8Dh            ; calculated address
.text:00D14464                                      ;
.text:00D14464                                      ; this data will be converted into code and extend the original function
.text:00D14465 45                db  45h ; E
.text:00D14466 DC                db 0DCh
.text:00D14467 8B                db  8Bh
.text:00D14468 15                db  15h
.text:00D14469 2C                db  2Ch ; , OFF32 SEGDEF [_data,DB8E2C]
.text:00D1446A 8E                db  8Eh
.text:00D1446B DB                db 0DBh
.text:00D1446C 00                db    0
.text:00D1446D 8A                db  8Ah
.text:00D1446E 12                db  12h
.text:00D1446F 88                db  88h
.text:00D14470 50                db  50h ; P
.text:00D14471 01                db    1
.text:00D14472 C6                db 0C6h
.text:00D14473 00                db    0
.text:00D14474 01                db    1
.text:00D14475 8D                db  8Dh
```

Figure 4: An example of a dynamic return

Using the *02_dynamic_return.py* IDA Python script, these dynamic returns can be patched, the functions extended, and the hidden code exposed. An example of this is shown in Figure 5.

```
.text:00D14456 90                   nop                        ; call    $+5
.text:00D14457 90                   nop
.text:00D14458 90                   nop
.text:00D14459 90                   nop
.text:00D1445A 90                   nop
.text:00D1445B 90                   nop                        ; pop     eax
.text:00D1445C 90                   nop                        ; add     eax, 9
.text:00D1445D 90                   nop
.text:00D1445E 90                   nop
.text:00D1445F 90                   nop                        ; push    eax
.text:00D14460 90                   nop                        ; retn    0
.text:00D14461 90                   nop
.text:00D14462 90                   nop
.text:00D14463 90                   nop                        ; junk byte
.text:00D14464 8D 45 DC             lea     eax, [ebp-24h]     ; calculated address
.text:00D14464                                                 ;
.text:00D14464                                                 ; this data will be converted into code and extend the original function
.text:00D14467 8B 15 2C 8E DB 00    mov     edx, off_DB8E2C
.text:00D1446D 8A 12                mov     dl, [edx]
.text:00D1446F 88 50 01             mov     [eax+1], dl
.text:00D14472 C6 00 01             mov     byte ptr [eax], 1
.text:00D14475 8D 55 DC             lea     edx, [ebp-24h]
.text:00D14478 8D 45 D8             lea     eax, [ebp-28h]
.text:00D1447B E8 94 61 6F FF       call    sub_40A614
.text:00D14480 8D 45 D4             lea     eax, [ebp-2Ch]
.text:00D14483 8B 15 98 8C DB 00    mov     edx, off_DB8C98
.text:00D14489 8A 12                mov     dl, [edx]
.text:00D1448B 88 50 01             mov     [eax+1], dl
.text:00D1448E C6 00 01             mov     byte ptr [eax], 1
.text:00D14491 8D 55 D4             lea     edx, [ebp-2Ch]
.text:00D14494 8D 45 D8             lea     eax, [ebp-28h]
```

Figure 5: An example of a patched dynamic return

Stack String Deobfuscation Preparation and Code Re-analysis

Before moving on to additional DanaBot anti-analysis techniques, we've included three IDA Python scripts:

- *03_uppercase_jumps.py*
- *04_letter_mapping.py*
- *05_reset_code.py*

The first two scripts are preparation steps to help with stack string deobfuscation described in a later section. The first script patches out a code pattern that is used to uppercase letters (this removes a small function basic block that interferes with stack string reconstruction) and the second script renames variables that store the letters used in stack strings.

Before running the third script, check that IDA Pro's "Options->Compiler…" is set to "Delphi" (see Figure 6.)

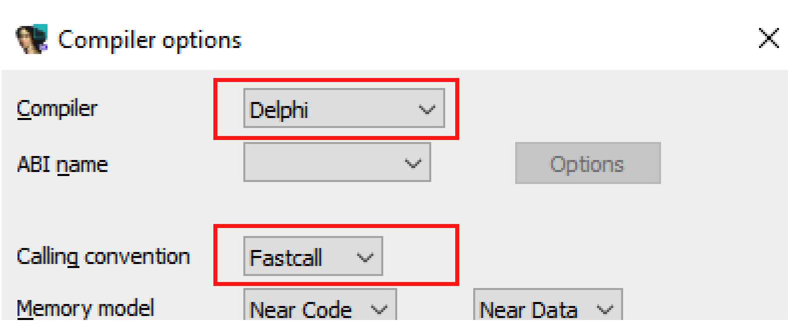


Figure 6: IDA Pro's Compiler options

Since the previous scripts patched a lot of existing code and exposed a bunch of new code, the *05_reset_code.py* script helps reset and re-analyze the modified code in IDA Pro to get a cleaner IDB database. Once the script and analysis completes, some manual clean up may be required. Our general method is:

- Search → Sequence of bytes…
- Search for the standard function prolog: 55 8B EC
- Sort by Function
- For each result without a defined function:
  - Right click → Create function…
  - Look for any addresses that are causing issues in the Output window
  - Right click → Undefine
  - Right click → Code

Junk StrAsg and StrCopy Function Calls

DanaBot adds a lot of junk code to slow down and complicate reverse engineering. One of the junk code patterns is adding extraneous *StrAsg* and *StrCopy* function calls. These functions are part of the standard Delphi library and are used to assign or copy data between variables. Figure 7 shows an example snippet of code with a number of these calls. If we trace the variable arguments we can see that they are usually assigned to themselves or a small set of other variables that aren't used in actual malware code.

```
char sub_4DAA24()
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]

  v124 = &savedregs;
  v123[1] = &unk_4DC551;
  v123[0] = NtCurrentTeb()->NtTib.ExceptionList;
  __writefsdword(0, v123);
  v231 = 0;
  v122[2] = &savedregs;
  v122[1] = &unk_4DC4B9;
  v122[0] = NtCurrentTeb()->NtTib.ExceptionList;
  __writefsdword(0, v122);
  System::__linkproc__ UStrCopy(junk1, 1, 1, &junk2);
  v221 = 158 - v222;
  v219 = 0;
  do
  {
    ++v219;
    System::__linkproc__ UStrCopy(junk2, 1, 0, &junk2);
  }
  while ( v219 < 1 );
  System::__linkproc__ UStrLAsg(&junk1, junk1);
  System::__linkproc__ UStrLAsg(&junk1, junk1);
  System::__linkproc__ UStrLAsg(&junk2, junk1);
  if ( v220 > i )
  {
    for ( i = 0; i != 11; ++i )
    {
      v217 = sub_407818(v226);
      v227 = 181 * v222;
      v225 = v223 + 141;
      System::__linkproc__ UStrCopy(junk2, 1, 0, &junk2);
      System::__linkproc__ UStrCopy(junk2, 1, 0, &junk1);
      System::__linkproc__ UStrCopy(junk1, 1, 1, &junk1);
```

Figure 7: Example of junk StrAsg and StrCopy function calls

The IDA Python script *06_fake_UStrLAsg_and_UStrCopy.py* tries to find and patch these junk calls. Figure 8 shows the result in the example from Figure 7.

```
char sub_4DAA24()
{
  // [COLLA   RET 0001 al          char;   SS KEYPAD CTRL-"+" TO EXPAND]
              TOTAL STKARGS SIZE: 0
  v127 = &s
  v126 = &unk_4DC551;
  ExceptionList = NtCurrentTeb()->NtTib.ExceptionList;
  __writefsdword(0, &ExceptionList);
  v234 = 0;
  v124 = &savedregs;
  v123 = &unk_4DC4B9;
  v122 = NtCurrentTeb()->NtTib.ExceptionList;
  __writefsdword(0, &v122);
  v224 = 158 - v225;
  for ( i = 0; i < 1; ++i )
    ;
  if ( v223 > j )
  {
    for ( j = 0; j != 11; ++j )
    {
      v220 = sub_407818(v229);
      v230 = 181 * v225;
      v228 = v226 + 141;
    }
  }
```

Figure 8: Example of patched junk StrAsg and StrCopy function calls

Stack Strings

The next obfuscation method is DanaBot’s version of creating “stack strings”. The malware assigns letters of the alphabet to individual variables and then uses those variables, pointers to those variables, and various Delphi character/string handling functions to construct strings one character at a time. Figure 9 is an example construction of the string “wow64.dll”.

```
BYTE5(v27) = *gp_w[0];                          // w
BYTE4(v27) = 1;
w_System::Move(&v27, (&v27 + 4));
v26 = *gp_o;                                    // o
v25 = 1;
LOBYTE(v3) = 2;
System::__linkproc__ PStrNCat(v3, &v25, ExceptionList);
w_System::Move(v24, &v27);
v26 = *gp_w[0];                                 // w
v25 = 1;
LOBYTE(v4) = 3;
System::__linkproc__ PStrNCat(v4, &v25, ExceptionList);
w_LStrFromPCharLen_0(v24, ExceptionList, v17);
ExceptionList = v28;
w_IntToStr(6, &v23);                            // 6
v15 = v23;
w_IntToStr(4, &v22);                            // 4
v14 = v22;
HIBYTE(v5) = BYTE1(gp_d);                       // d
LOBYTE(v5) = *gp_d;
w_LStrFromPCharLen(0, v5);
v13 = v21;
HIBYTE(v6) = BYTE1(gp_l);                       // l
LOBYTE(v6) = *gp_l;
w_LStrFromPCharLen(0, v6);
v12 = v20;
HIBYTE(v7) = BYTE1(gp_l);                       // l
LOBYTE(v7) = *gp_l;
w_LStrFromPCharLen(0, v7);
System::__linkproc__ LStrCatN(v19, v12, v13, &g_dot, v14, v15, ExceptionList);
```

Figure 9: Example stack string of "wow64.dll"

These stack strings litter most of the malicious functions in DanaBot and very easily lead to reverse engineering fatigue. On top of that, while some of the constructed strings are used for malware purposes, most of them turn out to be junk strings. Figure 10 is a snippet of output from a script that will be introduced below. As can be seen in the figure, most of the strings are random DLL, executable, and Windows API names.

```
setting comment zqoskwuerypenubeys at 0xce37f4
setting comment swmystemebobiledll at 0xce3ac9
setting comment wmidll at 0xce3d5b
setting comment sppcomapidll at 0xce3f06
setting comment mpcisbdaonstdll at 0xce4220
setting comment aaclientdll at 0xce45ba
setting comment abtrokerexe at 0xce486b
setting comment gpapidll at 0xce4aa7
setting comment kbdltdll at 0xce4cd9
setting comment unimdmatdll at 0xce4f38
setting comment cnbdll at 0xce528e
setting comment cnbpdll at 0xce556e
setting comment netutilsdll at 0xce59f8
setting comment mwsmmicrosoftananagementresourcesdll at 0xce77a2
setting comment sbsmscordbidll at 0xce817c
setting comment appidpolicyconverterexe at 0xce8877
setting comment mfpsdll at 0xce907b
setting comment apimswincorestringldll at 0xce921f
setting comment eehixtensdll at 0xce9a34
setting comment epnreabdll at 0xcea1a5
setting comment piulcverfncrementongounteralue at 0xcead16
setting comment imjppdmgexe at 0xceb414
setting comment cmscigdll at 0xceb696
setting comment adsmsextdll at 0xceb94c
setting comment mvstoccicrosoftisualtudioools fficeontainerontrolnidll at 0xcebc20
setting comment tvdiatadll at 0xcec529
```

Figure 10: Example script output showing junk strings

The best way to extract these stack strings is by emulating the construction code, but due to the following reasons we experimented with another deobfuscation technique:

- There are thousands of these strings
- There are not clear start/end patterns to automatically extract the construction code
- They rely on standard Delphi functions which aren't particularly easy to emulate
- Most of them are junk strings
- The sheer amount of construction code hinders malware analysis the most

The goal of the IDA Python scripts *07_stack_string_letters_to_last_StrCatN_call.py* and *08_set_stack_string_letters_comments.py* is not to extract a wholly accurate stack string, but enough of the string to determine whether the string is junk or not. After some trial and error experimentation, the scripts also try their best to remove the stack string construction code to allow for much easier analysis. If the string turns out to be legitimate, the original construction code is saved as comments so a proper extraction of the string can be done if/when needed.

Empty Loops and Junk Math Loops

After removing the junk *StrAsg* and *StrCopy* function calls and the stack strings, there will be a bunch of empty loops. The IDA Python script *09_empty_loops.py* can be used to remove these loops. There will also be loops left that just contain junk math code (see

Figure 11.) The IDA Python script *10_math_loops.py* will remove these junk code math loops.

```
do
{
  ++j;
  m = 7 * i;
}
while ( j < 14 );
```

Figure 11: Example junk math loops

Junk Strings and Junk Global Variables

As we saw in the stack strings section above, there were a lot of DLL, executable, and Windows API name based junk strings. These junk strings exist as normal strings as well, see Figure 12 as an example.

Strings

Address	Length	Type	String
.text:00CC9...	0000000E	C (1...	ec.dll
.text:00CC9...	00000018	C (1...	IMJPAPI.DLL
.text:00CC9...	0000000C	C (1...	R.DLL
.text:00CC9...	00000070	C (1...	osoft.VisualStudio.Tools.Applications.Contract.v9.0.dll
.text:00CC9...	00000042	C (1...	stem.ServiceModel.WasHosting.dll
.text:00CC9...	00000016	C (1...	acerpt.exe
.text:00CC9...	00000018	C (1...	oledb32.dll
.text:00CC9...	00000014	C (1...	srwmi.dll
.text:00CC9...	00000012	C (1...	OKSE.dll
.text:00CCA...	00000020	C (1...	exicons0009.dll
.text:00CCA...	0000001C	C (1...	msidcrl30.dll
.text:00CCA...	00000010	C (1...	r20.dll
.text:00CCA...	00000010	C (1...	Cui.exe
.text:00CCA...	00000040	C (1...	i-ms-win-core-handle-l1-1-0.dll
.text:00CCA...	00000018	C (1...	pidgenx.dll
.text:00CCA...	00000018	C (1...	mshwgst.dll
.text:00CCA...	00000016	C (1...	InkDiv.dll
.text:00CCA...	00000014	C (1...	krskf.dll
.text:00CCA...	00000016	C (1...	dmintf.dll
.text:00CCA...	00000040	C (1...	stem.Diagnostics.StackTrace.dll
.text:00CCA...	00000016	C (1...	PMONTR.DLL
.text:00CCA...	00000030	C (1...	cationNotifications.exe
.text:00CCA...	00000014	C (1...	c100u.dll
.text:00CCA...	0000003E	C (1...	i-ms-win-core-synch-l1-1-0.dll
.text:00CCA...	00000014	C (1...	csnap.dll
.text:00CCA...	00000024	C (1...	ostNavigators.dll
.text:00CCA...	00000016	C (1...	msv1_0.dll
.text:00CCA...	0000002A	C (1...	msvcr110_clr0400.dll
.text:00CCA...	0000000E	C (1...	ex.dll

Line 5379 of 5913

Figure 12: Example junk strings

While we haven’t found good patterns to automatically remove references to these junk strings, the IDA Python script *11_rename_junk_variables.py* renames them as “junk” to ease manual analysis.

DanaBot also adds a lot of junk code involving global variables and various math operations, see Figure 13 for an example.

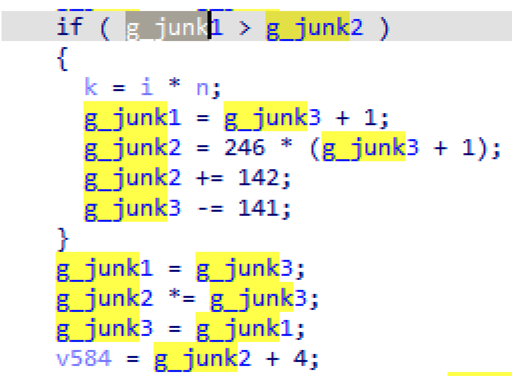


Figure 13: Example junk global variable math

The IDA Python script *12_rename_junk_random_variables.py* attempts to locate and rename these variables as “junk” to help with analysis.

Miscellaneous Tips and Tricks

Based on our experience reverse engineering DanaBot over the years, we have found the following miscellaneous tricks and tips to be helpful. The first is using the Interactive Delphi Reconstructor (IDR) program to export standard Delphi library function and variable names. We use Tools → MAP Generator and Tools → IDC Generator to export MAP and IDC files. While IDR creates an IDA IDC script, we don’t use it directly as it degrades the quality of the IDA Pro disassembly/decompilation. Instead, we use the scripts *idr_idc_to_idapy.py* and *idr_map_to_idapy.py* to extract the information from the generated IDC and MAP files and use the output scripts to import the naming information.

DanaBot resolves some of its Windows API functions by hash, so we use OALabs’ HashDB IDA Plugin (which recently added support for DanaBot’s hashing algorithm) to resolve the names by hash.

Finally, we make liberal use of IDA Pro’s right click → Collapse item feature to hide the remaining junk code, especially the renamed junk strings and global variables.

Before and After Example

As an overall example, Figure 14 is a screenshot for a section of DanaBot code before the deobfuscation scripts have been applied. The details of the code don’t particularly matter for this discussion, but the snippet shows DanaBot’s initialization of its 455-byte binary structure used in its initial “system information” command and control beacon.

```
    dword_DB8AC0 *= 197;
    dword_DB8AC4 = sub_407360(v696, v728, v775, v783, v790, v796, v802, v809, v
    dword_DB8AC0 = dword_DB8ABC;
    if ( 2 * dword_DB8ABC > dword_DB8ABC )
    {
      v1016 = dword_DB8AC0 + 4;
      dword_DB8AC4 = System::__linkproc__ TRUNC((dword_DB8AC0 + 4));
      dword_DB8ABC = 252 - dword_DB8AC4;
      dword_DB8AC0 = dword_DB8AC4 - 249;
      dword_DB8AC4 = sub_407818(dword_DB8AC4);
    }
    sub_407918(&s455, 455, 0);
    for ( k = 0; k != 8; ++k )
    {
      System::__linkproc__ UStrLAsg();
      for ( m = 0; m != 6; ++m )
      {
        dword_DB8ABC -= 69;
        dword_DB8ABC -= dword_DB8AC0;
        v1016 = dword_DB8ABC + 14;
        sub_407818((dword_DB8ABC + 14));
        dword_DB8AC0 = dword_DB8ABC * dword_DB8ABC;
        dword_DB8AC4 = dword_DB8ABC * dword_DB8ABC + 111;
      }
      dword_DB8AC0 = dword_DB8AC4 + dword_DB8ABC;
      dword_DB8ABC += dword_DB8AC4;
      if ( dword_DB8ABC > dword_DB8AC4 )
      {
        dword_DB8ABC -= 149;
        dword_DB8AC0 = 147 * dword_DB8ABC;
        dword_DB8AC0 = System::__linkproc__ TRUNC(dword_DB8ABC);
        dword_DB8AC4 = dword_DB8AC0;
        v1016 = dword_DB8AC0 + 4;
        dword_DB8ABC = System::__linkproc__ TRUNC((dword_DB8AC0 + 4));
```

Figure 14: Example of code before deobfuscations

Figure 15 is an example of the same code snippet after applying the deobfuscation scripts.

```
junk_random_3 *= 197;
junk_random_4 = idr609611_Random(v34, v35, v36);
junk_random_3 = junk_random_5;
if...
idr609737__FillChar(&s455, 455, 0);
for...
junk_random_3 = 0;
junk_random_5 = 0;
junk_random_3 = idr609696__ROUND(0);
*&s455.command = 1024;
idr610929__PStrNCpy(&s455.arg, &gp_s744->botid, 32u);
j = m + k;
idr611389__UStrEqual(L"msobjs.dll", junk_string_259);
if...
idr611330__UStrFromString(v34, v35, v36); // sqloledbdll
idr611318__UStrToString(255, *gp_build_hash, v55);
idr610929__PStrNCpy(&s455.arg2, v55, 0x20u);
v58 = k + 134;
j = idr609696__ROUND((k + 134));
junk_random_4 = junk_random_3;
if...
junk_random_5 += 108;
junk_random_3 = 0;
v67 = junk_string_257;
if...
if...
generate_random_data(v53);
idr610998__LStrFromWStr(v34);
get_md5_hex_digest(v53[1], &v54);
encode_len_crc32_data(v54, &s455.rand2, 0x108);
i = 230 - j;
k = 0;
do...
v58 = junk_random_5 + 4;
junk_random_4 = idr609701__TRUNC((junk_random_5 + 4));
```

Figure 15: Example of code after deobfuscations

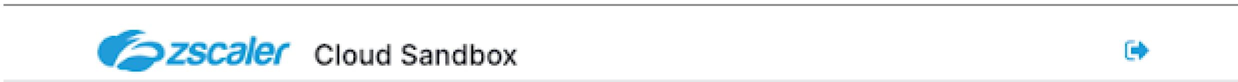
Conclusion

While there is still room for improvement, the DanaBot malware code is much easier to analyze and reason about. Expanding the scope to the entire binary, the deobfuscation techniques significantly reduce the complexity and time spent while reverse engineering the malware. We look forward to making further improvements/additions and welcome other researchers' contributions to the existing scripts to peel away more layers of DanaBot's obfuscation.

Zscaler Detection Status

W32/Danabot

Cloud Sandbox Detection



SANDBOX DETAIL REPORT

• High Risk • Moderate Risk • Low Risk

Report ID (MD5): 014751C83DB75E84BFF37F67036... Analysis Performed: 11/17/2022 9:07:47 AM File Type: dll

CLASSIFICATION

Class Type
Malicious

Category
Malware & Botnet

Threat Score
88

MITRE ATT&CK

This report contains 10 ATT&CK techniques mapped to 6 tactics

VIRUS AND MALWARE

No known Malware found

SECURITY BYPASS

- AV Process Strings Found
- May Try To Detect The Virtual Machine To Hinder Analysis

NETWORKING

- May Use The Tor Software To Hide Its Network Traffic
- URLs Found In Memory Or Binary Data

STEALTH

- Tries To Detect Virtualization Through RDTSC Time Measurements
- Disables Application Error Messages

SPREADING

No suspicious activity detected

INFORMATION LEAKAGE

- Enumerates The File System

EXPLOITING

- Known MD5
- May Try To Detect The Windows Explorer Process

PERSISTENCE

- Creates Temporary Files
- May Use Bcdedit To Modify The Windows Boot Settings
- PE File Contains Sections With Non-Standard Names

SYSTEM SUMMARY

- Dynamic Yara Hits
- One Or More Processes Crash
- Queries Information About The Installed CPU
- Binary Contains Paths To Debug Symbols
- Classification Label

DOWNLOAD SUMMARY

Original file	10 MB
Dropped files	2 MB
Packet capture	No network traffic

ORIGIN

Origin information not identified

FILE PROPERTIES

File Type dll

Digital Certificate Vendor File is not digitally signed

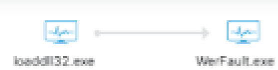
File Size
10,740,736 bytes

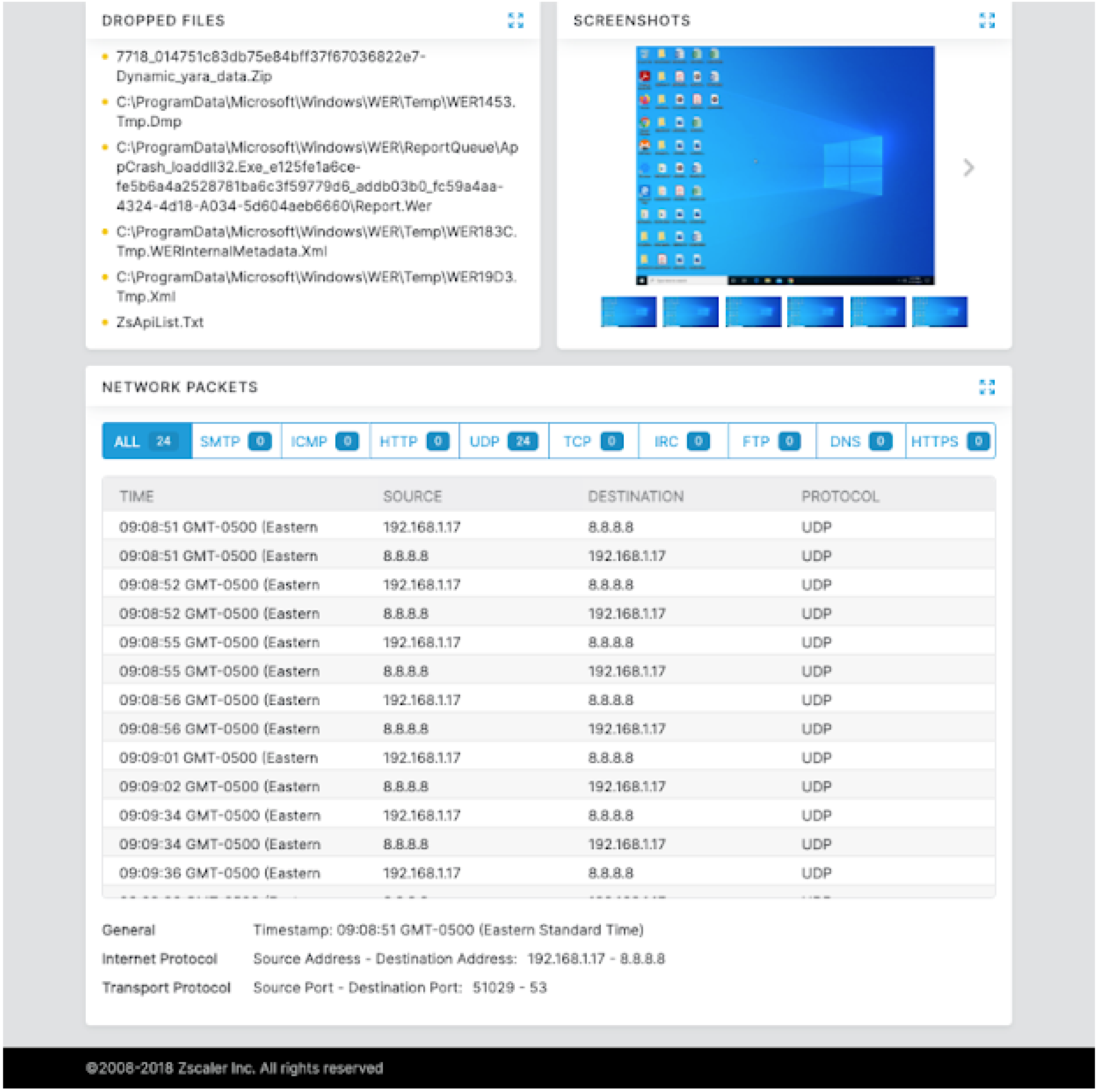
MD5
014751c83db75e84bff37f67036822e7

SHA1
868a60a08d6e3b94ff4b3fe3696b5d5f-f2733e12

SSDEEP
196608:IMnUzPCDNtqRHWJhmhf/alqJ81Lq-YVhqXg8UINc94ArNGmT/qZA:IMniCDNtq1a8g5Lqfg8UeUmDk

PROCESS SUMMARY





Indicators of Compromise

IOC	Notes
8c6224d9622b929e992500cb0a75025332c9cf901b3a25f48de6c87ad7b67114	SHA256 hash of DanaBot version 2646 main component