# Gootkit Banking Trojan | Deep Dive into Anti-Analysis Features

**labs.sentinelone.com**/gootkit-banking-trojan-deep-dive-anti-analysis-features/

Daniel Bunce

*In this post, Daniel discusses the Gootkit malware banking trojan and its use of Anti Analysis techniques.*

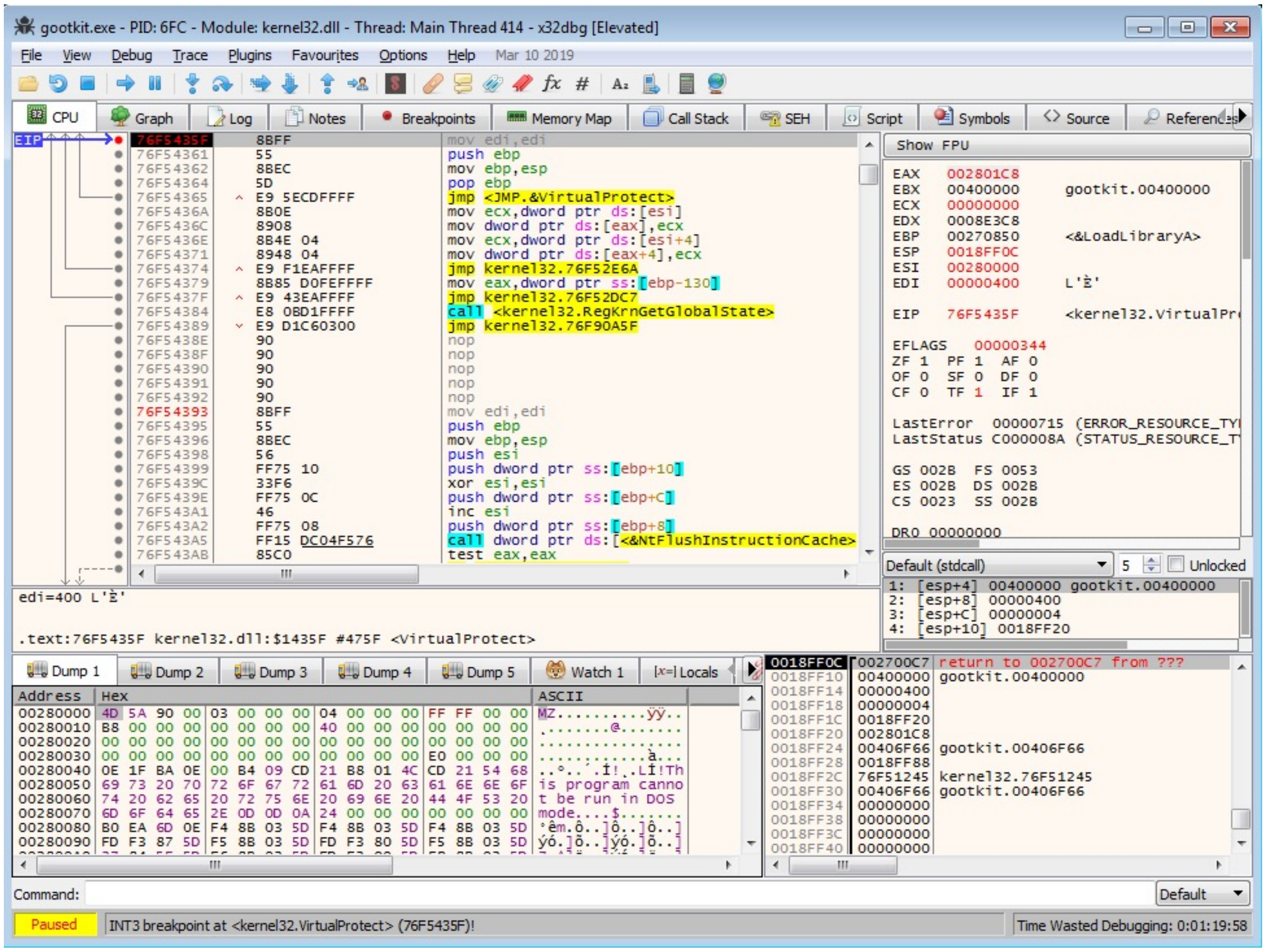
The Gootkit Banking Trojan was discovered back in 2014, and utilizes the Node.JS library to perform a range of malicious tasks, from website injections and password grabbing, all the way up to video recording and remote VNC capabilities. Since its discovery in 2014, the actors behind Gootkit have continued to update the codebase to slow down analysis and thwart automated sandboxes. This post will take a look into the first stage of Gootkit, which contains the unpacking phase and a malicious downloader that sets up the infected system, and its multiple anti-analysis mechanisms.

## Unpacking

**MD5 of Packed Sample:** `0b50ae28e1c6945d23f59dd2e17b5632`
With this specific sample, the unpacking routine is fairly trivial, as it performs self-injection. Simply put, the unpacker will:

> Allocate a region of memory -> Decrypt shellcode and copy to the allocated region -> Execute the shellcode, decrypting the first stage Gootkit executable -> Overwrite unpacked with decrypted executable -> Change protections on the decrypted executable and transfer execution to it.

Therefore, in order to unpack it, place breakpoints on both `VirtualAlloc` and `VirtualProtect`, and look out for executable headers appearing in the allocated regions of memory.


gootkit.exe - PID: 6FC - Module: kernel32.dll - Thread: Main Thread 414 - x32dbg [Elevated]
File View Debug Trace Plugins Favourites Options Help Mar 10 2019
CPU Graph Log Notes Breakpoints Memory Map Call Stack SEH Script Symbols Source References
Show FPU
Default (stdcall) 5 Unlocked
Dump 1 Dump 2 Dump 3 Dump 4 Dump 5 Watch 1 Locals
Command:
Default

```
EIP  76F5435F   8BFF               mov edi,edi
     76F54361   55                 push ebp
     76F54362   8BEC               mov ebp,esp
     76F54364   5D                 pop ebp
     76F54365 ^ E9 5ECDFFFF        jmp <JMP.&VirtualProtect>
     76F5436A   8B0E               mov ecx,dword ptr ds:[esi]
     76F5436C   8908               mov dword ptr ds:[eax],ecx
     76F5436E   8B4E 04            mov ecx,dword ptr ds:[esi+4]
     76F54371   8948 04            mov dword ptr ds:[eax+4],ecx
     76F54374 ^ E9 F1EAFFFF        jmp kernel32.76F52E6A
     76F54379   8B85 D0FEFFFF      mov eax,dword ptr ss:[ebp-130]
     76F5437F ^ E9 43EAFFFF        jmp kernel32.76F52DC7
     76F54384   E8 0BD1FFFF        call <kernel32.RegKrnGetGlobalState>
     76F54389 v E9 D1C60300        jmp kernel32.76F90A5F
     76F5438E   90                 nop
     76F5438F   90                 nop
     76F54390   90                 nop
     76F54391   90                 nop
     76F54392   90                 nop
     76F54393   8BFF               mov edi,edi
     76F54395   55                 push ebp
     76F54396   8BEC               mov ebp,esp
     76F54398   56                 push esi
     76F54399   FF75 10            push dword ptr ss:[ebp+10]
     76F5439C   33F6               xor esi,esi
     76F5439E   FF75 0C            push dword ptr ss:[ebp+C]
     76F543A1   46                 inc esi
     76F543A2   FF75 08            push dword ptr ss:[ebp+8]
     76F543A5   FF15 DC04F576      call dword ptr ds:[<&NtFlushInstructionCache>]
     76F543AB   85C0               test eax,eax

edi=400 L'È'

.text:76F5435F kernel32.dll:$1435F #475F <VirtualProtect>
```

```
EAX    002801C8
EBX    00400000     gootkit.00400000
ECX    00000000
EDX    0008E3C8
EBP    00270850     <&LoadLibraryA>
ESP    0018FF0C
ESI    00280000
EDI    00000400     L'È'

EIP    76F5435F     <kernel32.VirtualPr

EFLAGS   00000344
ZF 1  PF 1  AF 0
OF 0  SF 0  DF 0
CF 0  TF 1  IF 1

LastError  00000715 (ERROR_RESOURCE_TY
LastStatus C000008A (STATUS_RESOURCE_T

GS 002B  FS 0053
ES 002B  DS 002B
CS 0023  SS 002B

DR0 00000000

1: [esp+4] 00400000 gootkit.00400000
2: [esp+8] 00000400
3: [esp+C] 00000004
4: [esp+10] 0018FF20
```

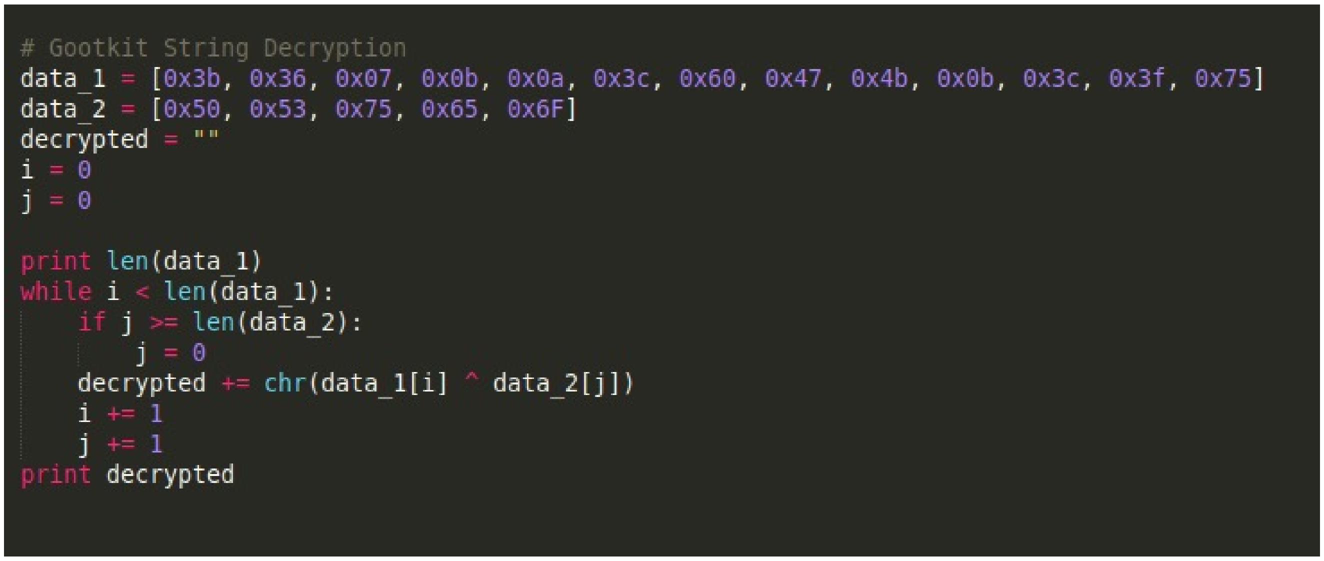
```
Address   Hex                                               ASCII
00280000  4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00  MZ..........ÿÿ..
00280010  B8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00  ¸.......@.......
00280020  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00280030  00 00 00 00 00 00 00 00 00 00 00 00 E0 00 00 00  ............à...
00280040  0E 1F BA 0E 00 B4 09 CD 21 B8 01 4C CD 21 54 68  ..º..´.Í!¸.LÍ!Th
00280050  69 73 20 70 72 6F 67 72 61 6D 20 63 61 6E 6E 6F  is program canno
00280060  74 20 62 65 20 72 75 6E 20 69 6E 20 44 4F 53 20  t be run in DOS 
00280070  6D 6F 64 65 2E 0D 0D 0A 24 00 00 00 00 00 00 00  mode....$.......
00280080  B0 EA 6D 0E F4 8B 03 5D F4 8B 03 5D F4 8B 03 5D  °êm.ô..]ô..]ô..]
00280090  FD F3 87 5D F5 8B 03 5D FD F3 80 5D F5 8B 03 5D  ýó.]õ..]ýó.]õ..]
```

```
0018FF0C  002700C7  return to 002700C7 from ???
0018FF10  00400000  gootkit.00400000
0018FF14  00000400
0018FF18  00000004
0018FF1C  0018FF20
0018FF20  002801C8
0018FF24  00406F66  gootkit.00406F66
0018FF28  0018FF88
0018FF2C  76F51245  kernel32.76F51245
0018FF30  00406F66  gootkit.00406F66
0018FF34  00000000
0018FF38  00000000
0018FF3C  00000000
0018FF40  00000000
```

```
Paused    INT3 breakpoint at <kernel32.VirtualProtect> (76F5435F)!    Time Wasted Debugging: 0:01:19:58
```

**MD5 of Unpacked Sample:** `c342af62302936720e52679bc431d5a8`
Immediately upon opening the sample in IDA, you'll notice the use of the `CreateThread` API – this is used excessively throughout the binary, potentially as an anti-dynamic analysis method. It becomes quite difficult to debug the program due to the fact that multiple threads are running at once; however, this can be avoided by focusing on one thread per execution. Static analysis methods are also hindered, due to the levels of obfuscation utilized by the sample. Whilst there are quite a few strings in plaintext, nearly all of the important strings used are decrypted at run time, using a simple but effective XOR algorithm. Not only are the strings encrypted, they are also stored as stack strings, making it more complex to extract

the important data.
As mentioned previously, the algorithm is fairly simple. Essentially what happens is there are 2 different “strings”. The first string (typically shorter), will loop around, XOR’ing each byte with a byte of the second string. An example of this algorithm in Python can be seen below.



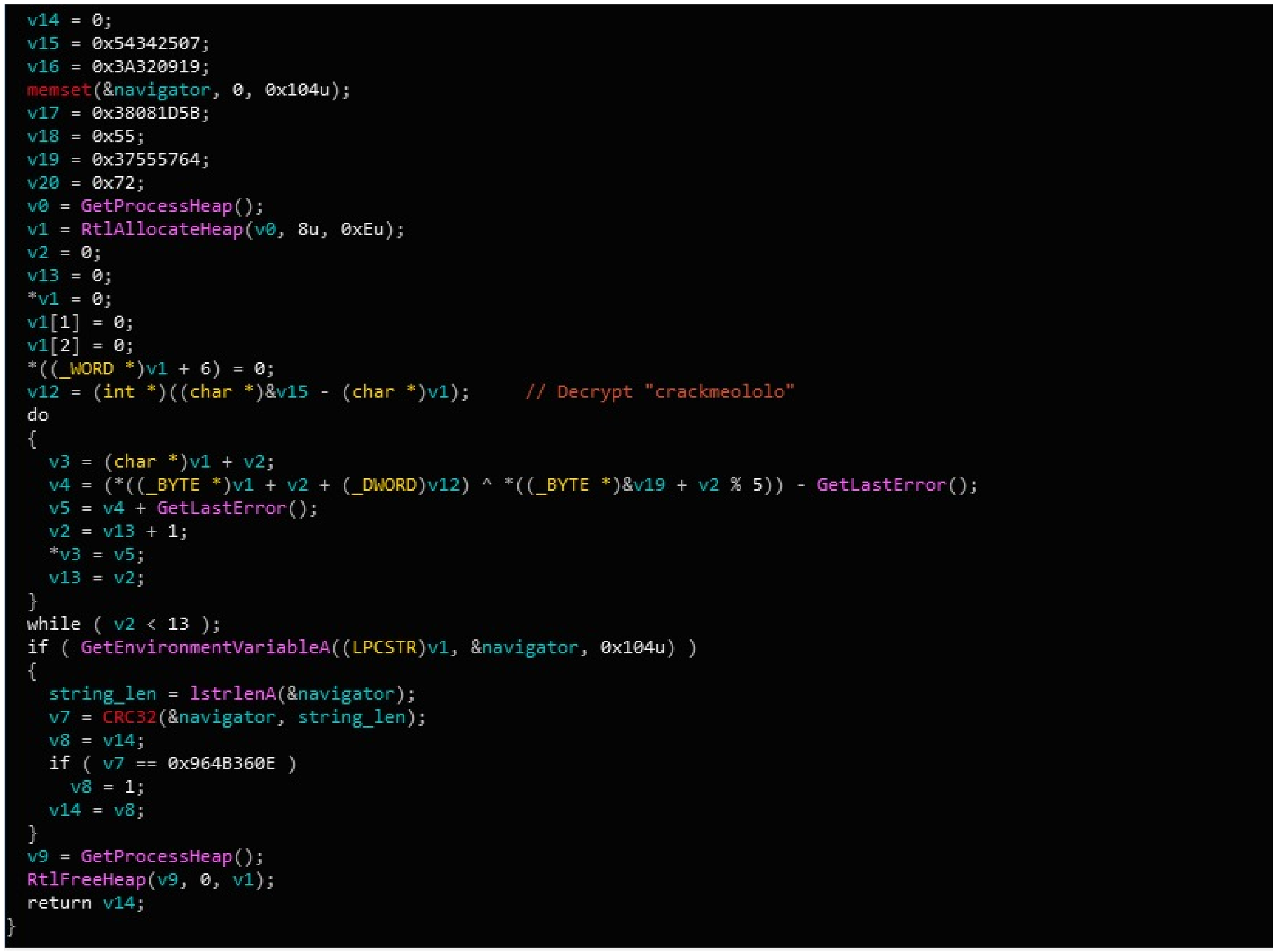
```python
# Gootkit String Decryption
data_1 = [0x3b, 0x36, 0x07, 0x0b, 0x0a, 0x3c, 0x60, 0x47, 0x4b, 0x0b, 0x3c, 0x3f, 0x75]
data_2 = [0x50, 0x53, 0x75, 0x65, 0x6F]
decrypted = ""
i = 0
j = 0

print len(data_1)
while i < len(data_1):
    if j >= len(data_2):
        j = 0
    decrypted += chr(data_1[i] ^ data_2[j])
    i += 1
    j += 1
print decrypted
```

The example above will return the string `kernel32.dll`.
Before Gootkit begins to perform its malicious routines, it first checks the arguments passed to it – this determines the path it follows. The possible arguments that Gootkit accepts are:
`--reinstall`
`--service`
`-test`
`--vwxyz`
If no argument is given, Gootkit will perform a setup routine, and then execute itself with the `--vwxyz` argument. The `-test` argument simply causes the process to exit, whereas the `--reinstall` argument will reinstall Gootkit using the persistence method that we will be covering in the next post. Finally, the `--service` argument will simply set an additional environment variable, specifically the variable name *USERNAME_REQUIRED*, with the value set as *TRUE*. In this post we, will be focusing primarily on the setup phase, to understand the steps Gootkit takes before executing itself with the `--vwxyz` argument.

## Anti-Analysis Functionality

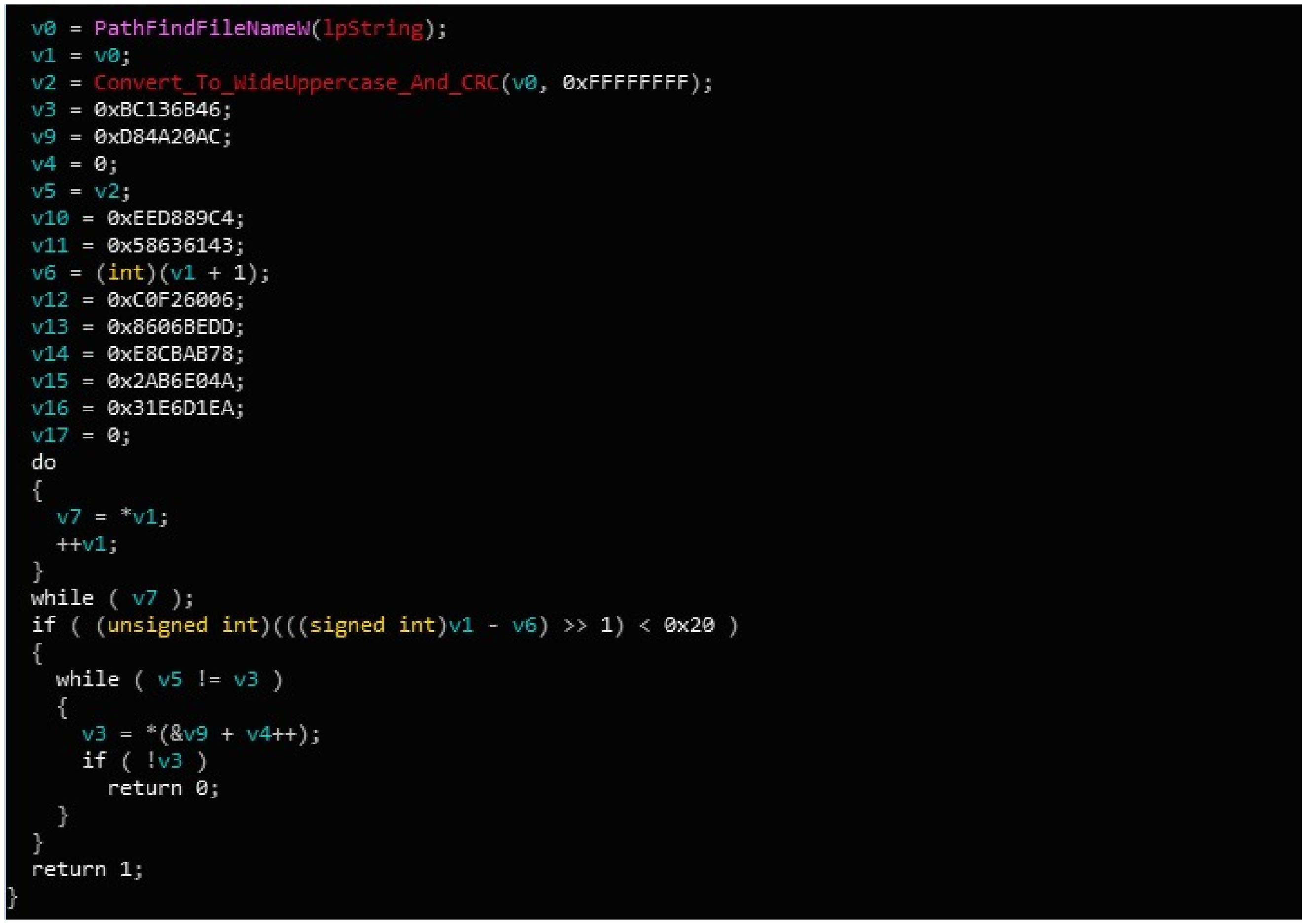
As mentioned previously, Gootkit packs plenty of Anti-Analysis features to evade sandboxes, prevent execution in a Virtual Machine, and slow down analysis. Interestingly, the functions responsible for these features are skipped if a specific environment variable is set. The variable that is set during runtime is named `crackmeololo`, and the value given to it is `navigator`. When it comes to checking the value, rather than compare it to a string, Gootkit will utilize CRC-32/JAMCRC hashing in order to check the validity. If the CRC hashes don't match, the system checks begin.



```
  v14 = 0;
  v15 = 0x54342507;
  v16 = 0x3A320919;
  memset(&navigator, 0, 0x104u);
  v17 = 0x38081D5B;
  v18 = 0x55;
  v19 = 0x37555764;
  v20 = 0x72;
  v0 = GetProcessHeap();
  v1 = RtlAllocateHeap(v0, 8u, 0xEu);
  v2 = 0;
  v13 = 0;
  *v1 = 0;
  v1[1] = 0;
  v1[2] = 0;
  *((_WORD *)v1 + 6) = 0;
  v12 = (int *)((char *)&v15 - (char *)v1);     // Decrypt "crackmeololo"
  do
  {
    v3 = (char *)v1 + v2;
    v4 = (*((_BYTE *)v1 + v2 + (_DWORD)v12) ^ *((_BYTE *)&v19 + v2 % 5)) - GetLastError();
    v5 = v4 + GetLastError();
    v2 = v13 + 1;
    *v3 = v5;
    v13 = v2;
  }
  while ( v2 < 13 );
  if ( GetEnvironmentVariableA((LPCSTR)v1, &navigator, 0x104u) )
  {
    string_len = lstrlenA(&navigator);
    v7 = CRC32(&navigator, string_len);
    v8 = v14;
    if ( v7 == 0x964B360E )
      v8 = 1;
    v14 = v8;
  }
  v9 = GetProcessHeap();
  RtlFreeHeap(v9, 0, v1);
  return v14;
}
```

The first check that Gootkit performs is a filename check. Simply put, there is a hardcoded list of CRC hashed filenames inside the binary, which are compared against the hash of the current filename. If a match is found, Gootkit will create a batch file that will delete the original executable. The process will then exit. A list of the filenames that Gootkit searches for can be seen below.

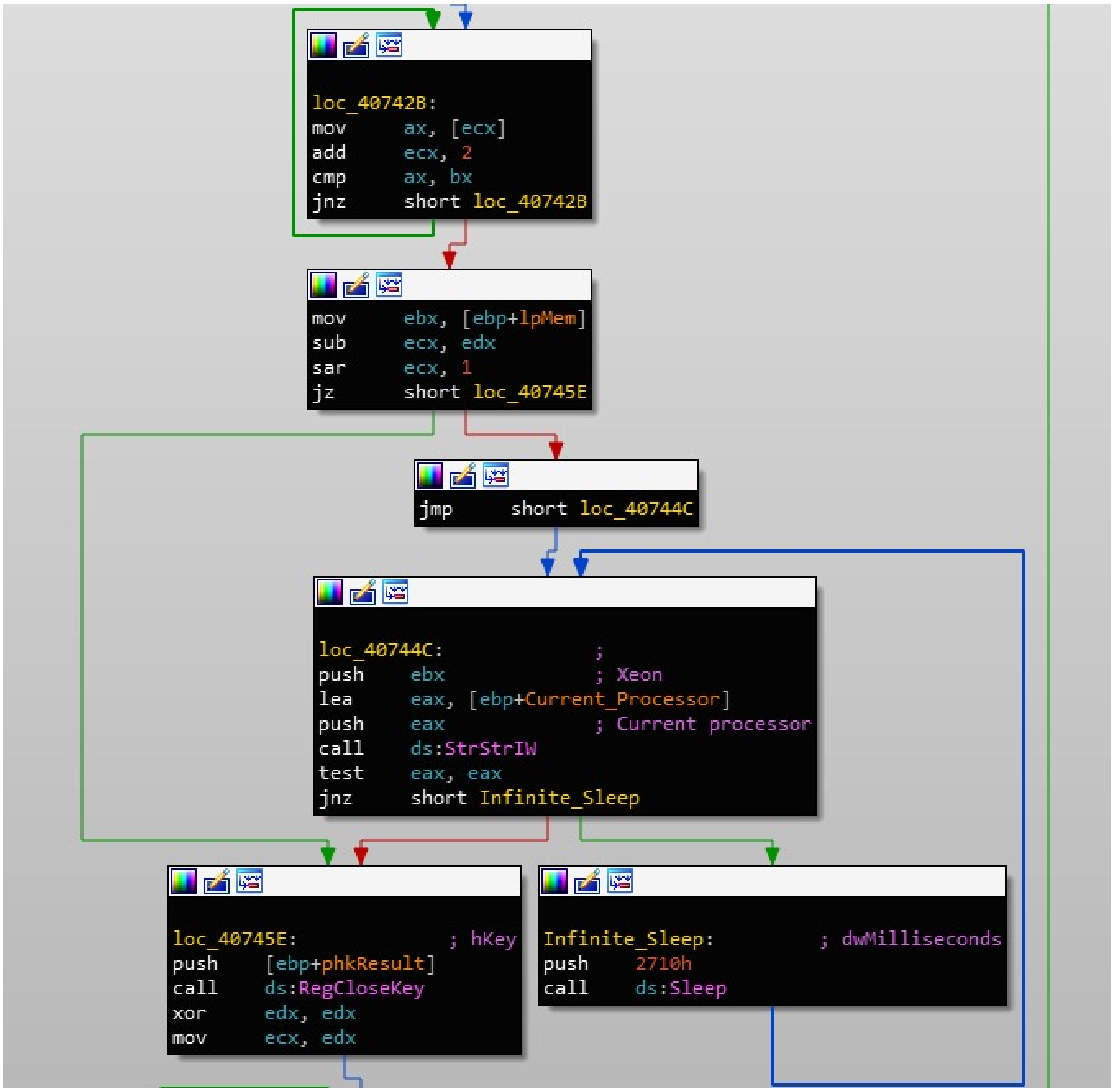
```
SAMPLE.EXE
MALWARE.EXE
BOT.EXE
SANDBOX.EXE
TEST.EXE
KLAVME.EXE
MYAPP.EXE
TESTAPP.EXE
```



```
  v0 = PathFindFileNameW(lpString);
  v1 = v0;
  v2 = Convert_To_WideUppercase_And_CRC(v0, 0xFFFFFFFF);
  v3 = 0xBC136B46;
  v9 = 0xD84A20AC;
  v4 = 0;
  v5 = v2;
  v10 = 0xEED889C4;
  v11 = 0x58636143;
  v6 = (int)(v1 + 1);
  v12 = 0xC0F26006;
  v13 = 0x8606BEDD;
  v14 = 0xE8CBAB78;
  v15 = 0x2AB6E04A;
  v16 = 0x31E6D1EA;
  v17 = 0;
  do
  {
    v7 = *v1;
    ++v1;
  }
  while ( v7 );
  if ( (unsigned int)(((signed int)v1 - v6) >> 1) < 0x20 )
  {
    while ( v5 != v3 )
    {
      v3 = *(&v9 + v4++);
      if ( !v3 )
        return 0;
    }
  }
  return 1;
}
```

The next checks are performed almost immediately after the filename check. Gootkit will create another thread, where it will output the string "*MP3 file corrupted*" using `OutputDebugStringA`, and then check the environment variable `crackmeololo` once again. If the CRC hashes match, it will continue on to decrypt the on board configuration – if not, it will perform a more in depth check of the environment.

First, it begins by opening the registry key *HardwareDESCRIPTIONSystemCentralProcessor0*, and then queries the *ProcessorNameString*, comparing the value to *Xeon*. The *Xeon* processor is used in servers primarily, and not in laptops or desktops. This is a good indicator that the malware is running in a sandbox, so if it is detected, Gootkit will enter an endless sleep-loop cycle.



```
loc_40742B:
mov     ax, [ecx]
add     ecx, 2
cmp     ax, bx
jnz     short loc_40742B
```

```
mov     ebx, [ebp+lpMem]
sub     ecx, edx
sar     ecx, 1
jz      short loc_40745E
```

```
jmp     short loc_40744C
```

```
loc_40744C:                 ;
push    ebx                 ; Xeon
lea     eax, [ebp+Current_Processor]
push    eax                 ; Current processor
call    ds:StrStrIW
test    eax, eax
jnz     short Infinite_Sleep
```

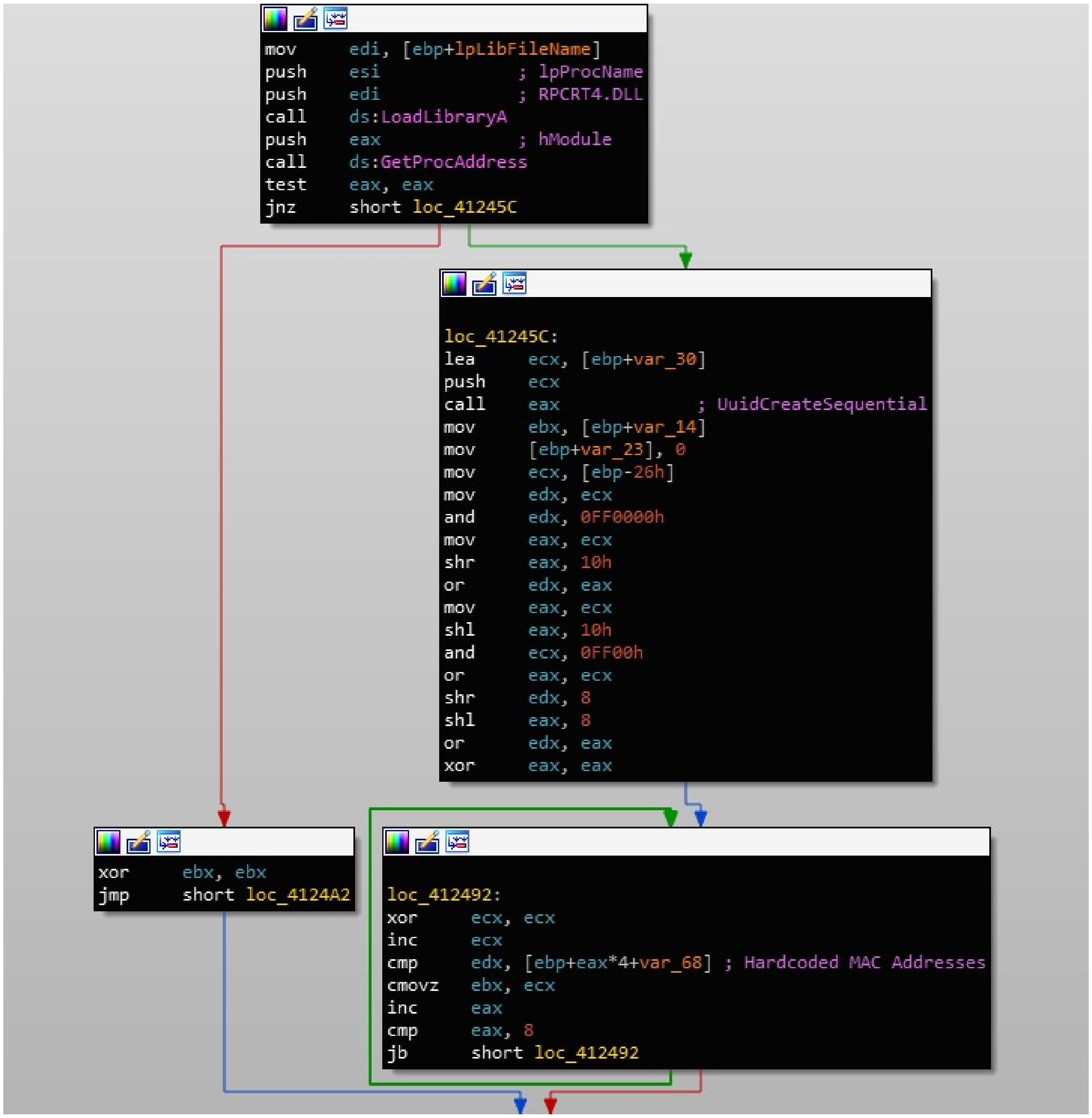
```
loc_40745E:                 ; hKey
push    [ebp+phkResult]
call    ds:RegCloseKey
xor     edx, edx
mov     ecx, edx
```

```
Infinite_Sleep:             ; dwMilliseconds
push    2710h
call    ds:Sleep
```

If *Xeon* is not detected, execution will resume; however, the next check is a lot more intensive. Similar to the filename check, Gootkit also contains a hardcoded list of MAC address identifiers used to detect sandboxes or VMs. After loading *RPCRT4.DLL*, it will call `UuidCreateSequential`, which uses the MAC Address to create a GUID. If any of the values match, it will enter an infinite sleep-loop cycle once again. A list of the hardcoded MAC Addresses along with the corresponding vendors can be seen below.

| | |
|---|---|
| F01FAF00 | Dell |
| 00505600 | VMWare |
| 8002700 | PCS System Technology GmbH |

| | |
|---|---|
| 000C2900 | VMWare |
| 00056900 | VMWare |
| 0003FF00 | Microsoft |
| 001C4200 | Parallels |
| 00163E00 | XenSource |



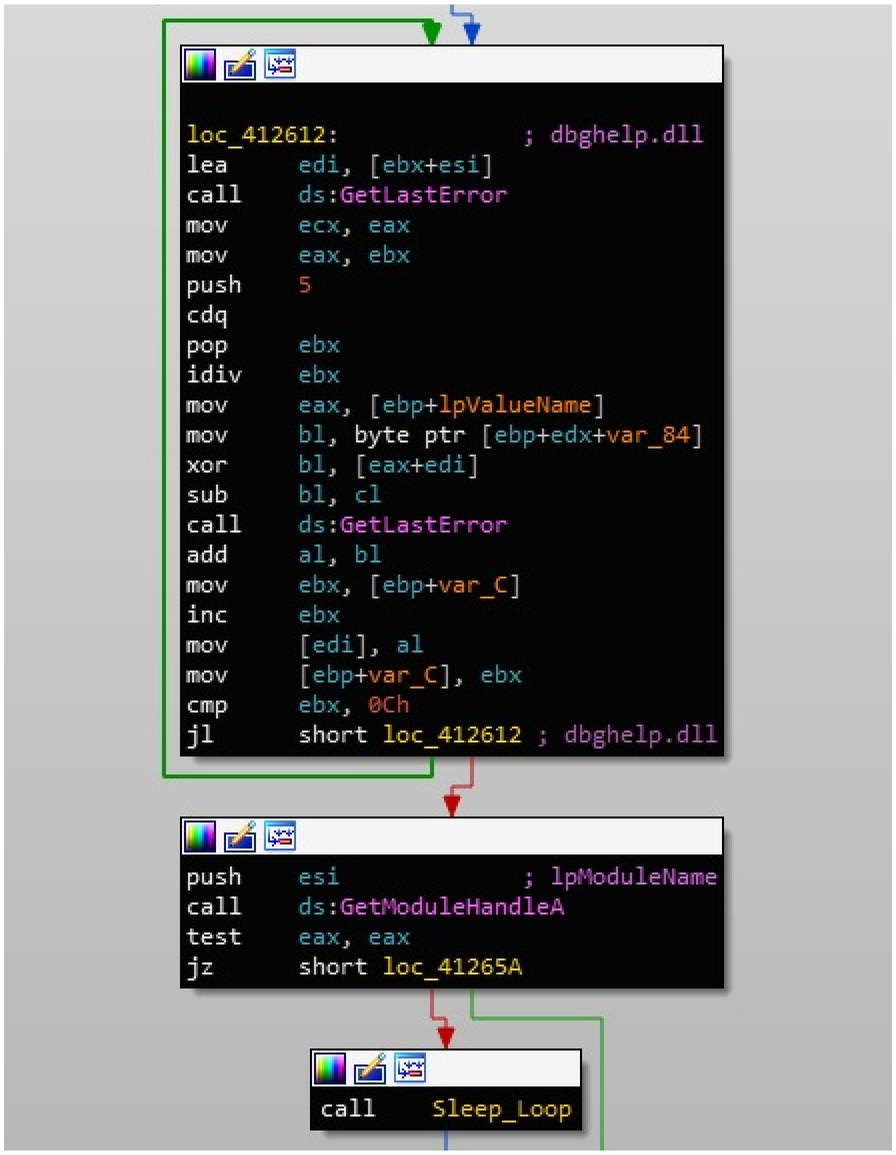
```
mov     edi, [ebp+lpLibFileName]
push    esi             ; lpProcName
push    edi             ; RPCRT4.DLL
call    ds:LoadLibraryA
push    eax             ; hModule
call    ds:GetProcAddress
test    eax, eax
jnz     short loc_41245C
```

```
loc_41245C:
lea     ecx, [ebp+var_30]
push    ecx
call    eax                 ; UuidCreateSequential
mov     ebx, [ebp+var_14]
mov     [ebp+var_23], 0
mov     ecx, [ebp-26h]
mov     edx, ecx
and     edx, 0FF0000h
mov     eax, ecx
shr     eax, 10h
or      edx, eax
mov     eax, ecx
shl     eax, 10h
and     ecx, 0FF00h
or      eax, ecx
shr     edx, 8
shl     eax, 8
or      edx, eax
xor     eax, eax
```

```
xor     ebx, ebx
jmp     short loc_4124A2
```

```
loc_412492:
xor     ecx, ecx
inc     ecx
cmp     edx, [ebp+eax*4+var_68] ; Hardcoded MAC Addresses
cmovz   ebx, ecx
inc     eax
cmp     eax, 8
jb      short loc_412492
```

Next, Gootkit will call `GetModuleHandleA` in an attempt to get a handle to either *dbghelp.dll* and *sbiedll.dll*, in an attempt to detect a present debugger or the sandbox *Sandboxie*. If a handle is returned successfully, an infinite sleep cycle will occur. Continuing on, the current username will be retrieved with a call to `GetUserNameA`, and compared to *CurrentUser* and *Sandbox*. The computer name will then be retrieved and compared to *SANDBOX* and *7SILVIA*. As you may have guessed, if any of these match, the sample will enter into an infinite sleep cycle.

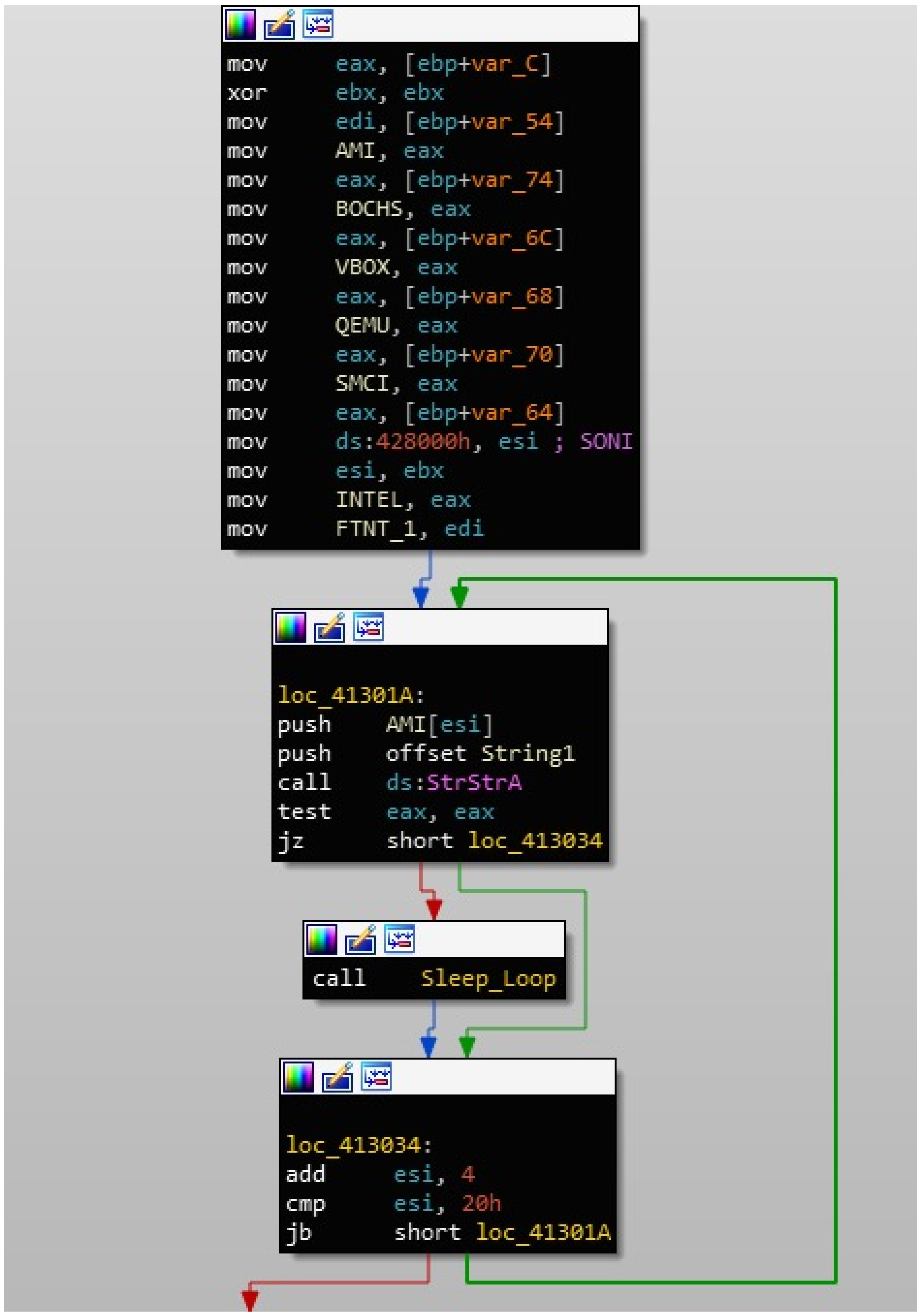


```
loc_412612:                 ; dbghelp.dll
lea     edi, [ebx+esi]
call    ds:GetLastError
mov     ecx, eax
mov     eax, ebx
push    5
cdq
pop     ebx
idiv    ebx
mov     eax, [ebp+lpValueName]
mov     bl, byte ptr [ebp+edx+var_84]
xor     bl, [eax+edi]
sub     bl, cl
call    ds:GetLastError
add     al, bl
mov     ebx, [ebp+var_C]
inc     ebx
mov     [edi], al
mov     [ebp+var_C], ebx
cmp     ebx, 0Ch
jl      short loc_412612 ; dbghelp.dll
```

```
push    esi                 ; lpModuleName
call    ds:GetModuleHandleA
test    eax, eax
jz      short loc_41265A
```

```
call    Sleep_Loop
```

Continuing on, Gootkit will query *HARDWAREDESCRIPTIONSystemSystemBiosVersion* and compare the value to; *AMI*, *BOCHS*, *VBOX*, *QEMU*, *SMCI*, *INTEL – 6040000*, *FTNT-1*, and *SONI*. Once again, match = infinite sleep cycle.



```
mov     eax, [ebp+var_C]
xor     ebx, ebx
mov     edi, [ebp+var_54]
mov     AMI, eax
mov     eax, [ebp+var_74]
mov     BOCHS, eax
mov     eax, [ebp+var_6C]
mov     VBOX, eax
mov     eax, [ebp+var_68]
mov     QEMU, eax
mov     eax, [ebp+var_70]
mov     SMCI, eax
mov     eax, [ebp+var_64]
mov     ds:428000h, esi ; SONI
mov     esi, ebx
mov     INTEL, eax
mov     FTNT_1, edi

loc_41301A:
push    AMI[esi]
push    offset String1
call    ds:StrStrA
test    eax, eax
jz      short loc_413034

call    Sleep_Loop

loc_413034:
add     esi, 4
cmp     esi, 20h
jb      short loc_41301A
```

Yet another registry query is performed, this time with the key *HARDWAREDescriptionSystemVideoBiosVersion*, with the value being compared to *VirtualBox*. Finally, it queries *SOFTWAREMicrosoftWindowsCurrentVersionSystemBiosVersion* or *HARDWAREDESCRIPTIONSystemSystemBiosVersion* for 3 values that correspond to *Joe Sandbox* and *CWSandbox*:

```
55274-640-2673064-23950: Joe Sandbox
76487-644-3177037-23510: CWSandbox
76487-337-8429955-22614: CWSandbox
```

If all checks are passed, then execution of the sample will continue, by setting up persistence and retrieving the payload from the C2 server. Before doing that, it will check its filename once again, using the same CRC hashing we saw earlier.
In the next post, we will take a look at the persistence method used by Gootkit, and take a look at the `--reinstall` pathway, as well as the communications routine used by the sample to retrieve the final stage.