Infect If Needed | A Deeper Dive Into Targeted Backdoor macOS.Macma

sentinelone.com/labs/infect-if-needed-a-deeper-dive-into-targeted-backdoor-macos-macma/

Phil Stokes

SentinelLABS
Infect If Needed | A Deeper Dive Into Targeted Backdoor macOS Macma
By Phil Stokes
READ BLOG >

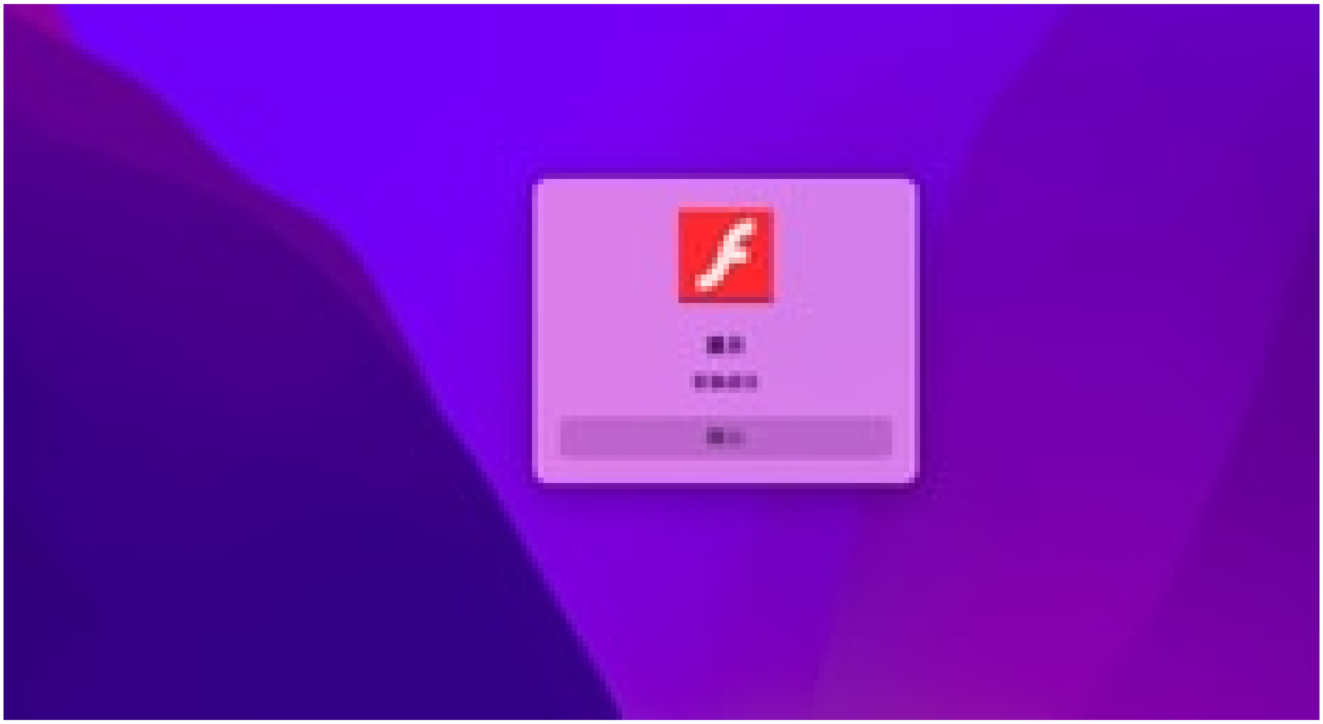
Last week, Google's Threat Analysis Group published details around what appears to be APT activity targeting, among others, Mac users visiting Hong Kong websites supporting pro-democracy activism. Google's report focused on the use of two vulnerabilities: a zero day and a N-day (a known vulnerability with an available patch).

By the time of Google's publication both had, in fact, been patched for some months. What received less attention was the malware that the vulnerabilities were leveraged to drop: a backdoor that works just fine even on the latest patched systems of macOS Monterey.

Google labelled the backdoor "Macma", and we will follow suit. Shortly after Google's publication, a rapid triage of the backdoor was published by Objective-See (under the name "OSX.CDDS"). In this post, we take a deeper dive into macOS.Macma, reveal further IoCs to aid defenders and threat hunters, and speculate on some of macOS.Macma's (hitherto-unmentioned) interesting artifacts.

How macOS.Macma Gains Persistence

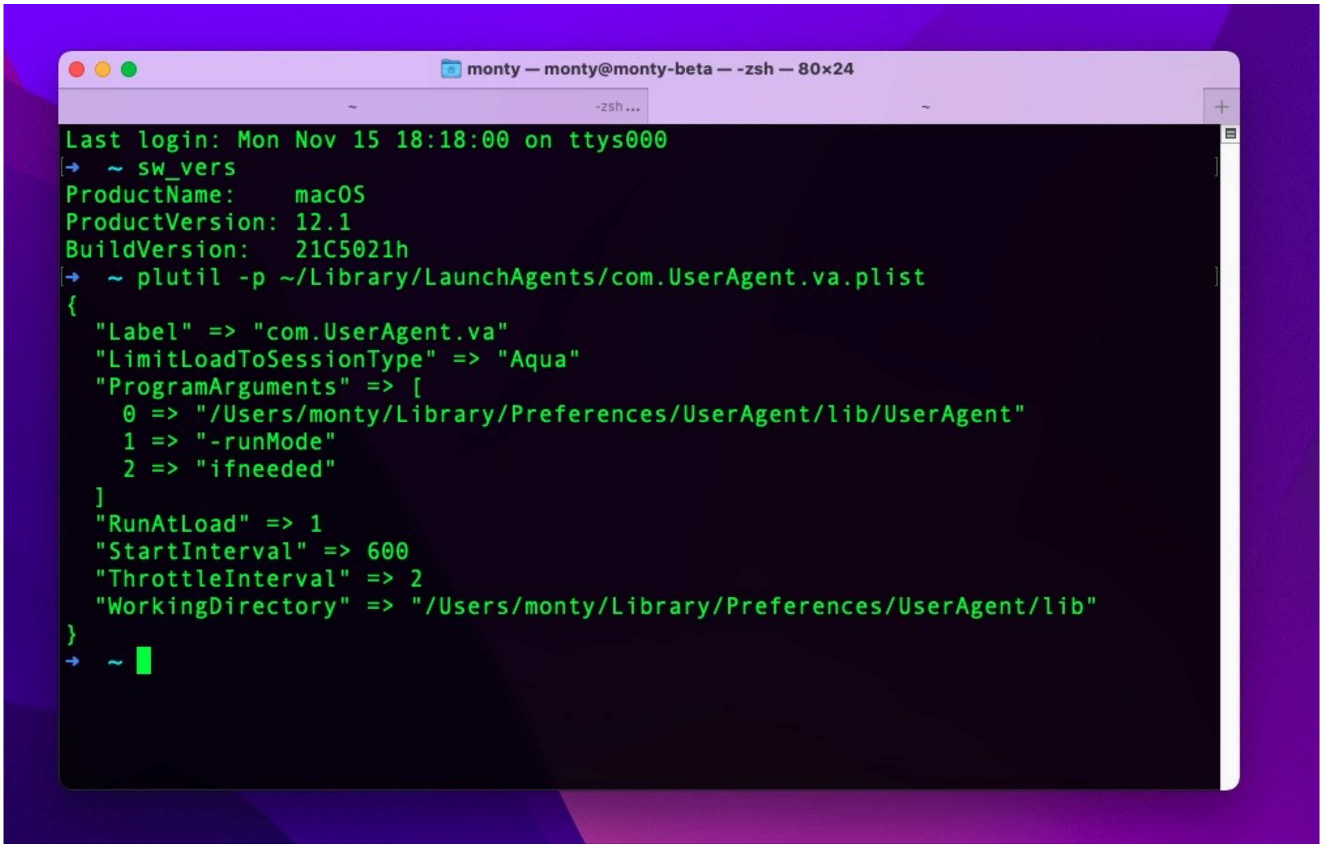
Thanks to the work of Google's TAG team, we were able to grab two versions of the backdoor used by the threat actors, which we will label `UserAgent 2019` and `UserAgent 2021`. Both are interesting, but arguably the earlier 2019 version has greater longevity since the delivery mechanism appears to work just fine on macOS Monterey.



The 2019 version of macOS.Macma will run just fine on macOS Monterey

`UserAgent 2019` is a Mach-O binary dropped by an application called "SafariFlashActivity.app", itself contained in a .DMG file (the disk image sample found by Google has the name "install_flash_player_osx.dmg"). `UserAgent 2021` is a standalone Mach-O binary and contains much the same functionality as the 2019 version along with some added AV capture capabilities. This version of macOS.Macma is installed by a separate Mach-O binary dropped when the threat actors leverage the vulnerabilities described in Google's post.

Both versions install the same persistence agent, `com.UserAgent.va.plist` in the current user's `~/Library/LaunchAgents` folder.



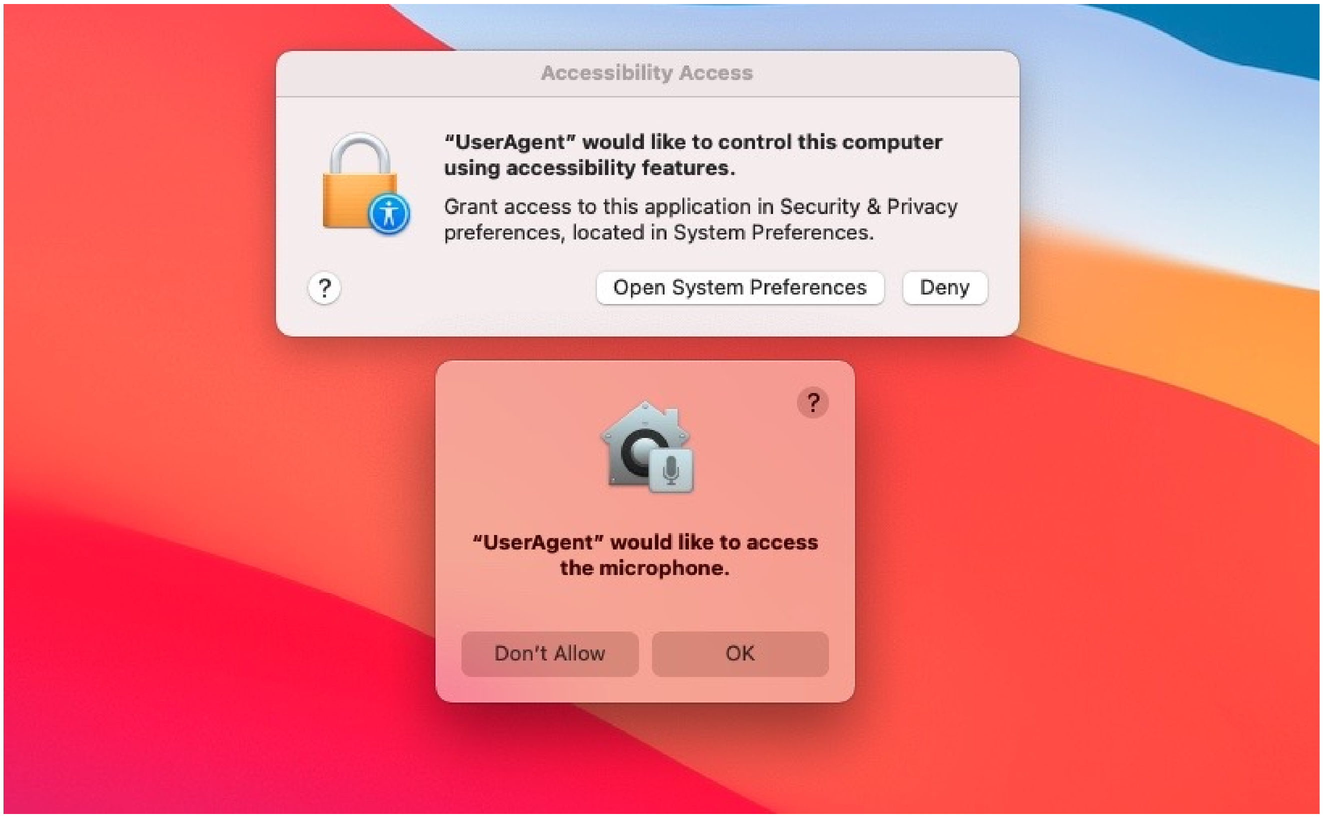
```
Last login: Mon Nov 15 18:18:00 on ttys000
➜  ~ sw_vers
ProductName:    macOS
ProductVersion: 12.1
BuildVersion:   21C5021h
➜  ~ plutil -p ~/Library/LaunchAgents/com.UserAgent.va.plist
{
  "Label" => "com.UserAgent.va"
  "LimitLoadToSessionType" => "Aqua"
  "ProgramArguments" => [
    0 => "/Users/monty/Library/Preferences/UserAgent/lib/UserAgent"
    1 => "-runMode"
    2 => "ifneeded"
  ]
  "RunAtLoad" => 1
  "StartInterval" => 600
  "ThrottleInterval" => 2
  "WorkingDirectory" => "/Users/monty/Library/Preferences/UserAgent/lib"
}
➜  ~ 
```

Macma’s persistence agent, com.UserAgent.va.plist

The property list is worth pausing over as it contains some interesting features. First, aside from the path to the executable, we can see that the persistence agent passes two arguments to the malware before it is run: `-runMode`, and `ifneeded`.

The agent also switches the current working directory to a custom folder, in which later will be deposited data from the separate keylogger module, among other things.

We find it interesting that the developer chose to include the `LimitLoadToSessionType` key with the value “Aqua”. The “Aqua” value ensures the LaunchAgent only runs when there is a logged in GUI user (as opposed to running as a background task or running when a user logs in via SSH). This is likely necessary to ensure other functionality, such as requesting that the user gives access to the Microphone and Accessibility features.



"UserAgent" would like to control this computer using accessibility features.

Grant access to this application in Security & Privacy preferences, located in System Preferences.

"UserAgent" would like to access the microphone.

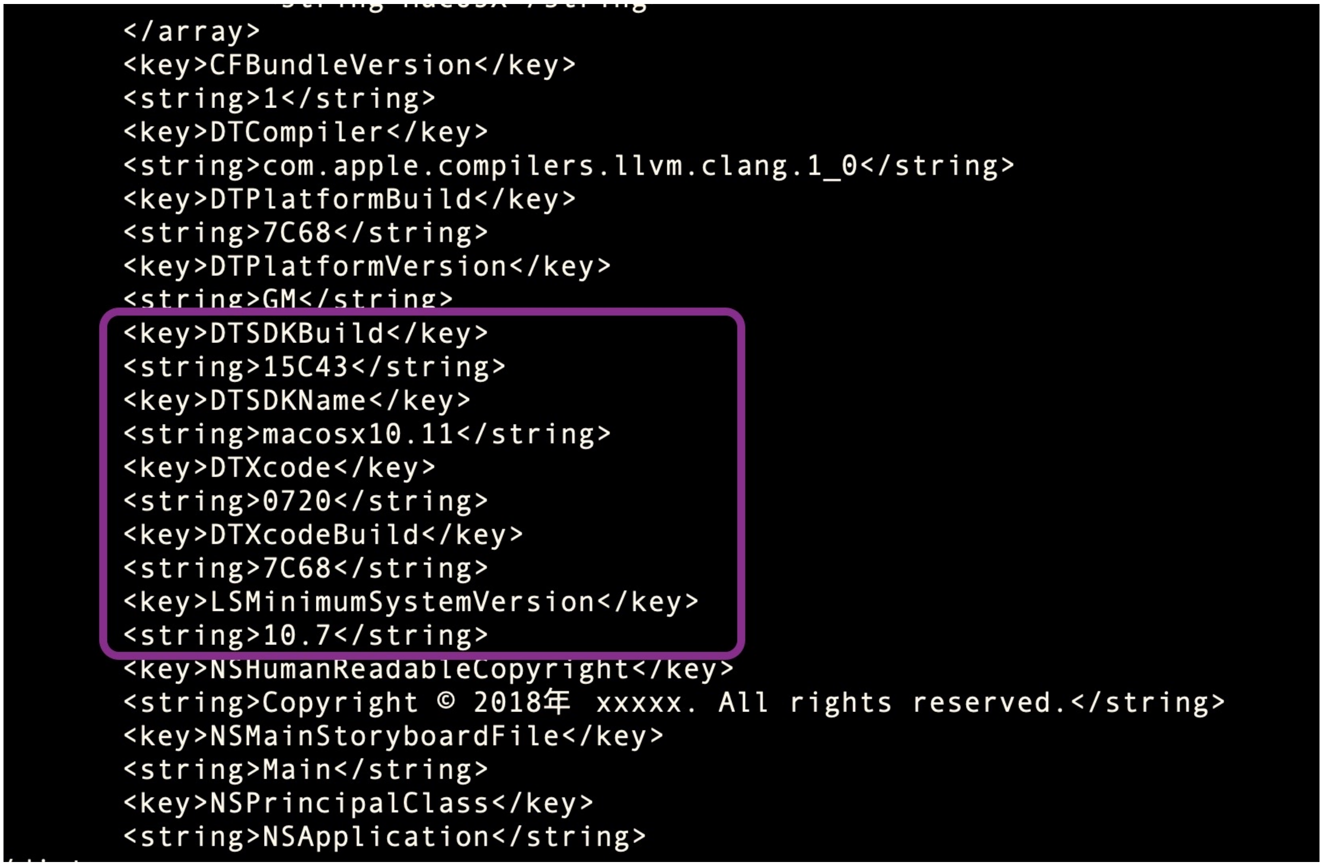
Victims are prompted to allow macOS.Macma access to the Microphone

However, since `launchd` defaults to "Aqua" when no key is specified at all, this inclusion is rather redundant. We might speculate that the inclusion of the key here suggests the developer is familiar with developing other LaunchAgents in other contexts where other keys are indeed necessary.

Application Bundle Confusion Suggests A "Messy" Development Process

Since we are discussing property lists, there's some interesting artifacts in the SafariFlashActivity.app's Info.plist, and that in turn led us to notice a number of other oddities in the bundle executables.

One of the great things about finding malware built into a bundle with an Info.plist is it gives away some interesting details about when, and on what machine, the malware was built.



```
        </array>
        <key>CFBundleVersion</key>
        <string>1</string>
        <key>DTCompiler</key>
        <string>com.apple.compilers.llvm.clang.1_0</string>
        <key>DTPlatformBuild</key>
        <string>7C68</string>
        <key>DTPlatformVersion</key>
        <string>GM</string>
        <key>DTSDKBuild</key>
        <string>15C43</string>
        <key>DTSDKName</key>
        <string>macosx10.11</string>
        <key>DTXcode</key>
        <string>0720</string>
        <key>DTXcodeBuild</key>
        <string>7C68</string>
        <key>LSMinimumSystemVersion</key>
        <string>10.7</string>
        <key>NSHumanReadableCopyright</key>
        <string>Copyright © 2018年 xxxxx. All rights reserved.</string>
        <key>NSMainStoryboardFile</key>
        <string>Main</string>
        <key>NSPrincipalClass</key>
        <string>NSApplication</string>
```

macOS.Macma was built on El Capitan

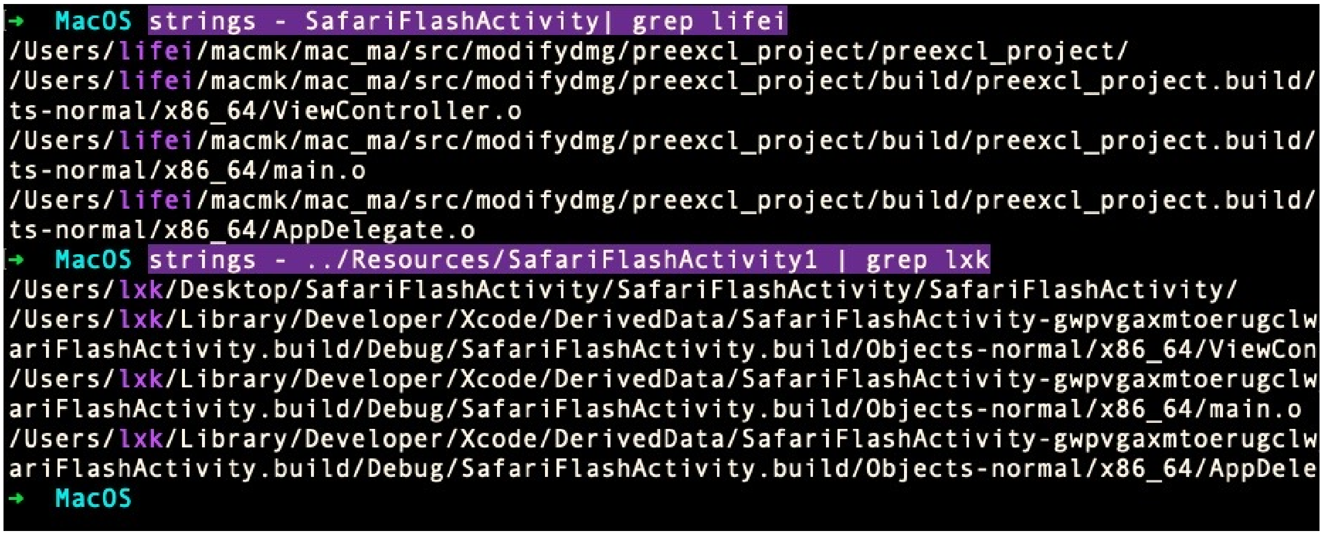
In this case, we see the malware was built on an El Capitan machine running build 15C43. That’s curious, because build 15C43 was never a public release build: it was a beta of El Capitan 11.2 available to developers and AppleSeed (Apple beta testers) briefly around October to November 2015. On December 8th, 2015, El Capitan 11.2 was released with build number 15C50, superseding the previous public release of 11.1, build 15B42 from October 21st.

At this juncture, let’s note that the malware was signed with an ad hoc signature, meaning it did not require an Apple Developer account or ID to satisfy code signing requirements.

Therein lies an anomaly: the bundle was signed without needing a developer account, but it seems that the macOS version used to create this version of macOS.Macma was indeed sourced from a developer account. Such an account could possibly belong to the author(s); possibly be stolen, or possibly acquired with a fake ID. However, the latter two scenarios seem inconsistent with the *ad hoc* signature. If the developer had a fake or stolen Apple ID, why not codesign it with that for added credibility?

While we’re speculating about the developer or developers’ identities, two other artifacts in the bundle are worthy of mention. The main executable in `../MacOS` is called “SafariFlashActivity” and was apparently compiled on Sept 16th, 2019. In the `../Resources` folder, we see what appears to be an earlier version of the executable, “SafariFlashActivity1”, built some nine days earlier on Sept 7th.

While these two executables share a large amount of code and functionality, there are also a number of differences between them. Perhaps the most intriguing are that they appear – by accident or by design – to have been created by two entirely different users.

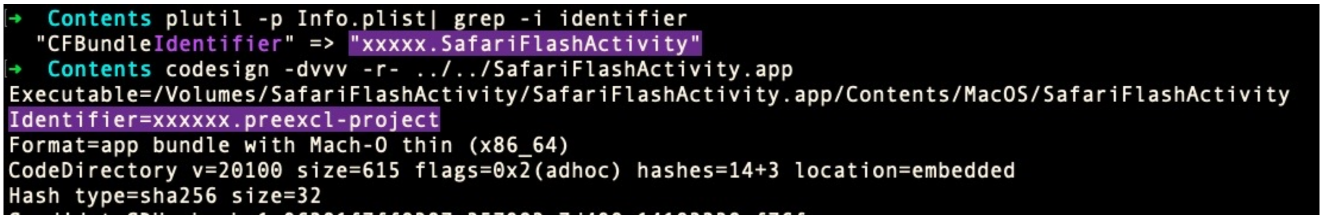


```
→  MacOS strings - SafariFlashActivity| grep lifei
/Users/lifei/macmk/mac_ma/src/modifydmg/preexcl_project/preexcl_project/
/Users/lifei/macmk/mac_ma/src/modifydmg/preexcl_project/build/preexcl_project.build/
ts-normal/x86_64/ViewController.o
/Users/lifei/macmk/mac_ma/src/modifydmg/preexcl_project/build/preexcl_project.build/
ts-normal/x86_64/main.o
/Users/lifei/macmk/mac_ma/src/modifydmg/preexcl_project/build/preexcl_project.build/
ts-normal/x86_64/AppDelegate.o
→  MacOS strings - ../Resources/SafariFlashActivity1 | grep lxk
/Users/lxk/Desktop/SafariFlashActivity/SafariFlashActivity/SafariFlashActivity/
/Users/lxk/Library/Developer/Xcode/DerivedData/SafariFlashActivity-gwpvgaxmtoerugclw
ariFlashActivity.build/Debug/SafariFlashActivity.build/Objects-normal/x86_64/ViewCon
/Users/lxk/Library/Developer/Xcode/DerivedData/SafariFlashActivity-gwpvgaxmtoerugclw
ariFlashActivity.build/Debug/SafariFlashActivity.build/Objects-normal/x86_64/main.o
/Users/lxk/Library/Developer/Xcode/DerivedData/SafariFlashActivity-gwpvgaxmtoerugclw
ariFlashActivity.build/Debug/SafariFlashActivity.build/Objects-normal/x86_64/AppDele
→  MacOS
```

User strings from two binaries in the same macOS.Macma bundle

The user account "lifei" (speculatively, Li Fei, a common-enough Chinese name) seems to have replaced the user account "lxk". Of course, it could be the same person operating different user accounts, or two entirely different individuals building separately from a common project. Indeed, there are sufficiently large differences in the code in such a short space of time to make it plausible to suggest that two developers were working independently on the same project and that one was chosen over the other for the final executable embedded in the `../MacOs` folder.

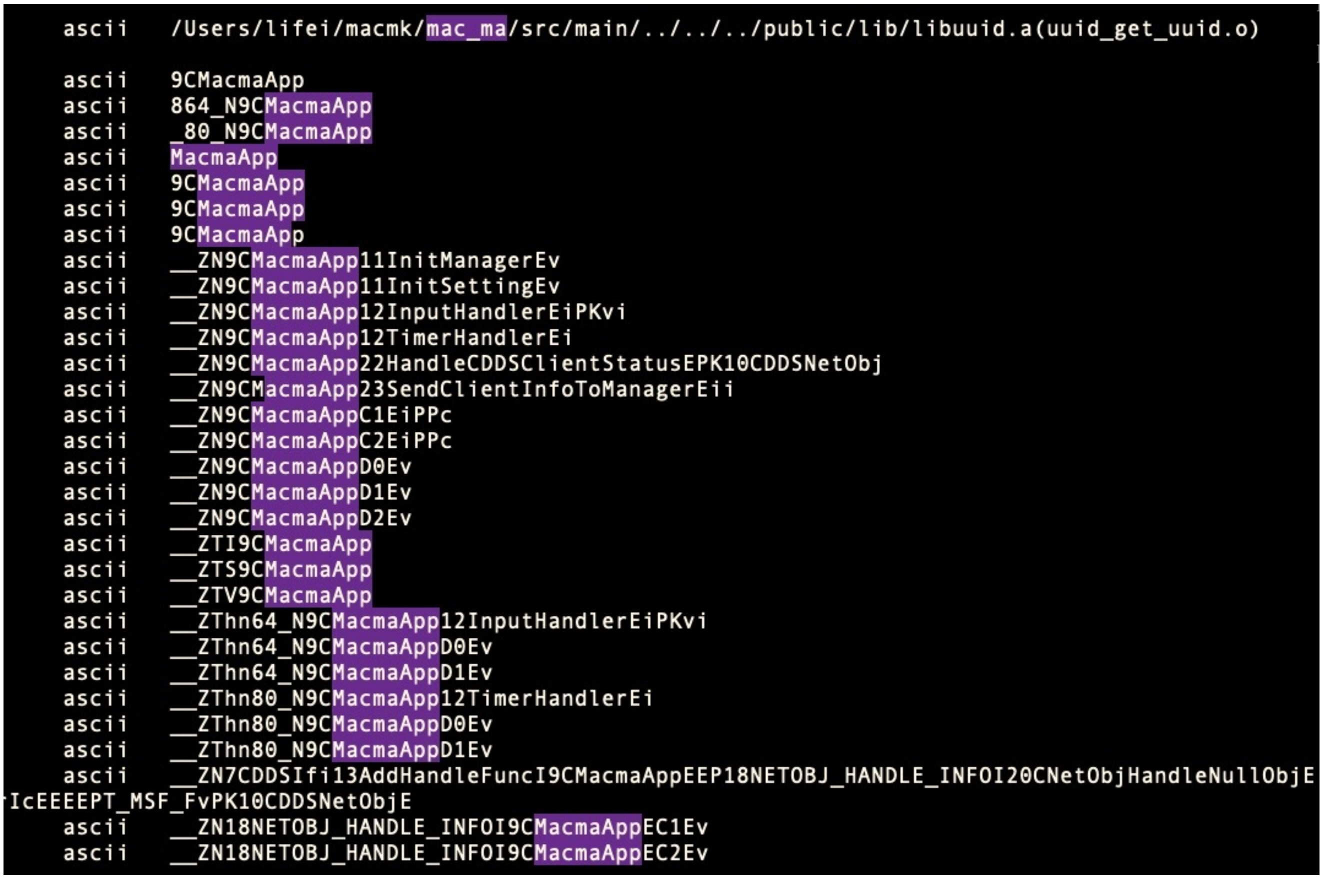
Note that in the "lifei" builds, we see both the use of "Mac_Ma" for the first time, and "preexcel" — used as the team identifier in the final code signature. Neither of these appear in the "lxk" build, where "SafariFlashActivity" appears to be the project name. This bifurcation even extends to an unusual inconsistency between the identifier used in the bundle and that used in the code signature, where one is `xxxxx.SafariFlashActivity` and the other is `xxxxxx.preexcl-project`.



```
→  Contents plutil -p Info.plist| grep -i identifier
  "CFBundleIdentifier" => "xxxxx.SafariFlashActivity"
→  Contents codesign -dvvv -r- ../../SafariFlashActivity.app
Executable=/Volumes/SafariFlashActivity/SafariFlashActivity.app/Contents/MacOS/SafariFlashActivity
Identifier=xxxxxx.preexcl-project
Format=app bundle with Mach-O thin (x86_64)
CodeDirectory v=20100 size=615 flags=0x2(adhoc) hashes=14+3 location=embedded
Hash type=sha256 size=32
```

Inconsistent identifiers used in the bundle and code signature of macOS.Macma

In any case, the string "lifei" is found in several of the other binaries in the 2019 version of macOS.Macma, whereas "lxk" is not seen again. In the 2021 version, both "lifei" and "lxk" and all other developer artifacts have disappeared entirely from both the installer and UserAgent binaries, suggesting that the development process had been deliberately cleaned up.



```
    ascii   /Users/lifei/macmk/mac_ma/src/main/../../../public/lib/libuuid.a(uuid_get_uuid.o)

    ascii   9CMacmaApp
    ascii   864_N9CMacmaApp
    ascii   _80_N9CMacmaApp
    ascii   MacmaApp
    ascii   9CMacmaApp
    ascii   9CMacmaApp
    ascii   9CMacmaApp
    ascii   __ZN9CMacmaApp11InitManagerEv
    ascii   __ZN9CMacmaApp11InitSettingEv
    ascii   __ZN9CMacmaApp12InputHandlerEiPKvi
    ascii   __ZN9CMacmaApp12TimerHandlerEi
    ascii   __ZN9CMacmaApp22HandleCDDSClientStatusEPK10CDDSNetObj
    ascii   __ZN9CMacmaApp23SendClientInfoToManagerEii
    ascii   __ZN9CMacmaAppC1EiPPc
    ascii   __ZN9CMacmaAppC2EiPPc
    ascii   __ZN9CMacmaAppD0Ev
    ascii   __ZN9CMacmaAppD1Ev
    ascii   __ZN9CMacmaAppD2Ev
    ascii   __ZTI9CMacmaApp
    ascii   __ZTS9CMacmaApp
    ascii   __ZTV9CMacmaApp
    ascii   __ZThn64_N9CMacmaApp12InputHandlerEiPKvi
    ascii   __ZThn64_N9CMacmaAppD0Ev
    ascii   __ZThn64_N9CMacmaAppD1Ev
    ascii   __ZThn80_N9CMacmaApp12TimerHandlerEi
    ascii   __ZThn80_N9CMacmaAppD0Ev
    ascii   __ZThn80_N9CMacmaAppD1Ev
    ascii   __ZN7CDDSIfi13AddHandleFuncI9CMacmaAppEEP18NETOBJ_HANDLE_INFOI20CNetObjHandleNullObjE
IcEEEEPT_MSF_FvPK10CDDSNetObjE
    ascii   __ZN18NETOBJ_HANDLE_INFOI9CMacmaAppEC1Ev
    ascii   __ZN18NETOBJ_HANDLE_INFOI9CMacmaAppEC2Ev
```

User lifei’s “Macma” seems to have won the ‘battle of the devs’

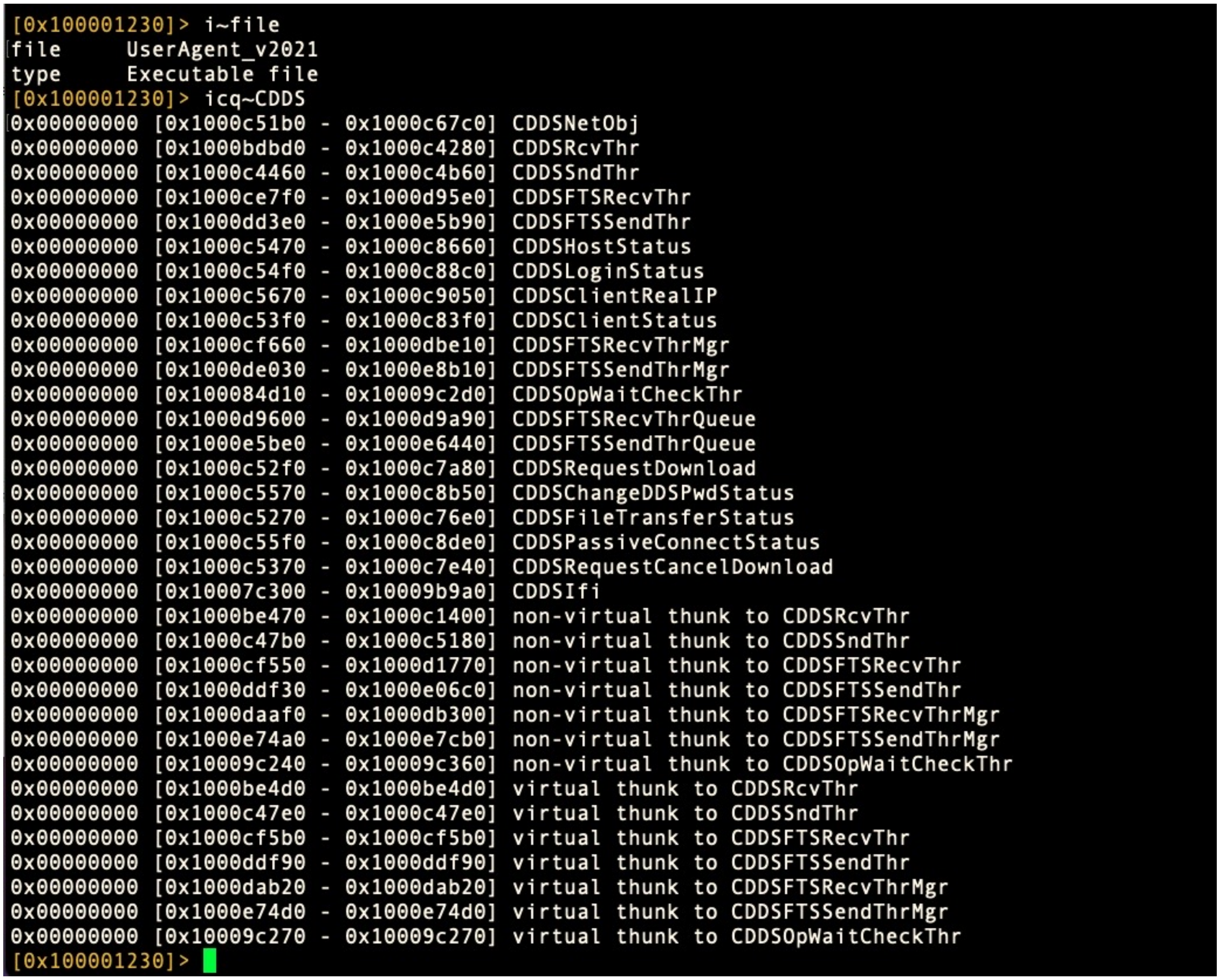
Finally, if we return to the various (admittedly, falsifiable) compilation dates found in the bundle, there is another curiosity: we noted that the malware appears to have been compiled on a 2015 developer build of macOS, yet the Info.plist has a copyright date of 2018, and the executables in this bundle were built well-over 3 years later in September 2019 according to the (entirely manipulatable) timestamps.

What can we conclude from all these tangled weeds? Nothing concrete, admittedly. But there do seem to be two plausible, if competing, narratives: perhaps the threat actor went to extraordinary, and likely unnecessary, lengths to muddle the artifacts in these binaries. Alternatively, the threat actor had a somewhat confused development process with more than one developer and changing requirements. No doubt the truth is far more complex, but given the nature of the artifacts above, we suspect the latter may well be at least part of the story.

For defenders, all this provides a plethora of collectible artifacts that may, perhaps, help us to identify this malware or track this threat actor in future incidents.

macOS.Macma – Links To Android and Linux Malware?

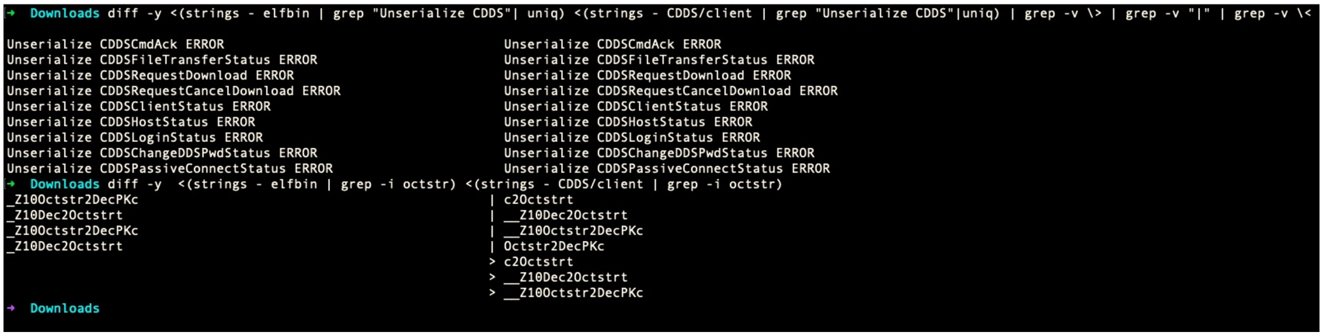
Things start to get even more interesting when we take a look at artifacts in the executable code itself. As we noted in the introduction, an early report on this malware dubbed it “OSX.CDDS”. We can see why. The code is littered with methods prefixed with CDDS.



```
[0x100001230]> i~file
file     UserAgent_v2021
type     Executable file
[0x100001230]> icq~CDDS
0x00000000 [0x1000c51b0 - 0x1000c67c0] CDDSNetObj
0x00000000 [0x1000bdbd0 - 0x1000c4280] CDDSRcvThr
0x00000000 [0x1000c4460 - 0x1000c4b60] CDDSSndThr
0x00000000 [0x1000ce7f0 - 0x1000d95e0] CDDSFTSRecvThr
0x00000000 [0x1000dd3e0 - 0x1000e5b90] CDDSFTSSendThr
0x00000000 [0x1000c5470 - 0x1000c8660] CDDSHostStatus
0x00000000 [0x1000c54f0 - 0x1000c88c0] CDDSLoginStatus
0x00000000 [0x1000c5670 - 0x1000c9050] CDDSClientRealIP
0x00000000 [0x1000c53f0 - 0x1000c83f0] CDDSClientStatus
0x00000000 [0x1000cf660 - 0x1000dbe10] CDDSFTSRecvThrMgr
0x00000000 [0x1000de030 - 0x1000e8b10] CDDSFTSSendThrMgr
0x00000000 [0x100084d10 - 0x10009c2d0] CDDSOpWaitCheckThr
0x00000000 [0x1000d9600 - 0x1000d9a90] CDDSFTSRecvThrQueue
0x00000000 [0x1000e5be0 - 0x1000e6440] CDDSFTSSendThrQueue
0x00000000 [0x1000c52f0 - 0x1000c7a80] CDDSRequestDownload
0x00000000 [0x1000c5570 - 0x1000c8b50] CDDSChangeDDSPwdStatus
0x00000000 [0x1000c5270 - 0x1000c76e0] CDDSFileTransferStatus
0x00000000 [0x1000c55f0 - 0x1000c8de0] CDDSPassiveConnectStatus
0x00000000 [0x1000c5370 - 0x1000c7e40] CDDSRequestCancelDownload
0x00000000 [0x10007c300 - 0x10009b9a0] CDDSIfi
0x00000000 [0x1000be470 - 0x1000c1400] non-virtual thunk to CDDSRcvThr
0x00000000 [0x1000c47b0 - 0x1000c5180] non-virtual thunk to CDDSSndThr
0x00000000 [0x1000cf550 - 0x1000d1770] non-virtual thunk to CDDSFTSRecvThr
0x00000000 [0x1000ddf30 - 0x1000e06c0] non-virtual thunk to CDDSFTSSendThr
0x00000000 [0x1000daaf0 - 0x1000db300] non-virtual thunk to CDDSFTSRecvThrMgr
0x00000000 [0x1000e74a0 - 0x1000e7cb0] non-virtual thunk to CDDSFTSSendThrMgr
0x00000000 [0x10009c240 - 0x10009c360] non-virtual thunk to CDDSOpWaitCheckThr
0x00000000 [0x1000be4d0 - 0x1000be4d0] virtual thunk to CDDSRcvThr
0x00000000 [0x1000c47e0 - 0x1000c47e0] virtual thunk to CDDSSndThr
0x00000000 [0x1000cf5b0 - 0x1000cf5b0] virtual thunk to CDDSFTSRecvThr
0x00000000 [0x1000ddf90 - 0x1000ddf90] virtual thunk to CDDSFTSSendThr
0x00000000 [0x1000dab20 - 0x1000dab20] virtual thunk to CDDSFTSRecvThrMgr
0x00000000 [0x1000e74d0 - 0x1000e74d0] virtual thunk to CDDSFTSSendThrMgr
0x00000000 [0x10009c270 - 0x10009c270] virtual thunk to CDDSOpWaitCheckThr
[0x100001230]> 
```

Some of the CDDS methods found in the 2021 UserAgent executable

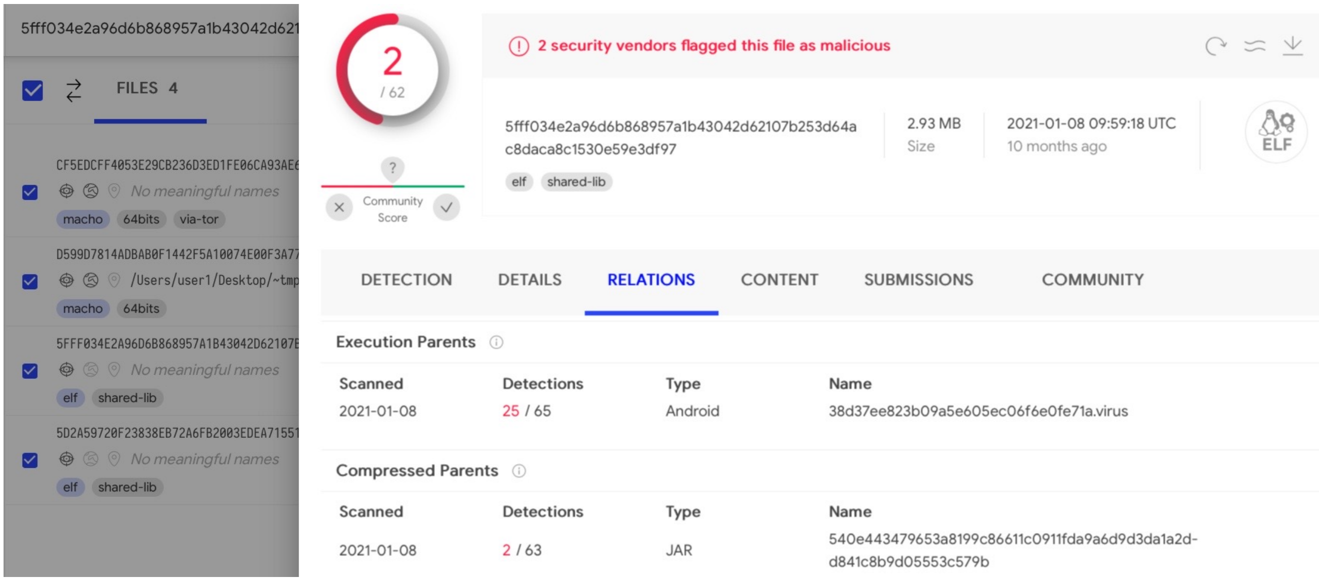
That code, according to Google TAG, is an implementation for a DDS – Data Distribution Service – framework. While our searches turned up blank trying to find a specific implementation of DDS that matched the functions used in macOS.Macma, we did find other malware that uses the same framework.



```
→  Downloads diff -y <(strings - elfbin | grep "Unserialize CDDS"| uniq) <(strings - CDDS/client | grep "Unserialize CDDS"|uniq) | grep -v \> | grep -v "|" | grep -v \<

Unserialize CDDSCmdAck ERROR                                    Unserialize CDDSCmdAck ERROR
Unserialize CDDSFileTransferStatus ERROR                        Unserialize CDDSFileTransferStatus ERROR
Unserialize CDDSRequestDownload ERROR                           Unserialize CDDSRequestDownload ERROR
Unserialize CDDSRequestCancelDownload ERROR                     Unserialize CDDSRequestCancelDownload ERROR
Unserialize CDDSClientStatus ERROR                              Unserialize CDDSClientStatus ERROR
Unserialize CDDSHostStatus ERROR                                Unserialize CDDSHostStatus ERROR
Unserialize CDDSLoginStatus ERROR                               Unserialize CDDSLoginStatus ERROR
Unserialize CDDSChangeDDSPwdStatus ERROR                        Unserialize CDDSChangeDDSPwdStatus ERROR
Unserialize CDDSPassiveConnectStatus ERROR                      Unserialize CDDSPassiveConnectStatus ERROR
→  Downloads diff -y  <(strings - elfbin | grep -i octstr) <(strings - CDDS/client | grep -i octstr)
_Z10Octstr2DecPKc                                              | c2Octstrt
_Z10Dec2Octstrt                                                | __Z10Dec2Octstrt
_Z10Octstr2DecPKc                                              | __Z10Octstr2DecPKc
_Z10Dec2Octstrt                                                | Octstr2DecPKc
                                                               > c2Octstrt
                                                               > __Z10Dec2Octstrt
                                                               > __Z10Octstr2DecPKc
→  Downloads
```

Android malware drops an ELF bin that contains the same CDDS framework



5fff034e2a96d6b868957a1b43042d621

☑ ⇄ FILES 4

☑ CF5EDCFF4053E29CB236D3ED1FE06CA93AE6
No meaningful names
macho 64bits via-tor

☑ D599D7814ADBAB0F1442F5A10074E00F3A77
/Users/user1/Desktop/~tmp
macho 64bits

☑ 5FFF034E2A96D6B868957A1B43042D62107B
No meaningful names
elf shared-lib

☑ 5D2A59720F23838EB72A6FB2003EDEA71551
No meaningful names
elf shared-lib

2 / 62

? Community Score

2 security vendors flagged this file as malicious

5fff034e2a96d6b868957a1b43042d62107b253d64a c8daca8c1530e59e3df97

elf shared-lib

2.93 MB
Size

2021-01-08 09:59:18 UTC
10 months ago

ELF

DETECTION DETAILS RELATIONS CONTENT SUBMISSIONS COMMUNITY

Execution Parents

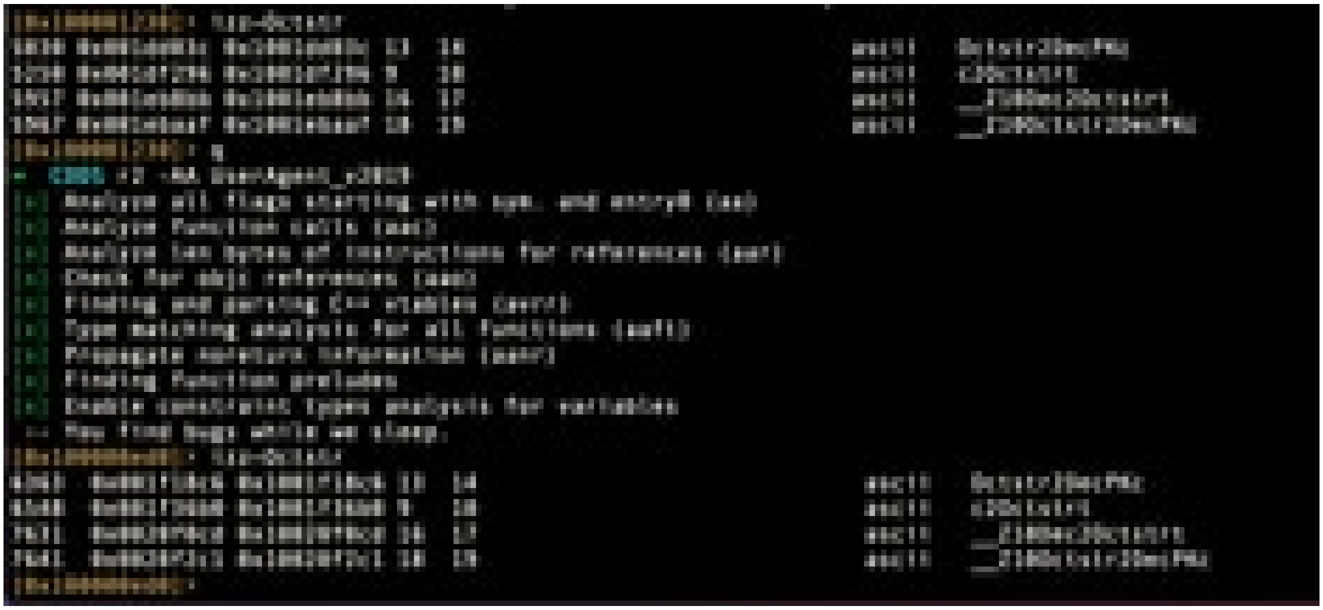
Scanned	Detections	Type	Name
2021-01-08	25 / 65	Android	38d37ee823b09a5e605ec06f6e0fe71a.virus

Compressed Parents

Scanned	Detections	Type	Name
2021-01-08	2 / 63	JAR	540e443479653a8199c86611c0911fda9a6d9d3da1a2d-d841c8b9d05553c579b

Links to known Android malware droppers

These ELF bins and both versions of macOS.Macma's UserAgent also share another commonality, the strings "Octstr2Dec" and "Dec2Octstr".



```
[illegible]
[illegible]
[illegible]
[illegible]
[illegible]
[illegible]
[illegible] UserAgent_[illegible]
[x] Analyze all flags starting with sym. and entry0 (aa)
[x] Analyze function calls (aac)
[x] Analyze len bytes of instructions for references (aar)
[x] Check for objc references (aao)
[x] Finding and parsing C++ vtables (avrr)
[x] Type matching analysis for all functions (aaft)
[x] Propagate noreturn information (aanr)
[x] Finding function preludes
[x] Enable constraint types analysis for variables
-- You find bugs while we sleep.
[illegible]
[illegible]
[illegible]
[illegible]
[illegible]
[illegible]
```

Commonalities between macOS.Macma and a malicious ELF Shared object file

These latter strings, which appear to be conversions for strings containing octals and decimals, may simply be a matter of coincidence or of code reuse. The code similarities we found also have links back to installers for the notorious Shedun Android malware.

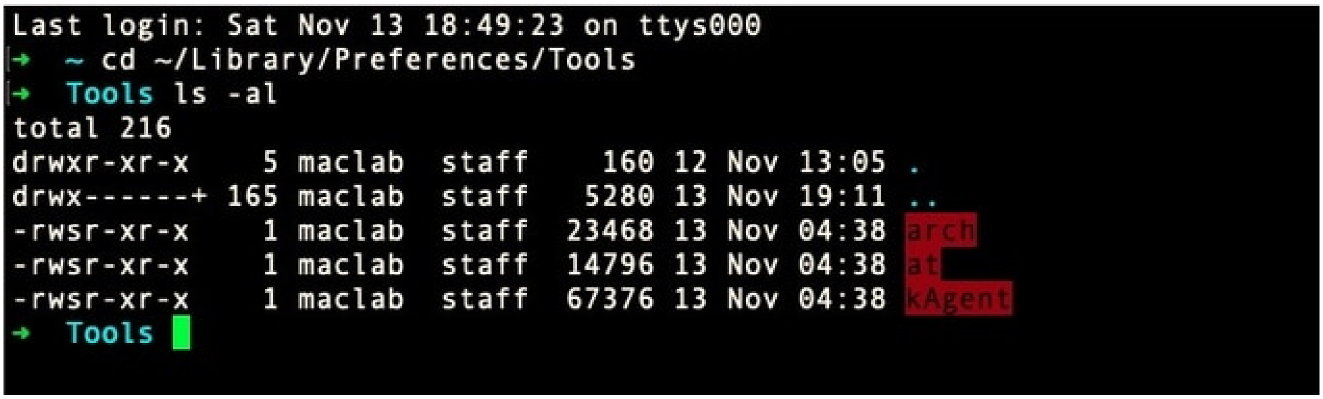
In their report, Google's TAG pointed out that macOS.Macma was associated with an iOS exploit chain that they had not been able to entirely recover. Our analysis suggests that the actors behind macOS.Macma at least were reusing code from ELF/Android developers and possibly could have also been targeting Android phones with malware as well. Further analysis is needed to see how far these connections extend.

Macma's Keylogger and AV Capture Functionality

While the earlier reports referred to above have already covered the basics of macOS.Macma functionality, we want to expand on previous reporting to reveal further IoCs.

As previously mentioned, macOS.Macma will drop a persistence agent at `~/Library/LaunchAgents/com.UserAgent.va.plist` and an executable at `~/Library/Preferences/lib/UserAgent`.

As we noted above, the LaunchAgent will ensure that before the job starts, the executable's current working directory will be changed to the aforementioned "lib" folder. This folder is used as a repository for data culled by the keylogger, "kAgent", which itself is dropped at `~/Library/Preferences/Tools/`, along with the "at" and "arch" Mach-O binaries.



```
Last login: Sat Nov 13 18:49:23 on ttys000
➜  ~ cd ~/Library/Preferences/Tools
➜  Tools ls -al
total 216
drwxr-xr-x    5 maclab  staff    160 12 Nov 13:05 .
drwx------+ 165 maclab  staff   5280 13 Nov 19:11 ..
-rwsr-xr-x    1 maclab  staff  23468 13 Nov 04:38 arch
-rwsr-xr-x    1 maclab  staff  14796 13 Nov 04:38 at
-rwsr-xr-x    1 maclab  staff  67376 13 Nov 04:38 kAgent
➜  Tools 
```

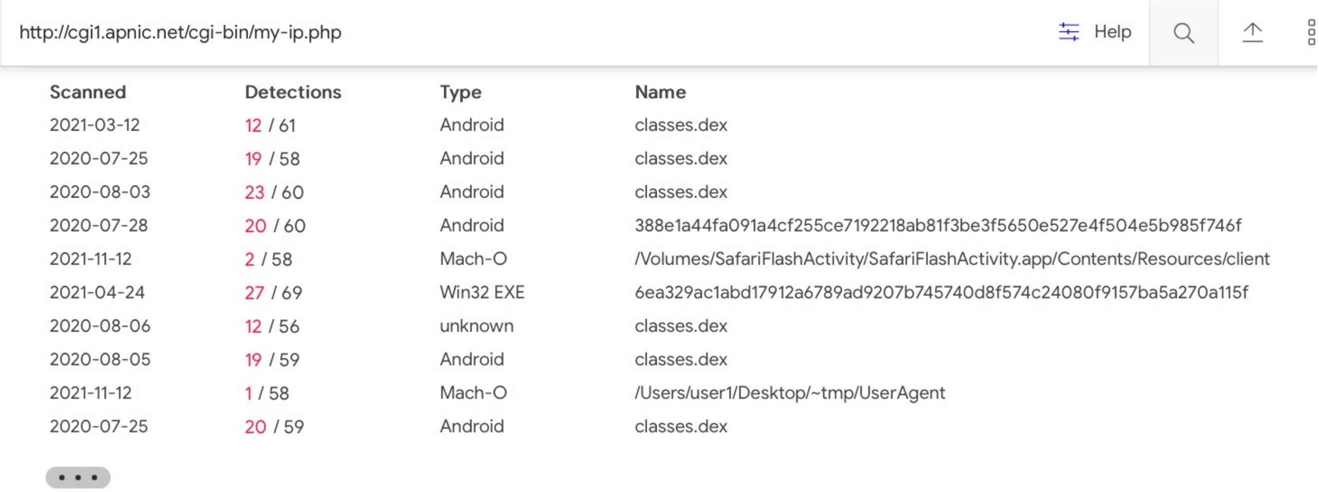
Binaries dropped by macOS.Macma

The kAgent keylogger creates text files of captured keystrokes from any text input field, including Spotlight, Finder, Safari, Mail, Messages and other apps that have text fields for passwords and so on. The text files are created with Unix timestamps for names and collected in directories called "data".

The file 1636804188 contains data captured by the keylogger

We also note that this malware reaches out to a remote .php file to return the user's IP address. The same URL has a long history of use.

```
http://cgi1.apnic.net/cgi-bin/my-ip.php
```



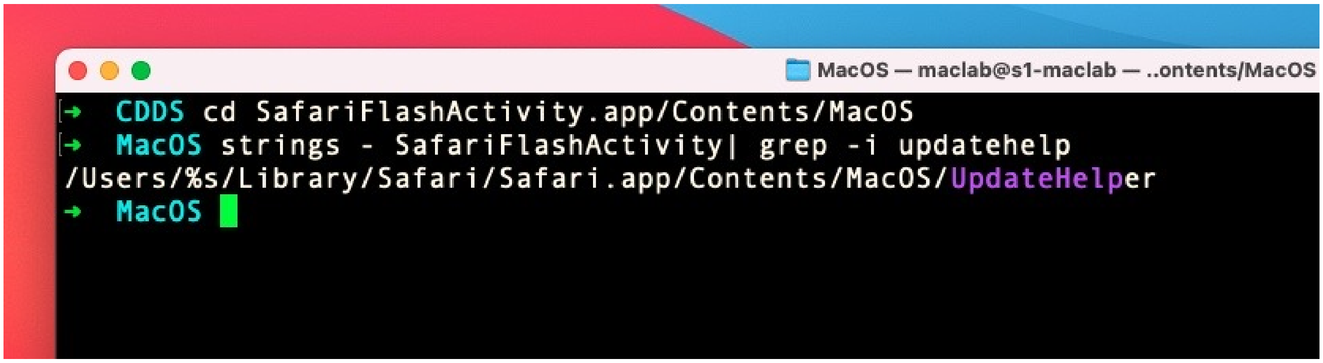
http://cgi1.apnic.net/cgi-bin/my-ip.php

Scanned	Detections	Type	Name
2021-03-12	12 / 61	Android	classes.dex
2020-07-25	19 / 58	Android	classes.dex
2020-08-03	23 / 60	Android	classes.dex
2020-07-28	20 / 60	Android	388e1a44fa091a4cf255ce7192218ab81f3be3f5650e527e4f504e5b985f746f
2021-11-12	2 / 58	Mach-O	/Volumes/SafariFlashActivity/SafariFlashActivity.app/Contents/Resources/client
2021-04-24	27 / 69	Win32 EXE	6ea329ac1abd17912a6789ad9207b745740d8f574c24080f9157ba5a270a115f
2020-08-06	12 / 56	unknown	classes.dex
2020-08-05	19 / 59	Android	classes.dex
2021-11-12	1 / 58	Mach-O	/Users/user1/Desktop/~tmp/UserAgent
2020-07-25	20 / 59	Android	classes.dex

Both Android and macOS malware ping this URL

Finally, one further IoC we noted in the `../MacOS/SafariFlashActivity` "lifei" binary that never appeared anywhere else, and we also did not see dropped on any of our test runs, was:

```
/Users/%s/Library/Safari/Safari.app/Contents/MacOS/UpdateHelper
```



```
→  CDDS cd SafariFlashActivity.app/Contents/MacOS
→  MacOS strings - SafariFlashActivity| grep -i updatehelp
/Users/%s/Library/Safari/Safari.app/Contents/MacOS/UpdateHelper
→  MacOS
```

Malware tries to drop a file in the Safari folder

This is worth mentioning since the target folder, the User's Library/Safari folder, is TCC protected since Mojave. For that reason, any attempt to install there would fall afoul of current TCC protections (bypasses notwithstanding). It looks, therefore, like a remnant of the earlier code development from El Capitan era, and indeed we do not see this string in later versions. However, it's unique enough for defenders to watch out for: there's never any legitimate reason for an executable at this path to exist on any version of macOS.

Conclusion

Catching APTs targeting macOS users is a rare event, and we are lucky in this instance to have a fairly transparent view of the malware being dropped. Regardless of the vector used to drop the malware, the payload itself is perfectly functional and capable of exfiltrating data and spying on macOS users. It's just another reminder, if one were needed, that simply investing in a Mac does not guarantee you safe passage against bad actors. This may have been an APT-developed payload, but the code is simple enough for anyone interested in malfeasance to reproduce.

Indicators of Compromise

SHA1

000830573ff24345d88ef7916f9745aff5ee813d; *UserAgent 2021 payload, Mach-O*
07f8549d2a8cc76023acee374c18bbe31bb19d91; *UserAgent 2019, Mach-0*
0e7b90ec564cb3b6ea080be2829b1a593fff009f; *(Related) ELF DYN Shared object file*
2303a9c0092f9b0ccac8536419ee48626a253f94; *UserAgent 2021 installer, Mach-0*
31f0642fe76b2bdf694710a0741e9a153e04b485; *SafariFlashActivity1, Mach-0*
734070ae052939c946d096a13bc4a78d0265a3a2; *(Related) ELF DYN Shared object file*
77a86a6b26a6d0f15f0cb40df62c88249ba80773; *at, Mach-0*
941e8f52f49aa387a315a0238cff8e043e2a7222; *install_flash_player_osx.dmg, DMG*
b2f0dae9f5b4f9d62b73d24f1f52dcb6d66d2f52; *client, Mach-0*
b6a11933b95ad1f8c2ad97afedd49a188e0587d2; *SafariFlashActivity, Mach-0*
c4511ad16564eabb2c179d2e36f3f1e59a3f1346; *arch, Mach-0*
f7549ff73f9ce9f83f8181255de7c3f24ffb2237; *SafariFlashActivityInstall, shell script*

File Paths

~/Library/Preferences/Tools/at
~/Library/Preferences/Tools/arch
~/Library/Preferences/Tools/kAgent
~/Library/LaunchAgents/com.UserAgent.va.plist
~/Library/Preferences/UserAgent/lib/Data/
~/Library/Preferences/UserAgent/lib/UserAgent
~/Library/Safari/Safari.app/Contents/MacOS/UpdateHelper

Identifiers

xxxxx.SafariFlashActivity
xxxxxx.preexcl.project