PYTHON VS MODERN DEFENSES

13th of August 2022

DEFCON30 – Adversary Village

ADVERSARY VILLAGE @ D3FC0N

whoami

Diego Capriotti @naksyn

- Offensive Cyber Security Team Lead @ Axians Italy
- ITA Army veteran Engineer Officer
- Past roles: Electronic Warfare, Cyber Security
- Side interests: playing chess, tinkering with Software Defined Radios

Python vs Modern Defenses

- Modern Defenses – Basic Concepts
- A Bypass Strategy
- Leveraging Python

1

MODERN DEFENSES

Basic concepts

MODERN DEFENSES – Basic concepts

- EDR Visibility
- Memory Scanning
- ML-based detection
- IoC and IoA-based detection

EDR VISIBILITY

To Detect and Respond you must first see what’s happening on a system. EDRs get data from optics available on target OS and also employ proprietary techniques.

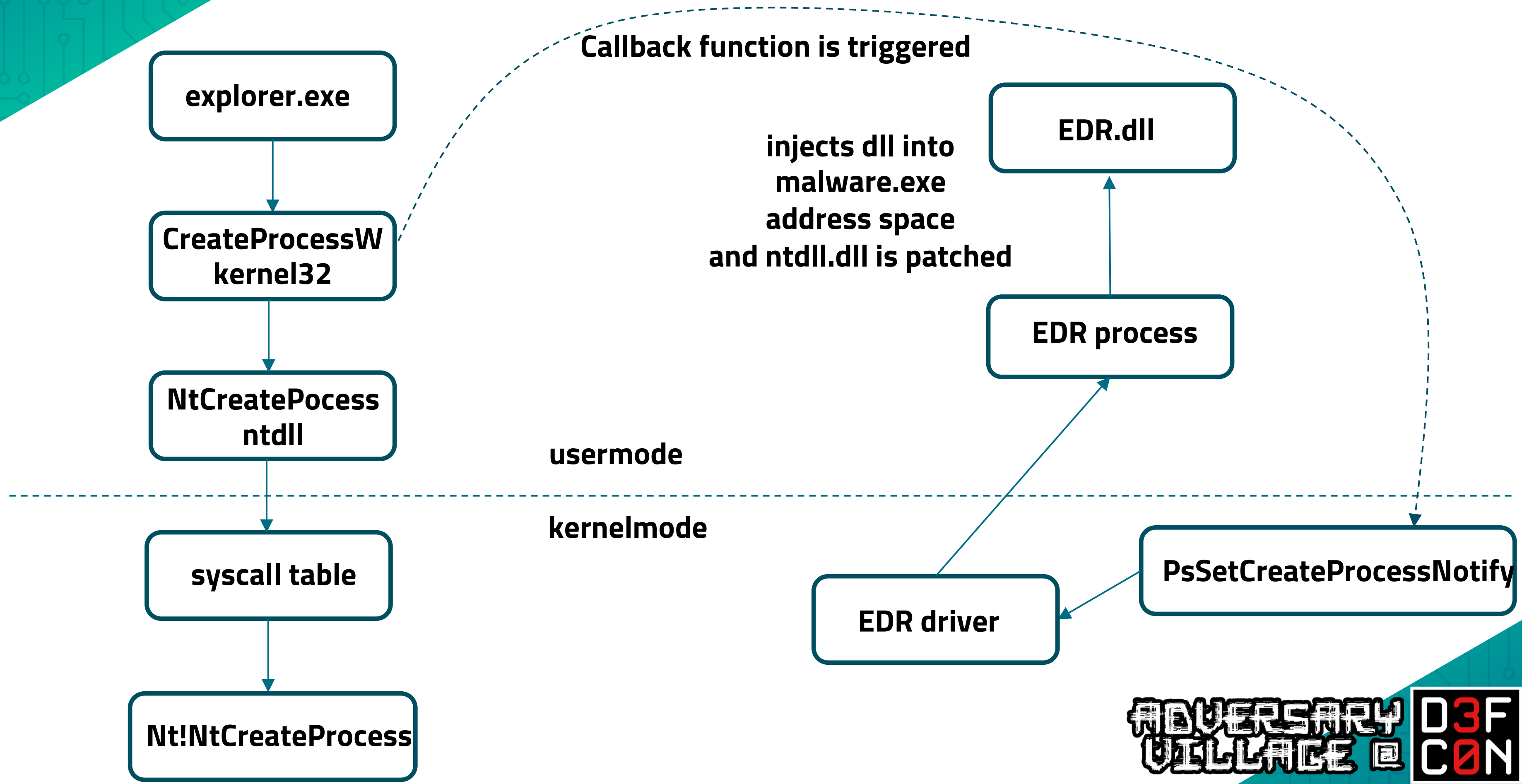
EDR VISIBILITY

Two common ways of increasing visibility on Windows

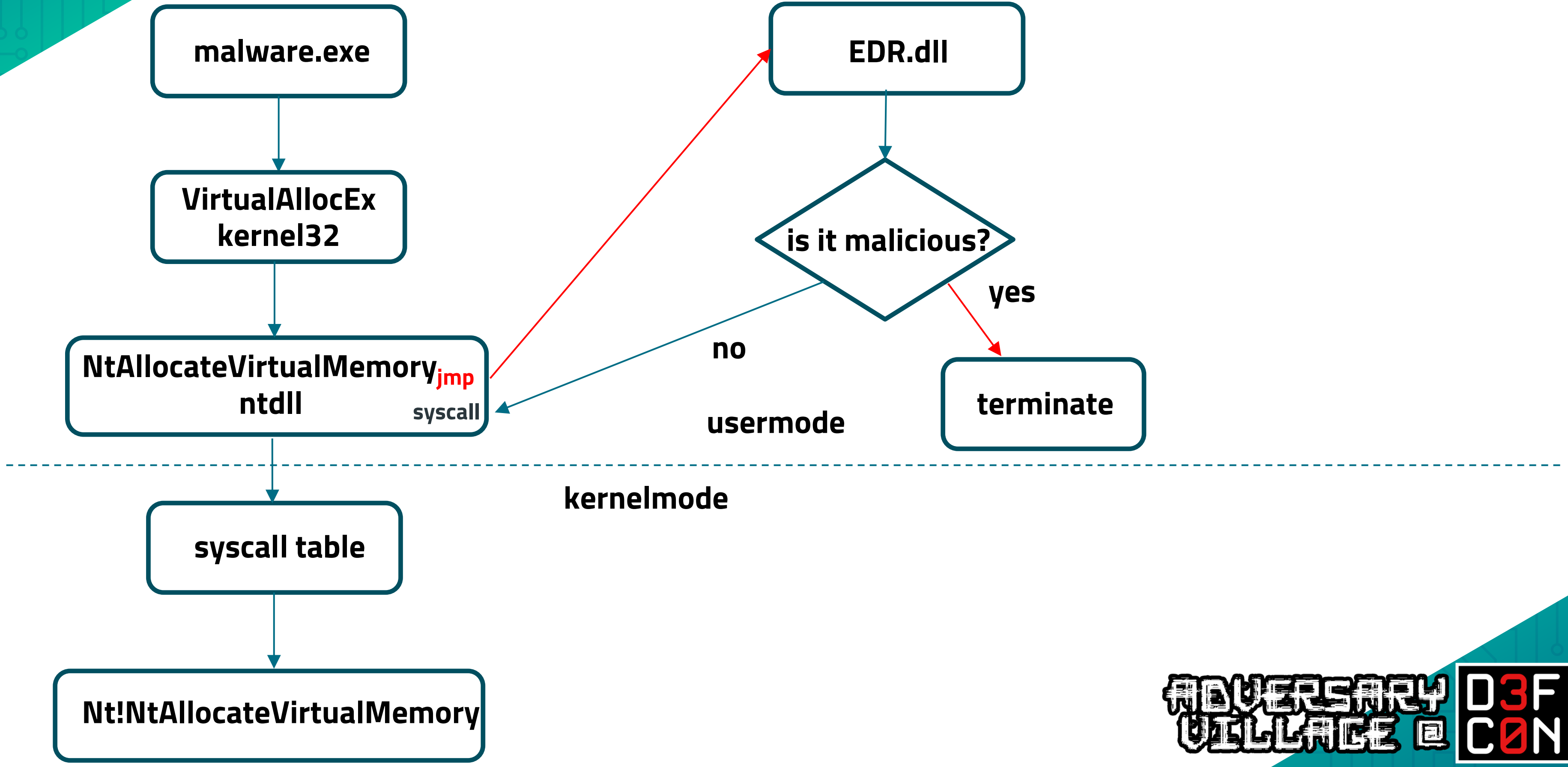
- Using Kernel Callbacks to:
  - Inject EDR's dll into new processes
  - Getting process tree information
  - Getting image loading events

- Using Usermode Hooks to:
  - Inspect Windows API calls

EDR VISIBILITY – Kernel callbacks



EDR VISIBILITY – Usermode Hooks



MEMORY SCANNING

In-memory scanning techniques look for patterns in the code and data of processes. Scanning is resource intensive and could be periodic or triggered by events/conditions/analysts.

MEMORY SCANNING

- Inner workings of AV/EDR are undisclosed
- Examples of triggered scans:
  - Unusual process-tree
  - Suspicious binary (ML detection)
  - Unusual sequence of API-calls
  - Unusual access to files/process handles
  - Suspicious traffic (amount, type, reputation)

https://www.artstation.com/artwork/vBJx6

MEMORY SCANNING

- Common malicious indicators detected by memory scanning:
  - Known-bad signature-based IoC
  - Reflectively loaded Dlls
  - Injected threads
  - RWX permissions
  - Inline/IAT/EAT hooking
  - modules with modified/unmatching PE header
  - implanted PE files (manually loaded, not corresponding to any legitimate module)
- Great tools: @hasherazade's PeSieve @forestorr's Moneta

ML-based Detection

Machine Learning can detect variant malware files that can evade signature-based detection. Malware possesses several “features” that can be used for training machine learning models.

Moon et al. - Compact feature hashing for machine learning based malware detection

ML-based Detection

Essentially, the workflow we follow to build any machine learning-based detector, including a decision tree, boils down to these steps:

- Collect example of malware and goodware [...]
- Extract features from each training example to represent the example as an array of numbers[...]
- Train the ML system to recognize malware using the features we extracted
- Test the approach [...]

Malware Data Science: Attack Detection and Attribution (2018) - Joshua Saxe

ML-based Detection

Some features used to determine wether a file is good or bad:

- Wether it's digitally signed
- The presence of malformed headers
- The presence of encrypted data
- Wether it has been seen on more than 100 network workstations

Lots of different features are used in ML detection

Malware Data Science: Attack Detection and Attribution (2018) - Joshua Saxe

“What if a binary commonly classified as benignware is used maliciously?

What if it’s also widely used for legit custom applications?

IoC and IoA-based detection

An Indicator of Compromise (IoC) is digital evidence that a cyber incident has occurred. An Indicator of Attack (IOA) is digital or physical evidence that a cyberattack is likely to occur.

IoC is static, IoA is dynamic.

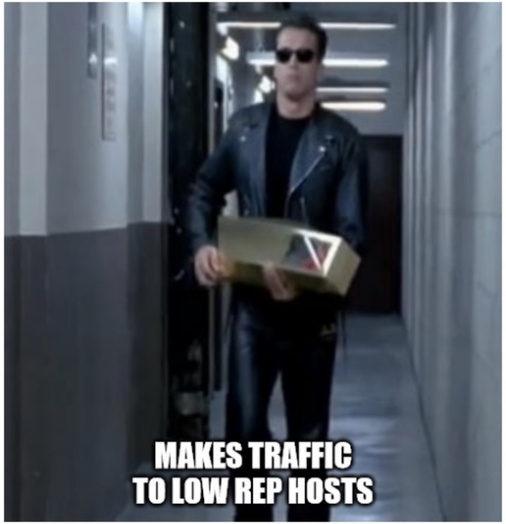
IoC and IoA-based detection

IoC is retroactive

IoC provides forensic intelligence but can't help detect an attack attempt. Signatures also generates instances of false positives.

IoC and IoA-based detection

IoA is proactive



IoA can detect a threat not characterized by static signatures.

Does not provide sufficient forensic intelligence.

“What if an attacker is directly launching a widely used signed binary (AKA LOLbin) and operations are done natively from that process?

ADVERSARY VILLAGE @ D3F C0N

2

A BYPASS STRATEGY

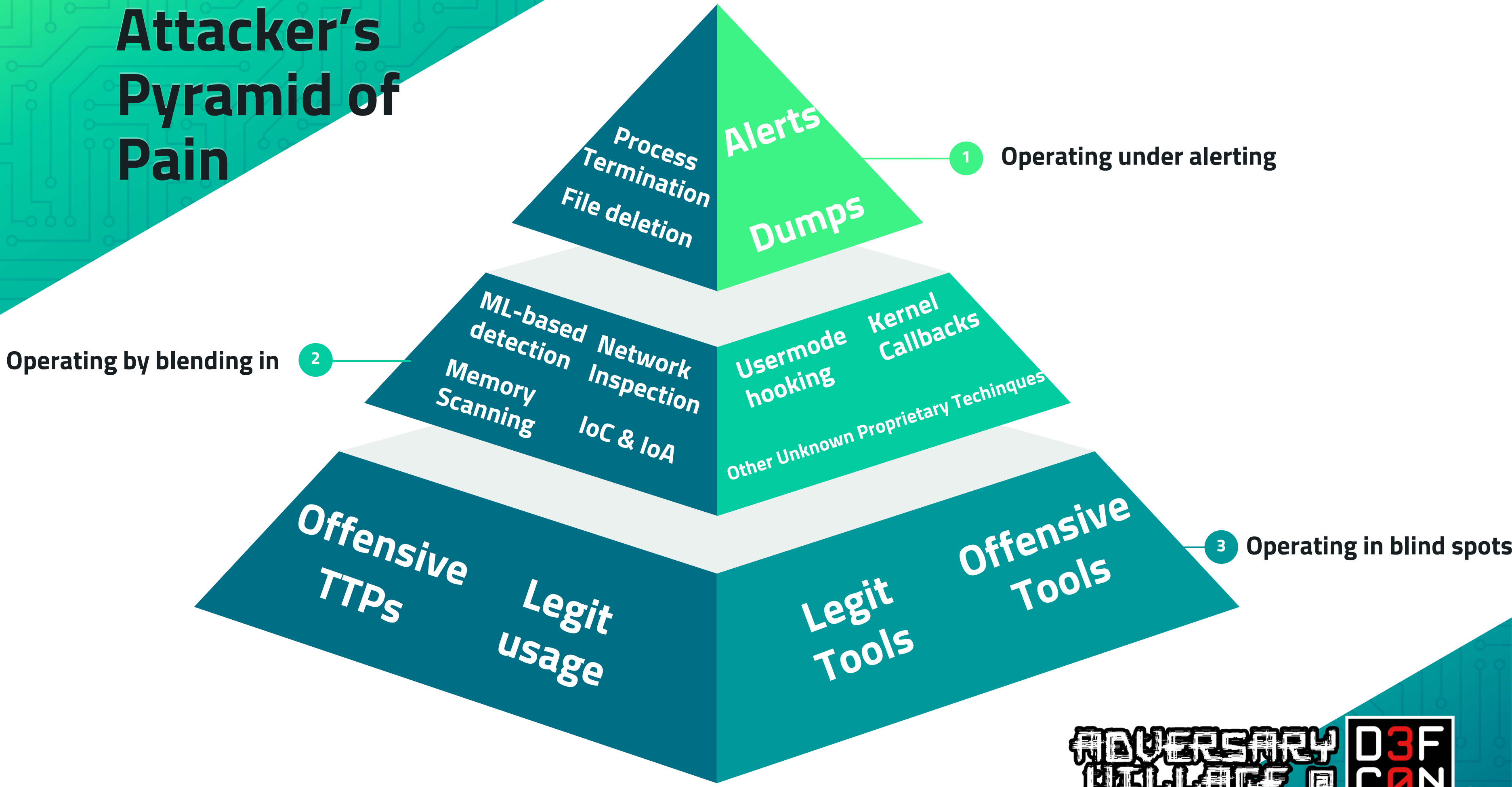
A BYPASS STRATEGY

- Main Categories of EDR Evasion
- Constraints
- Strategy

Main Categories of EDR evasion

- Avoiding the EDR
  - proxying traffic or pure avoidance
- Blending into the environment
- EDR tampering
- Operating in blind spots
  - Exploiting lack of visibility

Attacker's Pyramid of Pain



ADVERSARY VILLAGE @ D3FCON

Constraints

- Operational Scenario/limitations:
  - Operations done from an EDR-equipped box
  - No remote process Injections
  - No dropping on disk custom/unknown artifacts
  - C2 agent execution is a last resort
- Desired capabilities:
  - Dynamic module loading
  - Compatibility with community-driven tools
  - Traffic tunneling without spawning new processes

Strategy

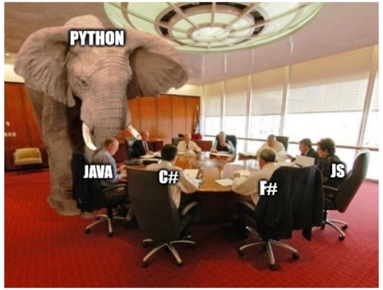
Choosing the language

- Operating in EDR’s blind spots
  - Choose a set of common non-native languages
  - Exclude languages that can natively provide optics to EDRs.
  - embeddable packages are desirable
  - how much existing tooling can be reused?
  - Check if capabilities can be developed

ADVERSARY VILLAGE @ D3F C0N

Strategy

Choosing the language



- Python language has several benefits:
  - Python >3.7 comes with an «Embeddable zip package»
  - Signed interpreter
  - Limited visibility of python code execution for EDRs
  - Lots of offensive tooling available
  - The interpreter natively runs API calls AKA “lot of different telemetry coming from the same binary”

ADVERSARY VILLAGE @ D3FC0N

3

LEVERAGING PYTHON

LEVERAGING PYTHON

https://github.com/naksyn/Pyramid

- Execution Method
- Dynamically Importing Python Modules
  - Bloodhound-python and impacket
- Using BOFs with Python
  - Dumping lsass with nanodump
- In-process tunneling
  - Listen, I really need to run an agent!

Execution Method

- Dropping “Python Embeddable Package” and running python.exe (or pythonw.exe) directly.
  - less probability of triggering IoAs and IoCs – no uncommon process tree patterns.
  - less probability of triggering ML detection - signed files
  - No visibility for dynamic code execution for stock python.exe - ref. PEP-578 – Python Runtime Audit Hooks

Execution Method

Lack of visibility

- PEP-578 Runtime Audit Hooks – introduced to “solve” the limited context for Python code

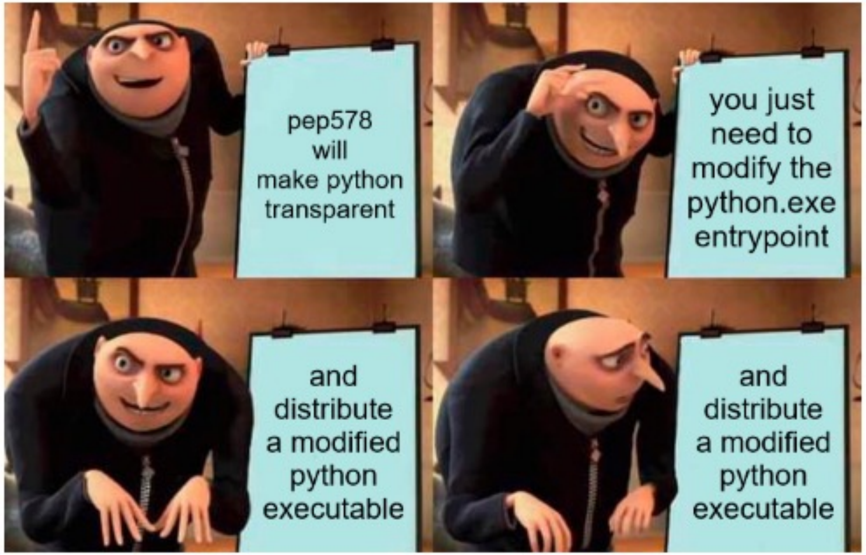
API Function	Event Name	Arguments	Rationale
`compile`, `exec`, `eval`, `PyAst_CompileString`, `PyAST_obj2mod`	`compile`	`(code, filename_or_none)`	Detect dynamic code compilation, where `code` could be a string or AST. Note that this will be called for regular imports of source code, including those that were opened with `open_code`.
`exec`, `eval`, `run_mod`	`exec`	`(code_object,)`	Detect dynamic execution of code objects. This only occurs for explicit calls, and is not raised for normal function invocation.
`import`	`import`	`(module, filename, sys.path, sys.meta_path, sys.path_hooks)`	Detect when modules are imported. This is raised before the module name is resolved to a file. All arguments other than the module name may be `None` if they are not used or available.

ADVERSARY VILLAGE @ D3F C0N

Execution Method

Why no visibility?

- PEP-578 audit hooks are not enabled in stock python.exe
- deploying PEP-578 is complex



Dynamically Importing Python Modules

- Been around for quite some time
- Amazing prior work done by **@scythe_io** (in-memory Embedding of CPython), **@xorrior** (Empyre), **@n1nj4sec** (pupy), **@ajpc500** (Medusa)
  - Each project has its own goals and design choices

Dynamically Importing Python Modules

- PEP 302 – New Import Hooks
  - import hooks allow you to modify the logic in which Python modules are located and how they are loaded.
  - involves defining a custom "Finder" class and either adding finder objects to sys.meta_path
  - sys.meta_path holds entries that implement Python's default import semantics

```
sound/                          Top-level package
      __init__.py               Initialize the sound package
      formats/                  Subpackage for file format conversions
              __init__.py
              wavread.py
              wavwrite.py
              aiffread.py
              aiffwrite.py
              auread.py
              auwrite.py
              ...
      effects/                  Subpackage for sound effects
              __init__.py
              echo.py
              surround.py
              reverse.py
```

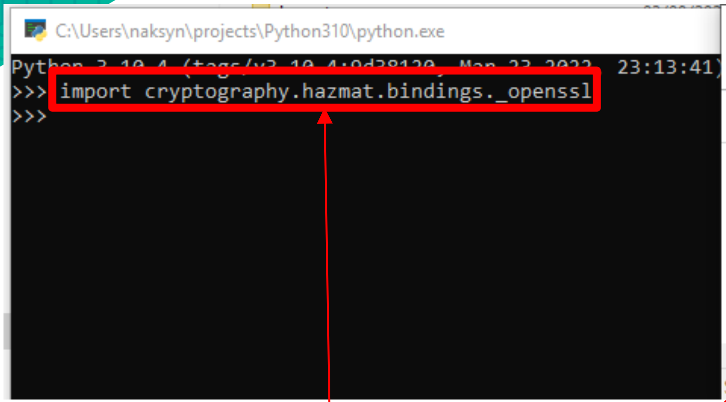
Dynamically Importing Python Modules

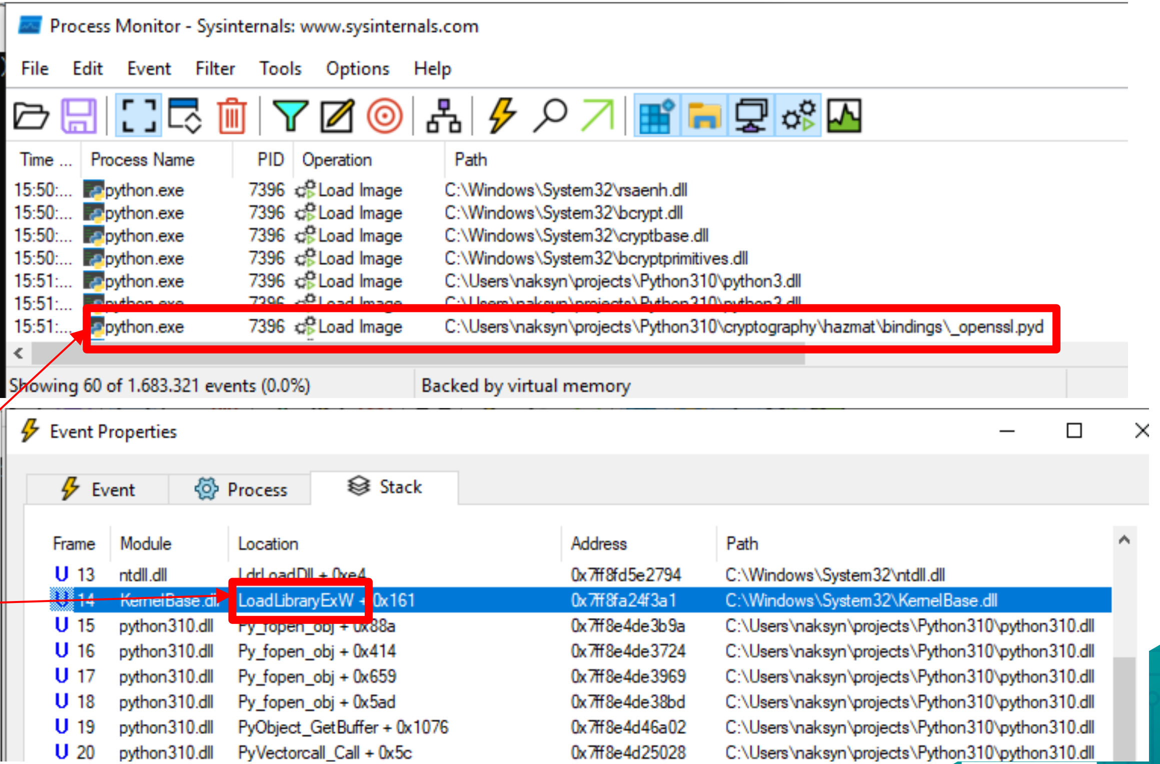
- Using Import Hooks we can:
  - Use a custom Finder class
  - In-memory download a Python package as a zip
  - Add the zip file finder object to sys.meta_path
  - Import the zip file in memory
- Problems:
  - Python module dependencies nightmare
  - In-memory Dynamic loading of *.pyd extensions is not natively supported

Dynamically Importing Python Modules

- Dynamic Loading used in Pyramid:
  - Based on @xorrior Empyre Finder class
  - uses fixed packages dependencies to in-memory import impacket, bloodhound-python and paramiko

Dynamically Importing Python Modules





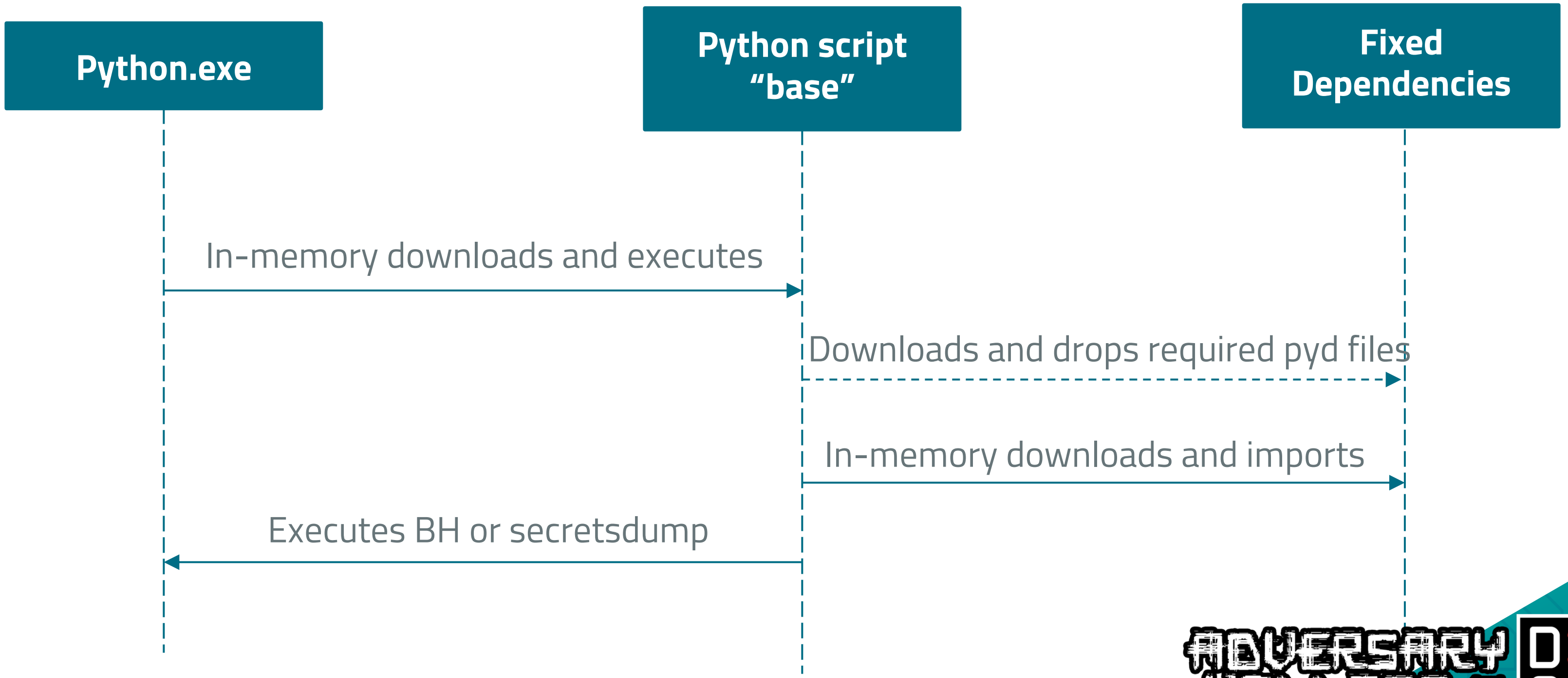
Normal behavior for Importing a pyd file on disk

Dynamically Importing Python Modules

- How the problems were solved in Pyramid:
  - Python module dependencies nightmare : solved by providing fixed dependencies packages
  - *.pyd In-memory loading would require re-engineering the CPython interpreter losing its digital signature. An acceptable solution – per our scenario – would be dropping on-disk the official Pypy Wheels containing the needed pyd files and maintain normal loading behaviour.

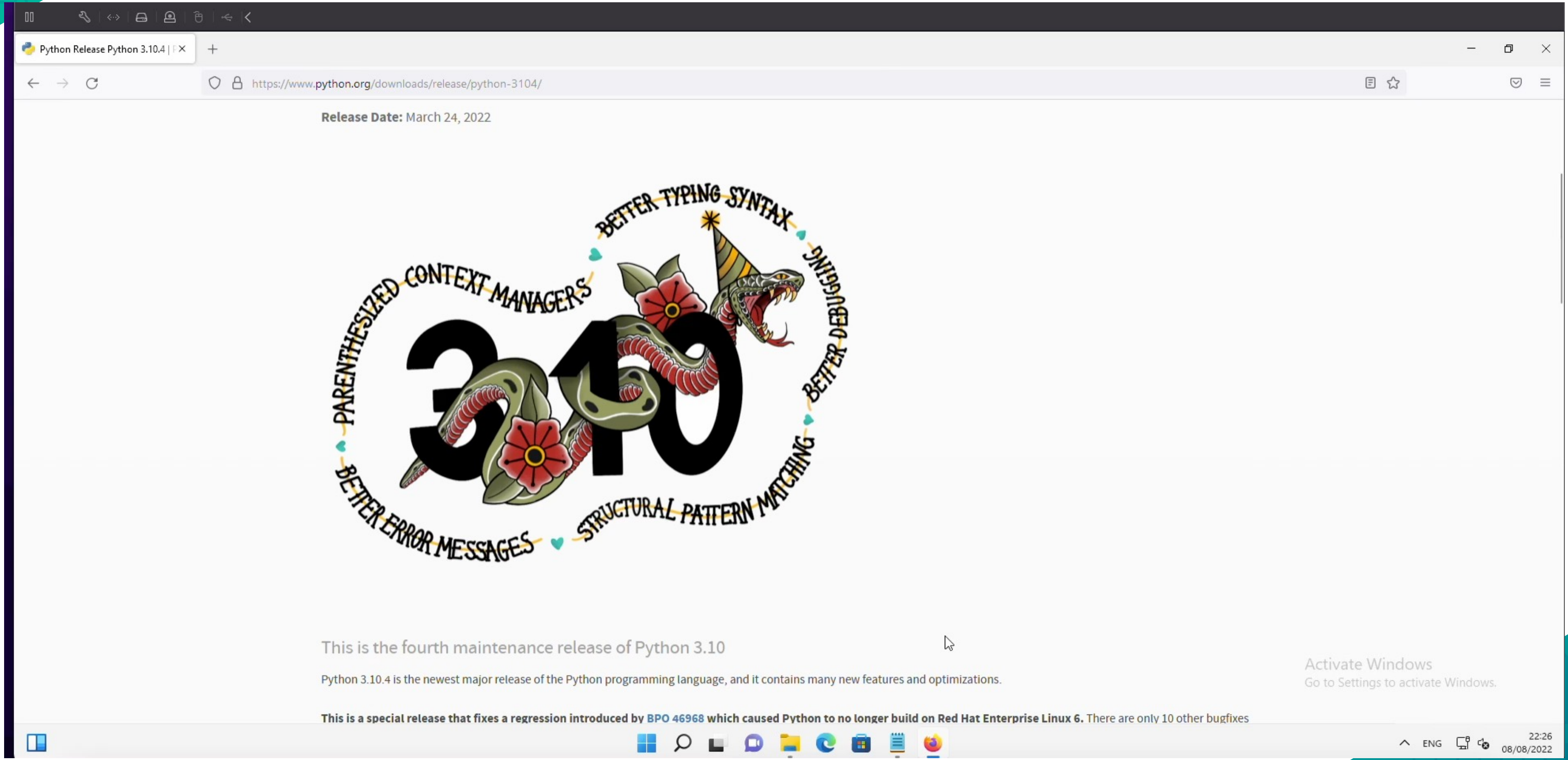
Demo

Dynamically importing and executing BloodHound-Python



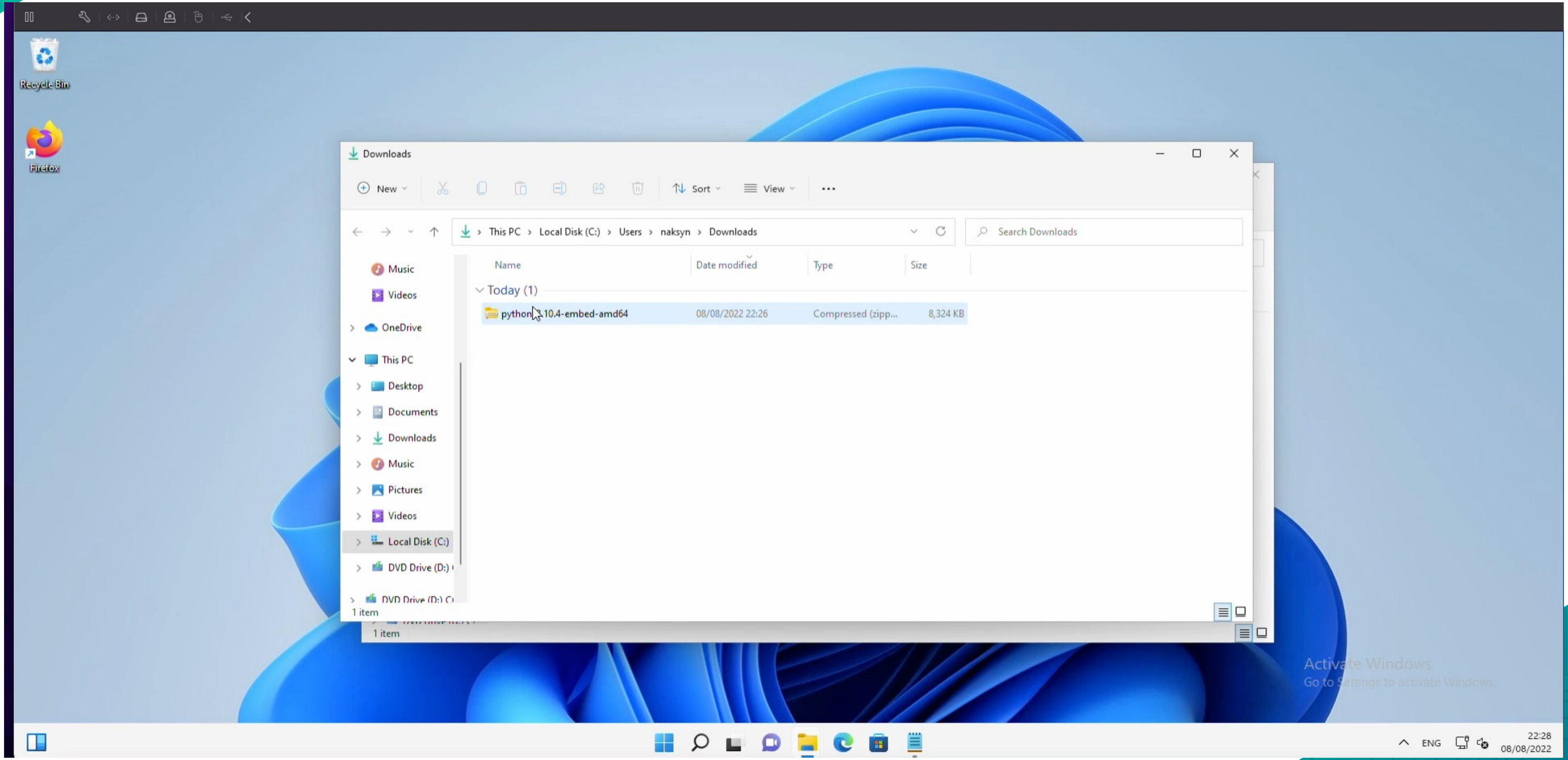
Demo

Dynamically importing and executing BloodHound-Python



Demo

Dynamically importing and executing Impacket secretsdump



Using Beacon Object Files with Python

- BOFs are a way to rapidly extend the Cobalt Strike's Beacon agent with new post-exploitation features by executing a compiled C program withing the Beacon process.
- Lot of amazing community-driven BOFs are available
- Achieving a way to execute BOFs with one's own technique or C2 is a great way to augment capabilities.

Using Beacon Object Files with Python

- Feature implemented in Pyramid tool:
  - leverages **@trustedsec** COFFloader and **@falconforce** bof2shellcode
  - Complex BOFs such as nanodump should be modified to be compatible
  - COFF loader is converted to shellcode and BOF is appended
  - resulting shellcode can be dynamically injected with Python into python.exe achieving in-process BOF execution.

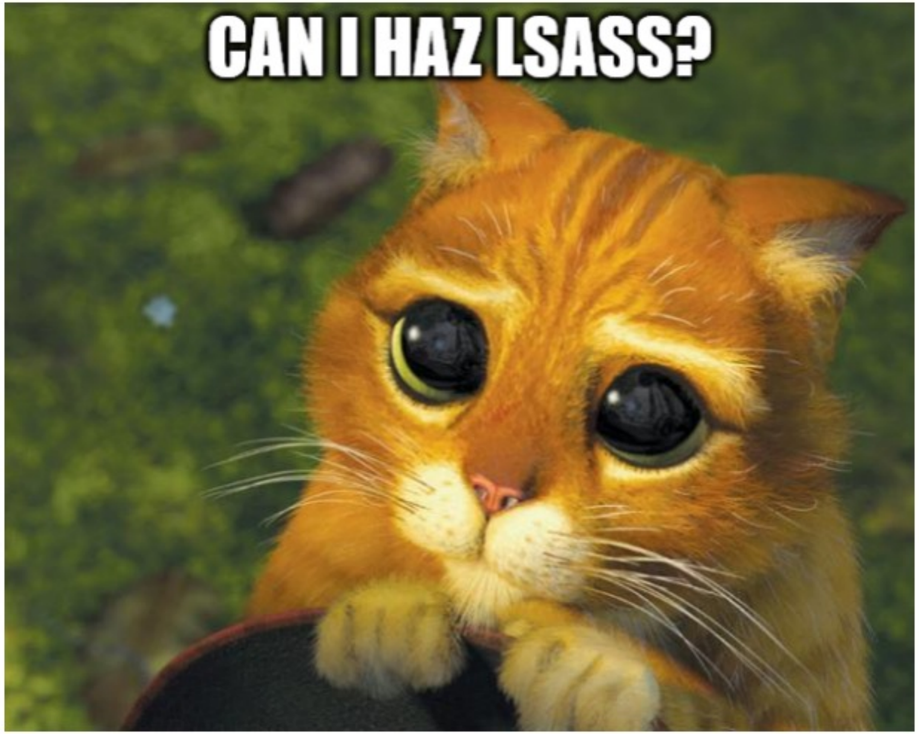
Using Beacon Object Files with Python

- COFFloader (shellcode) looks for the BOF in memory via the magic header and 4-byte integer length.
- BOF is then fed to COFFLoader
- command line arguments are parsed (can be unstable)

Modified COFFLoader (shellcode)	Magic Header + BOF length	BOF

bof2shellcode output

Using Beacon Object Files with Python

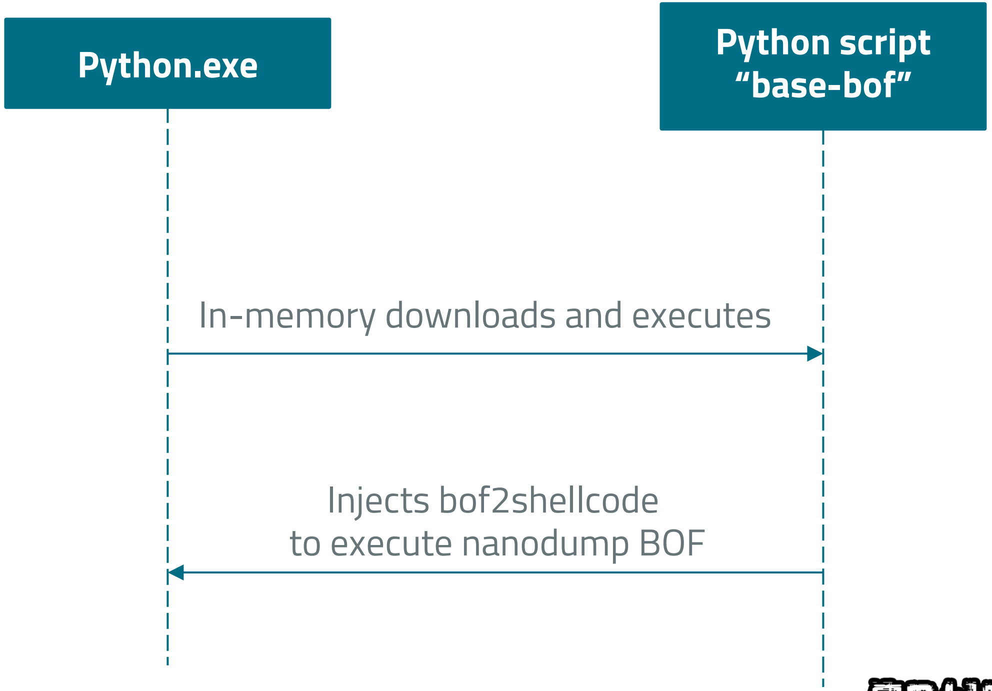


Using Beacon Object Files with Python

- We can dump lsass using @helpsystems nanodump but we need to modify the BOF by:
  - Stripping internal Beacon API calls
  - Hardcoding command line parameters to increase stability
  - Stripping cmd line parsing functions
- Compile the BOF, then use bof2shellcode
- Inject the shellcode blob into python.exe natively using Python

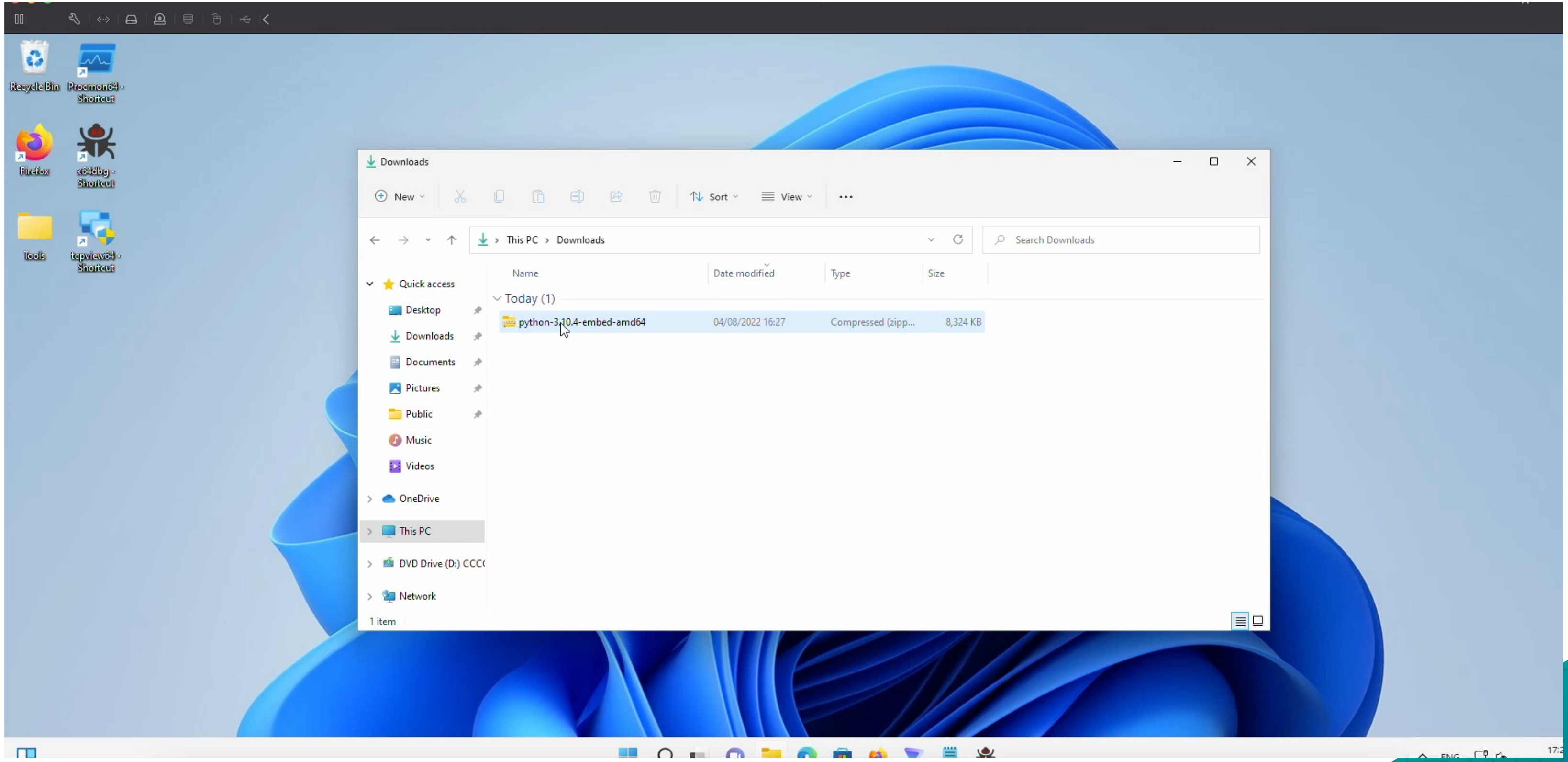
Demo

Injecting shellcode within python.exe to achieve BOF execution and dumping lsass Using process forking



Demo

Injecting shellcode within python.exe to achieve BOF execution and dumping lsass Using process forking

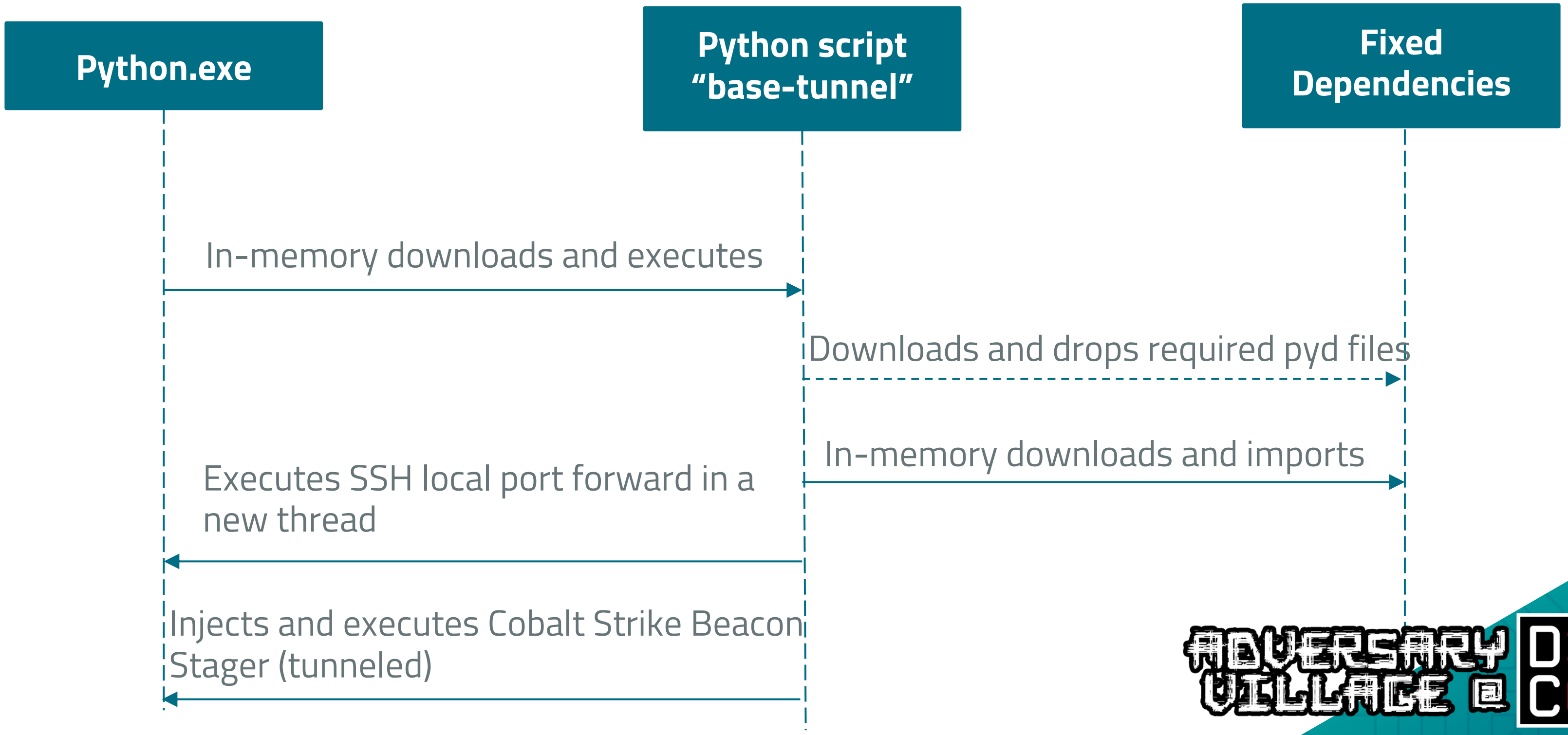


In-process agent tunneling

- Agent tunneling can be useful to:
  - Decouple agent communications with real C2
  - Blend-in with SSH instead of HTTP/S or DNS
  - Exploit the signed python.exe context to mask C2
  - Create reusable agent payloads with 127.0.0.1 as C2 host
  - Make C2 server not easily reachable from the internet
- Mind your OPSEC
  - SSH credentials are stored in python.exe memory – use at least burnable-temporary creds and whitelist IPs on SSH server

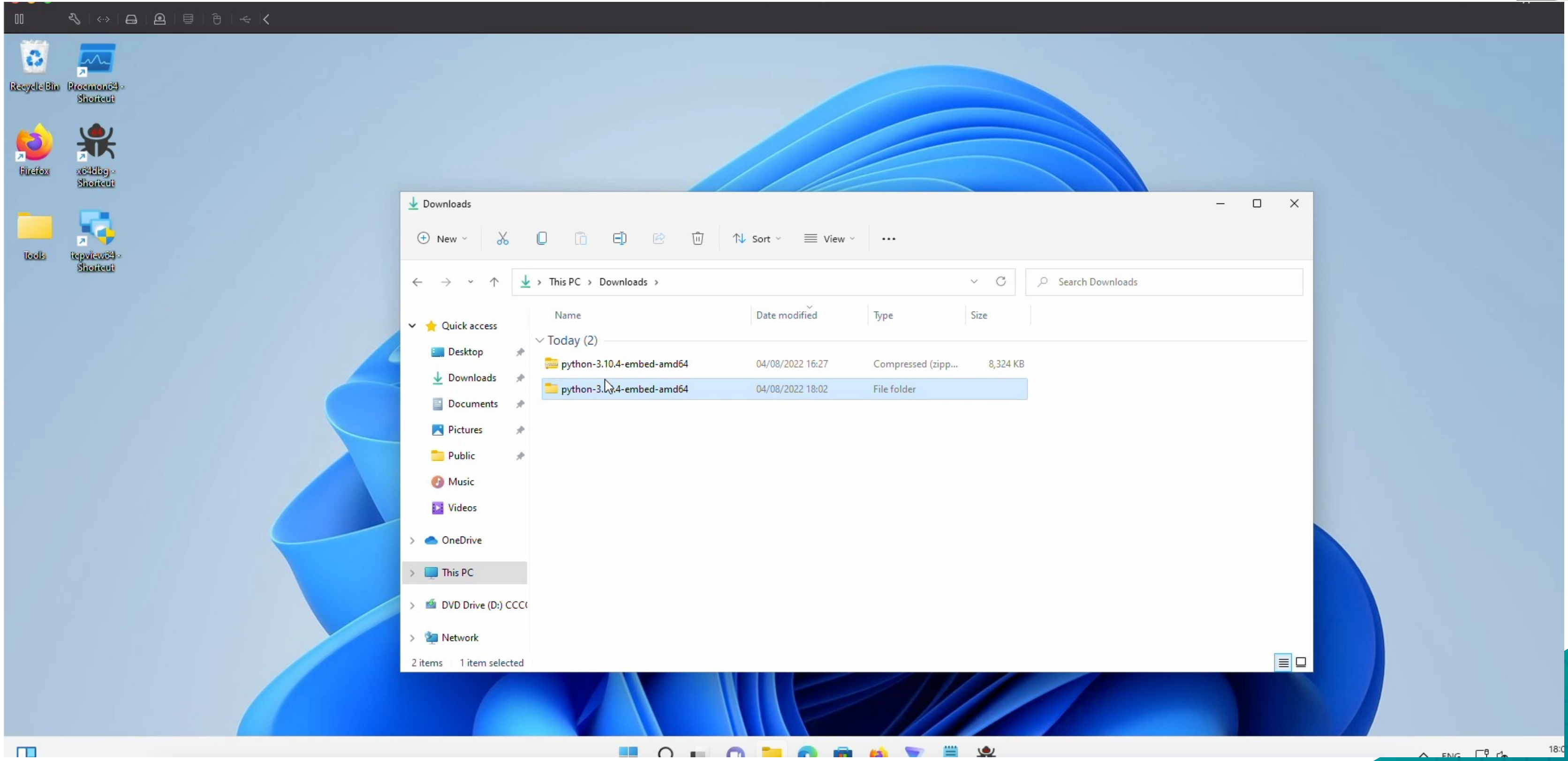
Demo

Dynamically importing and executing BloodHound-Python



Demo

In-process tunneling a Cobalt Strike Beacon with Python



Conclusions

- The main takeaways for the talk are:
  - You can use Python Language to dynamically execute Python tools without falling into EDR visibility.
  - Python Embeddable package provide an “attack avenue” with a signed context under which an attacker can operate.
  - Python interpreter has a huge “telemetry fingerprint” so it can be difficult for EDRs to spot anomalies coming from it.
  - You can execute BOFs, dynamic code and in-process tunneling from within python.exe increa[illegible] of not being detected.

THANKS!

Any questions?

You can find me on:

Twitter: **@naksyn**

Discord**: naksyn#9538**

References

https://synzack.github.io/Blinding-EDR-On-Windows/
https://www.ibs.it/malware-data-science-attack-detection-libro-inglese-joshua-saxe-hillary-sanders/e/9781593278595
https://github.com/forrest-orr/moneta
https://github.com/hasherezade/pe-sieve
https://www.upguard.com/blog/what-are-indicators-of-attack#toc-2
https://www.xorrior.com/In-Memory-Python-Imports/
https://github.com/EmpireProject/EmPyre
https://www.scythe.io/library/an-in-memory-embedding-of-cpython-with-scythe
https://peps.python.org/pep-0578/
https://peps.python.org/pep-0302/
https://utcc.utoronto.ca/~cks/space/blog/python/ZipimportAndNativeModules
https://www.sciencedirect.com/science/article/pii/S240595952100093X
https://github.com/helpsystems/nanodump
https://github.com/FalconForceTeam/BOF2shellcode
https://medium.com/falconforce/bof2shellcode-a-tutorial-converting-a-stand-alone-bof-loader-into-shellcode-6369aa518548